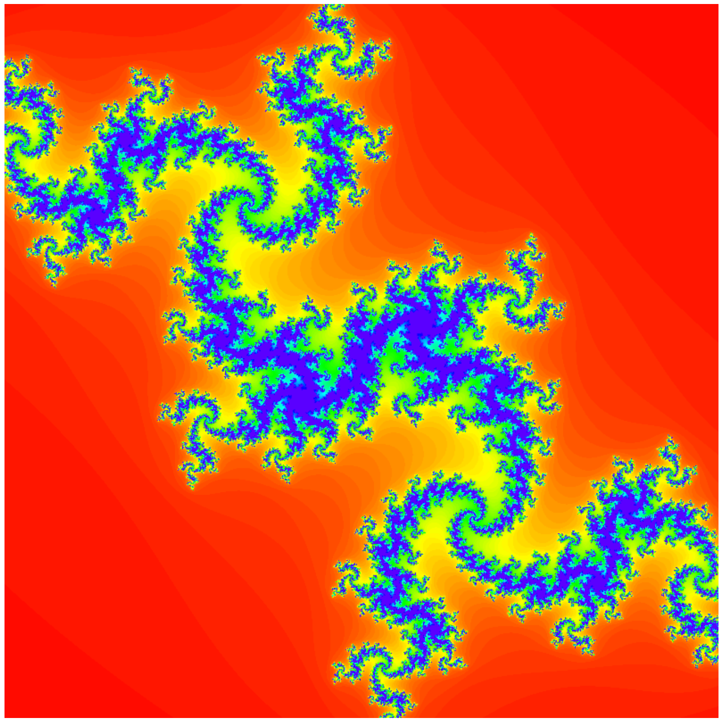
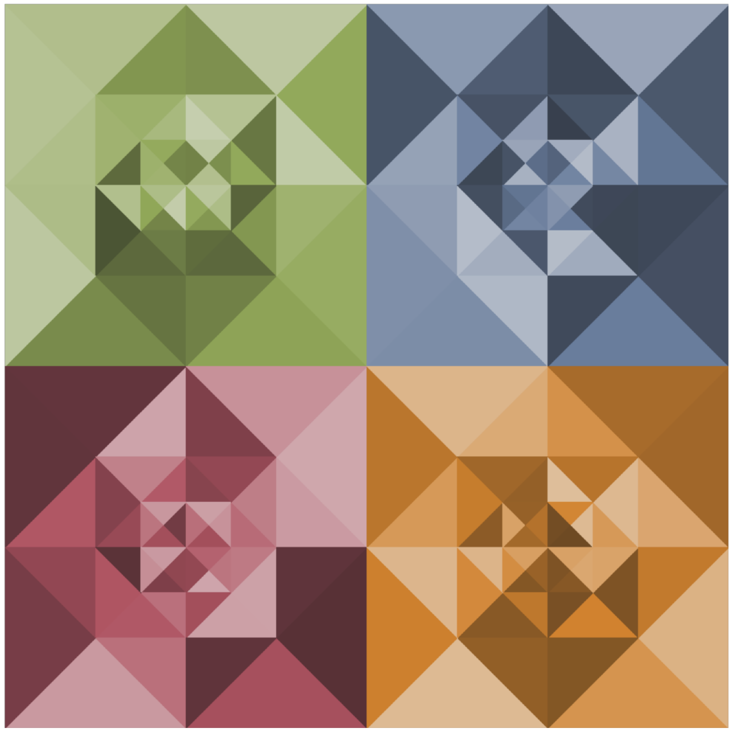



# Data Structures

## *February 1 – Templates*

*Wade Fagen-Ulmschneider, Craig Zilles*

## Example:

assignmentOpSelf.cpp

```cpp
#include "Cube.h"
…
Cube& Cube::operator=(const Cube &other) {

  _destroy();
  _copy(other);

  return *this;
}
```

## Example:

assignmentOpSelf.cpp

```cpp
#include "Cube.h"

int main() {
  cs225::Cube c(10);
  c = c;
  return 0;
}
```

**Square.h**

```
#pragma once

#include "Shape.h"

class Square : public Shape {
  public:
    double getArea() const;

  private:
    // Nothing!
};
```

**Shape.h**

```
class Shape {
  public:
    Shape();
    Shape(double length);
    double getLength() const;

  private:
    double length_;
};
```

**Square.cpp**

```
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
…
```

## Derived Classes

**[Public Members of the Base Class]:**

**main.cpp**

```
int main() {
  Square sq;
  sq.getLength(); // Returns 1, the length init'd
                  // by Shape's default ctor
  ...
}
```

