601.220 Intermediate Programming

Summer 2022, Meeting 18 (July 20th)

Today's agenda

- Review exercises 31 and 32
- Day 33 recap questions
- Day 34 recap questions
- Final project introduction
- Exercise 33

Reminders/Announcements

- HW7 is due **tomorrow** (Thursday, July 21st)
- Final project teams should be final at this point, and you should have access to your team repository
  - Let us know ASAP if there are any issues

Exercise 31 review

Converting `int_set` to (generic) `my_set<T>`: this is essentially just a syntactic change.

Put `template<typename T>` before the class definition (in `my_set.h`) and member function implementations (in `my_set.inc`.)

Substitutions:

- `int_node` → `my_node<T>`
- `int_node` → `my_node` (names of constructor and destructor functions)
- `int_set` → `my_set<T>`
- `int_set` → `my_set` (names of constructor and destructor functions)
- `int` → `T` (except for `size` field and `get_size()` member function)

Exercise 31 review

The output stream insertion operator needs to use its own type parameter (since it's not a member of `my_set<T>`)*

also necessary for TTrie in HW7

```
// in my_set.h
-template<typename U>
  friend std::ostream& operator<<(std::ostream& os,
                                  const my_set<U> &s);

// in my_set.inc
-template<typename U>
  friend std::ostream& operator<<(std::ostream& os,
                                  const my_set<U> &s) {
    my_node<U> *n = s.head;
    // ...code to print member values...
    return os;
  }
```

Exercise 31 review

Assignment operator

```
// in my_set.h
  my_set<T>& operator=(const my_set<T>& other);

// in my_set.inc
  template<typename T>
  my_set<T>& my_set<T>::operator=(const my_set<T>& other) {
    if (this != &other) {
      my_node<T> *n = other.head;
      while (n != nullptr) {
        add(n->get_data();
        n = n->get_next();
      }
    }
    return *this;
  }
```

Exercise 32 review

In `B`'s constructor:

```
a = 27;
```

The member variable `a` is `private` in the base class `A`, so `B`'s constructor can't access it directly.

Exercise 32 review

A *aptr;

In `main1.cpp`:

```
aobj.d = 17.5;   // d is protected in A, main cannot
                 // access directly

aptr->setb(15); // even though aptr is pointing to an object
                 // of type B, its type is A* (pointer to A)
                 // and A does not have a setb member function

bobj = a5;       // B's assignment operator requires an
                 // argument of type const B& (reference to
                 // const B), but a5's type is A
```

Exercise 32 review

After making the `show` member function `virtual` in `A` (note that additional output is generated):

```
--- orig_output.txt 2022-07-12 09:53:44.533488783 -0400
+++ revised_output.txt  2022-07-12 09:53:44.533488783 -0400
@@ -6,10 +6,14 @@

 A is 3
 test A
+B is 2
+test B

 non-virtual display A
 A is 3
 test A
+B is 2
+test B

 A obj killed
 A is 10
```

Exercise 32 review

`A`'s `show()` member function:

```
virtual void show() { std::cout << "A is " << a << std::endl; test(); }
```

`A`'s `display()` member function:

```
void show() { A::show(); std::cout << "B is " << b << std::endl; test(); }
```

display

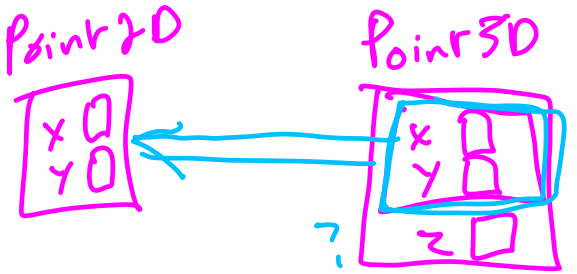
The call to `show()` in `A::display()` now resolves to `B::show()` when called on a `B` object. `B::show()` calls `A::show()` directly, then prints additional output ("B is _"), then calls `test()`, which is a non-virtual member function in `A`, but resolves to `B::test()`, because `B` also defines a member function of the same name, and the call is in `B::show()`.

Day 33 recap questions

1. Explain what object slicing is in C++.
2. What is the `override` specifier in C++?
3. Explain what function hiding is in C++?
4. In C++, how do you make an abstract class?
5. Can we create an object from an abstract class?

1. Explain what object slicing is in C++.

An assignment of a derived class object to a base class object will "slice off" the fields of the derived class object, because they can't be stored in the base class object.



