Objects Continued, STL Start

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

Poll Everywhere

pollev.com/tqm

- How are you? Any questions?

Administrivia

- HW00 & HW01
  - Due yesterday night
- Pre semester Survey
  - Anonymous
  - Due Friday the 12th
- HW02 posted after lecture today
  - I *think* it will be less word than HW01
  - Autograder posted later in the day or tomorrow
- Next Check-in posted tomorrow
  - Can request a re-open for HW01 or HW00 in it

Lecture Outline

- **Misc C++**
- Const Objects
- std::vector
- Exceptions
- std::move
- std::optional
- std::list

Aside: auto

- In C++ (and C23 onwards) you can declare a variable with the keyword "auto“ which tells the compiler to automatically deduce the type.

- This only works if there is enough context for the compiler to deduce a type.

```
// Calculate and return a vector
// containing all factors of n
vector<int> Factors(int n);

void foo(void) {
  // Manually identified type
  vector<int> facts1 = Factors(324234);

  // Inferred type
  auto facts2 = Factors(12321);

  // Compiler error here
  auto facts3;
}
```

Compiler knows return value of Factors()

?????????
No information to infer type

Aside: inline member functions

- You can put the definition for member functions in the class/struct declaration.

- This is usually reserved for very simple 1-liner member functions like getters and setters.

- Larger functions *can* be in here but stylistically should not be.

```
class Point {
 public:
  Point(int x, int y);     // constructor
  int GetX() { return m_x; }
  int GetY() { return m_y; }
  double DotProd(Point p);
  void SetLocation(int x, int y);

 private:
  int m_x;  // data member
  int m_y;  // data member
};  // class Point
```

Aside: operator overload

- In C++ we can define how language operators work on different types.
- These can be member or non-member (e.g. “normal”) functions
  - Prefer non-member functions. Only make something a member if it NEEDS to be.

- Usually you only comparison operators are implemented (if any)
  - **`operator<`** **Only defining < is normal. Only one needed for ordering things**
  - **`operator==`** **Also nice to define in some cases**
  - `operator>`
  - `operator<=`
  - `operator >=`
  - `operator !=`
  - **`operator<=>`** **Only came in C++20, acts more like the CompareTo function where it returns -1, 0, or 1. Defining this is sufficient to support all comparisons**

.hpp

```
class Point {
 public:
  bool operator==(const Point& other);

  ...
 };  // class Point
```

.cpp

```
bool Point::operator==(const Point& other) {
  reutrn this->m_x == other.m_x && this->m_y == other.m_y;
}
```

Aside: operator overload

- In C++ we can define how language operators work on different types.
- These can be member or non-member (e.g. “normal”) functions
  - Prefer non-member functions. Only make something a member if it NEEDS to be.

- Other operator overloads exist, but usually only make sense for data structures/iterators or “fundamental” types
  - `operator[]`
  - `operator*`
  - `operator++`
  - `operator+`
  - `operator%`
  - `operator<<`
  - `operator,`

.hpp

```
class Point {
 public:
  int GetX() { return m_x; }
  int GetY() { return m_y; }

 private:
  int m_x;  // data member
  int m_y;  // data member
};  // class Point

int operator*(const Point& lhs, const Point& rhs);
```

.cpp

```
int operator*(const Point& lhs, const Point& rhs) {
  return lhs.GetX() * rhs.GetX() + lhs.GetY() * rhs.GetY();
}
```

and many more

Aside: range for loop

- Syntactic sugar similar to Java's foreach

```
for (declaration : expression) {
  statements
}
```

  - `declaration` defines the loop variable
  - `expression` is an object representing a sequence
    - Strings, and most STL containers work with this

```
string str("hello");
// prints out each character
for (char c : str) {
  cout << c << endl;
}
```

Lecture Outline

- Misc C++
- **Const Objects**
- std::vector
- Exceptions
- std::move
- std::optional
- std::list

Const objects

- Just like with primitive types and structs, we can have a const object
- A const object cannot change its data members*

```
int main() {
  const Point p(3, 2);
}
```

- A const object cannot call a non-const member functions
  - Consider how we previously declared the point object:
  - If we left it as this, then this code would not compile

```
class Point {
 public:
  Point(int x, int y);
  int GetX() { return m_x; }
  int GetY() { return m_y; }
 …
```

```
int main() {
  const Point p(3, 2);
  cout << p.GetX() << endl;
}
```

Const objects

- We need to mark member functions that do not modify any data members as const.

```
class Point {
 public:
  Point(int x, int y);
  int GetX() const { return m_x; }
  int GetY() const { return m_y; }
 …
```

- This tells the compiler that it is ok for const objects to call these member functions.
- Compiler will give error if you try to declare a function const that modifies the data members.
- This code becomes OK now:

```
int main() {
  const Point p(3, 2);
  cout << p.GetX() << endl;
}
```

