Programming Abstractions

CS106B

Cynthia Bailey Lee
Julie Zelenski

Topics:

- **Memory and Pointers**
  - Picking up where we left off with Monday's lecture, implementing ArrayStack
    - Arrays in C++
    - new/delete dynamic memory allocation
    - Uninitialized memory
  - C/C++ `struct` feature
  - What is a pointer?

Arrays in C++

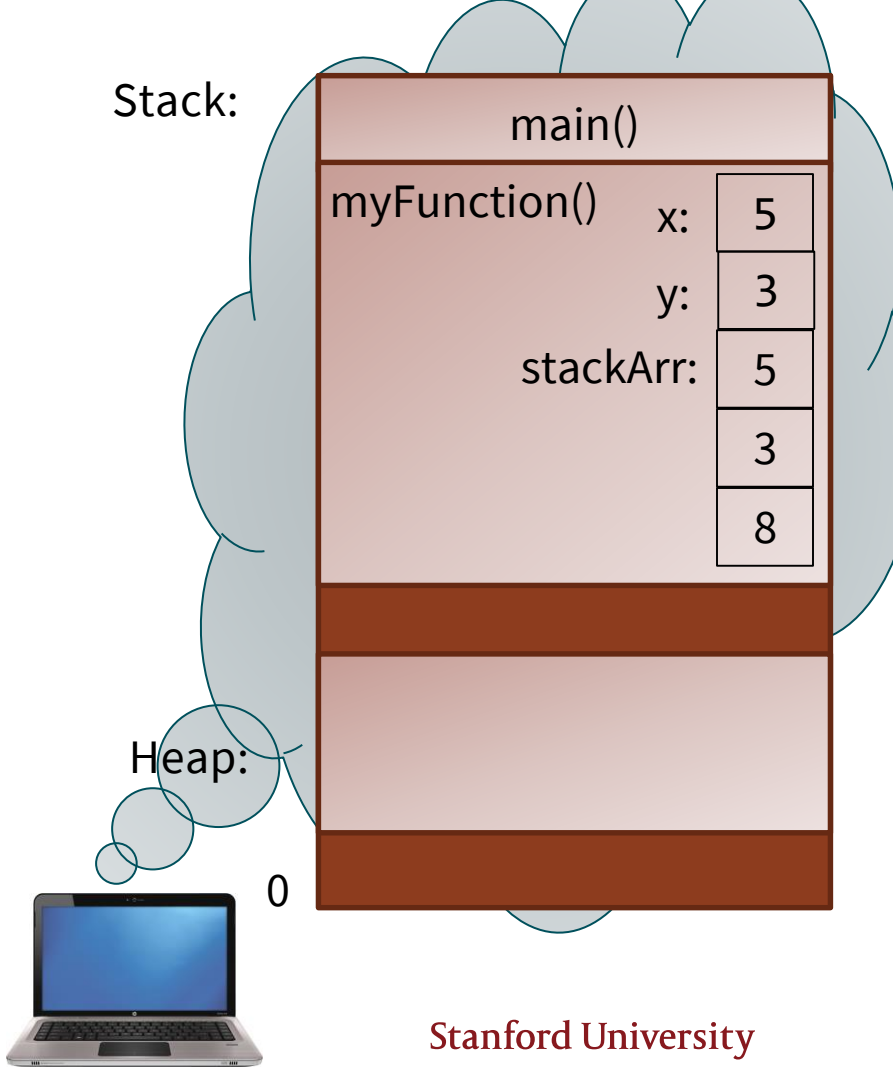
Like a Vector, but way more basic

Arrays in a memory diagram

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

    return y;
}
```

What happens when `myFunction()` returns?

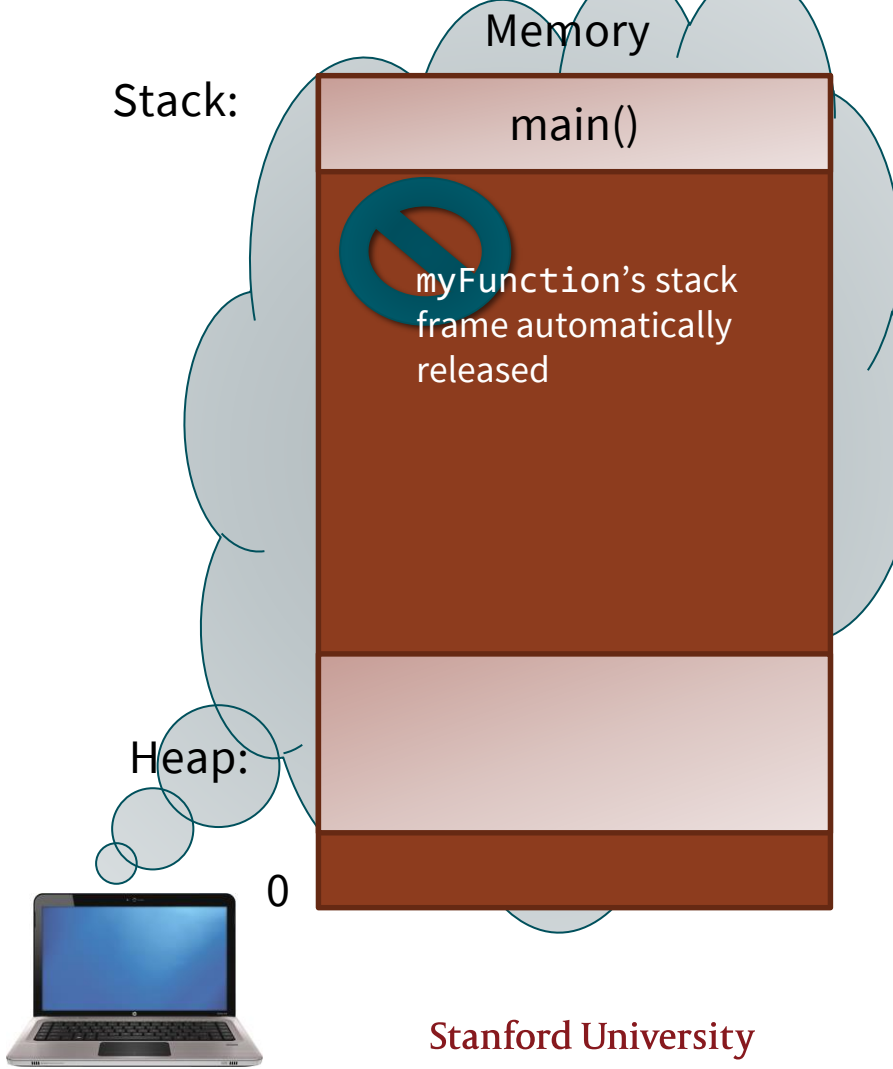


Arrays in a memory diagram

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

    return y;
}
```