E.g.:

```
// assume Point3D derives from Point2D
Point2D p1(3.0, 4.0);
Point3D p2(5.0, 6.0, 7.0);

p1 = p2; // only the x and y values of p2 are
         // copied to p1
```

Value semantics and inheritance

Value semantics (copying and assigning object contents) tends not to be particularly useful in class hierarchies (where inheritance is used to define "is-a" relationships between classes.)

It is not uncommon for base and derived classes to prohibit the use of the copy constructor and assignment operator by making them private.

2. What is the `override` specifier in C++?

The `override` specifier can be used in a derived class to indicate that a member function is intended to override a member function in the base class. If the derived class function (marked with `override`) does not *actually* override a base class function, the compiler reports an error.

Why `override` is (sometimes) useful

The problem `override` is designed to solve:

- base class defines a `virtual` member function
- derived class defines a `virtual` member function intended to override the base class function, but it messes up somehow (e.g., wrong number or type(s) of parameters), so that the derived class function doesn't actually override the base class function

One reason this could happen is because someone changes the definition of the member function in the base class.

Opinion about `override`

Virtual member functions in base classes should be pure virtual (i.e., abstract.)

If they are, then the `override` specifier isn't that essential, because if a derived class doesn't override all of the pure virtual member functions in the base class, it won't be instantiable.

3. Explain what function hiding is in C++?

Function hiding occurs when a derived calss defines a member function with the same name as member function(s) in the base class.

This "hides" the identically-named functions in the base class.

Note that those functions could still be called using the scope resolution operator (`::`). E.g.:

```
Base::foobar(123, 'a'); // if function(s) called "foobar"
                        // in Base would normally be hidden,
                        // this would call call
                        // Base::foobar(int, char)
```

4. In C++, how do you make an abstract class?

A class that has at least one pure `virtual` member function is an abstract class.

E.g.:

```
class Animal {
  virtual ~Animal() {}
  virtual void vocalize() = 0;
};
```

Opinion: `virtual` functions in base classes should always be pure virtual. The reason is that if derived classes will be overriding the function to implement varying behavior, then there is probably nothing useful that the base class can do to define functionality for the member function.

5. Can we create an object from an abstract class?

No. Example:

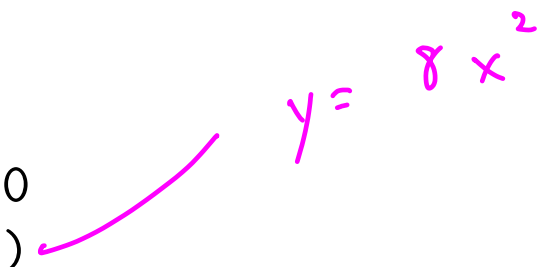
```
class Animal {
public:
  virtual ~Animal() { }
  virtual void vocalize() = 0;
};

class Dog : public Animal {
  virtual ~Dog() { }
  virtual void vocalize() { std::cout << "woof\n"; }
};

// ...

Animal *a = new Animal(); // compile error
Dog *d = new Dog();       // fine
```

