Introduction to Object Oriented C++

Introduction

Reaching back to the history of C++, Bjarne Stroustrup started out by creating a “C with classes” as part of his PhD work. From early on in the language’s development, classes were a focus, in an effort to add formalized object oriented features to the language.

As you might expect, the concept and syntax of a class in C++ is quite similar to what you are used to in Java. However, as an in-depth examination is taken, there are important differences and concepts that need to be highlighted in C++.

Trivia: The only difference between a `class` and `struct` is the default visibility of members. Default visibility for a `struct` is `public`, whereas it is `private` for a `class`.

Basic Class Structure

In Java the class is defined in a single `.java` file. Conventionally, in C++ a class is split into two sections, a declaration and definition (or implementation). The declaration is placed into a header file (using an extension like `.hpp`) and the definition in an implementation file (using an extension like `.cpp`). C++ doesn’t enforce separate header and implementation files, but it is recommended.

The general form of a class declaration in C++ looks like:

```
class <class name>
{
        <access specifier>:
                <members>
};
```

- `<class name>` is any valid identifier
- (optional) `<access specifier>` is any of `public`, `protected`, or `private`. If no access specifier is provided, default visibility is `private`.
- (optional) `<members>` is any number of data field or method declarations.

Note the semi-colon after the closing brace. Java does not need it, C++ requires it.

The class implementation file contains the definition for any of the methods defined in the declaration. Methods have a small bit of extra syntax from functions, in order to identify which class they belong to. The general form of a method in an implementation file is:

```
<return type> <class name>::<method name>(<parameters)
{
        … method body …
}
```

- `<return type>` The return type of the method. This may be void in the case there is no return value.
- `<class name>` The name of the class the method belongs to.
- `::` The scope resolution operator. This is placed between the `<class name>` and the `<method name>`.
- `<method name>` The name of the method, as declared in the class declaration.

The next two code sections demonstrate the full header and implementation file contents of a class.

--- Person.hpp ---

```
#pragma once
#include <string>

class Person
{
  public:
    Person(std::string nameFirst, std::string nameLast, unsigned short age);

    std::string getFullName();
    std::string getNameFirst() { return m_nameFirst; }
    std::string getNameLast() { return m_nameLast; }
    unsigned short getAge() { return m_age; }

  private:
    std::string m_nameFirst;
    std::string m_nameLast;

    unsigned short m_age;
};
```

--- Person.cpp ---

```
#include "Person.hpp"

Person::Person(std::string nameFirst, std::string nameLast, unsigned short age)
{
    m_nameFirst = nameFirst;
    m_nameLast = nameLast;
    m_age = age;
}

std::string Person::getFullName()
{
    return m_nameFirst + " " + m_nameLast;
}
```

There is a lot to discuss from just these two small sections of code. Some things are familiar from Java, some things are quite different.

Person.hpp

`#pragma once` : This is a compiler directive, telling the compiler if it has already processed this file, to not process it again. Header files are `#include`'d throughout a C++ program. In order to prevent the compiler from trying to parse the same code twice, this directive is necessary. In legacy code, when the `#pragma once` directive didn't exist at the time of writing, you might see a header file wrapped in a different style, like this…

```
#ifndef _PERSON_HPP_
#define _PERSON_HPP_
#include <string>
      … header file code goes here …
#endif
```

`#ifndef`, `#define`, and `#endif` are compiler directives that do what they sound like they should do. The `#ifndef` "if not defined" tests to see if `_PERSON_HPP_` has been defined or not. If it hasn't been defined, the code between the `#ifndef` and `#endif` is compiled. If it has been defined, the code between the two directives is not compiled. The `#define` directive defines `_PERSON_HPP_` (to exist, doesn't give it a value).

Class Declaration

A class is declared with the `class` keyword followed by the name for the class (any valid identifier). Following the class name is the declaration of the class, placed between opening and closing braces `{}`.

In C++ there is no access, or visibility, modifier before the `class` keyword, that concept doesn't exist in C++.

The Java compiler enforces the filename and the name of the class are the same. A class named `Person` in Java, must be placed in a file named `Person.java`. C++ does not have this requirement or enforcement. The name of a class has is not required to have any naming considerations with the header/implementation files. However, it is common practice to have the filenames follow the name of the classes in the manner demonstrated above.

Access Specifiers

They keyword `public` (and the colon) declare that any code following has public visibility. The `private` keyword is used to declare any code following it has private visibility. There can be any number of `public`, `protected`, or `private` declarations; including multiple of each. This is different from Java's approach where each class member is defined with a visibility modifier.

Constructors

Constructors have the same name as the class. The default constructor has no parameters, any number of overloaded constructors may be defined. C++ follows the same rule as Java regarding when the compiler creates or doesn't create default constructors...if any overloaded constructor is provided, the compiler does not create a default constructor.

Constructors – Direct Initialization / Member Initializer Lists

The way the `Person` class constructor was written follows the pattern from Java. However, C++ has another, preferred, syntax for initialization class members from constructor arguments. The approach is called *direct initialization*, it looks like the following:

```
Person::Person(std::string nameFirst, std::string nameLast, unsigned short age) :
    m_nameFirst(nameFirst),
    m_nameLast(nameLast),
    m_age(age)
{
}
```

Notice there is a colon `:` following the parameters section of the constructor, following by a series of member initialization statements. The code beginning with the colon and finishing before the opening brace is called the initializer list; not to be confused with an `std::initializer_list`! All of the code in the initializer list executes before the body of the constructor.

Who cares?

Remember that C++ has a reference type, an alias to an actual value. Remember also that reference variables can not be null. Therefore, if a member field of a class is defined as a reference, it must be defined before the body of a constructor begins execution. How do you do that? Set the value of the reference field in the initializer list!

It is recommended to initialize member fields using an initializer list. The values used to initialize do not have to come from the constructor parameters, they can be hard-coded values, function or method calls, or a single constructor parameter may be used to initialize multiple member fields.

Methods

Methods are declared in the header file, this is demonstrated with the various `get*` methods. Notice that `getFullName()` is declared, but not implemented, while the rest of the methods are declared and defined right in the header file. In C++ a method may be declared and defined in the header file, not requiring any code in the implementation file. It is common practice to place small, one-line, implementations in the header file, while all other methods are separately declared and defined.

Data Fields

Data fields are declared in the same manner as Java.

Person.cpp

The definition, or implementation, for methods is located in the corresponding implementation file.

The fist line in the implementation file is a `#include` directive to have the compiler insert the header file. By doing this, the compiler processes the `Person.hpp` header file and learns the declaration of the `Person` class. After processing the class declaration, the compiler processes the method implementations.

All constructors and methods are *scoped* to the `Person` class, using the *scope resolution* operator `::`, as part of the constructor or method name. This is why you see `Person::` in front of the constructor and method names.

The first item implemented in this file is the `Person` constructor. Because a constructor has no return type, no return declaration exists. Because this constructor is part of the `Person` class, the name is identified as `Person::Person`, then the parameters are provided the same as the declaration.

Side Note: C++ does not require parameters are given names in the declaration, but does require them for the implementation. Different people and organizations follow different style rules regarding this. Throughout these notes both the declaration and implementation provide the parameter names in both places.

Class Usage

In Java, all class instances (objects) are allocated from the heap. In C++, an instance (object) may exist on the stack or be heap allocated. This section discusses using the `Person` class as a stack object and dynamically allocated using raw pointers.

