# Programming Abstractions

CS106B

Cynthia Lee

# Topics:

- **Big-O**
  - Some practice examples

- **Memory and Pointers**
  - Arrays in C++
  - new/delete dynamic memory allocation
  - What is a pointer?

# Big-O

Applying to algorithms

# Applying Big-O to Algorithms

- Code example:

```
for (int i = data.size() - 1; i >= 0; i--){
   for (int j = 0; j < data.size(); j++){
      cout << data[i] << data[j] << endl;
   }
}
```

*is O(  ), where n is* `data.size()`.

# Applying Big-O to Algorithms

- Code example:

```
for (int i = data.size() - 1; i >= 0; i -= 3){
   for (int j = 0; j < data.size(); j += 3){
      cout << data[i] << data[j] << endl;
   }
}
```

*is O(  ), where n is* `data.size()`.

# Applying Big-O to Algorithms

- Code example:

```
for (int i = data.size() - 1; i >= 0; i -= 3){
   for (int j = 0; j < data.size(); j += 3){
      cout << data[i] << data[j] << endl;
   }
}
cout << BinarySearch(data, 3) << endl;
```

*is O(  ), where n is* `data.size()`.

# Applying Big-O to Algorithms

- Code example:

```
for (int i = 0; i < 5; i++) {
    cout << data[i] << endl;
}
```

*is O(  ), where n is* `data.size()`.

*(ok to make assumption this will not crash)*

# Arrays in C++

Like a Vector, but more primitive

# 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**.
  - This is a bad thing! More on that in a minute.

- Another array creation syntax that we will not use:

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

# Initialized?

```
type* name = new type[length];   // uninitialized
type* name = new type[length](); // initialize to 0
```

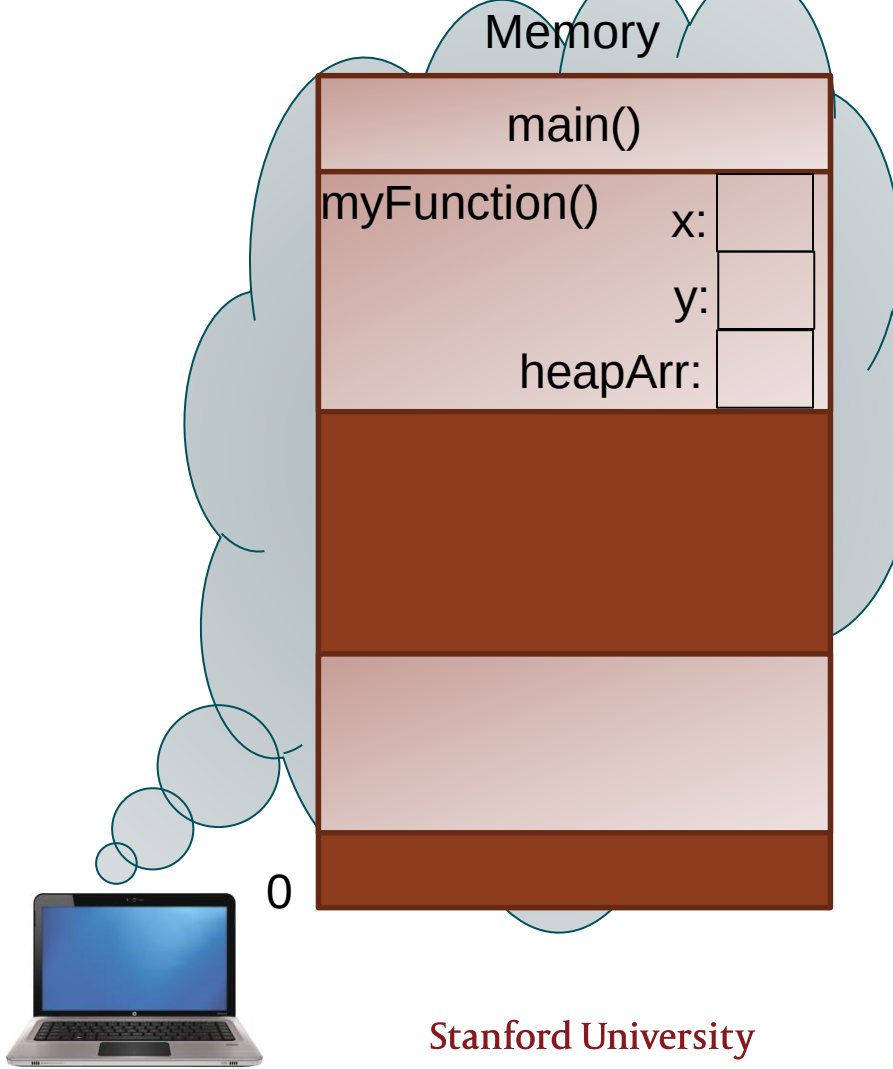
- If `()` are written after the array `[]`, it will set all array elements to their default zero-equivalent value for the data type. *(slower)*
- If no `()` are written, the elements are uninitialized, so whatever garbage values were stored in that memory beforehand will be your elements.

```
int* a = new int[3];
cout << a[0];              // 2395876
cout << a[1];              // -197630894

int* a2 = new int[3]();
cout << a[0];              // 0
cout << a[1];              // 0
```

