CS 106X, Lecture 29
Life After CS 106X

Plan for today

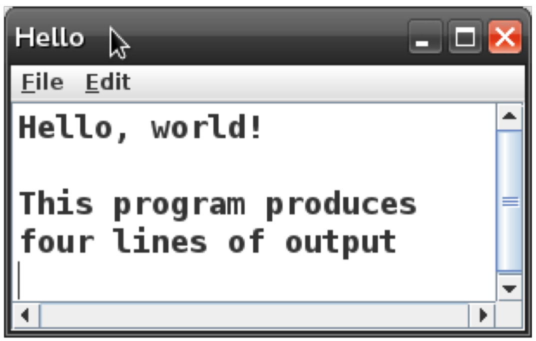
- The Stanford Libraries
  - User Input/Output
  - Collections
  - Graphics
- Memory Management
- **Announcements**
- Other Languages

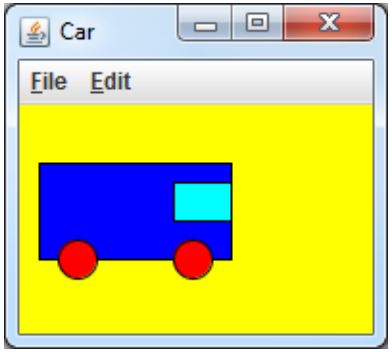
Plan for today

- **The Stanford Libraries**
  - User Input/Output
  - Collections
  - Graphics
- Memory Management
- **Announcements**
- Other Languages

The Stanford Libraries

- All quarter we have relied on the **Stanford C++ libraries**.
  - `GWindow`, `SimpIO`, `Vector`, `Set`, `Grid`, `BasicGraph`...





- How are C++ programs written *without* the Stanford libraries?

Plan for today

- **The Stanford Libraries**
  - **User Input/Output**
  - Collections
  - Graphics
- Memory Management
- **Announcements**
- Other Languages

Demo

The Stanford Libraries

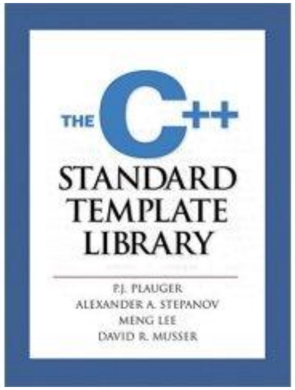
- What do the Stanford Libraries do?
  - Creates a new graphical **window**
  - Puts a scrollable **text area** into it (for text programs)
  - Redirects `cin`, `cout` and `cerr` commands to go through that window
  - contains a **`main` method** that calls your program class's `main` method
  - Contains helpful functions such as `getInteger`

Plan for today

- **The Stanford Libraries**
  - User Input/Output
  - **Collections**
  - Graphics
- Memory Management
- **Announcements**
- Other Languages

STL

- **Standard Template Library (STL)**: A set of classes and algorithms for C++, many of which use templates.
  - **container** classes (collections)
  - algorithms
  - functional programming
  - iterators



- Stanford C++ library collections largely duplicate ones from STL, but make much of the functionality easier to use.

Stanford → STL

Stanford C++ lib	STL
`Graph`	-
`Grid`	- *(use a 2D array)*
`HashMap`	`unordered_map` *(C++11)*
`HashSet`	`unordered_set` *(C++11)*
`Lexicon`	-
`LinkedList`	`list`
`Map`	`map`
`Set`	`set`
`PriorityQueue`	`priority_queue`
`Queue`	`queue` `deque` *(double-ended queue)*
`Stack`	`stack`
`Vector`	`vector`
others	`array`, `bitset`, `multiset`, `multimap`

Vector → vector

Stanford C++ lib	STL
Vector	vector
add	**push_back**
clear	clear
get (or [])	at (or [])
insert	emplace
isEmpty	**empty**
remove	**erase**
set	assign
size	size
toString	-
==, !=	==, !=

Stanford Vector

- Using a vector to store a sequence of integers:

```
#include "vector.h"

// add five integers
Vector<int> v;
for (int i = 1; i <= 5; i++) {     // {2, 4, 6, 8, 10}
    v.add(i * 2);
}

// insert an element at the start
v.insert(0, 42);                   // {42, 2, 4, 6, 8, 10}

//delete the third element
v.remove(2);                       // {42, 2, 6, 8, 10}
```

STL vector

- Using a vector to store a sequence of integers:

```
#include <vector>

// add five integers
vector<int> v;
for (int i = 1; i <= 5; i++) {    // {2, 4, 6, 8, 10}
    v.push_back(i * 2);
}

// insert an element at the start
v.insert(v.begin(), 42);          // {42, 2, 4, 6, 8, 10}

//delete the third element
v.erase(v.begin() + 2);           // {42, 2, 6, 8, 10}
```

Stanford Map

- Using a map to store prices of groceries:

```
#include "map.h"

// add some key/value pairs
Map<string, double> price;
price["snapple"] = 0.75;
price["coke"] = 0.50;

// read from the console and access the map
string item;
double total = 0;
while (cin >> item) {
    total += price[item];
}

// does map contain "coke"?
if (price.containsKey("coke")) { ... }
```