The use of the `Person` class is demonstrated in the following code sample, which is located in a file called `main.cpp`.

--- main.cpp ---

```
#include "Person.hpp"

#include <iostream>

int main(int argc, char* argv[])
{
    Person p1("Lisa", "Smith", 22);
    std::cout << p1.getFullName() << std::endl;

    Person* p2 = new Person("Larry", "Jones", 33);
    std::cout << p2->getFullName() << std::endl;
    delete p2;

    std::shared_ptr<Person> p3 = std::make_shared<Person>("Linda", "Sable", 44);
    std::cout << p3->getFullName() << std::endl;

    return 0;
}
```

The first two lines of this file include some necessary header files. In particular, the `Person.hpp` file is included to inform the compiler of the `Person` class. Without it, the compiler won't know how to resolve `Person`. This takes some getting used to when coming from Java, where the Java compiler is able to resolve symbols more readily.

The first line inside the `main` function demonstrates declaring an object on the stack; something not possible in Java. In this code, `p1` is the variable used to access the value of the `Person` object. Additionally, the `Person` object is placed on the program stack, it is not heap allocated.

** Draw diagram of the stack allocated object **

In Java, the first statement would be written as:

```
Person p1 = new Person("Lisa", "Smith", 28);
```

** Draw heap allocation diagram **

In Java `p1` is a reference variable used to refer to the dynamically allocated (from the heap) `Person` object. This is the only way in Java to create a class instance. There is an equivalent to this in C++, that will be demonstrated in just a bit.

The following statement shows how to access the `getFullName()` member of the `p1` object.

```
std::cout << p1.getFullName() << std::endl;
```

This sends the name to `std::cout` using the *dot operator* `.` to access the `getFullName()` method. The dot operator is used to access the members of any *value type* that has members. I say value type, rather than stack allocated, to differentiate accessing an object through a pointer or a value. Both stack and heap allocated objects can be accessed as values or pointers, depending on how the code is written. Java has no similar differentiation, objects are only used through references and their members accessed using the dot operator.

The third line of code inside the main function demonstrates heap allocation of an object.

```
Person* p2 = new Person("Larry", "Jones", 56);
```

The `p2` variable is declared as a pointer type and assigned the result of the memory allocation on the right-hand side of the expression. This is the same approach as Java.

The following statement shows how to access the `getFullName()` member of the `p2` object.

```
std::cout << p2->getFullName() << std::endl;
```

This sends the name to `std::cout` using the *pointer operator* `->` to access the `getFullName()` method. The pointer operator is used to access the members of any pointer type that has members.

** Show to to derefrence and then use the . operator to access members **

The `delete` operator is used to return the heap allocated `Person` instance pointed to by the `p2` variable. This is not necessary in Java, because the garbage collector reclaims unused memory automatically.

The final two statements demonstrate using and accessing the `Person` class with smart pointers.

```
std::shared_ptr<Person> p3 = std::make_shared<Person>("Linda", "Sable", 44);
std::cout << p3->getFullName() << std::endl;
```

The first line declared a shared pointer over the `Person` class, storing it in the variable `p3`. The right-hand side of the expression dynamically allocates it as a shared pointer. From there, the syntax for accessing the members of `p3` is the same as raw pointers, using the pointer operator.

Unlike the raw pointer example where the delete operator was necessary to return the dynamically allocated memory, smart pointers have no such requirement. Once the smart pointer goes out of scope, the memory is returned to the operating system.

Translation Unit & Separate Compilation

Two concepts are necessary for C++ programmers to understand: *translation unit*, and *separate compilation*.

As has already been seen with compiler directives like `#ifndef`, `#define`, `#pragma` and `#include,` the developer of C++ code needs to not only think about how to give instructions to the compiler using C++, the developer needs to think about the operation of the overall C++ compiling process itself. With compiler directives, the programmer is giving the preprocessor instructions on how to process a file before the compiler processes the C++ code. This means the programmer is involved in the process of how the C++ files/code are converted, or translated, into executable instructions.

An important concept of this process is the *translation unit*. After the preprocessor has finished processing a (`.cpp`) source file, the resulting file is called a translation unit. The compiler takes a resulting translation unit and *translates* the C++ code into object code that is later combined with other object code, by the linker, and turned into an executable file (e.g., .exe, .dll, .so). The only information about types available to the compiler is in the translation unit; well, and the compiler itself for built-in language types. If a type is referenced in the translation unit, but not declared, a compiler error results. Therefore, it is necessary to `#include` all the header files that declare the types used in the final translation unit. For each translation unit the compiler starts over with its knowledge of types, no knowledge transfers between compilation of translation units.

Because each translation unit must be self-contained with respect to types used, it is essential to reduce the work of the compiler by writing separate header and implementation files. The header files contain the declarations and are `#include`'d as necessary. The implementation files contain the definitions, `#include`'ing header files as necessary.

Understanding the concept of a translation unit makes it easy to describe *separate compilation*. Separate compilation is where code only knows of type and function declarations, not needing access to the implementations. Type and function declarations are placed in header files and separately, the definitions for those types and functions are placed in an implementation file. They are, therefore, separately compiled; Simple.

Warning: When coming from Java, it is tempting to think, "Oh, I can still write all the code in the header file just like I'm used to with Java", but that is a mistake. Each time the C++ compiler begins work on a *translation unit* it also compiles all the code from the included header files. Each time a header file is visited for the first time, per translation unit, it must process all the code in the header file. If a header file is included in more than one translation unit in a project, it requires the compiler to process all of the code each time it is included. This is why it is generally okay to declare and define one-line methods in the header file, but not methods that are non-trivial.

Operator Overloading

Reference: https://stackoverflow.com/questions/4421706/what-are-the-basic-rules-and-idioms-for-operator-overloading

One of the most interesting, and unique, capabilities of C++ is the ability to define, and define, how the operators work on user defined classes. The standard operators include mathematical operators such as `+`, `-`, `*`, `/` and also relational operators like `==` `!=`.

Java does not have any analogous capability, it has a different philosophy when it comes to (not doing) operator overloading.

Operators are (generally, but not always) implemented as methods in a class. An overloaded operator is a function and can be called using either the usual operator syntax or by the name of the function. There is no single general form for all operators, each can have differences that affect the return type and number and type of parameters.

Let's start by taking a look at the mathematical operators. Typically these operators are overloaded in the context of a class that provides a mathematical capability, such as a matrix library. However, in order to show that isn't the only domain for them, I'll demonstrate their use in the context of a couple of (very simple) classes that represent a company and its employees.