# Arrays in a memory diagram

```
int myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x;
    heapArr[1] = y;
    heapArr[2] = x + y;

    return y; // bad -- memory leak!
}
```

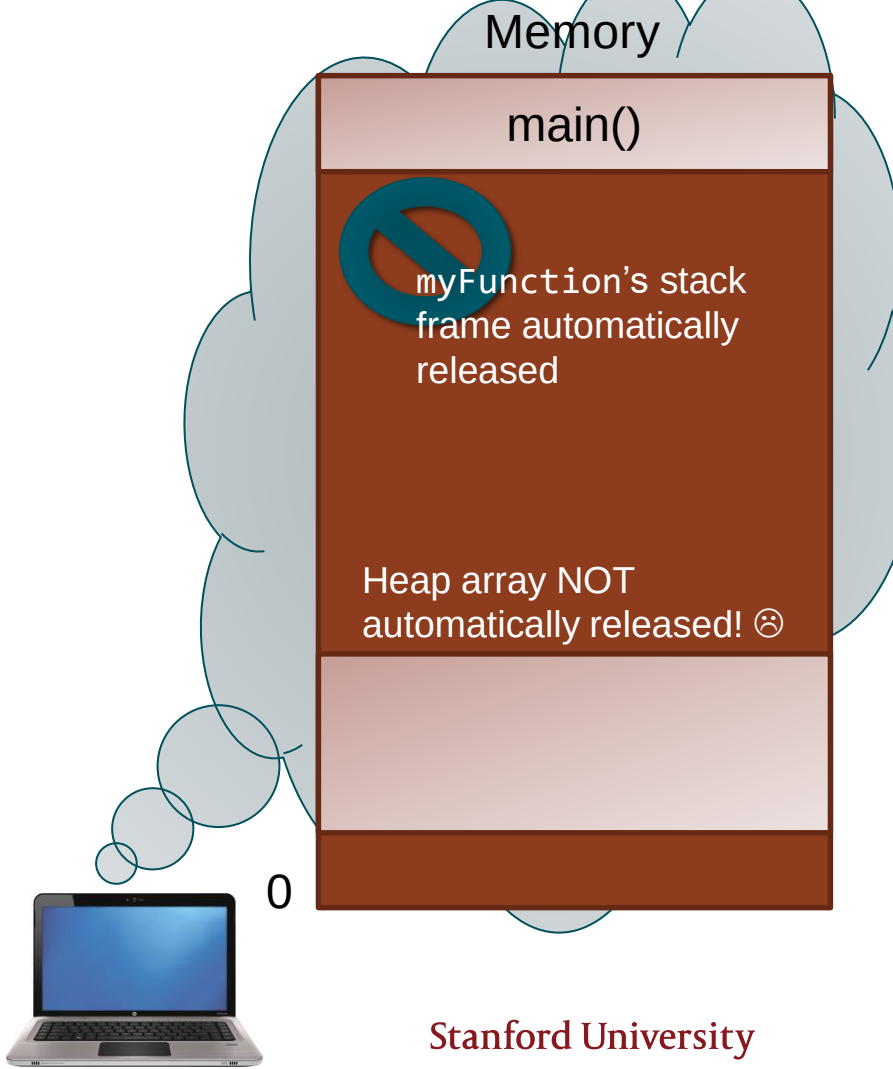
What happens when `myFunction()` returns?



# Arrays in a memory diagram

```
int myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x;
    heapArr[1] = y;
    heapArr[2] = x + y;

    return y; // bad -- memory leak!
}
```

What happens when `myFunction()` returns?