Const for non-inline functions

- For functions that aren't only in the header: If we wanted to make them const, we need to do it in the cpp and hpp files:

```
class Point {
 public:
  Point(int x, int y);     // constructor
  int GetX() const { return m_x; }
  int GetY() const { return m_y; }
  double DotProd(Point p) const;
  void SetLocation(int x, int y);

 private:
  int m_x;  // data member
  int m_y;  // data member
};  // class Point
```

```
Point::Point(int x, int y) {
  m_x = x;
  m_y = y;
}

double Point::dot_prod(Point p) const {
  double prod = m_x * p.m_x;
  prod += (m_y * p.m_y);
  return prod;
}

void Point::SetLocation(int x, int y) {
  m_x = x;
  m_y = y;
}
```

Lecture Outline

- Misc C++
- Const Objects
- **std::vector**
- Exceptions
- std::move
- std::optional
- std::list

C++ Documentation

- As said, there is a LOT to C++
  - There are a lot of functions, objects, features, etc
  - We will NOT have time to talk about them all

- We highly recommend you make use of a C++ reference
  - cplusplus.com
    - Most find this one easier to read
    - Probably has the information you need
  - cppreference.com
    - Much more detailed (in Travis' opinion)
    - Is in various languages (scroll to the bottom)

STL Containers ☺

- A container is an object that stores (in memory) a collection of other objects (elements)
  - Implemented as class templates, so hugely flexible
  - More info in *C++ Primer* §9.2, 11.2

- Several different classes of container
  - Sequence containers (`vector`, `deque`, `list`, ...)
  - Associative containers (`set`, `map`, `multiset`, `multimap`, `bitset`, ...)
  - Differ in algorithmic cost and supported operations

`vector`

C++ equivalent of ArrayList

- A generic, dynamically resizable array
  - https://cplusplus.com/reference/vector/vector/
  - Elements are store in contiguous memory locations
    - Can index into it like an array
    - Random access is O(1) time
  - Adding/removing from the end is cheap (amortized constant time)
  - Inserting/deleting from the middle or start is expensive (linear time)

- Most common member function: `push_back()`
  - Adds an element to the end of the vector

- **Probably the most important data structure**
  - More on this later

Vector example

```
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char* argv[]) {
  vector<int> vec {6, 5, 4};
  vec.push_back(3);
  vec.push_back(2);
  vec.push_back(1);

  cout << "vec.at(0)" << endl << vec.at(0) << endl;
  cout << "vec.at(1)" << endl << vec.at(1) << endl;

  // iterates through all elements
  for (size_t i = 0U; i < vec.size(); ++i) {
    cout << vec.at(i) << endl;
  }

  return EXIT_SUCCESS;
}
```

Most containers are in a module of the same name

Constructs a vector with three initial elements

Add three integers to the vector

Print all the values in the array

Vector iterator

```
int main(int argc, char* argv[]) {
  vector<int> vec {6, 5, 4};

  vector<int>::iterator it = vec.begin();
  it = vec.insert(it, 3);
  ++it;
  it = vec.insert(it, 1);
  it = vec.end();
  it = vec.insert(it, 2);


  cout << "Iterating:" << endl;
  for (it = vec.begin(); it < vec.end(); ++it) {
     cout << *it << endl;
  }

  return EXIT_SUCCESS;
}
```

Can get an iterator to the beginning of the vector

Insert 3 // {3, 6, 5, 4}

Advances iterator to index 1

Inserts 1 to index 1 {3, 1, 6, 5, 4}

Sets iterator to the end

Same as push_back(2);

Accesses the current element of the iterator

range for loop vector example

- If you need to iterate over every element in a sequence, you should use a range for loop.
  - Why? It is harder to mess it up that way

```
int main(int argc, char* argv[]) {
  vector<int> vec {6, 5, 4};
  vec.push_back(3);
  vec.push_back(2);
  vec.push_back(1);

  // iterates through all elements
  for (int element : vec) {
     cout << element << endl;
  }

  return EXIT_SUCCESS;
}
```

Other vector functions

- pop_back()
  - Removes the last element of the vector
- empty()
  - Returns true if the vector is empty
- clear()
  - Removes all elements currently in the vector
- erase(iterator position)
  - Erases from the element at the specified position
- A bunch more:
  - https://www.cplusplus.com/reference/vector/vector/

Temporal Safety

- A concern in systems programming is that we can sometimes still try to access/use some data after it no longer exists
  - After the data is deallocated from the heap
  - After the data is popped off of the stack
  - The object is destructed
  - Etc.

- An important part of understanding how our basic data structures work, is so that we know how these issues can come up.

Temporal Safety

- What is the issue in this code?

```
#include <iostream>
#include <vector>

using namespace std;


int main(int argc, char** argv) {
  vector<int> v {3, 4, 5};
  int& first = v.front();

  cout << first << endl;

  v.push_back(6);

  cout << v.size() << endl;
  cout << first << endl;
}
```

Lifetimes & Reference Invalidation

- If you read the documentation for many C++ classes, there will be sections on iterator / reference invalidation.

- An example from push_back:
  - If after the operation the new size() is greater than old capacity() a reallocation takes place, in which case all iterators (including the end() iterator) and all references to the elements are invalidated.

