WARWICK
THE UNIVERSITY OF WARWICK

Special member functions

Tom Latham

Reminder: Compiler-provided special member functions

- Last week we encountered some "special member functions" of classes that handle resource management and are called:
  - Copy constructor
  - Move constructor
  - Copy assignment operator
  - Move assignment operator
  - Destructor
- If you do not specify them, the compiler creates them for you
- In the majority of cases the compiler-provided versions will work perfectly well, which is why we had not had to know about them up to then!
- We mentioned the “rule of all or nothing”. For more details see:

http://en.cppreference.com/w/cpp/language/rule_of_three

http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-zero

http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-five

And links therein.

So what's the problem?

- Last week we just needed to explicitly tell the compiler that we were happy with the implementation that it provided by writing in the function declarations and adding "= default" at the end
- But what implementation does the compiler provide?
- And under what circumstances might one need to write your own implementation?
- Let's look at these questions with the help of some illustrative code:

https://github.com/cpp-pg-mpags/SpecialMemberFunctions

Please clone this repo (if you haven't already done so from looking at the Containers Overview slides) and we'll take a look at it together…

What does the compiler provide?

- The compiler-provided **destructor** does nothing
- The compiler-provided **copy constructor** and **copy-assignment operator** make a copy of each data member, i.e. for built-in types (including raw pointers) their values are copied, and for class types their copy constructor or copy-assignment operator are invoked
- Similarly, the **move constructor** and **move-assignment operator** will move each data member, i.e. for built-in types (including raw pointers) their values will be copied, and for class types their move constructor or move-assignment operator are invoked
- As you can see from the code in the master branch, this is all fine for our MyArray type where the underlying storage is statically sized
- Checkout the **ImplementDefaultsWithPrintout** branch to see my guess as to how they are probably implemented (with some added printout)

What if I don't want my object to be copied (or moved)?

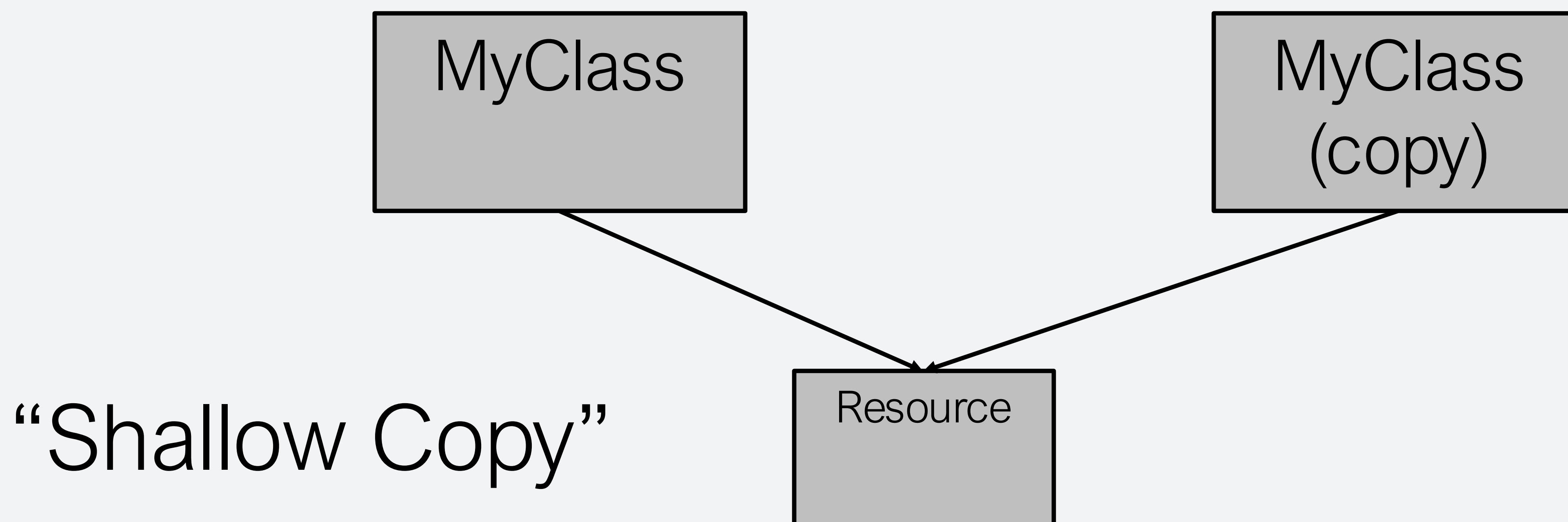
- This can be achieved rather easily in modern C++
- Check out the **DeleteCopyAndMove** branch and take a look at MyArray.hpp
- In a similar way as we did for the Cipher class last week to explicitly request the compiler-default versions, we can just write the function declarations in the header file and add "`= delete`" to the end to explicitly remove any of these copy/move constructors/assignment operators
- NB you can prevent copying (by using "`= delete`") but still allow moving (by using "`= default`") if that makes sense for your object

When do I need to write my own?