STL map

- Using a map to store prices of groceries:

```
#include <map>

// add some key/value pairs
map<string, double> price;
price["snapple"] = 0.75;
price["coke"] = 0.50;

// read from the console and access the map
string item;
double total = 0;
while (cin >> item) {
    total += price[item];
}

// does map contain "coke"?
if (price.find("coke") != price.end()) { ... }
```

STL Iterators

- An iterator is like a pointer to an element inside a collection.
- Bundles together position and data
- Used across many collections

Iterator example

```
// looping over the elements of a vector
vector<int> v;
...
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it){
    cout << *it << endl;
}
```

- Each collection has a `begin` and `end` iterator to its front/back.
- Iterators use "pointer-like" syntax *(operator overloading)*:

  `++`*`itr`* to advance by 1 element; `--`*`itr`* to go back

  `*`*`itr`* to access the element the iterator is currently at

```
// shorter version, for-each loop and implicit iterator (C++11)
for (int k : v) {
    cout << k << endl;
}
```

Iterator example

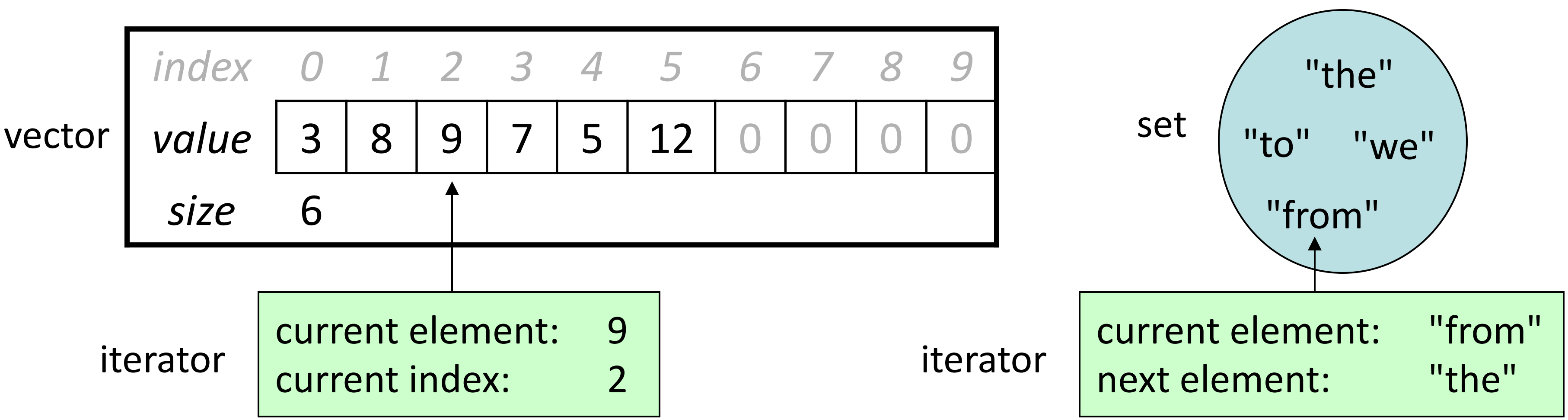
```
// looping over the elements of a vector
vector<int> v;
...
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it){
    cout << *it << endl;
}
```

Iterator example

```
// looping over the elements of a set
set<int> v;
...
for (set<int>::iterator it = v.begin(); it != v.end(); ++it) {
    cout << *it << endl;
}
```

STL iterators

- Iterators provide a consistent way to interface with collection elements; even if they are not indexed!



More iterators

- Many container members accept an iterator to indicate position.
  - example: vector's `insert`, `erase`, `assign`, etc.
  - most of these members return a new iterator, must re-assign!

```
// remove all odd numbers from a vector (iterating backwards)
vector<int> v;
...
for (vector<int>::iterator it = v.end(); it != v.begin(); --it) {
    if (*it % 2 != 0) {      // odd
        it = v.erase(it);   // delete element at this position
    }
}
```

  - Most Stanford collections also have `begin()` and `end()` members to return iterators that behave the same way, in an effort to be compatible with STL.

STL algorithms

- A huge collection of useful functions and algorithms that accept containers (or, more commonly, iterators) as parameters:

```
adjacent_find    generate_n                  move                 replace_if
all_of           includes                    move_backward        reverse
any_of           inplace_merge               next_permutation     reverse_copy
binary_search    is_heap                     none_of              rotate
copy             is_heap_until               nth_element          rotate_copy
copy_backward    is_partitioned              partial_sort         search
copy_if          is_permutation              partial_sort_copy    search_n
copy_n           is_sorted                   partition            set_difference
count            is_sorted_until             partition_copy       set_intersection
count_if         iter_swap                   partition_point      set_union
equal            lexicographical_compare     pop_heap             shuffle
equal_range      lower_bound                 prev_permutation     sort
fill             make_heap                   push_heap            sort_heap
fill_n           max                         random_shuffle       stable_partition
find             max_element                 remove               stable_sort
find_end         merge                       remove_copy          swap
find_first_of    min                         remove_copy_if       swap_ranges
find_if          min_element                 remove_if            transform
find_if_not      minmax                      replace              unique
for_each         minmax_element              replace_copy         unique_copy
generate         mismatch                    replace_copy_if      upper_bound
```

