More C++
Computer Systems Programming, Spring 2024

Instructor: Travis McGaha

TAs:

CV Kunjeti
Felix Sun
Heyi Liu
Kevin Bernat
Lang Qin
Sean Chuang
Serena Chen
Yuna Shao

Poll Everywhere

pollev.com/tqm

- What is/are your primary programming language(s)?

Administrivia

- HW0 out later today (or tomorrow)
  - Due a week from tomorrow.
  - Can already setup your docker environment, please do that.
  - Should have everything you need after this lecture.

- Pre-semester survey out today on canvas
  - For credit, but answers are anonymous
  - Due January 31st at 11:59 pm

- Recitation tomorrow will get some practice with this material and have time at the end to help with docker setup.

Lecture Outline

- **C++ Continued**
  - **std::map & std::unordered_map (start)**
  - **std::set & std::unordered_set**
  - std::optional
  - std::variant
  - const
  - Objects (start)

Type Inference (C++11)

- The `auto` keyword can be used to infer types
  - Simplifies your life if, for example, functions return complicated types
  - The expression using `auto` must contain explicit initialization for it to work

```
// 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

auto and Iterators

- Life becomes much simpler!

```
for (vector<Tracer>::iterator it = vec.begin(); it < vec.end(); it++) {
  cout << *it << endl;
}
```

```
for (auto it = vec.begin(); it < vec.end(); it++) {
  cout << *it << endl;
}
```

Look at all this space!!!

Another beautiful feature of C++ ☺

Updated `iterator` Example

```
#include <vector>


using namespace std;

int main(int argc, char** argv) {

  vector<int> vec{};

  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);

  cout << "Iterating:" << endl;
  // "auto" is a C++11 feature not available on older compilers
  for (auto& p : vec) {
    cout << p << endl;
  }
  cout << "Done iterating!" << endl;
  return EXIT_SUCCESS;
}
```

Look at how much more simplified this is!
No begin(), end(), or dereferencing! :O

STL `map`

- One of C++'s *associative* containers: a key/value table, implemented as a search tree
  - http://www.cplusplus.com/reference/map/
  - General form: `map<key_type, value_type> name;`
  - Keys must be *unique*
    - **`multimap`** allows duplicate keys
  - Efficient lookup (O(log n)) and insertion (O(log n))
    - Access `value` via **`operator[]`** (example: **`map_name[key]`**)
      - if key doesn't exist in map, it is added to the map with a "default" value
  - Elements are type `pair<key_type, value_type>` and are stored in *sorted* order (key is field **`first`**, value is field **`second`**)
    - Key type must support less-than operator (<)

Independent types

map Example

#include <map>

map_example.cpp

```
int main(int argc, char** argv) {
  map<int,string> table{};
  map<int,string>::iterator it{};

  table.insert(pair<int,string>(2, "hello"));
  table[4] = "NGNM";
  table[6] = "mutual aid"; // inserts a value
  table[6] = "sleep";  // updates a value
  cout << "table[6]:" << table[6] << endl;

  it = table.find(4);

  if (it != table.end()) {
    cout << "4 exists as a key in the map" << endl;
  }

  cout << "iterating:" << endl;
  for (pair<int, string>& p : table) {
    cout << "[" << p.first << "," << p.second << "]" << endl;
  }
  return 0;
}
```

Map elements

Equivalent behavior

Returns iterator. (end if not found)
can also use map.contains() to see if a key exists

Access the key and value stored in the pair

STL `set`

- One of C++'s *associative* containers: a container of unique values, implemented as a search tree
  - http://www.cplusplus.com/reference/set/
  - General form: `set<element_type> name;`
  - elements must be *unique*
    - **`multiset`** allows duplicate elements
  - Efficient lookup (O(log n)) and insertion (O(log n))
  - Inserting an element that already exists does nothing
  - Can use `count(element)` to see if the element exists
  - Elements are stored in *sorted* order
    - element type must support less-than operator (<)

set Example

set_example.cpp