We'll start with an `Employee` class with the following declaration & implementation:

```
class Employee
{
  public:
    Employee(std::string nameFirst, std::string nameLast, unsigned short years);

    std::string getFullName();
    std::string getNameFirst() { return m_nameFirst; }
    std::string getNameLast() { return m_nameLast; }
    unsigned short getYearsOfService() { return m_yearsOfService; }

  private:
    std::string m_nameFirst;
    std::string m_nameLast;

    unsigned short m_yearsOfService;
};

Employee::Employee(std::string nameFirst, std::string nameLast, unsigned short years)
:
    m_nameFirst(nameFirst),
    m_nameLast(nameLast),
    m_yearsOfService(years)
{
}

std::string Employee::getFullName()
{
    return m_nameFirst + " " + m_nameLast;
}
```

Next, a `Company` class with the following initial declaration & implementation (we'll be adding operators to it):

```
class Company
{
  public:
    Company(std::string name);

    std::string getName() { return m_name; }
    std::size_t getNumberOfEmployees() { return m_employees.size(); }
    std::shared_ptr<Employee> findbyName(std::string nameFirst, std::string nameLast);

  private:
    std::string m_name;
    std::vector<std::shared_ptr<Employee>> m_employees;
};

Company::Company(std::string name) :
    m_name(name)
{
}

std::shared_ptr<Employee> Company::findbyName(std::string nameFirst, std::string nameLast)
{
    auto result = std::find_if(
        m_employees.begin(), m_employees.end(),
        [nameFirst, nameLast](std::shared_ptr<Employee> employee) {
            return nameFirst == employee->getNameFirst()
                   && nameLast == employee->getNameLast();
        });

    return *result;
}
```

The `Company` class currently lacks a way to add new `Employee` instances. One approach is to write an `addEmployee(…)` method, and that is probably a great idea. Another approach, is to overload the `+=` operator to allow adding of `Employee` instances. The general form of the `+=` operator is:

```
<class>& operator+=(const <type>& rhs) { … body… }
```

Note the use of a `const` reference as the parameter. The operator does not require this, the parameter may be passed in by value, but there are many reasons not to do that. Passing by reference prevents a copy being made. Making the reference `const` prevents any accidental changes from being made to the parameter. Again, there is no requirement the parameter must be `const` or a reference. If it makes sense for your application, the parameter can be passed by reference and modified...but I don’t recommend ever doing that!!

For the `Company` class, the declaration and implementation are:

```
Company& operator+=(const std::shared_ptr<Employee>& employee);


Company& Company::operator+=(const std::shared_ptr<Employee>& employee)
{
    if (employee == nullptr)
        return *this;

    this->m_employees.push_back(employee);

    return *this;
}
```

With this operator overloaded, the following code instantiates a `Company` object and adds three `Employee` instances:

```
Company myCompany("Big Business");

myCompany += std::make_shared<Employee>("Lisa", "Smith", 12);
myCompany += std::make_shared<Employee>("Larry", "Stackhouse", 8);
myCompany += std::make_shared<Employee>("Linda", "Sable", 2);
```

Pretty cool!

Let's go another step and use the `-=` operator to remove an `Employee` from a `Company`. Start with the -= operator declaration and implementation:

```
Company& operator-=(const std::shared_ptr<Employee>& employee);

Company& Company::operator-=(const std::shared_ptr<Employee>& employee)
{
    if (employee == nullptr)
        return *this;

    auto iterator = std::remove_if(
        m_employees.begin(), m_employees.end(),
        [employee](std::shared_ptr<Employee> test) {
            return *test == *employee;
        });

    m_employees.erase(iterator);

    return *this;
}
```

The implementation requires some explanation. The first thing to notice is the use of the `std::remove_if` algorithm, followed by the `.erase` member of the `m_employees` `std::vector`. `std::remove_if` takes a unary predicate that returns `true` if an element should be removed from the collection. The algorithm works by moving the elements that should not be removed to the front of the container and places the elements to be removed as unspecified values at the end. Following the `std::remove_if` algorithm, the container's `.erase` method needs to be called to remove the unspecified values. The reason for this is to improve the performance of erasing elements by removing them all in a single pass, rather than repeated passes. This is called the *Erase-Remove* idiom.

One more thing to notice in the `-=` operator, the test to see if two employees are equal to each other. There is no default equality `==` operator for a class. Therefore, it is necessary to add an overloaded `==` relational operator to the `Employee` class. It is the responsibility of the programmer to define what "equality" means. In this case, we'll say that equality is satisfied if the first and last names match, ignoring the years of service. Relational operators must return a `bool` result, or something than can be converted to a `bool`. The following is the declaration and implementation for the `==` operator on the Employee class:

```
bool operator==(const Employee& rhs);

bool Employee::operator==(const Employee& rhs)
{
    return m_nameFirst == rhs.m_nameFirst &&
           m_nameLast == rhs.m_nameLast;
}
```

With this operator in place, the test inside the `std::remove_if` predicate will compile and return true only if the `Employee` object names match. The `-=` operator is complete and can be used as the next code segment demonstrates:

```
myCompany -= myCompany.findbyName("Larry", "Stackhouse");
```

Assignment Operator

Consider the following code:

```
Employee e1("Luke", "Seamons", 0);
Employee e2("Lapriel", "Sanders", 0);

e1 = e2;

std::cout << e1.getFullName() << std::endl;
```

The output from this code is "`Lapriel Sanders`". The assignment operator `=` was used to assign the value stored in `e2` into `e1`. The assignment operator may be overloaded, but isn't for the `Employee` class. The code worked because the C++ compiler provides an automatic member-by-member assignment operator if none is provided. Normally C++ doesn't provide a default implementation for operators, they are overloaded or they don't exist, the assignment operator is an exception.

What does member-by-member assignment mean? It means the C++ compiler uses the assignment operator on all the data fields (regardless of visibility modifier) to copy from the right-hand side object into the left-hand side object. For `Employee`, the assignment operator was invoked on `m_nameFirst`, `m_nameLast`, and `m_yearsOfService`. `m_yearsOfService` is a primitive object, assignment for it means to copy the value. `m_nameFirst` and `m_nameLast` are `std::string` class types, defined by the standard library. The `std::string` class overloads the assignment operator, ensuring a true copy of the data from one string is made into another.

If we want to change the behavior of the assignment operator for `Employee`, we can overload the operator. Let's change the assignment behavior to add a "Copy of " prefix to all strings during assignment. The general form for the assignment operator is:

```
<class>& operator=(const <type>& rhs) { … body… }
```

For the `Employee` class, the declaration and implementation are:

```
Employee& operator=(const Employee& rhs);

Employee& Employee::operator=(const Employee& rhs)
{
    m_nameFirst = "Copy of " + rhs.m_nameFirst;
    m_nameLast = "Copy of " + rhs.m_nameLast;
    m_yearsOfService = rhs.m_yearsOfService;

    return *this;
}
```

Now, when `e1 = e2;` is called, "`Copy of `" is pre-pended to the first and last names. Again, like any of the operators, the C++ compiler doesn't enforce a rule on what must be done, that is up to the programmer to decide.

Additional Operator Overloading Notes

There are a few notes to mention about the mathematical operators.

- If you implement the `+` operator and the `=` operator, it does not mean the `+=` operator is also overloaded. Each operator must be individually implemented.
- If you implement the `<` operator and the `==` operator, it does not meant the `<=` operator is also overloaded.
- Remember there are pre and post `++` and `--` operators. The declaration for each is:
    - post-increment: `class operator++(int);`
        - The semantics for post-increment is to make a copy of the value, increment the value, then return the copy made before incrementing the value. Notice the return type for post-increment is by-value (copy).
    - pre-increment: `class& operator++();`
        - The semantics for pre-increment is to increment the value, then return the incremented value. Notice the return type for pre-increment is by-reference (alias).

There is a well-known Stack Overflow post that talks about the "basic rules and idioms for operator overloading". It is good to read and keep in mind as you are writing C++ code.

