EECS 280 – Lecture 9

Derived Classes and Inheritance

2/17/2021

Review: Constructors

```
class Triangle {
private:
  double a;
  double b;
  double c;

public:

  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {

    // Nothing to do in function body.
    // This isn't always the case.
  }

  Triangle()
    : a(1), b(1), c(1) { }
};
```

Same "name" as the class.

Parameters receive arguments provided when making a Triangle object.

A special syntax for initializing members in a constructor.

A default constructor takes no arguments.

Delegating Constructors

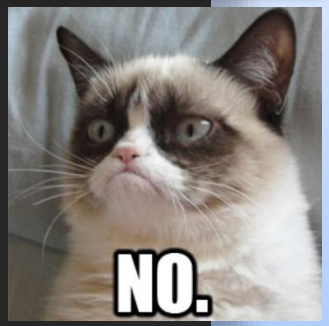
```
class Triangle {
  ...
  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {
    // Be assertive!
    assert(0 < a && 0 < b && 0 < c);
    assert(a + b > c  &&  a + c > b  &&  b + c > a);
  }

  Triangle(double side_in)
    : a(side_in), b(side_in), c(side_in) {
    // Be assertive!
    assert(0 < a && 0 < b && 0 < c);
    assert(a + b > c  &&  a + c > b  &&  b + c > a);
  }

  Triangle(double side_in)
    : Triangle(side_in, side_in, side_in) { }

};
```

Check that the inputs will make a valid Triangle.

Code duplication!

Better

Is this a good approach?



Delegating Constructors

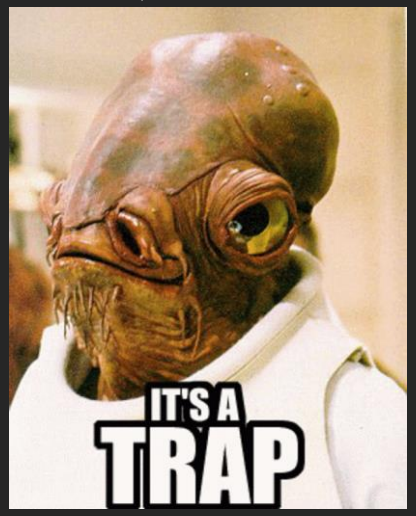
```
class Triangle {
  ...
  Triangle(double a_in, double b_in, double c_in)
    : a(a_in), b(b_in), c(c_in) {
    // Be assertive!
    assert(0 < a && 0 < b && 0 < c);
    assert(a + b > c  &&  a + c > b  &&  b + c > a);
  }


  Triangle(double side_in)
    : Triangle(side_in , side_in, side_in) {
  }

  Triangle(double side_in) {
    Triangle(side_in , side_in, side_in);
  }
};
```

DO

DON'T

Delegation has to be done in an initializer list.

Compiler basically sees:
`Triangle temp(side_in, side_in, side_in);`



Exercise: `Rectangle`

```
class Chicken {
private:
  int age;
  string name;
  int roadsCrossed;

public:









};
```

- Write a `Chicken` ADT using classes.
- It should have…
  - A constructor that takes in a name, an age, and a number of roads crossed and initializes the corresponding members appropriately.
  - A constructor that takes in only a name. It should initialize the chicken's `name` and also set its `age` and `roadsCrossed` to `0`.
  - The following member functions:
    - `getName()`, `getAge()` - return those things
    - `crossRoad()` – increases # of roads crossed by 1
    - `talk()` – prints out "bawwk" to cout
- Hint: It's a lot like `Triangle`, but less geometric.

*Generally, there's no time for this exercise in lecture. But here it is in case you'd like to do it on your own.

Solution: Chicken

```
class Chicken {
private:
  int age;
  string name;
  int roadsCrossed;

public:
  Chicken(const string &name_in,
          int age_in, int roads_in)
    : age(age_in), name(name_in),
      roadsCrossed(roads_in)
  {
    assert(check_invariants());
    cout << "Chicken ctor" << endl;
  }

  Chicken(const string &name_in)
    : Chicken(name_in, 0, 0) { }

  ...
```

Data representation

A delegating constructor.

```
  ...

  const string & getName() const {
    return name;
  }

  int getAge() const {
    return age;
  }

  void crossRoad() {
    ++roadsCrossed;
  }

  void talk() const {
    cout << "bawwk" << endl;
  }

private:
  bool check_invariants() {
    return age > 0
      && roadsCrossed > 0;
  }
};
```

