What's a pointer, why good, why bad?

- Pointer is a memory address, it's an indirect reference to memory or an object.
  - Rather than an int, we say we have a pointer to an int
  - If x is an int, xptr can be a pointer to x
    - Same thing works with Date, Dice, Student, ...
    - Not much use to have pointer to int except in C to understand arrays, but pointers to objects are very useful

- Pointers may force us to think about the machine and memory
  - With great power comes great responsibility
  - Knowledge is powerful, but freedom from it liberating

- Pointers allow us to work at a lower level, but permit inheritance and a higher level of design/programming
  - Built-in array and tvector, C-style string and <string>
Pointers, Memory, Abstractions

- A pointer is the a variable/value that is a memory address
  - Addresses like 1, 2, 3, …, 0x0024ab03
    - Hexadecimal or base-16 digit represents 4 bits
    - Character is 8 bits, integer is 32 bits, double 64 bits (ymmv)
  - Every variable is stored somewhere in memory, typically we can ignore where

```
double x = 32.6; // 0x00 0x08 0x0c 0x??
int y = 18; // 32.6 18
string s = "hello"; // 0x??
```

- The string variable s may be "same size" as double x
  - Storage for the letters is elsewhere, string references it, so memory used by string is more, though size of s isn't
Pointers, Heap, Copies

- Memory allocated statically (auto) vs. on the dynamically (heap)
  - Static = auto = stack
  - Dynamic = heap

```cpp
date ghd(2,2,2003);
date * foolptr = new Date(4,1,2003);
date * x = foolptr;
date y = ghd;
```

- Objects are copied in C++
  - Semantics: copy, don't share
- Pointers are copied (object not)
  - Semantics: object not copied, object is shared
Pointer basics and terminology

- new, dereference, selector operator, copy semantics

```cpp
CD c1("Beatles", "Rubber Soul", 1965);
CD c2("Nirvana", "Nevermind", 1991);
CD * c3 = new CD("REM", "Reveal", 2001);
CD * c4;  // what is the value of c4?
CD c5;    // what is the value of c5?
cout << c1.title() << endl;
cout << c3->title() << endl;
cout << (*c3).title() << endl;
c5 = c2;  c2.changeTitle("Incesticide");
cout << c5.title() << endl;
c4 = c3;  c3->changeTitle("Out of Time");
cout << c4->title() << endl;
```

- What happens if we print `c4->title()` on first line? Why?
What's the point? (e.g., sharing)

- What's the difference between a vector of Dates and a vector of pointers to Dates? What about Courses, Students, etc.?

```cpp
#include <vector>

typedef std::vector<Date> tv;
typedef std::vector<Date*> tvp;

// Which takes up more space? What are values in vectors?

// What happens when we write
tv[0] = tv[2];  // if we change tv[2], affect tv[0]?
tvp[0] = tvp[3]; // change *(tvp[3]), affect tvp[0], *tvp[0]?
```

- Consider example of sorting by both name and age
  - Should we have two vectors of students?
  - Should we have two vectors of student pointers?
  - Is there a reason to prefer one to the other?
The trouble with pointers

- **Don't use the address-of operator, &**

```cpp
Dice * makeDie(int sides) {
    return new Dice(sides);
}
Dice * makeDie(int sides) {
    Dice d(sides);
    return &d;
}
```

- **What about the code below with different versions?**

```cpp
Dice * cube = makeDie(4);
cout << cube->NumSides() << endl;
```

- **Pointer Advice**
  - Always initialize pointer variables, 0/NULL or new
    - 0/NULL means errors are reproduceable
    - Possible to assign another pointer value too
  - Never use the address-of operator
  - Don't call new unless you want another object allocated
Constructors/Destructors

- Every object created must be constructed
  - If no constructor is provided, one is provided for you
  - If you have a non-default constructor, the default-default constructor is *not* automatically provided
- When subclass object constructed, parent and up are too
  - Parent objects can be implicitly constructed via default constructor
  - Alternatively, explicit constructor must be called and it must be called in an initializer list
- Constructors initialize state and allocate resources
  - Resources can be dynamic objects, files, sockets, ...
  - Who (or what) de-allocates resources?
Destructors and Delete

