601.220 Intermediate Programming

Spring 2023, Day 34 (April 17th)

Today's agenda

- Review exercise 33
- Day 34 recap questions
- Final project introduction
- Work on final project

Reminders/Announcements

- Final project due 11 pm on Friday, April 28th
  - No late submissions, please plan accordingly

Exercise 33 review

Add accessor function for member variable `a` to class `A`:

```
int get_a() const { return a; }
```

`B` (and other classes derived from `A`) will need these to get the values of this member variable. (The member variable `d` can be accessed directly because it is `protected` rather than `private`.)

Exercise 33 review

`A::toString` member function:

```
virtual std::string toString() const {
  std::stringstream ss;
  ss << "[Aclass: a = " << a
     << ", d = " << d
     << ", size = " << sizeof(*this)
     << "]";
  return ss.str();
}
```

Exercise 33 review

`B::toString` member function:

```
virtual std::string toString() const override {
  std::stringstream ss;
  ss << "[Bclass: a = " << get_a()
     << ", b = " << b
     << ", d = " << d
     << ", size = " << sizeof(*this)
     << "]";
  return ss.str();
}
```

Because the `a` member variable in the base class `A` is `private`, it's necessary to call a getter function to access its value.

Exercise 33 review

in `main()`, the following statement does not compile:

```
bobj = aobj;
```

With a cast of `bobj` to `A&` (reference to `A`), we can assign to just the `A` part of `bobj`:

```
static_cast<A&>(bobj) = aobj;
```

this is silly

Exercise 33 review

`fun()` pure virtual member function in class `A`:

```
virtual int fun() const = 0;
```

Implementation in class `B`:

```
virtual int fun() const override {
  return int(get_a() * b * d);
}
```

Note that `A` is no longer instantiable, so variable definitions like

```
A aobj(10);
```

are no longer allowed.

Exercise 33 review

```
class C : public A {
private:
  int e;

public:
  C(int val = 0) : e(val) { } // sets a and d to 0 using
                              // A's default ctor
  virtual std::string toString() const override {
    std::stringstream ss;
    ss << "[Cclass: a = " << get_a()
       << ", d = " << d
       << ", e = " << e
       << " size = " << sizeof(*this)
       << "]";
    return ss.str();
  }
```

Exercise 33 review

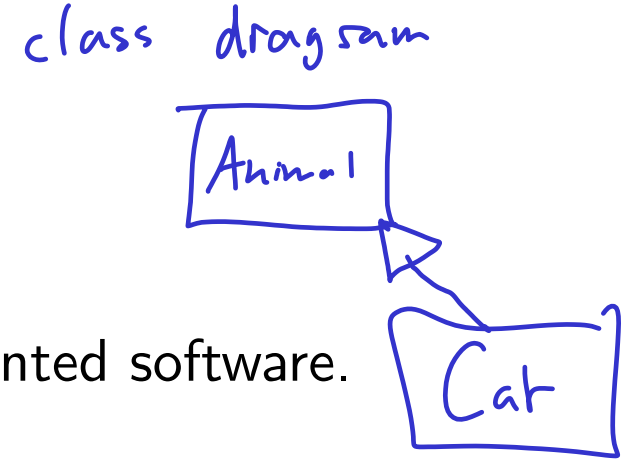
```
    // C class continued

    virtual int fun() const override {
      return int(get_a() * d * e);
    }
};
```

Day 34 recap questions

1. What is UML?
2. What type of class relationship is likely to exist between a class that represents bathroom objects and one that represents apartment objects?
3. What type of class relationship is likely to exist between a class that represents apartment objects and one that represents housing objects?
4. BONUS: which of bathroom, apartment, housing would likely be an abstract class?

1. What is UML?



UML = Unified Modeling Language

It's a visual notation for describing object-oriented software.

It has many types of diagrams, including:

- Class diagrams: show relationships between classes (you will create one for the final project)
- Sequence diagrams: show how objects interact with each other

Opinion: class diagrams are useful as "sketches" to explore possible software designs and to discuss them with team members. When considering design options, a class diagram can quickly give you a sense of what classes are necessary and how they relate to each other.