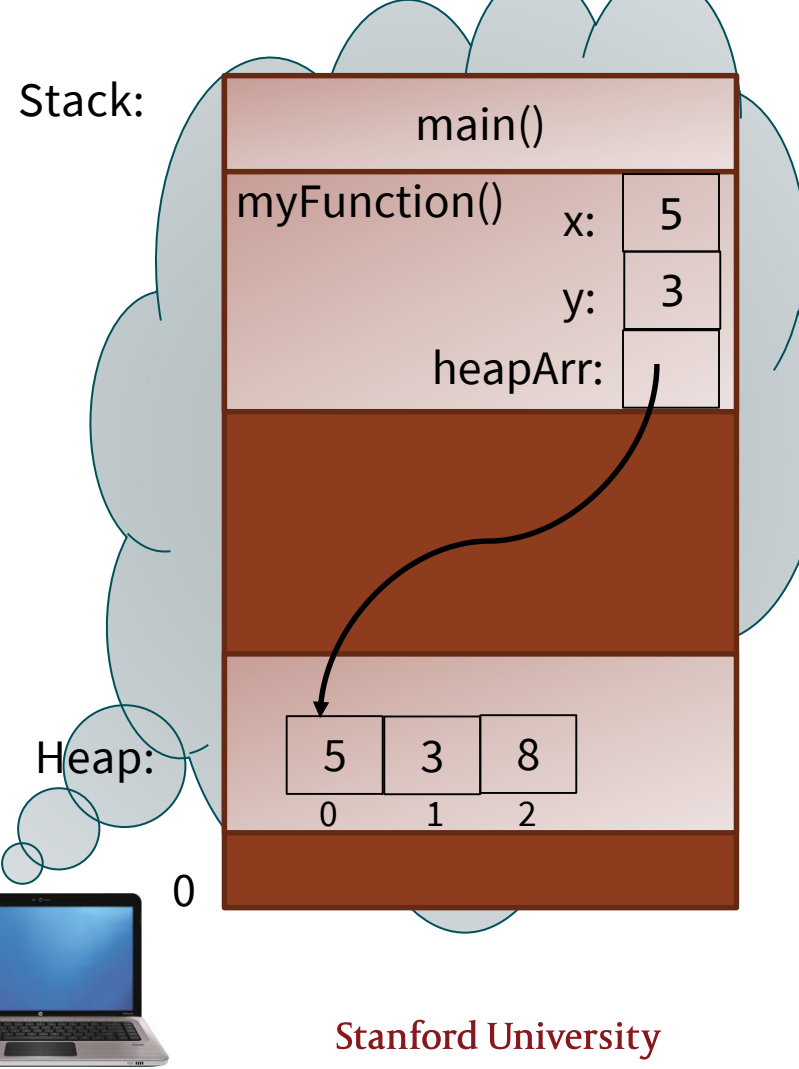
What happens when `myFunction()` returns?



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;
    delete [] heapArr;
    return y;
}
```

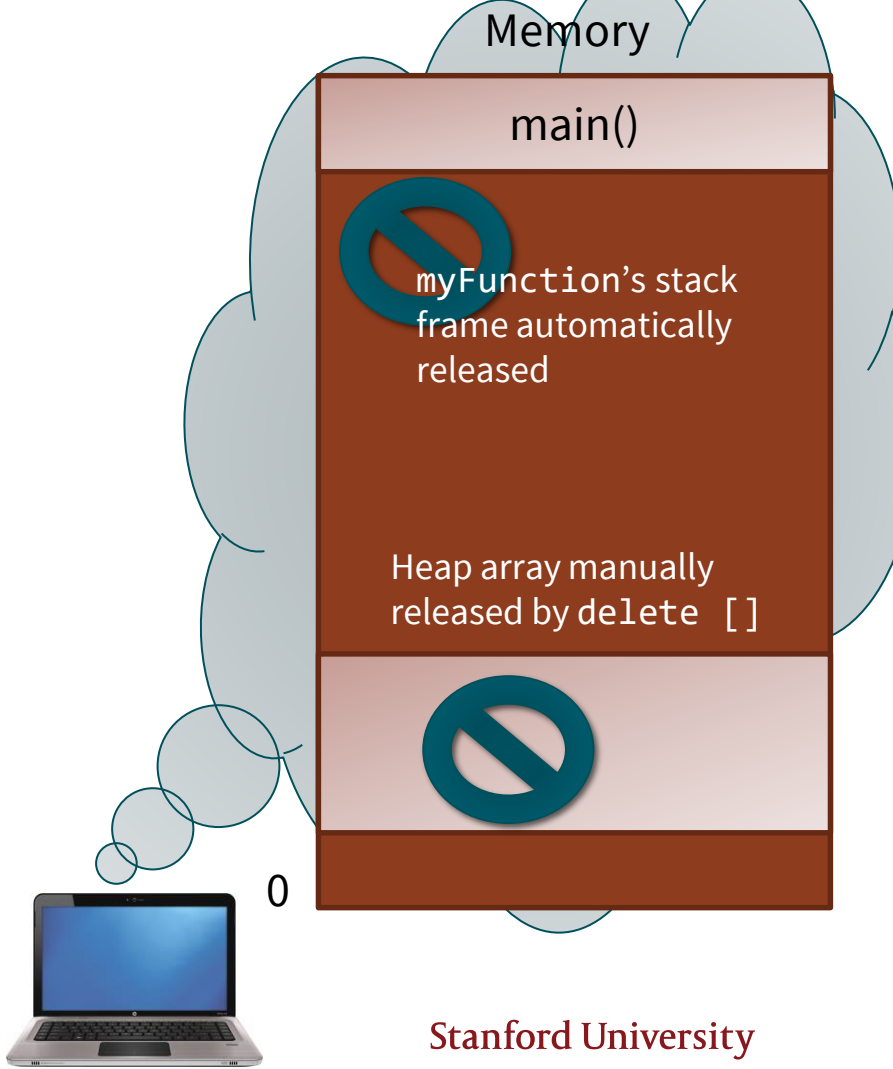
What happens when `myFunction()` returns?



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;
    delete [] heapArr;
    return y;
}
```

What happens when `myFunction()` returns?