- Even if we don’t have to allocate and deallocate things ourselves much in C++, we must still be aware of it.

Ed Discussion

- On Ed there are a bunch of functions from the vector class and their description. Decide which ones ***should*** be const

Ed Discussion

- What does this code print?

```
int main() {
  vector<int> v {3, 5};
  v.push_back(2);

  vector<int> plato = v;
  plato.push_back(16);

  for (int i : v) {
    i *= 2;
  }


  for (int i : v) {
    cout << i;
  }
}
```

Ed Discussion

- How many times is a copy made (for either vec or string)?

```
int main() {
  string name = "ILMC";
  vector<string> subs = make_subs(name);


  for (string sub : subs) {
    cout << sub << endl;
  }
}
```

```
vector<string> make_subs(const string input) {
  vector<string> output{};
  // assume initial capacity is 1
  // capacity is doubled on resize

  size_t i = 0;

  while (i < input.size()) {
    string sub = input.substr(i);
    output.push_back(sub);
    i += 1;
  }

  string& first = output.at(0);
  return output;
}
```

Ed Discussion

- Implement the function rect() which takes in a vector of vector of integers. The function modifies the vector of vectors so that all rows are extended to be the same length (by adding 0’s to the rows).

```
void rect(vector<vector<int>>& m);
```

- For example, the following input

```
vector<vector<int>> m {
  {3, 4, 5},
  {2, 1},
  {},
  {0, 1, 2, 0, 0},
};

rect(m);
```

```
// what it should look
// like after calling rect
vector<vector<int>> m {
  {3, 4, 5, 0, 0},
  {2, 1, 0, 0, 0},
  {0, 0, 0, 0, 0},
  {0, 1, 2, 0, 0},
};
```

Ed Discussion

- Implement the function `intersect()` which takes in two `vector<int>`s and returns a `vector<int>` that has all the integers that can be found in both vectors (no duplicates). The order of the elements in the result do not matter.

```
vector<int> intersect(const vector<int>& v1, const vector<int>& v2) {


}
```

Lecture Outline

- Misc C++
- Const Objects
- std::vector
- **Exceptions**
- std::move
- std::optional
- std::list

Exceptions are things in C++

- I do not like exceptions
  - They are gross
  - Often used incorrectly.
    - Exceptions should truly only be used for “exceptional” behaviour. E.g. behaviour that is truly not intended to happen during run time.
      E.g. `new` will throw an exception if it cannot perform an allocation.
    - Rust does not have exceptions :)))))))))
      If an error happens it calls `panic("error message")` which exits the program
      I like this!
  - Exceptions aren’t always the best for readability:
    - What exception(s) does this function throw?
      Does it even throw an exception?

```
string Queue::remove();
```

    - Need to read the comment usually, easier to mess up :’(
    - Unfortunately constructors don’t “return” anything so exceptions are the norm there.

Exceptions are things in C++

- Exception stuff are in `<exception>` and `<stdexcept>`
- Throw syntax

```
string Queue::remove() {
  if (this->size() <= 0U) {
    throw out_of_range("Error!");
  }
```

- Try/catch syntax

```
string result;
try {
  result = q.remove();
} catch (const exception& err) {
  // handle error
  string res = err.what();
  // above gets an explanation
}
```

- Yucky :face_vomiting:

Exceptions are not free

- Exceptions have a (sometimes ignorable) runtime cost
- Compiler doesn't have to store extra things in your function so that at runtime your program can handle an exception being thrown

- Most code *can* throw an exception tho.
  By far the most common: if your code directly or indirectly calls `new`. `new` will throw an exception when Out Of Memory

- Functions that cannot throw an exception should be declared noexcept

```
int add(int x, int y) noexcept;

int add(int x, int y) noexcept {
  return x + y;
}
```

Lecture Outline

- Misc C++
- Const Objects
- std::vector
- Exceptions
- **std::move**
- std::optional
- std::list

Minimizing Allocations

- Memory allocations require time, sometimes a lot of time to compute.

- If performance is our goal, we should minimize the number of allocations we make. (this usually also means we minimize the number of copies made)

