True story.

I wrote a paper, sent it off for review. Several months later:

"Nice paper, Mike, but could you please make a few editorial changes… On page 1, …..."
This is written in LaTeX, a document "markup language" that we researchers often use to write our papers in, rather than, say, MS-Word.

LaTeX looks a bit like HTML but much more complicated. Like, C++ complicated.

And it requires not just a single tool, but an entire software ecosystem to run. Which ages, and sometimes breaks.

"How hard can this be?"

• Goal: Edit paper source to make a few changes, create PDF thru LaTeX, & submit to publisher
  – But new Macbook runs new OS, and old install of LaTeX now broken
  – OK, let's re-install; oh no, package management system MacPorts no longer supported
  – Browse package management alternatives, decide to install homebrew
  – Read homebrew docs, lookup which packages I need for LaTeX plus any others I might want; check what packages I had on old laptop (who did I give my old laptop to?)
  – Download all source files for homebrew itself; install ruby; verify; compile; install homebrew package management system
  – Download the source packages for homebrew that I want, including LaTeX; compile; install; verify [This takes hours!]
  – OK, back in business now

Umm, is this the right room for CS138?

• Roughly, this convoluted example of a task list is a real-world example of a STACK:
  – As we start each new task, we find we have to pause and perform a sub-task first
    • Except for the last task, which we know how to perform completely, as it turns out
  – As we complete one task, we then resume the context of the "calling" task, and complete it!
    • ... and so on, until we're all done

• The "To do A, I have to do B; to do B, I have to do C" shaggy dog story is also called Yak Shaving
The original meme

[From an old Ren and Stimpy cartoon]

The C/C++ (Java, Python, etc) memory model

- Let's pause the execution of a program in the middle of a function call, say `main -> f -> g`
- The *run-time stack* manages the storage needs (local variables + parameters) for each pending function call
- The *heap* (aka "freestore") has the actual storage for all dynamically-allocated "objects" i.e., those that were created via a call to `new` or `malloc`
- Total storage is finite, tho you can ask the OS for more at run-time (it might say no)

The memory model
The memory model

- In C/C++, we typically **compile** the source code into the local OS / hardware-specific language
  - Also, the run time environment (RTE) may be wholly or partly embedded in the compiled code (static versus dynamic linking)

- In Java and C#, we compile source code into a **universal virtual machine** (VM) language (JVM/CLR)
  - The VM is then the run-time system, and must be implemented to run on various OSs / hardware platforms

- Some languages, like Scheme/Racket, Python, and PHP are (usually) **interpreted**
  - The RTE translates and then executes the source code on the fly

Creating linked structures in C++

- We'd like to create list-like structures in C++
  - We saw how to do this in C using C-structs and pointers in CS137
  - C++ structs are similar but also different too

- C++ structs are much more powerful than C structs
  - They are absolutely equivalent to C++ classes**
  - We will only lightly brush on the more powerful stuff for now

- The C++ Standard Library defines several useful containers (vector,list,map,...) which you should use "in real life"
  - We're going to "roll our own" for pedagogic reasons only

**except for default access rights of members

Creating struct/class instances

Instances of classes/structs are called objects; there are two ways to create them in C++:

1. Direct instantiation
   `structName s;`
   - Space is allocated on the **run-time stack**
     - But instances disappear at the end of their defining scope
     - This is a problem if we want to pass around linked structures
     - Use a period to select field (or method)

```
Coord c;
c.x = 10;
c.y = 20;
```

2. Dynamic instantiation (usually, through a pointer)
   `structName* sPtr = new structName;`
   - Space for the object is allocated on the heap; it persists until explicitly deleted by programmer
     - The ptr sPtr is allocated on the stack, tho.
   - Must remember to delete instances when no longer needed
   - Use an "arrow" (minus-greaterThan) to select field/method

```
Coord* p = new Coord;
p->x = 10;
p->y = 20;
... // play around
delete p;
```
```cpp
#include <string>
#include <iostream>
using namespace std;

struct Coord {
    int x, y;
}; // Need ";" at end of a struct def!

void print (Coord c) {
    cout << "x = " << c.x << " y = " << c.y << endl;
}

void printPtr (Coord* p) {
    cout << "x = " << p->x << " y = " << p->y << endl;
}

int main (int argc, char* argv[]) {
    Coord* p2; // ptr
    Coord a;
    a.x = 3;
    a.y = 5;
    p2 = p1; // ptr copy
    Coord b = a; // object copy
    print (*p1);
    p2->x = 42;
    print (*p2);
    Coord* p3 = &a; // ptr
    printPtr (p1);
    print (a);
    p3->y = 83;
    print (a);
    delete p1; // good!
    delete p2; // error
    delete p3; // error
    return 0;
}
```

See anything wrong here?

```cpp`
See diagram
```
```cpp
#include<string>
#include<iostream>
using namespace std;

struct Coord {
    int x, y;
};  // Need ";" at end of struct!

void print (Coord c) {
    cout << "x = " << c.x << " y = " << c.y << endl;
}

void printPtr (Coord* p) {
    cout << "x = " << p->x << " y = " << p->y << endl;
}

int main (int argc,
          char* argv[]) {
    Coord a;
    a.x = 3;
    a.y = 5;
    Coord b = a;
    printPtr (&a);
    printPtr (&b);
    a.x = 17;
    printPtr (&a);
    printPtr (&b);
    Coord* p1;
    p1 = new Coord;
    p1->x = 4;
    p1->y = 12;
    print (*p1);  // copy object
    printPtr (p1); // copy ptr
    delete p1;
    return 0;
}
```

(Review) Static and dynamic memory allocation

- Memory for your variables come from either:
  1. **Run-time stack** handles automated allocation of storage for params and local variables as procedures are called
     - This storage disappears automatically at the end of the proc call:
       Balloon b;
  2. **Freestore** (aka the heap) handles all programmatic requests for storage via `new` (for C++ objects/structs, but also `int` etc.) and `malloc` (and relatives, in C) and
     - User must return this storage when done via `free/delete`
     - Pointer points to the stack; the object it points to is on the heap
       Balloon *p = new Balloon;
C++ pointers

- A pointer is actually a (strange) number
  - It's the (numeric) address of the thing that you are pointing to, which may reside on the heap or on the stack
  - ... and you really don't need to know what that value is, numerically
  - There is such a thing as "ptr arithmetic", as you saw in CS137

- There is a special C++11 value called `nullptr`
  - If a ptr has the value `nullptr` this means, "I'm not pointing to anything currently"
  - If you just declare a ptr, it will contain random garbage as its initial value
    - So, after declaring a ptr, it's usually best to set it to point to something or set it to `nullptr` right away
  - [Previously, C++03 used the NULL macro for this purpose]

### `nullptr` vs. `NULL`

- In the old days of C, giving the (integer) value zero to a ptr was used to indicate that the ptr wasn't pointing to anything

- Later, the macro `NULL` was created as somewhat safer and more intuitive way to express the same idea
  - The (old) C++ standards requires `NULL` to evaluate to zero; mostly this worked OK but there were some (corner) situations in which it didn't
  - ... because it could be treated as an integer too

- C++11 defines the special constant `nullptr`
  - It's a `keyword` in the C++11 language (can't reuse it as a variable name)
  - It can be treated as any kind of ptr or as a `bool`, but `not` as an integer

C++ pointers

- ... can point to anything:
  - `struct/class` instances (aka `objects`)
  - strings (which are objects)
  - `ints, doubles, bools, ...`
  - even other `ptrs`

- We will use them mostly to point to `struct/class` instances, tho
  - For most of our examples, we'll assume we're managing a "list" of strings

### `nullptr` vs. `NULL`

- `NULL` works for any standard version of C++

- `nullptr` requires that the compiler support C++11
  - To get it to work with g++, you `may` have to do this
    `g++ -std=c++0x file.cc` For g++ v4.6.X
  - or
    `g++ -std=c++11 file.cc` For g++ v4.7 or later

- For our purposes, you can use either `NULL` or `nullptr`; it doesn't really matter
  - Eventually, one day, you will use `nullptr` 😊
(Review) Static and dynamic memory allocation

- Memory for your variables come from either:
  1. **Run-time stack** handles automated allocation of storage for params and local variables as procedures are called
     - This storage disappears automatically at the end of the proc call:
       ```
       Balloon b;
       ```
  2. **Freestore** (aka the heap) handles all programmatic requests for storage via `new` (for C++ objects/structs, but also `ints etc.) and `malloc` (and relatives, in C) and
     - User must return this storage when done via `free`/`delete`
     - Pointer `p` is on the stack; the object it points to is on the heap
       ```
       Balloon *p = new Balloon;
       ```

```cpp
#include <iostream>
#include <string>
#include <cassert>
using namespace std;

struct Node {
    int value;
    Node* next;
};

int main (int argc, char* argv[]) {
    Node stackNode;
    stackNode.value = 99;
    stackNode.next = nullptr;
    Node* pHeapNode = new Node;
    pHeapNode->value = 15;
    pHeapNode->next = &stackNode;
    cout << pHeapNode->value << " " << stackNode.value << endl;
    cout << pHeapNode << endl;
    cout << pHeapNode->next << endl;
    cout << &stackNode << endl;
}
```

```cpp
#include <iostream>
#include <string>
using namespace std;

void swap1 (int x, int y){
    const int temp = x;
    x = y;
    y = temp;
    cout << x << y <<endl;
}

int main (int argc, char* argv[]) {
    int x = 5;
    int y = 37;
    swap1 (x, y);
    cout << x << y << endl;
}
```

[See diagram]
```cpp
#include <iostream>
#include <string>
using namespace std;

void swap1 (int x, int y){
    const int temp = x;
    x = y;
    y = temp;
    cout << x << y << endl;
}

void swap2 (int* px, int* py){
    const int temp = *px;
    *px = *py;
    *py = temp;
    cout << *px << *py << endl;
}

int main (int argc, char* argv[])
    int x = 5;
    int y = 37;
    swap1 (x, y);
    cout << x << y << endl;
    swap2 (x, y);
    cout << x << y << endl;
    return 0;
```
```cpp
#include <iostream>
#include <string>
using namespace std;

struct Node {
    string val;
    Node* next;
};

int main (int argc,
          char* argv[]) {
    Node* p = new Node;
    p->val = "first";
    p->next = nullptr;
    Node* q, r;
    // Note: r NOT a ptr
    r.val = "flurble";
    Node* s;
    s = new Node;
    s->val = "third";
    s->next = q;
    Node* temp = s;
    while (nullptr != temp) {
        cout << temp->val << endl;
        temp = temp->next;
    }
    delete p; // Clean up!
    delete q;
    delete s;
    // Don't delete r, tho!
}
```

---

### Introducing the linked list

- A **linked list** is a **data structure** that (typically) uses dynamically allocated elements called **nodes** to implement a kind of sequence.