Dynamic Memory Allocation

Keywords **new** and **delete**

Always a pair: `new` and `delete`

- Think of `new` as making a hotel room reservation.
  - `new int[5]` = "I'd like 5 connecting rooms, each big enough for 1 `int` value, please."
- Think of `delete` as checking out of the hotel room.
  - `delete [] arr` = "My trip is done. Stop charging me for these rooms, and you can give them to other guests."

Always a pair: `new` and `delete`

Many things can go wrong with dynamic memory that are analogous to the hotel situation:

- Leave town but forget to check out—you'll keep getting charged for the room and it can't go to another guest
  - When you forget `delete`, you get a memory leak
- Check out of the room but then try to go back in—another guest might already be using it and will be very angry!
  - After you call `delete`, be sure not to try to use that memory again!


```
int* arr = new int[10];
 …
delete [] arr;
arr[0] = 5; // no!!
```

Uninitialized Memory

(CODE DEMO)

Danger in C/C++: uninitialized memory!

```
type* name = new type[length];    // uninitialized
type* name = new type[length](); // initialized with zeroes
```

- If () are written after [], all elements are zeroed out (slower but good if needed)
- If () are missing, the elements store uninitialized ("random"/garbage) values

```
int* a1 = new int[3];
cout << a1[0];              // 2395876
cout << a1[1];              // -197630894

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

Pointers

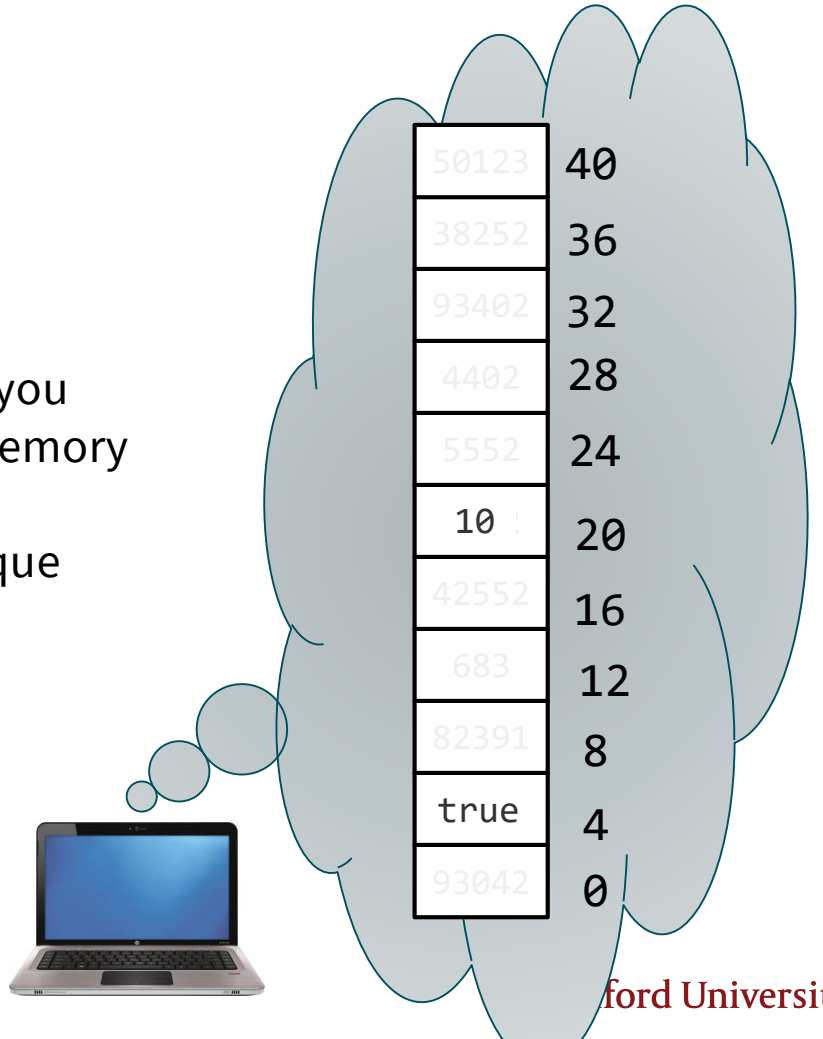
TAKING A DEEPER LOOK AT THE SYNTAX OF THAT ARRAY ON THE HEAP

Memory addresses

```
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

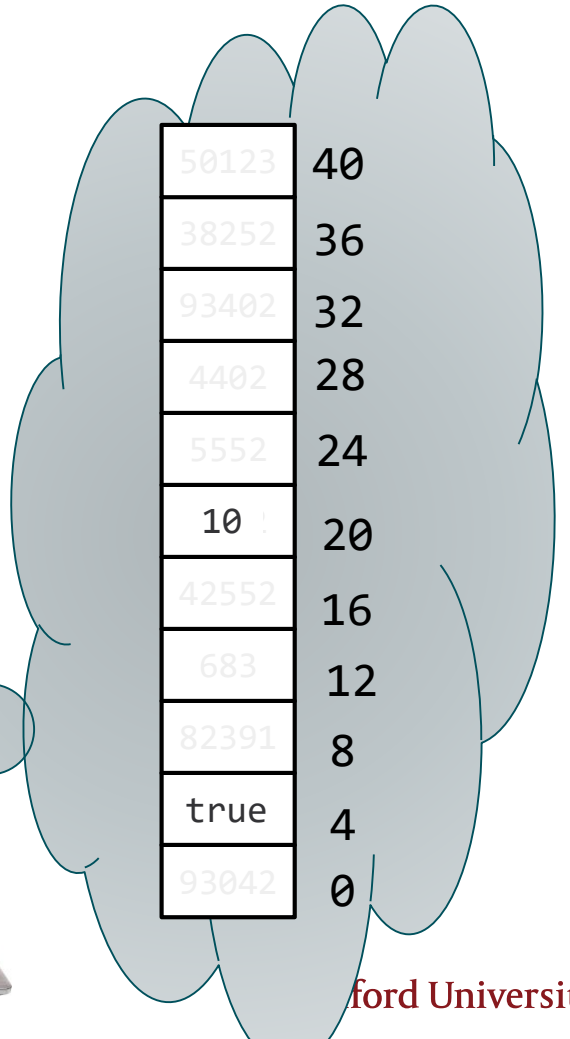
```
bool kitkat = true;
int candies = 10;
```

