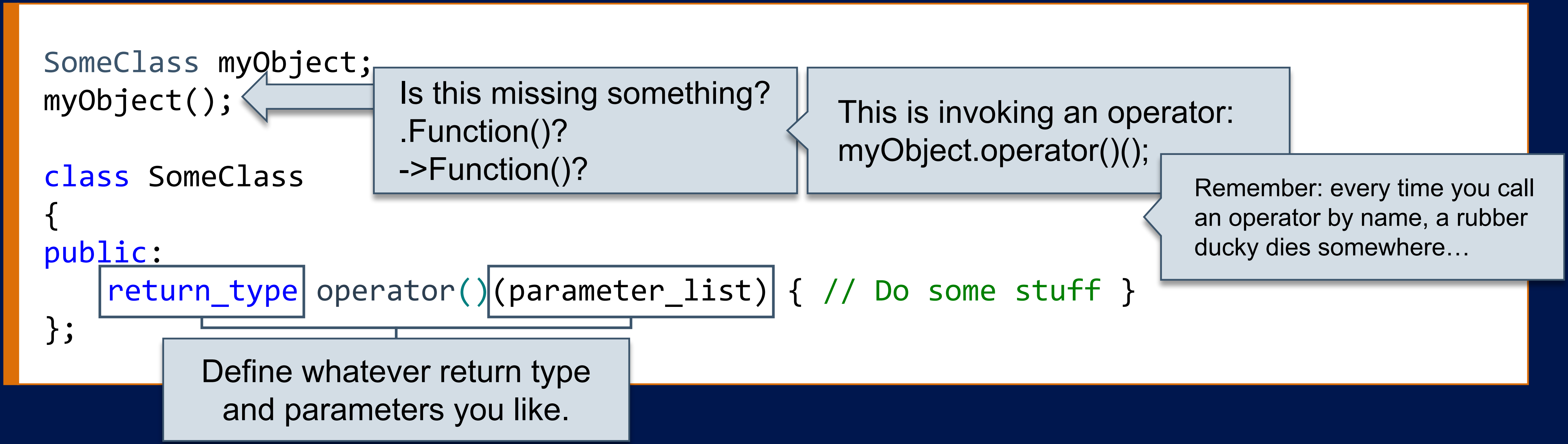
COP3503

Functors

AKA Function Objects

What Is a Functor (or Function Object)?

- A class that can act like a function
- It can do this by overloading **`operator()`** – the **call operator**.
  - This lets you invoke, or call, the object as you would a function.



```cpp
SomeClass myObject;
myObject();

class SomeClass
{
public:
    return_type operator()(parameter_list) { // Do some stuff }
};
```

Is this missing something?
.Function()?
->Function()?

This is invoking an operator:
myObject.operator()();

Remember: every time you call an operator by name, a rubber ducky dies somewhere…

Define whatever return type and parameters you like.

Functor Example

```
// Creating a random number generator
std::mt19937 mt(time(0));
std::uniform_int_distribution<int> dist(0, 100);

// Generate a random number within the range
int random = dist(mt); // Invoking dist.operator()(mt)

// dist() is more concise than:
dist.generate();
dist.Execute();
dist.MakeItSo();
dist.gen();
dist.value(); // etc…
```

Functor Example

```
class DieRoller
{
    map<int, int> _results;
    int _max;
public:
    DiceRoller(int sides) { _max = sides; }

    // Primary function doing most of the work
    int operator()()
    {
        // Roll a number, store the result
        int roll = Random::Int(1, _max);
        _results[roll]++;
        return roll;
    }
    void Results()
    {
        // Print results for all rolls
        // Number of rolls, distribution, etc
    }
};
```

It can do more by keeping track of some data.

```
// Generate random numbers from 1-10
DieRoller tenSided(10);

int value1 = tenSided();
int value2 = tenSided();
int value3 = tenSided();

vector<int> numbers;
for (int i = 0; i < 5; i++)
    numbers.push_back(tenSided());

tenSided.Results();
```

Primary usage:
Act like a function

It’s a class; it can have other functions as well.

