Programming Abstractions

CS106B

Cynthia Lee

Topics:

- Last week:
  - Making your own class
  - Arrays in C++
  - new/delete
- **This week: Memory and Pointers**
  - First revisit some topics from last week in more detail:
    - Deeper look at new/delete dynamic memory allocation
    - Deeper look at what a pointer is
  - Then new topics:
    - Linked nodes
    - Linked List data structure
    - (if we have time) Binary tree data structure

Hat tip to Victoria Kirst of Google for some of today's slides!

`new` and `delete`

(revisit from last week)

Arrays

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

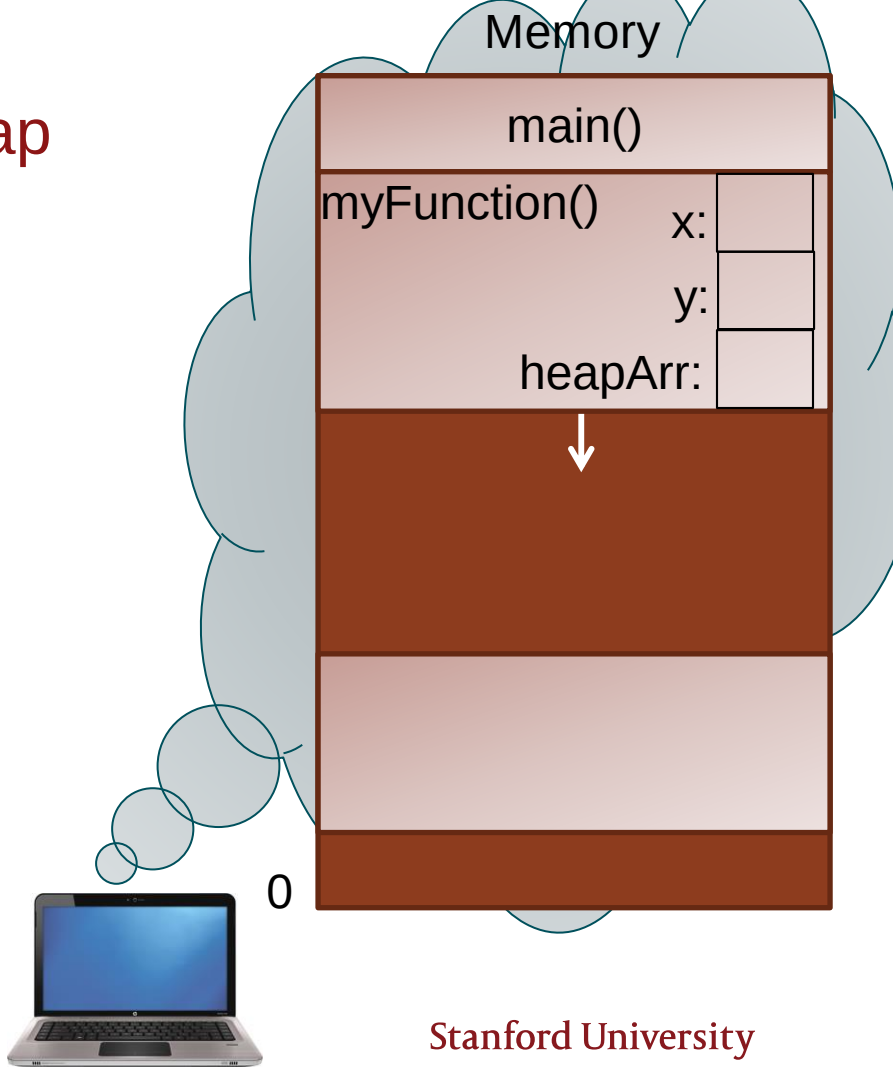
- A **dynamically allocated** array.
- The variable that refers to the array is a **pointer**.
- The memory allocated for the array must be manually released, or else the program will have a **memory leak**. (>_<)

```
delete[] name;
```

- Manually releases the memory back to the computer.