- Objects are (or should be at most times) destructed when they’re no longer accessible or used
  - For static/automatic variables this happens when object goes out of scope
  - For heap-allocated variables this happens when the delete operator (analog of new) is called on a pointer to an object

```cpp
Student * s = new Student("Joe");
delete s; // return storage to heap
```

- When object is destructed, the destructor function is called automatically: `Foo::~Foo() {...}

- It’s easy to mess up when deleting, can’t delete the same object twice, can’t delete an object not allocated by new, ...
  - Yahoo story on never calling delete: too many problems!
ADTs, vectors, linked-lists: tradeoffs?

- `tvctor` is a class-based implementation of a lower-level data type called an array (compatible with STL/standard vector)
  - `tvctor` grows dynamically (doubles in size as needed) when elements inserted with `push_back`
  - `tvctor` protects against bad indexing, vector/arrays don’t
  - `tvctor` supports assignment: `a = b`, arrays don’t

- As an ADT (abstract data type) vectors support
  - *Constant-time* or O(1) access to the k-th element
  - *Amortized* linear or O(n) storage/time with `push_back`
    - How?

- Adding a new value in the middle of a vector is expensive, linear or O(n) because shifting required
Cost

“An engineer is someone who can do for a dime what any fool can do for a dollar.”

- Types of costs:
  - Operational
  - Development
  - Failure
- Is this program fast enough? What’s your purpose? What’s your input data?
- How will it scale?
- Measuring cost
  - Wall-clock or execution time
  - Number of times certain statements are executed
  - Symbolic execution times
    - Formula for execution time in terms of input size
  - Advantages and disadvantages?

Some complexity notes courtesy of Paul Hilfinger
Data processing example

- Scan a large (~ $10^7$ bytes) file
- Print the 20 most frequently used words together with counts of how often they occur
- Need more specification?

- How do you do it?
Possible solutions

1. **Use heavy duty data structures (Knuth)**
   - Hash tries implementation
   - Randomized placement
   - Lots o’ pointers
   - Several pages

2. **UNIX shell script (Doug McIlroy)**
   ```bash
   tr -cs "[:alpha:]" "[\n*]" < FILE | \
   sort | \
   uniq -c | \
   sort -n -r -k 1,1 | \
   head -20
   ```
   - **Which is better?**
     - K.I.S.?
What is big-Oh about? (preview)

- Intuition: avoid details when they don’t matter, and they don’t matter when input size (N) is big enough
  - For polynomials, use only leading term, ignore coefficients

\[ y = 3x \quad y = 6x - 2 \quad y = 15x + 44 \]
\[ y = x^2 \quad y = x^2 - 6x + 9 \quad y = 3x^2 + 4x \]

- The first family is \( O(n) \), the second is \( O(n^2) \)
  - Intuition: family of curves, generally the same shape
  - More formally: \( O(f(n)) \) is an upper-bound, when \( n \) is large enough the expression \( cf(n) \) is larger
  - Intuition: linear function: double input, double time, quadratic function: double input, quadruple the time
More on O-notation, big-Oh

- Big-Oh hides/obscures some empirical analysis, but is good for general description of algorithm
  - Allows us to compare algorithms *in the limit*
    - 20N hours vs \( N^2 \) microseconds: *which is better?*

- O-notation is an upper-bound, this means that \( N \) is \( O(N) \), but it is also \( O(N^2) \); we try to provide *tight* bounds.

