Lecture 4a: Object Life-Cycle
(Cows and Barns and Chickens and Things)

CS 70: Data Structures and Program Development

Tuesday, February 11

Learning Targets

1. I can identify when objects are initialized and destroyed
2. I know the purpose of constructors, default constructors, destructors, copy constructors, and assignment operators
3. I can identify when these functions are implicitly called in a piece of code
4. I can use the default and delete keywords in a class declaration

Recall: Data Life-Cycle

Every *individual* piece of data, over the course of its life:

1. **Allocation**: acquire memory for the data
2. **Initialization**: create the data
3. **Use**: read and/or modify the data
4. **Destruction**: clean up the data
5. **Deallocation**: relinquish the data's memory

When an object is initialized, its *constructor* is invoked.

When an object is destroyed, its *destructor* is invoked.

A Simple Chicken Class

chicken.hpp:

```
class Chicken {
  public:
    bool isHatched();

  private:
    bool hatched_;
};
```

chicken.cpp:

```
bool Chicken::isHatched() {
    return hatched_;
}
```

Using the class in main.cpp:

```
int main() {
  Chicken henny;
  cout << henny.isHatched() << endl;
}
```

What happens?

Synthesized Default Constructor

```
Chicken::Chicken() {
  //All members are default-initialized.
  //Primitive type members have undefined value.
  //Object members are default-constructed.
  //Nothing more to do!
}
```

Our Own Default Constructor

chicken.hpp:

```
class Chicken {
  public:
    Chicken();
    bool isHatched();

  private:
    bool hatched_;
};
```

chicken.cpp:

```
Chicken::Chicken():hatched_{false}
{}

bool Chicken::isHatched() {
    return hatched_;
}
```

Using the class in main.cpp:

```
int main() {
  Chicken henny; // Could also do Chicken henny{};
  cout << henny.isHatched() << endl; // hatched_ is false
}
```

Sometimes We Don't Want a Default Constructor

cow.hpp

```
class Cow{
  public:
    Cow() = delete; // Don't synthesize
    Cow(size_t numSpots, size_t age);

    void moo(size_t numMoos);

  private:
    size_t spots_;
    size_t age_;
};
```

main.cpp

```
Cow bessy; // Will not compile!
```

A Barn

```
class Barn{
  public:
    Barn() = delete;
    Barn(size_t numSpots, size_t age);

  private:
    Cow lonelyCow_;
};
```

```
Barn::Barn(size_t numSpots, size_t age) :
    lonelyCow_{numSpots, age}
{}
```

A Problematic Barn

```
class Barn{
  public:
    Barn() = delete;
    Barn(size_t numCows, string filename);
  private:
    Cow* cowArr_;
    size_t numCows_;
};
```

A Problematic Barn (continued)

```
  Barn::Barn(size_t numCows, string filename) :
      numCows_{numCows}, cowsArr_{new Cow[numCows]}
  {
    ifstream fin{filename};
    for (size_t i = 0; i < numCows; ++i) {
      size_t numSpots;
      size_t age;
      cin >> numSpots;
      cin >> age;
      //Initialize the cow at cowArr_[i]?
    }
}
```

Fixing the Barn

```
class Barn{
  public:
    Barn() = delete;
    Barn(size_t numCows, string filename);

  private:
    size_t numCows_;
    Cow** cowArr_;
};
```

Fixing the Barn (continued)

```
Barn::Barn(size_t numCows, string filename) :
    numCows_{numCows}, cowsArr_{new Cow*[numCows]}
    //This default constructs Cow*s (that's okay!)
{
  ifstream fin{filename};
  for (size_t i = 0; i < numCows; ++i) {
    size_t numSpots;
    size_t age;
    cin >> numSpots;
    cin >> age;

    cowArr_[i] = new Cow{numSpots, age};
  }
}
```

Using the Barn

```
void f() {
  Barn barney{2, "cowcensus.txt"};
}
```

What's the problem?

Memory leak! The Barn is gone but the Cows are still around.