Memory on the stack and heap

```
void myFunction() {
    int x = 5;
    int y = 3;
    int *heapArr = new int[2];
    heapArr[0] = randomInteger(0,3);
    // bad -- memory leak coming!
}
```



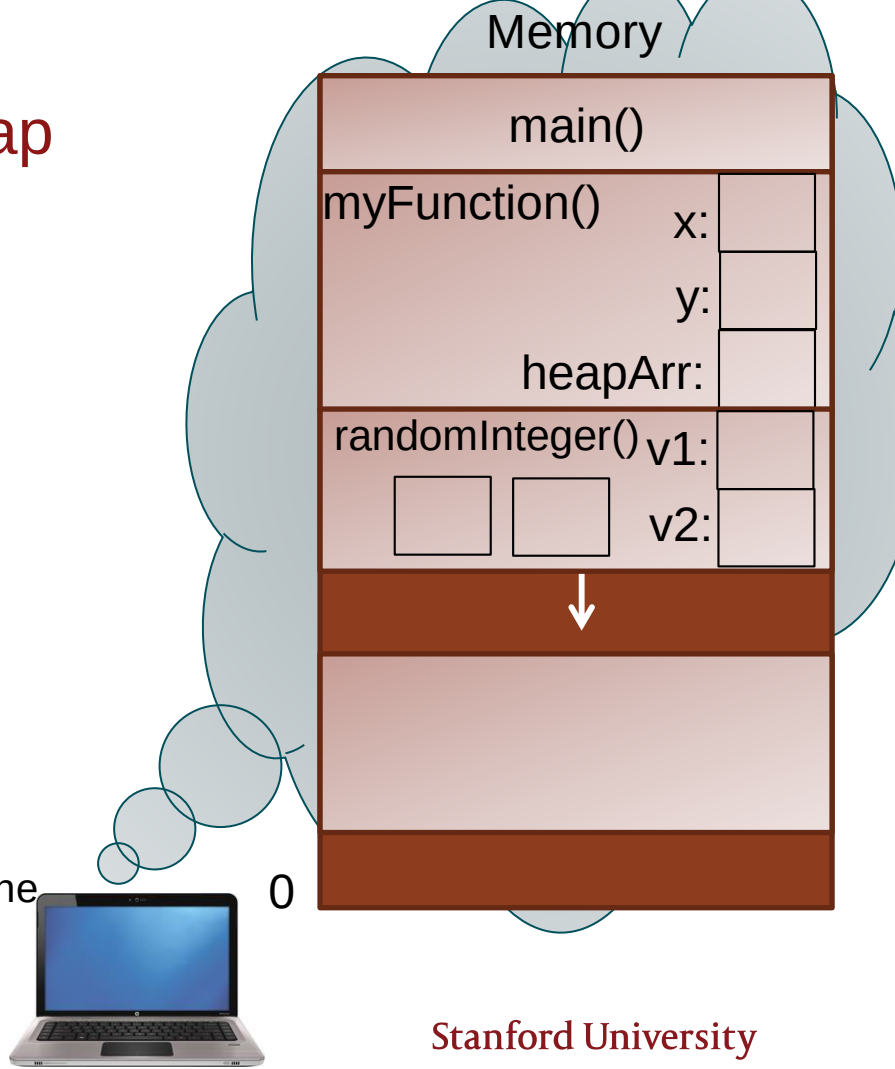
Memory on the stack and heap

```
void myFunction() {
   int x = 5;
   int y = 3;
   int *heapArr = new int[2];
   heapArr[0] = randomInteger(0,3);
   // bad -- memory leak coming!
}

void randomInteger(int low, int high) {
   int var1 = 5;
   double var2 = 3.14159;
…
```

What happens when `myFunction()` and `randomInteger()` return?

Why do we need to delete `heapArr`, but not the other variables (`x, y, v1, v2`)?