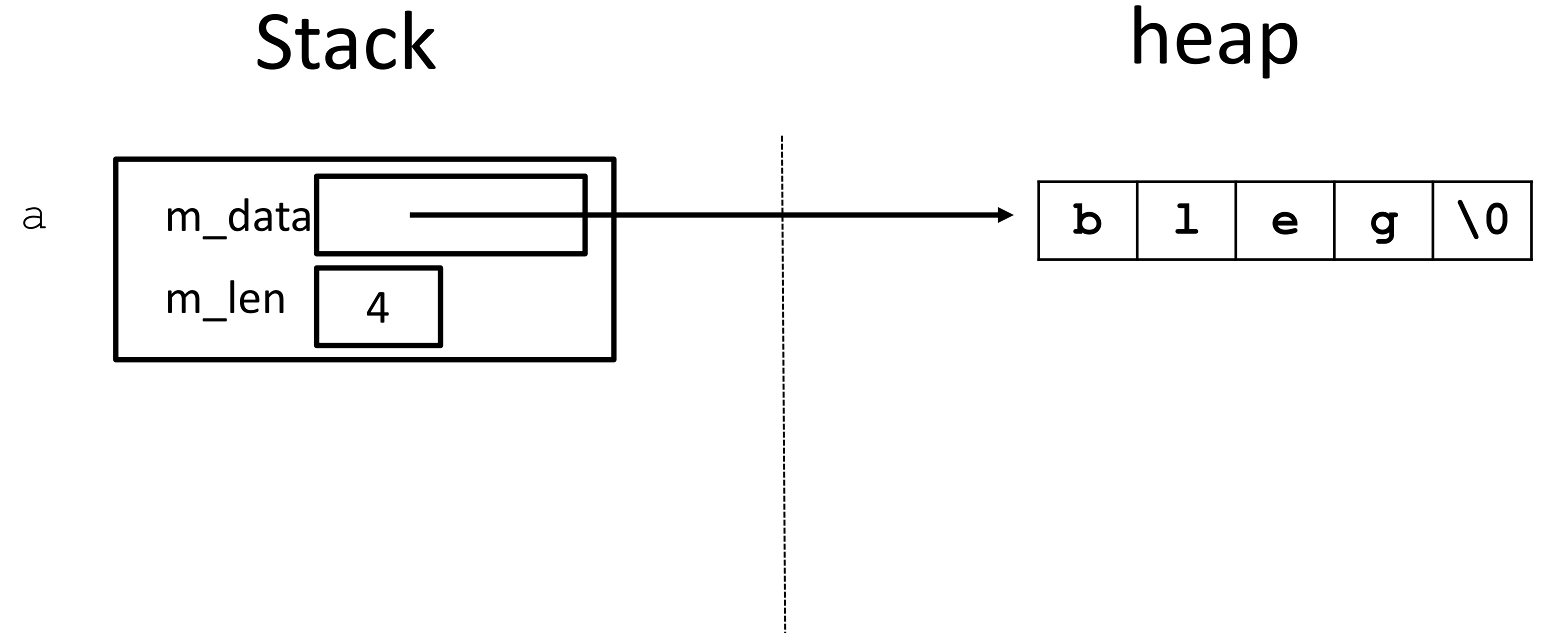
- This can include
  - Making references instead of copies
  - Using functions like `vector::reserve(size_t new capacity)`
    - Java arraylist lets you specify capacity in the constructor.
    - std::string also has a reserve function
  - Using move semantics

Copy Semantics: close up look

- Internally a string manages a heap allocated C string and looks something like:

```
int main(int argc, char **argv) {
  std::string a{"bleg"};


}
```



Copy Semantics: close up look

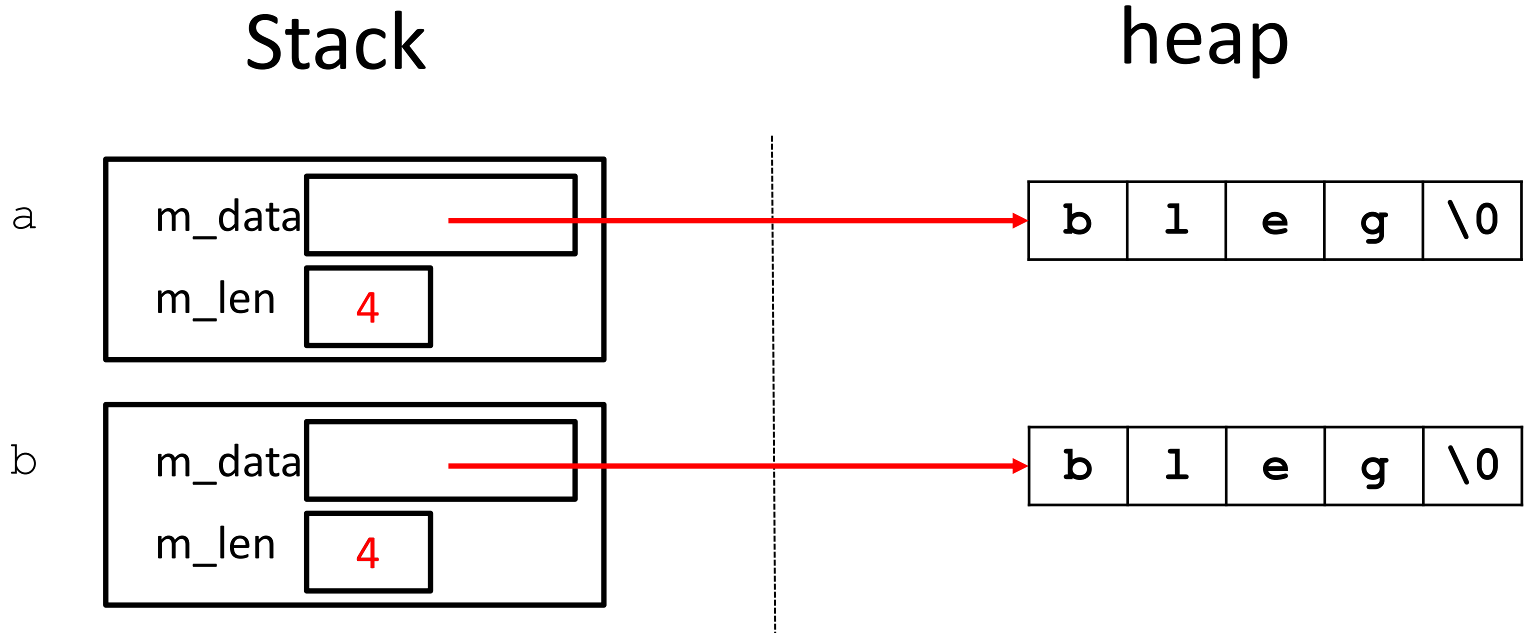
- When we copy construct string **b**

```
int main(int argc, char **argv) {
  std::string a{"bleg"};

  std::string b{a};
}
```