- Each node has
  - a "value", which can be simple like a **string** or contain several parts
  - a **link** field, which is a pointer to the next **Node** in the list
    - If there is no next element, use **nullptr** as the value

- There is usually a special variable of type **Node** that points to the "first" element of the list
  - There are many variations of the linked list, as we’ll see
    e.g., each node may have multiple pointers to other nodes

- The linked list (in several variations) will be our main workhorse (w. variations) in implementing our ADTs
  - Note that we don't really need lots of standalone pointers `p`, `q`, `s`, etc. to create a linked list
    - If we want to create a list, all we need is a ptr to the first element, then let each node's `next` variable point to the next element
    - We’ll use the convention that the last element's `next` is the `nullptr` (then we can tell when we're at the end!)
Introducing ADTs

• An abstract data type (ADT) is a mathematical structure that has well-defined and widely recognizable behaviour
  — It contains data that may be accessed only in a prescribed manner, by a set of named operations
  — Each operation has
    • A signature, which describes the params + the type of the returned value
    • A pre-condition (logic stmt) that specifies what is assumed to be true before the operation may be applied
    • A post-condition (logic stmt) that describes the value / effect of the operation
  — The formality/rigour of the descriptions vary 😊

Some examples of ADTs

• A vector / sequence is an ordered data container that allows random access to individual elements, but usually allows adds/deletes only at the end

• A stack is an ordered data container with a LIFO (last in => first out) policy on inserts/deletes
  — But no random access to arbitrary elements

• A queue is an ... FIFO ...
  — Also, no random access to arbitrary elements

Some examples of ADTs

• A set is an unordered data container that can contain a given element at most once
  — A multi-set is ... zero, one, or more times

• A dictionary / map is an unordered data container of pairs
  — If (a,b) and (a,c) are in M, then b==c ***
  — Can do lookup based on the first ("key") element
    • M[a] returns b iff (a,b) in M
  — A multi-map is like a map, except that *** is not assumed

• The C++ Standard Library has implementations of these and many more besides! (and so does Java, C#, ...)
  — And Boost.org provides even more cutting-edge C++ libraries

ADTs are abstract

• There may be multiple ways to implement a given ADT, but the abstract meaning is the same regardless
  — That is, the abstract specification of, e.g., a stack, is the same
  — This point is important!

• Often, a programming language will supply means to create a hard interface around the ADT
  — An interface enforces a limited access of clients to the internal details, which the clients should not be touching directly
e.g., interface of Stack supports push and pop operations, but not at
  — C++/Java/C# provide strong language-level support for building interfaces via class definitions (C not so much)
What kind of data do ADTs hold?

- Many, but not all, ADTs are primarily data containers
  - They keep a collection of data of a particular kind organized in some interesting way
    e.g., stack, queue, tree, vector, ...

- Data containers can contain just about any kind of data!
  - For now, we'll assume that elements are strings, cos they're more fun to play with
  - When we get into OOP, we will show you how to define generic containers that can hold any kind of element kind

Style of ADTs

- Initially, we are going to use a functional programming style to create our ADTs in C-ish C++
  - This is clean mathematically, but sometimes inefficient if done naively due to needless parameter copying
  - Then, we'll use a more C-like procedural model using the C++ feature of reference parameters
  - Finally, we'll implement them in an object-oriented (OO) style

The general format of an ADT operation in this style is:
```
newADT operationName (oldADT, otherParams)
```
i.e., pass in the ADT as a param, get a new one as a result for mutator ops

The stack ADT

- A stack is an ordered container of data that enforces a LIFO policy on adds/removes
  - Permitted operations: push, pop, peek, isEmpty
  - ... plus a "constructor" (ctor), which we'll call initStack for now

- Nits:
  - Sometimes, pop and peek are combined into one operation that performs both ops (and is called pop)
  - Sometimes, peek is called top

The stack ADT: A little more formally

- `initStack` (no args) creates + returns a (new) empty stack to the caller

- `push` takes stack `s` and an element `e_{new}` and returns a new stack that's identical to `s` except with the new element on "top": `e_{new} e_2 ... e_N`

- `pop` takes a stack `s` and returns a new stack that's identical to `s` except with the top element removed: `e_2 ... e_N`

- `peek` takes a stack and returns (but does not remove) the first elt: `e_1`

- `isEmpty` takes a stack and returns true or false, depending if the stack is empty: i.e., does $N=0$?

  [Assume `s` is a stack whose current sequence is $e_1 ... e_N$ and `e_{new}` is the new element]
Pre- and post-conditions

- These are statements of logic that involve the parameters, the return value, and (maybe) the global program state.

Pre-condition
- Is there anything we need (assume) to be true about the parameters (+ maybe the global state) before using this operation?
- Want it to be as logically weak as possible, while still saying what we need.

Post-condition
- What is the relationship between the input params + (eventual) return value (+ maybe the global state)?
  i.e., What does this operation actually do?
- Want it to be as logically strong as possible without saying too much.

The stack ADT: Even more formally

pop : stack -> stack
Pre:
Post:

peek : stack -> element
Pre:
Post:

nuke : stack -> stack
Pre:
Post:

[Assume s is a stack whose current sequence is e₁ ⋯ eₙ and eᵱₑₙ is the "new" element, if any]

The stack ADT: Even more formally

initStack :-> stack
Pre:
Post:

isEmpty : stack -> boolean
Pre:
Post:

push: stack X element -> stack
Pre:
Post:

[Assume s is a stack whose current sequence is e₁ ⋯ eₙ and eᵱₑₙ is the "new" element, if any]

// This is roughly what we want, eventually ...
// It make take a while to get there ... 
// ... but how should we define the Stack type itself?

Stack initStack () {...}
Stack push (Stack s, string val) {...}
bool isEmpty (Stack s) {...}
Stack pop (Stack s) {...}
string peek (Stack s) {...}

int main (int argc, char* argv[]) {
    Stack s;
    s = initStack ();
    s = push (s, "alice");
    s = push (s, "bob");
    cout << peek (s) << endl;
    s = pop (s);
    ...
}
Stack interface vs. implementation

• This is the interface for a stack that will be the same (ideally) for any reasonable implementation in a C-like style

• First, we’ll look at an implementation that uses a linked list
  – Then we will use a vector-based approach
  – But the abstract interface should be the same!

• Again, for now we assume an element type of string
  – Generic stacks (coming later) will take any kind of element, like a vector does

Implementing a stack as a linked list

Consider:

```cpp
void push (Node* first, string val) {
  Node* newNode = new Node;
  newNode->val = val;
  newNode->next = first;
  // now what?
  first = newNode;  // ... but ...
}
```

• First interesting design question:
  – What data type (parameter/return type) should we use to model a stack that is implemented as a linked list?

• Recall our linked list from the other day:
  – We don’t need a ptr variable on the stack for each element; instead we keep a ptr to only the first element, and then every Node instance (on the heap) has a ptr to the next element
  – So we just have to keep track of a single:
  – So the data type that best models a stack with this approach is:

Some possible approaches that work

1. Use C++ reference parameters
   – A good idea, but ... we’ll do this later

2. The object-oriented approach:
   – Define a class with methods push, pop, etc.
   – Also a good idea, but we’ll do this later too

3. The functional approach:
   – Return the new value of first as the value of push
```cpp
Node* push (Node* first, string val)
```
Implementing a stack as a linked list

```cpp
Node* push (Node* first, string val) {
    Node* newNode = new Node;
    newNode->val = val;
    newNode->next = first;
    // now what? This!
    return newNode;       // no buts here!
}

// … in the main program
Node* s = nullptr;  // new empty stack
// pass a stack in, get a new stack back
s = push (s, "alice");
s = push (s, "bob");
```

Implementing a stack as a linked list

- Now let's implement `pop`, `peek`, and `isEmpty`,
- … and the "constructor" operation `initStack`
- ... and an "extra" operation `nuke`
  - ... that cleans up any mess we made when we're done with the stack, like a destructor in the OO world
  - i.e., delete the heap-based Nodes we were using

Lecture 6

CS138 W17
Hiding the implementation ... a bit

- Hmm, we are representing lists by using a pointer to the first element
  - It's kind of ugly for clients to have to say Node* when they are thinking Stack
  - Also it's ugly for them to have to know the initialization details

- So let's do two things:
  1. Add typedef Node* Stack; at the beginning of the program
  2. Replace Node* by Stack in all of the param lists

```cpp
#include<string>
#include<iostream>
#include<cassert>
using namespace std;

struct Node {
    string val;
    Node* next;
};

typedef Node* Stack;

Stack initStack () {
    return nullptr;
}

bool isEmpty (Stack s) {
    return nullptr == s;
}

Stack push (Stack s, string val) {
    Node* newNode = new Node;
    newNode->val = val;
    newNode->next = s;
    return newNode;
}

string peek (Stack s) {
    assert (!isEmpty(s));
    return s->val;
}

Stack pop (Stack s) {
    assert (!isEmpty(s));
    Node* newNode = s->next;
    delete s;
    return newNode;
}

Stack nuke (Stack s) {
    while (!isEmpty(s)) {
        s = pop(s);
    }
    return s;
}

int main (int argc, char* argv[]) {
    Stack s = initStack();
    s = push (s, "alice");
    s = push (s, "bob");
    s = push (s, "carol");
    cout << peek(s) << endl;
    s = pop(s);
    cout << peek(s) << endl;
    s = nuke(s);
}
```
(Review) ADTs and the stack