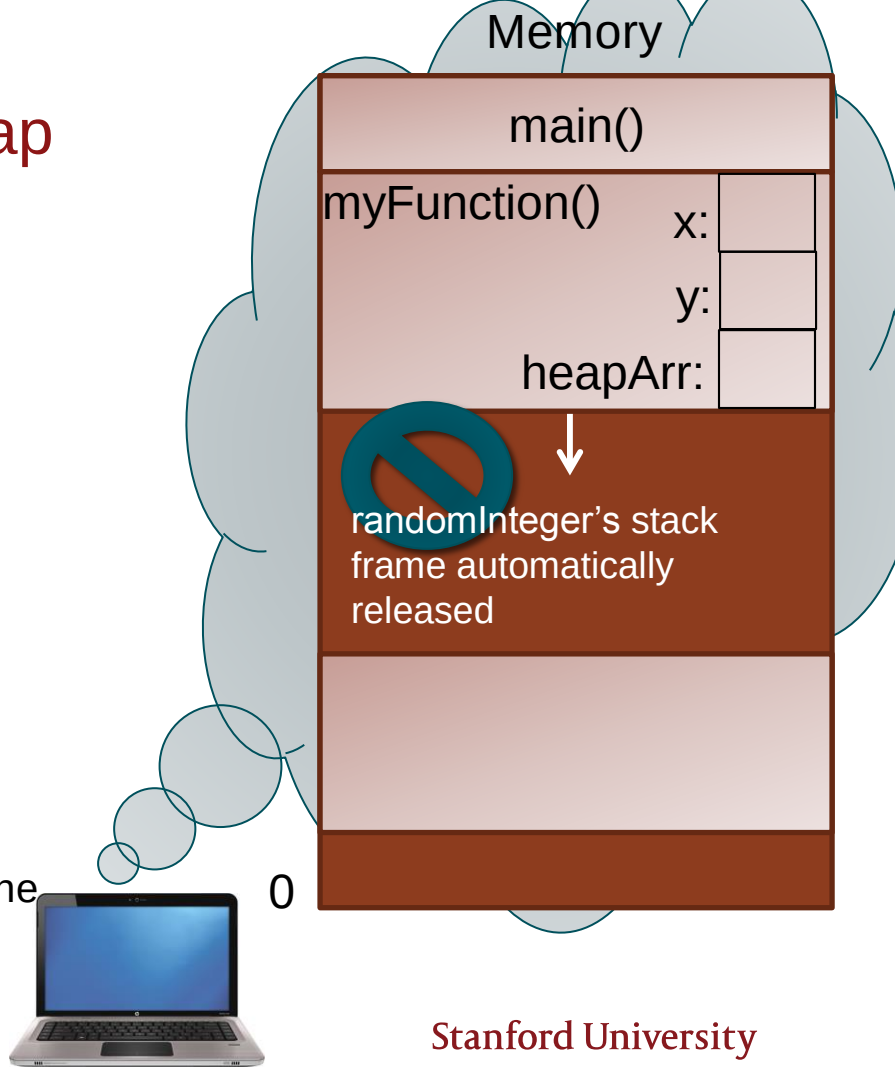
Memory on the stack and heap

```
void myFunction() {
   int x = 5;
   int y = 3;
   int *heapArr = new int[2];
   heapArr[0] = randomInteger(0,3);
   // bad -- memory leak coming!
}

void randomInteger(int low, int high) {
   int var1 = 5;
   double var2 = 3.14159;
…
```

What happens when `myFunction()` and `randomInteger()` return?

Why do we need to delete `heapArr`, but not the other variables (`x, y, v1, v2`)?



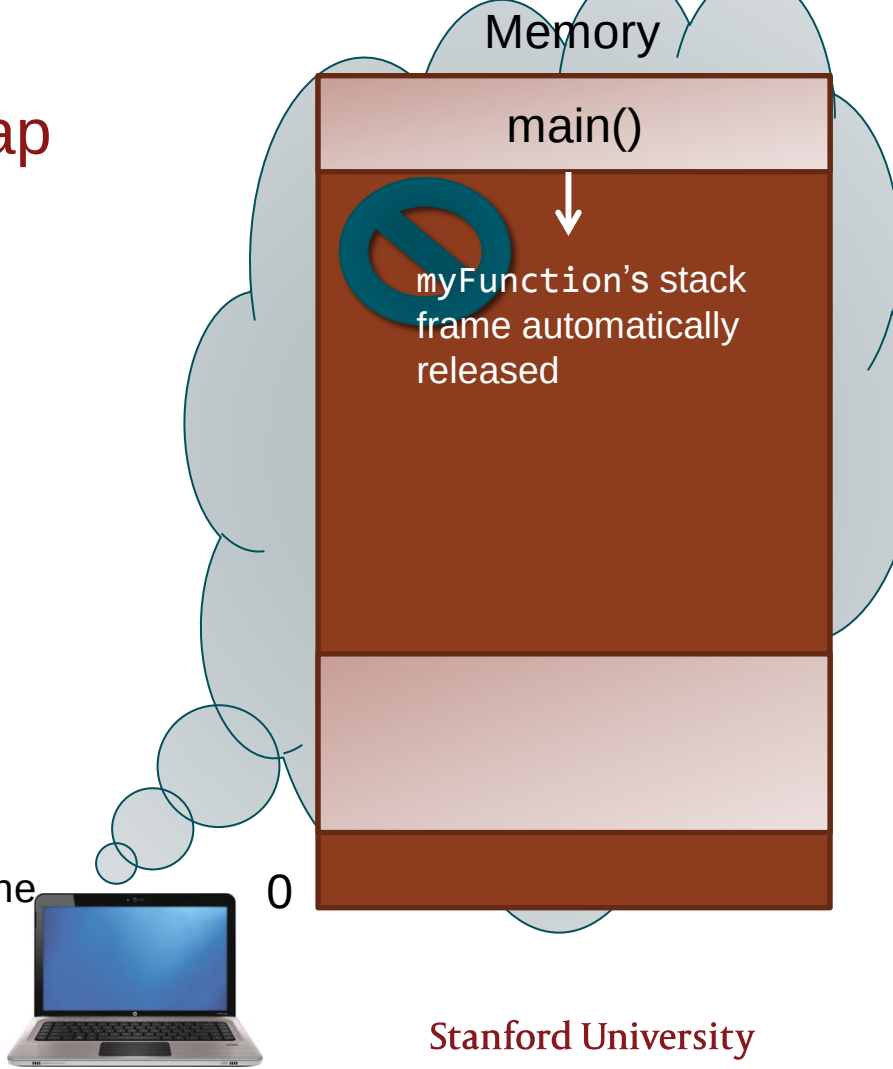
Memory on the stack and heap

```
void myFunction() {
   int x = 5;
   int y = 3;
   int *heapArr = new int[2];
   heapArr[0] = randomInteger(0,3);
   // bad -- memory leak coming!
}

void randomInteger(int low, int high) {
   int var1 = 5;
   double var2 = 3.14159;
…
```