An Alternative to Function Pointers

```
// Custom comparison function
bool Ascending(int a, int b)
{
    return a > b;
}
```

```
// Sort with a custom comparison function (passed as a pointer)
void SortStuff(vector<int>& numbers, bool(*compare)(int, int))
{
    if (compare(numbers[i], numbers[i + 1])) { // swap }
}

// Pass a pointer to the function
SortStuff(someVector, Ascending);
```

Same usage in the function

Different data types

```
// Functor
struct Ascending
{
    bool operator()(int a, int b)
    {
        return a > b;
    }
};
```

```
// Sort with a custom comparison, passed as a functor
void SortStuff(vector<int>& numbers, Ascending compare)
{
    if (compare(numbers[i], numbers[i + 1])) { // swap }
}

// Pass a temporary instance of the class to SortStuff()
SortStuff(someVector, Ascending());
```

If we get the same results:
Why bother using functors?

Remember: Functors
can store data too!

STL, Iterators, and Custom Functions

- A lot of STL functionality uses iterators over some range of elements.
- The functions require an operation to perform on those elements.

Formally, the operation is called a **predicate**.

Short version: A predicate is a function (or functor) that returns a Boolean.

```
// Count all elements in a range meeting some condition
int count = std::count_if(iterator_first, iterator_last, condition_predicate);
```

Iterate over all elements within this range.

For each element in the range, call this function and pass the element to it.

Using Predicates With count_if

```
// Make a list of random numbers from 1-40
vector<int> numbers;
for (int i = 0; i < 10; i++)
    numbers.push_back(Random::Int(1, 40));
```

- **Goal**: Count how many numbers in the list are greater than 20.
- Create either a function or a functor, to pass to std::count_if

```
bool GreaterThan20(int number)
{
    return number > 20;
}
```

```
struct GreaterThan20
{
    bool operator()(int number)
    { return number > 20; }
};
```

```
// Use count_if, and pass it either the function or functor (both will work)
int count1 = std::count_if(numbers.begin(), numbers.end(), GreaterThan20);
int count2 = std::count_if(numbers.begin(), numbers.end(), GreaterThan20());
```

That doesn't answer: why use functors over function pointers?

What if you wanted to use a number other than 20?

You can't customize the function pointer, but you **can** customize the functor!

Customizing a Functor

```
struct GreaterThan20
{
     bool operator()(int number)
     { return number > 20; }
};
```

- Don’t hard-code a 20 in this class.

```
struct GreaterThan_X
{
     // Constructor customizes the functor
     GreaterThan_X(int checkValue)
     { _x = checkValue; }

     bool operator()(int number)
     {
          return number > _x; // Flexibility!
     }

private:
     int _x; // Compare against this, not 20
};
```

```
// Use a functor with flexibility!
int over10 = std::count_if(numbers.begin(), numbers.end(), GreaterThan_X(10));
int over20 = std::count_if(numbers.begin(), numbers.end(), GreaterThan_X(20));
int over35 = std::count_if(numbers.begin(), numbers.end(), GreaterThan_X(35));
```

3 different results from 1 “function”
(i.e., 3 instances of the functor)

Functors allow for flexibility;
small code can do big work!

Templates and std::function

- A lot of STL functionality uses iterators over some range of elements.

```
bool GreaterThan20(int number);
struct GreaterThan_X { };

int count1 = std::count_if(numbers.begin(), numbers.end(), GreaterThan20);
int count2 = std::count_if(numbers.begin(), numbers.end(), GreaterThan_X(20));
```

Templates are used to support function pointers and functors (and also, any data type).

```
template <typename CustomComparison>
void SortStuff(vector<int>& numbers, CustomComparison functor_OR_functionPointer)
{
    // Invoke THE THING... Whatever it is
    if (functor_OR_functionPointer(numbers[i], numbers[i + 1]))
    { // swap elements }
}
```

There's another alternative, especially if you want to store a function pointer (or functor).

```
#include <functional>
std:function<>
```

What Is std::function?