**[Private Members of the Base Class]:**

## Polymorphism

*Object-Orientated Programming (OOP) concept that a single object may take on the type of any of its base types.*

## Virtual

Cube.cpp

```
Cube::print_1() {
  cout << "Cube" << endl;
}

Cube::print_2() {
  cout << "Cube" << endl;
}

virtual Cube::print_3() {
  cout << "Cube" << endl;
}

virtual Cube::print_4() {
  cout << "Cube" << endl;
}

// In .h file:
virtual print_5() = 0;
```

RubikCube.cpp

```
// No print_1() in RubikCube.cpp



RubikCube::print_2() {
  cout << "Rubik" << endl;
}

// No print_3() in RubikCube.cpp



RubikCube::print_4() {
  cout << "Rubik" << endl;
}

RubikCube::print_5() {
  cout << "Rubik" << endl;
}
```

# Runtime of Virtual Functions

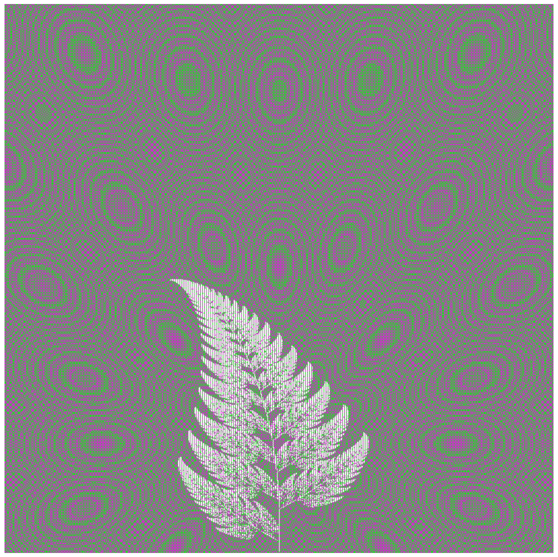
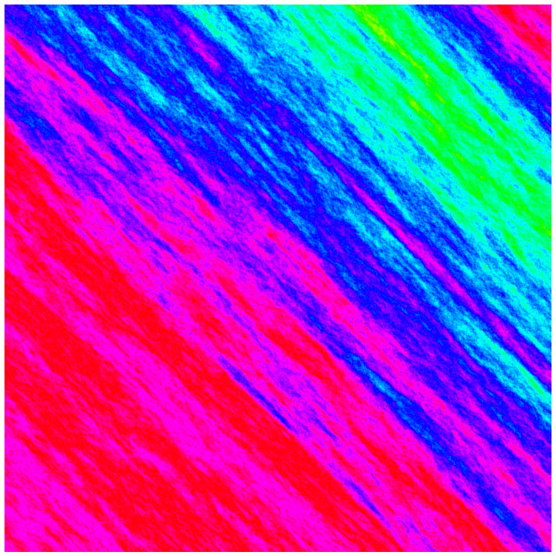
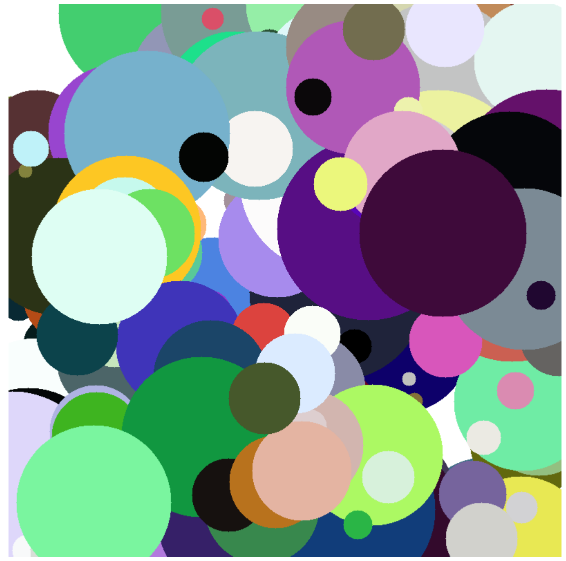
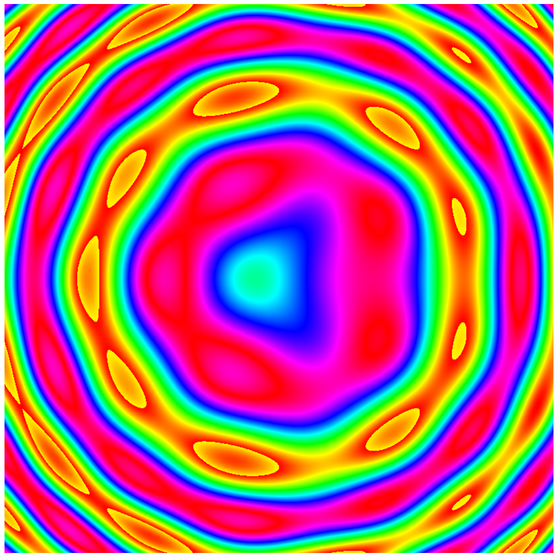
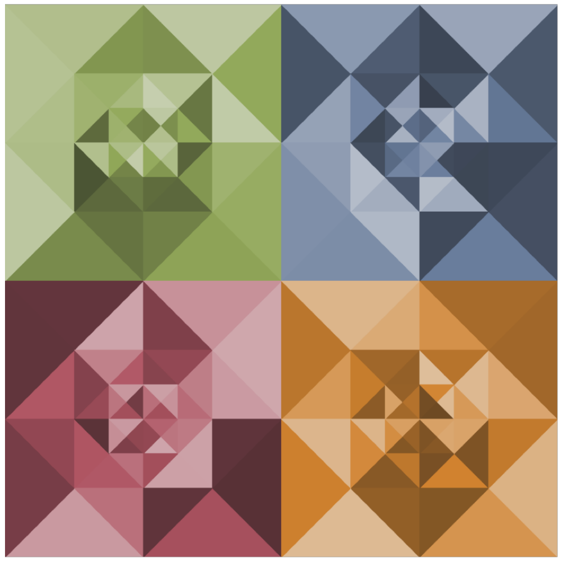
| virtual-main.cpp | `Cube c;` | `RubikCube c;` | `RubikCube rc;` `Cube &c = rc;` |
|---|---|---|---|
| `c.print_1();` | | | |
| `c.print_2();` | | | |
| `c.print_3();` | | | |
| `c.print_4();` | | | |
| `c.print_5();` | | | |