2. What type of class relationship is likely to exist between a class that represents bathroom objects and one that represents apartment objects?

Aggregation = "Has-A"

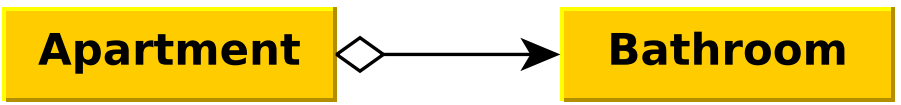


Figure 1: UML aggregation ("has-a") relationship

3. What type of class relationship is likely to exist between a class that represents apartment objects and one that represents housing objects?

Generalization = "Is-A"

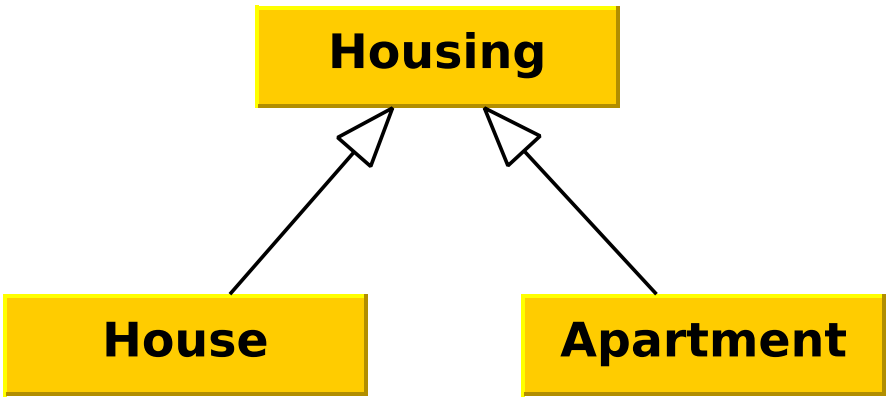


Figure 2: UML generalization ("is-a") relationship

4. BONUS: which of bathroom, apartment, housing would likely be an abstract class?

Housing is an abstract concept. I.e., there is no structure that is "housing" without being some specific *kind* of housing.