Lions, Tigers, R-Values References, and Constructors, o my!

There are seven topics to cover when speaking of constructors in C++. The first five are *default constructors*, *overloaded constructors*, *constructor delegation*, *copy constructors*, and *move constructors*. The sixth topic discusses r-value references, which are necessary to understand before discussing move constructors. Finally, the seventh topic, complementary to move constructors but not a constructor, is the *move assignment operator*. In addition to these topics, there is another constructor topic yet to come, *inheriting constructors*, which is presented in the section on inheritance.

To demonstrate each of the constructor types and the overall discussion of constructors, we'll write a `Matrix` class. This class will have just enough functionality to demonstrate the syntax and use of each of the constructor types and semantics. The complete class declaration from the `Matrix.hpp` file is:

```
class Matrix
{
  public:
    Matrix();
    Matrix(std::size_t cols, std::size_t rows);
    Matrix(initializer_list<initializer_list<int32_t>> list);
    Matrix(const Matrix& matrix);
    Matrix(Matrix&& matrix);
    ~Matrix();

    Matrix& Matrix::operator=(Matrix&& rhs);
    int32_t& operator()(std::size_t row, std::size_t col);

    std::size_t getColumns() const { return m_cols; }
    std::size_t getRows() const { return m_rows; }

  private:
    std::size_t m_rows;
    std::size_t m_cols;
    int32_t** m_data;

    void buildMemory(std::size_t rows, std::size_t cols);
};
```

There is quite a bit to discuss from just this code. The first five items in the public section are different constructors, each of which is detailed in the next sub-sections. Following the constructors is a class destructor. The next line is a new type of assignment operator, a move assignment operator. Then the parenthesis operator is overloaded to provide array-like access (not to create a Functor class). *(I've since changed it to be a raw pointer, because shared pointers haven't been covered yet) The last item of interest is the shared pointer being used to manage the dynamic memory used to hold the matrix data.*

Shared Pointer – Data Management *(see note above)*

In order to better demonstrate the various constructor and related topics, the `Matrix` class provides semi-manual management of the memory used to store the data. Smart pointers are used, but because of the nature of the memory allocation, the data is allocated and removed using raw pointer techniques, but managed through a shared pointer. It is definitely possible, and probably recommended, to use either `std::array` or `std::vector` to manage the matrix data, but I wanted to work at a little lower level to give better insights into this topic.

At the time a matrix is created, memory needs to be allocated to store the matrix data, that is the purpose of the `buildMemory` method. This statement `int32_t** data = new int32_t*[rows];` allocated an array of pointers to integers (`int32_t*`). The intention is that each element (a row) in this array is a pointer to another array of integers used to hold the matrix values.

** Draw a diagram of how the memory is allocated **

The next step is to allocate an array for each row in the matrix, as the next code segment shows:

```
for (decltype(rows) row = 0; row < rows; row++)
{
        data[row] = new int32_t[cols];
}
```

After this loop is done, the memory for the matrix is allocated. The next step is to place the raw pointer under the management of a smart pointer (`std::shared_ptr`). Because the memory is already allocated, no need for `std::make_shared`, instead create an `std::shared_ptr` passing it the data pointer, along with a lambda to cleanup the memory when the reference count hits zero. The next code segment demonstrates this:

```
m_data = std::shared_ptr<int32_t*[]>(data, [rows](int32_t** data) {
        for (decltype(rows) row = 0; row < rows; row++)
        {
                delete data[row];
        }
        delete[] data;
});
```

The second parameter to the `std::shared_ptr` is a lambda that is invoked when the reference count hits 0, it cleans up the dynamically allocated memory.

Default Constructors

The first of the five constructors is a default constructor.

A default constructor is one which has no arguments, or has defaults for all arguments. Just as in Java, if there is no user defined default constructor, the compiler writes a trivial (empty body) one. Similarly, if any constructor is written, the compiler does not provide a default constructor.

Constructor Delegation

Java has the concept of *constructor chaining*, which allows the developer to specify which super class constructor is chained by using the `super` keyword to ensure a specific constructor from the parent class is invoked. C++ allows the same, but the term *constructor delegation* is used instead. The syntax for constructor delegation is different from the approach Java takes, it follows the pattern of class member initializer lists.

The Matrix default constructor uses constructor delegation to invoke the `Matrix(std::size_t rows, std::size_t cols)` overload constructor.

```
Matrix::Matrix() :
    Matrix(2, 2)    // Delegate to an overloaded constructor
{
}
```

Overloaded Constructors

The second and third constructors are overloaded constructors. The second constructor accepts a number of rows and column and builds a matrix of that size, initializing the values to 0. The code for this constructor is shown below:

```
Matrix::Matrix(std::size_t rows, std::size_t cols)
{
    buildMemory(rows, cols);
}
```

The third constructor, shown next, accepts an `std::initializer_list` (nested) and constructs a matrix based on the values stored inside.

```
Matrix::Matrix(initializer_list<initializer_list<int32_t>> list) :
    Matrix(list.size(), list.begin()->size())
{
    std::size_t r = 0;
    for (auto row = list.begin(); row != list.end(); row++, r++)
    {
        std::size_t c = 0;
        for (auto column = row->begin(); column != row->end(); column++, c++)
        {
            m_data[r][c] = *column;
        }
    }
}
```

This constructor begins by delegating to an overloaded constructor to build the matrix. Then, in the body of the constructor, the `std::initializer_list` is iterated through, taking values from it and placing them in the matrix. Providing this kind of overloaded constructor allows the following declaration to work:

```
Matrix m({{0, 1, 2},
          {3, 4, 5},
          {6, 7, 8}});
```

Copy Constructors

The fourth constructor is a special constructor known as a *copy constructor*. It has a unique signature, which makes it readily identifiable as the copy constructor for a class. The general form is:

```
<class>(const <class>& obj);
```

Where `<class>` is the name of the class. The copy constructor accepts a reference (preferably a `const` reference) to an object of the class itself.

If a copy constructor is not provided by the programmer, the C++ compiler provides a default member-by-member copy. The member-by-member copy is exactly the same as what happens with the default assignment operator the compiler provides (if one is not provided by the programmer).

A copy constructor is called whenever a new object is created and initialized with the data from another object of the same class type. The following shows an example of when a copy constructor is invoked:

```
Matrix m1({{0, 1, 2},
           {3, 4, 5},
           {6, 7, 8}});
Matrix m2 = m1;
```

When `m2` is declared, it is initialized with the data from `m1`, which happens during the copy constructor.

Another typical time a copy constructor is invoked is when an object is passed by-value as an argument into a function or method. In this case, a copy of the object needs to be made, the copy constructor is used to make that copy.

When an object is returned by-value from a function or method, the copy constructor is called...but, stay tuned for move constructors, they can modify this behavior!

What needs to be done in a copy constructor. The values from one object need to be copied into the object being created. The semantics of that copy are left to the programmer, but usually is means a deep copy should be made. The copy constructor for the Matrix class is:

```
Matrix::Matrix(const Matrix& matrix)
{
    buildMemory(matrix.m_rows, matrix.m_cols);

    for (std::size_t row = 0; row < m_rows; row++)
```

```
        {
            std::memcpy(m_data[row], matrix.m_data[row], sizeof(int32_t) * m_cols);
        }
    }
```

The first step in this constructor is to create its own memory, rather than simply copying the std::shared_ptr from the source object. After creating the memory, the values from the source matrix are copied into the newly created object.