STL algorithm examples

- Most STL algorithms operate on iterators (why?). Sort a vector:

```
#include <algorithm>
sort(v.begin(), v.end());
```

- Count occurrences of a value in a set:

```
int zachs = count(s.begin(), s.end(), “Zach");
```

- Find the largest element value in a vector:

```
int biggest = *max_element(v.begin(), v.end()); // note the *
```

- Copy the last 5 elements from v1 to the start of v2:

```
copy(v1.begin(), v1.begin() + 5, v2.begin());
```

Aside: auto

```
for (vector<int>::iterator it = v.end(); it != ...

for (auto it = v.end(); it != v.begin(); --it) {
    …
}
```

```
auto name = value;
```

auto tells C++ to infer the type of a variable automatically!

- Pro: lets C++ handle types, abbreviates programs, easier to code
- Con: harder to tell variable types, may lead to harder debugging

Aside: typedef

```
typedef longTypeName shortTypeName;
```

- **typedef**: Gives a nickname/shorthand to a data type.

```
// long type name!
std::map<std::string,
         std::map<std::string, double>>::iterator

// shorter with typedef
typedef std::map<std::string, double> HWToGrade;
typedef std::map<std::string, HWToGrade> StudentToHWMap;

StudentToHWMap myMap;
myMap["Adam"]["HW3"] = 89.0;
```

So what's the problem?

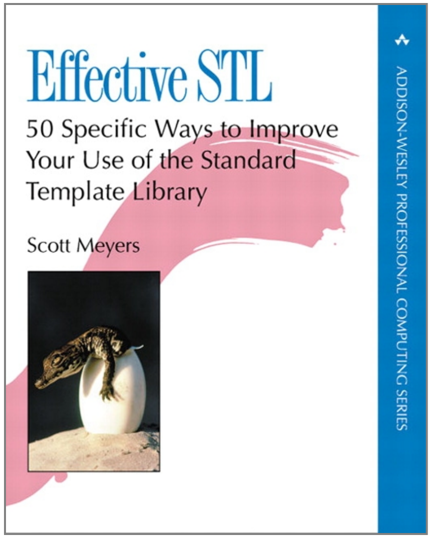
- STL seems useful and powerful. Why didn't we just use it?
  - requires heavy use of **pointers** and pointer syntax early (iterators)
  - **iterators** can be hard to use and understand at first
  - some algorithms require understanding **function pointers**
  - STL emits very confusing **syntax error** messages on bad code
  - some STL classes are bloated and confusing
  - some STL classes are missing important features we wanted
    - can't just use integer indexes to do things on a vector; argh!
    - set doesn't have a `contains` member; collections don't have `toString`
    - no Lexicon (trie) type; no Graph; no Grid; etc.
  - missing hash-based sets and maps *(until C++11)*
  - bad runtime error/crash messages if you do wrong things
    - (e.g. access past end of a vector; does not do bounds-checking)

So what's the problem?

- STL seems useful and powerful. Why didn't we just use it?
  - requires heavy use of **pointers** and pointer syntax early (iterators)
  - **iterators** can be hard to use and understand at first
  - some algorithms require understanding **function pointers**
  - STL emits very confusing **syntax error** messages on bad code
  - some STL classes are bloated and confusing
  - some STL classes are missing important features we wanted
    - can't just use integer indexes to do things on a vector; argh!
    - set doesn't have a `contains` member; collections don't have `toString`
    - no Lexicon (trie) type; no Graph; no Grid; etc.
  - missing hash-based sets and maps *(until C++11)*
  - bad runtime error/crash messages if you do wrong things
    - (e.g. access past end of a vector; does not do bounds-checking)

To learn more, ...

- Buy *Effective STL*, by Scott Meyers
- read online C++ / STL references
  - cplusplus.com
  - cppreference.com
  - Wikipedia: STL



- try re-writing 106B/X assignments using STL!
  - Can you implement them without using any functionality from the Stanford libraries? (aside from maybe the overall GUI)

- take **CS 106L** or look at their materials

Plan for today

- **The Stanford Libraries**
  - User Input/Output
  - Collections
  - **Graphics**
- Memory Management
- **Announcements**
- Other Languages

Stanford Library GUI

- The Stanford Libraries provide many useful features:
  - Easily create and display a graphical window
  - Draw shapes on a canvas
  - Add interactive elements (text boxes, buttons, checkboxes,…)
  - And more…

- In C++, there is no single way to display GUIs:
  - QT
  - Motif
  - FLTK
  - Ncurses
  - More…

Libraries

- **Benefits of libraries:**
  - simplify syntax/rough edges of language/API
  - avoid re-writing the same code over and over
  - possible to make advanced programs quickly
  - leverage work of others