we could get something like:

This is another memory allocation, and we need to copy over the characters of the string



Move Semantics (C++11)

- "Move semantics" move values from one object to another without copying ("stealing")
  - A complex topic that uses things called "*rvalue references*"
    - Mostly beyond the scope of this class

```
int main(int argc, char **argv) {
  std::string a{"bleg"};

  // moves a to b
  std::string b{std::move(a)};
  std::cout << "a: " << a << std::endl;
  std::cout << "b: " << b << std::endl;

  return EXIT_SUCCESS;
}
```

a: ""
b: "bleg"

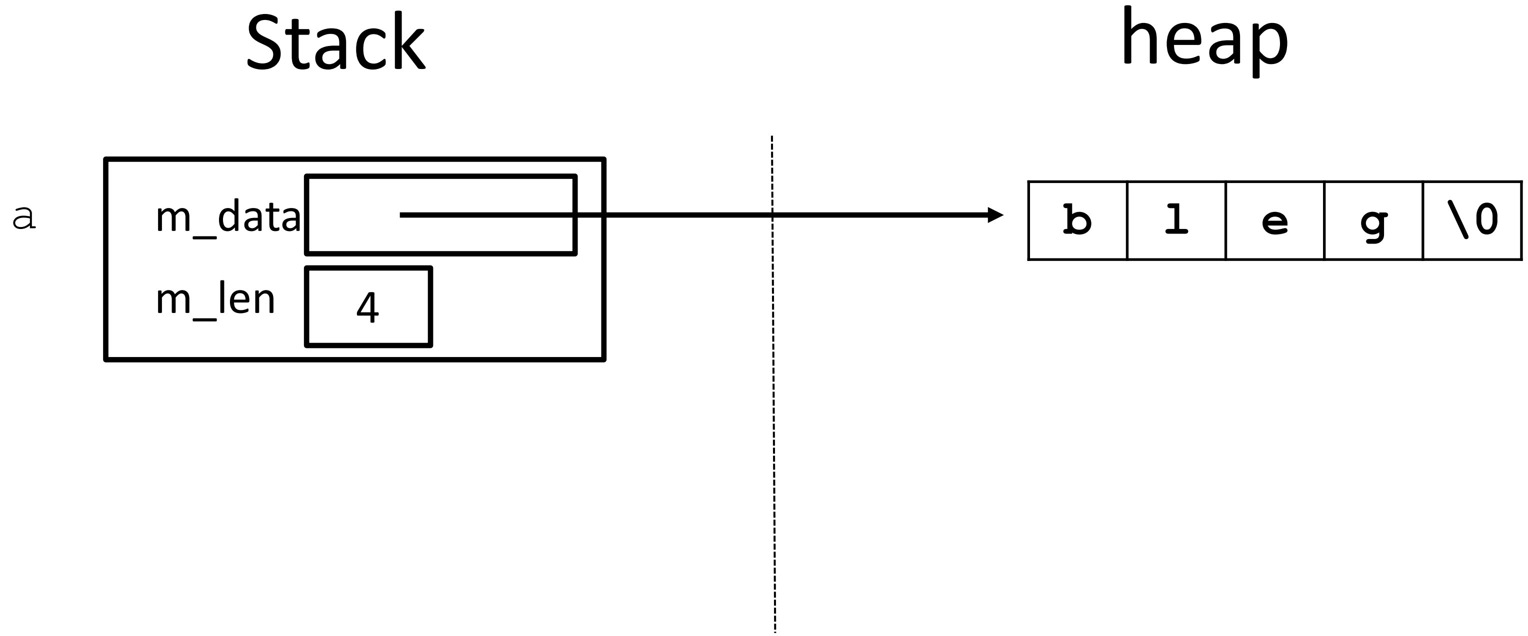
Note: we should NOT access 'a' after we move it. It is undefined to do so, it just so happens it is set to the empty string

Move Semantics: close up look

- Internally a string manages a heap allocated C string and looks something like:

```
int main(int argc, char **argv) {
  std::string a{"bleg"};


}
```



