Comp 11 Lectures

Mike Shah

Tufts University

June 29, 2017

Pointers 2 and Memory Management

Comp 11 - Pre-Class warm up

Have some fun at this site.
https://cdecl.org/

```
// What are the types of a
    and b?
int* a, b;
```

Listing 1: Careful with initialization!

Comp 11 - Pre-Class warm up Answer

Have some fun at this site.
`https://cdecl.org/`

- int* a is a pointer to an integer
- b is an integer;

Lecture

Randy Pausch

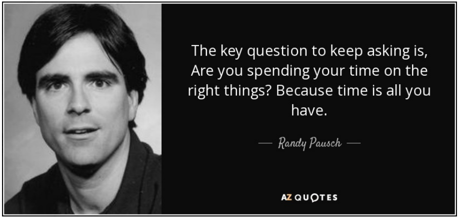


Figure 1: My all time favorite lecturer, Randy Pausch. Worth googling, but a great computer science communicator, and was an expert at Human-Computer Interaction

Pointers Are Magic

Magic!

Figure 2: Magic!

Pointers Are Fun

- We learned that pointers give us power and the ability to access and pass around data efficiently.
- We learned pointers are nothing more than an address.
- We learned a little bit about the stack and heap memory.

Figure 3: Not quite magic to us

Code

- Today, we are going to learn how to take control of memory!
- That is, up until now C++ has taken care of how memory is managed.
- Examples are that when something goes out of scope, it is effectively unaccessible.
- Another example is that data allocated on the stack(when calling a function) is automatically cleared for us, so that our programs can reuse that memory.

new and delete

Two C++ Keywords

- The **new** keyword allows us as programmers to allocate memory on the heap. We can do this wherever we want!
- We can allocate a block of memory that is freely available, and the memory allocator will give us new memory to store data in.
- When we are done with the memory, we **delete** it, so that we can use it ourselves.
- **delete** is our second C++ keyword, that reclaims memory so we can reallocate that memory later.
- Now, we are simply managing memory ourselves.

Why Manage Memory?

- Maybe it sounds like extra work, but this again is an advantage of C++ that we are in control of our programs.
- Secondly, this allows us to build our own data structures.
- We have learned about strings and vectors for example, which behind the scenes, a C++ programmer has written.
  - For example, what do you think happens when we add another element to a vector?
  - Or what happens when we concatenate strings together?
  - The answer is, more memory needs to be allocated
- Finally, it will allow us to build a new data structure, the linked list!
- Let's take a quick look at the two scenarios we use new and delete.

Cheat Sheet 1

```
struct Student{
    std::string name;
  int age;
};

// (1) Give me memory for 1 new student
// Again, this is a pointer, because we are thinking
// about memory.
// This will allocate memory for one new student
// (which is essentially a string + an int)
Student* mike = new Student;

// (2) Delete the Student.
// Simply delete our object 'mike' when we are done.
delete mike;
```

Listing 2: Allocating a single item on the heap

Cheat Sheet 2

```
// (1) Give me memory for 50 integers
// And remember, we use a pointer because we are
// thinking about memory addresses, and where this is
    allocated.
int* intArray = new int[50];

// (2) Delete the array.
// Note the brackets, indicating that intArray is more than
// one item. Because, well, it is an array of 50 items!
delete[] intArray;
```

Listing 3: Allocating a new array on the heap

Observations

- The first observation is that we use new and delete in pairs.
- It is a best practice to reclaim our memory(delete) as soon as we are done with it.
- When we allocate memory, again, it is allocated on the heap. As long as we know the address, we can always access it! (Until we delete that memory of course!)
- We can allocate(with new) memory anywhere (in main, in functions, in our struct, etc.).
- Pay attention to using brackets with delete! Use brackets when deleting an array, otherwise you are only deleting a single element (which would be only the first element of the array–and then we get a memory leak!).

Example 1 - Allocating one new object

```
#include <iostream>
#include <string>

struct Student{
  int age;
  std::string name;
};

int main(){

  Student* mike = new Student;
  delete mike;

}
```

Listing 4: Allocating a single item on the heap

Example 2 - Allocating a new array

```
#include <iostream>
#include <string>

struct Student{
  int age;
  std::string name;
};

int main(){

  Student* studentDataBase = new Student[5000];
  delete[] studentDataBase;

}
```

Listing 5: Allocating a new array on the heap

(BAD) Example 3 - Creating a memory leak

```
#include <iostream>

int main(){

  int* intArray = new int[50];
    // Yikes--we forgot to do delete[]!
    // C++ may not even complain, but our memory is lost
    forever
  delete intArray;

};
```

Listing 6: Example of what not to do

Why is our memory lost in the previous example?