- **Drawbacks of libraries:**
  - learn a "dialect" of the language (”Stanford C++" vs. "real C++")
  - lack of understanding of how lower levels or real APIs work
  - some libraries can be buggy or lack documentation
  - limitations on usage; e.g. Stanford libraries cannot be re-distributed for commercial purposes

Plan for today

- The Stanford Libraries
  - User Input/Output
  - Collections
  - Graphics
- **Memory Management**
- **Announcements**
- Other Languages

Smart pointers

- **smart pointer**: A stack-allocated container that can store a pointer to data on the heap and free it automatically later.
  - added to C++ in the C++11 version of the language
  - prior to this, many coders used **Boost** library or others

- C++ smart pointer types: `#include <memory>`

```
unique_ptr      // exactly 1 "owner"; best one
shared_ptr      // multiple "owners"; use sparingly
weak_ptr        // use sparingly
auto_ptr        // deprecated; do not use!
```

  - common concept: notion of "**ownership**" of a heap-allocated pointer; who is responsible for deleting/freeing it later?

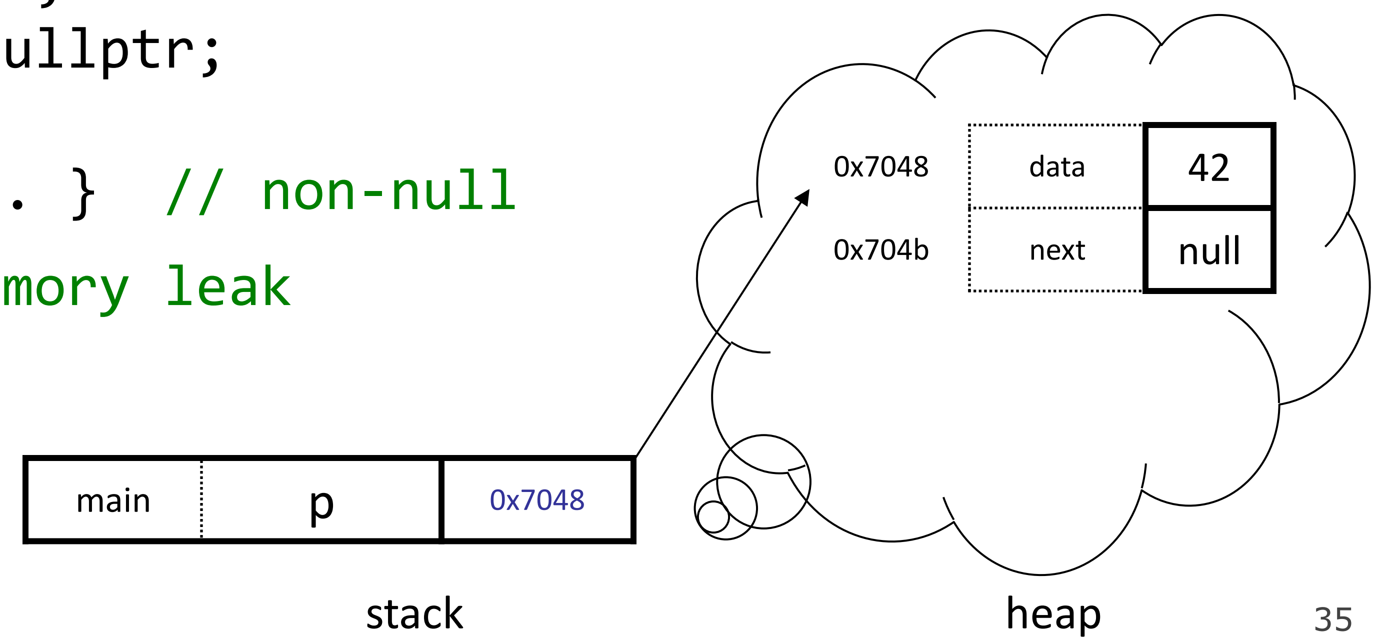
unique_ptr

`unique_ptr<T> name(heapObject);`

- **unique_ptr** is a stack-allocated container that "owns" and manages the memory for a heap-allocated object.
  - You can use the unique_ptr generally the same way that you use a normal pointer. (It overrides operators like *, `->`, `++`, `--`)
  - When the unique_ptr falls out of scope, it automatically deletes the heap memory it is owning.
  - No more than one unique_ptr can own a given object at a time. *(hence the name "unique")*