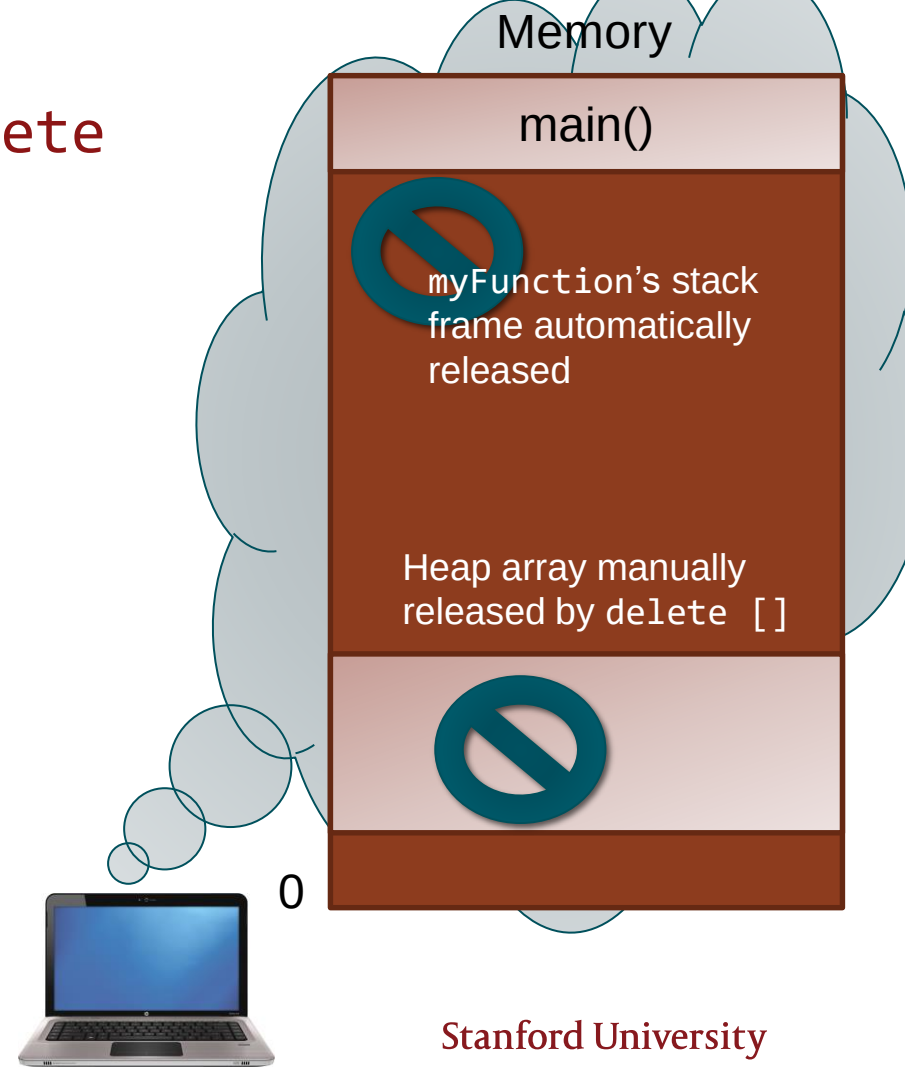


# **Always a pair**: new and delete

```
int myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x;
    heapArr[1] = y;
    heapArr[2] = x + y;
    delete [] heapArr; // fixed memory leak!
    return y;
}
```