R-Value References

Additional Reference: http://www.thbecker.net/articles/rvalue_references/section_01.html

The concept of an *r-value reference* is more easily explained by starting with an *l-value*. An l-value is an expression that appears on the left or right hand side of an assignment operator. An l-value is sometimes known as a "locator value"; you can think of l-value as meaning "left value", but that isn't a great description. Consider the following code:

```
int x = 44;
int y = 66;
y = x;
```

In all three of these examples `x` and `y` are l-values, including the use of `x` on the right-hand side of the assignment operator in the last statement.

An *r-value* is an expression that can only appear on the right-hand side of the assignment operator. The following demonstrates:

```
int x = 44;
int y = 66;
int z = x * y;
```

In these statements, the expression `x * y` is a r-value, it can only exist on the right-hand side of an assignment operator; doesn't make any sense (in C++) for it to appear on the left.

Now that we understand l-values and r-values, let's move (no pun intended...you'll see) to l-value references. An l-value reference is a reference (an alias) to a value stored in memory. Consider the following code:

```
int x = 44;
int& xRef = x;
```

The variable `xRef` is an l-value reference to the variable `x`. You already know this, this is the reference type already presented! Looking back at the statement int `z = x * y;`, we know that `x * y` is an r-value. If it is possible to obtain a reference to an l-value, shouldn't it be possible to obtain a reference to the `x * y` r-value. Yes, you can, just like this…

```
int x = 44;
int y = 66;
int&& xyRef = x * y;

std::cout << xyRef << std::endl;
```

R-value references are declared using `&&` along with the type.

More motivation for r-value references...

When talking about copy constructors, one of the places they can be invoked is when a copy of an object is made while returning it by-value from a function/method. Let's add a function that creates a matrix, initializes the values, then returns it by-value:

```
Matrix makeMatrix(std::size_t rows, std::size_t cols)
{
    Matrix m(rows, cols);
```

```
        int32_t value = 0;
        for (std::size_t row = 0; row < rows; row++)
        {
            for (std::size_t col = 0; col < cols; col++)
            {
                m(row, col) = value;
                value++;
            }
        }

        return m;
    }
```

The last statement, `return m;`, results in a copy of the variable `m` being made, using the copy constructor, which is then returned and either used to initialize a new matrix or copied again, using the assignment operator, to an already existing object; ugh, two copies!

** demonstrate this by commenting out the move operations and leaving only the regular operations...will need to add the assignment operator **

We can use the `makeMatrix` function like `Matrix m = makeMatrix(2, 4);` where `m` is an l-value. Alternatively, we can use `makeMatrix` like `Matrix&& m = makeMatrix(2, 4);` where `m` is an r-value reference. In this case, `m` is an r-value reference to the object constructed by the copy constructor.

For the worst-case use of a copy constructor and assignment operator, consider the following code:

```
    Matrix m1;
    m1 = makeMatrix(2, 4);
```

The first line causes the default constructor to be invoked. The second line causes the copy constructor to be invoked ***and then*** the assignment operator to assign the copy constructed object to `m1`. In other words, two copy operations took place to update `m1` with the results of the `makeMatrix(2, 4)` call. This is our motivation for the next two subjects, move constructors and move assignment, which is where we see why it is important to understand r-value references.

Move Constructors

A *move operation* is one in which the ownership of values from one object are transferred (or swapped) to another object. This primarily applies to things like pointers. In a move operation, the ownership of a pointer is transferred from a source object to a destination object. Move operations are appropriate when the source object is temporary and/or is about to be destroyed.

The fifth constructor is a special constructor known as a *move constructor*. A move constructor is is automatically invoked when the object being created is temporary, such as when an object is returned by-value from a function/method. Earlier this was discussed as being where a copy constructor is invoked, which is true, unless a move constructor exists, which is invoked instead of the copy constructor.

It has a unique signature, which makes it readily identifiable as the move constructor for a class. The general form is:

```
    <class>(<class>&& obj);
```

Where `<class>` is the name of the class. The move constructor accepts an r-value reference to an object of the class itself.

As noted above, the semantics for a move constructor are to transfer ownership from a (temporary) source object to a destination object. This is more easily demonstrated with the `Matrix` move constructor:

```
    Matrix::Matrix(Matrix&& matrix)
    {
        m_rows = matrix.m_rows;
        m_cols = matrix.m_cols;
        m_data = matrix.m_data;

        matrix.m_rows = 0;
        matrix.m_cols = 0;
        matrix.m_data = nullptr;
    }
```

This code shows the temporary object (the r-value reference) transfers ownership of its values to the (destination) object being initialized. To emphasize, the move operation does not copy the values from the source to the destination, it transfers those values/ownership to the destination. This is particularly important for (smart) pointers.

** Draw a diagram illustrating this transfer of ownership **

Some say that rather than transferring ownership, the move constructor should swap ownership. Whatever project you work on, follow the project guidelines, don't fight over it.

Move Assignment Operator

Move assignment occurs when a temporary, or about to be destroyed, object is being assigned to another object. Referring to the previous section regarding the move constructor. When an object is returned from a function/method, it is first copy constructed and then assigned to the destination object. The overhead associated with copy construction can be reduced by creating a move constructor, but we are still left with the overhead of making a copy during the assignment operation. In the same line of thinking as the move constructor, a move assignment operator can be provided.

Move assignment is very similar to what takes place in a move constructor, but rather than transferring ownership, ownership is swapped. Let's take a look at the `Matrix` move assignment operator:

```
    Matrix& Matrix::operator=(Matrix&& rhs)
    {
        if (this != &rhs)
        {
            std::swap(m_rows, rhs.m_rows);
            std::swap(m_cols, rhs.m_cols);
            std::swap(m_data, rhs.m_data);
        }

        return *this;
    }
```

The key difference between the move constructor and move assignment is the swap of the values. This allows the source object to go out of scope, and as it does, decrease the reference count on any smart pointers (which may go to 0 and the memory release) or allow other allocated resources to be released. The move assignment operator is not responsible for freeing any resources, that is left to take place normally as the object goes out of scope.

** Draw a diagram illustrating this swap of ownership **

R-Value References & Range-Based For Loops

Now that r-value references are understood, let's look at another place where they are useful, the range-based for loop. Consider the following code:

```
    std::vector<std::string> cities =
          { "Paradise", "Hyrum", "Nibley", "Hyde Park", "Smithfield", "Newton" };

    for (auto city : cities)
    {
          std::cout << city << std::endl;
```

```
}
```

The type for `city` is `std::string`, meaning that a copy of each element in the cities `std::vector` is made and then used in the loop. The usual solution to prevent the copy is to re-write the loop using a (const) l-value reference, as follows:

```
for (const auto& city : cities)
{
	std::cout << city << std::endl;
}
```

Another alternative is to use an r-value reference, rather than an l-value reference:

```
for (auto&& city : cities)
{
	std::cout << city << std::endl;
}
```

In this case, there isn't an advantage to using an r-value reference. Where it becomes an advantage is if there is a function used to generate or return values used in the loop, then copies can be avoided by capturing the (temporary) objects returned from the generator function as r-value references, and the objects themselves having move constructors.

