Programming Abstractions

CS106X

Cynthia Lee

Topics:

- Last week, with Marty Stepp:
  - Making your own class
  - Arrays in C++
- **This week: Memory and Pointers**
  - Revisit some topics from last week
  - Deeper look at what a pointer is
    - Hexadecimal!
  - Dynamic Memory allocation
  - Linked nodes
  - Linked List data structure
  - (if we have time) Binary tree data structure

Arrays

(revisit from last week)

Arrays (11.3)

```
type* name = new type[length];
```

- A **dynamically allocated** array.
- The variable that refers to the array is a **pointer**.
- The memory allocated for the array must be manually released, or else the program will have a **memory leak**. (>_<)

- Another array creation syntax that we will not use*:

```
type name[length];
```

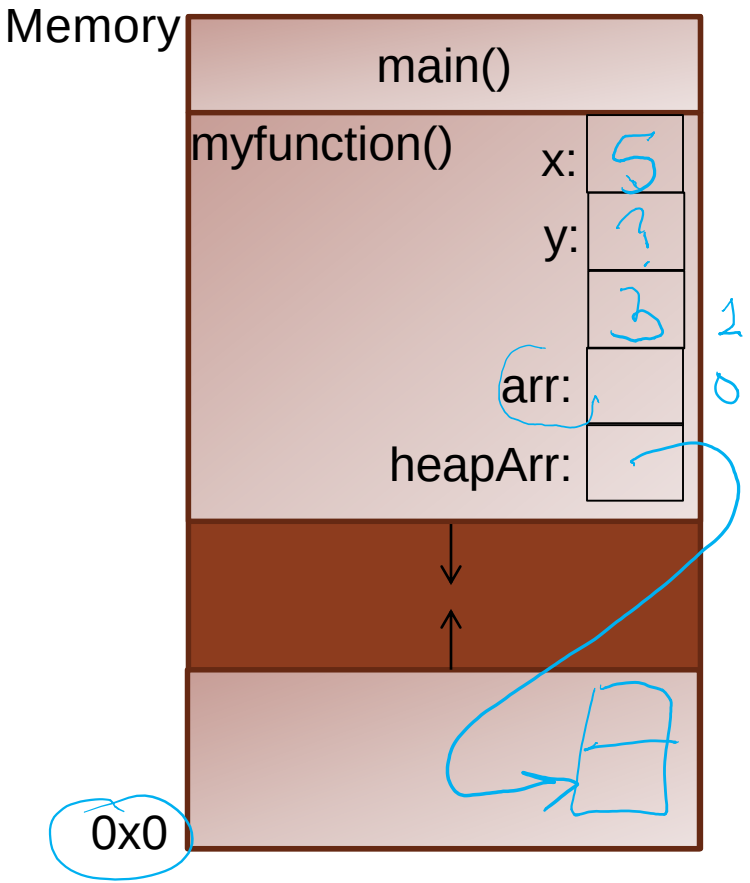
  - A fixed array; initialized at declaration; can never be resized.
  - Stored in a different place in memory; the first syntax uses the *stack* and the second uses the *heap*. *(discussed later)*

* For 106X, I would like you to know this syntax and be able to diagram what it does, but Marty Stepp was right in that we won't generally be *using* it in code you write (for style/design reasons).

Arrays on the stack and heap

```
void myfunction() {
    int x = 5;
    int y;
    int arr[2];
    arr[1] = 3;
    int *heapArr = new int[2];
    // bad -- memory leak coming!
}
```

What happens when `myfunction()` returns?



ArrayList

(revisit from last week)

Freeing array memory

```
delete[] name;
```

- Releases the memory associated with the given array
- Must be done for all arrays created with `new`
  - Or else the program has a *memory leak*. (No garbage collector like Java)
  - Leaked memory will be released when the program exits, but for long-running programs, memory leaks are bad and will eventually exhaust your RAM

```
int* a = new int[3];
a[0] = 42;
a[1] = -5;
a[2] = 17;
  ...
delete[] a;
```

Destructor (12.3)

```
// ClassName.h
~ClassName();
```

```
// ClassName.cpp
ClassName::~ClassName() { ...
```

destructor: Called when the object is deleted by the program.

(when the object goes out of {} scope; opposite of a constructor)

- Useful if your object needs to do anything important as it dies:
  - saving any temporary resources inside the object
  - freeing any dynamically allocated memory used by the object's members
  - …
- Does our `ArrayList` need a destructor? If so, what should it do?