* see notes at end for answers to day 34 recap questions

Final project

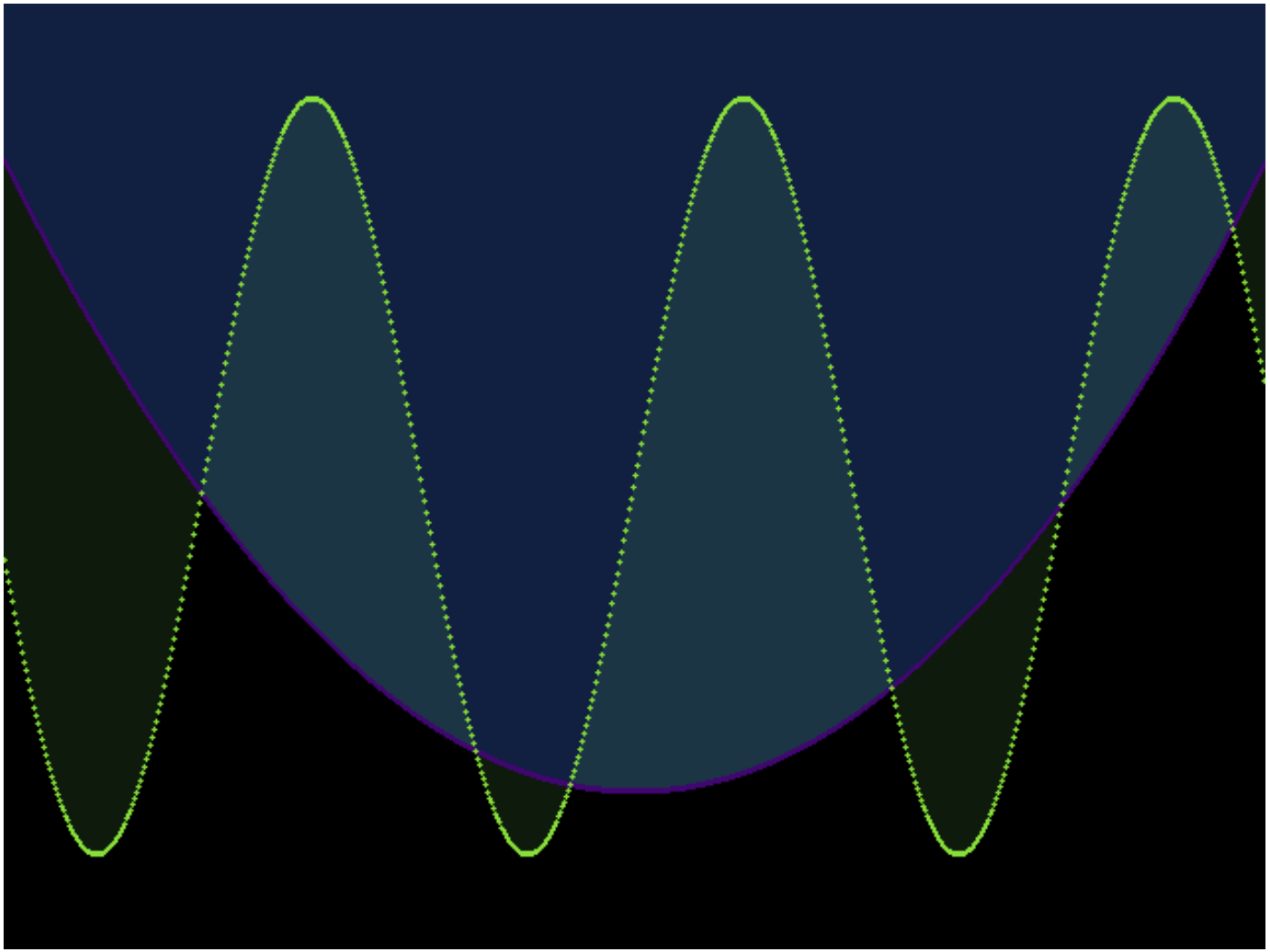
Plot mathematical functions, render as an image file.

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*)

"leaf" expression

expressions

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

arguments is a sequence of 0 or more prefix expressions.

Prefix vs. infix

Prefix expression:

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

Equivalent infix expression:

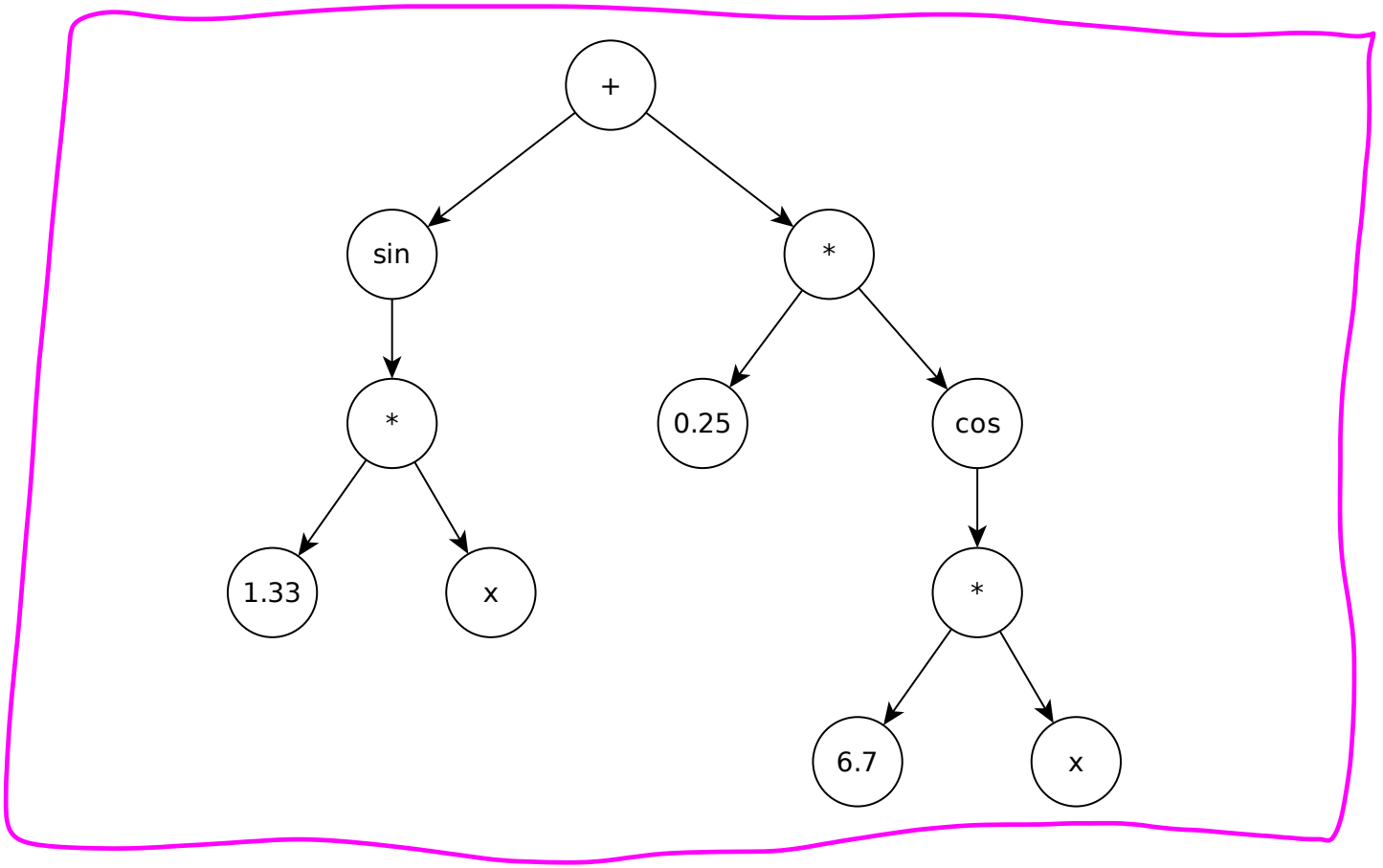
$(\sin 1.33x) + (0.25(\cos 6.7x))$

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.

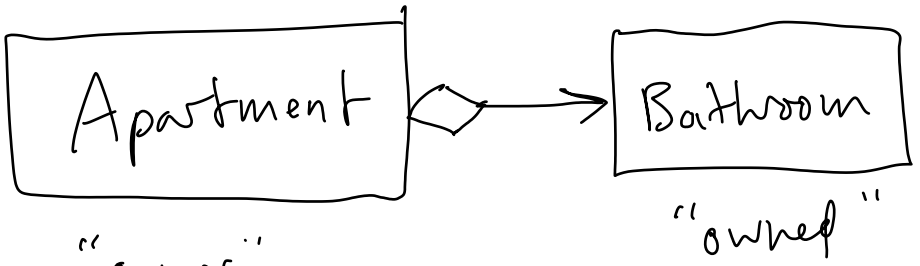
Exercise 33

- Abstract classes, pure virtual member functions
- Breakout rooms 1–10 are “social”
- Use Slack to let us know if you have a question!

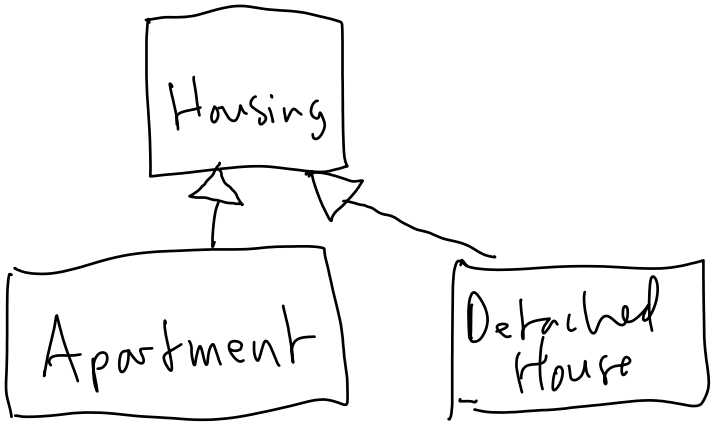
Notes

Day 34 recap questions

2. "Has-A", "aggregation"



3. "Is-A", "generalization", "inheritance"



4. Housing

Notes

Notes

Notes

Notes