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

You can ask for any variable's address using the & operator.

```
cout << &candies << endl;    // 20
cout << &kitkat << endl;     // 4
```



Memory addresses

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

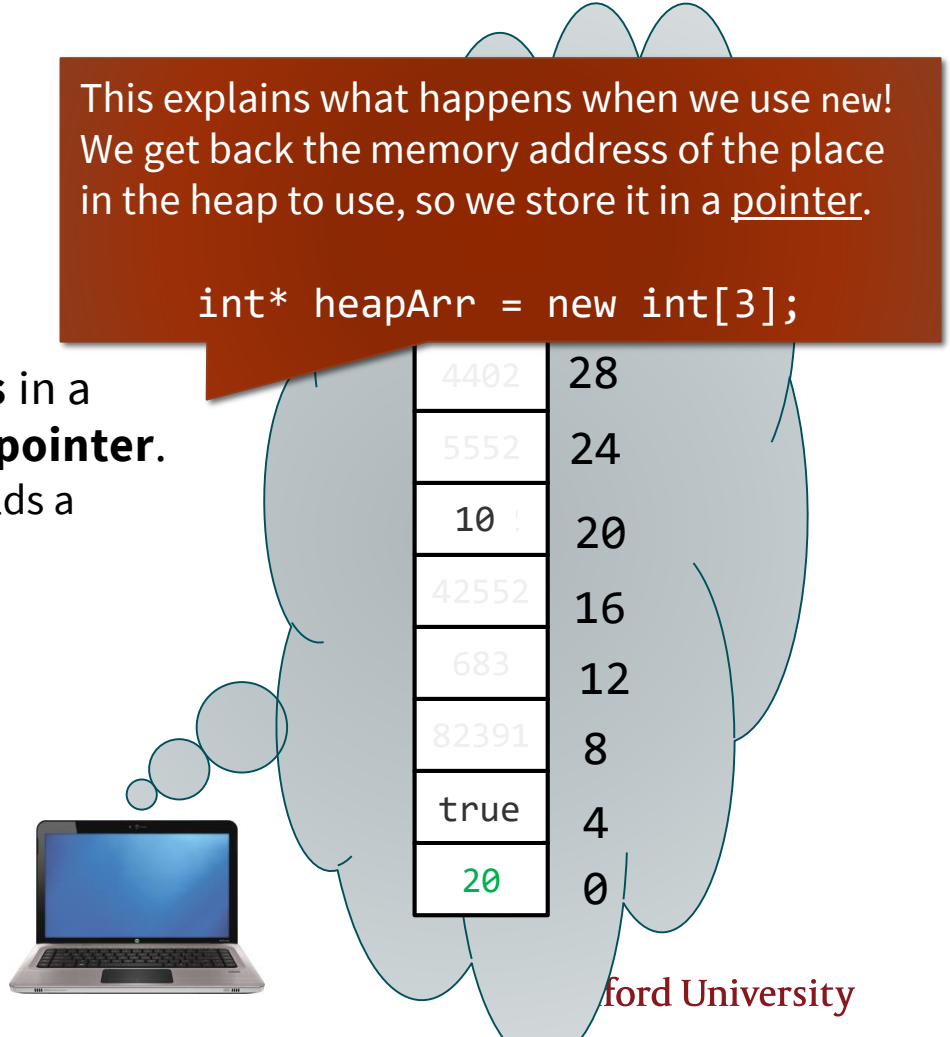
You can **store memory addresses** in a special type of variable called a **pointer**.

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

```
int* ptrC = &candies;     // 20
bool* ptrB = &kitkat;     // 4
```

This explains what happens when we use new! We get back the memory address of the place in the heap to use, so we store it in a pointer.

```
int* heapArr = new int[3];
```



Memory addresses

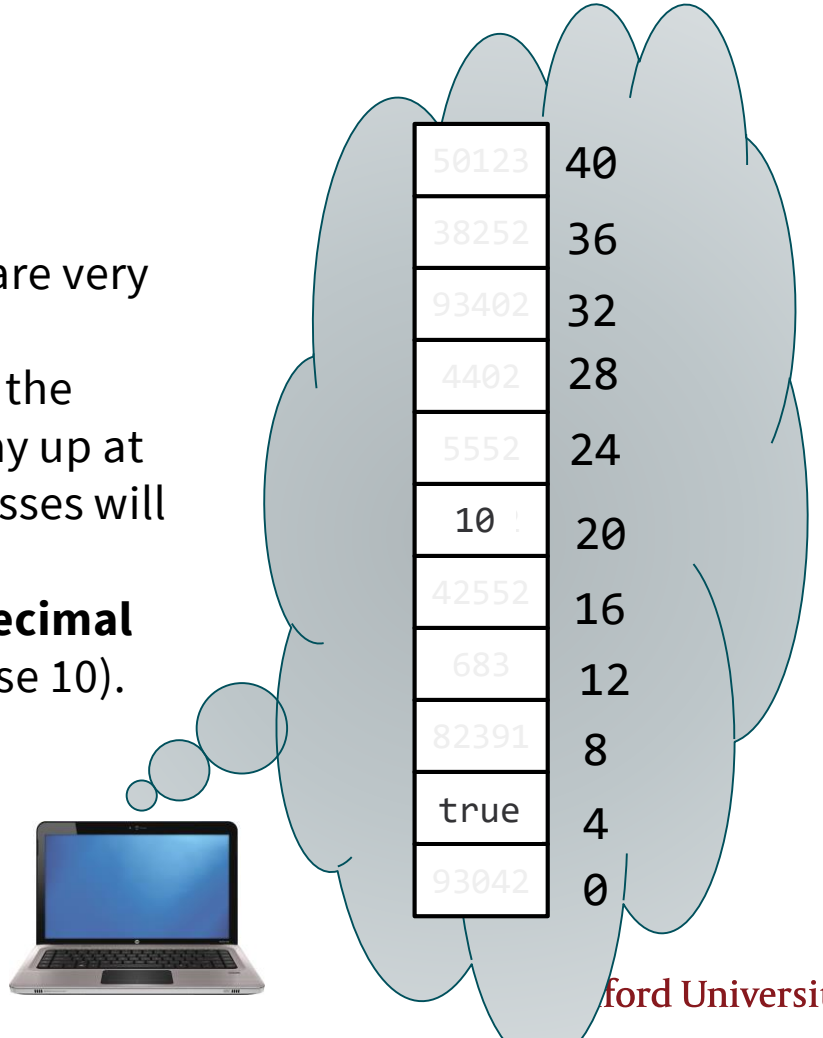
In our example here, the memory addresses of our local variables are very small numbers.

