Lab 7 - Dynamic Memory

ReadMe

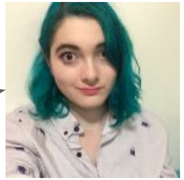
Reminders

- Lab 7 due Sunday, November 11th at 8pm
- Project 4 due Monday, November 19th at 8pm

Agenda

- Review - dynamic memory
- Worksheet
- Discuss Project 4
- Lab 7

Slide Credit: Carolyn



What is Dynamic Memory

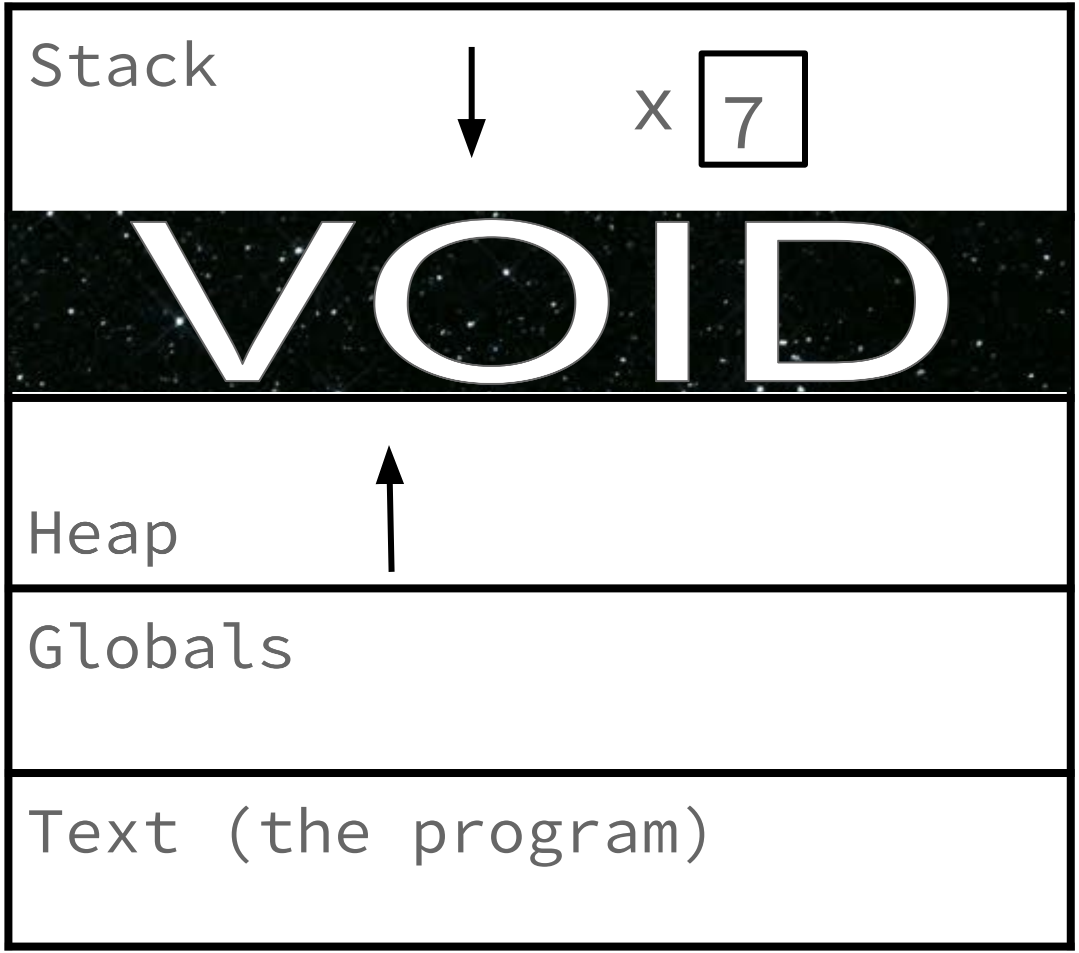
Dynamic Memory allows you to write programs that can handle memory usage efficiently, and lets us use our computers more effectively.