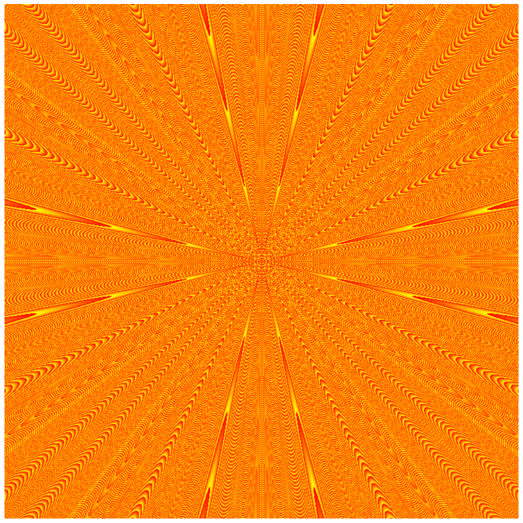
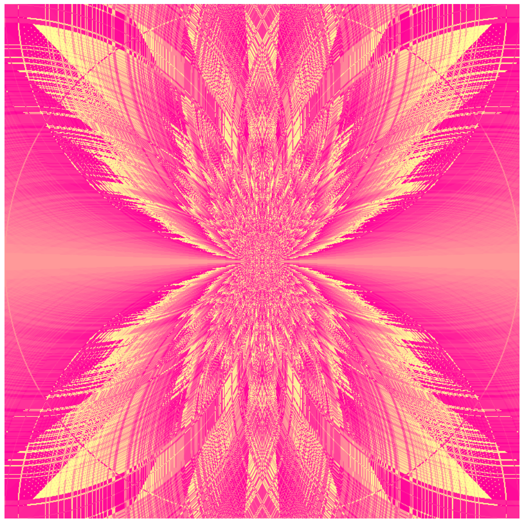
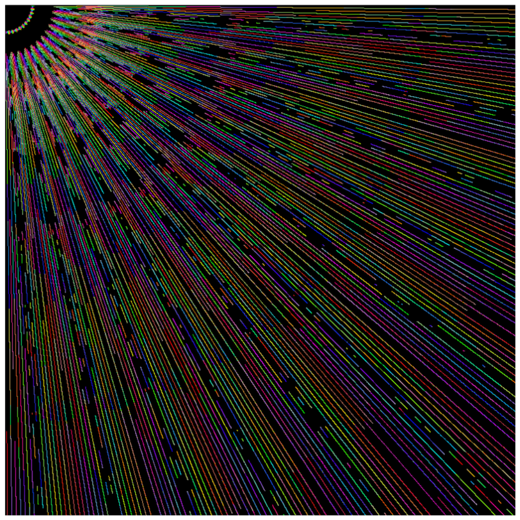
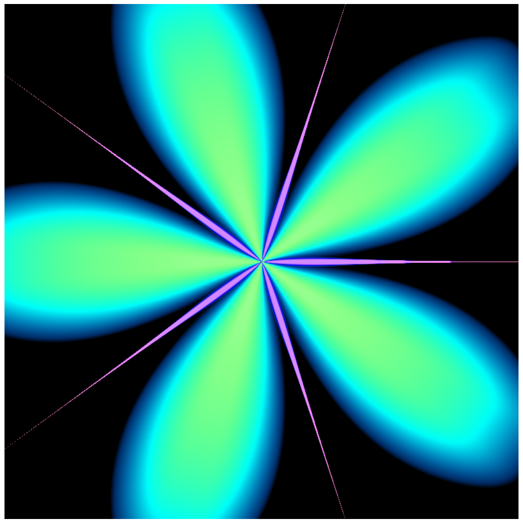
## Why Polymorphism?