Remember that in a real situation, the stack part of memory is waaaaaay up at the end of memory, so the addresses will be quite large!

We typically **write them in hexadecimal (base 16)** instead of deciaml (base 10).

Example:

`0x7ffee40f1494`

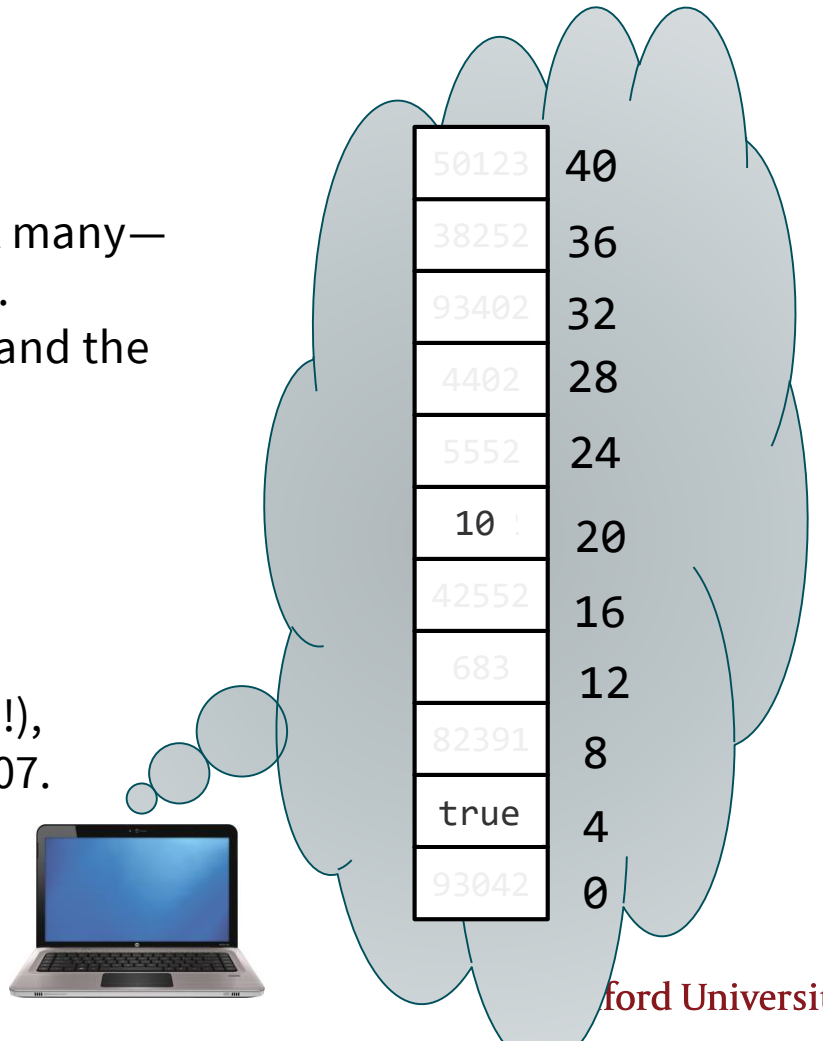


Memory addresses

"Pointer" isn't one type in C++ but many—
it **depends on what it points to**.

You can declare a pointer using * and the type pointed-to:

- `int* p`
- `bool*`
- `string*`
- `double*`
- `Queue<GridLocation>*`
- `int**` ← Yes this is possible (!!), you'll see this in CS107.

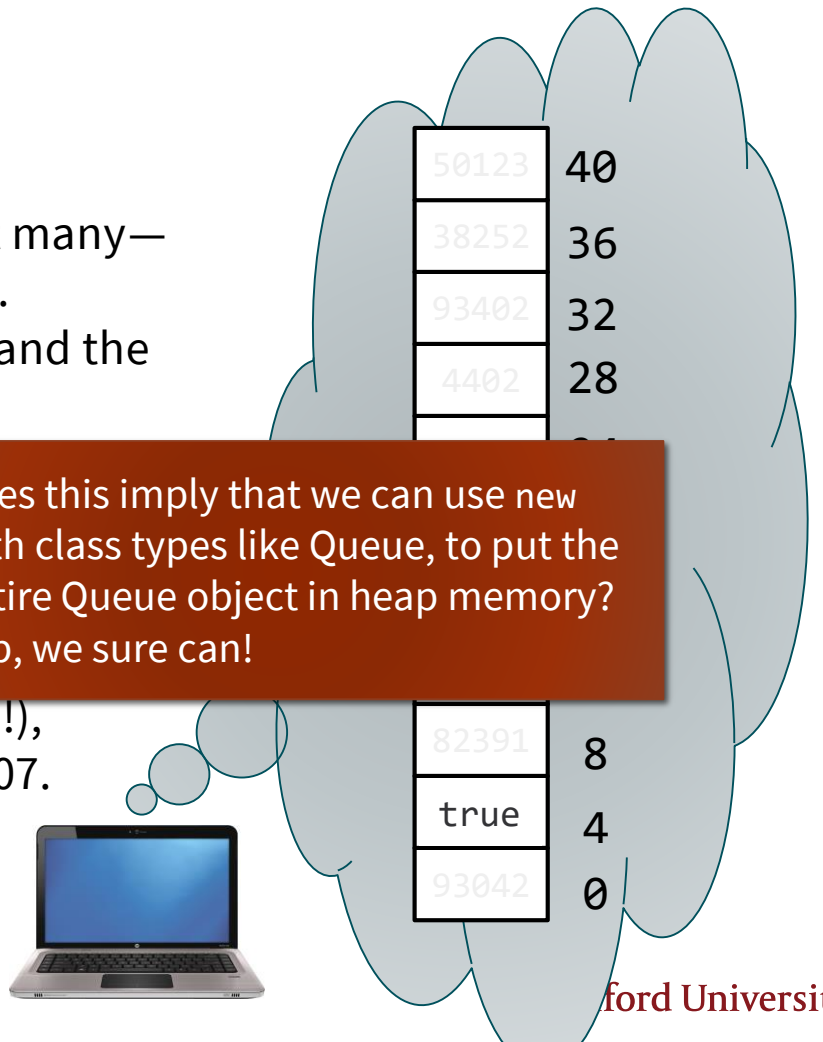


Memory addresses

"Pointer" isn't one type in C++ but many—
it **depends on what it points to**.