- ADTs have an *explicit interface/API* to separate out the high-level functionality that *clients* should see from the low-level details that the *implementer* must be deal with
  - *Implementer*: writes the "dirty" code that *implements* the low-level details of the API functions (*push*, *pop*, etc.) using pointers, nodes, library data structures like *vector*, etc.
  - *Client*: writes code "elsewhere" in the system (e.g., in *main*) that *uses* the high-level API functions of the interface
    - Clients should have to understand only the high-level meaning, not the dirty details

(Review) ADTs and the stack

- An ADT is a abstract data type, usually a "data container", that defines a set of operations for manipulating its elements
  - You (the client) should use *only* the operations for accessing the elements; don't bang on the data structures/ptrs/nodes directly!
  - The meaning of an ADT is implicitly defined by the meaning of its operations
    - e.g., a *stack* is a LIFO data container, where *push* adds elements onto the top and *pop* removes them, ...
  - The meaning of each operation can be specified by a *signature*, *pre-condition*, and *post-condition*
    - e.g., *pop* takes an existing stack and returns a new stack; the existing stack must not be empty, and the new stack will be the same as the existing one but with the "top" element removed

```
#include<string>
#include<iostream>
#include<cassert>
using namespace std;  
struct Node {
    string val;
    Node* next;
};
// This typedef will need to change if we // use a vector instead of a linked list typedef Node* Stack;
Stack initStack () {
    return nullptr;
}
bool isEmpty (Stack s) {
    return nullptr == s;
}
```

I don't know what a new, empty stack looks like on the inside, I just know I get one when I call *initStack*.  

(We've seen this code already.)

Client developer
Stack push (Stack s, string val) {
    Node* newNode = new Node;
    newNode->val = val;
    newNode->next = s;
    return newNode;
}

string peek (Stack s) {
    assert (!isEmpty(s));
    return s->val;
}

Stack pop (Stack s) {
    assert (!isEmpty(s));
    Node* newNode = s->next;
    delete s;
    return newNode;
}

Stack nuke (Stack s) {
    while (!isEmpty(s)) {
        s = pop(s);
    }
    return s;
}

int main (int argc, char* argv[]) {
    Stack s = initStack();
    s = push (s, "alice");
    s = push (s, "bob");
    s = push (s, "carol");
    cout << peek(s) << endl;
    s = pop(s);
    cout << peek(s) << endl;
    s = nuke(s);
    return 0;
}

(We've seen this code already.)

• It's possible to have multiple distinct stacks at the same time!

int main (int argc, char* argv[]) {
    Stack s1 = initStack();
    Stack s2 = initStack();
    s1 = push (s1, "alice");
    s1 = push (s1, "bob");
    s1 = push (s1, "carol");
    s2 = push (s2, "first");
    s2 = push (s2, "second");
    // Memory leak! Oh no!
    s1 = s2;
    return 0;
}

(We've seen this code already.)

• ... but you don't get "copy semantics" with this approach
Why *don't* we get copy semantics?

- The C-like programming model provides only weak support for programmer-defined abstractions
  - In this design, we’re just renaming a pointer type with a fancy name; C doesn’t give us the power of real OO
    e.g., defining how new objects are created, (re)defining what assignment means
  - We need C++-style structs/classes to do the job properly (more on this later):

```c
struct Coord {
  int x, y;
};
Coord c;
c.x = 37;
c.y = 5;
// copy semantics here!
Coord d = c;
```

Stack interface vs. implementation

- Take a good look at the interface of the various stack operations, and ignore the code for a minute
  - It’s possible to understand how to use a stack by simply studying the interface (if there’s enough detail specified by comments)
  - The actual details of the implementation don’t matter to the client much as long as they do the job correctly
  - We will see how we might implement a stack differently while still using the same interface
  - Later we will use an OO style to change the syntax slightly but preserving the ideas

- So on to the vector-based approach!
  - And, yes, I will talk about efficiency and parameter copying afterwards

```
// Exactly the same main program as w. linked list
int main (int argc, char* argv[]) {
  // Note that we can have two distinct
  // stacks at the same time!
  Stack s1;  // ... *except* that no initStack is required
  Stack s2;
  s1 = push (s1, "alpaca");
  s1 = push (s1, "beaver");
  s1 = push (s1, "cat");
  s1 = push (s1, "dog");
  s2 = push (s2, "one");
  s2 = push (s2, "two");
  cout << peek(s1) << endl;
  cout << peek(s2) << endl;
  s1 = pop(s1);
  cout << peek(s1) << endl;  // and we punted on nuke
}
```

Complexity of stack operations

- Linked list implementation
  - push, pop, peek, nuke

- Vector implementation
  - push, pop, peek, nuke
SE teachable moment: Adapters

• You might say to yourself:
  – *Hey, the vector class does the job for me. Why don’t I just use that directly in my main program instead of bothering with defining a special interface and code that basically just redirects function calls*

• The answer is that when you design an ADT for use by others, the interface is the most important part!

An adapter

• The stack-implemented-by-a-vector is an example of what’s called an *adapter*

• The stack ADT has a small, well-understood API: push, pop, ...
  – But `vector` has lots of other functionality stacks really don’t need! e.g., random access to all elements via `at()`, `begin`, `end`, `capacity`

• Designing an interface that provides *exactly* what clients need — and no more!! — prevents the client from depending on "extra" features they shouldn't, and also protects them from possible future changes in the underlying implementation

// Using a vector directly instead of a Stack adapter class
int main (int argc, char* argv[])
{
  vector<string> s1;
  s1.push_back("alpha");
  s1.push_back("beta");
  s1.push_back("gamma");
  vector<string> s2;
  string v = s1.back();
  cout << v << endl;
  s1.pop_back();
  cout << s1.back() << endl;
  s2.push_back(v);
  cout << s2.back() << endl;
  s2.pop_back();
  cout << s2.back() << endl; // Assertion failure
}

An adapter

• If we discover a more efficient way of implementing the stack interface, clients don’t need to change their code as long as the interface doesn’t change too e.g., suppose we found that the vector class ran slowly on certain kinds of machines or that there was a licensing problem with the code

• This sounds like "B&D design", but it’s perhaps the single most important sw engineering design principle there is:
  – *Information hiding:* Separate intf from impl. Hide impl details. Have clients depend only on well-designed, unlikely-to-change interfaces.
  – The intf should be much less likely to change than the impl details!

• The adapter is an example of a well known *design pattern*
As the implementer, should I use a linked list or a vector or ...? Why?

- Well that’s a good question; in fact, there already is a generic `Stack` class in the C++ Standard Library!
  - It’s an adaptor based on a `deque` (which is similar to a `vector`), and in a style very similar to what we did (but without the egregious copying; more later), so really we should choose the pre-existing library `Stack” in real life”

- If there were no such standard library class, then I would probably choose a `vector`-based implementation (using reference parameters!!) over the homebrew linked list, as
  - The STL `vector` class is in wide use and does the job completely
  - It’s likely to be simpler to implement (ptrs + linked structures => bugs!)
  - It’s about the same efficiency (assuming we use reference parameters)

One more efficiency issue

- The naïve functional-style vector-based approach we have used does a LOT of copying of vectors!
  - It is **not at all realistic** for efficiency, tho it is mathematically OK
  - A realistic implementation in a functional style sometimes require tricks to make things efficient, but we won’t go there
  - Instead, we will (later) show you how to do this in a state-based OO style that is fine, efficiency-wise
  - The linked list approach we used does not have this problem

As the implementer, should I use a linked list or a vector or ...? Why?

- The general point, tho, is that you don’t — and shouldn’t — need to understand the low-level details of *how* something is implemented in order to use it effectively
  - And implementers of library classes will usually try to hide their non-essential impl details from you; and you should be **grateful** for this information hiding
  - And when you design classes for use within a larger software system, think about what are the essential, public parts, and which should be hidden away from clients as private

Reference parameters

- C/Java support "call-by-value" (only) for parameter passing
  - Copy value of parameter onto call `stack frame` (aka `activation record`)
    - This has real overhead cost if the parameter is a “big” entity, like an object that contains many sub-objects
    - Only the return value (if any) is copied back
    - Can change parameter values inside procedure, but changes do not propagate back to the calling environment
    - Can use ptr parameters to "cheat" (it’s a common C practice, but it’s complicated and error prone; we can avoid this in C++)
      - Changes to ptr don’t propagate back, but changes to values pointed to do!
      - Can use ptrs to ptrs to change ptrs!

