UPE Tutoring:

CS 31 Final Review

Sign-in http://bit.ly/33LgeIj

Slides link available upon sign-in

Classes

- A **class** is like a struct, but with **member functions** as well as member variables. Classes are at the core of Object-oriented Programming (OOP).

```
class Person {
  int age;
  string name;          // Strings are actually objects from the string class!
  double money;
  void doubleMoney();   // Member function, declared but not yet implemented.
};
```

- We call an instance of a class an **object**. In this way, OOP involves different types of objects interacting with each other.

Classes – Member Functions

Now, let's fill in the doubleMoney function of the Person class from the previous slide. There are two ways to do this:

```
// 1. Inside the class definition.
class Person {
  int age;
  string name;
  double money;
  void doubleMoney() { money *= 2; }
};

// 2. Outside of the class definition.
Person::doubleMoney() {
  money *= 2;
}
```

The **scope resolution operator** (e.g. `Person::`) tells the compiler that we are defining the doubleMoney function of the Person class.

Note: we don't need the dot operator to refer to Person's money variable when we are in one of Person's member functions!

Classes – Access Specifiers

We should now have a working `Person` class! Let's try working with some objects. We refer to their member functions and variables with the dot operator.

```
int main() {
  Person bobby;
  bobby.age = 5;
  bobby.name = "Bobby";
  bobby.money = 3.49;
  bobby.doubleMoney();
}
```

Classes – Access Specifiers (cont.)

We should now have a working `Person` class! Let's try working with some objects. We refer to their member functions and variables with the dot operator.

```
int main() {
  Person bobby;
  bobby.age = 5;   //ERROR!
  bobby.name = "Bobby";   //ERROR!
  bobby.money = 3.49;   //ERROR!
  bobby.doubleMoney();   //ERROR!
}
```

Classes – Access Specifiers (cont.)

We should now have a working `Person` class! Let's try working with some objects. We refer to their member functions and variables with the dot operator.

```
int main() {
  Person bobby;
  bobby.age = 5;   // Good to go :^)
  bobby.name = "Bobby";
  bobby.money = 3.49;
  bobby.doubleMoney();
}
```

The compiler doesn't let us access any of bobby's members! This is because class members have the **private** access specifier by default and cannot be accessed by an outside class or function. To fix this, we adjust our class:

```
class Person {
public:
  int age;
  string name;
  double money;
  void doubleMoney() { money *= 2; }
};
```

Classes – Access Specifiers (cont.)

- The reason we were able to access the members of a struct earlier is because they are **public** by default.
- One big problem with our `Person` class so far is that if we make its member variables `age`, `name`, and `money` private, we have no way to change them. If we make them public, they can be set to an invalid state by any code that instantiates a `Person` object!

```
int main() {
  Person p;
  p.age = -5;   // This doesn't make sense (unless we're Benjamin Button).
}
```

Classes – Encapsulation

To fix this, we make Person's member variables private and add public **accessor** (getter) and **mutator** (setter) functions that set rules on how to access and change them. This is called **encapsulation**!

```
class Person {
public:
  void setAge(int yrs);
  int getAge();
  void setName(string nm);
  string getName();
  void doubleMoney();
  double getMoney();
private:  // Same member variables!
};
```

```
void Person::setAge(int yrs) {
  if (yrs >= 0)
    age = yrs;
}
int Person::getAge() { return age; }

void Person::setName(string nm) {
  if (nm.length > 0)
    name = nm;
}
string Person::getName() { return name; }

void Person::doubleMoney() { money *= 2; }
double Person::getMoney() { return money; }
```

Classes – Encapsulation (cont.)

- Generally, we want to make our member variables private. Therefore, we make them accessible through public member functions that access or mutate them in ways that are reasonable towards our implementation.
  - This keeps objects in a valid state and hides the “nitty gritty” details of our implementation from anyone who wants to use our class
- Now, you just have to call `doubleMoney` on a `Person` and you know that their money will be doubled.

Encapsulation Example

```
int main() {
  string name;
  cout << "What is my name? " << endl;
  getline(cin, name);

  Person p;
  p.setName(name);
  p.setName("");
  cout << "I Am " <<
      p.getName() << "\n";
  p.setAge(49);
  p.setAge(-1);
  cout << "I am " << p.getAge() <<
      " years old.\n";
}
```

```
> What is my name? the Walrus
> I Am the Walrus
> I am 49 years old.
```

Classes vs. Structs

- Technically, the only difference between a class and a struct is the default access specifier.
- By convention, we use structs to represent simple collections of data and classes to represent complex objects.
    - This comes from C, which has only structs and no member functions.