C++ Style Bird ADTs

- Let's consider ADTs to represent birds…

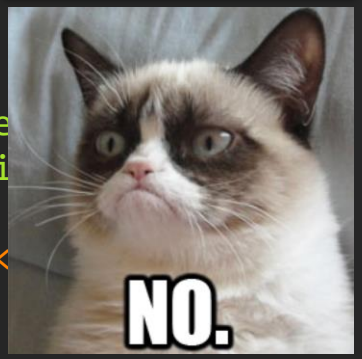
Lots of code duplication!

C++ Style Bird ADTs

```
class Chicken {
private:
  int age;
  string name;
  int roadsCrossed;

public:
  Chicken(const string &name_in)
    : age(0), name(name_in),
      roadsCrossed(0) {
    cout << "Chicken ctor" << endl;
  }

  string getName() const { return name; }
  int getAge() const { return age; }

  void crossRoad() {
    ++roadsCrossed;
  }
  void talk() const {
    cout << "bawwk" << endl;
  }
};
```

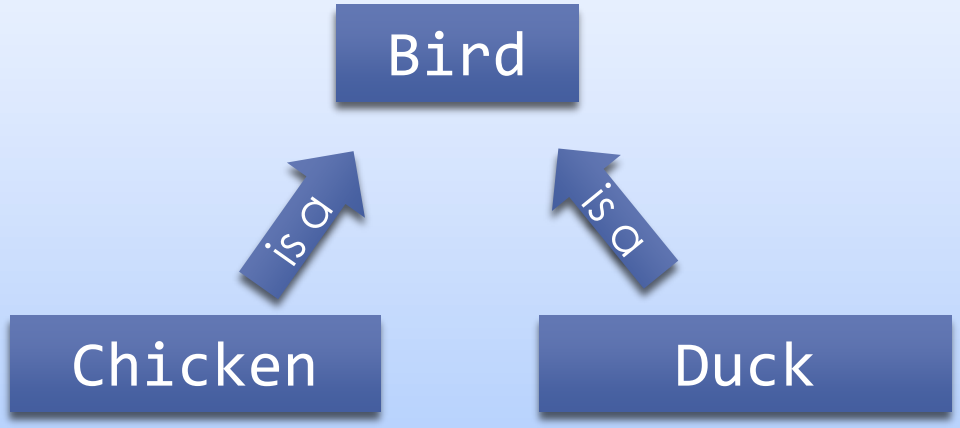
```
class Duck {
private:
  int age;
  string name;
  int numDucklings;

public:
  Duck(const string &name
    : age(0), name(name_i
      numDucklings(0) {
    cout << "Duck ctor" <
  }

  string getName() const { return name; }
  int getAge() const { return age; }

  void babyDucklings() {
    numDucklings += 7;
  }
  void talk() const {
    cout << "quack" << endl;
  }
};
```

Is this a good approach?



Inheritance

- Consider ADTs that represent birds…
- Intuitively, `Chicken` and `Duck` ADTs are both specific kinds of `Birds`.



- We will use `Bird` as a **base type**, with `Chicken` and `Duck` as **derived types**.
- This is called **inheritance**.

Demo: Basic Inheritance

- Open `L09.0_Inheritance_basic` on Lobster and follow along.

lobster.eecs.umich.edu

Birds and Inheritance

Base Class

```
class Bird {
private:
  int age;
  string name;

public:
  Bird(const string &name_in)
    : age(0), name(name_in) {
    cout << "Bird ctor" << endl;
  }

  string getName() const {return name;}
  int getAge() const { return age; }

  void haveBirthday() { ++age; }

  void talk() const {
    cout << "tweet" << endl;
  }
};
```

All birds have these members

Derived Class

```
class Chicken : public Bird {
private:
  int roadsCrossed;

public:
  Chicken(const string &name_in)
    : Bird(name_in), roadsCrossed(0) {
    cout << "Chicken ctor" << endl;
  }

  void crossRoad() { ++roadsCrossed; }

  void talk() const {
    cout << "bawwk" << endl;
  }
};
```

Additional members for Chickens

Pass to base class constructor!