- Wouldn’t it be nice to allow changes to parameters to propagate back to the calling environment, if/when we want?
Reference parameters

- C++ (and C#) also supports the idea of reference parameters
  - Put an ampersand after the parameter type in a function declaration
    ```cpp
    void swap (int& x, int& y) {...}
    void push (Stack& first) {...}
    ```
  - The parameter is not copied onto the call stack; instead a reference back to the variable in the calling context is made (like a ptr)
  - Any changes you make in procedure will propagate back to the caller
    - The reference parameter is just another name (i.e., an alias) for the variable in the calling context
    - This approach to parameter passing is called call-by-reference
  - The reference parameter acts kinda like a ptr, but you use it "normally" as a variable of that type, not by dereferencing with *
  - Can have a ref or const ref return type ... but wait for CS247
  - Ref params are the norm in C++; get used to them

```cpp
#include <iostream>
#include <string>
#include <cassert>
using namespace std;

void swap1 (int x, int y){
    const int temp = x;
    x = y;
    y = temp;
    cout << x << y <<endl;
}

void swap2 (int* px, int* py){
    const int temp = *px;
    *px = *py;
    *py = temp;
    cout << *px<< *py <<endl;
}

void swap3 (int& x, int& y){
    const int temp = x;
    x = y;
    y = temp;
}

int main (int argc, char* argv[]){
    const int x = 5;
    const int y = 37;
    swap1 (x, y);
    cout << x << y << endl;
    swap2 (&x, &y);
    cout << x << y << endl;
    swap3 (x, y);
    cout << x << y << endl;
}
```

const& parameters

- A const reference parameter is a reference parameter that you aren't allowed to change inside the procedure
  - The compiler will prevent you from doing so!
- Semantically, it's almost the same as a value parameter
  - It is legal (but mostly useless) to change a value parameter

```cpp
string peek (Stack s) {
    assert (!isEmpty(s));
    return s.back();
}

// This is more space efficient
string peek (const Stack& s) {
    assert (!isEmpty(s));
    return s.back();
}
```

```cpp
string peek (Stack s) {
    assert (!isEmpty(s));
    return s.back();
}

// This is more space efficient
string peek (const Stack& s) {
    assert (!isEmpty(s));
    return s.back();
}
```

- const ref params are also more space efficient than value params, as you don't copy the whole object onto the run time stack, just a reference to it
  - This is the preferred way in C++ to declare parameters that you don't intend to change inside the procedure
  - The actual gain in efficiency may be trivial or significant, depending on the parameter
"Relax, I've only had two beers"

- Using a (non-\texttt{const}) reference parameter is like giving away the keys to your car at a frat party
  - Just how much do you trust that procedure, anyway?

- Often, you really do need to do it, but evaluate the risk
  - Use a \texttt{const ref} if the param should not be changed by the procedure

\begin{verbatim}
typedef Node* Stack;

// peek is an \texttt{observer}, not a \texttt{mutator} operation
string peek1 (Stack & first) {
    string ans = first->val;
    first = nullptr;
    return ans;
}

string peek2 (Stack first) {
    string ans = first->val;
    first = nullptr;
    return ans;
}

string peek3 (const Stack & first) {
    string ans = first->val;
    first = nullptr;
    return ans;
}
\end{verbatim}

\begin{verbatim}
// …
struct Employee {
    string name;
    int monthlySalary; // … other stuff too, probably
};

// red text means common C++ usage
void GiveRaise1 (Employee e, int raise) {…}
void GiveRaise2 (const Employee e, const int int raise) {…}
void GiveRaise3 (Employee &e, int &raise) {…}
void GiveRaise4 (const Employee &e, …) {…}
void GiveRaise5 (Employee *e, …) {…}
void GiveRaise6 (const Employee *e, …) {…}
void GiveRaise7 (Employee *const e, …) {…}
void GiveRaise8 (const Employee *const e, …) {…}

int main (…) {
    Employee  *pFrank = new Employee;
    pFrank -> name = "Frank Lee";
    pFrank -> monthlySalary = 5000;
    // …
    int franksRaise = 700;
    GiveRaise1 (*pFrank, franksRaise);
    // …
}
\end{verbatim}

Reference parameters vs. ptrs

- The C language does not have references; it requires using pointer parameters to make changes "propagate back" to the calling environment
  - Many C++ systems started out life as a C system (and/or being developed by C programmers), so you will often see this style (ptr params) in industrial C++ code

- The better, cleaner, easier-to-understand C++ approach is to use reference (and \texttt{const ref}) parameters
  - So C++ objects should \texttt{usually} be passed by reference, not value or by pointer
  - The I often "forget" for strings \& similar small objects and pass them by value

- OK to use value params for "small things" (basic types / scalars) like \texttt{int, double, bool,} etc, assuming you don't need changes to propagate back
  - And if you do, you can always use a reference param for them too
void GiveRaise1 (Employee e, int raise) {...}
- When this fcn is called, a copy of the caller’s object/struct instance e and integer raise are created on stack.
- These copies may be changed within the fcn, but the values do not propagate back to caller and are lost at end of method call.
- (OO full truth: the method for making a copy is defined by the Employee copy constructor; we’ll see this in CS247)

void GiveRaise2 (const Employee e, const int raise){...}
- A copy of object e and integer raise are created on stack; they may not be changed within the fcn/method.
- This is pretty similar to GiveRaise1 in terms of its effect on the calling environment.

void GiveRaise3 (Employee &e, int &raise) {...}
- Any mention of object e or integer raise inside the method actually refers to the variables in the caller’s environment.
- Thus, changes can be made to raise or e inside GiveRaise3 propagate back to the calling environment immediately.

void GiveRaise4 (const Employee &e, ...) {...}
- Read this as ”e is a ref to a thing that’s a const Employee”
- Any mention of e refers to caller’s environment, but you can’t change the object e refers to in any way.
- Effectively, this is similar to GiveRaise2 except that it may be more efficient if Employee is a “large” object.

void GiveRaise5 (Employee *e, ...) {...}
- Call w/o dereferencing: GiveRaise5(pFrank,...) [same for 6, 7, 8]
- Read this as ”e is a pointer to an Employee” [more C-style than C++]
- A copy of a pointer is made on the stack; if you manage to change the member variables of the object e points to, these changes will persist in the calling environment.
- You can also make e to point to a new or different object, but that change will be lost at the end of the method call, and not propagate back.

void GiveRaise6 (const Employee *e,...) {...}
- Read this as ”e is a pointer to a thing that’s a const Employee”
- That is, e can be changed to point to a different const Employee, but any instance it points to cannot be changed internally.
- If you do change e to point to another object, this is not propagated back to the caller.

void GiveRaise7 (Employee *const e, ...) {...}
- Read this as ”e is a const pointer to an Employee”
- You can change the internal values of the object e points to, but you can’t make e point to a different object.
- The const doesn’t add much, as with GiveRaise2

void GiveRaise8 (const Employee *const e, ...) {...}
- Read this as ”e is a const pointer to a const Employee”
- Can’t change the value of the object e points to; can’t change e to point to another object.
- A strong guarantee for C, but GiveRaise4 makes the same guarantee and is easier to understand.
• Usually, use the style of
  – GiveRaise3 if you want changes to propagate back to the calling environment, or
  – GiveRaise4 if you want to prevent any changes from propagating back
    • ... but using value params as in GiveRaise1 is also common

• Reference parameters are your new BFF!
  – Prefer them over C-style pointers for parameters that are struct/class instances

---

References vs. reference parameters

• In C++, normal identifiers (variables) can be references too, but generally we use them only to give a more convenient name to something long …

  // Reasonable use
  Employee & e = emplList[Wloo].find(empNum);
  cout << e.getName() << " " << e.getAddr() << endl;

  // Probably useless, and a bad idea
  int x = 13;
  int & y = x;
  y = 36;       // Changes value of x!

• So C++ reference parameters are just parameters that happen to be references, if that makes sense to you
  – If not, don't sweat it too much

---

Reference params and ADTs

• We can now easily switch to reference params in our ADT definitions
  – Old style: functional and stateless
    i.e., pass in old ADT, get new one back as return value of proc
    • Naïve implementation may result in lots of unneeded copying of large objects, depending on impl chosen
  – New style: stateful.
    • Pass in "car keys" of the ADT instance to procs; changes may be made to its internals before being passed back (unless passed as a const reference)
    • No egregiously unnecessary copying

---

// vector-based approach of Stack is now quite reasonable
// No more crazy egregious copying of vector parameters!
#include <iostream>
#include <string>
#include <vector>
#include <cassert>
using namespace std;

// Simple impl using a vector; as before
typedef vector<string> Stack;

// Note that we don't need to define initStack or nuke if we
// use vectors as the underlying implementation mechanism
bool isEmpty (const Stack & s) {
    return s.size() == 0;
}

void push (Stack & s, string e) {
    s.push_back(e);
}

void pop (Stack & s) {
    assert (!isEmpty(s));
    s.pop_back();
}

string peek (const Stack & s) {
    assert (!isEmpty(s));
    return s.back();
}

// Old fcnl way w. value params
bool isEmpty (Stack s) {
    return s.size() == 0;
}

Stack push (Stack s, string e) {
    s.push_back(e);
    return s;
}

Stack pop (Stack s) {
    assert (!isEmpty(s));
    s.pop_back();
    return s;
}

string peek (Stack s) {
    assert (!isEmpty(s));
    return s.back();
}

// Main program as w. linked list w. ref params
int main (int argc, char* argv[]) {
    // Note that we can have two distinct
    // stacks at the same time!
    Stack s1;
    Stack s2;
    push (s1, "alpaca"); // Old: s1 = push (s1, "alpaca");
    push (s1, "beaver"); // etc.
    push (s1, "cat");
    push (s1, "dog");
    push (s2, "one");
    push (s2, "two");
    cout << peek(s1) << endl; // Same as non-ref param vers
    cout << peek(s2) << endl;
    pop(s1); // Old: s1 = pop(s1);
    cout << peek(s1) << endl;
}

const bool altTrue = false;

Lecture 7

CS138 W17
Reference parameters

- **C/Java support only call-by-value (CBV)**
  - A new copy is made of each variable passed as a parameter to a function; this can be expensive if they are "big" objects (e.g., vector)
  - Change made to the parameter affect only the new local variable; they do not propagate back to the calling environment
  - This is the default approach in C++ (because it's backward compatible with C)

- **C++/C# support call-by-reference (as well as CBV)**
  - Changes made to a reference parameter inside a procedure propagate back to the calling environment
  - Use a const reference parameter when you want to be able to use just the value of the parameter; it's more efficient than CBV, esp. if the object is "big"
  - Reference parameters are the preferred approach in C++ (but not the default)