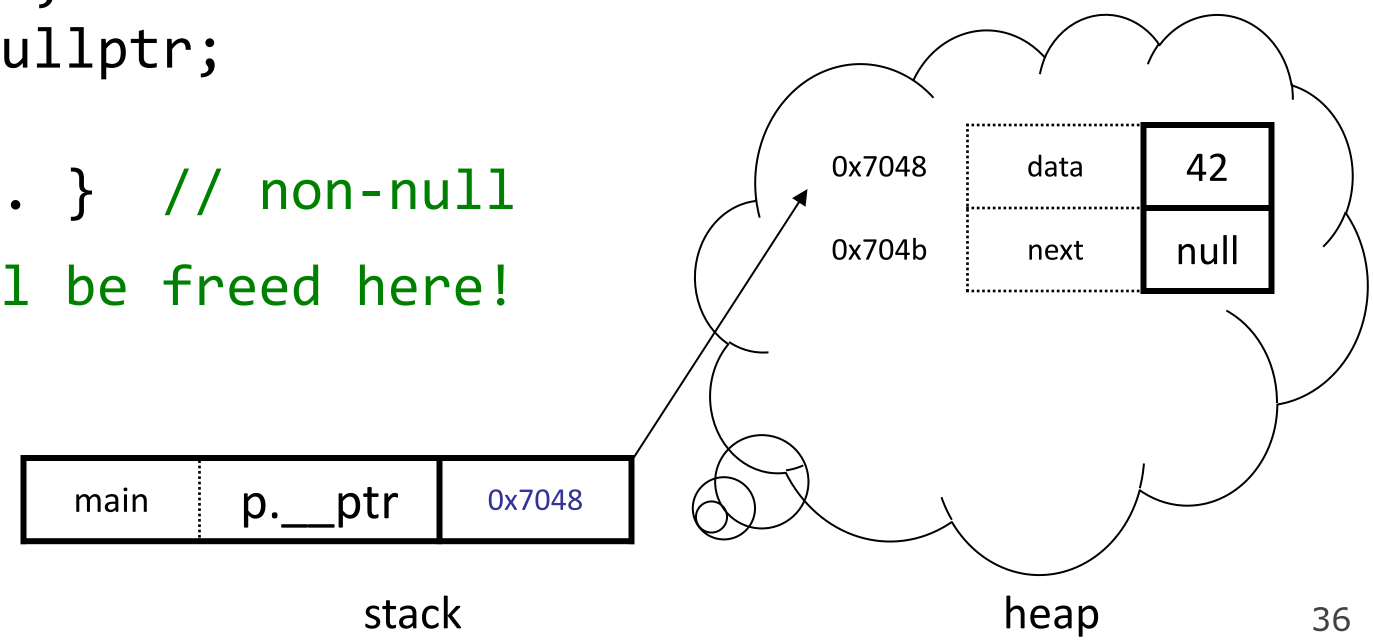
Normal pointer usage

```
void foo() {
    ListNode* p = new ListNode();
    p->data = 42;
    p->next = nullptr;
    ...
    p = new ListNode();   // oops, memory leak!
    p->data = 19;
    p->next = nullptr;
    ...
    if (p) { ... }  // non-null
    // oops, memory leak
}
```



unique_ptr usage

```
void foo() {
    unique_ptr<ListNode> p(new ListNode());
    p->data = 42;           // access underlying pointer
    p->next = nullptr;      // using -> operator
    ...
    p.reset(new ListNode());   // frees prior node
    p->data = 19;
    p->next = nullptr;
    ...
    if (p) { ... }  // non-null
    // node will be freed here!
}
```



get and release

```
void foo() {
    unique_ptr<ListNode> p(new ListNode());
    ...
    // return raw pointer;  p still "owns" it
    // (and will free it later at end of p's scope)
    ListNode* raw1 = p.get();

    // return raw pointer; p stops owning it
    // and won't free it any more (up to you now)
    ListNode* raw2 = p.release();

    // node will not be freed here
}
```

unique_ptr as parameter

- Cannot pass a `unique_ptr` as a parameter nor = assign it.

```
// does not compile
void foo(unique_ptr<ListNode> p) {
    ...
}

int main() {
    unique_ptr<ListNode> p(new ListNode());
    foo(p);   // does not work
    unique_ptr<ListNode> p2(new ListNode());
    p = p2;   // does not compile  (operator= disabled)
    return 0;
}
```

move()

- The move function transfers smart pointer ownership.

```
// this version does compile!
void foo(unique_ptr<ListNode> p) {
    ...
}

int main() {
    unique_ptr<ListNode> p(new ListNode());
    foo(move(p));

    // can't use p->... here (p transferred ownership)

    return 0;
}
```