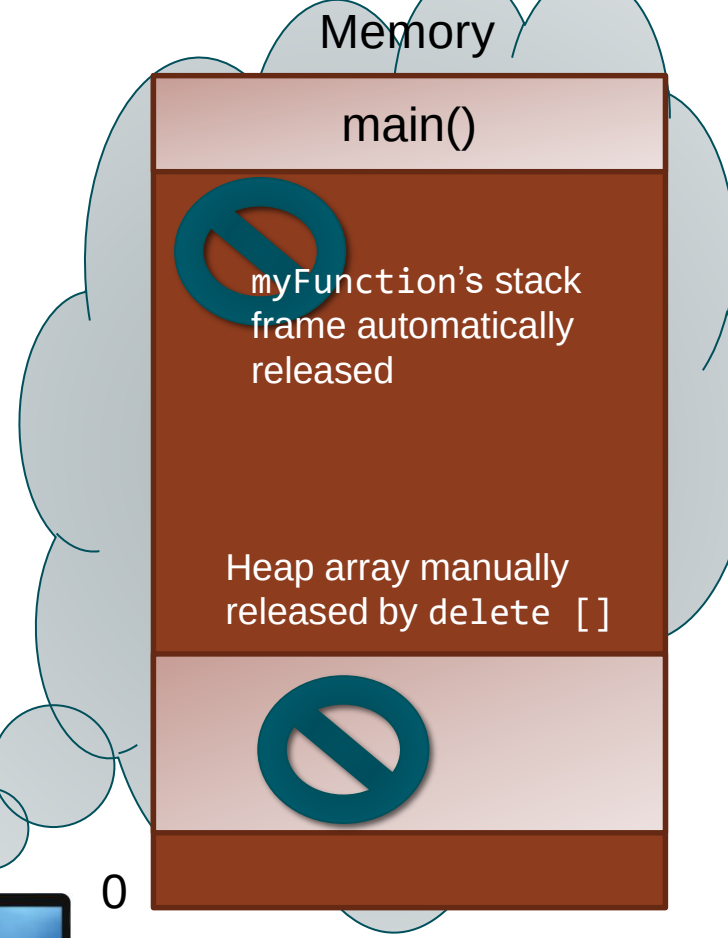
# **Always a pair**: new and delete

```
int myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x;
    heapArr[1] = y;
    heapArr[2] = x + y;
    delete [] heapArr; // fixed leak!
    return y;
}
```



# **Always a pair**: `new` and `delete`

```
int myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x;
    heapArr[1] = y;
    heapArr[2] = x + y;
    delete [] heapArr; // fixed leak!
    return y;
}
```



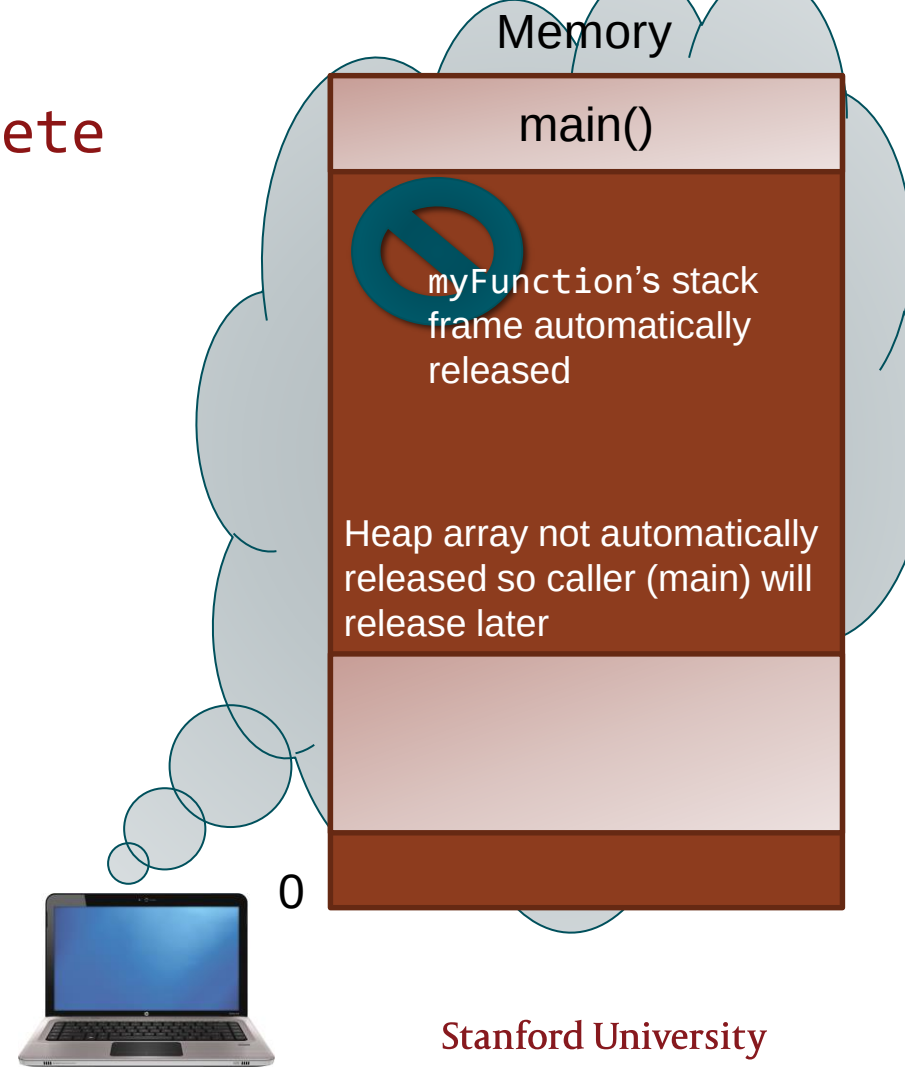

"Why would you want to do that?"

