Classes

Encapsulation is a concept that is used to protect member variables and ensure that the object is always in a certain state.

Inheritance With Access Specifiers

When creating classes in C++ we should use the **access specifier** below that best fits our scenario. By default, when defining a `class`, all members are `private` unless otherwise specified. In contrast, when we define a `struct`, all members are `public` unless otherwise specified. This is the only difference between a `class` and a `struct` in C++, and all other concepts are interchangable between the two. Both can have member functions, variables, friends, destructors, constructors, etc.

```
class Test {
  int w;  // W is private in this context
  public:
   int x;  // X is accessible from outside or inside of the class
  protected:
   int y;  // Y is accessible from within the class, by members of the same class, and any derived classes
  private:
   int z;  // Z is only accessible from within the class or by members of the same class
};
```

```
struct Test {
  int w;  // W is public in this context
  public:
   int x;  // X is accessible from outside or inside of the class
  protected:
   int y;  // Y is accessible from within the class, by members of the same class, and any derived classes
  private:
   int z;  // Z is only accessible from within the class or by members of the same class
};
```

When using **inheritance access specifiers** we should pay attention to how member access is impacted. Consider the code below, where we notice a change in the way we access `public` and `protected` members of the base class, `A`. In main, we try to access some of these members and show which ones we can and cannot access. In each inheriting class, we define new public members that access private or protected members to show that we have the ability to do so

through inheritance, but only within the scope of the class or member definitions.

```
#include <iostream>

class A {
  // Private is the default access modifier within classes in c++
  int private_y; // This member will never be accessible from any derived class

public:
  A() {};
  // Because the destructor was declared virtual, every deriving class will call this destructor in sequence when
being destroyed
  // So for class PublicA;  ~PublicA() -> ~A() is the order destructors will be called when leaving scope...
  //   or deleteing the object on the heap
  virtual ~A() { std::cout << "Deleting A\n";};

  int pub_x;

protected:
  int protected_x;

private:
  int private_x; // This member will never be accessible from any derived class
};

class PublicA : public A {
public:
  ~PublicA() {std::cout << "Deleting PublicA\n";};

  int a = pub_x; // In this context, pub_x is protected
  int b = protected_x; // In this context, protected_x is protected
};

class ProtectedA : protected A {
public:
  ~ProtectedA() {std::cout << "Deleting ProtectedA\n";};

  int a = pub_x; // In this context, pub_x is protected
  int b = protected_x; // In this context, protected_x is protected
};
```

```
class PrivateA : private A {
public:
  ~PrivateA() {std::cout << "Deleting PrivateA\n";};

  int a = pub_x; // In this context, pub_x is private
  int b = protected_x; // In this context, protected_x is private
};



int main (int const argc, char const * argv[])  {
  // Test destructor for base class A
  A * baseA = new A; // Allocate A on the heap
  delete baseA; // Free (delete) A from the heap

  PublicA publicA;
  publicA.pub_x = 5; // Valid, since pub_x, a, and b are all public
//  publicA.protected_x = 5; // Error! protected_x is protected in this context

  ProtectedA protectedA;
//  protectedA.pub_x = 5; // Error! pub_x is protected in this context
//  protectedA.protected_x = 5; // Error! protected_x is protected in this context

  PrivateA privateA;
//  privateA.pub_x = 5; // Error! pub_x is private in this context
//  privateA.protected_x = 5; // Error! protected_x is private in this context



  // Destructor called for each object on the stack at exit
  // Note: Since these are on the stack, they will be destroyed in reverse order
}
```

In summary, this chart helps to describe the various combinations and results between base class access identifiers and their derived class's inheritance access specifier.

When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

`const` member functions that return bool can be referred to as **predicates**

```
class Test {
  public:
bool empty() const;// empty() is a predicate in this context
};
```

Multiple-Inheritance is when a class inherits from more than one parent object or class.

Abstract Classes

An **abstract class** is one that cannot be instantiated without first being inherited from. This means that by itself an abstract class can only be used as a **base class** for further implementation. An abstract class *may not* be multiple-inherited, but they *may* contain state values (member variables), and/or implementation (methods). Abstract classes can be inherited without implementing the abastract methods, though such a derived class is abstract itself.