```
#include <set>

int main(int argc, char** argv) {
  set<string> names {};

  names.insert("bjarne");        ← Doesn't insert duplicate elements
  names.insert("ken");
  names.insert("dennis");
  names.insert("travis");
  names.insert("bjarne");
                                                    prints "true"
  bool exists = names.contains("bjarne");
  cout << "Is bjarne in the set?: " << exists << endl;

  numbers.erase("travis");       ← Removes the element "travis"

  for (string& name : names) {
    cout << name << endl;        ← Prints every name in the set
  }
  return EXIT_SUCCESS;
}
```

Unordered Containers (C++11)

- `unordered_map`, `unordered_set`
  - And related classes `unordered_multimap`, `unordered_multiset`
  - Average case for key access is O(1), so generally preferred
    - But range iterators can be less efficient than ordered `map`/`set`
  - See *C++ Primer*, online references for details

Unordered vs Ordered Containers

- The comparison between `unordered_map` vs `map` is similar to how HashMap vs TreeMap are related in java.
  - Both use the same interface
  - Have different implementations
  - If you want things to be in sorted order, use `map` (TreeMap)
  - In almost all other cases, use `unordered_map` (HashMap)

unordered_map Example

#include <unordered_map>

```
int main(int argc, char** argv) {
  unordered_map<int,string> table{};
  unordered_map<int,string>::iterator it{};

  table.insert(pair<int,string>(2, "hello"));
  table[4] = "NGNM";
  table[6] = "mutual aid"; // inserts a value
  table[6] = "sleep";  // updates a value
  cout << "table[6]:" << table[6] << endl;

  if (table.contains(4)) {
    cout << "4 exists as a key in the map" << endl;
  }

  cout << "iterating:" << endl;
  for (auto& p : table) {
    cout << "[" << p.first << "," << p.second << "]" << endl;
  }
  return 0;
}
```

Lecture Outline

- **C++ Continued**
  - std::map & std::unordered_map (start)
  - std::set & std::unordered_set
  - **std::optional**
  - **std::variant**
  - const
  - Objects (start)

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;
}
```

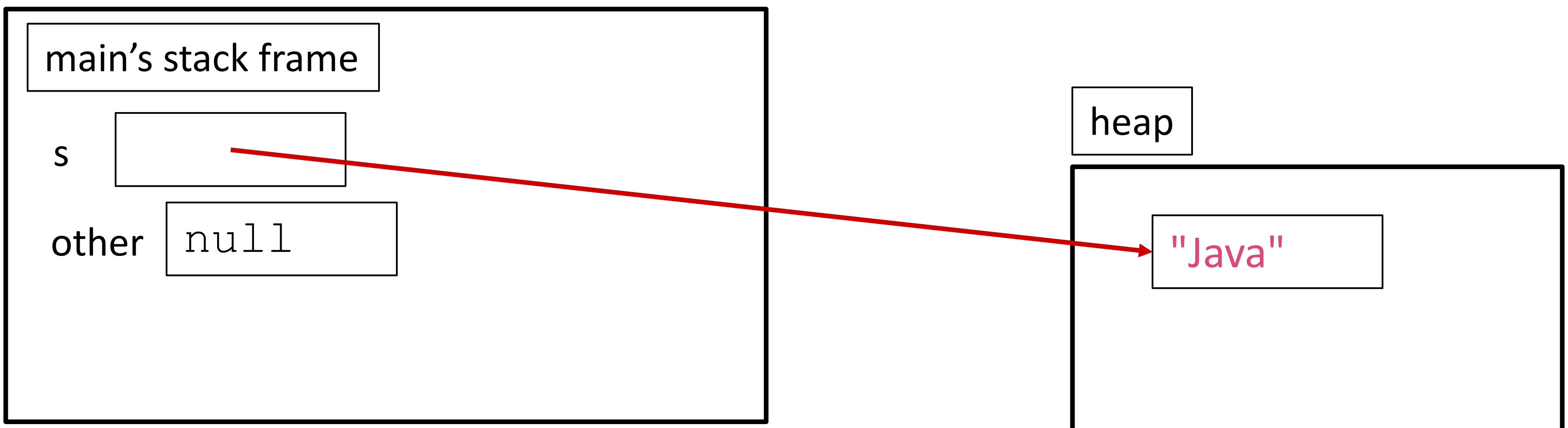
- What about this C++?

```
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;
```