What happens when `myFunction()` and `randomInteger()` return?

Why do we need to delete `heapArr`, but not the other variables (`x, y, v1, v2`)?



Always a pair: new and delete

Sample codes from Friday:

```
// a simple main
int main() {
    int* a = new int[3];
     a[0] = 42;
     a[1] = -5;
     a[2] = 17;
     delete[] a;
      return 0;
}
```

```
// constructor and destructor
// in ArrayList.cpp

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


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

What is a pointer?

Anything wrong with this struct?

```
struct Album {
  string title;
  int year;

  string artist_name;
  int artist_age;
  string artist_favorite_food;
  int artist_height; // in cm
};
```

Anything wrong with this struct?

```
struct Album {
  string title;
  int year;

  string artist_name;
  int artist_age;
  string artist_favorite_food;
  int artist_height; // in cm
};
```

Style-wise seems awkward - "**artist_**" prefix on fields

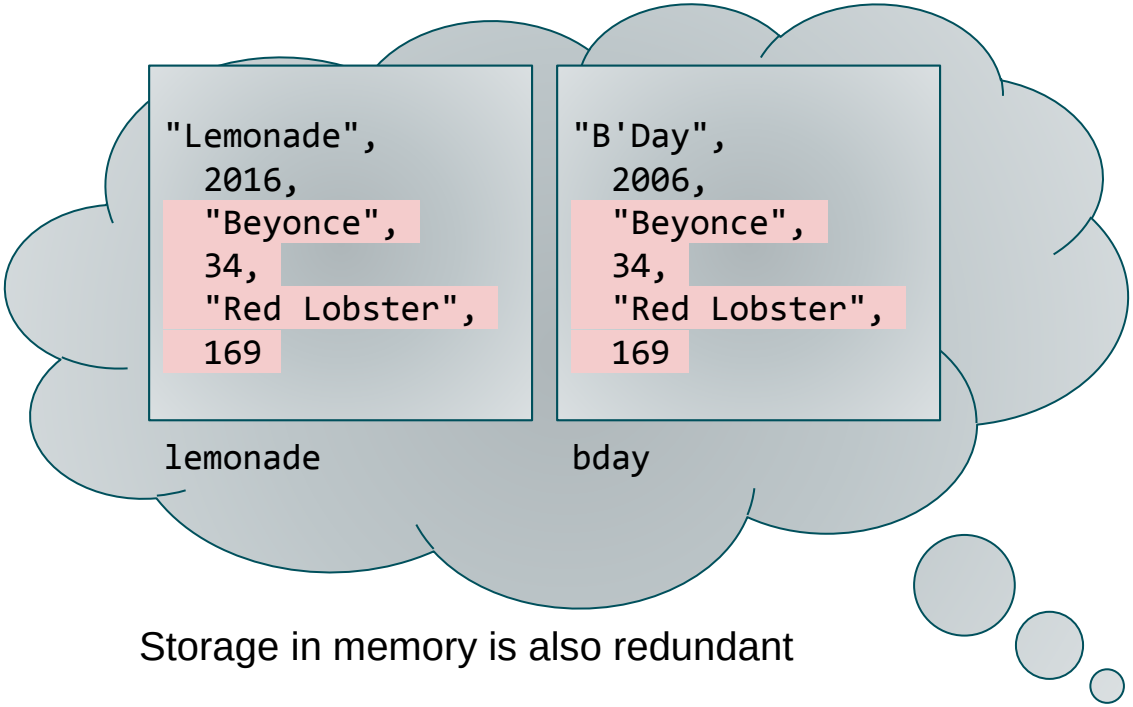
Anything else? How many times do you construct the artist info?

