Programming Abstractions

CS106X

Cynthia Lee

Stanford University

Topics this week:

- **Memory and Pointers**
  - Revisit some topics from last week
  - Deeper look at what a pointer is
    - Hexadecimal!
    - Address-of operator: &
  - Dynamic Memory allocation
  - Dereference operator: *
  - Dynamic Memory with classes
    - The -> operator
  - Linked nodes
  - Linked List data structure
  - (if we have time) Binary tree data structure

- **TODAY IS THE LAST DAY OF TOPICS FOR THE MIDTERM**

Heap memory works like a hotel registration desk

(Congratulations Golden Globe winner Grand Budapest Hotel)

Only a creepy killer would access a hotel room that isn’t theirs (either never was, or was but checked out already)

(Another great film about unusual people who work at hotels)

`new` and `delete` Hotel Analogy

`delete` is like checking out of a hotel

- You say that you’re checking out and give back your key
- Now the hotel can give that room to somebody else

- Do NOT re-enter the room (awkward and/or lawsuit)
  - Sometimes happens if you have two guests in the same room (two pointers pointing to the same thing) and one checks out (deletes) but the other keeps it
- Do NOT check out twice (redundant)
  - Sometimes happens if you have two guests in the same room and they both check out
- Do NOT lose your key! (stuck forever)
  - You can’t check out unless you have it (you can’t call delete unless you have the pointer--memory leak)

Dynamic memory allocation

```
int * p1 = new int;  //0x12
*p1 = 5;
int * p2 = new int;  //0x4
*p2 = 7;
int * p3 = new int;  //0x20
*p3 = 8;
*p1 = *p2;
cout << p1 << " " << *p1 << endl;
p1 = p2;
cout << p1 << " " << *p1 << endl;
delete p1;
//what could go here?
```

How many of these lines of code could legally be the next line?

~~delete p2;~~

delete p3; ✓

~~*p1 = 10;~~

p1 = p3; ✓

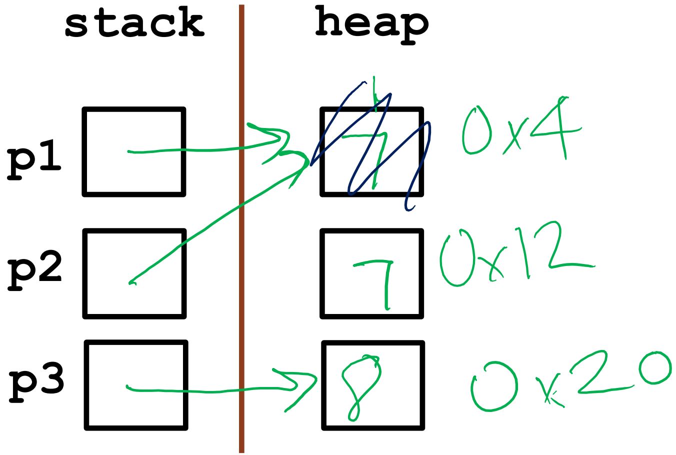
~~cout << p1 << endl;~~

~~cout << *p2 << endl;~~

A. 0-2 of them
B. 3 of them
C. 4 of them
D. 5+ of them

Dynamic memory allocation

```
int * p1 = new int;   //0x12
*p1 = 5;
int * p2 = new int;   //0x4
*p2 = 7;
int * p3 = new int;   //0x20
*p3 = 8;
*p1 = *p2;
cout << p1 << " " << *p1 << endl;
p1 = p2;
cout << p1 << " " << *p1 << endl;
delete p1;
//what can go here?
```

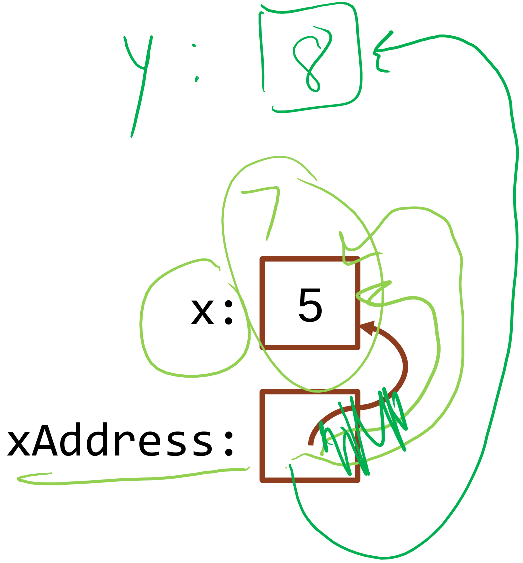


Pointers

Dereference operator

Stanford University

Dereference operator



You’ve learned the address-of operator &:

```
int   x = 5;
int   *xAddress = &x;
cout << xAddress << endl;      // 0x28FE50
```

- It tells you the address of any variable

int y = 8;

It has a partner, the dereference operator *:

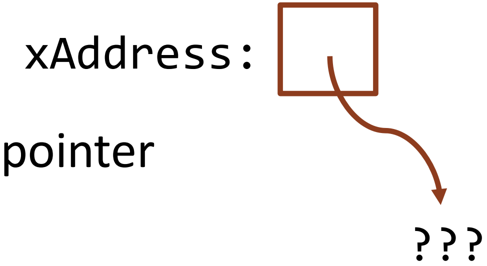
```
int   x = 5;
int   *xAddress = &x;      // this * is not dereference op!
cout << *xAddress << endl; // 5
*xAddress = 7;
cout << x << endl;         // 7
```

xAddress = &y;

- It tells you what the pointer points to
  - Follows the “arrow” to the end, for reading *or* writing the value

Dereferencing an uninitialized pointer

```
int     *randAddress;         // uninitialized!
cout <<  randAddress << endl; // [prints nonsense]
cout << *randAddress << endl; // ???
```



- There is no problem printing an uninitialized pointer
  - Will print nonsense, but do so safely (no danger of crash)
- There IS a problem with dereferencing an uninitialized pointer
  - Example above may print nonsense
  - *Or* may attempt to print a restricted area—CRASH!
  - *The fact that either is possible is a huge problem for debugging*

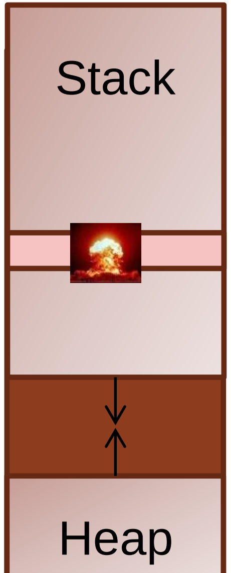
Software engineering tip:

Initialize pointers always.

- If not to a useful value, then at least to NULL.
- NULL *forces* your program to crash (this is a good thing, really!) if you try to dereference

Common error:

```
int* foo;
…
*foo = 555;
```



Prevention:

```
int* foo = NULL;
…
*foo = 555;
```

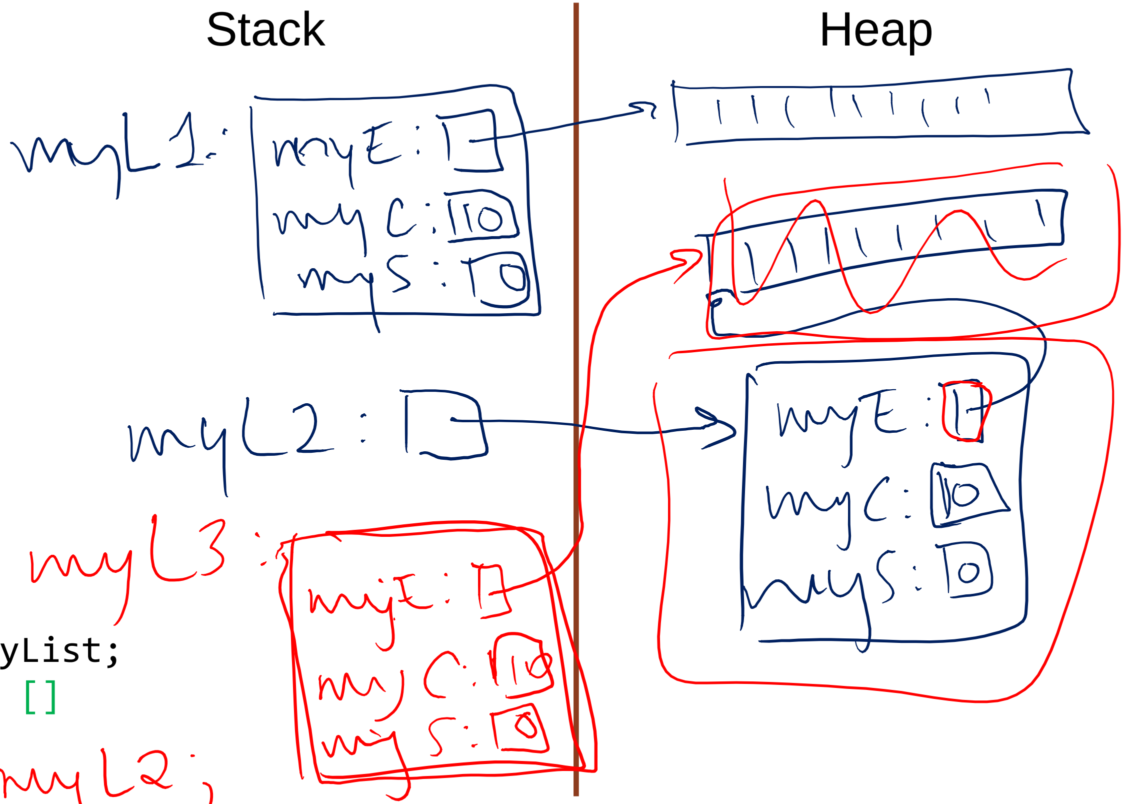
Stack and Heap Memory *with classes*

Quickly returning to memory diagrams and our ArrayList implementation
Introducing the -> operator