Destructors

A destructor is a special function automatically executed when an object goes out of scope; it is the complement to a constructor. Java classes do not have the concept of a destructor. Instead, Java relies upon the interface `Dispose` to provide similar, but not exactly the same, functionality.

The name of a class destructor is the name of the class prefixed with the ~ (tilde) character. Just as a constructor has no return value, a destructor has no return value. While a constructor may accept parameters, a destructor has no parameters. A class may have any number of constructors (default, overloaded, move, etc), a class has either zero or one destructor.

The purpose of a destructor is to release any resources acquired during the lifetime of an object. A typical resource is dynamically allocated memory. With the use of smart pointers, there isn't a need to write a destructor as they clean up after themselves. But think about the smart pointer classes, how do they know when to release memory back to the OS? They have destructors that decrement reference counts and implement logic to release memory when the reference count hits 0. Other resource types that may need to be closed in destructors are file handles, database connections, or network connections.

I'll demonstrate the use of a destructor by using raw pointers for memory management in the `Matrix` class, rather than using smart pointers. Because of this, a destructor is needed to release the dynamically allocated memory.

The `Matrix` class makes use of a (double) raw pointer `int32_t**` to track the memory (rather than a smart pointer, to give us something to do in a destructor). The `buildMemory` method is:

```
void Matrix::buildMemory(std::size_t rows, std::size_t cols)
{
    m_rows = rows;
    m_cols = cols;
    m_data = new int*[rows];
    for (decltype(rows) row = 0; row < rows; row++)
    {
        m_data[row] = new int32_t[cols];
        std::memset(m_data[row], 0, sizeof(int32_t) * cols);
    }
}
```

Because of the use of raw pointers, it necessitates the need for a destructor to release the memory as an object goes out of scope. The destructor is:

```
Matrix::~Matrix()
{
    if (m_data != nullptr)
    {
        for (decltype(m_rows) row = 0; row < m_rows; row++)
        {
            delete[] m_data[row];
        }
        delete[] m_data;
        m_data = nullptr;
    }
}
```

The first test in the destructor is to verify `m_data` actually points to something. In the move constructor, `m_data` is set to `nullptr` before it goes out of scope. If `m_data` wasn't set to `nullptr`, its destructor would (incorrectly) release the memory back to the OS. By setting `m_data` to `nullptr`, it ensures no attempt to incorrectly release memory occurs.

The main body of the destructor releases the dynamically allocated memory. It isn't absolutely necessary to set `m_data` to `nullptr`, because the object is going out of scope.

Rule Of Five

Reference: https://en.wikipedia.org/wiki/Rule_of_three_%28C++_programming%29

The rule Rule of Five in C++ suggests that if any one of the following are provided by the programmer, then the default implementations of the others are not sufficient and all five should be provided. These five items are:

- Copy Constructor
- Move Constructor
- Assignment Operator
- Move Assignment Operator
- Destructor

The need to include a destructor depends on the nature of the class. If the class is using smart pointers, and no other resources, a destructor is likely not necessary, but the other four items may need to be provided. It might be more appropriate is most situations to say there is a Rule of Four.

Class Inheritance & Polymorphism

C++ and Java share a lot in common when it comes to inheritance and polymorphism. Both have class-based inheritance, both have the concept of virtual (or overridable) methods, both have the concept of abstract methods, and both have the concept of abstract classes. There are some differences in terminology and implementation specifics, but both languages share these capabilities. As you might expect by now, C++ has more detail and nuance than Java, where necessary these are highlighted.

All of the discussion in this section is based on the same set of code. The code is a highly simplified approach for how one might go (or might not) about implementing classes for a game. There is a base `Entity` abstract class, from which two concrete classes are derived, `Monster` and `Vehicle`. Using these three classes, the core topics in inheritance and polymorphism are covered.

Terminology

- *Base* or *parent* class. Java uses the term *super* class to refer to the class a *sub-class* is derived from. C++ prefers the terms *base* or *parent* class.
- *Derived* or *child* class. Java uses the term *sub-class* to refer to a class *derived* from a *super* class. C++ prefers the use of *derived* or *child* class.
- *Abstract method*. Java uses the keyword `abstract` to mark a method as abstract, one without an implementation. C++ uses the term *pure virtual method*, along with a different syntax.
- *Abstract class*. Java uses the keyword `abstract` to mark a class as abstract, one that can't be instantiated. C++ uses the term abstract class, but requires no specific syntax to mark it as abstract. Any class that has a pure virtual method, is by definition an abstract class. In Java, a class may be marked as abstract without having any abstract methods. In C++ a class is abstract only if it has one or more pure virtual methods.
- *Virtual method*. In Java, all methods are virtual, all be overridden (replaced). C++ requires the keyword `virtual` to decorate a method to mark it as virtual, therefore, one that can be overridden.

Entity Class – Code Tour

```
class Entity
{
  public:
    Entity(double facing, double posX, double posY);

    virtual void update(double elapsedTime);
    void report();

    std::uint32_t getId() { return m_id; }
    double getFacing() { return m_facing; }
    double getSpeed() { return m_speed; }
    double getPosX() { return m_posX; }
    double getPosY() { return m_posY; }

  protected:
    double m_facing;
    double m_speed;
    double m_posX;
    double m_posY;

    virtual std::string getType() = 0;
```

```
        virtual void reportUnique() = 0;
        void move(double elapsedTime);

      private:
        static std::uint32_t nextId;
        std::uint32_t m_id;
};
```

A few things to note about the class declaration. Notice the use of the `virtual` keyword on the `update` method. Notice the `virtual` and `= 0;` syntax on the `getType` and `reportUnique` methods; these are pure virtual methods. Finally, notice the static keyword on the `nextId` variable; this is `static` variable, visible only to the `Entity` class.

Static variables declared as part of a class require an "implementation" in the `.cpp` file. The `nextId` definition is:

```
std::uint32_t Entity::nextId = 0;
```

The `Entity` class constructor is:

```
Entity::Entity(double facing, double posX, double posY) :
    m_id(nextId++),
    m_speed(0),
    m_facing(facing),
    m_posX(posX),
    m_posY(posY)
{
}
```

The only item of note here is the use of the static `nextId` variable to assign a unique number to each `Entity` instance upon creation.

The `Entity::update` method is:

```
void Entity::update(double elapsedTime)
{
    move(elapsedTime);
}
```

Nothing remarkable about the implementation, but remember, this is marked as `virtual` in the declaration, meaning it can be overridden by a derived class.

The `Entity::move` method is:

```
void Entity::move(double elapsedTime)
{
    auto vectorX = std::cos(m_facing);
    auto vectorY = std::sin(m_facing);

    m_posX += (vectorX * elapsedTime * m_speed);
    m_posY += (vectorY * elapsedTime * m_speed);
}
```

Again, nothing remarkable, updates the position of the `Entity` based upon a few object properties and the `elapsedTime`.

The `Entity::report` method is:

```
void Entity::report()
{
    std::cout << "--- " << getType() << " Report ---" << std::endl;
    std::cout << "id: " << m_id << std::endl;
    std::cout << "Position: (" << m_posX << ", " << m_posY << ")" << std::endl;
    std::cout << "Facing: " << m_facing << std::endl;
    std::cout << "Speed: " << m_speed << std::endl;
```

```
        reportUnique();

        std::cout << std::endl;
    }
```

The interesting part of the implementation is the call to the two pure virtual methods `getType` and `reportUnique` methods; run-time polymorphism.