```
Album lemonade = {
  "Lemonade",
  2016,
  "Beyonce",
  34,
  "Red Lobster",
  169
};

Album bday = {
  "B'Day",
  2006,
  "Beyonce",
  34,
  "Red Lobster",
  169
};
```

Redudant code to declare an initialize these two album variables, lemonade and bday

It's redundantly stored, too



Storage in memory is also redundant

How do we fix this?

```
struct Album {
  string title;
  int year;

  string artist_name;
  int artist_age;
  string artist_favorite_food;
  int artist_height; // in cm
};
```

Should probably be another struct?

Does this fix the redundancy?

```
struct Artist {
  string name;
  int age;
  string favorite_food;
  int height; // in cm
};
```

```
struct Album {
  string title;
  int year;
  Artist artist;
};
```

```
Artist britney = { "Britney Spears", 34, "Snickers", 163};

Album blackout = { "Blackout", 2007, britney };
Album circus = { "Circus", 2008, britney };
Album femme_fatale = { "Femme Fatale", 2011, britney };
```

What does this look like in memory?

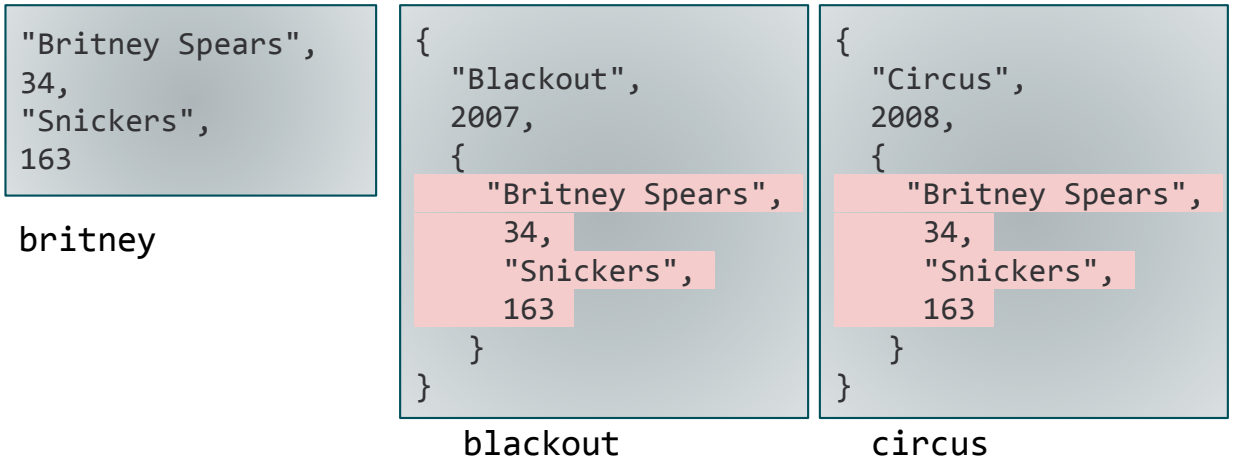
What does it mean when you have a struct field?

```
struct Album {
  string title;
  int year;
  Artist artist;
};
```

This embeds **all the fields** of the `Artist` struct into the `Album` struct.

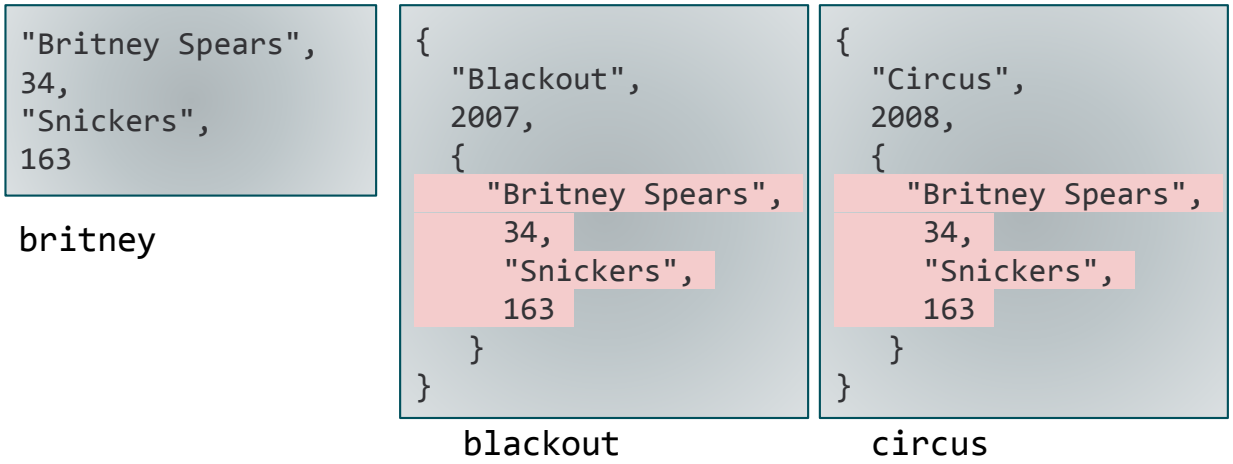
```
struct Artist {
  string name;
  int age;
  string favorite_food;
  int height; // in cm
};
```

Still stored redundantly