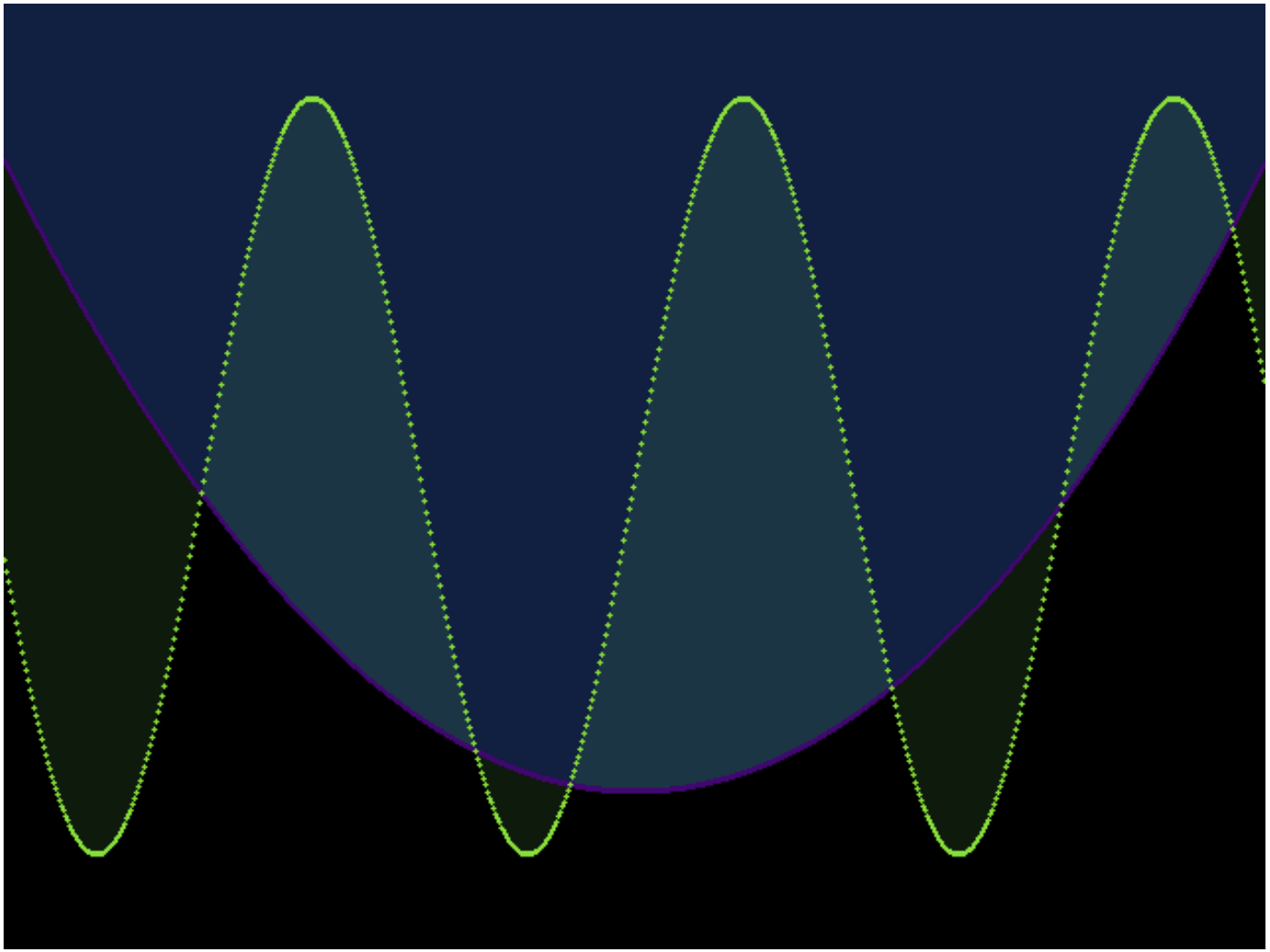
Final project

Plot mathematical functions, render as an image file.

Example plot input file:

```
Plot -4.0 -12.0 4.0 12.0 640 480
Function fn1 ( - ( * x x ) 8.0 )
Function fn2 ( * 9.6 ( sin ( * x 2.3 ) ) )
FillAbove fn1 0.55 31 58 117
FillBetween fn1 fn2 0.12 118 222 108
Color fn2 135 223 57
Color fn1 65 7 113
```

Example plot image



Functions, prefix expressions

All functions are *y* = *expr*, where *expr* is a *prefix expression*

Expression types:

- `x`
- `pi`
- literal floating point value
- (*function arguments*)

Functions are `sin`, `cos`, `+`, `-`, `*`, and `/`.

arguments is a sequence of 0 or more prefix expressions.

Prefix vs. infix

LISP / scheme

Prefix expression:

```
( + ( sin ( * 1.33 x ) ) ( * 0.25 ( cos ( * 6.7 x ) ) ) )
```

Equivalent infix expression:

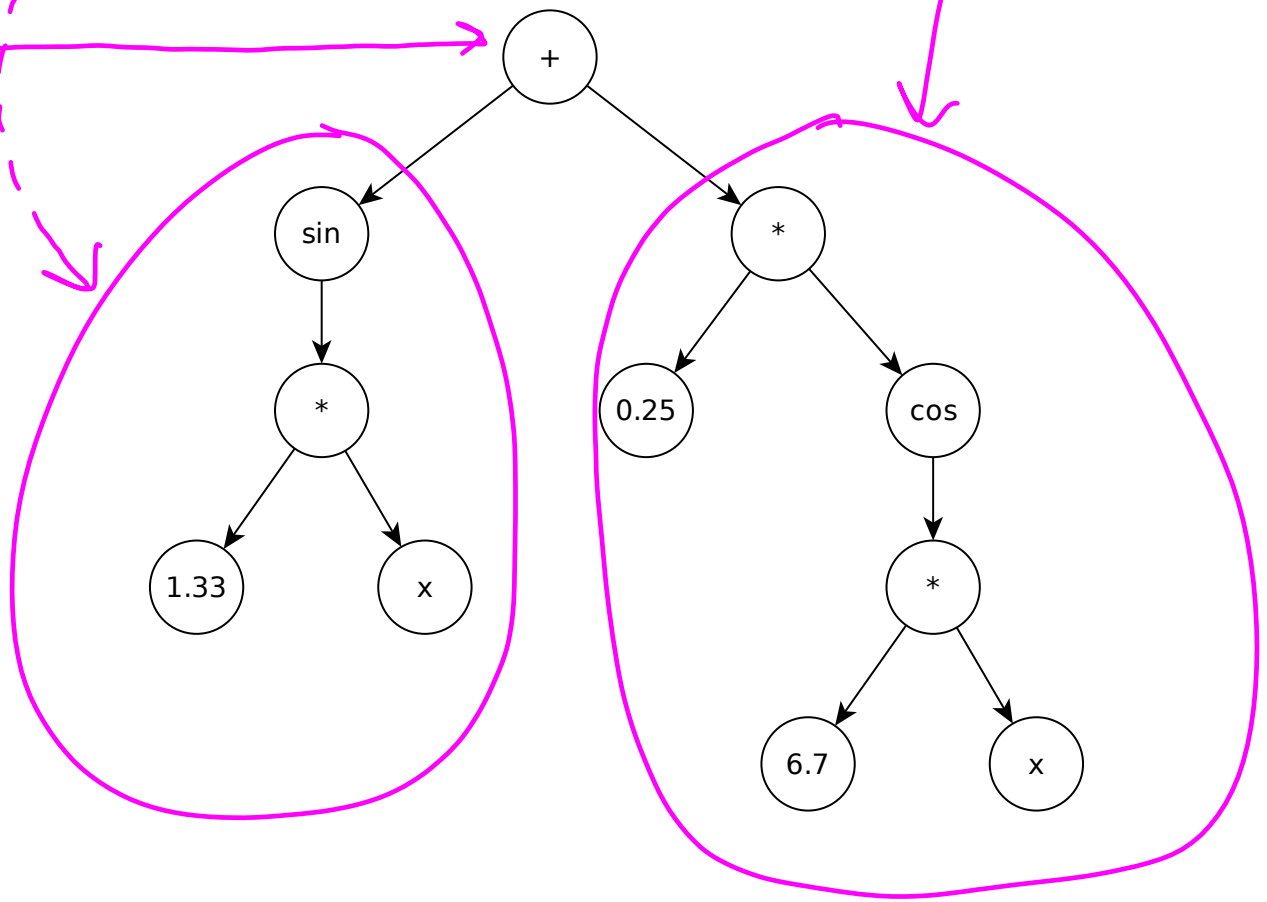
$(\sin 1.33x) + (0.25(\cos 6.7x))$

Expression trees

Expressions can be represented as *trees*. The *leaf nodes* (nodes without children) are `x`, `pi`, or numeric literal. Functions are represented as a *function node* with child expression trees representing the function arguments.

Example expression tree

```
( + ( sin ( * 1.33 x ) ) ( * 0.25 ( cos ( * 6.7 x ) ) ) )
```



Evaluating expressions

The `Expr` base class represents an expression tree node.

It has the following pure virtual member function:

```
virtual double eval(double x) const = 0;
```

For each type of expression tree node, you will create a derived class which overrides the `eval` function with appropriate behavior.

Rendering the plot image

The `Image` class is fairly similar to the `Image` struct type in the midterm project.

The plot image will start with all pixels set to black, RGB $(0, 0, 0)$. For each fill directive and function directive, determine which pixel colors should be changed.

Work on final project

Talk to us if you have questions!

Notes

Notes

Notes

Notes

Notes