It lets us do lots of cool things:

- Dynamically sized arrays!
- Control objects' lifetimes!
- Decide what types of objects we want to use at runtime! (Recall: Euchre project)
    - We don't need to know at compile time whether we want to use a Simple Player or a Human Player -- we can decide "on the fly"

Local variables
are stored here

```
int main{
    int x = 7;
}
```



Stack

x 7

VOID

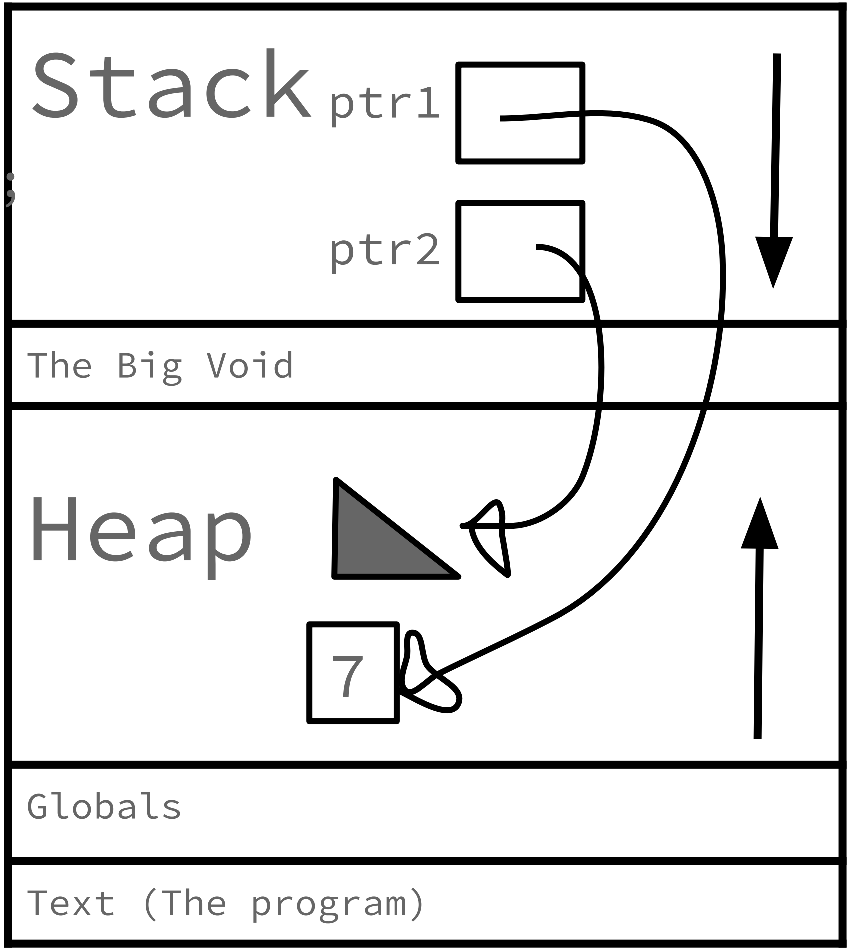
Heap

Globals

Text (the program)

```
int *ptr1 = new int(7);
Triangle *ptr2 = new Triangle(3,4,5);
```

If we use the new keyword
we store the object that we
construct in the heap.
We get back the address of
The object on the heap