Synthesized Destructor

```
Barn::~Barn() {
     //No special instructions
     //When this function returns
     //all data members are destroyed
     //(last to first)
}
```

Our Own Destructor

In barn.hpp:

```
class Barn{
    public:
        Barn() = delete;
        Barn(size_t numCows, string filename);
        ~Barn();

    private:
        size_t numCows_;
        Cow** cowArr_;
};
```

In barn.cpp:

```
Barn::~Barn() {
    //Whatever needs to happen when Barns are destroyed
}
```

Cleaning Up The Barn

```
Barn::~Barn() {
  for (size_t i = 0; i < numCows_; ++i) {
    delete cowArr_[i];
  }
  delete[] cowArr_;
}
```

Sometimes We Want The Synthesized Destructor

```
class Cow{
  public:
    Cow() = delete; // Don't synthesize
    Cow(size_t numSpots, size_t age);
    ~Cow() = default; // Synthesize

    void moo(size_t numMoos);

  private:
    size_t spots_;
    size_t age_;
};
```

Exercise

What functions are called on each line?

```
void barnyard() {
  Chicken* a = new Chicken{};

  Cow b{2, 3};

  Chicken c[3];

  Barn d{3, "cows.txt"};

  delete a;
}
```

Secret Cow Cloning Program

```
void printCow(Cow c) {
  cout << c.getNumSpots() << " " << c.getAge() << endl;
}

int main() {
  Cow bessie{5, 8};
  printCow(bessie);
}
```

How does c get initialized?

Synthesized Copy Constructor

```
Cow::Cow(const Cow& other) :
    spots_{other.spots_}, age_{other.age_}
{
  // All data members are copy-constructed.
  // Nothing more to do!
}
```

Explicitly invoking the Copy Constructor:

```
Cow audrey{5, 8};
Cow audrey2{audrey}; // copy constructed
```

Using The Synthesized Copy Constructor

```
class Cow{
  public:
    Cow() = delete; // Don't synthesize
    Cow(size_t numSpots, size_t age);
    ~Cow() = default; // Synthesize

    Cow(const Cow& other) = default; // Synthesize

    void moo(size_t numMoos);

  private:
    size_t spots_;
    size_t age_;
};
```

A Problematic Barn

```
class Barn{
  public:
    Barn() = delete;
    Barn(size_t numCows, string filename);
    ~Barn();

    Barn(const Barn& other) = default;

  private:
    size_t numCows_;
    Cow** cowArr_;
};
```

A Problematic Barn (continued)

```
int main() {
  Barn barney{2, "cowcensus.txt"};
  Barn barney2{barney};
}
```

What’s wrong?

Fixing Barn: Our Own Copy Constructor

```
class Barn{
  public:
    Barn() = delete;
    Barn(size_t numCows, string filename);
    ~Barn();

    Barn(const Barn& other);

  private:
    size_t numCows_;
    Cow** cowArr_;
};
```

Fixing Barn: Our Own Copy Constructor (continued)

```
Barn::Barn(const Barn& other) :
  numCows_{other.numCows_}, cowArr_{new Cow*[numCows_]}
{
  for (size_t i = 0; i < numCows_; ++i) {
    cowArr_[i] = new Cow{other.cowArr_[i]};
  }
}
```

Exercise

What functions are called on each line?

```
Cow* cowsAbound(Cow a, Cow& b, Barn c) {
    Cow d{b};
    Cow* e = new Cow{2, 3};
    return e;
}

int main() {
    Cow w{4, 9};
    Cow x{2, 12};
    Barn y{4, "cowstats.txt"};
    Cow* z = cowsAbound(w, x, y);
    delete z;

    return 0;
}
```

Assignment

```
Cow bessie{5, 8};
Cow bartholomoo{3, 10};
bartholomoo = bessie;
```

What will happen?

Assignment Operator

```
Cow bessie{5, 8};
Cow bartholomoo{3, 10};
bartholomoo = bessie;
```

equivalent to…

```
bartholomoo.operator=(bessie);
```

Technically, `operator=` returns the object that was just modified.

That's so you can do things like `x = y = z`
(but don't do that).

Synthesized Assignment Operator

```
Cow& Cow::operator=(const Cow& rhs) {
  //Overwrite each data member
  numCows_ = rhs.numCows_;
  cowArr_ = rhs.cowArr_;

  //Return the object we just modified
  return *this;
}
```

Note: `this` is an implicit parameter to every member function. It stores the address of the object that the function was called on.

So `*this` is a name for the object itself!