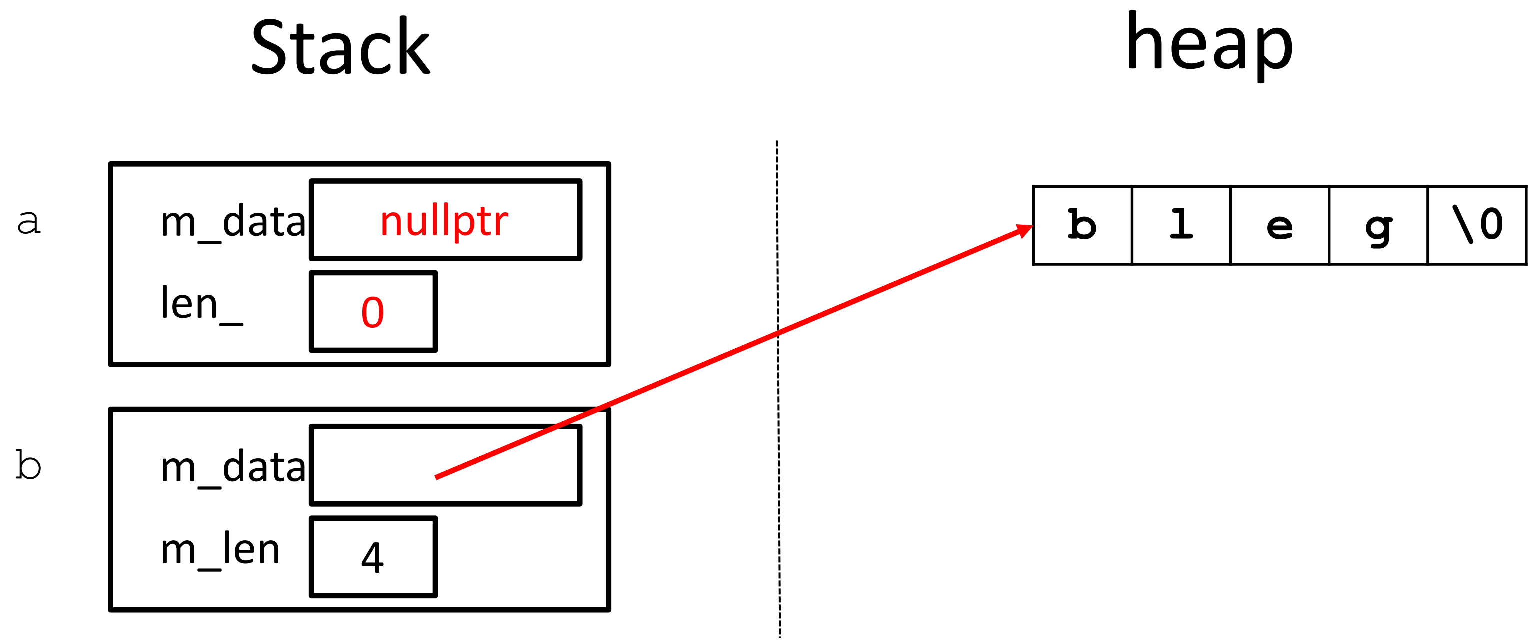
Move Semantics: close up look

- When we use move to construct string **b**

```
int main(int argc, char **argv) {
  std::string a{"bleg"};

  std::string b{std::move(a)};
}
```

we could get something like:



Move Semantics: Use Cases

- Useful for optimizing away temporary copies
- Preferred in cases where copying may be expensive
  - Consider we had a vector of strings… we could transfer ownership of memory to avoid copying the vector and each string inside of it.
- Can be used to help enforce uniqueness

- Rust is a systems programming language that is gaining popularity and by default it will move variables instead of copy them.

- Optional demo: rust_move.rs

Move Semantics: Details

- Implement a "Move Constructor" with something like:

```
Point::Point(Point&& other) {
  // ...
}
```

- Implement a "Move assignment" with something like:

```
Point& Point::operator=(Point&& rhs) {
  // ...
}
```

Move Semantics: Details

- "Move Constructor" example for a fake **`String`** class:

```
String::String(String&& other) {
  this->len_ = other.len_;
  this->ptr_ = other.ptr_;

  other.len_ = 0;
  other.ptr_ = nullptr;
}
```

std::move

- Use `std::move` to indicate that you want to move something and not copy it

```
Point p {3, 2};             // constructor
Point a {p};                // copy constructor

Point b {std::move(p)}; // move constructor
```

noexcept & std::move

- Move assignment and move ctor are extra important to have as noexcept

- Cause some data structures (like std::vector) will only call them if they are no except

- Why?
  - Move constructors and move assignment should never really do anything that warrants throwing an exception. No memory allocation happens in them. Delete doesn't throw an exception

Demo: VerboseInteger.cpp

- What happens when we resize a vector?

- Note what happens when we make move operations noexcept

Lecture Outline

- Misc C++
- Const Objects
- std::vector
- Exceptions
- std::move
- **std::optional**
- std::list

Functions that sometimes fail

- It is pretty common to write functions that sometimes fail. Sometimes they don't return what is expected

- Consider we were building up a Queue data structure that held strings, that could
  - Add elements to the end of a sequence
    - `void add(string data);`
  - Remove elements from the beginning of a sequence
    - `???? remove(????);`
  - How do we design this function to handle the case where there are no strings in the queue (e.g. it errors?)