Constructors

- There is yet another problem with our `Person` class: initializing its members one-by-one is annoying but if we don't do it, our `Person` starts in an invalid state!
- A **constructor** is a member function that has the same name as the class, no return type, and automatically performs initialization when we declare an object:

```
class Person {
public:
  Person();
  // Same stuff as before!
};
```

Constructors (cont.)

- Constructors can be defined inside or outside of a class, just like normal functions.
- Like normal functions, constructors can (and usually are) overloaded with different numbers and types of parameters to suit different purposes.
- Unlike normal functions, they <u>cannot</u> be called with the dot operator.
- A **default constructor** is one with no arguments; the compiler generates an empty one by default.
- The "default default" constructor leaves primitive member variables (int, double, etc.) uninitialized and calls the default constructors of class members
    - Example: any string members will be created with the default string constructor

Constructors – Basic Syntax

Let's add constructors to our Person class!

```
class Person {
public:
  Person();                                   // 1
  Person(int yrs, string nm, double cash);   // 2
  // Same stuff as before!
};

Person::Person() {
  age = 0;
  name = "";
  money = 0.0;
}
```

```
Person::Person(int yrs, string nm, double cash) {
  setAge(yrs);
  setName(nm);
  money = cash;
}

int main() {
  Person p1;   // Constructor 1 is called.
  // Constructor 2 is called.
  Person p2(44, "Elon Musk", 13000000000.0);
  p1 = Person(19, "Freshman at UCLA", -100000.0);
  Person p3(); /* Constructor 1 NOT called:
                  The compiler thinks we're
                  defining a function!   */
  p1.Person(); // Illegal!
}
```

Constructors – Initializer Lists

An **initializer list** is an alternate, concise syntax for constructors.

```
Person::Person()
    : age(0), name(""), money(0.0) {}

Person::Person(int yrs, string nm, double cash)
    :age(yrs), name(nm), money(cash) {}
```

- Initializer lists are required to initialize const and reference type member variables
- They are also preferred for class member variables (like name); otherwise, the default constructor for that class is called.
- Because of this, they are required for classes with no default constructor. (We'll elaborate more on this later.)

Constructors – Pitfalls

- We may still need to check for invalid parameters in a constructor, which initializer lists can't do unless we call functions within them (assume `stuffCount` is a member variable).

```
SomeClass::SomeClass(int stuff) : stuffCount(checkStuff(stuff)) {}
```

- If we declare a constructor, the compiler will not longer generate a default constructor!

```
int main() {
  Person p1;  // Illegal if no defined default constructor!
  Person p2(5, "Squam", 3.51);
}
```

Constructors – Order of Construction

- When we instantiate an object, we begin by initializing its member variables *then* by calling its constructor. (Destruction happens the other way round!)
- The member variables are initialized by first consulting the initializer list. Otherwise, we use the default constructor for the member variable as a fallback.
- For this reason, member variable without a default constructor <u>must</u> be initialized through the initializer list.

Constructors – Order of Construction (cont.)

Suppose each Person now has a Pet.

```
class Pet {
public:
  Pet(string nm) { ... }
};

class Person {
public:
  Person();
private:
  Pet fluffy;
};
```

```
Person::Person()
    /* To properly instantiate the
       Pet, we pass it a name through
       the initializer list. */
    : fluffy("Steve") {
  // Other initialization stuff ...
}
```

Practice Question: Construction

What is the output of the following code snippet?

```
class Cat {
public:
  Cat(string name) {
    cout << "I am a cat: " << name << endl;
    m_name = name;
  }
private:
  string m_name;
};

class Person {
public:
  Person(int age) {
    cout << "I am " << age << " years old. ";
    m_cat = Cat("Alfred");
    m_age = age;
  }
private:
  int m_age;
  Cat m_cat;
};

int main() {
  Person p(21);
}
```

Solution: Construction

This code won't compile! The `Cat` class does not have a default constructor, meaning that its arguments need to be passed in as part of the initializer list.

```
class Person {
public:
  Person(int age) {
    cout << "I am " << age << " years old. ";
    m_cat = Cat("Alfred");
```

To fix this issue, we need to pull out the initialization of `m_cat` like so:

```
Person(int age) : m_cat("Alfred") { ... }
```

If we apply this fix, we would find that the output is as follows:

```
I am a cat: Alfred
I am 21 years old.
```

This ordering is a consequence of the order of construction, where member variables are constructed before the constructor is called.

