31251 – Data Structures and Algorithms

Week 2

Luke Mathieson / Marco Tomamichel

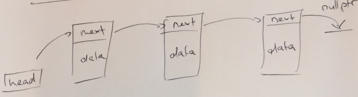
This Week's Fun!

- **Data Structures**
  - Linked Lists
  - Queues
  - Stacks
- Getting used to C++
  - Simple I/O
  - Classes and the importance of destructors
  - Exceptions
  - Compilation with multiple files

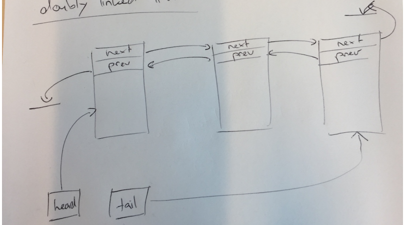
Linked Lists

- Simplest case: linked list
    - Every item of the list contains data and a pointer to the next item.
    - We store a pointer to the first item, the *head* of the list.
    - We can traverse the list easily in one direction following the pointers.
- A bit more advanced: doubly linked list
    - Every item of the list contains data and a pointer to the next and previous item.
    - We store a pointer to the first and last item, the *head* and *tail* of the list.
    - We can traverse the list in both directions.

linked list



doubly linked list



Queues

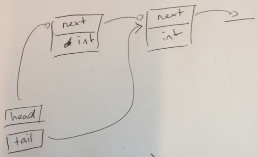
- The (basic) Queue is the basic FIFO (first-in-first-out) data structure.
- It keeps things in order (like a list), but...
- Things can only be added to the back, and
- Things can only be taken off the front.
- Normally has an unbounded capacity.

A Pure-ish Virtual Class for a Queue of `ints`

```
class intQueue {

public:

  virtual ~intQueue() {};
  virtual void enqueue(int n) = 0;
  virtual int dequeue() = 0;
  virtual int peek() = 0;

};
```

Implementing Queue with linked list

n items



enque : O(1)
deque : O(1)

without tail pointer

enque : O(n)

↑

because we need to traverse
the whole list!

Other types of Queue

- A Deque is a double-ended queue - you can add and remove at both ends!
  - This is really useful for implementing other data structures.
- A Priority Queue is a queue, but elements are inserted with a priority, and come out in priority order.

Stacks

- A Stack is like a queue, but it's a last-in-first-out (LIFO) data structure.
    - It's like a ... stack of things.
- You can add to the "top", and
- remove from the "top".

A Pure Virtual Class for a Stack of ints

```
class intStack {

public:

  virtual ~intStack() {};
  virtual void push() = 0;
  virtual int pop() = 0;
  virtual int peek() = 0;

};
```

Stacks and Queues

- Stacks and Queues are two of the most used data structures that “do” something.
- Buffers of all kinds are Queues (things go in, and get processed in order eventually).
- Stacks are built into the programming languages you’re using - they control how the program functions.
- You’ll see more applications of them as the course progress too.

This Week's Fun!

- Data Structures
  - Queues
  - Stacks
- **Getting used to C++**
  - Simple I/O
  - Classes and the importance of destructors
  - Exceptions
  - Compilation with multiple files

cin and cout

- As with everything else, C++ has a number of ways to handle basic text input and output.
- We'll cover the "standard" cin and cout.
  - These are the same idea as Java's System.in and System.out.
  - But they come with special operators: >> and <<.

cin and cout

- They are in the iostream library in the C++ Standard Library.
- Write to standard output with cout << [stuff to write] ;
- Read with cin >> [destination];
    - To read a whole line: getline(cin, [variable to read in to])
- cerr also exists for errors (equivalent to System.err in Java).
- And now a demo!

File I/O

- Uses the same abstractions as `cin` and `cout`, but with a little more fiddling.
  - Use the library `fstream`.
  - Create an `ofstream` for writing.
  - Create an `ifstream` for reading.
  - For details: `http://www.cplusplus.com/doc/tutorial/files/`

Classes and Destructors

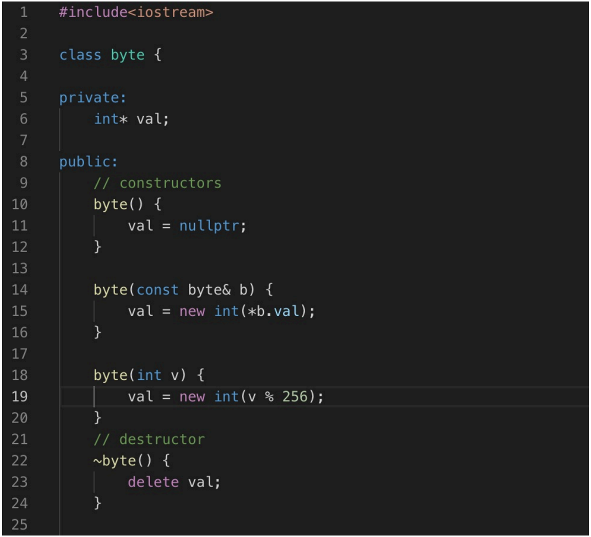
- You may have noticed a weirdly named function last week: `~intLinkedList()`.
- This is a *destructor*.
- This is a special method that's run when an object has the special `delete` operator called on it.
  - Syntax: `delete [pointer to thing to delete]`.
  - For arrays: `delete[] [array variable]`.