No point if we are just deleting at the end of the function when it would automatically release anyway. But what if we want to return the array?

0

# **Always a pair**: new and delete

```
int* myFunction() {
    int x = 5;
    int y = 3;
    int* heapArr = new int[3];
    heapArr[0] = x;
    heapArr[1] = y;
    heapArr[2] = x + y;
    return heapArr;
}

delete [] heapArr; //to be done later!
```



# What is a pointer?

More detail on that “pointer” type that we use for arrays

# A first struct

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

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

- Like a class, but simpler—just a collection of some variables together into a new type
- You can declare a variable of this type in your code now, and use "." to access fields:

```
Album lemonade;
lemonade.year = 2016;
lemonade.title = "Lemonade";
cout << lemonade.year << endl;
```

# Anything wrong with this struct design?

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

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

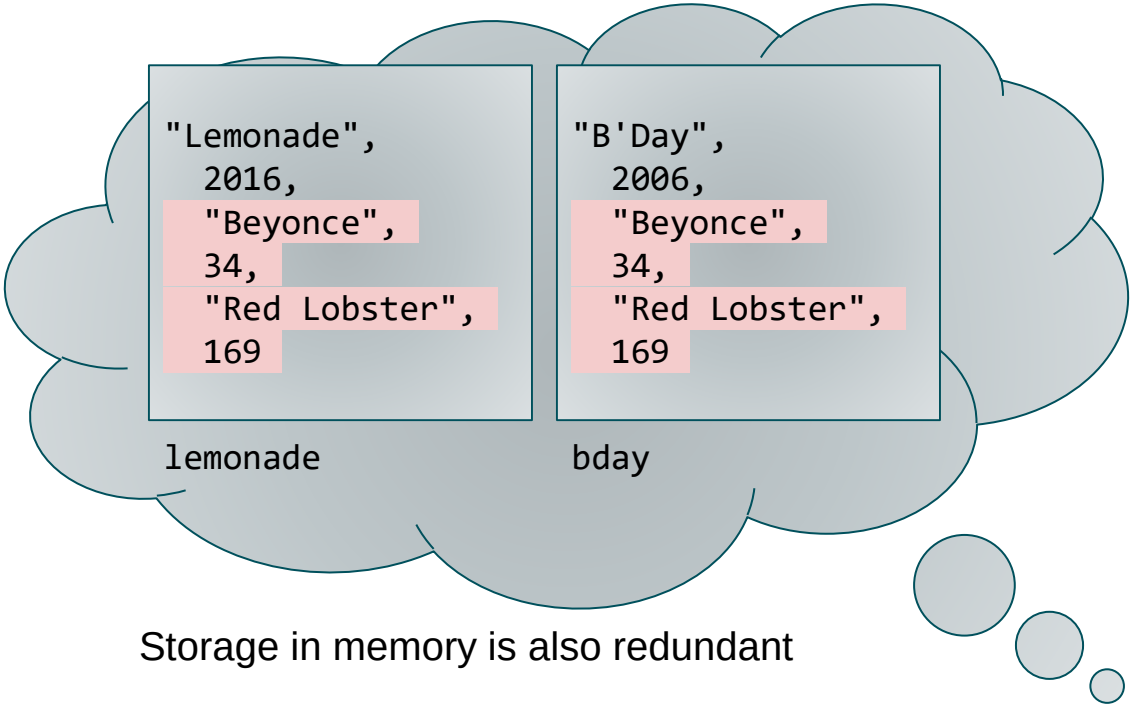
Style-wise seems awkward - "**artist_**" prefix on fields

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 and 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?