```
Artist britney = { "Britney Spears", 34, "Snickers", 163};
Album blackout = { "Blackout", 2007, britney };
Album circus = { "Circus", 2008, britney };
```

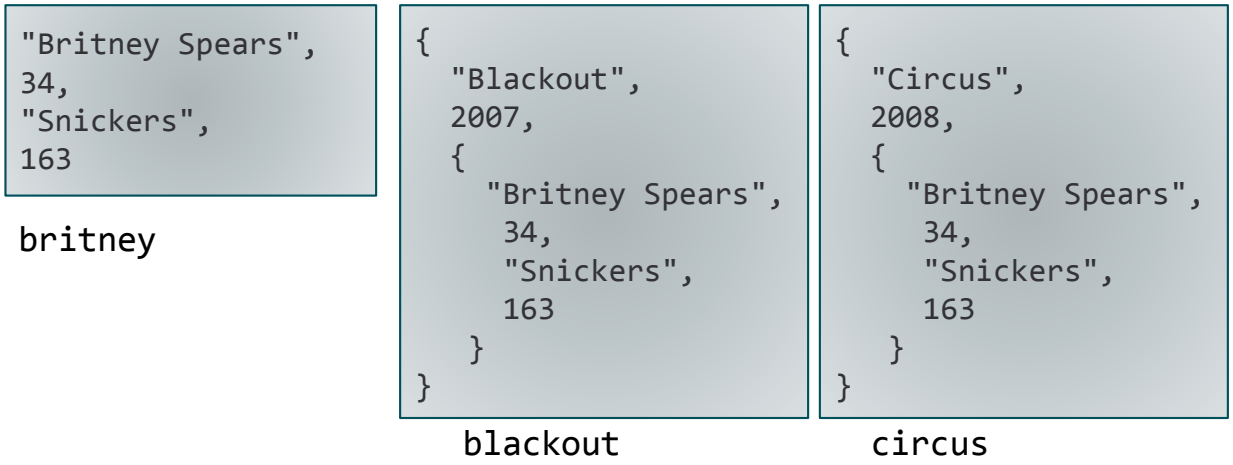
Still stored redundantly




```
Artist britney = { "Britney Spears", 34, "Snickers", 163};
Album blackout = { "Blackout", 2007, britney };
Album circus = { "Circus", 2008, britney };
```

All the fields of **britney** are **copied** in this step!

Still stored redundantly



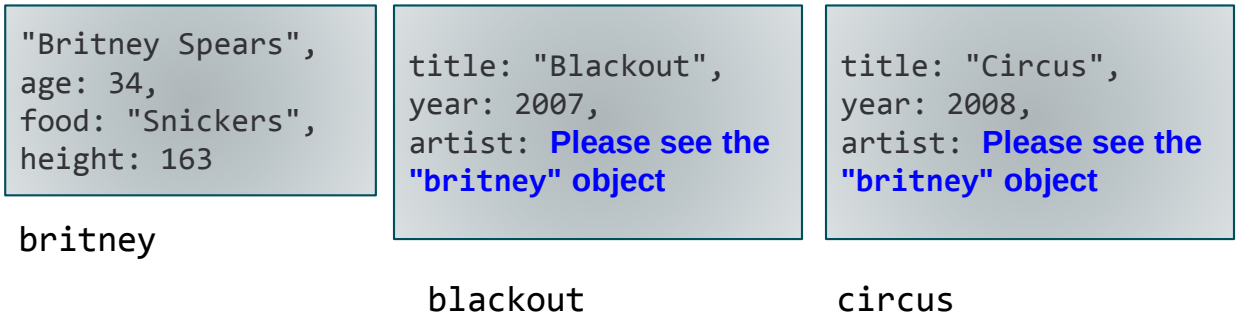