---

// New way with ref params
bool isEmpty (const Stack& s){
    return s.size()==0;
}

void push (Stack& s, string e){
    s.push_back(e);
}

void pop (Stack& s) {
    assert (!isEmpty(s));
    s.pop_back();
}

string peek (const Stack& s){
    assert (!isEmpty(s));
    return s.back();
}

---

// Old way with value params
bool isEmpty (Stack s) { return s.size()==0; }

Stack push (Stack s, string e) {
    s.push_back(e);
    return s;
}

Stack pop (Stack s) {
    assert (!isEmpty(s));
    s.pop_back();
    return s;
}

string peek (Stack s) {
    assert (!isEmpty(s));
    return s.back();
}

---

The 3 kinds of variables you meet in C++

- **Global variables** are defined outside of any enclosing function/class/struct
  - They come into existence when declared and die at the end of the program
  - We haven't really used them in CS138; their use is generally frowned upon

- **Local variables** (incl. params) are defined within a function/method body
  - They come into existence (eventually) when the function is called, and die when it terminates (if not before)

- **Member / instance variables** are a sub-part of a larger variable that is an instance of a struct or class
  - They are born when the instance is created, and die when the instance dies

[We ignore static variables and other minor exceptions for now]
Scopes of identifiers and ARs

• In C/C++, an **identifier** (a name you picked, e.g., for a variable or procedure) is visible from its declaration until the end of the current **scope**, which could be the:
  – End of current `{}` block
  – End of a procedure body
  – End of loop / if / switch body, etc.
  – Global variables are visible globally (until end of file scope)

• A scope is a (usually) contiguous chunk of a program that delineates boundaries of visibility for identifiers
  – Before the 1999 standard, all declarations in C had to come at the beginning of a scope (not true for C++)

Scopes of identifiers and ARs

"Outside of a dog, a book is a man’s best friend. Inside of a dog, it’s too dark to read."
— Groucho Marx on scopes

• When the current scope is exited, the AR for it is destroyed, along with the storage for the variables
  – Control returns to the calling fcn at the point the call was made
  – So you need to take care where things are declared within a function ...
  – If you want to return a variable as the value of a function at the very end, then that variable had better be declared at the top level scope within the function

Scopes of identifiers and ARs

• As your program executes, scopes are entered and exited

• When a new scope is entered, an **activation record** (AR, area of storage on the run-time stack, also called a **stack frame**) is created for it
  – ARs contain storage for parameters, the (eventual) return value (if any), and local variables PLUS it remembers the location of where the call was made from in the calling environment
    • Sometimes the same fcn can be called in multiple places within a fcn
  – When a variable declaration is encountered, space for that variable is created in the current AR

• Note that the "truth" about when variable scopes begin/end is slightly more complicated than what we've been implying
  – When a new function call is made, an AR with storage for the parameters and return value (if any) is created on the run-time stack
  – As the function executes and declares new local variables, storage is allocated within the AR for them
  – When execution leaves a sub-scope of the function, the storage for any local variables declared inside that sub-scope is deleted
  – At the end of the function call, the return value (if any) is copied back to the calling environment and the AR is deleted
```cpp
#include <iostream>
#include <string>
using namespace std;

int max (int x, int y) {
    if (x > y)
        return x;
    else
        return y;
}
int main (int argc, char* argv[]) {
    cout << max (5,17) << endl;
}
```

```cpp
#include <iostream>
#include <string>
using namespace std;

int max (int x, int y) {
    int bigger;
    if (x > y)
        bigger = x;
    else
        bigger = y;
    return bigger;
}
int main (int argc, char* argv[]) {
    cout << max (5,17) << endl;
}
```
int main (int argc, char* argv[]) {
    int x, y;
    cin >> x >> y;
    if (x == y) {
        cout << "they're equal" << endl;
    } else {
        int ans = max (x, y);
        cout << "max value is " << ans << endl;
    }
    // note: ans is not visible from here on
    // ...
}

#include <iostream>  int main (int argc, char* argv[]) {
#include <string>  int k;
using namespace std;

    // We'll use this version
    int fact (int n) {
        if (n <= 1) {
            return 1;
        } else {
            return n * fact(n-1);
        }
    }

    // Compact, somewhat cryptic version;
    // we would probably write this instead
    // but the above is easier to talk about
    int fact2 (int n){
        return n <= 1 ? 1 : n * fact2 (n-1);
    }

    // Summary
    // The run-time call stack
    // At any given moment, you might have several scopes open
    // due to nested procedure calls
    // e.g., main calls f calls g calls h
    // – If you pause the execution of a program, the run-time call stack will
    // contain an activation record (aka stack frame) for each still-active
    // procedure call in the pending call chain
    // – When the current (i.e., most deeply nested) call completes, the return
    // value (if any) is copied back to the calling context and the AR is
    // deleted, along with storage for parameters and local variables created
    // during that procedure call
    // – Eventually, all calls will finish (including the original implicit call to
    // main made by the system), and their ARs will be popped/deleted too

    // [See diagram]

    Recursion
    // You may have noticed that the fact function calls
    // itself; as you know, this is called recursion
    // – In CS137, you saw GCD, mergesort, quicksort, etc.

    // Recursion is an elegant and powerful concept for
    // problem solving
    // – But it’s nothing magical to implement; you just get
    // multiple ARs for the same function in the call stack at the
    // same time.

    [See diagram]
Recursion

- Recursion is a general technique for solving a large problem, typically by progressively breaking down the input until it's "small enough" to solve easily, then combining the results back together.

- (Usually, but not always) there are three basic parts to a recursive solution to a problem:
  1. **Trivial base case(s)** that can be solved easily and directly
  2. **Reduction operator** that makes the data "smaller", closer to a base case
  3. **Composition operator** that composes the answer to the "smaller" problem to get the full answer

```c
int fact1 (int n) {
   if (n <= 1) {
      return 1;
   } else {
      return n * fact1 (n-1);
   }
}
```

```
const int KEY_NOT_FOUND = -1;

// recursive binary search
int binary_search(vector<string> v, string key, int imin, int imax) {
   if (imax < imin) {
      return KEY_NOT_FOUND;
   } else {
      int imid = (imin + imax) / 2;
      if (v.at(imid) > key) {
         return binary_search(v, key, imin, imid - 1);
      } else if (v.at(imid) < key) {
         return binary_search(v, key, imid + 1, imax);
      } else {
         return imid;
      }
   }
}
```

Towers of Hanoi

- We’ll come back to recursion is more detail when we study more ADTs (esp. trees) ... but before we go, let’s look at a classic example:
  - The towers of Hanoi [Lucas, 1883]

- **Classic formulation:**
  - In a temple in Hanoi, there are 64 differently-sized disks stacked on a pole in increasing order of size
  - The monks must move the disks one at a time from one pole to a second pole, using a third pole as a temporary
  - No larger disk may ever sit on top of a smaller disk
  - The solution is really short and elegant!

Q: if they move one disk per second, how long to move the whole stack?
The queue ADT

More formally:
• initQueue creates + returns a new empty queue
• enter (enqueue) takes an element \(e_{\text{new}}\) and a queue \(q\) and returns a new queue with the new element at the end: \(e_1 \ldots e_n e_{\text{new}}\)
• leave (dequeue) takes a queue \(q\) and returns a new queue that’s identical to \(q\) except with the first element removed: \(e_2 \ldots e_n\)
• first (front) takes a queue + returns (but doesn’t remove) first element: \(e_1\)
• isEmpty Takes a queue and returns true or false, depending if the queue is empty: i.e., does \(N=0\)?
• nuke just cleans up ... no observable meaning to the client

[For each, assume \(q\) is a queue whose current sequence is \(e_1 \ldots e_n\) and \(e_{\text{new}}\) is the new element]

The queue ADT: Even more formally

initQueue: \(\rightarrow\) queue
  
  Pre: 
  Post: 

enter: queue \(X\) value \(\rightarrow\) queue
  
  Pre: 
  Post: 

leave: queue \(\rightarrow\) queue
  
  Pre: 
  Post:
The queue ADT: Even more formally

first : queue -> value
Pre: !isEmpty
Post: returned value is 1

isEmpty : queue -> boolean
Pre: true
Post: return value is the same as N==0

nuke : queue -> queue
Pre: true
Post: return value empty; old nodes deleted

typedef <mumble> Queue
// If we were using the truly functional style, we'd do this
Queue initQueue() {...}
bool isEmpty (Queue q) {...}
Queue enter (Queue q, string val) {...}
Queue leave (Queue q) {...}
string first (Queue q) {...}
Queue nuke (Queue q)

int main (int argc, char* argv[]) {
    Queue q1;
    q1 = enter (q1, "early");
    q1 = enter (q1, "timely");
    q1 = enter (q1, "late");
    cout << first(q1) << endl;
    q1 = leave (q1);
    cout << first(q1) << endl;
    q1 = nuke (q1);
    return 0;
}

typedef <mumble> Queue
// But we're going to use reference parameters instead
void initQueue (Queue& q) {...}
bool isEmpty (const Queue & q) {...}
void enter (Queue & q, string val) {...}
void leave (Queue & q) {...}
string first (const Queue & q) {...}
void nuke (Queue & q)

int main (int argc, char* argv[]) {
    Queue q1;
    enter (q1, "early");
    enter (q1, "timely");
    enter (q1, "late");
    cout << first(q1) << endl;
    leave (q1);
    cout << first(q1) << endl;
    nuke (q1);
}
Implementing a queue

• A queue implementation requires access to both ends of the list
  – enter adds new elements onto the "end"
  – leave removes old elements from the “beginning”

• This means we’re going to need to keep track of two indices / pointers / somethings for each queue instance
  – Hmmmm

• We’ll try using a vector, and then a linked list
  – And yes, there is a robust queue data type in the C++ Standard Library that you should use in real life