Using The Synthesized Assignment Operator

```
class Cow{
  public:
    Cow() = delete; // Don't synthesize
    Cow(size_t numSpots, size_t age);
    ~Cow() = default; // Synthesize

    Cow(const Cow& other) = default; // Synthesize
    Cow& operator=(const Cow& rhs) = default; // Synthesize

    void moo(size_t numMoos);

  private:
    size_t spots_;
    size_t age_;
};
```

A Problematic Barn

```
class Barn{
  public:
    Barn() = delete;
    Barn(size_t numCows, string filename);
    ~Barn();

    Barn(const Barn& other);
    Barn& operator=(const Barn& rhs) = default;

  private:
    size_t numCows_;
    Cow** cowArr_;
};
```

Our Own Assignment Operator

```
class Barn{
  public:
    Barn() = delete;
    Barn(size_t numCows, string filename);
    ~Barn();

    Barn(const Barn& other);
    Barn& operator=(const Barn& rhs);

  private:
    size_t numCows_;
    Cow** cowArr_;
};
```

A Bad Idea…

```
Barn& Barn::operator=(const Barn& rhs) {
  numCows_ = rhs.numCows_;
  for (size_t i = 0; i < numCows_; ++i) {
    cowArr_[i] = rhs.cowArr_[i];
  }
}
```

An Almost Good Idea…

```
Barn& Barn::operator=(const Barn& rhs) {
  for (size_t i = 0; i < numCows_; ++i) {
    delete cowArr_[i];
  }
  delete [] cowArr_;

  numCows_ = rhs.numCows_;
  cowArr_ = new Cow*[numCows_];
  for (size_t i = 0; i < numCows_; ++i) {
    cowArr_[i] = new Cow{rhs.cowArr_[i]};
  }
}
```

But what happens when…

```
Barn barney{2, "cowcensus.txt"};
barney = barney;
```

A Good Idea: Copy and Swap

```
#include <utility>
Barn& Barn::operator=(const Barn& rhs) {
  //First make a copy
  Barn tmp{rhs}; //We trust the copy constructor!

  //Next swap members with the copy
  std::swap(numCows_, tmp.numCows_); //swap is built-in!
  std::swap(cowArr_, tmp.cowArr_);   //(remember the #include)

  //Now *this is a copy of rhs
  return *this;

  //tmp has *this' old stuff...
} //tmp is destroyed here (we trust the destructor!)
```

Summary

- **Constructors**
  - Invoked when an object is initialized
  - Set up the object’s members
  - Which constructor is invoked depends on parameters
- **Default Constructor**
  - Invoked for default initialization
  - Constructor with no parameters
- **Destructor**
  - Invoked when an object is destroyed
  - Cleans up the object’s members
  - Name is ~ClassName()

Summary (continued)

- **Copy Constructor**
  - Invoked when a copy is made (e.g. parameter passing)
  - Takes a const reference of the same type
  - Makes a copy (used for parameter passing etc.)
- **Assignment operator**
  - Invoked when an object is assigned to an existing object
  - Defined by a member function named `operator=`
  - Takes a const reference to the right hand side of =
  - Returns a reference to the object that was modified

Rules

Always define, default or delete

- The default constructor
- The destructor
- The copy constructor
- The assignment operator

The Rule of 3

- If you need to define one of these…
  - Destructor
  - Copy constructor
  - Assignment operator
- …then you probably need to define them all
- (Otherwise probably default them all)

Tricky Synax

What's happening here?

```
Cow bessie{5, 8};
Cow bartholomoo = bessie;
```

It turns out that this is equivalent to

```
Cow bessie{5, 8};
Cow bartholomoo{bessie};
```

(`bartholomoo` is being initialized!)

Exercise

What functions are called on each line?

```
void cowParty() {
  Cow a{4, 9};
  Cow b{2, 12};
  Cow c{a};
  Cow d = b;
  Cow& e = d;
  b = a;
  e = b;
  Barn f{4, "cowstats.txt"};
  Barn g{3, "cowlist.txt"};
  g = f;
}
```

Warning

What's happening here?

```
Cow bessie = Cow{5, 8}; //Don't ever write this!
```

This:

1. Constructs a Cow on the right-hand side
2. Copy constructs `bessie` using that Cow
3. Later destroys that temporary Cow

That's so much more work than just initializing `bessie`!
(So pay attention to CS70 C++ idioms!)