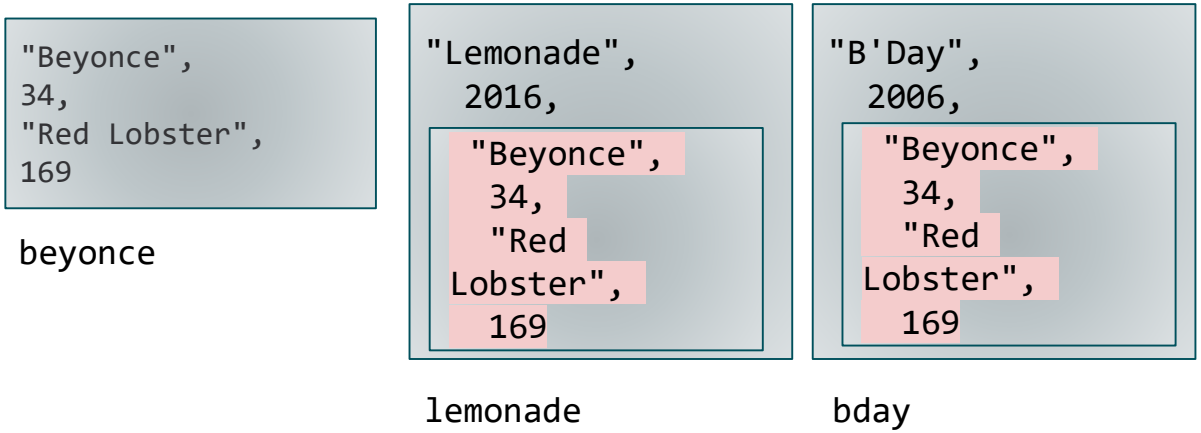
```
struct Artist {
  string name;
  int age;
  string favorite_food;
  int height; // in cm
};
```

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

```
Artist beyonce = { "Beyonce", 34, "Red Lobster", 169};

Album bday = { "B'Day", 2006, beyonce };
Album lemonade = { "Lemonade", 2016, beyonce };
```

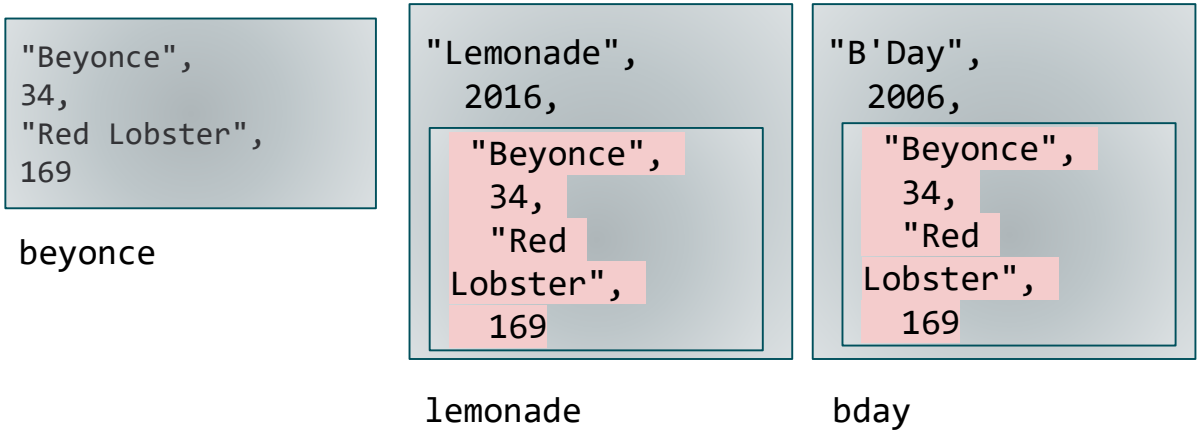
**What does this look like in memory?**

# Still stored redundantly




```
Artist beyonce = { "Beyonce", 34, "Red Lobster", 169};

Album bday = { "B'Day", 2006, beyonce };
Album lemonade = { "Lemonade", 2016, beyonce };
```