An example of an abstract classs `Animal` that uses its own constructor. Notice that derived classes `Human` and `Dog` each have specific constructors with respect to their parent class, `Animal`. The use of the `virtual` keyword in defining **pure virtual functions** makes this an **abstract class**, where the implementation of `speak` is definined for each derived class (`Human`, and `Dog`, in this case) -

```
#include <iostream>


class Animal {
public:
  // A constructor the AbstractClass depends on in order to be instantiated
  Animal(std::string name_) : name(name_) {};
```

```
  // Implementation the AbstractClass takes with when derived from
  void showName() {
    std::cout << "Name: " << name << std::endl;
  }

  // A pure virtual function, required to be implemented by deriving classes
  virtual void speak() = 0; //...() = 0; required to be considered pure virtual
  // The declaration above makes this class abstract; We need to define speak() before instantiating
  // The class can be instatiated by deriving from this base class, and instantiating the derived class

protected:
  // Protected member allows deriving classes to inherit while acting as private
  std::string name;
};

class Human : public Animal {
public:
  // A constructor for the Human class with respect to it's base class (Animal)
  Human(std::string name_) : Animal(name_) {};

  // speak() is defined for all Humans
  void speak() {
    std::cout << name << ": Hello!\n";
  }
};

class Dog : public Animal {
public:
  // A constructor for the `Dog` class with an additional unique parameter
  // Must still respect it's base class constructor parameter, and pass to Animal's ctor
  Dog(std::string name_, std::string sound_="Bark!")
  : Animal(name_), sound(sound_) {};

  // speak() is defined for all Dogs
  void speak() {
    std::cout << name << ": " << sound << std::endl;
  }

private:
  // Note: The `sound` value could not be inherited further
```

```
  std::string sound;
};

int main (int const argc, char const * argv[])  {
  Human h("Shaun");
  h.speak();

  Dog d("Spot", "Bark!");
  d.speak();

  Dog f("Fluffy", "Yap!");
  f.speak();
}
```

Abstract class with no pure virtual member functions -

```
#include <iostream>

/*****************************************************************************/
// KeyData struct to hold data related to key ciphers
struct KeyData {
  explicit KeyData(std::string key) : keyWord_(std::move(key)) {};
  explicit KeyData() : keyWord_(GetKey()) {};
  // Pure virtual dtor to make abstract class despite no pure virtual members
  // + This works, because a base class should have a virtual dtor anyway
  virtual ~KeyData() = 0;

  // Allows getting keyWord when using KeyData default ctor
  static std::string GetKey()
  {
    std::string key;
    std::cout << "Enter the keyword: ";
    std::getline(std::cin, key);
    std::cout << "Keyword entered: \"" << key << "\"\n";
    return key;
  }

protected:
  std::string keyWord_;
};
```

```
// Definition of pure virtual dtor (required)
KeyData::~KeyData() {}
```

Interfaces

An **interface** has no implementation, and contains only a virtual destructor and pure virtual functions. `virtual` destructors ensure that when an interface is destroyed, the correct destructors will be called down the inheritance hierarchy. Interfaces have no state or implementation, they may be multiple-inherited.

```
// TODO: Example of an interface
```

Constructors and Resource Management

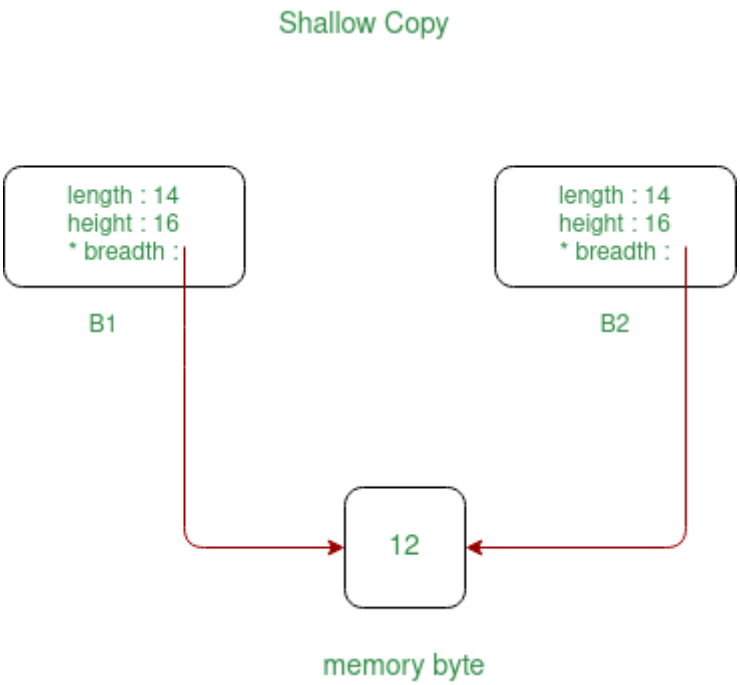
When dealing with **dynamic memory allocation** both operators `new` or `delete` are aware of constructors and destructors. In contrast, `malloc` and `free` are *not* aware of class constructors or destructors.