Stack and Heap Memory *with classes*

```
// in ArrayList.h
class ArrayList {
public:
   …
private:
   int* myElements;
   int myCapacity;
   int mySize;
};

// in arraylistclient.cpp
int main() {
   ArrayList  myL1;
   ArrayList *myL2 = new ArrayList;
   delete myL2;     // note no []
   return 0;
}
```


Stack

Heap

myL3 = * myL2;

Stack and Heap Memory with Stanford Library ADTs

- Our Stanford Library data structures work in a similar way
  - You can choose to put instances on the Stack or Heap
  - *Either way*, they internally put large data structures on the Heap
  - Constructor allocates heap memory with “new”
  - Destructor frees heap memory with “delete”

Useful shortcut for pointers to objects: -> operator

```
// in arraylistclient.cpp
int main() {
   ArrayList  myL1;
   myL1.add(3);
   ArrayList *myL2 = new ArrayList;
   (*myL2).add(3);                   // clunky syntax
   myL2->add(3);                     // equivalent version with ->
   delete myL2;
   return 0;
}
```

- The -> operator is just shorthand for combining dereference and member variable/function access (works for classes and structs)

```
Move *myMarbleMove = new Move(1, 3, 1, 5);
myMarbleMove->startRow = 2;
myMarbleMove->startCol = 4;
```

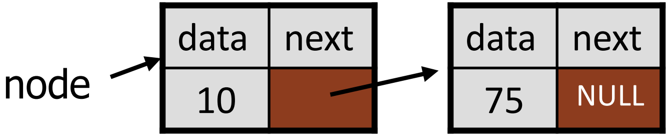
Linked Nodes

A great way to exercise your pointer understanding

Linked Node

```
struct LinkNode {
   int data;
   LinkNode *next;
}
```

- We can chain these together in memory:



```
LinkNode *node1 = new LinkNode;      // complete the code to make picture
node1->data = 10;
node1->next = NULL;
LinkNode *node = new LinkNode;
node->data = 10;
node->next = node1;
```

FIRST RULE OF LINKED NODE/LISTS CLUB:

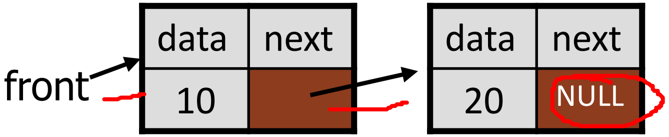
DRAW A PICTURE OF LINKED LISTS

Do no attempt to code linked nodes/lists without pictures!

List code example: Draw a picture!

```
struct LinkNode {
   int data;
   LinkNode *next;
}
```

Before:



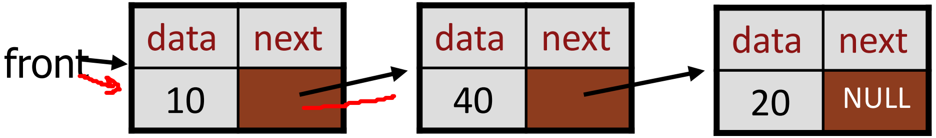
front

data	next
10	

data	next
20	NULL

```
front->next->next = new LinkNode;
front->next->next->data = 40;
```

A. After:



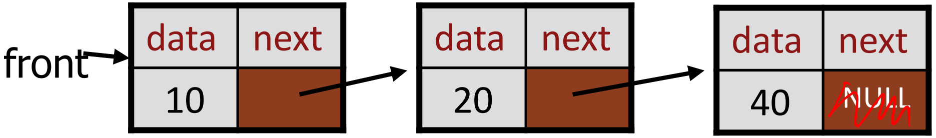
front

data	next
10	

data	next
40	

data	next
20	NULL

[✓ B.] After:



front

data	next
10	

data	next
20	

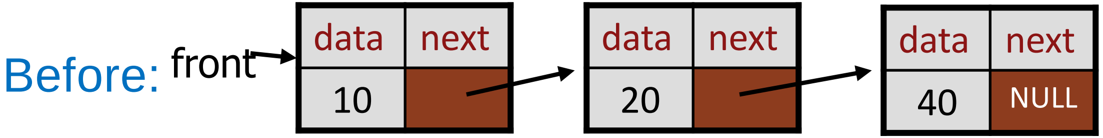
data	next
40	~~NULL~~

uninitialized

C. Using “next” that is NULL gives error

D. Other/none/more than one

List code example: Draw a picture!



```
struct LinkNode {
   int data;
   LinkNode *next;
}
```

Write code that will put these in the reverse order.