Inheritance

Before getting into the syntax of class inheritance, it is useful to discuss the `public`, `protected`, and `private` visibility modifiers in C++. These modifiers have somewhat different meanings in C++ than Java, it is important to understand them correctly in each language. C++ doesn't have a similar concept to Java packages, which are part of the equation for Java class visibility modifiers. The `public` and `private` modifiers are essentially the same, with `protected` having the biggest change in meaning, also C++ does not have Java's default visibility; if not specified, class visibility is public.

- `public` – All code within and using the class has visibility.
- `protected` – All code within the class and derived classes have visibility, but code using an instance of the class does not. In Java, protected is visible to all code in the class, derived classes, and code that uses instances of the class. Code outside the package does not have visibility.
- `private` – Only code within the class has visibility. Derived classes and code using an instance of the class do not have visibility.

The following table shows this same information, maybe more understandable to some.

Access Location	**`public`**	**`protected`**	**`private`**
Within the class	yes	yes	yes
Derived class	yes	yes	no
Use of object	yes	no	no

The general syntax for declaring a derived class from another class is:

```
class <derived> : <visibility modifier> <parent>
{
        ... rest of class declaration ...
};
```

Where `<derived>` is the name of the derived class and `<parent>` is the name of the already declared class. The declaration for the `Vehicle` class begins as:

```
class Vehicle : public Entity
{
        ... rest of class declaration ...
};
```

Notice the use of a visibility modifier `public` before the name of the parent class. This visibility modifier may be `public`, `protected`, or `private`. The next table describes what happens to members of the parent class based upon the visibility modifier used.

Inherited/Existing	`public`	`protected`	`private`
`public`	public	protected	private
`protected`	protected	protected	private
`private`	not visible	not visible	not visible

The header row is the visibility indicated when inheriting. The left column is the existing visibility of the member. For example, if a parent class has a `public` member and a derived class inherits with `protected` visibility, the member from the parent class now has `protected` visibility in the derived class. By *not visible*, I mean those members are not visible in the derived class; they exist, but are not accessible.

Most of the time `pubic` is used when inheriting from another class, but once in a while you might come across derived classes that use `protected` or `private`.

Inheritance – Constructors

Invoking Constructors

Java has the concept that at every level in an inheritance hierarchy a constructor is invoked. Which constructor is used at each level in the hierarchy can be specified through the use of constructor chaining. C++ has the same concept, but rather than the term constructor chaining, the term constructor delegation is used.

Inheriting Constructors

In Java, when deriving a sub-class from a super class, constructors are not inherited. The idea being that constructors are not class members and only members are inherited. By default, C++ does not inherit constructors either, but it is possible to inherit constructors.

The `Monster` class demonstrates how to inherit constructors with the `using Entity::Entity;` line in the class declaration.

Polymorphism

C++ supports the kinds of polymorphism you are used to in Java.

Virtual Methods

As noted earlier, in Java, all class methods can be polymorphically overridden, this is not true in C++. In C++, in order for a method to be polymorphically overridden, it must be marked as `virtual` in the parent class. Then, in the derived class it can be overridden; the keyword `virtual` in the derived class is optional, but recommended.

Java provides the `@Override` annotation to mark a method as overriding a super class method. The annotation is not required, but is recommended because the compiler will verify there is a method being overridden from a super class. C++ has a similar, optional (but recommended), annotation. The keyword `override` may be placed in the method declaration to have the compiler verify the method exists in a parent class.

Non-Virtual Methods (don’t do this)

C++ allows a derived class to override a non-virtual method from a parent class, but the behavior of the code is much different from virtual methods, and therefore, requires a separate example. Consider the following code:

***** need to still make this example *****

Abstract Methods & Classes

An abstract method in C++ is called a *pure virtual method*. They syntax for a pure virtual method looks like:

```
virtual void myMethod() = 0;
```

The `= 0;` syntax following the method declaration indicates this method has no implementation, a pure virtual method in C++ terminology.

Any class with one or more pure virtual methods is an abstract class and can not be instantiated.

Derived classes may or may not implement any or all pure virtual methods from their parent class. If any pure virtual method is not implemented, it remains as a pure virtual method in the derived class, and therefore, the derived class is also abstract.

Vehicle Class – Code Tour

```
class Vehicle : public Entity
{
  public:
    enum class Color
    {
        Red,
        Blue,
        Silver,
        White,
    };

    Vehicle(Color color, double facing, double posX, double posY);

    virtual void update(double elapsedTime) override;

  protected:
    virtual std::string getType() override { return "Vehicle"; }
    virtual void reportUnique() override;

  private:
    Color m_color = Color::Silver;
};
```

The `Vehicle` class is derived from the `Entity` class. Interestingly, an `enum Color` is nested within the class, ensuring the identifier `Color` doesn't collide with any other definition. Also note the implementation of the pure virtual method `getType` as part of the class declaration.

```
Vehicle::Vehicle(Color color, double facing, double posX, double posY) :
    m_color(color),
    Entity(facing, posX, posY)
{
}
```

The `Vehicle` class defines a new constructor that accepts a `Color`, in addition to the other parameters accepted by the `Entity` class. This constructor uses member initialization for the color, while delegating the remaining parameters to the `Entity` constructor.

The `Entity` class defined an update method as virtual, providing an implementation. The `Monster` class overrides the `update` method, providing a new implementation, but also invoking the `Entity` class's `update` as part of its implementation.

```
void Vehicle::update(double elapsedTime)
{
    if (m_speed == 0)
```

```
        {
            m_speed = 0.75;
        }

        Entity::update(elapsedTime);
    }
```

The `Entity` class defines a non-virtual `report` method that makes a call to a pure virtual method (on the `Entity` class) named `reportUnique`. The `Vehicle` class provides an implementation for the `reportUnique` method.

```
    void Vehicle::reportUnique()
    {
        std::cout << "Color: ";
        switch (m_color)
        {
            case Color::Red:
                std::cout << "Red";
                break;
            case Color::Blue:
                std::cout << "Blue";
                break;
            case Color::Silver:
                std::cout << "Silver";
                break;
            case Color::White:
                std::cout << "White";
                break;
        }

        std::cout << std::endl;
    }
```

Monster Class – Code Tour

```
    class Monster : public Entity
    {
      public:
        using Entity::Entity;
        Monster(std::string name, double facing, double posX, double posY);

        virtual void update(double elapsedTime) override;

        std::string getName() { return m_name; }

      protected:
        virtual std::string getType() override { return "Monster"; }
        virtual void reportUnique() override;

      private:
        std::string m_name = "anonymous";
    };
```

The `Monster` class is derived from the `Entity` class. Note the `Monster` class is inheriting the constructor from the `Entity` class, with the using `Entity::Entity;` statement. If there were more than one constructor in the `Entity` class, all of them would be inherited, there is no way to pick and choose which to inherit. This class also provides an implementation for the `getType` pure virtual method from the `Entity` class.

Notice that `m_name` is initialized to the string `"anonymous"` in the declaration. The reason for this is because the `Entity` constructor is inherited and may be used to create an instance of the `Monster` class. When that is done, it is still desired for the instance to have a name, therefore the member initialization.

The remainder of the `Monster` class implementation is unremarkable, being the substantially similar to the `Vehicle` class, only specialized for a `Monster`.