# Still stored redundantly

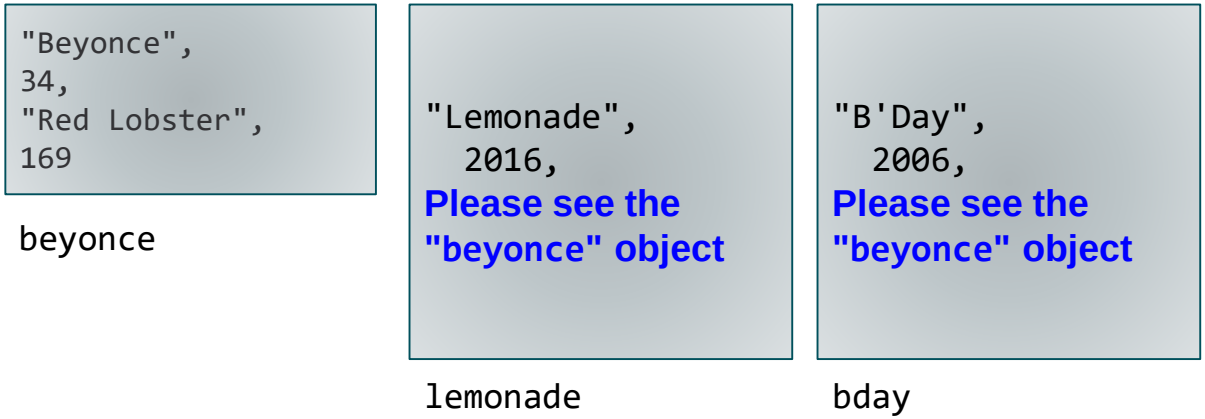



```
Artist beyonce = { "Beyonce", 34, "Red Lobster", 169};

Album bday = { "B'Day", 2006, beyonce };
Album lemonade = { "Lemonade", 2016, beyonce };

beyonce.favorite_food = "Twix";
```

**What happens to the data?**

(a) all 3 change to Twix
(b) only beyonce changes to Twix
(c) only lemonade/bday change to Twix

# What do we really want?




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

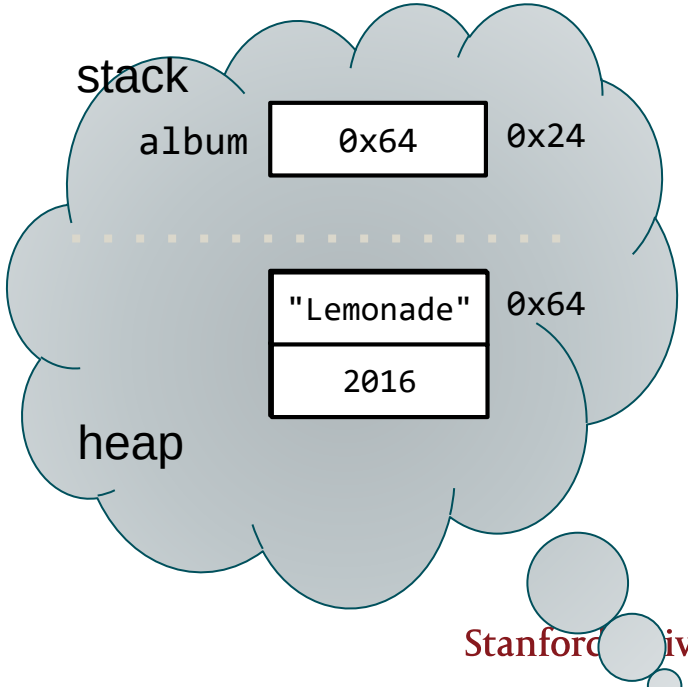
**How do we do this in C++?**

**…pointers!**

# new with pointers!

Example:

```
Album* album = new Album;
album->title = "Lemonade";
album->year = 2016;
```



