COMP 1633: Intro to CS II

Copying Objects

And a bit of inheritance

Charlotte Curtis

April 3, 2024

Where we left off

- Thinking recursively
- Recursion with linked lists
- Tail recursion
- Some more examples with recursion

> *Textbook Chapter 14*

```
void hanoi(int n, int src,
           int dest, int spare) {
    if (n == 1) {
        cout << "Move disk from " << src
             << " to " << dest << endl;
    } else {
        hanoi(n-1, src, spare, dest);
        hanoi(1, src, dest, spare);
        hanoi(n-1, spare, dest, src);
    }
}
```

Today's topics

- Back to classes! Last day of new stuff
- Copying objects
- Copy constructors
- Overloading the assignment operator
- Touch on inheritance, if we have time

> *Textbook Section 11.4*

Copying objects

- In assignment 4, you can treat a `Team` just like a built-in type
- This means you can do things like:

```
Team t1;
source >> t1;
Team t2 = t1;
```

- Or have a function that returns a *copy* of an `Team`:

```
Team copy_of_first() {
    return head->team;
}
```

- But what does it mean to copy an object?

Default copying

- Copying occurs whenever you:
    i. Assign an object to another instance
    ii. Pass an object to a function by value
    iii. Return an object from a function by value
    iv. Initialize an object with another instance
- With the exception of case i, these all involve **creating a new object** (we'll come back to case i later)
- C++ provides a default **copy constructor** to handle this

Case iv: Initializing an object

- With **primitives**, it makes sense to do the following:

```
int x = 5; // allocate an int with value 5
int y = x; // allocate another int with value 5
```

- This works for **objects** too:

```
Team t1; // allocate an Team with default values
Team t2 = t1; // allocate another Team with the same values
```

- The compiler will use the **copy constructor** to initialize `t2`, equivalent to:

```
Team t2(t1); // instantiate an Team with the same values as t1
```

- Just like the default constructor, a default copy constructor is provided

Default copy constructor

- The default copy constructor does a **shallow copy**, where the **value** of each member variable is copied
- If the member is a **statically allocated** array, the array is copied **element by element** (the behaviour described way back in lecture 10)
- Let's draw a diagram of the following copy operations:

```
struct Foo {
    int var1;
    double var2;
    char var3[4];
};
```

```
Foo f1 = {1, 2, "hi"};
Foo f2 = f1;
```

Copying pointer members

What if the member variable is a **pointer**, such as the `head` of a linked list?

```
class StringStack {
public:
...
private:
    struct Node {
        std::string data;
        Node *next;
    };
    Node *head;
};
```

- Just like other data types, the **value** of the pointer is copied
- This means that the copy will point to the **same** `Node` as the original
- What if we pass `StringStack` objects by value?

Passing objects by value

- Recall: passing an object by value creates a **local copy** in the function
- When the function finishes, the local copy is **destroyed**

```
void addstuff(StringStack s, std::string stuff) {
    s.push(stuff);
}
```

- Let's trace what happens when we call `addstuff` with a `StringStack`

> *Main takeaway: shallow copy is not enough for dynamically allocated data*

Overriding the default copy constructor

- To enable a **deep copy**, we need to override the default copy constructor
  - **Overload**: same function name, different signature
  - **Override**: same function name, same signature, **replaces** previous
- The copy constructor has one parameter: a **reference** to another instance

```
class StringStack {
public:
    StringStack(); // parameterless constructor
    ~StringStack(); // destructor
    StringStack(const StringStack &other); // copy constructor
};
```

> *Why does the parameter need to be a reference?*

Implementing the copy constructor

- The copy constructor is just like any other function, BUT:
  - it **cannot** call any methods that take or return an object by value
  - This **includes** the assignment operator for the class
- Since the copy constructor is called automatically in these scenarios, we would end up with **infinite recursion**
- The copy constructor must be written from scratch