Birds and Ducks

Base Class

```
class Bird {
private:
  int age;
  string name;

public:
  Bird(const string &name_in)
    : age(0), name(name_in) {
    cout << "Bird ctor" << endl;
  }

  string getName() const {return name;}
  int getAge() const { return age; }

  void haveBirthday() { ++age; }

  void talk() const {
    cout << "tweet" << endl;
  }
};
```

All birds have these members

Derived Class

```
class Duck : public Bird {
private:
  int numDucklings;

public:
  Duck(const string &name_in)
    : Bird(name_in), numDucklings(0) {
    cout << "Duck ctor" << endl;
  }

  void babyDucklings() {
    numDucklings += 7;
  }

  void talk() const {
    cout << "quack" << endl;
  }
};
```

Additional members for Ducks

Pass to base class constructor!

Calling a Base-Class Ctor

- A constructor in a derived class **always** calls a constructor for a base class.
- If no explicit call is made in the member initializer list, the default constructor of the base is implicitly called.

```
class Chicken : public Bird {
  Chicken(const string &name_in)
    : Bird(name_in), roadsCrossed(0) {}
  ...
};
```

OK: explicit call to base ctor

```
class Chicken : public Bird {
  Chicken() : roadsCrossed(0) {}
  ...
};
```

Error: base class has no default ctor

```
class Square : public Rectangle {
  Square() {}
  ...
};
```

OK: implicit call to default base ctor

Varieties of Inheritance

- C++ offers three different varieties of inheritance.
  - `public`
  - `protected`
  - `private`
- We only cover public inheritance. Don't worry about the other ones.

```
class Duck : public Bird {
   ...
   ...
   ...
};
```

Specify which kind of inheritance you want like this. If left blank, defaults to `private`.[1]

1 (Or with a `struct` defaults to `public`.)

Anita Borg

Operating Systems Research
Founder of the Institute for Women and Technology
and the Grace Hopper Conference

Anders Hejlsberg

Lead architect of several programming languages, including Delphi, C#, and TypeScript

Break Time!

We'll start again in one minute.

Demo: Birds

- Open `L09.1_Birds` on Lobster and follow along.

lobster.eecs.umich.edu

Inheritance and Memory

- In memory, a derived class object has a base class part.
- You don't have to do anything special to access it.
- Member access with . or -> will find base members too.

```
int main() {
  Chicken c("Myrtle");
  Duck d("Scrooge");
  Bird b("Big Bird");
  c.age;
}
```

Technically doesn't work here since age is private, though.

"Automatically" refers to the age member in the Bird part of the Chicken c.

Compound Object Lifetimes

- Constructors are called whenever a class object is created for the first time.

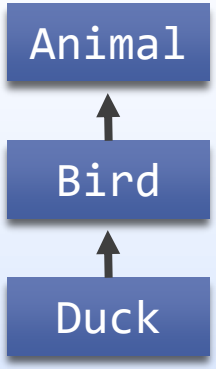
```
Triangle()
  : a(1), b(1), c(1) {
  cout << "Triangle ctor" << endl;
}
```

- Destructors are called whenever a class object's lifetime ends (depends on storage duration).
  - For local variables, when they go out of scope.

```
~Triangle() {
  cout << "Triangle dtor" << endl;
}
```

- We will talk about destructors in more detail later.

Ctors and Dtors in Derived Types



- Destructors are the analog of constructors, called when an object is destroyed.
- When creating or destroying an object of a class with a base type, a constructor or destructor is used for each level of the hierarchy.
- For **constructors**, we get **top-down** behavior.

```
int main() {
  Duck d("Scrooge");  // Animal ctor, Bird ctor, Duck ctor
  Bird b("Big Bird"); // Animal ctor, Bird ctor
  ...
```

- For **destructors**, we get **bottom-up** behavior.

```
  ...
  // b dies: Bird dtor, Animal dtor
  // d dies: Duck dtor, Bird dtor, Animal dtor
};
```

We will learn more about destructors later in the term.

Member Name Lookup

- When we use `.` or `->` for member access, how does the compiler look up the member's name?
- Start in the first class scope.
- If not found, try base class scope as well.
- Stop whenever a matching name is found.

```
class Bird {
  int age;
  string name;
  Bird(const string &name_in);
  void talk() const;
  string getName() const;
  int getAge() const;
};
```

```
int main() {
  Chicken c("Myrtle");
  Duck d("Scrooge");
  Bird b("Big Bird");
  c.getAge();
}
```

```
class Chicken : public Bird {
  int roadsCrossed;
  Chicken(const string &name_in);
  void talk() const;
};
```

We want to look up a member named getAge.