You can declare a pointer using * and the type pointed-to:

- `int*`
- `bool*`
- `string*`
- `double*`
- `Queue<GridLocation>*`
- `int**` ← Yes this is possible (!!), you'll see this in CS107.



Does this imply that we can use `new` with class types like Queue, to put the entire Queue object in heap memory? Yep, we sure can!

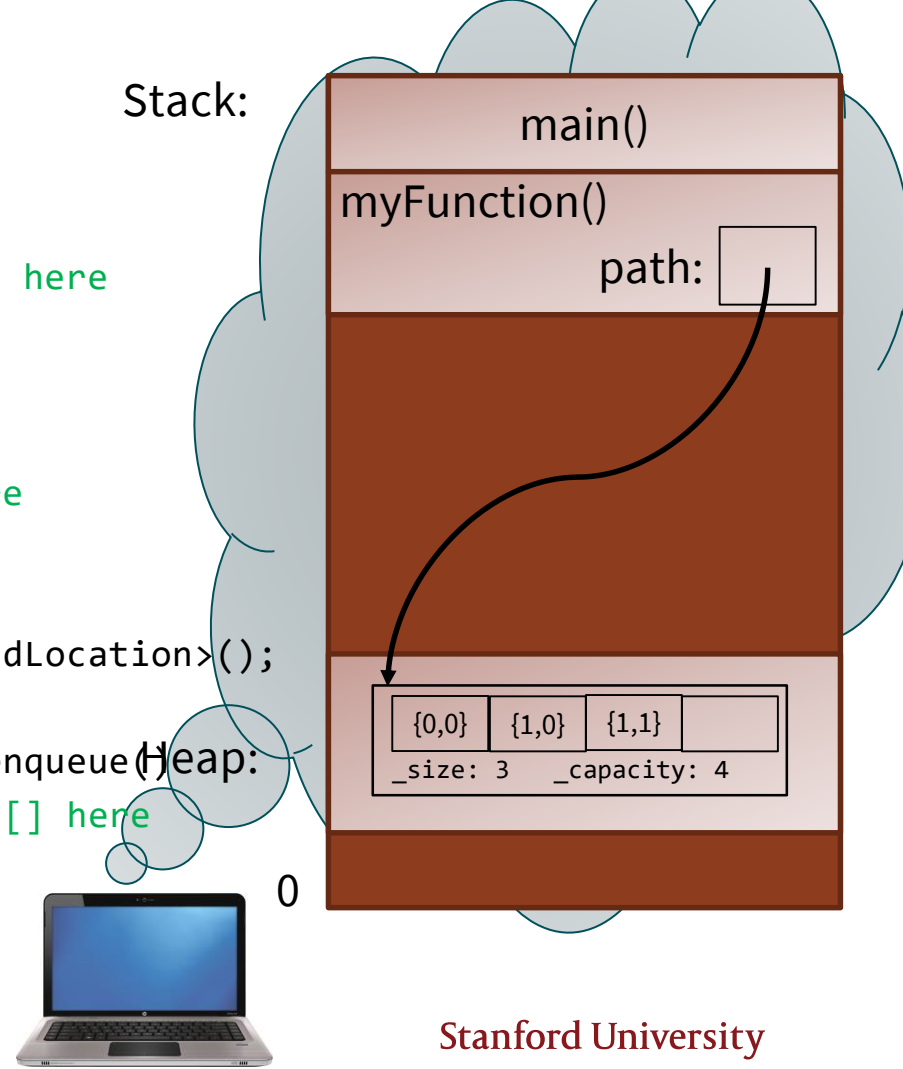
More on Dynamically-Allocated Memory

NEW AND DELETE FOR THINGS OTHER THAN ARRAYS

Dynamically-allocated object

```
// Array example
int* heapArr = new int[3];  // use [size] here
 …
heapArr[0]
stackArr[0]
delete [] heapArr;          // use [] here

// Object example
Queue<GridLocation>* path = new Queue<GridLocation>();
 …(enqueue a few things)…
path->enqueue(loc);   //instead of path.enqueue(loc)
delete path;                // don't use [] here
```



Introducing the C/C++ `struct`

LIKE A LIGHTWEIGHT CLASS

Like a lightweight class: C/C++ `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
  - A holdover from C, before the idea of objects (that combine variables and methods together) existed
- 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 to have to have "`artist_`" prefix on fields

How many times do we set and store the artist info?