Practice Question: Cat Construction

```
class Cat {
    string m_name;
  public:
    Cat(string name) {
      m_name = name;
      cout << "I am a cat named " << m_name << endl;
    }
};


int main() {
  Person p(20, "Pusheen");
  Cat c("Kitty");
}
```

```
class Person {
    int m_age;
    Cat* m_cat;
  public:
    Person(int age, string name) {
      m_age = age;
      cout << "I am " << age << " years old" << endl;
      m_cat = new Cat(name);
      cout << "MEOW" << endl;
    }
};
```

What is the output of this code?

Practice Question: Cat Construction

```
class Cat {
    string m_name;
  public:
    Cat(string name) {
      m_name = name;
      cout << “I am a cat named ” << m_name << endl;
    }
};


int main() {
  Person p(20, “Pusheen”);
  Cat c(“Kitty”);
}
```

```
class Person {
    int m_age;
    Cat* m_cat;
  public:
    Person(int age, string name) {
      m_age = age;
      cout << “I am ” << age << “ years old” << endl;
      m_cat = new Cat(name);
      cout << “MEOW” << endl;
    }
};
```

WAIT!! There’s a memory leak!!

How do we fix it?

Solution: Cat Construction

```
class Cat {
    string m_name;
  public:
    Cat(string name) {
      m_name = name;
      cout << “I am a cat named ” << m_name << endl;
    }
};


int main() {
  Person p(20, “Pusheen”);
  Cat c(“Kitty”);
}
```

```
class Person {
    int m_age;
    Cat* m_cat;
  public:
    Person(int age, string name) {
      m_age = age;
      cout << “I am ” << age << “ years old” << endl;
      m_cat = new Cat(name);
      cout << “MEOW” << endl;
    }

    ~Person() {
      delete m_cat;
    }
};
```

Destructors

- A **destructor** is a member function that is called automatically when an object of a class passes out of scope
  - The destructor should use `delete` to eliminate any dynamically allocated variables created by the object
- For example, suppose that our `Person` class creates a **dynamically allocated** `Pet` type object. This memory would need to be freed in the destructor!

Destructors (cont.)

Let's add a destructor to our class!

```
class Pet { ... };

class Person {
public:
  Person();
  ~Person();
  // Same stuff as before ...
private:
  Pet* fluffy;
};
```

```
Person::Person() {
  // Same stuff as before ...
  fluffy = new Pet("Steve");
}

Person::~Person() {
  delete fluffy;
}
```

Practice Question: Destruction

What is the output of the following code snippet?

```
class Cat {
public:
  Cat(string name) {
    cout << "I am a cat: " << name << endl;
    m_name = name;
  }
  ~Cat() { cout << "Farewell, meow." << endl; }
private:
  string m_name;
};
```

```
class Person {
public:
  Person(int age) {
    cout << "I am " << age << " years old. ";
    m_cat = new Cat("Alfred");
    m_age = age;
  }
  ~Person() { cout << "Goodbye!" << endl; }
private:
  int m_age;
  Cat *m_cat;
};

int main() {
  Person p(21);
}
```

Solution: Destruction

We would expect the following output:

```
I am 21 years old. I am a cat: Alfred
Goodbye!
```

Notice that the destructor for `m_cat` is never called: this is a memory leak, which should be addressed by adding `delete` in the `Person` destructor.

Friend - Functions

Sometimes you may want to grant special access to a function not owned by a class, to allow that function access to the class's private data members.

```
class Y {
    int dataInY;
    friend std::ostream& operator<<(std::ostream& out, const Y& o);
    // This doesn't declare a member function! It just marks a function with this specific
    // header as being a "friend" of the Y class.
};

std::ostream& operator<<(std::ostream& out, const Y& y) {
    return out << y.dataInY;
}
```

Friend - Classes

You can also mark an entire class as a friend!

```
class Storage {
    int nValue;
    friend class Display;           // Now all members of Display can access private
                                    // members of Storage.
};
class Display {
    …
    void displayItem(Storage &storage) {
        std::cout << storage.nValue << ‘\n’;
    }
};
```

Copy Constructors

- A **copy constructor** is a constructor that takes one parameter that is of the same type as the class
    - The object being constructed becomes an exact copy of the object passed in as a parameter
- The copy (or move) constructor is called automatically in 3 cases:
    - When a class object is being declared and initialized by another object of the same type
    - When a function returns a value of the class type
    - When a function takes in a parameter of the class type as pass-by-value

Copy Constructors (cont.)