Destructor solution

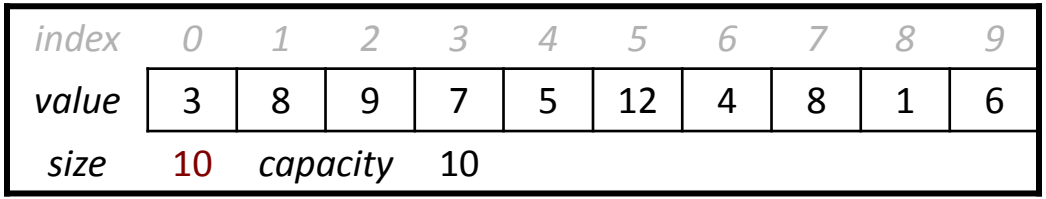
```
// in ArrayList.cpp

ArrayList::ArrayList() {
    myElements = new int[10]();
    mySize = 0;
    myCapacity = 10;
}


void ArrayList::~ArrayList() {
   delete[] myElements;
}
```

Running out of space

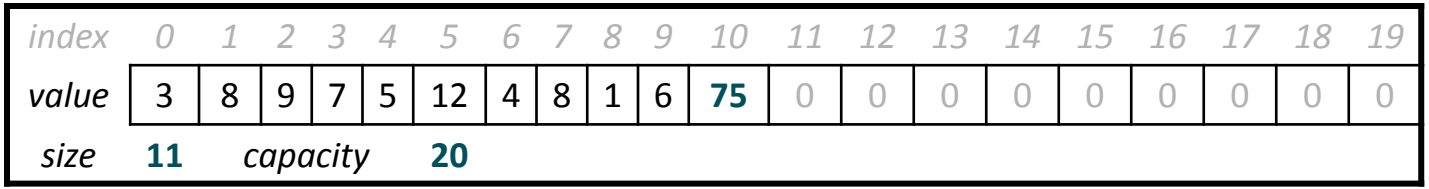
What if the client wants to add more than 10 elements?



index	0	1	2	3	4	5	6	7	8	9
value	3	8	9	7	5	12	4	8	1	6

size 10 capacity 10

- `list.add(75);  // add an 11th element`



index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
value	3	8	9	7	5	12	4	8	1	6	**75**	0	0	0	0	0	0	0	0	0

size **11** capacity **20**

- Answer: **Resize the array** to one twice as large.
  - Make sure to *free the memory* used by the old array!

Resize solution

```
// in ArrayList.cpp
void ArrayList::checkResize() {
    if (mySize == myCapacity) {
        // create bigger array and copy data over
        int* bigger = new int[2 * capacity]();
        for (int i = 0; i < myCapacity; i++) {
            bigger[i] = myElements[i];
        }
        delete[] myElements;
        myElements = bigger;
        myCapacity *= 2;
    }
}
```

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)

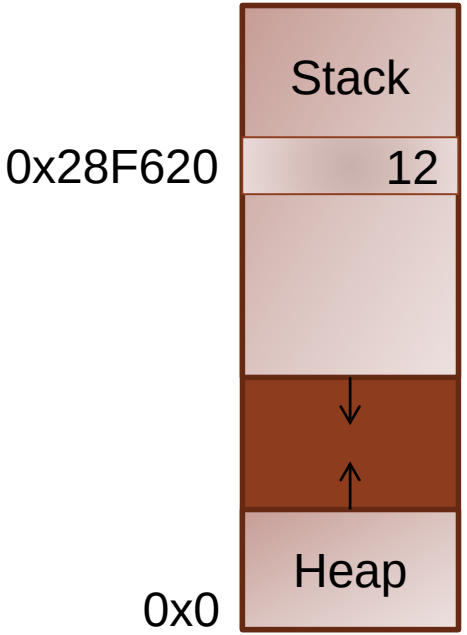
(Another great film about unusual people who work at hotels)

Pointers

Taking a deeper look at the syntax of that array on the heap

Memory addresses: the basics

- When you declare a variable, it is necessarily stored somewhere in memory
- You can ask for *any* variable's memory address with the **& operator**
- Memory addresses are usually written as hexadecimal (base-16 or “hex”) numbers
- Ex: 0x28F620
  - Prefix “0x” is a visual cue that this is a hex number (not really part of the number)



Examples of the & operator

```
int    x = 42;
int    y;
int    a[4] = {91, -3, 85, 17};
double d1 = 3.0;
double d2 = 3.2;

cout << x << endl;
cout << &x << endl;
cout << y << endl;
cout << &y << endl;
cout << a << endl;
cout << &a[0] << endl;
cout << &a[1] << endl;
cout << &a[2] << endl;
cout << &d << endl;
```

Pointer arithmetic

Surprise! operators `+`, `-`, `++`, `--` work on pointers

- Incrementing a ***T****** (pointer to type T) by 1 moves the pointer forward in memory to the next T

Thinking about what we just learned…

When we add 1 to a pointer, how much does the address change?

```
int     x = 42;
int     y;
int     a[4] = {91, -3, 85, 17};
double  d1 = 3.0;
...
int*    pi = &x;
double* pd = &d1;
cout << pi << endl;
pi++;
cout << pi << endl;
cout << pd << endl;
pd++;
cout << pd << endl;
```

A. You can't do ++ on a *pointer to* int

B. pi++ adds 1 and pd++ adds 1, of course!

C. pi++ adds 4 and pd++ adds 8

D. Other/none/more than one

This is why the language has you specify double* and int* rather than just having a generic* type (although it does have one of those too—void*).