A copy constructor for `StringStack`

```
StringStack::StringStack(const StringStack &other) {
    head = NULL;

    // copy the contents of other
    Node *curr = other.head;
    while (curr) {
        push(curr->data);
        curr = curr->next;
    }
}
```

> *Note: we can call `push` because it doesn't copy a `StringStack`*

Overriding the assignment operator

- Remember the 4 cases where copying occurs?
- Case i is **assignment**:

```
StringStack s1;
StringStack s2;
s1 = s2; // not the same as all in one line!
```

- The copy constructor is not called because `s2` already exists
- Anyone using this class would expect the assignment operator to behave like the copy constructor (i.e. deep copy)
- Good thing we know how to override operators

Overriding the assignment operator

- Almost the same as the copy constructor, but:
  - any existing data in the object must be **destroyed** first
  - the return type must be a **reference** to the object
- The reference requirement is to allow for **assignment chaining**:

```
int x, y, z;
x = y = z = 5; // this is legal!
```

- Finally, we need to consider the possible (legal, but weird) case:

```
StringStack s1;
... // do stuff with s1
s1 = s1; // self-assignment
```

Implementing the assignment operator

```
StringStack& StringStack::operator = (const StringStack &other) {
    if (this == &other) // check for self-assignment
        return *this;

    while (!empty()) // destroy existing data
        pop();

    Node *curr = other.head; // copy the contents of other
    while (curr) {
        push(curr->data);
        curr = curr->next;
    }

    return *this;
}
```

Side tangent: avoiding code duplication

- The copy constructor and assignment operator are **very similar**
- It can be tempting to just call one from the other, i.e.:

```
StringStack::StringStack(const StringStack &other) {
    *this = other;
}
```

- This is a **bad idea** because `=` only works on **existing objects**
- Instead, we can extract the common code into a **private helper function** and call it from both, e.g. `void copy_elements(const StringStack &other);`

The Rule of Three or "The Big Three"

Classes with **dynamically allocated data** should always have:

1. A **destructor** to free the memory
2. A **copy constructor** to make a deep copy
3. An **assignment operator** to make a deep copy

> *If you need to write one of these, you probably need to write all three. That said, copy constructors and assignment operators are **not required** for assignment 4.*

Copying check-in 1/2

Which of the following is **not** a case where the copy constructor is called?

A. Passing an object to a function by value

B. Returning an object from a function by value

C. Initializing an object with another instance

D. Allocating an object dynamically with `new`

E. None of the above, they all call the copy constructor

Copying check-in 2/2

When dealing with classes with dynamically allocated members, the default copy constructor might:

A. Cause **a dangling pointer** to be created

B. Cause **a memory leak** to occur

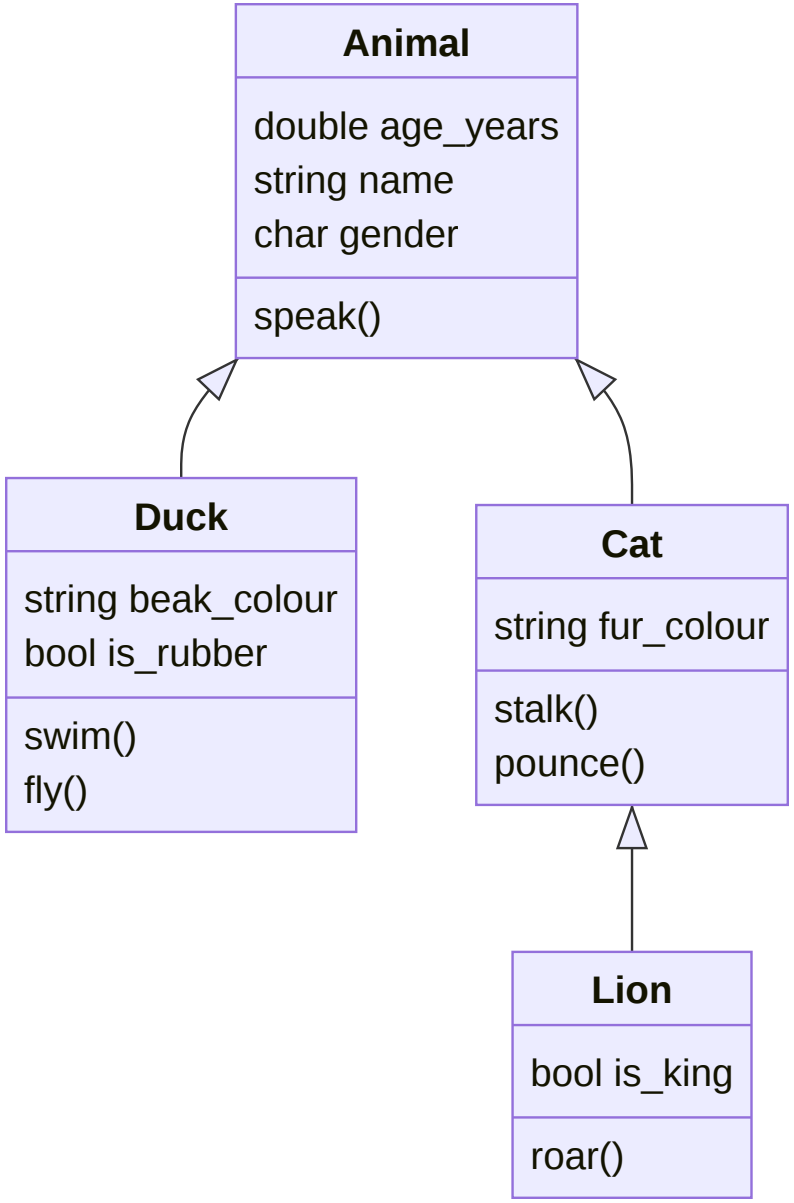
C. Cause a **segmentation fault** to occur

D. All of the above

E. None of the above

A (very brief) intro to Inheritance

- Classes are a great way to provide **abstraction** and **encapsulation**
- But there's a lot of **repetition** in the code we write
- Inheritance allows us to **reuse** code from other (similar) classes



Inheritance

AKA creating a new class from an existing one

- Inheritance is an "is-a" relationship
  - A `Duck` is an `Animal`
  - A `Cat` is an `Animal`
  - A `Lion` is a `Cat`
- The new class is called a **subclass**, **derived class**, or **child class**
- The existing class is called a **superclass**, **base class**, or **parent class**
- The derived class **inherits** all the members of the base class

What gets inherited?

- The derived class gets all the members of the base class **except** the constructor, destructor, copy constructor, and assignment operator
  - A `Cat` has private members `age_years`, `name`, and `gender`
  - A `Lion` can `stalk()` and `pounce()`, but also `roar()`
- All `Animal`s can `speak()`, but what does that mean?
- A derived class can **override** a member of the base class
  - We can **redefine** `Cat::speak()` to make it more specific

`private` members are still `private`

- Derived classes inherit **all members** of the base class; a `cat` has a `name`
- However, this throws a compiler error!

```
std::string Cat::speak() {
    return name + " says meow";
}
```

- The `private` members of the base class are **not accessible**
- This is to prevent breaking encapsulation - if this were allowed, someone else could inherit your class and start messing with the private members

Inheritance: main takeaway

- There is a whole lot more nuance to inheritance and no more time in this class
- There won't be more on the final than maybe a multiple choice question about "what is inheritance" or "why is inheritance useful"
- Inheritance allows us to extend the functionality of a class without having to rewrite all the code
- Next semester you'll be working in Java, which is completely object-oriented and uses inheritance extensively (but some nuance is different from C++)

Coming up next

- **Assignment 4** 🎉 is due on Monday!
- Lab tomorrow: copying
- Last lecture: exam discussion, practice coding on paper, spotting errors, drawing memory diagrams, and any other topics of interest

> *And that's all for new content!*