A **deep copy** is when we create or allocate new memory addresses *and* assign the values at these addresses to match that of an existing object. See the example below, where `A(A & rhs)` implements a deep-copy of the `rhs` object. In the examples below, we also take advantage of **C++11's range-based-for** by implementing `begin()` and `end` members that return an iterator (`int *`) to the beginning and end of the dynamic array.

```
#include <iostream>

class A {
  // Private is the default access modifier within classes in c++
  int private_y; // This member will never be accessible from any derived class

public:
  // ctor
  A(int size, int value)
  : size_(size), intArray_(new int[size]) {
    for (auto &e : *this) e = value;
  };

  // Copy ctor implementing a deep-copy of rhs
  A(A & rhs) :size_(rhs.size_), intArray_(new int[size_]) {
    std::copy(&rhs.intArray_[0], &rhs.intArray_[size_], intArray_);
  }
```

```
  // dtor of base class with dynamic member is virtual to handle destruction
  ~A() {
    std::cout << "Deleting A\n";
    delete [] intArray_;
  }

  // Assignment operator utilizes copy ctor to create local copy
  A & operator=(A rhs) {
    std::swap(intArray_, rhs.intArray_);
    std::swap(size_, rhs.size_);
  }

  // Implementing begin and end for use with objects derived from this class
  int * begin() { return &intArray_[0];};
  int * end() { return &intArray_[size_];};

  void print() {
    for (auto e : *this) {
      std::cout << e << std::endl;
    }
    std::cout << std::endl;
  }

protected:
  int size_;
  int * intArray_; // Dynamic memory requires defintition of ctor, dtor, and op=
};

int main (int const argc, char const * argv[])  {
  // Test destructor for base class A
  A * baseA = new A(5, 1);
  baseA->print();
  delete baseA;
}
```

A **shallow copy** is when we create a copy of an object using references to the original location of the data, as in the example below




```
#include <iostream>

class A {
  // Private is the default access modifier within classes in c++
  int private_y; // This member will never be accessible from any derived class

public:
  // ctor
  A(int size, int value)
  : size_(size), intArray_(new int[size]) {
    for (auto &e : *this) e = value;
  };

  // Copy ctor implementing a shallow-copy of rhs
  A(A & rhs) :size_{rhs.size_}, intArray_{rhs.intArray_} {};

  // Warning: If this object is ever used to initialize another, we will face an error on destruction
  //   Because we created a shallow-copy, we cannot delete the dynamic allocation without corrupting another
object
  ~A() {
    std::cout << "Deleting A\n"; //
    delete [] intArray_; // Error! Double free detected; Since we created a shallow copy in dynamic memory
  }
```

```
  // Assignment operator utilizes copy ctor to create local copy
  A & operator=(A rhs) {
    std::swap(intArray_, rhs.intArray_);
    std::swap(size_, rhs.size_);
  }

  // Implementing begin and end for use with objects derived from this class
  int *begin() { return &intArray_[0];};
  int *end() { return &intArray_[size_];};

  void set(int value) {
    for (int &e : *this) e = value;
  }

  void print() {
    for (auto e : *this) {
      std::cout << e << std::endl;
    }
    std::cout << std::endl;
  }

protected:
  int size_;
  int * intArray_; // Dynamic memory requires defintition of ctor, dtor, and op=
};

int main (int const argc, char const * argv[])  {
  A * baseA = new A(5, 1);
  baseA->print();

  A b(*baseA); //
  b.print();
  b.set(5); // Since the class uses a shallow copy, the changes are refelcted within both objects

  // baseA and b both point to the same location in memory for intArray_
  baseA->print();
  b.print();

  A c(b);
  c.print();
```

```
  c.set(10); // Since the class uses a shallow copy, the changes are refelcted within all objects

  baseA->print();
  b.print();
  c.print();

  delete baseA;
}
```

