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
  - 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

\[
\begin{array}{cccc}
0x00 & 0x08 & 0x0c & 0x00 \\
32.6 & 18 & "hello" & \text { ??}
\end{array}
\]

- 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

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
tvector<Date> tv(1000);
tvector<Date *> tvp(1000);
```

- Which takes up more space? What are values in vectors?
- What happens when we write
  ```cpp
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:
  ```cpp
  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?

- `tvector` is a class-based implementation of a lower-level data type called an array (compatible with STL/standard vector)
  - `tvector` grows dynamically (doubles in size as needed) when elements inserted with `push_back`
  - `tvector` protects against bad indexing, vector/arrays don’t
  - `tvector` 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`
    - Total storage used in n-element vector is approx. 2n, spread over all accesses/additions (why?)
- Adding a new value in the middle of a vector is expensive, linear or O(n) because shifting required
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

\[
\begin{align*}
y &= 3x & y &= 6x-2 & y &= 15x + 44 \\
y &= x^2 & y &= x^2-6x+9 & y &= 3x^2+4x
\end{align*}
\]

- The first family is \( \mathcal{O}(n) \), the second is \( \mathcal{O}(n^2) \)
  - Intuition: family of curves, generally the same shape
  - More formally: \( \mathcal{O}(f(n)) \) 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 N² 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?

```c
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)?

```java
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 $\leq k$?
- 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?
Some helpful mathematics

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

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

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

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

- 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>0.0001</td>
<td>0.000033</td>
<td>0.0001</td>
</tr>
<tr>
<td>100</td>
<td>0.000007</td>
<td>0.0010</td>
<td>0.000664</td>
<td>0.1000</td>
</tr>
<tr>
<td>1,000</td>
<td>0.000010</td>
<td>0.0100</td>
<td>0.010000</td>
<td>1.0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.000013</td>
<td>0.0100</td>
<td>0.132900</td>
<td>1.7 min</td>
</tr>
<tr>
<td>100,000</td>
<td>0.000017</td>
<td>0.1000</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?
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
{
}
```