    // This design can become very inefficient
    // over time, but it does "work"
    struct Queue {
        vector<string> store;
        int first;
    };

    void initQueue (Queue & q) {
        q.first=0;
    }
    
    bool isEmpty (const Queue & q) { // NOT this: q.store.empty() … but why?
        return q.store.size() == q.first;
    }

    void enter (Queue & q, string val) { // nit: similar to q.store.resize(0)
        q.store.push_back(val);
        initQueue (q); // clear may not reset capacity to 0, tho
    }

    void leave (Queue & q) {
        // Note that we do NOT call q.store.pop_back()
        // We just increment the index to the first element
        assert (!isEmpty(q));
        q.first++;
    }

    string first (const Queue & q) { // nit: similar to q.store.resize(0)
        assert (!isEmpty(q));
        return q.store.at(q.first);
    }

    void nuke (Queue & q) {
        q.store.clear(); // nit: similar to q.store.resize(0)
        initQueue (q); // clear may not reset capacity to 0, tho
    }

Implementing a queue with a vector

• Implementing a stack was easy using a vector. How can we implement a queue?
  – If we used a vector, we’d also have to keep track of where the first element was too
  – Naive vector approach: enter calls push_back

• But where do we store first?
  – Can’t make it a standalone global variable, as we want to be able to create multiple distinct queues
typedef <mumble> Queue

void initQueue (Queue& q) {…}
bool isEmpty (const Queue & q) {…}
void enter (Queue & q, string val) {…}
void leave (Queue & q) {…}
string first (const Queue & q) {…}
void nuke (Queue & q) {…}

int main (int argc, char* argv[]) {
    Queue q1;
    enter (q1, "early");
    enter (q1, "timely");
    enter (q1, "late");
    cout << first(q1) << endl;
    leave (q1);
    cout << first(q1) << endl;
    nuke (q1);
}

Implementing a queue with a vector

• Suppose we had a queue that was never more than three elements long at any given moment, but might have a gazillion elements in its lifetime
  – This would be really space-inefficient (unless we resized it sometimes, or used a circular array, but that’s less simple)
  – Instead, let’s use a struct with ptrs … in a minute

Implementing a queue as a linked list

• Let’s try using a linked list to implement a queue, similar to how we used one to implement a stack
  – … but with a pointer to both the first and last element
  – We’ll define initQueue, isEmpty, enter, leave, and first

typedef <mumble> Queue // Node* or … what?

// API below is the same as before!
void initQueue (Queue& q) {…}
bool isEmpty (const Queue & q) {…}
void enter (Queue & q, string val) {…}
void leave (Queue & q) {…}
string first (const Queue & q) {…}
void nuke (Queue & q)

int main (int argc, char* argv[]) {
    Queue q1;
    enter (q1, "early");
    enter (q1, "timely");
    enter (q1, "late");
    cout << first(q1) << endl;
    leave (q1);
    cout << first(q1) << endl;
    nuke (q1);
}
Implementing a queue as a linked list

- We still want to define a nice abstract, clean looking type for clients to use ...
  - ... but we now have to keep track of both a first + last ptr
  - ... which means we can’t use the trick of letting a Node* represent the ADT as we did with Stack
  - Instead, we need a structured representation to represent a queue
    i.e., a struct to model the queue itself, in addition to the Node struct type

A philosophical point

- Does the check in leave really need to be done?
  - No, as long as we leave the rest of the code as-is
  - If we used nullptr==q.last as the test for isEmpty, then it would not work
  - However, it’s a little dangerous to break the representational consistency of first and last both being NULL or both being non-NULL
  - Probably better to keep the representation consistent in case you decide to change your mind about the implementation later

Complexity of queue operations

- Linked list implementation
  - enter, leave, first, isEmpty, nuke

- Vector implementation
  - Umm, no thanks.
  - enter is Amortized Constant Time
  - The other operations are O(1), but the wasted memory mounts up as the Queue is used more and more
A note on defensive programming

- When you start to fool around with ptrs, it's easy to make nasty mistakes
  * e.g., forget to set a ptr, set it to the wrong value, set it to a node that gets deleted later
  * You may never notice the error, or see it manifested only much later on

- Two complementary approaches to understanding what went wrong:
  1. Using a debugger (long tutorial, but you will need this one day)
  2. Inserting print stmts and assertions (easy)

- Also, comment anything that's non-obvious. You will thank yourself later on (and your colleagues may thank you too)
  - But don't over-comment!

A note on defensive programming

```c
#include <cassert>   // not assert.h
```

- Why should we use assertions, if they make our program die?
  - Use it anywhere you want to be dead sure of some fact before proceeding; if it’s false, there’s no point in continuing

- Two usual kinds of cases:
  1. Check that your logic/assumptions are correct (up to you to get right)
     * If first is nullptr you expect that last will be too
  2. Check that “user” is being reasonable (up to user to “behave”, not you)
     * e.g., popping an empty stack

A note on defensive programming

- When to insert a (debugging) print stmt?
  - At the beginning of a function call (print function name + params)
  - At the end of a function call (print results / state of world)
  - Just before deleting something
  - At any major decision point
  - Any time you do something "interesting"

- It's also helpful to create special purpose print functions for whole data structure (again, for debugging use only)
  * e.g., print a whole stack, print all values of a struct instance
  - Printing ptr numeric values not very useful (it's a number, big deal)
    * Print the value of the things they point to instead

A note on defensive programming

- But clean up when you’re done!
  * Often for production code (that gets compiled and shipped to clients), we remove "type 1" assertions, diagnostic print stmts, and special print functions
  * OK to have them in during development + debugging, but when you’re ready to ship, we often get rid of them (comment/macro them out)

- "Type 2" assertions can sometimes be left in, as long as the check isn’t very expensive
  * e.g., checking if a ptr is nullptr is cheap, but checking if a list is sorted is expensive
// Here's our Queue, with helpful debugging output // Can add a guard like this
#include <iostream>
#include <string>
#include <cassert>
using namespace std;
const bool DEBUG = true;
// const bool DEBUG = false;
// many of these per Q
struct Node {
    string val;
    Node* next;
};
// one of these per Q
struct Queue {
    Node* first;
    Node* last;
};

void initQueue (Queue& q) {
    if (DEBUG) {
        cerr << "initQ" << endl;
    }
    q.first = nullptr;
    q.last = nullptr;
}

bool isEmptyQ (const Queue &q) {
    // ... or not ...
    cerr << "isMTQ" << endl;
    return nullptr == q.first;
}

void enter (Queue& q, string val) {
    cerr << "enter: " << val;                  // no endl yet
    Node* p = new Node;
    p->val = val;
    p->next = nullptr;
    if (nullptr == q.first) {
        assert (nullptr == q.last);
        cerr << "at front" << endl;            // endl here
        q.first = p;
    } else {
        assert (nullptr != q.last);
        cerr << "after" << q.last->val << endl; // and here
        q.last->next = p;
    }
    q.last = p;
}

string first (const Queue & q) {
    assert (!isEmptyQ (q));
    cerr << q.first->val << endl;
    return q.first->val;
}

void leave (Queue & q) {
    assert (!isEmptyQ (q));
    cerr << q.first->val << endl;
    q.first = q.first->next;
    if (nullptr == q.first) {
        q.last = nullptr;
    } else {
        assert (nullptr != q.last);
        cerr << "after" << q.last->val << endl; // and here
        q.last->next = p;
    }
    q.last = p;
}

void nuke (Queue & q) {
    cerr << "nuke " << endl;
    while (!isEmptyQ (q)) {
        cerr << "nuking element " << first(q) << endl;
        leave(q);
    }
}

// A non-standard API element for Queue, // used only for debugging!
void printQ (const Queue & q) {
    cerr << "print the values in order";
    // (implementation left to student)
Dynamic arrays

- We're going to discuss "dynamic arrays" now to give you an understanding of what's going on "under the hood" with vectors.

- However, if you feel the urge to use a dynamic array in your program (ignoring the next assg :-), you should almost certainly use an STL vector (or relative) instead:
  - Vectors (a library) are usually implemented using a dynamic array (a fundamental feature of the programming language), which is why we are studying them.
  - In fact there's not much need to use any kind of C-style arrays in C++ since we have vectors, plus other powerful container classes in the STL.

- C++ arrays can be declared in two ways:

  1. Statically (as in C)
     - Storage allocated on the stack
     - Array bound (N) must be a compile-time constant

     ```cpp
     int main (...) { 
       const int N = 5;
       int A[N]; // legal
       int m;
       cin << m;
       int B[m]; // illegal
     }
     ```

     [See diagram]

  2. Dynamically (C++ only)
     - Storage is allocated on the heap
     - Array bound can be a run-time value (positive integer)
     - Must delete when done
       - Need "[]"

     ```cpp
     string* myAlloc (int n) {
       assert (n>0);
       return new string[n];
     }
     ```

     [See diagram]

- Can't return an array from a procedure
  - But arrays are almost just pointers (with a special syntax for accessing elements, etc.)
  - So we can return a ptr instead, if we allocate the storage on the heap

```cpp
int main (...) { 
  int N;
  cin >> N;
  int* A = new int[N];
  string* B = myAlloc(N);
  for (int i=0; i<N; i++) {
    cin >> A[i] >> B[i];
  }
  // ... do more fun stuff
  delete [] A;
  delete [] B;
  return 0;
}
```
Dynamic arrays

- How does "delete []A" work?
  - The system needs to remember how many elements are in the heap-chunk that was allocated for the dynamic array, i.e., the extent

Q: The extent of the array must associated physically somehow with the block of storage on the heap, not with the ptr. Why?
A:

Q: Why not store the extent at the beginning of the array (i.e., address A)?
A:

No extent for you!

- The extent of a dynamically allocated array has to be stored somewhere ...
- ... BUT it is not accessible programmatically or any other way (modulo non-portable "cheating"):
  - Only the compiler/run-time truly knows where to find it
    - Some compilers may do things differently from what we did here
  - A dynamically allocated array is just an array, it is not an object and does not have an API ... so there is no size() method
    - It's a language feature built in to C++, not a library!