```
Artist britney = { "Britney Spears", 34, "Snickers", 163};
Album blackout = { "Blackout", 2007, britney };
Album circus = { "Circus", 2008, britney };

britney.favorite_food = "Twix";
```

What happens to the data?

(a) All 3 Snickers change to Twix (b) only britney Snickers changes to Twix
(c) only blackout/circus Snickers changes to Twix

What do we really want?



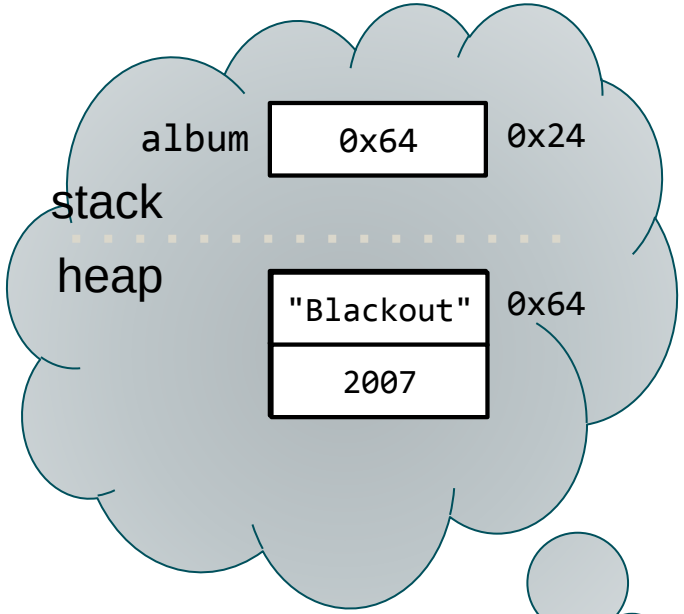
The album's artist field should **point to** the "`britney`" data structure instead of storing it.

How do we do this in C++?
…pointers!

new with objects

Example:

```
Album* album = new Album;
album->title = "Blackout";
album->year = 2007;
```



Pointers

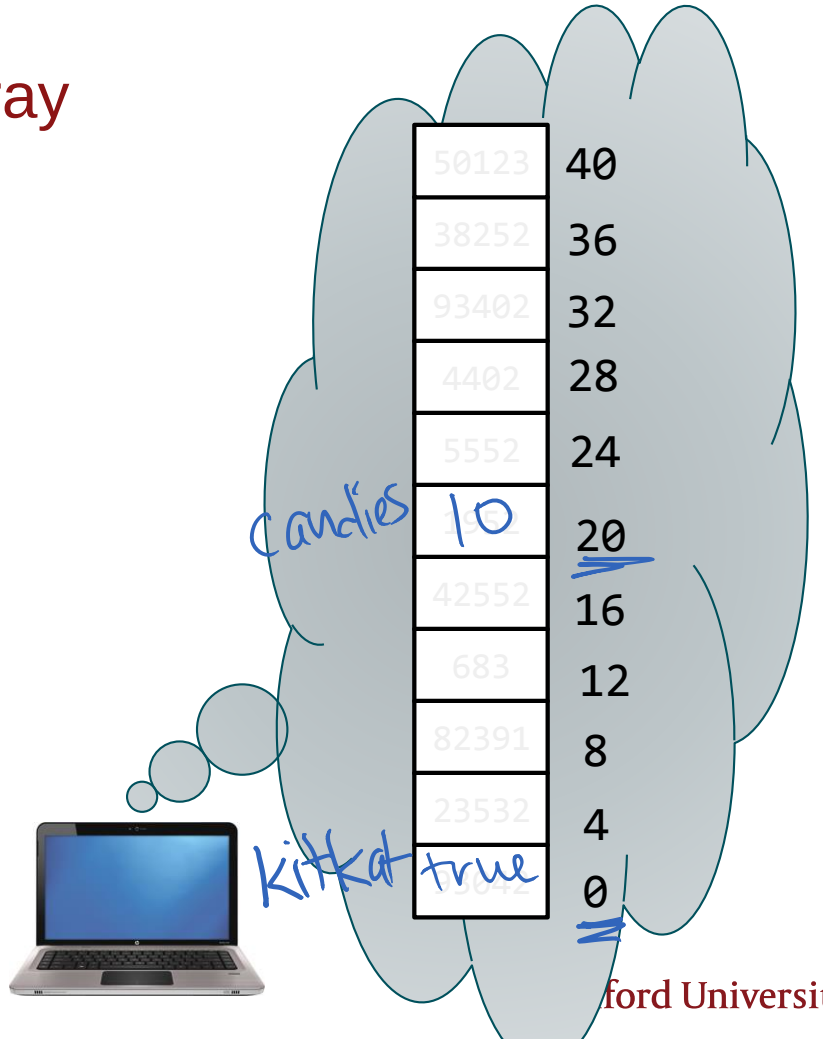
Taking a deeper look at the syntax of that array on the heap

Memory is a giant array

