In C++ we don't say "Missing asterisk." We say "error C2664: 'void std::vector<block, std::allocator<_Ty>>::push_back(const block &)': cannot convert argument 1 from 'std::_Vector_iterator<std::_Vector_val<std::_Simple_types<block>>>>' to 'block & &'" and I think that's beautiful.
Logistics

- Mid-quarter survey due tomorrow
  - Link: [https://canvas.uw.edu/courses/1643252/quizzes/1872234](https://canvas.uw.edu/courses/1643252/quizzes/1872234)
- HW2
  - Due Thursday (TONIGHT!) (7/20) @ 11:59pm
  - Late deadline is Sunday @ 11:59 PM with 2 late days
- HW3
  - Releases tomorrow (7/21), is due Thursday 8/3 @ 11:59 PM
- Exercise 8
  - Due Monday (7/24) @ 1 pm
- Quiz 2
  - Releases Monday (7/24) @ 2 pm, is due Wednesday (7/26) @ 11:59 pm
Dynamic Memory
New and Delete Operators

**New:** Allocates the type on the heap, calling its constructor if it is a class type

Syntax:

```c
    type* ptr = new type;
    type* heap_arr = new type[num];
```

**Delete:** Deallocates the type from the heap, calling the destructor if it is a class type. For anything you called `new` on, you should at some point call `delete` to clean it up

Syntax:

```c
    delete ptr;
    delete[] heap_arr;
```
Rule of 3

If you define any of:

1. Destructor
2. Copy Constructor
3. Assignment (operator=)

Then you should normally define all three. Otherwise you may run into strange behavior.

This is especially true if your class needs to allocate something on the heap. (Resources may not get allocated/deallocated correctly)
Design Considerations

- What happens if you don’t define a copy constructor? Or an assignment operator? Or a destructor? Why might this be bad?
  - In C++, if you don’t define any of these, one will be synthesized for you
  - The synthesized copy constructor does a shallow copy of all fields
  - The synthesized assignment operator does a shallow copy of all fields
  - The synthesized destructor calls the default destructors of any fields that have them

- How can you get the default copy constructor/assignment operator/destructor?

  Set their prototypes equal to the keyword “default”: `~SomeClass() = default;`
Exercise 1
Exercise 1

```cpp
class HeapInt{
public:
    HeapInt() { x_ = new int(5); }
private:
    int* x_;}

int main(int argc, char** argv) {
    HeapInt** hpint_ptr = new HeapInt*;
    HeapInt* hpint = new HeapInt();
    *hpint_ptr = hpint;
    delete hpint_ptr;
    return EXIT_SUCCESS;
}
```
Exercise 1: Memory Leaks

class HeapInt{
    public:
        HeapInt() { x_ = new int(5); }
    private:
        int* x_; 
};

int main(int argc, char** argv) {
    HeapInt** hpint_ptr = new HeapInt*;
    HeapInt* hpint = new HeapInt();
    *hpint_ptr = hpint;
    delete hpint_ptr;
    return EXIT_SUCCESS;
}

How can we fix this leak?
 delete hpint;
~HeapInt() { delete x_; }

How many bytes are leaked?
12 (8 for HeapInt object, 4 for the int)
Exercise 2
class HeapArr{
    public:
        HeapArr() { arr_ = new int[5]; }
        ~HeapArr() { delete [] arr_; }
    private:
        int* arr_;}

int main(int argc, char** argv) {
    HeapArr* hparr1 = new HeapArr;
    HeapArr* hparr2 = new HeapArr(*hparr1); // cctor
    delete hparr1;
    delete hparr2;
    return EXIT_SUCCESS;
}
class HeapArr{
public:
    HeapArr() { arr_ = new int[5]; }
    ~HeapArr() { delete [] arr_; }
private:
    int* arr_;}

int main(int argc, char** argv) {
    HeapArr* hparr1 = new HeapArr;
    HeapArr* hparr2 = new HeapArr(*hparr1);
    delete hparr1;
    delete hparr2;
    return EXIT_SUCCESS;  // if!
}
Templates!
Templates