Stack

ptr1

ptr2

The Big Void

Heap

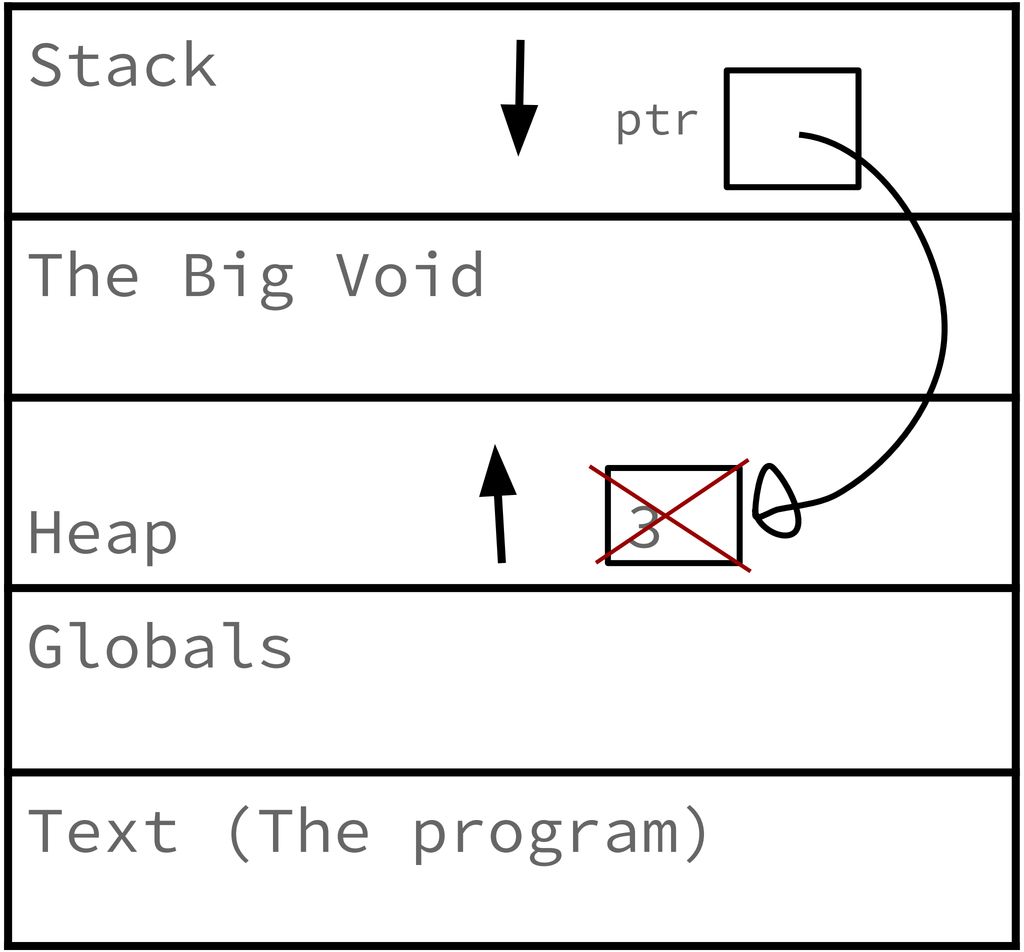
7

Globals

Text (The program)

```
int *ptr = new int(3);
delete ptr;
```

‘new’ allocates memory on the heap.
‘delete’ deallocates memory on the heap.
It frees the memory so we can now store new objects at that address