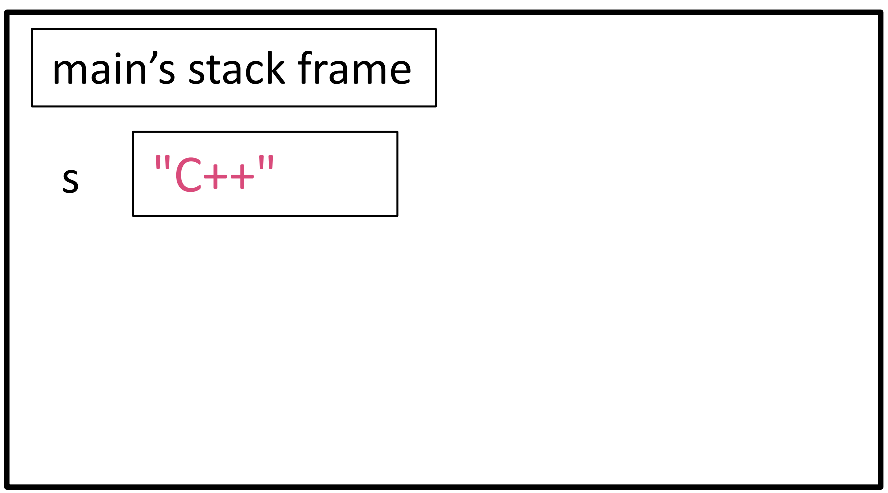


Aside: Java "Object" variables

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

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

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



More on this idea when I talk about pointers later

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();
```

- Java style: throw an exception in the case of an error
  return the value as normal
  - Exceptions not best for performance
  - Exception catching not always the easiest to handle

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

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();
}
```

std::variant

- Similar to how std::optional can store 1 type or nothing, std::variant can store **one of** two or more different values

```
int main() {
  variant<int, string> var {3};

  cout << holds_alternative<int>(var) << endl;
  cout << get<int>(var) << endl

  cout << holds_alternative<string>(var) << endl;
  cout << get<string>(var) << endl;
}
```

Lecture Outline

- **C++ Continued**
  - std::map & std::unordered_map (start)
  - std::set & std::unordered_set
  - std::optional
  - std::variant
  - **const**
  - Objects (start)

const

- `const`: this cannot be changed/mutated
  - Used *much* more in C++ than in C
  - Signal of intent to compiler; meaningless at hardware level
    - Results in compile-time errors

```
void BrokenPrintSquare(const int& i) {
  i = i * i;  // compiler error here!
  cout << i << endl;
}

int main(int argc, char** argv) {
  int j {2};
  BrokenPrintSquare(i);
  return EXIT_SUCCESS;
}
```

`const`

- `const`: this cannot be changed/mutated
  - Used *much* more in C++ than in C
  - Signal of intent to compiler; meaningless at hardware level
    - Results in compile-time errors

```
void FixedPrintSquare(const int& i) {
  int x {i * i};  // ok now!
  cout << x << endl;
}

int main(int argc, char** argv) {
  int j {2};
  BrokenPrintSquare(i);
  return EXIT_SUCCESS;
}
```

const variables

- Variables can be declared const on their own

```
int main(int argc, char** argv) {
  const int j {2};
  int x {3};
  j++;  // cannot modify a const variable!
  // Ok to not modify a non-const variable
  return EXIT_SUCCESS;
}
```

- Making a variable const means you do not want to modify it and the compiler should not allow other things to modify

- **Variables that are not const does not have to be modified**

`const` and references

yes

no

- Lets go over different cases:

```
int main(int argc, char** argv) {
  int x {5};                  // int
  const int y {6};            // (const int)
  y++;

  const int& z {y};           // const int reference
  z += 1;
  y++;

  const int& w {x};          // const int reference
  w += 1;
  x++;

  int& t {y};                 // int reference
  t += 1;
  y++;

  return EXIT_SUCCESS;
}
```

constmadness.cc

`const` Parameters

Make parameters const when you can

- A `const` parameter *cannot* be mutated inside the function
  - Therefore it does not matter if the argument can be mutated or not

- A non-`const` parameter *may* be mutated inside the function
  - Compiler won't let you pass in const parameters

```
void foo(const int& y) {
  std::cout << y << std::endl;
}

void bar(int& y) {
  std::cout << y << std::endl;
}

int main(int argc, char** argv) {
  const int a {10};
  int b {20};

  foo(a);    // OK
  foo(b);    // OK
  bar(a);    // not OK - error
  bar(b);    // OK

  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- What will happen when we try to compile and run?

A. Output "(2, 4, 2)"
B. Output "(2, 2, 2)"
C. Output "(2, 4, 4)"
D. Compiler error about arguments to foo (in main)
E. Compiler error about body of foo
F. We're lost…

const_ref_poll.cpp

```
void foo(int& x, int& y, int z) {
  x += 1;
  y *= 2;
  z -= 2;
}

int main(int argc, char** argv) {
  const int a {1};
  int b {2};
  const int& c {b};

  foo(a, b, c);
  cout << "(" << a << ", " << b
   << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```

Poll Everywhere

pollev.com/tqm

- What will happen when we try to compile and run?