Tip: Access levels are **only** considered after name lookup.

Name Hiding

- When we use `.` or `->` for member access, how does the compiler look up the member's name?
- Start in the first class scope.
- If not found, try base class scope as well.
- **Stop whenever a matching name is found.**

Hidden

```
class Bird {
  void talk() const {
    cout << "tweet" << endl;
  }
  string getName() const;
  int getAge() const;
};
```

We want to look up a member named talk.

```
int main() {
  Chicken c("Myrtle");
  Duck d("Scrooge");
  Bird b("Big Bird");
  c.talk();
}
```

```
class Chicken : public Bird {
  void talk() const {
    cout << "bawwk" << endl;
  }
};
```

Tip: Access levels are **only** considered after name lookup.

Name Hiding

- Name hiding can have tricky consequences.
- ONLY the **name**, not the signature, is considered for name lookup.

```
class Bird {
  void talk() const {
    cout << "tweet" << endl;
  }
  string getName() const;
  int getAge() const;
};
```

Hidden

```
class Chicken : public Bird {

  void talk(int volumeLevel) const {
    if (volumeLevel > 3) {
      cout << "BAWWWWK" << endl;
    }
    else {
      cout << "bawwk" << endl;
    }
};
```

We want to look up a member named talk.

```
int main() {
  Chicken c("Myrtle");
  c.talk();
}
```

Compiles error! Parameter types don't match.

Tip: Access levels are **only** considered after name lookup.

Accessing a Hidden Name

- We can use the scope resolution operator to access a hidden name (i.e. by using a **qualified name**).

```
class Bird {
private:
  int age;
  string name;

public:

  ...

  void talk() const {
    cout << "tweet" << endl;
  }
};
```

```
class Chicken : public Bird {
private:
  int roadsCrossed;

public:

  ...

  void talk() const {
    if (age >= 1) {
      cout << "bawwk" << endl;
    } else {
      // baby chicks tweet
      Bird::talk();
    }
  }
};
```

Problem: age is private in Bird

Call Bird's version of talk()

The protected Access Level

- We can make a member accessible to derived classes using **protected** instead of **private**.

```
class Bird {
protected:
  int age;
  string name;

public:

  ...

  void talk() const {
    cout << "tweet" << endl;
  }
};
```

Problem: this reveals implementation details of Bird

```
class Chicken : public Bird {
private:
  int roadsCrossed;

public:

  ...

  void talk() const {
    if (age >= 1) {
      cout << "bawwk" << endl;
    } else {
      // baby chicks tweet
      Bird::talk();
    }
  }
};
```

Better Solution

- Use a public "getter" function instead.

```
class Bird {
private:
  int age;
  string name;

public:

  ...

  int getAge() const {
    return age;
  }

  void talk() const {
    cout << "tweet" << endl;
  }
};
```

```
class Chicken : public Bird {
private:
  int roadsCrossed;

public:

  ...

  void talk() const {
    if (getAge() >= 1) {
      cout << "bawwk" << endl;
    } else {
      // baby chicks tweet
      Bird::talk();
    }
  }
};
```

Exercise: Compile Errors

- `L09.2_Birds_compile` on Lobster has compile errors.
  - Determine "why" each is being given.
  - What conceptual mistake is being made, or what needs to be done to the code to fix the problem?

Question

Which of the following does NOT describe a bug in the code?

A) Only chickens can cross roads.

B) A base class constructor call is missing somewhere.

C) A private member variable is accessed in the wrong place.

D) One of the classes forgets to declare its base class.

E) A const is missing somewhere.

Exercise: Runtime Errors

- `L09.3_Birds_runtime` on Lobster has **runtime** errors.
  - Make sure to click past the `memoryJunk` function at the beginning. It's just there to fill memory with random values and make the errors more obvious.
  - Find each bug and see if you can fix the code too!

Question

Which of the following does NOT describe a bug in the code?

A) Bird names all end up empty.

B) Ducks end up with a random number of ducklings.

C) The Chicken constructor never calls the Bird constructor.

D) Chickens should say bawwk, not tweet.

lobster.eecs.umich.edu (or link from eecs280.org)