- When you use **new**, your program is asking your operating system(OS) to give you some memory.
- When the OS grants your program that memory, that memory is exclusively for your program.
- That is, other programs cannot access that memory (If they try to, you get a segfault!)
- In general, the worse case scenario is you run out of memory, then your OS cleans up for you, or you restart your computer. No need for much panic.
- (If you have ever received a blue screen of death, it may have been because of a memory leak that occurred over some period of time)

Pointers 2 - Linked List Data Structure

Pointer to a pointer?

Question, if a pointer is just a variable (that holds an address), can I have a pointer point to a pointer? [1]

[1] Try saying that five times fast

A primitive linked list

- Well, of course we can!
- We are going to build a data structure called the linked list
- A linked list is a **struct** that has a pointer to another variable of its type.
- This forms a sort of chain

Singly Linked List

```
// We call our datatype a Node.
// Such that, a bunch of nodes connected makes a list
struct Node{
  // The first field is a pointer to another node.
    // That is, we can connect, or chain together a series of
     nodes
  Node* next;
    // This is the regular data that we may have.
    // So now we can chain together some series of data,
    // whether that be characters, students, files, etc.
    int data;
};
```

Listing 7: A simple linked list

Our first linked list

```
#include <iostream>
// Create a struct
struct Node{
    Node* next;
    int data;
};
int main(){
  // Allocate a new node.
  Node* firstNode = new Node;
  Node* secondNode = new Node;
    // de-reference first! I put in parenthesis to make this explicit
    // Then we access our member variable with '.' as usual, and
    // point to another address, in this case our secondNode.
    (*firstNode).next = secondNode;
    (*secondNode).next = NULL;
  // Delete to avoid memory leaks
    delete firstNode;
    delete secondNode;

  return 0;
}
```

Listing 8: A simple linked list of two items

A couple of notes - Part 1

- (*firstNode).next = secondNode;
- The above syntax is a little bit ugly, so we can use a more convenient syntax that does the same exact thing.

−> arrow de-references and access at the same time!

```
#include <iostream>
// Create a struct
struct Node{
  Node* next;
  int data;
};

int main(){
  Node* firstNode = new Node;
  Node* secondNode = new Node;
    // -> arrow de-references and access at the same time!
  firstNode->next = secondNode;
  secondNode->next = NULL;

  delete firstNode;
  delete secondNode;

  return 0;
}
```

Listing 9: New arrow syntax is simpler!

Visualization of what we have done

A couple of notes - Part 2

- If we are not pointing to anything, we point to NULL for safety.
- Why are we making everything a pointer?
- Well, it is true we could do the same example without pointers.
- We know how to get 'address of' a variable or object using & (ampersand). So we can still point to an address.
- But typically, we want our data structure to persist beyond the stack, and we want control over our memory.

Linked list without heap allocation, only on stack memory

```cpp
#include <iostream>
// Create a struct
struct Node{
  Node* next;
    int data;
};

int main(){
  // Allocate a node, but this is
  // only on the stack.
  // It will get removed later
  Node firstNode;
  Node secondNode;
  // Just point to whatever address is
  // on the stack.
  firstNode.next = &secondNode;
  secondNode.next = NULL;

  return 0;
}
```

Linked List Power

- We like linked lists because we can insert data as we need it.
- Simply allocate a new Node, and then append it to the end of the list.
- With the array, we are stuck to a fixed size. And we often want the ability to expand our data structures.

Initializing the data

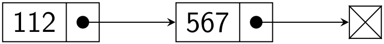
```
#include <iostream>
struct Node{
  Node* next;
  int data;
};
int main(){
  Node* firstNode = new Node;
  Node* secondNode = new Node;
  // Initializing each member variable field
  firstNode->data = 112;
  secondNode->data = 567;

  firstNode->next = secondNode;
  secondNode->next = NULL;

  delete firstNode;
  delete secondNode;
  return 0;
}
```

Listing 11: Do not forget to initialize the data

The real visualization



A common linked list traversal pattern

- A common pattern for traversing a linked list is to create an additional node called 'iter'.
- This node points to the 'head' (i.e. the first element of your list).
- The iterator then moves its next pointer
- You will do this in the lab today. Try it out, and draw a picture!

In-Class Activity

http:
//www.mshah.io/comp/11/activities/activity9/activity.pdf

Activity Discussion

Review of what we learned

- (At least) Two students
- Tell me each 1 thing you learned or found interesting in lecture.

5-10 minute break

To the lab!

Lab: `http://www.mshah.io/comp/11/labs/lab8/lab.pdf` [2]

[2] You should have gotten an e-mail and hopefully setup an account at `https://www.eecs.tufts.edu/~accounts` prior to today. If not–no worries, we'll take care of it during lab!