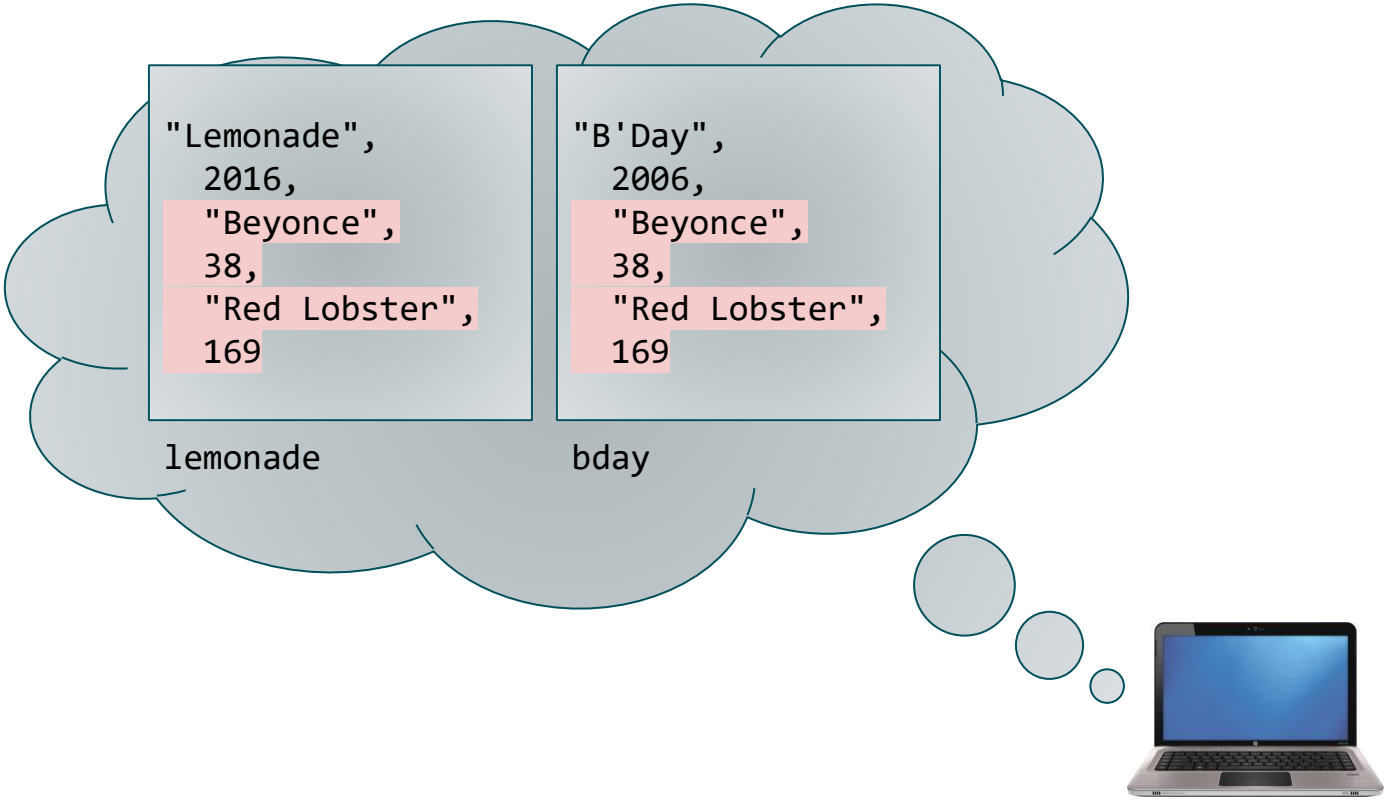
Album struct's design causes redundancy in code

```
void foo() {
    Album lemonade = {"Lemonade", 2016, "Beyonce", 38, "Red Lobster", 169};
    Album bday     = {"B'Day",    2006, "Beyonce", 38, "Red Lobster", 169};

    cout << lemonade.year << ", " << bday.year << endl; // 2016, 2006
}
```

- **Notice the redudant code to declare and initialize these two album variables,** `lemonade` **and** `bday`

It's redundantly stored, too



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?

Put a struct (Artist) in our struct (Album)

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

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

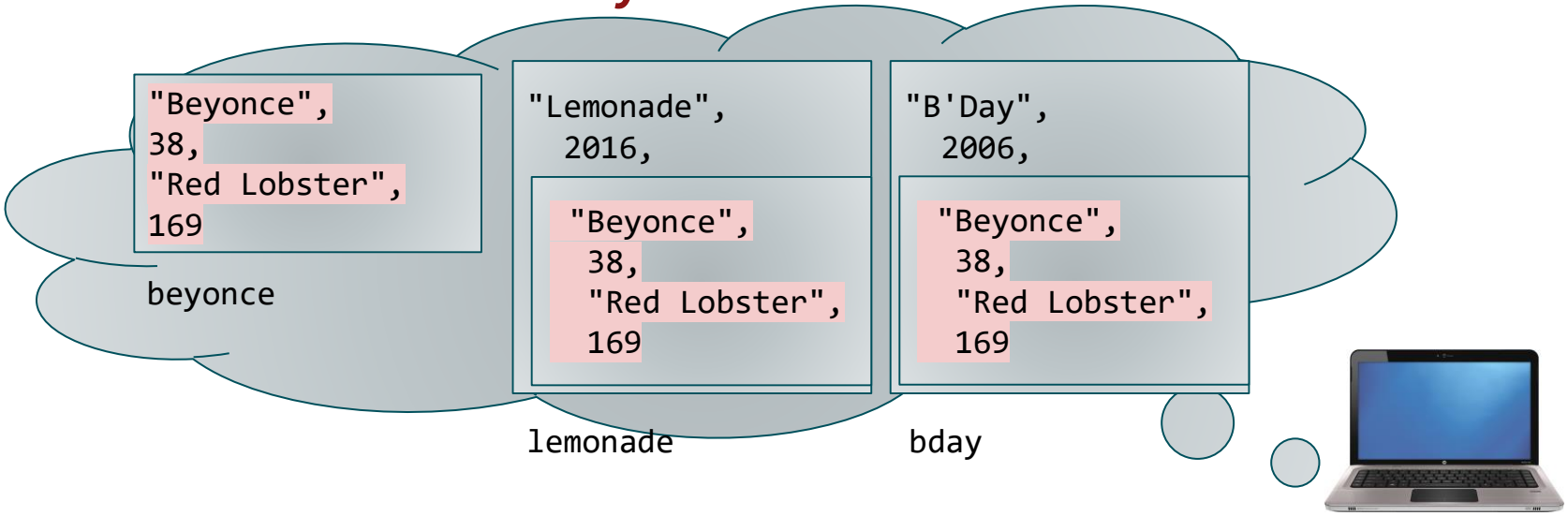
```
void foo() { //BEFORE
    Album lemonade = {"Lemonade", 2016, "Beyonce", 38, "Red Lobster", 169};
    Album bday     = {"B'Day",    2006, "Beyonce", 38, "Red Lobster", 169};

    cout << lemonade.year << ", " << bday.year << endl; // 2016, 2006
}
```

```
void foo() { //AFTER
    Artist beyonce = {"Beyonce", 38, "Red Lobster", 169};
    Album lemonade = {"Lemonade", 2016, beyonce};
    Album bday     = {"B'Day",    2006, beyonce};

    cout << lemonade.year << ", " << bday.year << endl; // 2016, 2006
}
```