Previous ways to handle failing functions

- Return an "invalid" value: e.g. if looking for an index, return -1 if it can't be found.
  - What if there is no nice "invalid" state?

```
// what is an invalid string?
string remove();
```

- C-style: return an error code or success/failure.
  Real output returned through output param

```
bool remove(string* output);
```

Aside: Java "Object" variables

- Does this java compile?

```
public static String foo() {
  return null;
}
```

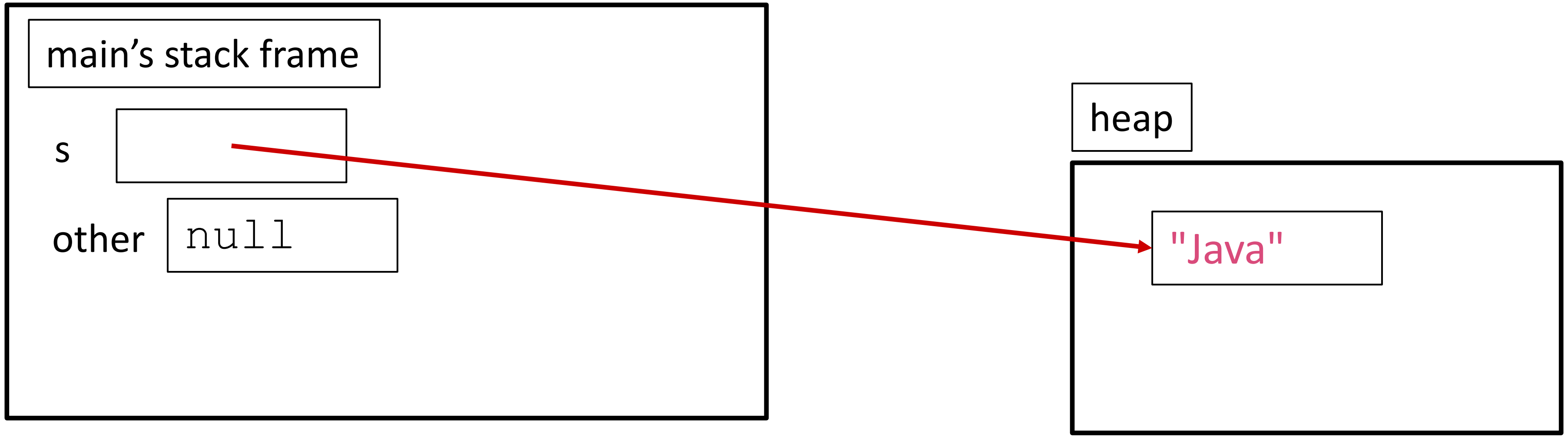
- What about this C++?

```
string foo() {
  return nullptr;
}
```

Aside: Java "Object" variables

- In high level languages (like java), object variables don't actually contain an object, they contain a reference to an object.
  - References in these languages can be null

```
String s = new String("Java");
String other = null;
```

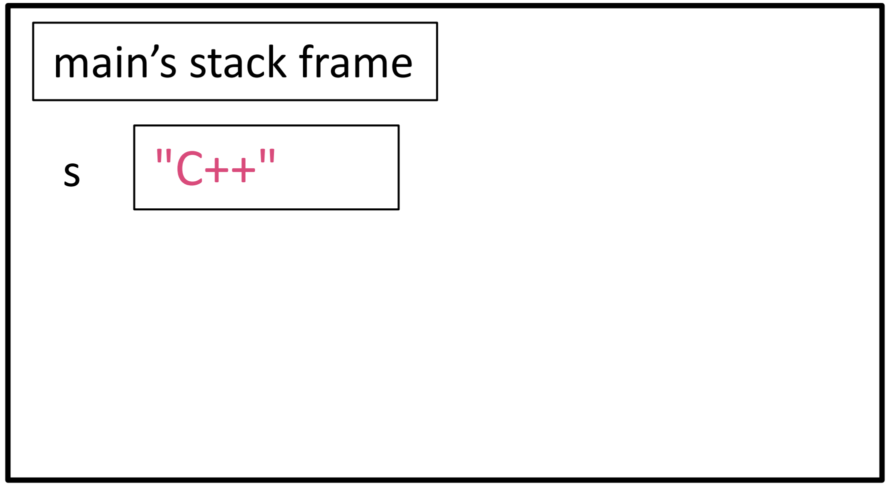


Aside: Java “Object” variables

- In C++, a string variable is itself a string object

```
string s{"C++"};

// does not do what you think it does
string other = nullptr;
```



Previous ways to handle failing functions

- Return a pointer to a heap allocated object, could return **`nullptr`** on error
  - Uses the heap when it is otherwise unnecessary ☹
  - Need to remember to **`delete`** the string

```
string* remove();
```

```
string* remove() {
  if (this->size() <= 0U) {
    return nullptr;
  }
  …
  return new string(…);
```