If you don't define a copy constructor, one will be automatically generated for you. However, this default copy constructor simply copies the contents of member variables and does not work correctly if your class has pointers or dynamic data as member variables.
Let's define our own!

```
Person::Person(const Person &other) {
  age = other.age;
  name = other.name;
  money = other.money;
  fluffy = new Pet;                  // If we just had ‘fluffy = other.fluffy’
  *fluffy = *(other.fluffy);         // these two people would share the same
                                     // Pet object in memory!
}
```

Assignment Operators

- An **assignment operator** is called when an already initialized object is assigned a new value from another existing object of the same type
- The assignment operator returns the object on the left side of the = sign (the calling object). Return type is usually `MyClass&`.
    - `MyClass` (and not void) so you can chain assignments, like `p1 = p2 = p3;`
    - Reference (&) so you return the actual changed object, not a copy of it.

```
Person p1(10, “Jim”, 10.0);
Person p2(15, “Tim”, 50.0);
p1 = p2; // assignment operator called
```

Assignment Operators (cont.)

Let's define an assignment operator for our Person class!

```
Person& Person::operator=(const Person& rtSide) {
  if (this == &rtSide)                   // The left side and the right side of the = sign are the same!
    return *this;
  else {
    age = rtSide.age;
    name = rtSide.name;
    money = rtSide.money;
    delete fluffy;                       // Delete the old instance of Pet before creating a new one.
    fluffy = new Pet;
    *fluffy = *(rtSide.fluffy);
    return *this;                        // Return the calling object (left side of the = sign).
  }
}
```

Pointers to Objects – Arrow Operator

Like the dot operator, the **arrow operator ->** can be used to access an object's member variables and functions. The arrow operator is used when we have a pointer to the object whose members we are trying to reference.

```
int main() {
  Person* p = new Person;
  p->setAge(20);
  p->setName("Bob");
  double money = p->getMoney();
  p->age = 10;          // ERROR: age is not a public variable!
}
```

Note: `p->setAge(20)` and `(*p).setAge(20)` are equivalent statements!

Pointers to Objects – The `this` Pointer

When defining member functions for a class, we sometimes want to refer to the calling object. The `this` pointer is a predefined pointer that points to the calling object.

```
int Person::getAge() {
  return age;
}
```

```
int Person::getAge() {
  return this->age;
}
```

Note: the above two definitions for the function `getAge()` are equivalent, but the left method is clearer and better stylistically.

Pointers to Objects – The `this` Pointer

- The `this` pointer allows us to access member variables even when they are shadowed by local variables.

```
Person::setAge(int age) {
    this->age = age;
}
```

- The `this` pointer also allows us to pass the current object into a function that takes an argument of its class.

```
void printPerson(Person *p);
Person::print() {
    printPerson(this);
}
p1.print()
```

Function Overloading

- You can have multiple definitions for the same function name in the same scope. However, the definition of the functions must differ from each other by the types and/or the number of arguments in the argument list.

```
class printData {
   public:
      void print(int i) {
         cout << "Printing int: " << i << endl;
      }

      void print(int i, double f) {
         cout << "Printing numbers: " << f << ‘ ‘ << i << endl;
      }

      void print(char* c) {
         cout << "Printing character: " << c << endl;
      }
};
```

```
int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print numbers
   pd.print(42, 500.263);

   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

Operator Overloading

- You can also overload basic operators in C++ (such as +, -, <<, etc.) so they work with user-defined structs and classes as well.

```
class Vector {
   public:
      double getX() {
         return m_x;
      }

      double getY() {
         return m_y;
      }

      void setX(int x) {
         m_x = x;
      }

      void setY(int y) {
         m_y = y;
      }

      Vector operator+(const Vector& vec) {
         Vector newVec;
         newVec.setX(this->m_x + vec.getX());
         newVec.setY(this->m_y + vec.getY());
         return newVec;
      }
  private:
      double m_x;      // x component of vector
      double m_y;      // y component of vector
};
```

Good luck!

Sign-in http://bit.ly/33Lgelj
Slides http://bit.ly/2LfiR8N
Practice https://github.com/uclaupe-tutoring/practice-problems/wiki

Questions? Need more help?

- Come up and ask us! We'll try our best.
- UPE offers daily computer science tutoring:
    - Location: ACM/UPE Clubhouse (Boelter 2763)
    - Schedule: https://upe.seas.ucla.edu/tutoring/
- You can also post on the Facebook event page.