- These default implementations can become problematic if your class has a data member that points to some resource that is either completely external or is dynamically allocated by the class
- This could be a database or file handle or a dynamically allocated object (in our illustrative case it is a dynamically allocated array)
- When copying, there is a choice to be made as to whether the new object should point to the same resource as the old one (so-called *shallow copy* where only the pointer is copied) or whether the resource being pointed to should also be copied (so-called *deep copy*)

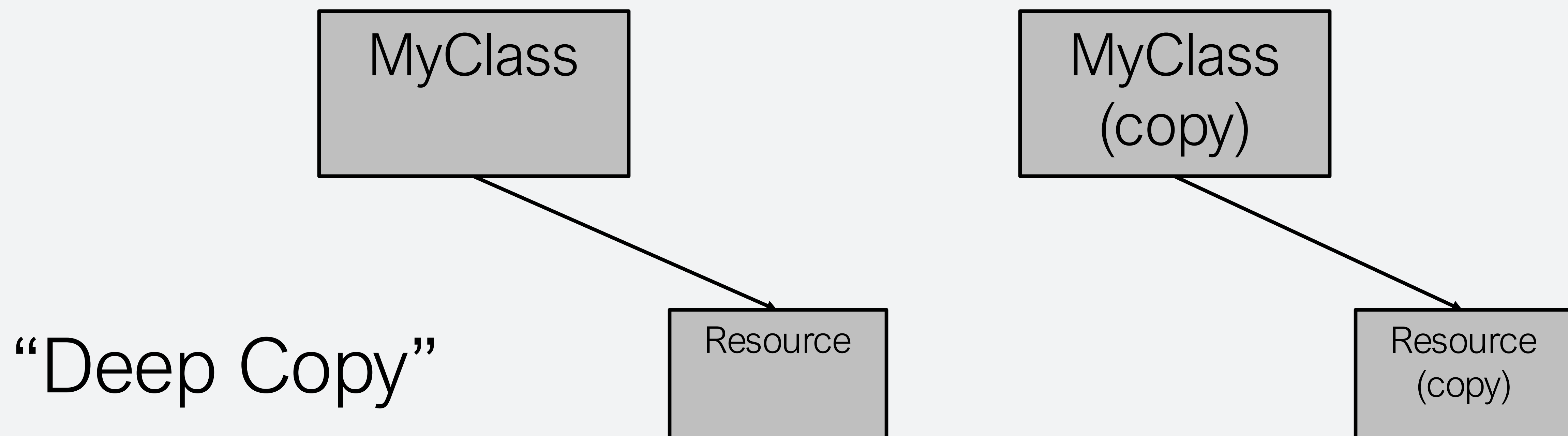
When do I need to write my own?

- These default implementations can become problematic if your class has a data member that points to some resource that is either completely external or is dynamically allocated by the class
- This could be a database or file handle or a dynamically allocated object (in our illustrative case it is a dynamically allocated array)
- When copying, there is a choice to be made as to whether the new object should point to the same resource as the old one (so-called *shallow copy* where only the pointer is copied) or whether the resource being pointed to should also be copied (so-called *deep copy*)



When do I need to write my own?

- These default implementations can become problematic if your class has a data member that points to some resource that is either completely external or is dynamically allocated by the class
- This could be a database or file handle or a dynamically allocated object (in our illustrative case it is a dynamically allocated array)
- When copying, there is a choice to be made as to whether the new object should point to the same resource as the old one (so-called *shallow copy* where only the pointer is copied) or whether the resource being pointed to should also be copied (so-called *deep copy*)



When do I need to write my own?

