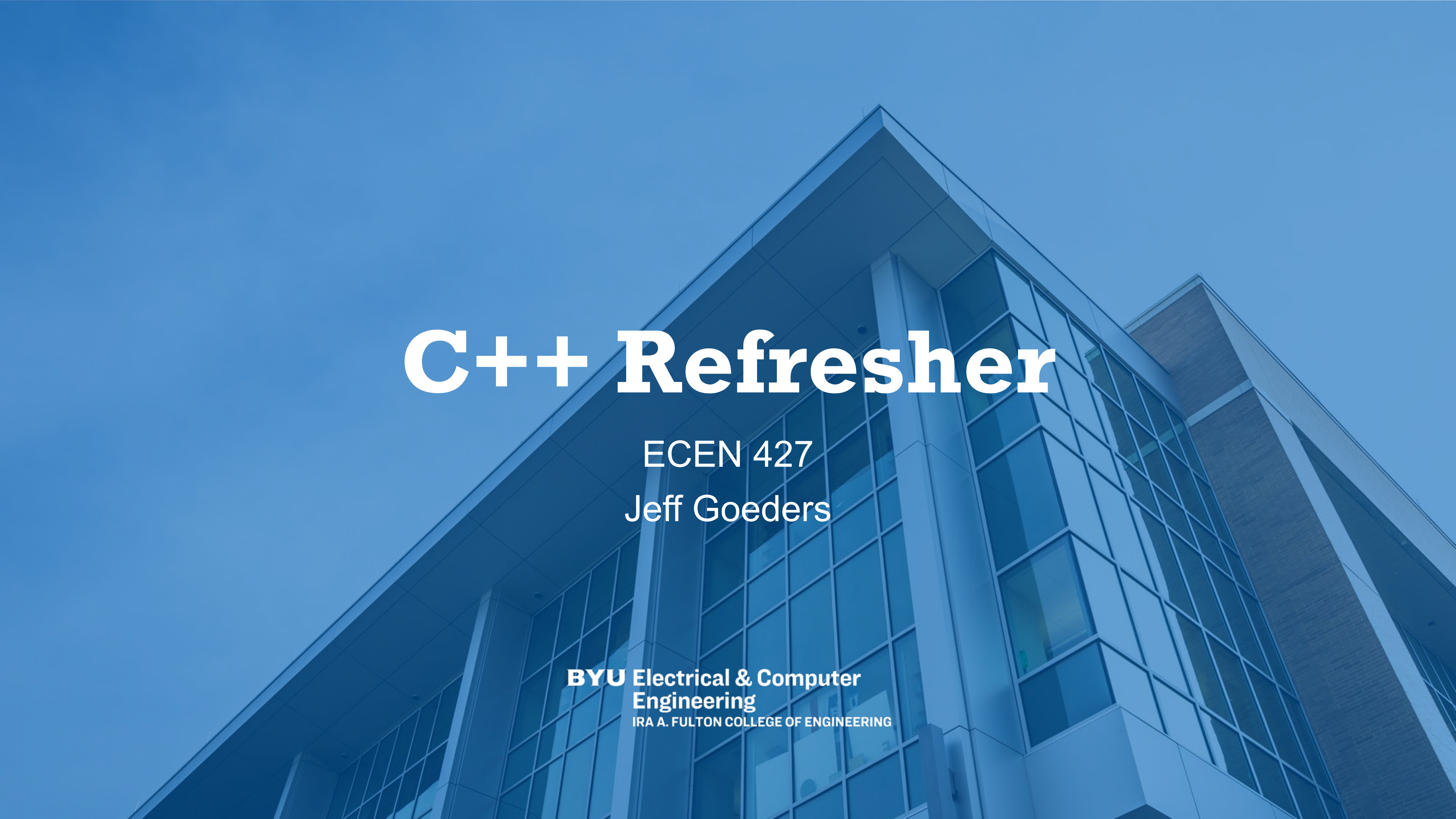

C++ Refresher

ECEN 427

Jeff Goeders

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

References

- References are aliases; pointers store addresses
  - References must be initialized and cannot be reseated
  - Pointers can be null and reassigned
  - References are safer and clearer for parameters
  - Pointers required for dynamic allocation and ownership semantics

```
int x = 10;

int& r = x;      // reference
int* p = &x;     // pointer

r = 20;          // modifies x
*p = 30;         // modifies x


printf(“x: %d”, r);  // print using ref
printf(“x: %d”, *p); // print using pointer
```

Reference Members in Classes

- Reference members must be initialized
  - Initialization must occur in constructor initializer list
  - References cannot be reseated

```
class Foo {
    int& ref;

public:
    Foo(int& x) : ref(x) {}
};

int main() {
    int a = 10;
    Foo f(a);   // ref aliases a
}
```

STL

- Standard Template Library
  - Provides various data structures
  - *Template:* They provide the template, you provide the type (T)
    - list of integers
    - map of int -> string
- std::vector<T>: This is a contiguous array
  - Fast random access, fast iteration, fast insertion at the end
  - Slow insertion/deletion in the middle
- std::list<T>: Linked list
  - No random access
  - Fast insertion/deletion anywhere
- std::map<T1,T2>: Key-Value mapping, like a dictionary
  - Fast to add and delete entries, and to lookup a value by a key

Range-Based Loop Example

New Method

```
std::vector<int> v = {1, 2, 3};

for (int x : v) {
    std::cout << x << std::endl;
}
```

OLD Method

```
for (std::vector<int>::iterator it = v.begin();
     it != v.end();
     ++it)
{
    std::cout << *it << std::endl;
}
```

You can also use auto…

```
for (auto x : v) {
    std::cout << x << std::endl;
}
```

Iterating Through map (Example)

```
std::map<std::string, int> m;

for (const auto& [k, v] : m) {
    std::cout << k << ": " << v << std::endl;
}
```

Erasing From a List

How to look through this vector/list and erase items?

```
std::vector<Foo*> v;
```

Can’t do this:

```
for (auto p : v) {
    if (p->isDead()) {
        delete p;
        v.erase(...);   // iterator invalidation → undefined behavior
    }
}
```

You’ll need to use iterators, and do so carefully

```
for (auto it = v.begin(); it != v.end(); ) {
    if ((*it)->isDead()) {
        delete *it;                  // if vector owns the objects
        it = v.erase(it);            // shifts elements, returns next iterator
    } else {
        ++it;
    }
}
```

Base Constructor and Function Call

```
class Base {
public:
    Base(int x) {}
    void f() {}
};

class Derived : public Base {
public:
    Derived() : Base(42) {}
    void g() { Base::f(); }
};
```

- The provided headers use inheritance
- GameObject class
  - Move, erase, draw, kill
  - Alien, Bullet, Bunker, BunkerBlock, Tank, UFO
- Look at how the constructor initialized the base object in the initialization list
- Look at how code in a subclass calls the base class method

How should we track global stuff?

- Games have lots of global state

- One approach:
  - Organize global data into classes
  - Enforce only one instance of the class

- There are many different approaches for managing global data. This is just one

Globals.h

Meyer’s Singleton

globals.h

```
class Globals {
public:
    static Graphics& getGraphics() {
        static Graphics g;
        return g;
    }

}
```

Function-local static

- Initialized on first use
- Thread-safe since C++11
- Avoids static initialization order issues

Static class member

globals.h

```
class Globals {
public:
    static Graphics& getGraphics() {
        return graphics;
    }

private:
    static Graphics graphics;
};
```

globals.cpp

```
Graphics Globals::graphics;
```