Stack

ptr

The Big Void

Heap

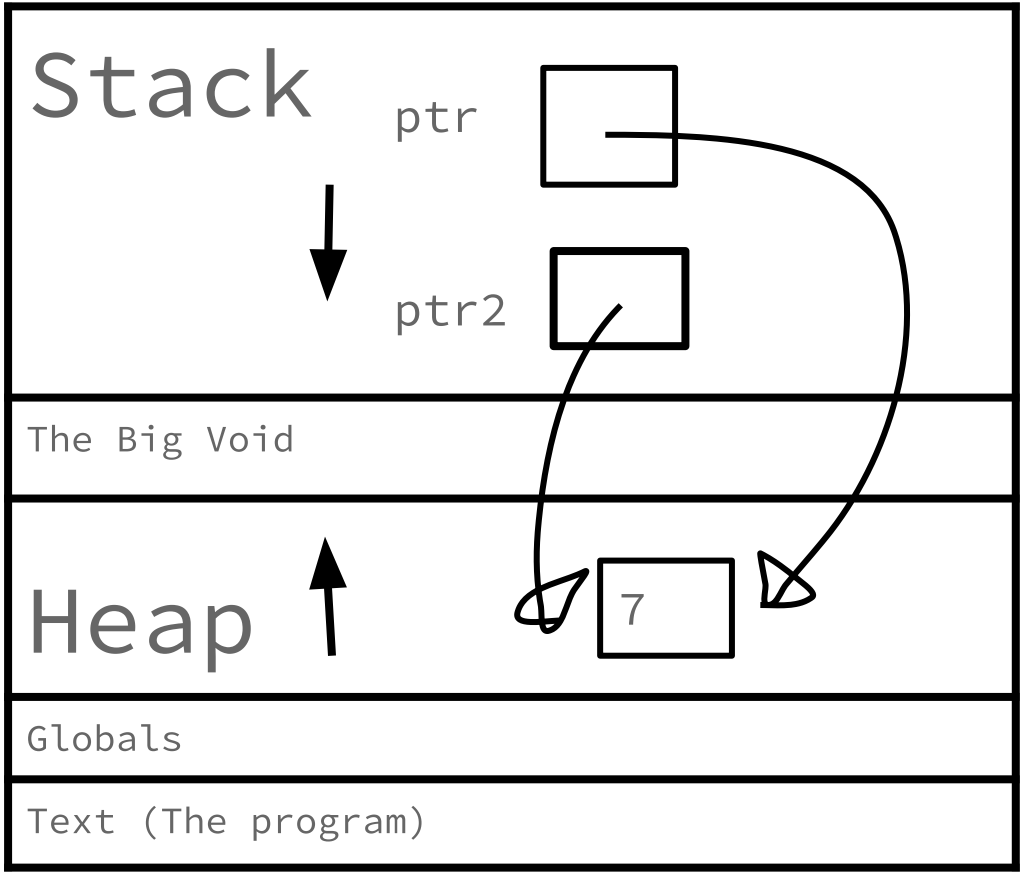
3

Globals

Text (The program)

```
int *ptr = new int(3);
delete ptr;
int *ptr2 = new int(7);
```

‘new’ allocates memory on the heap.
‘delete’ deallocates memory on the heap.
It frees the memory so we can now store new objects at that address



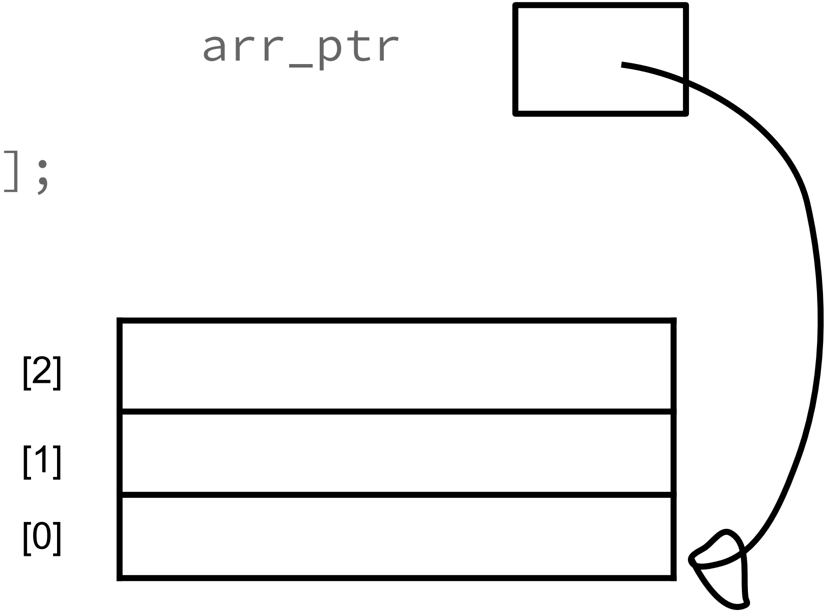
Dynamically allocated arrays

```
int size = 3;

int *arr_ptr = new int [size];

delete[] arr_ptr;
```

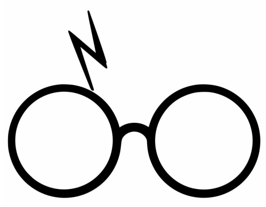
Remember to use this special syntax to delete the whole array!



Dynamic memory problems

Memory leak

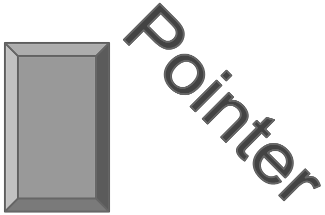
Orphaned memory



Double free

Bad delete

Dangling pointer



Memory leak

```
int *ptr = new
int(7);

int x = 3;

ptr = &x;
```

```
void foo() {
    int *ptr = new int(7);
    return;
}
```

We lose the address of the integer 7. When we lose the address of an object on the heap, the memory is **orphaned**.