- C++ syntax to generate code that works with *generic types*
- Generates **at compile time** instructions on each way a function is used:
  - e.g., calls to `foo<int>()` and `foo<double>()` generate two implementations
  - e.g., calls to `foo<int>()` and another `foo<int>()` require only one implementation
  - e.g., if `foo` is never used, zero implementations are generated
Template Function

template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}

Results?
add3<int>(3);       // uses add3<int>, returns 6
add3(5.5);
add3<char*>("a str");
add3<string>("a str");
Template Function

template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}

Results?
add3<int>(3);  // uses add3<int>, returns 6
add3(5.5);     // uses add3<double>, returns 8.5
add3<char*>("a str");
add3<string>("a str");
Template Function

template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}

Results?
add3<int>(3); // uses add3<int>, returns 6
add3(5.5); // uses add3<double>, returns 8.5
add3<char*>("a str"); // uses add3<char*>, return ->"tr"
add3<string>("a str");
Template Function

template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}

Results?
add3<int>(3); // uses add3<int>, returns 6
add3(5.5); // uses add3<double>, returns 8.5
add3<char*>("a str"); // uses add3<char*>, return ->"tr"
add3<string>("a str"); // Compiler error! No `+` for string
    // and int
Template Class

- Meant for supporting generic types:

```c
typedef uint64_t HTKey_t;
typedef void* HTValue_t;
typedef struct {
    HTKey_t key;
    HTValue_t value;
} HTKeyValue_t;
```

```cpp
template<typename K, typename V>
struct HTKeyValue {
    K HTKey;
    V* HTValue;
};
```
Exercise 3
Exercise 3

// template type definition
struct Node {

    // two-argument constructor
    // public field value
    // public field next
};

    // destructor cleans up the payload
    ~Node() { delete value; }
Exercise 3 Solution

template <typename T>  // template type definition
struct Node {
  ____________  // two-argument constructor
  ~Node() { delete value; }  // destructor cleans up the payload

  ____________  // public field value
  ____________  // public field next
};
Exercise 3 Solution

template <typename T> // template type definition
struct Node {
    ____________ // two-argument constructor

    ~Node() { delete value; } // destructor cleans up the payload

    T* value // public field value
    Node<T>* next // public field next
};
Exercise 3 Solution

template<typename T> // template type definition
struct Node {
    Node(T* val, Node<T>* node): value(val), next(node) {}
    // two-argument constructor

    ~Node() { delete value; } // destructor cleans up the payload

    T* value // public field value
    Node<T>* next // public field next
};
Containers!
C++ standard lib is built around templates

- **Containers** store data using various underlying data structures
  - The specifics of the data structures define properties and operations for the container
- **Iterators** allow you to traverse container data
  - Iterators form the common interface to containers
  - Different flavors based on underlying data structure
- **Algorithms** perform common, useful operations on containers
  - Use the common interface of iterators, but different algorithms require different ‘complexities’ of iterators
Common C++ STL Containers (and Java equiv)

- **Sequence** containers can be accessed sequentially
  - `vector<Item>` uses a dynamically-sized contiguous array (like `ArrayList`)
  - `list<Item>` uses a doubly-linked list (like `LinkedList`)
- **Associative** containers use search trees and are sorted by keys
  - `set<Key>` only stores keys (like `TreeSet`)
  - `map<Key,Value>` stores key-value pair<>’s (like `TreeMap`)
- **Unordered associative** containers are hashed
  - `unordered_map<Key,Value>` (like `HashMap`)
## Common C++ STL Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>vector</th>
<th>list</th>
<th>set</th>
<th>map</th>
<th>unordered_map</th>
</tr>
</thead>
<tbody>
<tr>
<td>.size()</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>.push_back()</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.pop_back()</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.push_front()</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.pop_front()</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>.operator<a href=""></a></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>.insert()</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>.find()</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
Common STL Containers

Many more containers and methods!

See full documentation here:
http://www.cplusplus.com/reference/stl
Exercise 4
Exercise 4

using namespace std;
vector<string> ChangeWords(const vector<string>& words,
                         map<string,string>& subs) {

}
Exercise 4 Solution

```cpp
using namespace std;

vector<string> ChangeWords(const vector<string>& words, map<string,string>& subs) {

    vector<string> result;
    for (auto& word : words) {
        if (subs.find(word) != subs.end()) {
            result.push_back(subs[word]);
        } else {
            result.push_back(word);
        }
    }

    return result;
}
```