All About Memory

Ryan Baker

December 31, 2024

Contents

Lecture Objectives

By the end of this lecture, you should:

- Understand pointers, references, and indirection
- Be able to use arrays to store and manipulate many variables
- Be able to reason about memory segments, and how they effect execution

1 Pointers

Computers work with memory. Everything, from variables, to functions, to the actual machine code itself is stored in memory. Memory is extremely important. Pointers are tools to accessing and manipulating memory.

1.1 Basics of Pointers

- What is a pointer?
  - A pointer is a variable (an integer) that represents a memory address
- Why use pointers? Pointer provide:
  - Dynamic memory allocation and deallocation
  - Passing large objects efficiently to functions
  - Manipulation of memory at a low-level
- Pointer syntax: `<type>* <name>`
  - **Example:** `int* ptr;` declares a pointer to an `int`
  - **Example:** `int* ptr = nullptr;` declares a `NULL` pointer to an `int`
    - `nullptr` is a keyword that represents a null pointer value
    - `NULL` is a pointer value that points to address 0 (invalid)
- Using pointers:
  - Address-of operator: `&`
    - Use the address-of operator in front of a variable to get its address
    - `std::cout << &x << std::endl;` prints the address of `x`
    - `int* ptr = &x;` sets the value of `ptr` to the address of `x`
  - Dereference operator: `*`
    - Use the dereference operator in front of a pointer to get the value
    - `std::cout << *ptr << std::endl;` prints the value pointed to
    - `int x = *ptr;` sets the value of `x` to the the memory pointed to by `ptr`

  - Together, the address-of and dereference operators allow you to access and manipulate memory with pointers

```cpp
// pointer.cpp
#include <iostream>

int main()
{
    int x = 42;
    int* ptr = &x;

    std::cout << x << std::endl;
    std::cout << *ptr << std::endl;
    std::cout << &ptr << std::endl;
    std::cout << ptr << std::endl;

    return 0;
}
```

1.2 Pointer Arithmetic

- What is pointer arithmetic?
  - Pointer arithmetic involves performing operations (such as addition or subtraction) on pointer variables, adjusting the memory address they reference based on the size of the data type they point to.
- Pointer arithmetic in action:

```cpp
// parithmetic.cpp
#include <iostream>

int main()
{
    int x = 10;
    int* a = &x; // address-of x
    int* b = a + 1; // increments by sizeof(int)
    std::cout << a << std::endl;
    std::cout << b << std::endl;
    return 0;
}
```

- **Key point:** Pointer arithmetic on `void*`s causes errors because `sizeof(void)` is undefined

1.3 Pointers to Pointers

- Pointers are variables that represent memory addresses of other variables
- This means that a pointer can "point to" another pointer
  - `sizeof(<pointer>)` = 8 (usually, for 64-bit systems)

    - If you're working on a 32-bit system: `sizeof(<pointer>)` = 4

```
// ptop.cpp
#include <iostream>

int main()
{
    int x = -29;
    int* ax = &x; // address of x
    int** aax = &ax; // address of address of x
    int*** aaax = &aax; // address of address of ...

    std::cout << x << std::endl;
    std::cout << *ax << std::endl;
    std::cout << **aax << std::endl;
    std::cout << ***aaax << std::endl;

    return 0;
}
```

2 References

Every reference is just a pointer in disguise. They exist to help make working with pointers a bit easier.

- What is a reference?
    - A reference is a way to "reference" an existing variable
    - There is no such thing as a `NULL` reference
    - Think of references as aliases
- Reference syntax: `<type>& <name> = <variable>`
    - **Example:** `int& ref = x;` "refers" `ref` to `x`

```
// reference.cpp
#include <iostream>

int main()
{
    int x = 42;
    int& ref = x;

    ref *= 2;

    std::cout << x << std::endl;

    return 0;
}
```

- **Example:** `increment(int x)` {...} vs. `increment(int& x)` {...}
    - When you pass by reference, the original value is modified
    - When you pass by value (not reference), the original is untouched