```
bool kitkat = true;
int candies = 10;
```

Whenever you declare a variable, you allocate a bucket (or more) of memory for the value of that variable

Each bucket of memory has a **unique address**



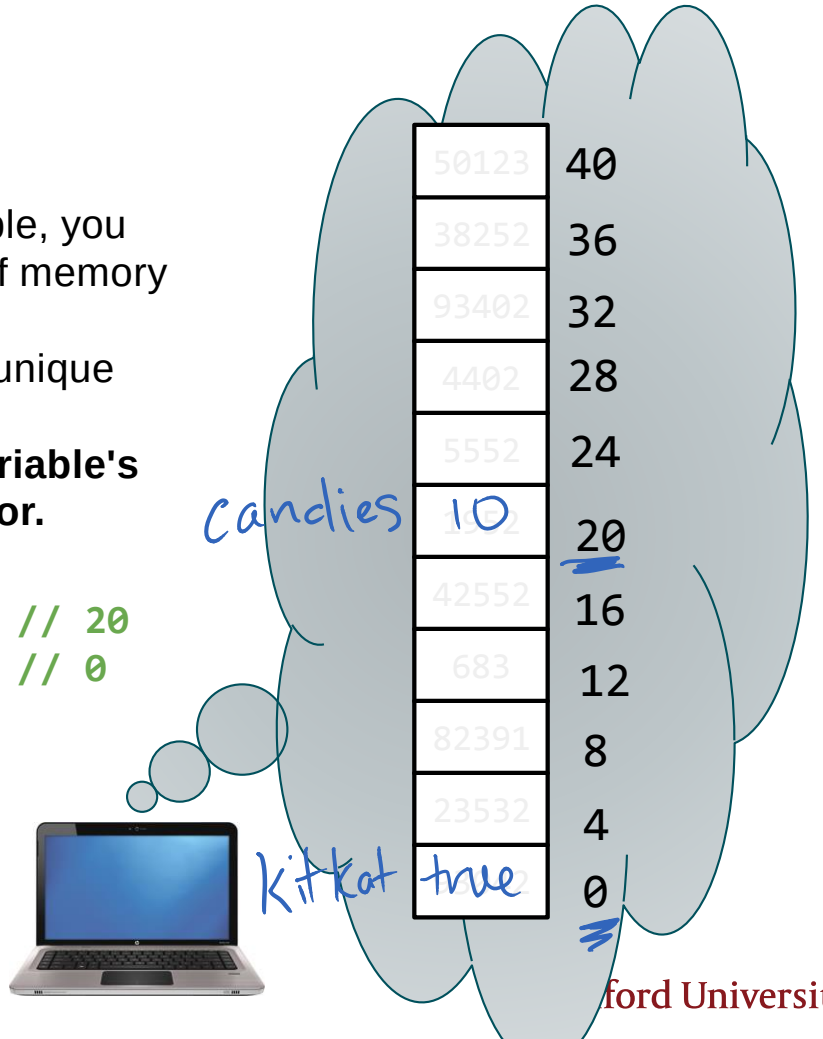
Memory addresses

Whenever you declare a variable, you allocate a bucket (or more) of memory for the value of that variable

Each bucket of memory has a unique address

You can get the value of a variable's address using the & operator.

```
cout << &candies << endl;    // 20
cout << &kitkat << endl;     // 0
```



Memory addresses

You can store memory addresses in a special type of variable called a **pointer**.

- i.e. A pointer is a variable that holds a memory address.

You can declare a pointer by writing *(The type of data it points at)**

- e.g. `int*`, `string*`

```
cout << &candies << endl;    // 20
cout << &kitkat << endl;     // 0
int* ptrC = &candies;    // 20
bool* ptrB = &kitkat;     // 0
```

Bonus fact — you can also do this:

```
cout << &ptrC << endl; //12
cout << &ptrB << endl; //4
```