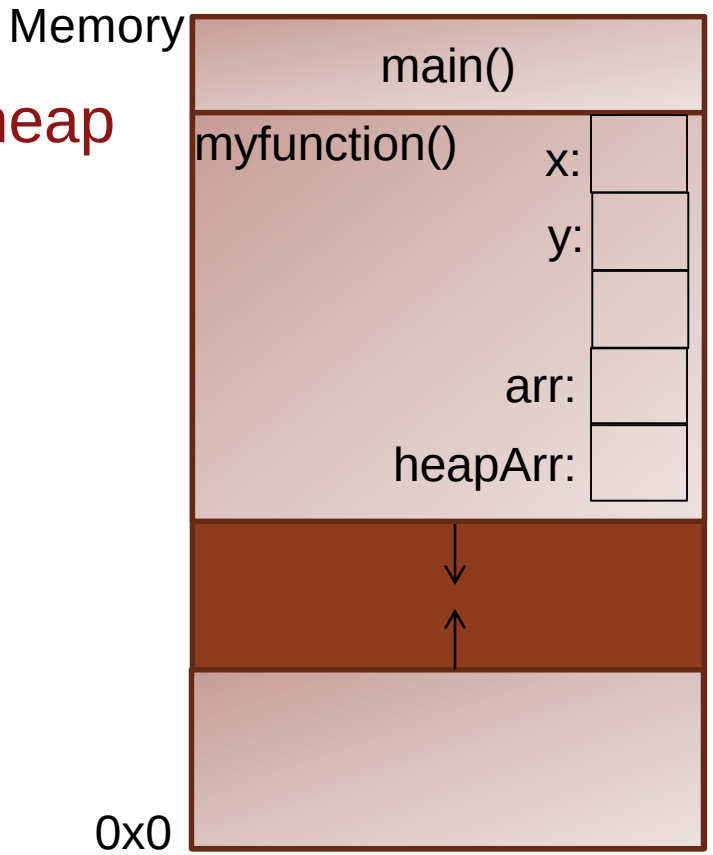
Data types in C++

- Each unique memory address describes **one byte** of memory space (1byte = 8bits, or 8 1's and 0's)
  - We say memory is "byte addressable"
- `int` addresses are 4 apart (32bits)
- `double` addresses are 8 apart (64bits)
- The above is true on *this* system--may differ on other systems
  - Of course C++ doesn't impose a standard, because that would be safe and convenient, and that's not how we roll....

- Bonus question: do int* and double* take up the (A) SAME or (B) DIFFERENT amounts of space?

2nd look: Arrays on the stack and heap

```
void myfunction() {
    int x = 5;
    int y;
    int arr[2];
    arr[1] = 3;
    int *heapArr = new int[2];
    // bad -- memory leak coming!
}
```



Arrays are actually pointers

Surprise!

- If you have a pointer p (see declaration of p below), you can use p[**k**] syntax to access memory **k** slots away from p (according to the size of the type pointed to)
  - **p[k]** is equivalent to ***(p+k)**
- (An array variable is really just a *pointer* to the array's first element, and so pointers and array variables work the same for indexing.)

```
...
int  a[4] = {91, -3, 85, 17};
...
int* p = a;
p[1] = 5;
p++;
cout << *p << endl;
*(p + 2) = 26;
cout << p[2] << endl;
cout << a[2] << endl;
```

What prints here?

A. 26, 26
B. 26, 85
C. 85, 26
D. 85, 85

In the news: Heartbleed

- Last spring, security experts warned that users of thousands of major websites needed to change their passwords due to potential exposure caused by the “Heartbleed” vulnerability

- Heartbleed exploited a **buffer overrun** bug in OpenSSL
  - SSL is the layer that secures web interactions, i.e., it’s what make the “s” in “https://” mean something

In the news: Heartbleed

- The protocol allows you to send “heartbeat” messages, which basically say:
  - *Are you still there? If you are, repeat this word back to me: “hello” [0x0005 bytes].*
  - Each char is one byte, so 5 letters

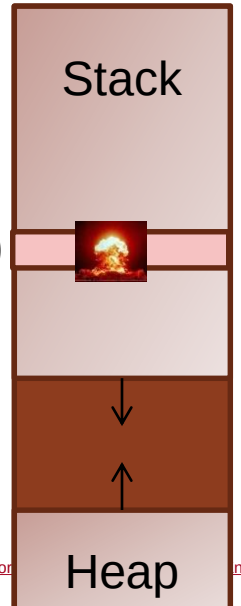
- Unfortunately, the software also let you send messages like this:
  - *Are you still there? If you are, repeat this word back to me: “hello” [0xFFFF bytes].*
  - That’s 65535 bytes—much more than the length of “`hello`”!
  - So the software would continue for-looping past the end of the “`hello`” array, sending information back
  - Which causes an error, right? **RIGHT??** Turns out, no.

Software engineering tip:

Initialize pointers always! If not to a useful value, then at least to NULL. NULL initialization *forces* your program to crash (this is a good thing, really!) if you try to use the pointer without assigning it a proper value

Common error:

```
int* foo;
…never initialized…
*foo = 555;
```



Prevention:

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