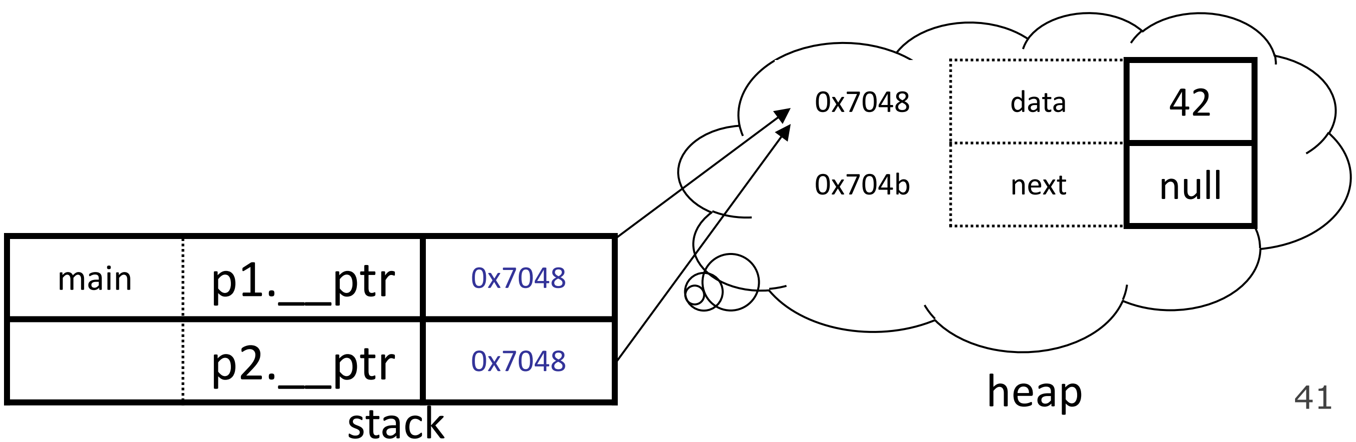
Returning unique_ptr

- C++ allows returning `unique_ptr` because of "move assignment".

```
// this version does compile!
unique_ptr<ListNode> foo() {
    unique_ptr<ListNode> p(new ListNode());
    return p;
}

int main() {
    unique_ptr<ListNode> p = foo();

    // can use p->... here (foo transferred ownership)

    return 0;
}
```

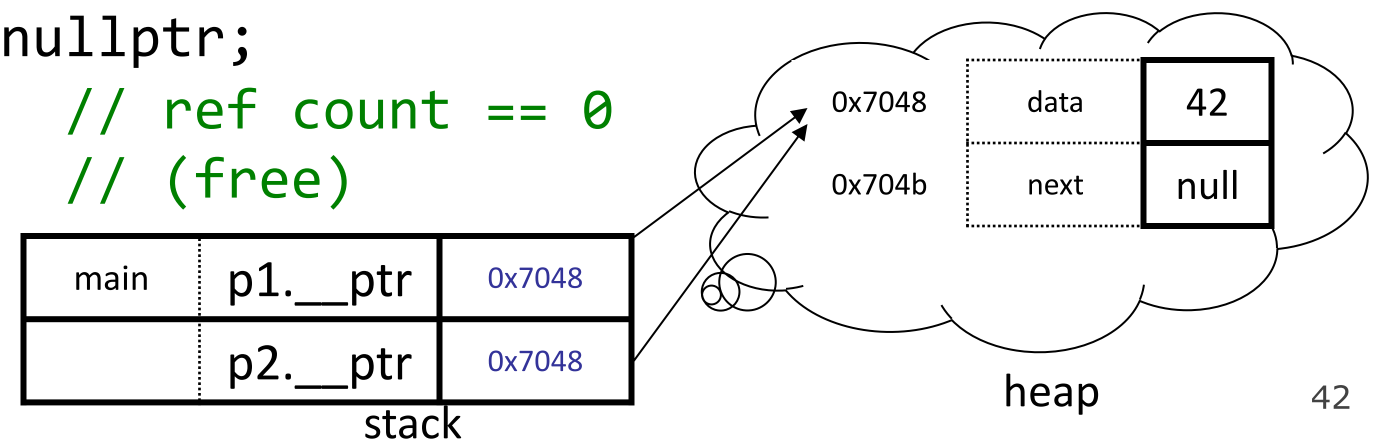
shared_ptr

- `shared_ptr` is like `unique_ptr`, but:
  - Multiple `shared_ptr`s can share "ownership" of same raw pointer.
- **reference count**: Number of shared_ptrs that own a given pointer.
  - As ptr is assigned to a shared_ptr, reference count increases.
  - When a shared_ptr falls out of scope, reference count decreases.
  - If reference count hits 0, pointer is freed.



shared_ptr usage

```
void foo() {
    shared_ptr<ListNode> p1(new ListNode());
    p1->data = 42;                        // ref count == 1
    p1->next = nullptr;
    ...
    shared_ptr<ListNode> p2 = p1;    // ref count == 2
    ...
    p1.reset();                           // ref count == 1

    p2->data = 19;
    p2->next = nullptr;
    p2.reset();  // ref count == 0
}                // (free)
```



Why not shared_ptr?

- `shared_ptr` seems more flexible/powerful than `unique_ptr`. Why not always use it?
  - It lets you be **less clear** about pointer ownership. ("easy" ~= lazy)
  - It is easier to introduce **memory leaks**. (dangling shared pointers)
  - It works poorly with multi-threaded code.

- General software design heuristic:
  You almost never need > 1 owner for an object.
  - If you think you need `shared_ptr`, you may have poor design and may be able to avoid using it by improving your code.

weak_ptr and auto_ptr

- **`weak_ptr`** can be used in conjunction with shared_ptr.
  - Holds a pointer to a shared object, but doesn't "**own**" it.
  - When weak_ptr is created, does *not* increase reference count.
  - When weak_ptr is destroyed, does *not* decrease reference count.

  - Helps avoid common problem/bug called *circular references*.
  - Useful if you want to refer to a shared_ptr temporarily.
  - Sometimes used internally in various collections / data structures.

- `auto_ptr` is an earlier, worse, version of unique_ptr.
  - It is bad; *deprecated* in the language; never use it.