A. Output "(2, 4, 2)"
B. Output "(2, 2, 2)"
C. Output "(2, 4, 4)"
D. Compiler error about arguments to foo (in main)
E. Compiler error about body of foo
F. We're lost…

const_ref_poll.cpp

```
void foo(int& x, int& y, int z) {
  x += 1;
  y *= 2;
  z -= 2;
}

int main(int argc, char** argv) {
  const int a {1};
  int b {2};
  const int& c {b};

  foo(a, b, c);
  cout << "(" << a << ", " << b
   << ", " << c << ")" << endl;

  return EXIT_SUCCESS;
}
```

Handwritten annotations: "Allowed" (x += 1, y *= 2, z -= 2); "Allowed" (const int& c {b}); "A violates const" (a in foo(a, b, c)); "C is passed by copy, so allowed" (foo(a, b, c)). Answer D underlined.

Lecture Outline

- **C++ Continued**
  - std::map & std::unordered_map (start)
  - std::set & std::unordered_set
  - std::optional
  - std::variant
  - const
  - **Objects (start)**

Structs in C

- In C, we only had **`struct`**s, which could only bundle together data fields

- Struct example definition:

```
struct Point {  // Declare struct, usually used typedef
  // Declare fields & types here
  int x;
  int y;
};
```

- What is missing from this compared to objects/classes in languages other languages?
  - Methods
  - Access modifiers (public vs private)
  - Inheritance

Classes in C++

- In C++, we have classes.
  - Think of these as C structs, but with methods, access modifiers, and inheritance.

- Class example definition:

Similar syntax for declaration

```
class Point {  // Declare class, typedef usually not used
 public:
  Point(int x, int y);   // constructor
  int get_x();           // getter
  int get_y();           // getter
 private:
  int x_;                // fields
  int y_;
};
```

Access modifiers

methods

Fields

- In C++, we call fields and methods "members"

Classes Syntax

- Class definition syntax (in a `.hpp` file):

```
class Name {
 public:
  // public member definitions & declarations go here

 private:
  // private member definitions & declarations go here
};  // class Name
```

don't forget!

  - Members can be functions (methods) or data (variables)

- Class member function definition syntax (in a `.cpp` file):

```
retType Name::MethodName(type1 param1, …, typeN paramN) {
  // body statements
}
```

  - (1) *define* within the class definition or (2) *declare* within the class definition and then *define* elsewhere

Class Definition (.hpp file)

Point.hpp

```
#ifndef POINT_HPP_
#define POINT_HPP_

class Point {
 public:
  Point(int x, int y);      // constructor
  int get_x() { return x_; }      // inline member function
  int get_y() { return y_; }      // inline member function
  double distance(Point p);       // member function
  void set_location(int x, int y); // member function

 private:
  int x_;  // data member
  int y_;  // data member
};  // class Point

#endif  // POINT_HPP_
```

Declarations

Inline definition ok for simple getters/setters

C++ naming conventions for data members

Class Member Definitions (.cpp file)

Point.cpp

This code uses bad style for demonstration purposes

```
#include <cmath>
#include "Point.hpp"

Point::Point(int x, int y) {
  x_ = x;
  this->y_ = y;  // "this->" is optional unless name conflicts
}

double Point::distance(Point p) {
  // We can access p's x_ and y_ variables either through the
  // get_x(), get_y() accessor functions or the x_, y_ private
  // member variables directly, since we're in a member
  // function of the same class.
  double distance = (x_ - p.get_x()) * (x_ - p.get_x());
  distance += (y_ - p.y_) * (y_ - p.y_);
  return sqrt(distance);
}

void Point::set_location(int x, int y) {
  x_ = x;
  y_ = y;
}
```

Equivalent to y_=y;

"this" is a Point* const

We have access to x_, could have used x_ instead.