Why do we `delete` things?

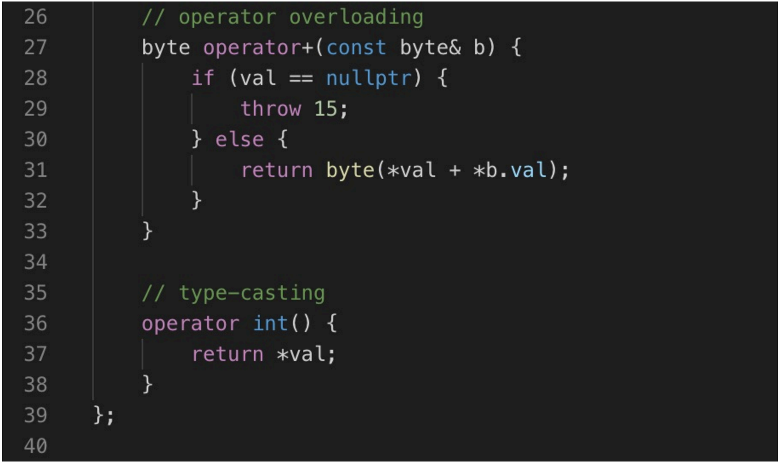
- `delete` is needed when we've created something with `new`.
- Otherwise the heap memory is not deallocated, and we have a memory leak.
- The programmer has to choose when to do this (so again, don't use `new` unless you mean it!).

Exceptions

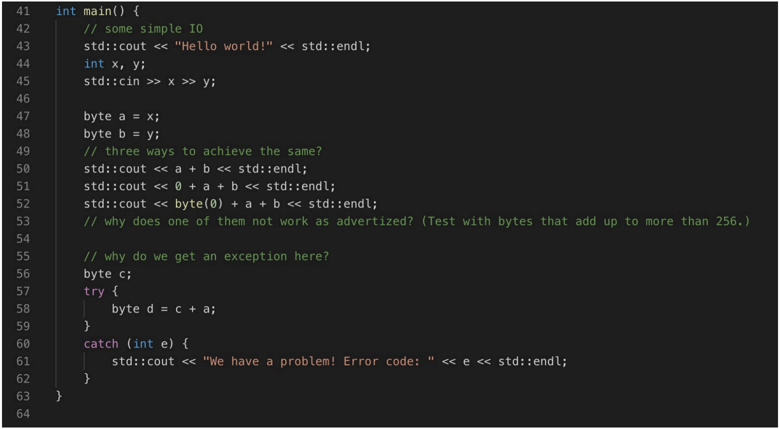
- C++ can `throw` exceptions, just like Java.
- It has `try ... catch([Exception Type] [parameter name]) ..` like Java.
- So why am I telling you this?
- Because C++ can throw *anything*. Demo!
- C++ does define a set of exceptions, defined in `<exception>`: `http://www.cplusplus.com/doc/tutorial/exceptions/`



```cpp
#include<iostream>

class byte {

private:
    int* val;

public:
    // constructors
    byte() {
        val = nullptr;
    }

    byte(const byte& b) {
        val = new int(*b.val);
    }

    byte(int v) {
        val = new int(v % 256);
    }
    // destructor
    ~byte() {
        delete val;
    }

```



```cpp
    // operator overloading
    byte operator+(const byte& b) {
        if (val == nullptr) {
            throw 15;
        } else {
            return byte(*val + *b.val);
        }
    }

    // type-casting
    operator int() {
        return *val;
    }
};

```



```cpp
int main() {
    // some simple IO
    std::cout << "Hello world!" << std::endl;
    int x, y;
    std::cin >> x >> y;

    byte a = x;
    byte b = y;
    // three ways to achieve the same?
    std::cout << a + b << std::endl;
    std::cout << 0 + a + b << std::endl;
    std::cout << byte(0) + a + b << std::endl;
    // why does one of them not work as advertized? (Test with bytes that add up to more than 256.)

    // why do we get an exception here?
    byte c;
    try {
        byte d = c + a;
    }
    catch (int e) {
        std::cout << "We have a problem! Error code: " << e << std::endl;
    }
}
```

Compiling with multiple files

Demonstrated in class (but a simple search will give lots of examples and even different, confusing ways to do it).