Entity, Vehicle, and Monster Usage

Let's take a look at using this code:

```
int main(int argc, char* argv[])
{
    std::vector<std::shared_ptr<Entity>> entities;

    entities.push_back(std::make_shared<Monster>("Mean Green", 0, 0.5, 0.5));
    entities.push_back(std::make_shared<Monster>(3.14159 * 2, 0.0, 0.0));
    entities.push_back(std::make_shared<Vehicle>(Vehicle::Color::Blue, 3.14159, 0.25, 0.25));

    std::cout << "******** Before Update ********" << std::endl << std::endl;
    for (auto&& entity : entities)
    {

        entity->report();
        entity->update(0.10);
    }

    std::cout << "******** After Update ********" << std::endl << std::endl;
    for (auto&& entity : entities)
    {
        entity->report();
    }

    return 0;
}
```

This code demonstrates the typical features of class-based run-time polymorphism you expect. The `std::vector` is a vector of `Entity` `std::shared_ptr`'s. `Entity` is an abstract class, but is used to track `Monster` and `Vehicle` instances.

The first `Monster` instance uses the derived class constructor. The second `Monster` instance uses the constructor inherited from the `Entity` class. The `Vehicle` instance uses its derived constructor, also note the syntax used to access the `Color enum`.

The first loop demonstrates run-time polymorphism in calling the `update` method. Internally, the `report` method utilizes run-time polymorphism, but that code is contained within the `Entity` class itself.

Inheritance – Destructors

It isn't obvious, but destructors follow the same rules as methods when it comes to inheritance. If a class is intended to be used polymorphically, the destructors in the class hierarchy should also be declared as `virtual`. If a polymorphic class fails to declare its destructor as `virtual`, when used through a base class pointer the object is deleted, the derived class destructor(s) will not be called. In order to ensure all destructors in a class hierarchy are invoked when an object is destroyed, the destructors must be marked as `virtual`.

Consider the following code:

```
namespace nonvirtual
{
    class Base
    {
      public:
        Base()
        {
            std::cout << "Base constructor" << std::endl;
        }
        ~Base()
        {
            std::cout << "Base destructor" << std::endl;
        }
    };

    class Derived : public Base
    {
      public:
        Derived()
        {
            std::cout << "Derived constructor" << std::endl;
        }
        ~Derived()
        {
            std::cout << "Derived destructor" << std::endl;
        }
    };
}


std::cout << "--- Non-Virtual Destructor Demonstration ---" << std::endl;
std::cout << "* non-polymorphic - good *" << std::endl;
nonvirtual::Derived* good = new nonvirtual::Derived();
delete good;

std::cout << std::endl;
std::cout << "* polymorphic - bad *" << std::endl;
nonvirtual::Base* bad = new nonvirtual::Derived();
delete bad;
```

Notice the destructors are not declared as `virtual`. The output from the code that uses these classes is:

```
--- Non-Virtual Destructor Demonstration ---
* non-polymorphic - good *
Base constructor
Derived constructor
Derived destructor
Base destructor

* polymorphic - bad *
Base constructor
Derived constructor
Base destructor
```

In the first case, where the class is used non-polymorphically, both destructors are invoked when the object is destroyed. In the second case, where the class is used polymorphically, only the base class destructor is invoked. If the derived class destructor releases resources acquired during its lifetime, it wouldn't be called, and therefore, those resources leak.

This problem is easily fixed by declaring the destructors as `virtual`, ensuring they work just as `virtual` methods. Consider the following code:

```
namespace isvirtual
{
    class Base
    {
      public:
        Base()
        {
            std::cout << "Base constructor" << std::endl;
        }
        virtual ~Base()
        {
            std::cout << "Base destructor" << std::endl;
        }
    };
```

```
        class Derived : public Base
        {
          public:
            Derived()
            {
                std::cout << "Derived constructor" << std::endl;
            }
            virtual ~Derived()
            {
                std::cout << "Derived destructor" << std::endl;
            }
        };
    }

    std::cout << "--- Virtual Destructor Demonstration ---" << std::endl;
    std::cout << "* non-polymorphic - good *" << std::endl;
    isvirtual::Derived* good = new isvirtual::Derived();
    delete good;

    std::cout << std::endl;
    std::cout << "* polymorphic - still good *" << std::endl;
    isvirtual::Base* stillGood = new isvirtual::Derived();
    delete stillGood;
```

The output from this code is:

```
--- Virtual Destructor Demonstration ---
* non-polymorphic - good *
Base constructor
Derived constructor
Derived destructor
Base destructor

* polymorphic - still good *
Base constructor
Derived constructor
Derived destructor
Base destructor
```

In both cases the destructors from both levels in the class hierarchy are invoked, regardless if it is being used polymorphically or not.

Deleting Methods, Default Methods, & final Specifier

If not defined, the C++ compiler writes default implementations for many constructors and methods on a class. For example a default copy constructor is written if one is not provided, similarly with the assignment operator. There may be a case where it is desired for an object to not be copied; think about `std::unique_ptr`. One way to do this is to declare the various constructors and operators that can make copies as protected (or private), which prevents code other than the class itself from making copies. The downside to this approach is that the intent isn't obvious, unless other programmers understand the idiom being used.

Beginning with C++ 11, there is an explicit approach to solving this problem, deleted methods (or functions). C++ allows a programmer to explicitly say a method should not be implemented, deleted in C++ terminology. Let's demonstrate this by creating a class that can't be copied by deleting the various operations that can make a copy.

```
class CantCopyMe
{
  public:
    CantCopyMe() = default;
    CantCopyMe(int value) :
        data(value) {}
    CantCopyMe(const CantCopyMe&) = delete;
    CantCopyMe& operator=(const CantCopyMe&) = delete;

  private:
    int data = 0;
};
```

Notice the first two constructors. Normally, when any overloaded constructor is written, the compiler won't write a default constructor. This class has an overloaded constructor (the second one) that normally prevents the default constructor from being automatically provided. If it is desired for the constructor to provide the default constructor (or any other compiler provided method) the constructor (or method) can be specified with the `= default;` syntax following. When this is done, the compiler will provide the specified constructor (or method).

The next two items in the class are the copy constructor and copy assignment operator. Because these create copies of the object, and we don't want that, the `= delete;` syntax follows them, telling the compiler to not write those operations!

With the class written this way, the compiler will catch any attempts to make copies of objects of that type.

```
CantCopyMe a;
CantCopyMe b;
CantCopyMe c = a;
b = a;
```

When this code is compiled, the compiler reports an error on line three, because the copy constructor is deleted. An error on line four is also reported, because the assignment operator is deleted.

What we haven't done as part of this class, but that we could/should, is to write the move constructor and move assignment operator, because we might want to allow move operations (like `std::unique_ptr`), while not allowing copy operations.

`final` Specifier

A clever developer might realize they can derive from `CantCopyMe` and then provide copy constructor and copy assignment implementations, thereby, allowing copies of the object to be made. A way to avoid this (can't prevent, a malicious developer can make a mess of anything) is to declare the `CantCopyMe` class as `final`, which prevents any inheritance from it.

```
class CantCopyMe final
```

Now, any attempt to derive from this class results in a compiler error.

In addition to declaring classes as `final`, virtual methods may also be declared as `final`.