A **conversion constructor** is a constructor with a single parameter that converts the argument at invocation to the type of the object. A constructor with multiple parameters is considered implicit if all but one parameter provide default values. For example, consider the code below

Implicit conversion ctors can cause trouble w/ function overloading through unintended type conversions. Below, we see examples of applying `explicit` to our constructors in class `B`, and compare the results to a similar class `A` that did not. See the main function for the final test and comparisons between the classes, then check the definitions to see why.

```
#include <iostream>

class A {
public:
  // No explicit declaration allows implicit usage, converting to class type
  A() {};
  A(int val) : x(val), y('~') {};
  A(char character, int value = 5) : y(character), x(value) {};
  ~A() {};

  void show() {
    std::cout << "x: " << x << std::endl;
    std::cout << "y: " << y << "\n\n";
  }

  int x;
  char y;
};

class B {
public:
  // Declaring constructors explicit forces more strict usage
```

```
  explicit B() {};
  explicit B(int val) : x(val), y('~') {};
  explicit B(char character, int value = 5) : y(character), x(value) {};
  ~B() {};

  void show() {
    std::cout << "x: " << x << std::endl;
    std::cout << "y: " << y << "\n\n";
  }

  int x;
  char y;
};

int main (int const argc, char const * argv[])  {
  // All of the below is valid for class A, since we did not declare destructors explicit
  A a = 10;
  A a1(10);
  A a2 = {10};
  A a3 = 'c';
  A a4('c');
  A a5{'c'};
  A a6('c', 10);
  A a7{'c', 10};
  A a8 = {'c', 10};
  A a9 = (A)10;
  A a10 = (A)'x';
  A a11 = A('x');
  A a12 = A('x', 10);
  A a13 = A{'x'};
  A a14 = A{'x', 10};

  // For B, since we declared constructor explicit...
//B b = 5; // Error! Implicit type conversion not allowed with explict ctor
  B b1(5);
//B b2 = {5}; // Error! Implicit type conversion not allowed with explict ctor
//B b3 = 'x'; // Error! Implicit type conversion not allowed with explict ctor
  B b4('x');
//B b5 = {'x'}; // Error! Implicit type conversion not allowed with explict ctor
  B b6('x', 5);
```

```
  B b7{'x', 5};
//B b8 = {'x', 5}; // Error! Implicit type conversion not allowed with explict ctor
  B b9 = (B)5;
  B b10 = (B)'x';
  B b11 = B('x');
  B b12 = B('x', 5);
  B b13 = B{'x'};
  B b14 = B{'x', 5};
}
```

Declaring static const member variables -

```
// SomeClass.hpp
class SomeClass {
public:
  CarFactory(std::string name, int number):
  someName(location), someNumber(number) {}
  ~Class(){}
private:
  std::string someName;
  int someNumber;
  const std::array<std::string, 4> someArray;
};
```

and in a seperate, **source file** -

```
// SomeClass.cpp


const std::array<std::string, 4>
    SomeClass::someArray({"Thing1", "Thing2", "Thing3", "Thing4"});
```

If you want a member variable that is an iterator of this array -

```
class SomeClass {
public:
  CarFactory(std::string name, int number):
  someName(location), someNumber(number) {}
  ~Class(){}
private:
  std::string someName;
  int someNumber;
```

```
  const std::array<std::string, 4> someArray;
  decltype(someArray)::iterator arrayIter = someArray.begin();
};
```

Operator Oveloading

Overloading the `ostream` operator `<<` to allow printing an object directly to stdout for `Person` objects.

```
std::ostream & operator<<(std::ostream & o, const Person & a) {
  o << "first_name=" << a.get_first_name() << ",last_name=" << a.get_last_name();
  return o;
}
```

Revision #14
Created 6 February 2021 02:42:53 by Shaun Reed
Updated 8 January 2022 00:51:33 by Shaun Reed