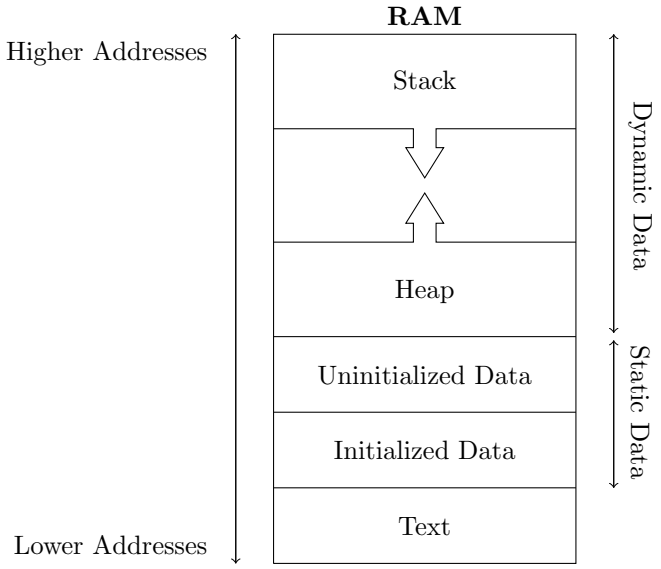
3 Arrays

- What is an array?
  - An array is a collection of variables stored contiguously
  - Truly, an array is simply a pointer to its first element
- When to use arrays?
  - We often need to represent an entire collection of data, rather than creating many many variables, we can create an array
  - **Example:** To represent the squares of a chess board, I could either create 64 variables: `int a1, int b1, ... int h8`, or use an array
- Array syntax: `<type> <name>[<size>] = {...}`
  - **Example:** `int chess_board[64];`
- Using arrays: `<name>[<index>]`
  - Indices start at 0
  - **Example:** `int square = chess_board[32];`
  - The square bracket (access) operator `[]` expands to a dereference:
    - `arr[5]` $\rightarrow$ `*(arr + 5)` (Recall pointer arithmetic)
    - This is why indices begin at 0
    - To prove this: `arr[5] == 5[arr]`

```
1  // access.cpp
2  #include <iostream>
3
4  int main()
5  {
6      int arr[5] = { 0, 1, 2, 3, 4 };
7
8      std::cout << arr[1] << std::endl;
9      std::cout << *(arr + 1) << std::endl;
10     std::cout << *(1 + arr) << std::endl;
11     std::cout << 1[arr] << std::endl;
12
13     return 0;
14 }
```

4 Memory Segments

The memory that your operating system assigns to your executable is split up into different segments. Each segment is designed to hold a different type of data.



4.1 Text Segment

The text segment, a.k.a. the "code" segment, contains the machine code instructions to run your program. It is fixed in size and read-only. It also contains integer and string literals present in your code.

4.2 Static Memory

Static memory is allocated at compile time and read directly from the executable. It contains global and static variables. This segment is divided into *initialized* and *uninitialized* data. The only difference is whether or not the variable has a compile-time initialized value.

4.3 Heap Memory

- What is the heap?
  - The heap is a memory segment that is variable in size and used for dynamic memory allocation.
  - It gives the programmer direct control over memory, allowing you to allocate and deallocate memory manually.
  - Think of it as a large pool of memory available for your program's use (though the term "heap" doesn't refer to this).
- How to use the heap?
  - Use the `new` operator to allocate memory:

        * `int* ptr = new int(42);` Allocates an `int` with value 42
    - Use the `delete` operator to free memory:
        * `delete ptr;` Frees the memory allocated to `ptr`

```cpp
// heap.cpp
#include <iostream>

int main()
{
    int* ptr = new int(42);
    std::cout << *ptr << std::endl;
    delete ptr;

    return 0;
}
```

- **Memory leaks:**
    - A memory leak occurs when memory is allocated on the heap but never freed, causing the program to consume more memory over time.
    - Forgetting to deallocate memory with `delete` leads to memory leaks
    - Dereferencing pointers that have been deleted is undefined behavior
    - In short, manual memory management is somewhat error prone, especially in large or collaborative projects.
    - Modern C++ provides alternatives to `new` and `delete` (covered in the future)

```cpp
// leak.cpp
#include <iostream>

int main()
{
    while (true)
    {
        // alloc'ed but never freed
        int* ptr = new int[1000000];
    }
    return 0;
}
```

4.4 Stack Memory

The stack is where local variables and function arguments live. The name "stack" refers to how the structure operates. It works like a stack of books, where the last one placed down is the first one popped off.

Every time a new function is called, a stack frame is pushed onto the stack. This stack frame contains the function's arguments and local variables. You may only ever directly access the top stack frame, hence scoped variables. When the function returns, its stack frame is popped off the top.