Formally:
  - A function \( g(N) \) is \( O(f(N)) \) if there exist constants \( c \) and \( n \) such that \( g(N) < cf(N) \) for all \( N > n \)
Big-Oh calculations from code

- **Search for element in vector:**
  - What is complexity of code (using O-notation)?
  - What if array doubles, what happens to time?

```cpp
for(int k=0; k < a.size(); k++) {
    if (a[k] == target) return true;
}
return false;
```

- **Complexity if we call N times on M-element vector?**
  - What about best case? Average case? Worst case?
Big-Oh calculations again

- Consider weekly problem 2: first element to occur 3 times
  - What is complexity of code (using O-notation)?

```cpp
for(int k=0; k < a.size(); k++) {
    int count = 1;
    for(int j=0; j < k; k++) {
        if (a[j] == a[k]) count++;
    }
    if (count >= 3) return a[k];
}
return ""; // no one on probation
```

- What if we initialize counter to 0, loop to <= k?
- What is invariant describing value stored in count?
- What happens to time if array doubles in size?
Big-Oh calculations again

- Add a new element at front of vector by shifting, complexity?
  - Only count vector assignments, not vector growing

```cpp
a.push_back(newElement); // make room for it
for(int k=a.size()-1; k >=0; k--) {
    a[k+1] = a[k]; // shift right
}
a[0] = newElement;
```

- If we call the code above N times on an initially empty vector, what’s the complexity using big-Oh?

- Now, what about complexity of growing a vector
  - If vector doubles in size? If vector increases by one?
Nested loops

- Nested loops often lead to *polynomial* bounds
  ```c
  for (int i=0; i < v.size(); i++)
      for (int j=0; j < v.size(); j++)
          if (i != j && v[i] == v[j])
              return true;
  return false;
  ```
- Time is $O(n^2)$ where $n = v.size()$
- Loop is inefficient though
  ```c
  for (int i=0; i < v.size(); i++)
      for (int j=i+1; j < v.size(); j++)
          if (v[i] == v[j]) return true;
  return false;
  ```
- New worst case time?
Amortization: Expanding Vectors

- How do we expand the capacity of a vector as the size grows?
- Cost of calling `push_back` N times, doubling capacity as needed

<table>
<thead>
<tr>
<th>Item #</th>
<th>Resizing cost</th>
<th>Cumulative cost</th>
<th>Resizing Cost per item</th>
<th>Capacity After push_back</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3-4</td>
<td>4</td>
<td>6</td>
<td>1.5</td>
<td>4</td>
</tr>
<tr>
<td>5-8</td>
<td>8</td>
<td>14</td>
<td>1.75</td>
<td>8</td>
</tr>
</tbody>
</table>

\[
2^{m+1} - 2^m, \quad 2^m, \quad 2^{m+2} - 2, \quad \text{around } 2, \quad 2^{m+1}
\]

- Worst case for one element? for N elements?
Some helpful mathematics

1. $1 + 2 + 3 + 4 + \ldots + N$
   - $N(N+1)/2$, exactly $= N^2/2 + N/2$ which is $O(N^2)$
     why?

2. $N + N + N + \ldots + N$ (total of $N$ times)
   - $N\times N = N^2$ which is $O(N^2)$

3. $N + N + N + \ldots + N + \ldots + N + \ldots + N$ (total of $3N$ times)
   - $3N\times N = 3N^2$ which is $O(N^2)$

4. $1 + 2 + 4 + \ldots + 2^N$
   - $2^{N+1} - 1 = 2 \times 2^N - 1$ which is $O(2^N)$

5. Impact of last statement on adding $2^{N+1}$ elements to a vector
   - $1 + 2 + \ldots + 2^N + 2^{N+1} = 2^{N+2} - 1 = 4\times 2^N - 1$ which is $O(2^N)$
## Running times @ $10^6$ instructions/sec