The linked list

- The generic "thing" we have been working with to implement our stacks, queues, etc. is called a "linked list". It has (at least)
  - A special ptr to the "first" element
  - A bunch of "node" instances, linked to each other to form a "line" via a "next" pointer
- Linked lists can be
  - unordered, or
  - ordered by insertion time, or
  - ordered by key value, or
  - ordered by some strange "other" convention

#include <iostream>
#include <string>
#include <cassert>
using namespace std;

string* myAlloc (int n) {    
  assert (n>0);    
  return new string[n];    
}

int main (int argc, char* argv[]) {    
  string* A = myAlloc(10);    
  string* B = myAlloc(20);    
  string* temp = A;    
  A = B;    
  // No problem "swapping"; they're just ptrs!    
  B = temp;    
  delete []A;    
  // If we stored extent w ptr, this    
  delete []B;    
  // would NOT work, but it does
}

[See diagram]
Variants of the linked list

- Doubly-linked list:
  - Link to first and last elements of list, can traverse forwards or backwards

```c
struct Node {
    string val;
    Node* next;
    Node* prev;
};
```

Lists and ordering

- So far, our lists (stack, queue) have been ordered by arrival time in some way
  - These are (sometimes) called ordered lists
  - i.e., the ordering doesn't depend on the value of the elements

- We can also create lists sorted by the ("key") value of the element
  - These are called sorted lists
  - You can imagine that the data stored in the node is much richer (i.e., more fields) than just a single string, but we'll use only string keys for simplicity

Sorted linked list

- Let's use the same Node and create a sorted list ADT
- We need to define:
  - `initList`, `insert`, `remove`, `has`, `isEmpty`, `print`
- This is going to be a bit trickier than what we've done so far

[The priority queue (coming soon) is a hybrid, partly based on order of arrival and partly based on data (the priority)]

Variants of the linked list

- Binary tree (more later)
  - Special "root" node
  - Each node has two "child" links, a left and a right
  - Can be sorted (BST, heap) or not, depending on use
  - Non-binary trees also exist

```c
struct Node {
    string val;
    Node* left;
    Node* right;
};
```
Getting linked structures right

• Linked structures are easy to get wrong!
  – You may not notice the incorrect link until much later; often it’s hard to figure out what went wrong
    e.g., forgetting to set link of last element to `nullptr` is not a problem if you never go to the end again

• Think defensively! Some tricks:
  – When you declare a ptr, set it to some meaningful value soon after
    • Newly declared ptrs have random garbage as default initial values
  – Set pointers to `nullptr` if they’re not pointing to something “active”
  – Create “state-reporting functions” to help in debugging
  – Use IO and assertions to check your assumptions
  – (eventually) Learn how to use a debugger

Suggestions:

1. Draw pictures! (Even I do this 30+ years later)

2. Break down into all possible "interesting" cases you can think of:
   – Empty list, one element list, "big" list
   – First element, middle elt, last elt
   – Thing not there, duplicate thing, all values the same, ...
   – See if you can merge some of the cases together later

3. Testing, testing, testing!
   – Don’t rely on what we put into Marmoset to tell you everything that’s wrong; it’s there to help but it’s not a crutch
   – Don’t be shy about creating large numbers of test cases; that’s what the pros do!

Keener Korner

You are your own best tester!

A simple process you can implement now!

1. Design a set of test cases for your code; give each a descriptive name
   – This might be done with a common driver (`main`) program that can read in the test case from a data file, or you may need a new `main` program for each (which you can automate with a simple shell script)

2. For each test case, create a plain text file with the expected output for each test run
   e.g., `t24-illegalWidth-expected.txt`

3. Run each test, store the output in a like-named file (one per test run)
   e.g., `t24-illegalWidth-out.txt`

4. Use `diff` to compare the expected to the actual
   e.g., `diff t24-illegalWidth-expected.txt t24-illegalWidth-out.txt`

Test like a pro!

• Regression testing
  – Keep your test cases around, retest *everything* when you make a change; again, this is what the pros do!
    • "A bug is simply a test case you forgot to write."
  – If you "break the build" by committing code that doesn’t pass the tests, you get to buy donuts for the next standup meeting

• Test-driven development is commonly part of most "agile development" processes
  – Develop test cases that satisfy the reqs *before* you write the code
  – Then write the simplest code that satisfies the tests
  – Keep the test cases up to date! The tests evolve as your code does!
Over-engineering and the Rule of 3

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

~ John Woods

Lecture 9
CS138 W17

The sorted (singly) linked list

```c
typedef SortedList // ... Hmmmm
SortedList initList (SortedList & first) {...}
bool isMT (const SortedList & first) {...}
bool lookup (const SortedList & first, string val) {...}
int size (const SortedList & first) {...}
void insert (SortedList & first, string val) {...}
void remove (SortedList & first, string val {...)
```
// We're implementing a sorted singly-linked list
#include <iostream>
#include <string>
#include <cassert>
using namespace std;

struct Node {
    string val;
    string otherStuff;  // we won't really touch this
    Node* next;
};

typedef Node* SortedList;

void initList (SortedList& first) {
    first = nullptr;
}

bool isEmpty (const SortedList& first) {
    return nullptr == first;
}

bool lookup (const SortedList & first, string val) {
    // Let's implement these!
    // Typically, nodes store key value plus other info.
    // Thus lookup would return the "other info" for that
    // key (e.g., lookup by student number returns name,
    // address, course marks etc.) Since this is simpler
    // version, we'll just return a boolean to indicate if
    // we found it.
    bool lookup (const SortedList & first, string val) {...}

    // Also ignoring "otherStuff" from here on in.
    void insert (SortedList& first, string val) {...}
    void remove (SortedList& first, string val) {...}
}

• For most of these operations:
  ─ A key value is provided as a parameter (unlike pop, leave)
  ─ Need to loop through the elements, starting at first
  ─ Remember that the list is sorted by val!
  ─ Stopping criterion is ... (it depends)
  ─ Then we do ... (it depends)

• For lookup:
  ─ Start cur pointing to the first element
  ─ Loop thru the list: cur = cur->next
  ─ Loop exit condition:
    ─ This works: when we find element we're looking for or reach list end
    ─ This is better: when we find element we're looking for or reach element
      larger that sought element or reach list end
  ─ Finally, return true or false, depending on if we found it or not

Lazy (aka short-circuit) evaluation

• What happens here if cur == nullptr?
  while (nullptr != cur && cur->val < val)
  ─ Doesn't the second part of the condition cause a run-time error?

• Answer: No, it doesn't. The run-time system is clever enough
  to evaluate only as much as is needed to determine the truth
  of the boolean condition
  ─ if (A | | B) will not try to test B if A is true
  ─ if (A && B) will not try to test B if A is false

• This is called lazy evaluation aka short-circuit evaluation;
  many modern programming languages support it.
For insert:
- Create a new node, set its val
- Start cur at first element
- Iterate thru list until you find cur->val ?? val
- Adjust previous element to point to new node
- Adjust new node to point to what previous element pointed to
- Note that first may need to change, but only if we insert new element at the beginning.

Insert

void insert (SortedList& first, string val) {…}  

[Let's brainstorm for cases to consider; it's OK if some overlap]
1. Empty list
2. Single element list (insert before / after)
3. Multi-element list; insert at beginning
4. Multi-element list; insert in middle
5. Multi-element list; insert at end
6. List already contains value (for each of above cases)

[Note: this would be an excellent basis to start designing your test cases from]
Lecture 10

CS138 W17

The sorted (singly) linked list

typedef SortedList // Hmm
SortedList initList (SortedList & first) {...}
bool isMT (const SortedList & first) {...}
bool lookup (const SortedList & first, string val) {...}
int size (const SortedList & first) {...}
void insert (SortedList & first, string val) {...}
void remove (SortedList & first, string val) {...}

• For remove:
  – Start at the beginning, iterate until the cur elt is >= the sought element
  – Adjust previous node to point to cur->next node
  – delete the Node
  – Note that first may have changed, but only if we deleted the first element

Remove

void remove (SortedList& first, string val) {...}

[Let's brainstorm for cases to consider; it's OK if some overlap]
Remove

```cpp
void remove (SortedList& first, string val) {…}
```

[Let's brainstorm for cases to consider; it's OK if some overlap]

1. Empty list (error; assert at beginning)
2. Non-empty list; remove first element
3. Non-empty list; remove middle element
4. Non-empty list; remove last element
5. Element not found (can't tell in advance! when do we stop?)
6. Multiple occurrences of element (which one is deleted?)

```cpp
// Final version, that combines all cases into
// two main scenarios; requires list be sorted already.
void remove (SortedList& first, string val) {
    // cerr << "Deleting " << val << "\n"; // debug
    // Scenario 1: List MT, abort
    assert (!isEmpty(first));
    Node* temp;
    if (first->val == val) {
        // Scenario 2: Non-MT list, delete first element
        temp = first;
        first = first->next;
        // cont'd
    } else {
        // Scenario 3-5: Non-MT list, delete non-first element
        Node* cur = first;
        while (cur->next != nullptr && val > cur->next->val) {
            cur = cur->next;
        }
        // My orig 2009 version: nullptr==cur (WRONG!)
        if (nullptr == cur->next || val != cur->next->val) {
            // Scenario 5: Non-MT list, element not present
            assert (false);  // or better: just return
            cerr << "Couldn't find " << val << "\n";
            return;
        }
        // Scenario 3-4: Non-MT list, element present
        temp = cur->next;
        cur->next = cur->next->next; // aka temp->next
    }
    // Common to both scenarios
delete temp;  // Note that we punt on scenario 6 for now
```