Example SmartPtr class

```
// if we tried to write such a library ourselves ...
template <typename T>
class SmartPointer {
public:
    SmartPointer();
    ~SmartPointer();
    ...
private:
    T* ptr;
};

SmartPointer::SmartPointer(T* ptr = nullptr) {
    ptr = p;
}

SmartPointer::~SmartPointer() {
    if (ptr) { delete ptr; ptr = nullptr; }
}
```

Plan for today

- The Stanford Libraries
  - User Input/Output
  - Collections
  - Graphics
- Memory Management
- **Announcements**
- Other Languages

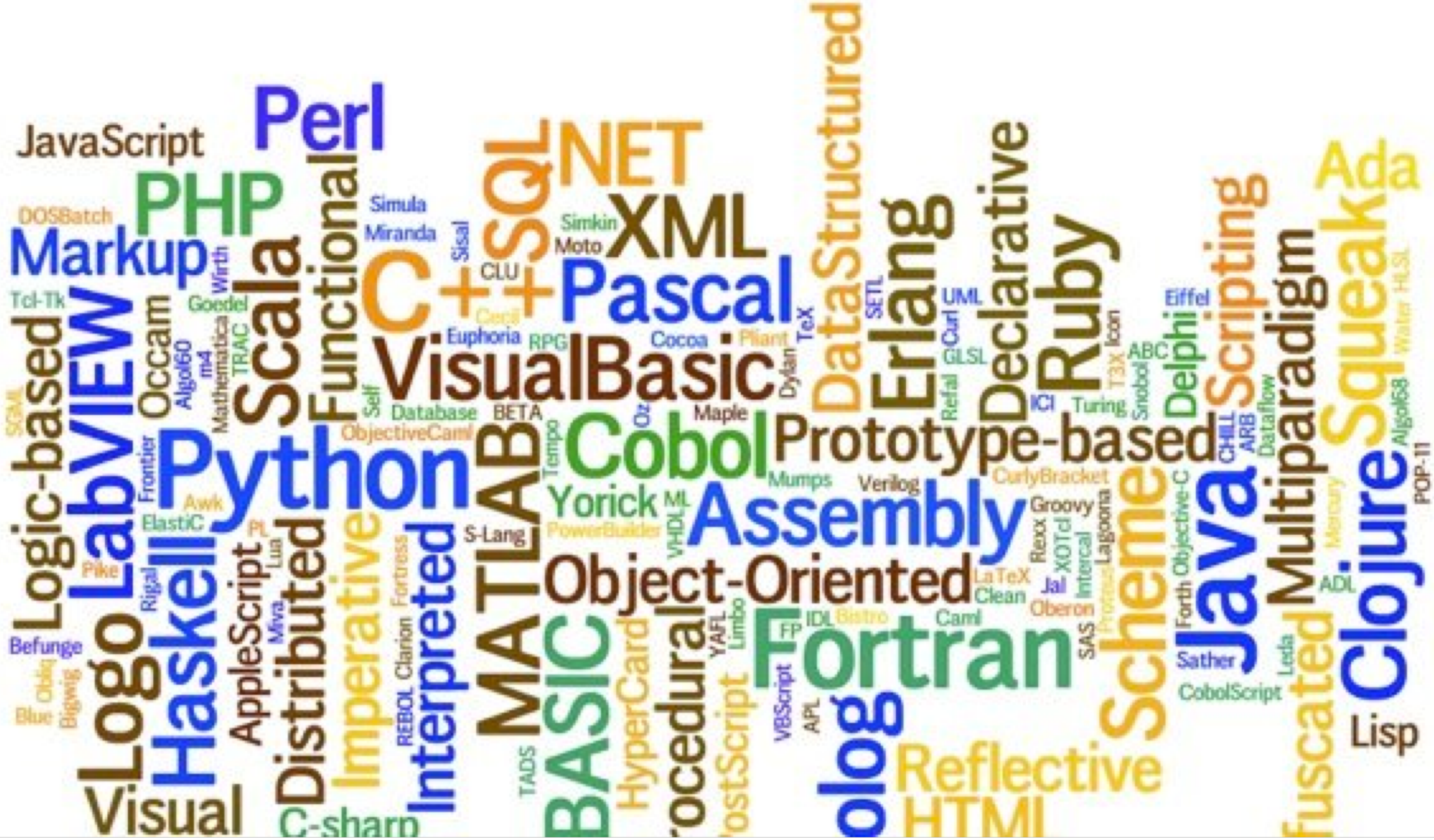
Announcements

- The CS 106X final exam is on ***Monday, Dec. 10 from 8:30AM-11:30AM in 420-041***.
  - review session on **TONIGHT, Dec. 5 from 7-8:30PM in Hewlett 103**.
  - Please notify us of academic accommodations or laptop needs by 5PM today!
- Ask-anything during lecture Friday. Submit questions here! https://goo.gl/forms/jYcH61FsxvooTtPQ2