<table>
<thead>
<tr>
<th>$N$</th>
<th>$O(\log N)$</th>
<th>$O(N)$</th>
<th>$O(N \log N)$</th>
<th>$O(N^2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000003</td>
<td>0.00001</td>
<td>0.000033</td>
<td>0.0001</td>
</tr>
<tr>
<td>100</td>
<td>0.000007</td>
<td>0.00010</td>
<td>0.000664</td>
<td>0.1000</td>
</tr>
<tr>
<td>1,000</td>
<td>0.000010</td>
<td>0.00100</td>
<td>0.010000</td>
<td>1.0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.000013</td>
<td>0.01000</td>
<td>0.132900</td>
<td>1.7 min</td>
</tr>
<tr>
<td>100,000</td>
<td>0.000017</td>
<td>0.10000</td>
<td>1.661000</td>
<td>2.78 hr</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.000020</td>
<td>1.0</td>
<td>19.9</td>
<td>11.6 day</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>0.000030</td>
<td>16.7 min</td>
<td>18.3 hr</td>
<td>318 centuries</td>
</tr>
</tbody>
</table>
Recursion Review

- Recursive functions have two key attributes
  - There is a *base case*, sometimes called the *exit case*, which does *not* make a recursive call
  - All other cases make recursive call(s), the results of these calls are used to return a value when necessary
    - Ensure that every sequence of calls reaches base case
    - Some measure decreases/moves towards base case
    - “Measure” can be tricky, but usually it’s straightforward

- Example: sequential search in a vector
  - If first element is search key, done and return
  - Otherwise look in the “rest of the vector”
  - How can we recurse on “rest of vector”?
Sequential search revisited

● What is postcondition of code below? How would it be called initially?
  ➢ Another overloaded function search with 2 parameters?

```cpp
bool search(const vector<string>& v, 
            int index, const string& target)
{
    if (index >= v.size()) return false;
    else if (v[index] == target) return true;
    else return search(v, index+1, target);
}
```

● What is complexity (big-Oh) of this function?
Recursion and Recurrences

```cpp
bool occurs(string s, string x)
{
    // post: returns true iff x is a substring of s
    if (s == x) return true;
    if (s.length() <= x.length()) return false;
    return occurs(s.substr(1, s.length()-1), x) ||
           occurs(s.substr(0, s.length()-1), x);
}
```

- In worst case, both calls happen
- Say $C(N)$ is the worst case cost of $\text{occurs}(s, x), N == s\text{.length()}$
  - $C(N) = 1$ if $N < x\text{.length()}$
  - $C(N) = 2C(N-1)$ if $N > x\text{.length()}$
- What is $C(N)$?
Binary search revisited

```cpp
bool bsearch(const vector<string>& v, 
              int start, int end, 
              const string& target)
{
    // base case
    if (start == end) return v[start] == target;

    int mid = (start + end)/2;
    if (v[mid] == target)    // found target
        return true;
    else if (v[mid] < target) // target on right
        return bsearch(v, mid +1, end);
    else                        // target on left
        return bsearch(v, start, mid -1);
}
```

● What is the big-Oh of bsearch?
Why we study recurrences/complexity?

- Tools to analyze algorithms
- Machine-independent measuring methods
- Familiarity with good data structures/algorithms

- What is CS person: programmer, scientist, engineer?
  *scientists build to learn, engineers learn to build*

- Mathematics is a notation that helps in thinking, discussion, programming
The Power of Recursion: Brute force

- Consider weekly problem 5: What is minimum number of minutes needed to type n term papers given page counts and three typists typing one page/minute? (assign papers to typists to minimize minutes to completion)
  - Example: \{3, 3, 3, 5, 9, 10, 10\} as page counts

- How can we solve this in general? Suppose we're told that there are no more than 10 papers on a given day.
  - How does the constraint help us?
  - What is complexity of using brute-force?
Recasting the problem

- Instead of writing this function, write another and call it

```cpp
int bestTime(const tvector<int>& v)
// post: returns min minutes to type papers in v
{
    return best(v,0,0,0,0);
}
```

- What cases do we consider in function below?

```cpp
int best(const tvector<int>& v, int index,
    int t1, int t2, int t3)
// post: returns min minutes to type papers in v
// starting with index-th paper and given
// minutes assigned to typists, t1, t2, t3
{
}
```