- If you’re using raw pointers then the compiler-default copying mechanism will give you the shallow-copy behaviour, which throws up serious questions about ownership and hence potential memory leaks or segmentation faults
- If you want deep-copy behaviour you must write your own copying functions and you must also implement a destructor to free the resource
- Regardless of which behaviour you want, it is best to use smart pointers to both simplify the code and clarify ownership
- For a shared resource, accessed with a `shared_ptr` data member, the compiler-provided shallow-copy behaviour is OK – much easier!
- But if you want deep-copy behaviour, or it is enforced using `unique_ptr`, you will have to write your own copying operations – NB `unique_ptr` has no copying operations itself by definition – it’s unique!
    - Check out the **DynamicStorageWithDefaults** branch to see this issue
    - Finally, check out the **DynamicStorage** branch to see the custom functions

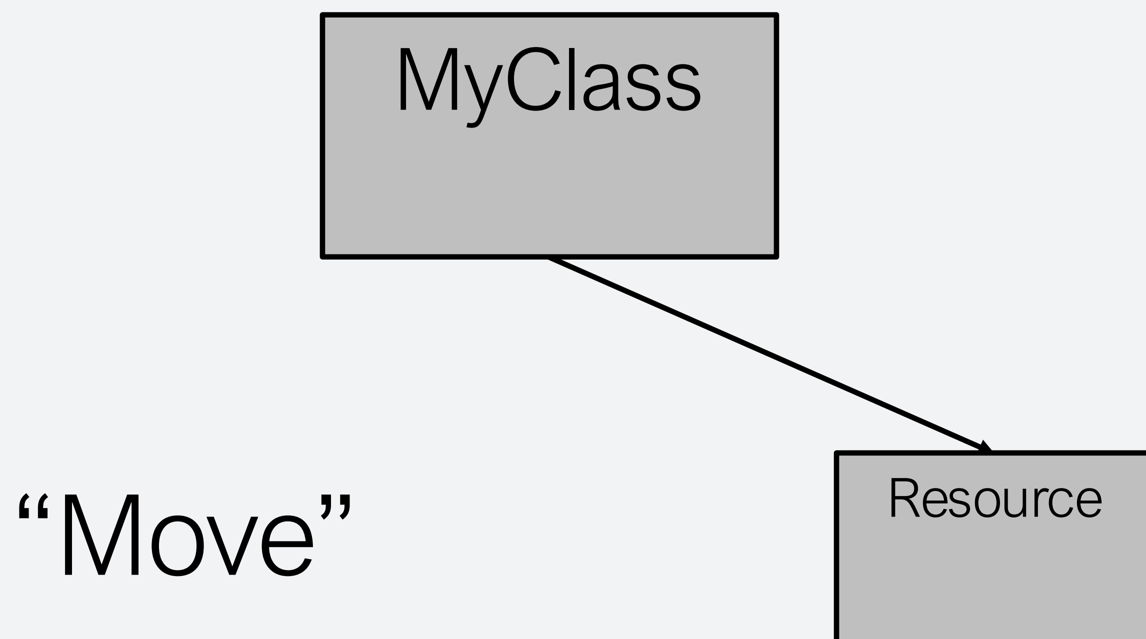
WARWICK
THE UNIVERSITY OF WARWICK

Move semantics

Tom Latham

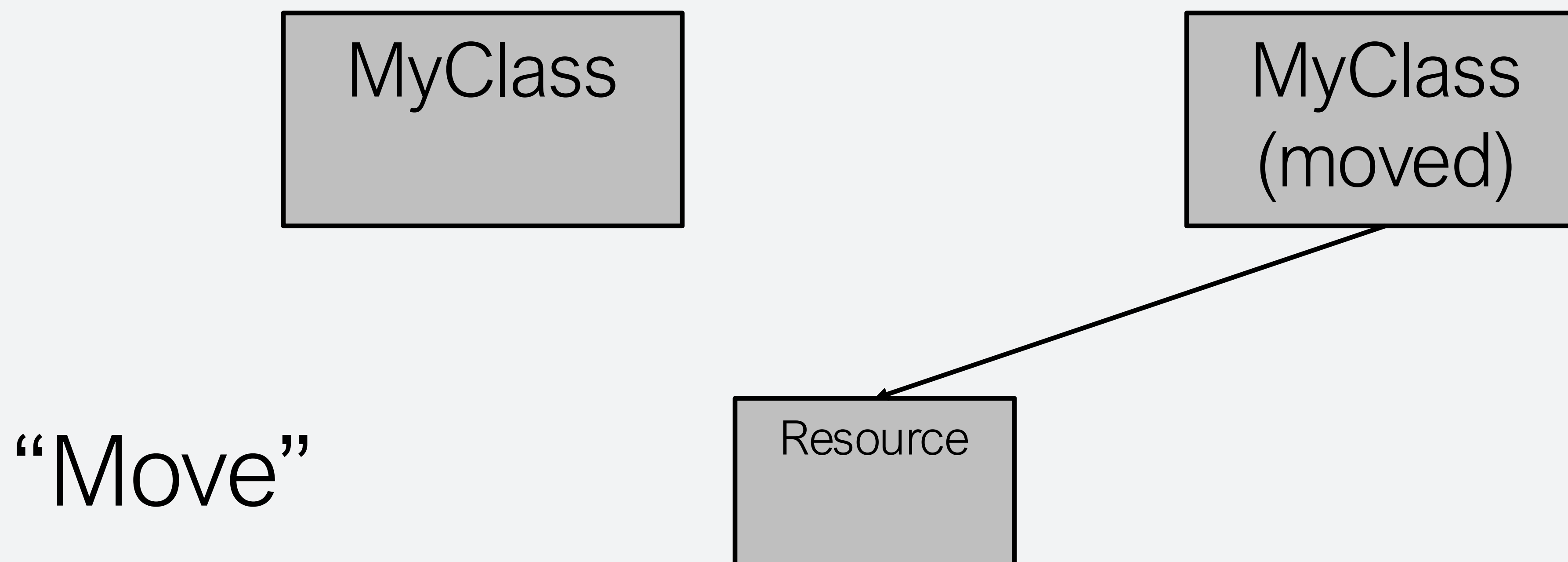
Move semantics

- Move semantics were introduced in C++11 to make certain operations more efficient when the object on the right-hand side of the operation is "expiring" (e.g. is a temporary object), termed an *xval*
- Essentially, where a copy operation would make a copy of all the dynamically allocated resources of an object, a move operation will "steal" those resources by copying the pointer to them and resetting the pointer in the original object so that it no longer points to them
- Since the original object is expiring this is fine since it no longer needs them



Move semantics

- Move semantics were introduced in C++11 to make certain operations more efficient when the object on the right-hand side of the operation is "expiring" (e.g. is a temporary object), termed an *xval*
- Essentially, where a copy operation would make a copy of all the dynamically allocated resources of an object, a move operation will "steal" those resources by copying the pointer to them and resetting the pointer in the original object so that it no longer points to them
- Since the original object is expiring this is fine since it no longer needs them



Move semantics

- Move operations should be invoked automatically when the right-hand object is temporary, e.g. the return from a function
- It is possible to explicitly indicate that an object can be "moved from" using the std::move utility function:

```
std::string mystr {"Hello"};

std::vector<std::string> myvec;
myvec.push_back( mystr );
myvec.push_back( std::move( mystr ) );
```

