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
  - Every variable is stored somewhere in memory, typically we can ignore where

\[
\begin{array}{c}
\text{double } x = 32.6;  \\
\text{int } y = 18;  \\
\text{string } s = "hello";
\end{array}
\]

- The string variable s is actually the same size as int y
  - Storage for the letters is elsewhere, string references it

- What about a permanent, but forwardable email address?
- Other kinds of references?
Pointers

• Using pointers presents some of the same problems as being Spiderman
  ➢ With great power comes great responsibility…
  ➢ New, more ugly, and harder to detect bugs
• Why pointers?
  ➢ Allow different sections of code to share data easily
  ➢ Enable linked data structures
• The BIG pointer concepts
  ➢ Pointer declaration
  ➢ NULL (0) pointer
  ➢ BAD (uninitialized) pointers
  ➢ Memory allocation from heap
  ➢ Pointer dereference
  ➢ Pointer assignment
void main() {
  int *x; // Allocate the pointers x and y
  int *y; // but not the _pointee_

  x = new int;

  *x = 42; // Dereference x - store 42 in pointee
  *y = 13; // CRASH!

  y = x; // Pointer assignment
  *y = 13; // Store 13 in its (shared) pointee
}

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 say we have 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, but pointer to class!!

- Pointers force us to think about the machine and memory
  - Knowledge is powerful, 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>
Pointer example

- 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?
c5 = c2; c2.changeTitle("Incesticide");
```

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

- 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?
Thinking about pointsort.cpp

- The class Group uses a `tvector<Student> myList`
  - What changes if this is `tvector<Student *> myList`?
    - Changes to Group::add
    - Changes to Group::print
  - Other changes needed?

- What if we want to sort by age to print, leaving original order the same (why would we want to do this?)
  - Use another vector, sort it differently
  - Why is another vector a good idea?
  - Could use vector of indexes
    - Both are indirect references

```plaintext
21 19 18 20
```
Local memory is ephemeral...

// -- The Ampersand Bug function
// Returns a pointer to an int TAB
int* TAB()
{
    int temp;
    return(&temp); // return a pointer to the local int
}
void Victim()
{
    int* ptr;
    ptr = TAB();
    *ptr = 42; // Runtime error! Pointee local to TAB
}
The trouble with pointers

- **Local vs. Heap allocation**

- **Another address of bug**

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

- **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
  - Don't call new unless you want another object allocated