- **Wrapper** class: A clean interface around some functionality
- Encapsulates some **callable** element, anything that be called/invoked:
  - function pointer, functor, even lambda expressions (more on these later!)

```
std::function<returnType(parameter list)> variableName;

// Stores a function taking in an integer, and returning a boolean
std::function<bool(int)> someFunction;

// Stores a function that takes no parameters, returns nothing
std::function<void()> otherFunction;

// And so on, to store any kind of function
std::function<int(int, char)> otherFunction2;
std::function<float(vector<string>&, bool)> otherFunction3;
```

std::function is the modern, recommended way of storing **any** kind of function-as-data variable.

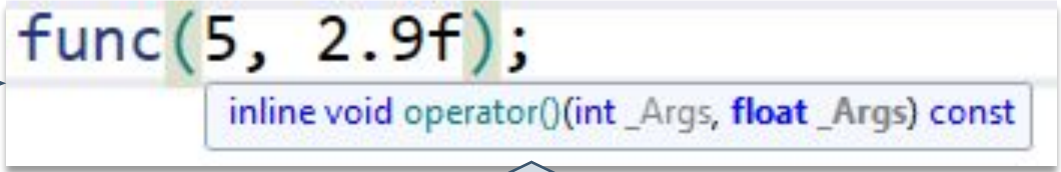
Using a std::function object

```cpp
void Foo(int a, float b)
{
    // Do some stuff
}

int main()
{
    // Initialize an instance of the class
    std::function<void(int, float)> func = Foo;

    // Call it like a function... Seems familiar...
    func(5, 2.9f);

    return 0;
}
```



```
func(5, 2.9f);
    inline void operator()(int _Args, float _Args) const
```

Behind the scenes, a functor!

Storing std::function<> in Classes

```
class FunctionHolder
{
    void (*_singleAction)();     // Store a single function
    vector<void(*)()> _actions;  // Store multiple functions
public:
    void AddAction(void (*a)()); // Add a pointer to the vector
    void DoAllActions();         // Call all stored functions
};
```

What if you wanted to store functions other than void()? What about return types, or parameters?

We can just add more templates to the equation!

```
#include <functional> // Need this f

class FunctionHolder
{
    std::function<void()> _singleAction;           // One stored std::function
    vector<std::function<void()>> _actions;   // Multiple std::function objects
public:
    void AddAction(std::function<void()> a); // Add a function
    void DoAllActions();                     // Call all stored functions
};
```

Same overall concept, storing functions in a different (better!) way

Storing std::function<> in Classes

```
#include <functional>
template <typename FunctionType>
class FunctionHolder
{
    std::function<FunctionType> _singleAction;    // One stored function
    vector<std::function<FunctionType>> _actions; // Many stored functions
public:
    void AddAction(std::function<FunctionType> a);    // Add a function
    void DoAllActions();                               // Call all stored functions
};
```

Still using std::function for storage, but templates to determine the type

```
FunctionHolder<bool(int)> holder;
holder.AddAction(GreaterThan20);     // Add a function pointer
holder.AddAction(GreaterThan_X(20)); // Add an instance of a functor

FunctionHolder<void()> holder2;
FunctionHolder<int(bool, double)> holder3;
```

Most “modern” C++ uses lots and **lots** of templates.

Templates and generic programming are very powerful, though not always easy to work with!

Recap

- Functors (function objects) are classes that can act like functions.
  - They implement operator() – the call operator.
- They are similar, but an alternative to function pointers.
  - Function pointers were here first!.
- Functors can contain variables and other functions.
  - They are more flexible in how they operate.
- They (and function pointers) are useful in many STL algorithms.
  - Many algorithms use iterators and predicates to operate on ranges.
- Templates and std::function make it easier to use any kind of callable element.

Conclusion

Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.