Programming Abstractions

CS106B

Cynthia Lee
Topics:

- Memory and Pointers
  - Quick review from our second lecture on classes, when we implemented ArrayStack
    - Arrays in C++
    - new/delete dynamic memory allocation
  - What is a pointer?
Arrays in C++

Like a Vector, but more primitive
Two kinds of arrays in C/C++

type* name = new type[length];

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

  Example: int* homeworkGrades = new int[7];

type name[length];

- **A statically allocated (stack-allocated) array**; can never be resized.
  - Memory does not need to be freed; will be automatically released.

  Example: int homeworkGrades[7];
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* 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

```c
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

```c
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;
}
```

Memory

main()

myFunction’s stack frame automatically released

Heap array manually released by delete []
What is a pointer?

More detail on that “pointer” type that we use for arrays
But first... 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?

```c
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?
void foo() {
    Album lemonade = {"Lemonade", 2016, "Beyonce", 38, "Red Lobster", 169};

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

- Notice the redundant code to declare and initialize these two album variables, lemonade and bday
It's redundantly stored, too

"Lemonade", 2016, "Beyonce", 38, "Red Lobster", 169


lemonade bday
How do we fix this?

```c
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)

```cpp
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};

    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
}
```
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
}

Still stored redundantly
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 copies 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

Before pointers:

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

After pointers:

```cpp
struct Album {
    string title;
    int year;
    Artist* artist;
};
```
new and delete *with structs*

Example:

```cpp
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;
```
Pointers

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

```c
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*

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