- Different operations will take place to construct the object in the vector for the two push_back calls above
  - A copy in the first case and a move in the second
- The state of mystr after it has been moved from is “valid but unspecified”

Where could we have used this in mpags-cipher?

- In a few places we make copies of strings, potentially some of these could be replaced by moves, or at least we could allow that option
- The input text argument to the `applyCipher` function is a good example
- At present it is an lvalue reference to a `const std::string` object but in one of the ciphers it is immediately copied into a local variable in the function and in the other two it is gradually copied (and the code could easily be rewritten to accommodate an immediate copy and then gradual modification)
- By overloading the `applyCipher` function, to additionally provide a version that takes an rvalue reference, we allow the caller to optionally do a move instead, which could be much more efficient if the text to be processed is very long
- See the **6.6** tag of https://github.com/MPAGS-CPP-2024/MPAGS-Code.git

Operator overloading

Tom Latham

WARWICK
THE UNIVERSITY OF WARWICK

Operator overloading

- The copy- and move-assignment operators are provided to allow you to perform assignment operations for your user-defined types in the same way as you do for the built-in types, i.e. using the = sign

- This is an example of operator overloading

- Built-in types in C++ also know how to perform mathematical operations: addition, subtraction, multiplication, division, etc.

- They can also be printed with std::cout <<

- More complex types can do more, like std::vector and its subscript[] operator

Operator overloading

- As with assignment, we can provide this functionality for our own types by providing specially named functions, either as members of our class or as free functions that take our class as an argument
- When the compiler encounters a line of code such as:

```
std::cout << employee << std::endl;
```

- It will look for a function called `operator<<` that takes a `std::ostream&` and an `Employee` as arguments

Stream operator

- We define the overload function like any other

```
std::ostream& operator<<(std::ostream& os, const Employee& employee)
{
  os << "  Name: " << employee.name() << std::endl;
  os << "Salary: " << employee.salary() << std::endl;
  return os; //Return the ostream object we were passed
}
```

- Similarly you can overload other operators, see: http://en.cppreference.com/w/cpp/language/operators
- Only overload those that make sense and are unambiguous!
  - Employee % Employee doesn't mean anything!