Class Usage (.cpp file)

usepoint.cpp

```
#include <iostream>
#include "Point.h"

using namespace std;

int main(int argc, char** argv) {
  Point p1{1, 2};  // construct a new Point on the Stack
  Point p2{4, 6};  // construct a new Point on the Stack

  cout << "p1 is: (" << p1.get_x() << ", ";
  cout << p1.get_y() << ")" << endl;

  cout << "p2 is: (" << p2.get_x() << ", ";
  cout << p2.get_y() << ")" << endl;

  cout << "dist : " << p1.distance(p2) << endl;
  return 0;
}
```

Calls constructor to define an object on the stack. (no “new” keyword)

Dot notation to call function (like java)

Constructors

- A constructor (ctor) initializes a newly-instantiated object
  - A class can have multiple constructors that differ in parameters
    - Which one is invoked depends on *how* the object is instantiated
  - A constructor is always invoked when creating a new instance of an object.

- Written with the class name as the method name:

```
Point(const int x, const int y);
```

  - C++ will automatically create a synthesized default constructor if you have ***no*** user-defined constructors
    - Takes no arguments and calls the default ctor on all non-"plain old data" (non-POD) member variables
    - Synthesized default ctor will fail if you have non-initialized const or reference data members

Zero arg

Created for you

Synthesized Default Constructor Example

SimplePoint.h

```
class SimplePoint {
 public:
  // no constructors declared!
  int get_x() { return x_; }      // inline member function
  int get_y() { return y_; }      // inline member function
  double Distance(SimplePoint p);
  void SetLocation(int x, int y);

 private:
  int x_;  // data member
  int y_;  // data member
};  // class SimplePoint
```

Default initializes fields:
- If primitive, garbage values (like normal vars)
- If object, run default (zero arg) ctor

SimplePoint.cc

```
#include "SimplePoint.hpp"

... // definitions for Distance() and SetLocation()

int main(int argc, char** argv) {
  SimplePoint x{};  // invokes synthesized default constructor
  return EXIT_SUCCESS;
}
```

Synthesized Default Constructor

- If you define *any* constructors, C++ assumes you have defined all the ones you intend to be available and will *not* add any others

```
#include "SimplePoint.hpp"

// defining a constructor with two arguments
SimplePoint::SimplePoint(int x, int y) {
  x_ = x;
  y_ = y;
}

void foo() {
  SimplePoint x{};          // compiler error:  if you define any
                          // ctors, C++ will NOT synthesize a
                          // default constructor for you.

  SimplePoint y{1, 2};    // works:  invokes the 2-int-arguments
                          // constructor
}
```

Because we defined a ctor already

Multiple Constructors (overloading)

```
#include "SimplePoint.hpp"

// default constructor
SimplePoint::SimplePoint() {
  x_ = 0;
  y_ = 0;
}

// constructor with two arguments
SimplePoint::SimplePoint(int x, int y) {
  x_ = x;
  y_ = y;
}

void foo() {
  SimplePoint x{};        // invokes the default constructor
  SimplePoint y{1, 2};  // invokes the 2-int-arguments ctor

}
```

Initialization Lists

- C++ lets you *optionally* declare an initialization list as part of a constructor definition
  - Initializes fields according to parameters in the list
  - The following two are (nearly) identical:

```
Point::Point(int x, int y) {
  x_ = x;
  y_ = y;
}
```

data member name

Expression

Body can be empty

```
// constructor with an initialization list
Point::Point(int x, int y) : x_(x), y_(y) {


}
```

Initialization vs. Construction

```
class Point3D {
 public:
  // constructor with 3 int arguments
  Point3D(int x, int y, int z) : y_(y), x_(x) {
    z_ = z;
  }

 private:
  int x_, y_, z_;   // data members
};  // class Point3D
```

First, initialization list is applied.

2) set y_

1) set x_

3) set z_ (garbage)

4) set z_

Next, constructor body is executed.

- Data members in initializer list are initialized in the order they are defined in the class, not by the initialization list ordering (!)
  - Data members that don't appear in the initialization list are *default initialized/constructed* before body is executed
- Initialization preferred to assignment to avoid extra steps
  - Real code should never mix the two styles