When we don’t free up memory we are no longer using, we have a **memory leak**.

Orphaned memory -> memory leak

Double free

```
int *ptr = new int(3);
delete ptr;
delete ptr;
```

```
int *ptr1 = new int(3);

int *ptr2 = ptr1;

delete ptr1;

delete ptr2;
```

A **double free** occurs when we attempt to free up memory that’s already been freed

Bad delete

```
int x = 7;

int *ptr = &x;

delete ptr;
```

A **bad delete** is when we attempt to deallocate memory that’s not stored on the heap.

Dangling pointer

```
int *ptr = new int(3);

delete ptr;

int *ptr2 = new int(7);

cout << *ptr << endl;
```

A **dangling pointer** stores a freed memory address.

Solution: set equal to null pointer after deleting to make sure we don’t use it

```
ptr = nullptr;
```

The Destructor

Recall that the destructor runs when the class instance goes out of scope.

If a variable goes out of scope - that means it is no longer needed and the compiler will get rid of it.

However if the class managed dynamic memory, the compiler doesn’t deal with freeing it. Instead, the programer must write any necessary code in the destructor, and compiler will make sure to run that at the appropriate time.

Rule of thumb: Often, if **new** is used in the constructor, **delete** is used in destructor.

When Should I Use Dynamic Memory?

You will rarely use dynamic memory to allocate individual variables.

Dynamic memory is often used to allocate blocks of memory at a time. It’s most often used in dynamic data structures like vectors and linked lists.

The Amazing Expanding Vector

How are vectors implemented under the hood?

Vectors have a fixed size array as part of the class. This array exists on the heap, while a pointer to the first element, capacity and size are member variables of the class.

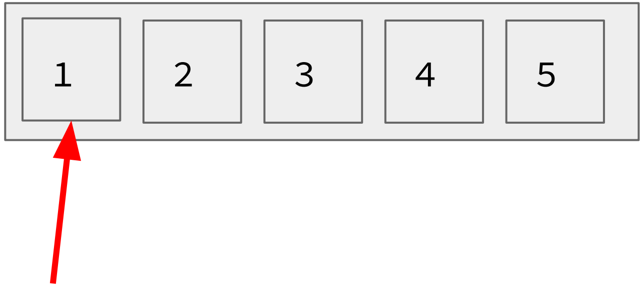
When the fixed size array becomes full, it “grows” by moving the data into a new space. This involves allocating enough space, copying over existing elements, and freeing up the new space.

Vectors typically double their array when they grow - this ends up in longer term efficiency.

Vectors: How they expand

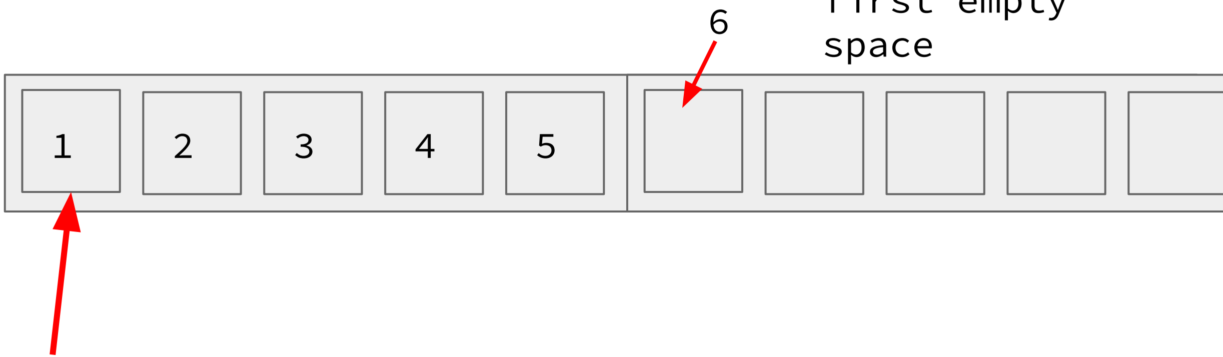
call push_back(6) on the vector…

No more room! Expand!



Pointer to first elt of array

add new elt to first empty space



New array with twice the capacity as the old one.

Next:

- Worksheet
- Project 4 Questions?
- Lab 7