CS24 Week 3 Lecture 2

Kyle Dewey
Overview

• C++
  • Classes
  • Constructors
  • Destructors
• List ADT
  • Array Lists
  • Linked Lists
Note on Minor C++ Points

- Differences with `#include`, `namespaces`, and terminal I/O won’t be covered in class
- You are still expected to know these
- Slides online for last lecture have this content
Car Class Example
Rectangle Class Example
Creating Class Instances

• Can be made either on the stack or the heap
  • On the stack: `Car c(speed);`
  • On the heap: `Car* c = new Car(speed);`

• Both of these examples call the same constructor

• For the heap, can free with: `delete c;`
Constructors in C++
 Constructors

• C++ lets us define multiple constructors for a class

• Each can be used to make a class instance
Default Constructor
Default Constructor

• A constructor of special mention is the *nullary*, AKA *default* constructor

• Takes no arguments

• Used when creating an array of objects on the stack: `Car c[20]`;

• Why?
Default Constructor

- A constructor of special mention is the *nullary*, AKA *default* constructor
- Takes no arguments
- Used when creating an array of objects on the stack: `Car c[20];`
- Why? - how would you pass the arguments if it weren’t this way?
Default Constructor

- What happens?

```cpp
class Foo {
private:
    int x;
};

... Foo f;
```
Default Constructor

- All OK - the compiler generates a *default constructor* for you

```cpp
class Foo {
    private:
        int x;
};
...
Foo f;
```
Default Constructor

- What happens?

```cpp
class Foo {
    private:
        int x;
};

... Foo f; f.x;
```
Default Constructor

- Undefined - \( f.x \) can be set, but not accessed

```cpp
class Foo {
private:
    int x;
};
...
Foo f; f.x;
```
Default Constructor

- What happens?

```cpp
class Foo {
    public:
        Foo(int y);
    private:
        int x;
};
...
Foo f;
```
Default Constructor

• Compile-time error: compiler cannot generate a default constructor

```cpp
class Foo {
public:
    Foo(int y);
private:
    int x;
};
...
Foo f;
```
Copy Constructor
Copy Constructor

- Used in contexts where we need to copy an object
- Declarations with initialization
- Function calls

```cpp
Car(const Car& other);
```
Copy Constructor

• What happens?

class Foo {
    public:
        Foo(int y);
    private:
        int x;
};
...
Foo a(1);
Foo b = a; // copy constructor
Copy Constructor

- All ok - the compiler generates a default copy constructor that copies everything

```cpp
class Foo {
    public:
        Foo(int y);
    private:
        int x;
};
...
Foo a(1);
Foo b = a; // copy constructor
```
Default Copy Constructor

- Caveat: the copy performed is a shallow copy

Diagram:
- Before Copy
  - Object 1
  - memory
Default Copy Constructor

- Caveat: the copy performed is a shallow copy
Default Copy Constructor

- If you want a *deep copy*, you must do it yourself with your own copy constructor
Default Copy Constructor

• If you want a *deep copy*, you must do it yourself with your own copy constructor
Destructors
Destructors

• Optionally, you can define a destructor for a class: `Car::~Car() {}`

• Destructors are called during deallocation
  • When is this for something on the stack?
  • When is this for something on the heap?
Destructors

- Optionally, you can define a destructor for a class: `Car::~Car() {}`

- Destructors are called during deallocation
  - When is this for something on the stack?
    - Return from scope that introduced it
  - When is this for something on the heap?
    - When `delete` is called on it
Destructors

• Useful for objects which dynamically allocate memory internally

• Why?
Destructors

• Useful for objects which dynamically allocate memory internally

• Why? - Allows for memory to be deallocated in synchronization with the object being deallocated
Additional Use of `const`

- We’ve seen `const` already in two positions:

  ```c
  void foo(const char* const s) {
    s[0] = 'a'; // disallowed
    s = NULL; // disallowed
  }
  ```

  What is pointed to is constant | The pointer itself is constant
Additional Use of `const`

- We can also tag whole methods with `const`, indicating that they may not change any state of the class they are called on

- Great for *accessors*, as opposed to *mutators*

```cpp
class Foo {
    public:
        Foo(int a) { b = a; }
        void setValue(int a) { b = a; }
        int getValue() const { return b; }

    private:
        int b;
};
```
List ADT
Motivation

• We often work with a series of items
  • Addresses in a phone book, cards in a deck, etc.

• Arrays can be painful
  • Fixed size
  • Error-prone (e.g., index too large)

• Repeated similar operations
Idea: A “List” ADT

- Handles the storage of elements and the addition of elements
- Holds common operations (e.g., checking if an item is contained within)
- Can protect against out-of-bounds
A List ADT

• What should the List ADT have at the logical/abstract/interface level?
A List ADT

• What should the List ADT have at the logical/abstract/interface level?

• Basic examples: get item, add item, insert item at a position, remove item, get size

• Many, many more examples possible
Idealized List ADT

List emptyList();
int getSize();
int getInt(int position);
bool containsInt(int item);
void addInt(int item);
void addIntAtPosition(int item, int position);
void removeFirstInt(int item);
Implementing in C++

• Classes? Constructors? Methods?

• Which methods should be marked const?

List emptyList();
int getSize();
int getInt(int position);
bool containsInt(int item);
void addInt(int item);
void addIntAtPosition(int item, int position);
void removeFirstInt(int item);
Implementing in C++

- Classes? Constructors? Methods?
- Which methods should be marked const?

List emptyList(); // Constructor
int getSize() constexpr;
int getInt(int position) constexpr;
bool containsInt(int item) constexpr;
void addInt(int item);
void addIntAtPosition(int item, int position);
void removeFirstInt(int item);
Implementing in C++

- For now, let’s implement this via an array
- What other issues are present because of this design decision?
Implementing in C++

- For now, let’s implement this via an array
- What other issues are present because of this design decision?
  - Size of the array?
  - Accessing out-of-bounds element?
  - Adding an element in the middle?
- How might we handle each?
Implementation in C++
Array-Based List

- What sort of operations were hard because arrays were used?
Array-Based List

- What sort of operations were hard or awkward because arrays were used?
- Constructor needed an array size
- Adding an element at an arbitrary position required pushing elements to the right
- Removing an element required pushing elements to the left
Other Approaches

• How might we improve on these issues? (Fixed size, making arbitrary addition and removal easier)
Other Approaches

• How might we improve on these issues? (Fixed size, making arbitrary addition and removal easier)

• Wide variety of answers

• Approach we will take: linked lists
Fixed Size

• Observation: with arrays, we must allocate in blocks

• We must pre-allocate room, and expanding this room is obnoxious

• We would like to allocate as we go along, in a piecewise fashion
Piecewise Allocation

• How can we represent the list in a way that makes piecewise allocation possible? (Not just extending onto an array)
Piecewise Allocation

• How can we represent the list in a way that makes piecewise allocation possible? (Not just extending onto another array)

• Piecewise implies separate chunks that hold onto single elements

• How do we keep track of chunks?
Linked Lists

- Idea: have each chunk (called a node) keep track of both a list element and another chunk
- Need to keep track of only the head node

List: 0, 1, 2, 3
Node Representation

• What might a node look like in C/C++?
Node Representation

- What might a node look like in C?

```c
struct Node {
    int item;
    struct Node* next;
};
```
Node Representation

• What might a node look like in C++?

```cpp
class Node {
  public:
    Node(int i, Node* n);
    int getItem() const;
    void setItem(int i);
    Node* getNext() const;
    void setNext(Node* n);
  private:
    int item;
    Node* node;
};
```
C++ Implementation of Linked Lists