Previous ways to handle failing functions

- "More Modern" Style: throw an exception in the case of an error
  return the value as normal
  - Exceptions can make your code a bit slower
  - Exception catching not always the easiest to handle
  - Exceptions aren't always the best for readability:
    - What exception(s) does this function throw?

```
string remove();
```

    - Need to read the comment usually, easier to mess up :'(

```
string remove() {
  if (this->size() <= 0U) {
    throw out_of_range("Error!");
  }
```

```
string result;
try {
  result = q.remove();
} catch (exception err) {
  // handle error
}
```

- Unfortunately constructors don't "return" anything so exceptions are the norm there.

std::optional

- **`optional<T>`** is a struct that can either:
  - Have some value T
    (`optional<string> {"Hello!"}`)
  - Have nothing
    (**`nullopt`**)

- **`optional<T>`** effectively extends the type **`T`** to have a "null" or "invalid" state

```
optional<string> foo() {
  if (/* some error */) {
    return nullopt;
  }
  return "It worked!";
}
```

Using an optional

- If we call a function that returns an optional, we need to check to see if it has a value or not

```
optional<string> foo() {
  if (/* some error */) {
    return nullopt;
  }
  return "It worked!";
}

int main() {
  auto opt = foo();
  if (!opt.has_value()) {
    return EXIT_FAILURE;
  }
  string s = opt.value();
}
```

Lecture Outline

- Misc C++
- Const Objects
- std::vector
- Exceptions
- std::move
- std::optional
- **std::list**

STL `list`

- A generic doubly-linked list
  - http://www.cplusplus.com/reference/stl/list/
  - Elements are ***not*** stored in contiguous memory locations
    - Does not support random access (*e.g.* cannot do `list[5]`)
  - Some operations are much more efficient than vectors
    - Constant time insertion, deletion anywhere in list
      - **`push_front()`** and **`pop_front()`** now exist!
      - Can iterate forward or backwards
  - Has a built-in sort member function
  - Doesn't copy! Manipulates list structure instead of element values

Iterate backward: --
Iterate forward: ++

`list` Example

```
#include <list>
#include <algorithm>
#include <string>

using namespace std;

void PrintOut(const string& p) {
  cout << " printout: " << p << endl;
}

int main(int argc, char** argv) {
  list<string> lst;

  lst.push_back("I wanna");
  lst.push_back("f");
  lst.push_back("my computer");
  cout << "sort:" << endl;
  lst.sort();
  cout << "done sort!" << endl;
  for_each(lst.begin(), lst.end(), &PrintOut);
  return 0;
}
```

Use case is similar to vector, but internal implementation is different

Won't copy or move elements, just modifies the next and prev pointers

Ed Discussion

- How many memory allocations occur in each piece of code?
  - Assume vector resizes will double capacity
  - std::list is a linked list in C++

```
int main() {
  vector nums {4, 8};  // size and capacity == 2
  nums.push_back(5);
  nums.push_back(9);
  nums.push_back(5);
  nums.push_back(0);
}
```

```
int main() {
  list nums {4, 8};
  nums.push_back(5);
  nums.push_back(9);
  nums.push_back(5);
  nums.push_back(0);
}
```

Ed Discussion

- Given a linked list object:
  - What does the copy constructor do?
  - What does the move constructor do?

```
struct node {
  node* next;
  string value;
};
```

```
class LinkedList {
 public:
  LinkedList() : m_head(nullptr),
                 m_tail(nullptr),
                 m_len(0) { }

  LinkedList(const LinkedList& other) {
    // TODO: copy constructor
  }

  LinkedList(LinkedList&& other) {
    // TODO: move constructor
  }

 private:
  node* m_head;
  node* m_tail;
  size_t m_len;
};
```

Ed Discussion

- How many copies are made? How many moves are made?
- Assume that there is noexcept move operations implemented for string and for a vector itself

```
int main() {
  string name = "ILMC";
  vector<string> subs = make_subs(name);


  for (const string& sub : subs) {
    cout << sub << endl;
  }
}
```

```
vector<string> make_subs(const string& input) {
  vector<string> output{};
  // assume initial capacity is 1
  ouptut.reserve(input.size());

  size_t i = 0;

  while (i < input.size()) {
    string sub = input.substr(i);
    output.push_back(std::move(sub));
    i += 1;
  }

  string& first = output.at(0);
  return output;
}
```

Ed Discussion

- Given the function strtoi, finish writing a version that uses an optional to indicate success/failure

```
#include <optional>
#include <string>
#include <stdexcept>

using namespace std;

optional<int> StrToInt(const string& to_convert) {
  // Wrap around std::stoi. Read the documentation!
  // you can ignore the base and pos parameters to stoi
  // - https://cplusplus.com/reference/string/stoi/
  // - https://en.cppreference.com/w/cpp/string/basic_string/stoi
}
```

That’s all for now!

- Out or soon to be out:
  - HW02
  - Check-in01
  - Pre-semester Survey

- Hopefully you are doing well ☺