# Pointers

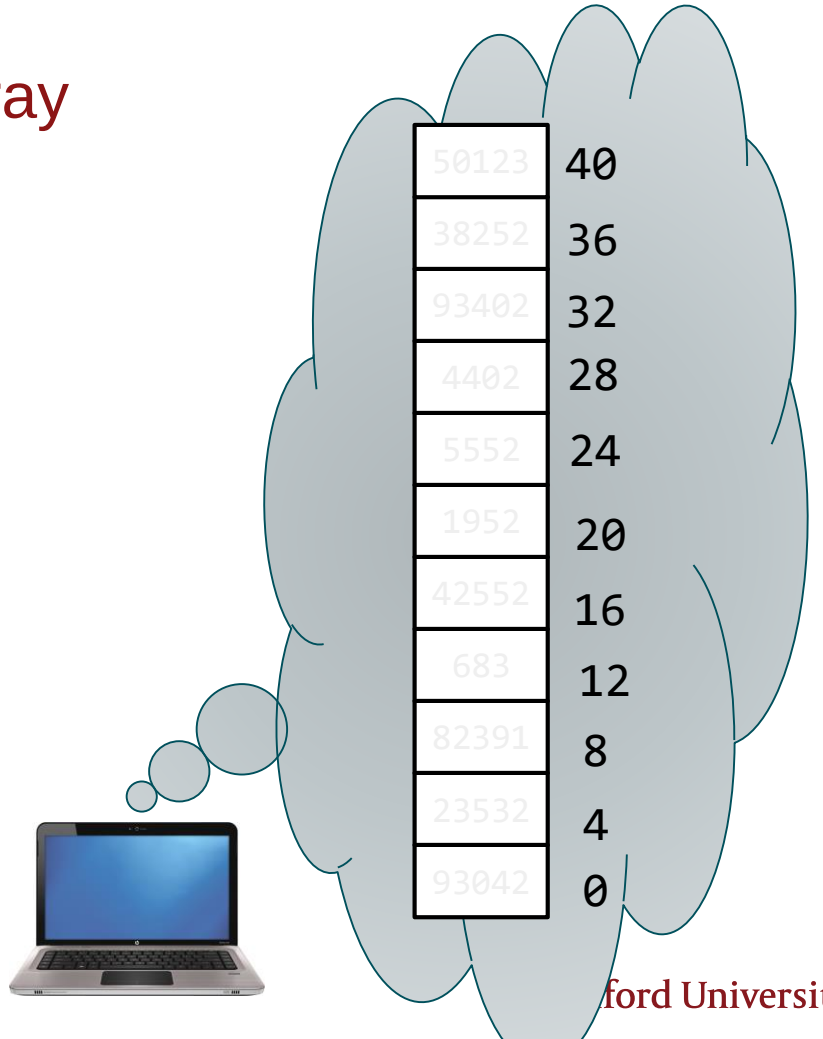
Taking a deeper look at the syntax of that array on the heap

# 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**



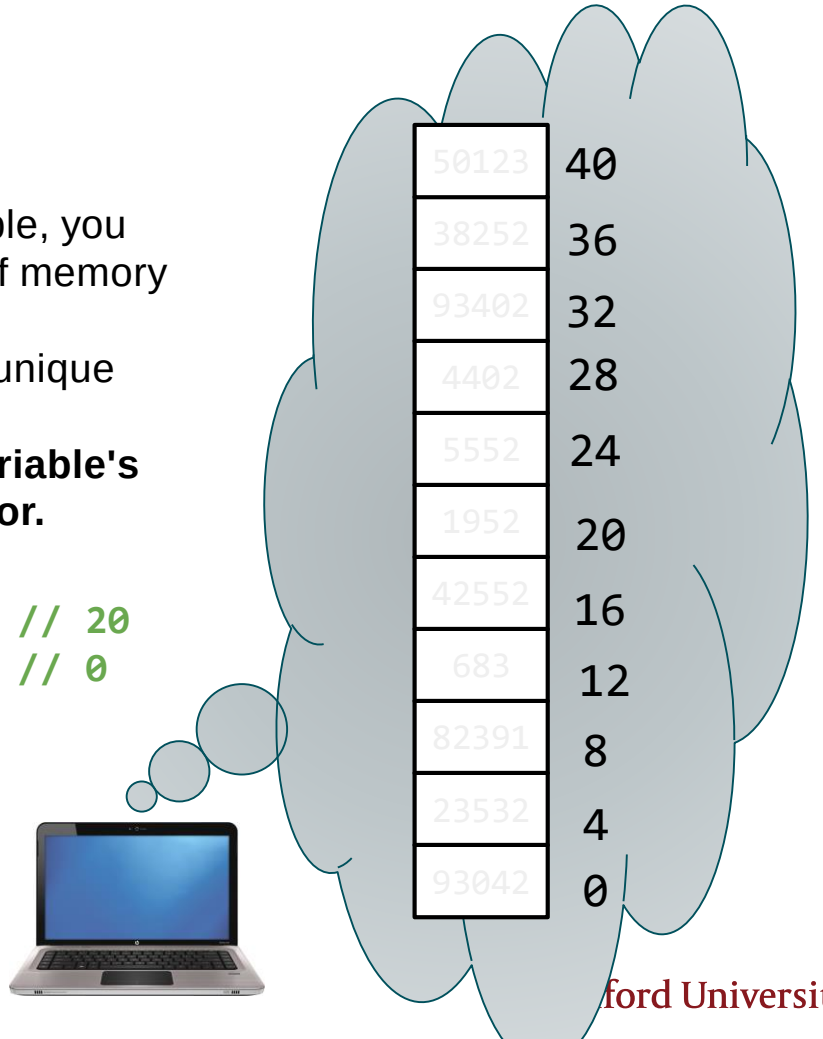
# Memory addresses

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
```