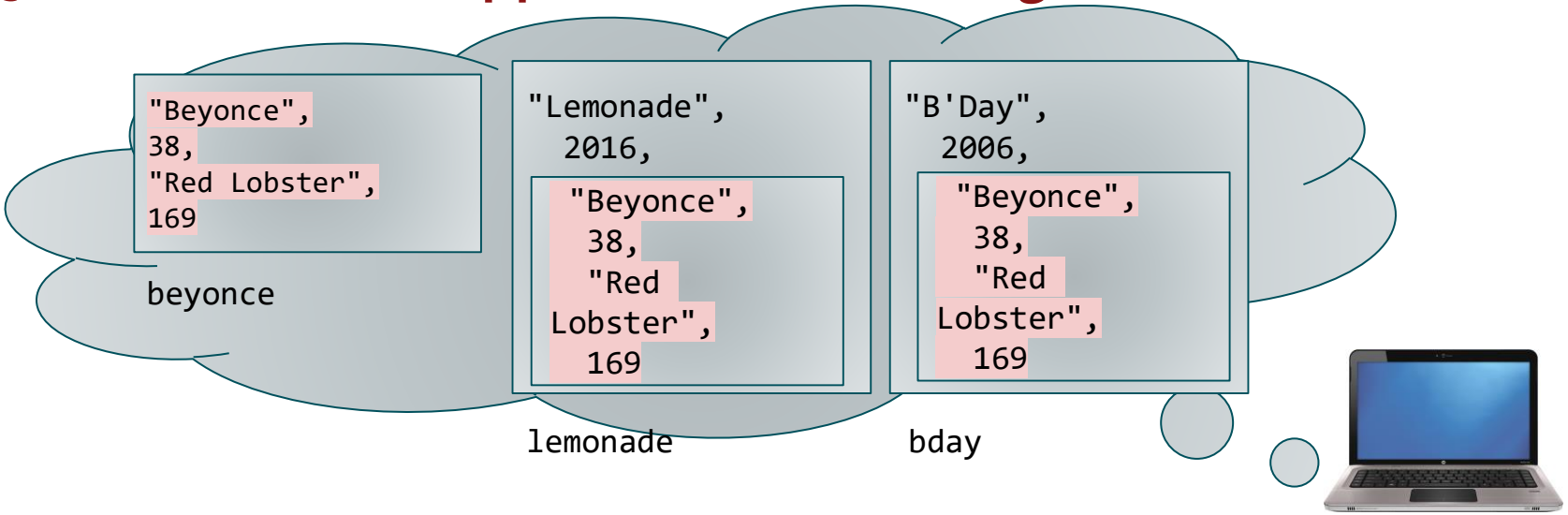
Still stored redundantly




```
void foo() { //This "AFTER" code is cleaner, but computer memory now store 3 copies!
    Artist beyonce = {"Beyonce", 38, "Red Lobster", 169};
    Album lemonade = {"Lemonade", 2016, beyonce};
    Album bday     = {"B'Day",    2006, beyonce};

    cout << lemonade.year << ", " << bday.year << endl; // 2016, 2006
}
```

QUIZ TIME: what happens when we change a value?

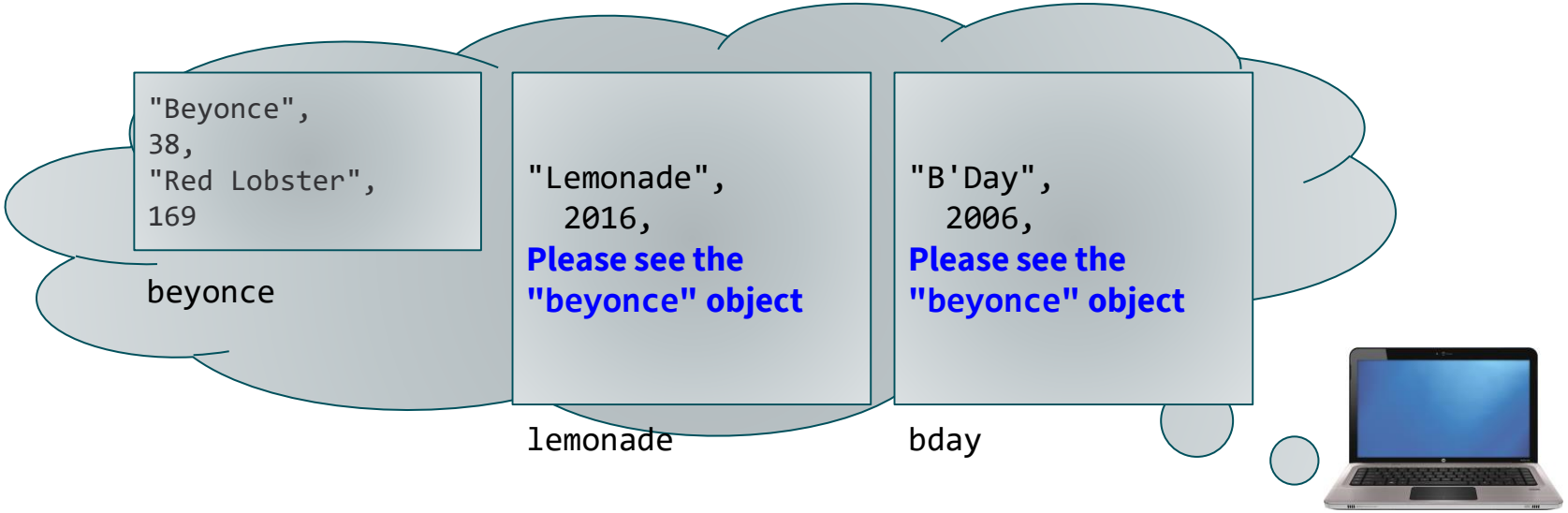



```
void foo() {
    Artist beyonce = {"Beyonce", 38, "Red Lobster", 169};
    Album lemonade = {"Lemonade", 2016, beyonce};
    Album bday     = {"B'Day",    2006, beyonce};
    beyonce.favorite_food = "Twix"; // New line of code
}
```

Question: what happens to the data in memory?

A. All 3 copies change to Twix
B. Only beyonce's copy changes
C. Only lemonade/bday's copeis change

Conceptually, what would we really like to happen?



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

How do we do this in C++?

...pointers!

Structs with pointers

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

Before pointers:

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

After pointers:

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

new and delete *with structs*

Example:

```
Artist* beyonce = new Artist;
beyonce->name = "Beyonce";
beyonce->age = 38;
beyonce->favorite_food = "Red Lobster";
beyonce->height = 169;

Album* lemonade = new Album;
album->title = "Lemonade";
album->year = 2016;
album->artist = beyonce;

beyonce->favorite_food = "Twix";
delete beyonce;
delete lemonade;
```