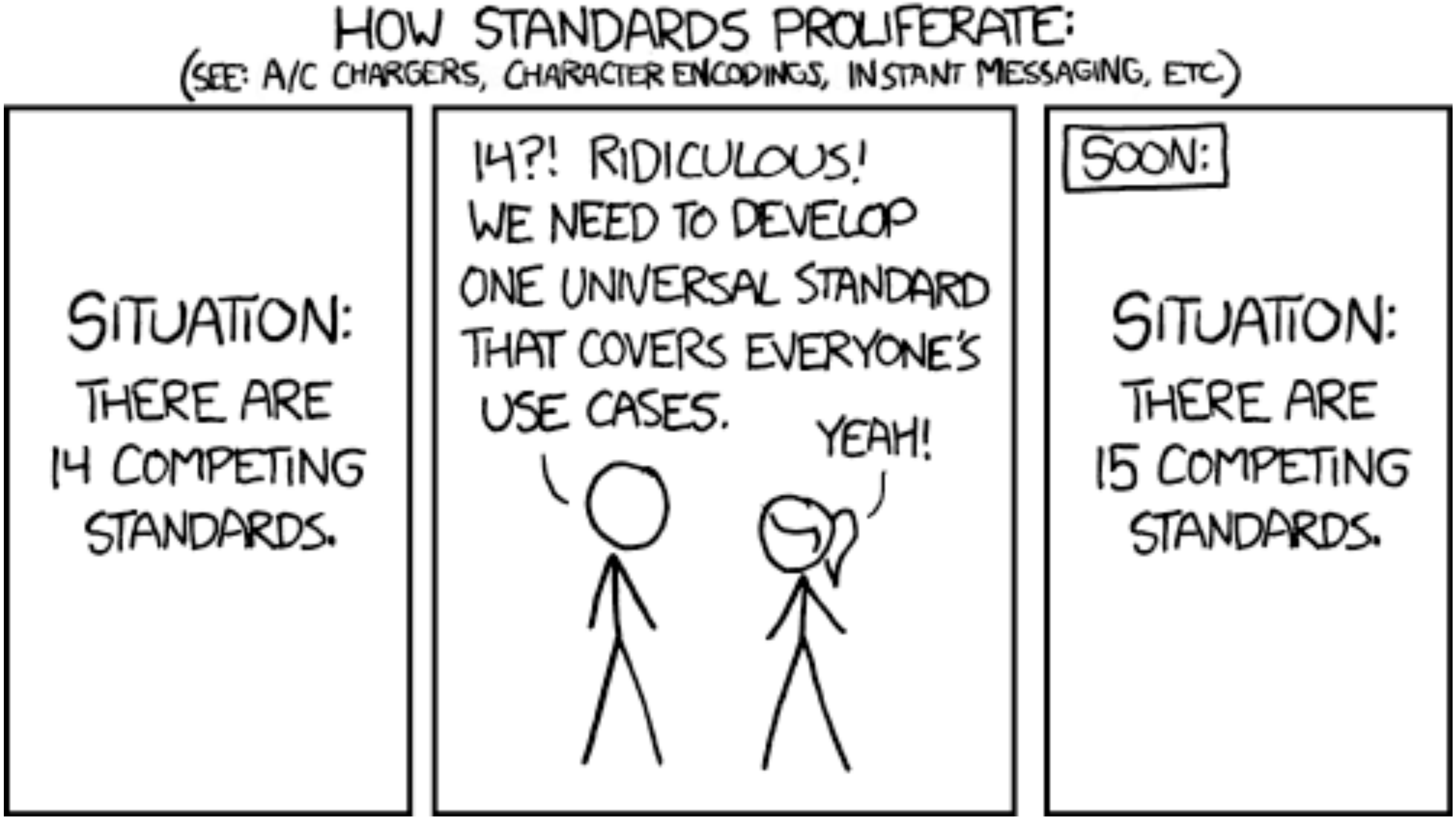
Plan for today

- The Stanford Libraries
  - User Input/Output
  - Collections
  - Graphics
- Memory Management
- **Announcements**
- **Other Languages**

Programming Languages



Programming Languages



https://imgs.xkcd.com/comics/standards.png

C++

```
vector<double> evens;
for(int i = 2; i < 100; i++) {
    if(i % 2 == 0) {
        evens.push_back(i);
    }
}
cout << evens << endl;
```

prints [2, 4, 6, 8, 10, 12, …]

Java

```
ArrayList<Double> evens = new ArrayList<>();
for(int i = 2; i < 100; i++) {
    if(i % 2 == 0) {
        evens.add(i);
    }
}
System.out.println(evens);
```

prints [2, 4, 6, 8, 10, 12, …]

Python

```
evens = []
for i in range(2, 100):
    if i % 2 == 0:
        evens.append(i)
print evens
```

prints [2, 4, 6, 8, 10, 12, …]

Javascript

```
var evens = []
for(var i = 2; i < 100; i++) {
    if(i % 2 == 0) {
        evens.push(i);
    }
}
console.log(evens);
```

prints [2, 4, 6, 8, 10, 12, …]

Recap

- The Stanford Libraries
  - User Input/Output
  - Collections
  - Graphics
- Memory Management
- Announcements
- Other Languages

- **Next Time:** Recapping CS106X