Designing test cases

- In Winter 2010, I discovered a bug in the remove code I had written for Winter 2009
  - It manifested itself only when you tried to remove an entry that wasn't actually there AND was greater than the last element in the list
  - But if you never tried that case, it looked correct.
  - Subtle and hard-to-spot errors like this are quite common in code involving linked structures!
Designing test cases

• Lessons learned:
  – Linked structures are really hard to get 100% correct
  – It's common for obscure bugs to be discovered later on, even after the code has worked pretty well for a long time, based on possible but statistically unlikely scenarios
  – Bugs often show up only through systematic testing
    • Unless you are an iOS developer 😊
  – Testing is good for your code: Be brutal and creative
  – Using debugged libraries is always better than re-inventing the wheel; you will probably get it wrong the first, second, third time you try

Some test cases for remove

• Element present
  – One element list
  – Remove first element
  – Remove middle element
  – Remove last element
  – Remove last-but-one element

• Element not present
  – No elements (assertion failure, we hope)
  – Before first element
  – Between middle elements
    – After last element
  – Element already deleted

Fun facts about designing test cases

• Fundamental truth of sw testing:
  – You simply cannot test "everything"; combinatorial explosion of possible worlds hits you very quickly
  – So testing becomes an optimization game: Given XX person-hours, what is the most effective way to design, implement, and repeatedly execute a testing infrastructure?
    • Learn to recognize when a single test case can be a proxy for several conditions

• It is much better to carefully and systematically design a set of cases that test various explicit situations than to throw a bunch of quasi-random data at the system

Fun facts about designing test cases

• In addition to testing against the abstract idea of "what the procedure is supposed to do" (black-box testing), there are approaches that test against "how the code is actually written" (white-box testing) and other approaches too
  – Test case design is an advanced topic which you will revisit in later courses

• Black box testing:
  – What possible different cases could be true of the incoming data?
    e.g., List with 0, 1, 2, K, BIG elements, val in/not in list

• White-box testing:
  – Construct different test cases to (ideally) exercise every line of code, every if-condition, execute each loop 0, 1, 2, K, BIG number of times, …
Complexity of sorted linked list ops

- **lookup** (to get to the element's correct place):
  - Average case
  - Worst case

- **insert, remove:**
  - Effectively, they require a lookup too
  - But once you arrive,
    - The C++ Standard Library container data structures sometimes return iterators that point to elements in the middle of the list

- The C++ Standard Library container class **list** is implemented as a plain old doubly-linked list

... compared to a sorted array

- Accessing an array element is:
  - a constant time operation! O(1)
  - e.g., A[15] == addressOfA + 15 * elementSize

- But arrays have a fixed size, unlike linked lists, we can't (easily) grow them
  - vectors have extra space at the end, and can grow as needed

- If you were to insert a new element "in place":
  - It's O(log N) to "find the right spot" (using binary search)...
  - ... then followed by O(N) copy actions, which is pretty expensive
    (cos you have to move about half of the list over one slot to the right)

(Review) The sorted linked list

![Sorted linked list diagram](image)

- **Supports:**
  - lookup, insert, remove, nuke O(N)
  - initSortedList, isMT, size‡ O(1)

‡ The SortedList type would probably have a pointer to the first element plus a size counter that is kept up to date with inserts and removes

Sorted linked list:
Pre- and post-conditions

**enter** : sortedList X value -> sortedList
Pre:
Post:

**remove** : sortedList X value -> sortedList
Pre:
Post:
(Review) ADTs and data structures so far

- Linked list (can be ordered, sorted by value, or neither)
  - Implemented by struct instances + pointers in C++

- Stack (ordered)
  - Implemented by vector, linked list

- Queue (ordered)
  - Implemented by linked list

- Vector/sequence (ordered)
  - Implemented by dynamic array in C++

The priority queue ADT

- A priority queue is like a queue, but each element has a value and an integer priority (usually, >=0)
  - `enter` is "same as before" (details vary by implementation choice)
    - From the outside, you can't really tell what happens until you call `leave`
  - `leave` means "remove the oldest element from among those with the most important (lowest or highest, depending) priority"
  - Thus, the PQ is a data container that is both sorted (queues are sorted by priority) and ordered (FIFO within each priority's own queue)

- Often used in network routing
  - Some kinds of data are more important than others; they get preferential treatment
    - e.g., QoS, media streaming, VoIP

// A correct but naïve implementation
struct Node {
  string val;
  int priority;
  Node* next;
};
typedef Node* PQ;

void initPQ (PQ& pq) {
  pq = nullptr;
}

void leavePQ (PQ& pq) {
  assert (!isMTPQ(pq));
  Node* p = pq;
  pq = pq->next;
  delete p;
}

void firstPQ (const PQ& pq,
              string& val,
              int& priority){
  assert (!isMTPQ(pq));
  val = pq->val;
  priority = pq->priority;
}

bool isMTPQ (const PQ& pq){
  return nullptr == pq;
}
void enterPQ (PQ& pq, string val, int priority) {
    Node* p = new Node;
    p->val = val;
    p->priority = priority;
    if (isMTPQ (pq) || priority < pq->priority) {
        p->next = pq;
        pq = p;
    } else {
        Node* temp = pq;
        // want to find first node > cur priority, not >=
        while (nullptr != temp->next
            && priority >= temp->next->priority ) {
            temp = temp->next;
        }
        p->next = temp->next;
        temp->next = p;
    }
}

We used ">" in sorted list insert

Complexity of PQ operations

- For this particular (naive) implementation only:
  - firstPQ:
    - Complexity:
  - leavePQ:
    - Complexity:
  - enterPQ:
    - Complexity:

An even simpler PQ implementation

- enterPQ:
  - Keep a linked list sorted in arrival order (i.e., like a queue)
    - Complexity:
  - leavePQ, firstPQ:
    - Start at beginning, look for lowest priority element
      - But how do we know what the lowest priority is? Umm, ...
      - Complexity:

[This is pretty awful]
PQ: A better implementation LOL

- We maintain a list-of-lists (LOL)!
- Specifically, we maintain a list of queues that is sorted by priority
  - The (outer) list is sorted numerically by priority
  - This is what you will implement in an upcoming assgt!
  - Can re-use code from queue and sorted linked list!

PQ: LOL implementation

```cpp
void first_PQ (const PQ& pq,
               string& val,
               int& priority)
pre:  !(isEmptyPQ(pq))
Post: Return the value of first element of first queue!

Complexity:
```

PQ: LOL implementation

```cpp
void leave_PQ (PQ& pq)
pre:  !(isEmptyPQ(pq))
Post: Perform leaveQ on first queue in LOL
  - This will delete the Qnode for the "first" element
  - If the queue for this priority is now empty, we should delete the PQnode too

Complexity:
```
**PQ: LOL implementation**

```cpp
void enter_PQ (const PQ& pq, string val, int p)
    - First, find out if there is an existing queue for priority `p`
      • If not, create a new PQnode, and insert it into the outer sorted list in its
        proper place; create a new (empty) queue that this node will point to
    - Now, add `val` to the queue for priority `p`
```

**Complexity:** [Assuming `N` elements in total, and `k` distinct active priorities]
- Step 1: Find appropriate queue (if it exists, or create one)
  • This is O(k)
- Step 2: Add new element at end of that queue
  • This is O(1)
- So the total time is O(k). If k<<N this is a big win!
- So the complexity of `enter_PQ` is "independent" of the number of nodes; it
  depends only on the number of different (active) priorities!

---

**PQ: Usually, tho, it's a heap**

- PQs are usually implemented using a **heap** data structure, which is a kind of **binary tree** (next lecture topic)
  ["Freestore"/'the heap" is NOT a heap in this sense! Confusing! I know!]

- The "heap property" (true for each node in tree; see CS240):
  • Value of parent node >= value of (both) children

- This means that the largest element is always at the root
  • Insert/delete means start at root and swap positions with children
  • Usually, the tree is kept "packed" so a vector can be used to store it
  • There are many varieties of heap implementations with different
    performance characteristics, but in the vanilla version `enterPQ,`
    `leavePQ,` and `lookupPQ` are all O(log N); `firstPQ` is O(1) tho
Heap insert/delete (simple version)

- **Insert** $O(\log_2 N)$
  - i.e., insert new value at "end", then fix the tree so it's a heap again
    - Insert a new node into the next empty slot
    - i.e., new left child of node with value 3
    - Compare to parent, swap if child is bigger
    - Continue to swap with parent until parent is bigger or reach root

- **Delete** $O(\log_2 N)$
  - i.e., return max value (at the root), then fix the tree so it's a heap again
    - Move "last" node into root (and delete from its old position)
    - Compare root to children, if not larger than both then swap with largest child
    - Continue until done (parent larger than both children or it's a leaf)

How a heap is usually implemented

- We usually implement a binary heap using a vector (?!!)
  - Element type is usually some kind of struct with a key (here, "priority") field

- Don't explicitly store ptrs, instead calculate index of a node's parent/children; for node at index $i$ of the underlying vector:
  - Left child is at index $2*i+1$
  - Right child is at index $2*i+2$

- Because of the way heaps are built, there isn't much wasted storage!
  - You'll see heaps again in CS240

LOL vs. heap: Which PQ is best?

<table>
<thead>
<tr>
<th>PQ implementation</th>
<th>enterPQ</th>
<th>leavePQ</th>
<th>firstPQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>List-of-lists (LOL)</td>
<td>$O(K)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Heap</td>
<td>$O(\log_2 N)$</td>
<td>$O(\log_2 N)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

- Assume $N$ elements, $K$ distinct priorities, similar freq of enterPQ/leavePQ
  - If $\log_2 N \gg K$,
  - If $\log_2 N \ll K$,

- If we have 1,000,000 elements, then $\log_2 N = 20$
  - So unless there are very few priorities, (i.e., $K$ is very small), the heap is likely to be the best choice

3. The C++ memory model, linked structures, and some ADTs

CS138 Winter 2017

Prof. Mike Godfrey
University of Waterloo