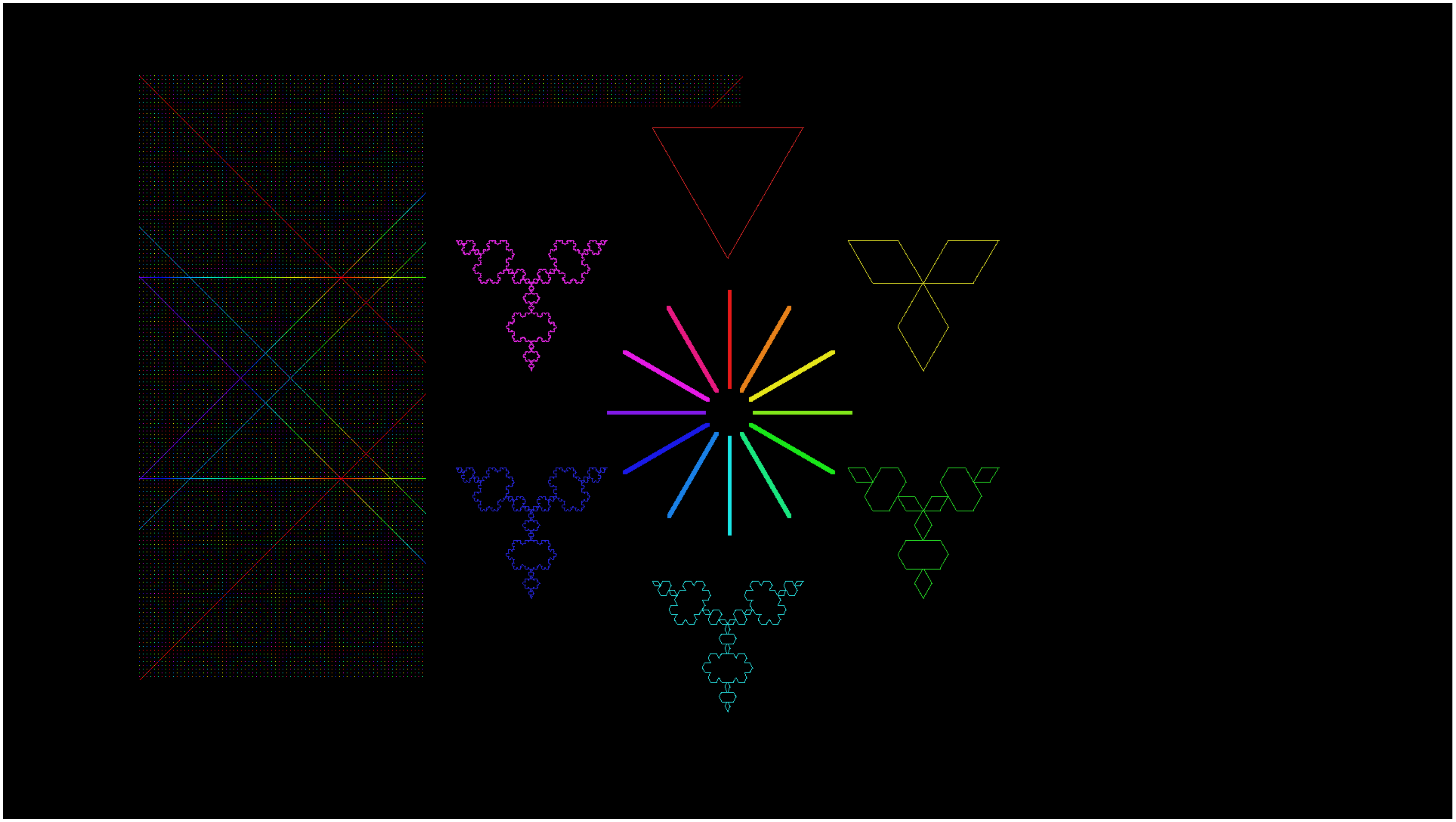
## animalShelter.cpp

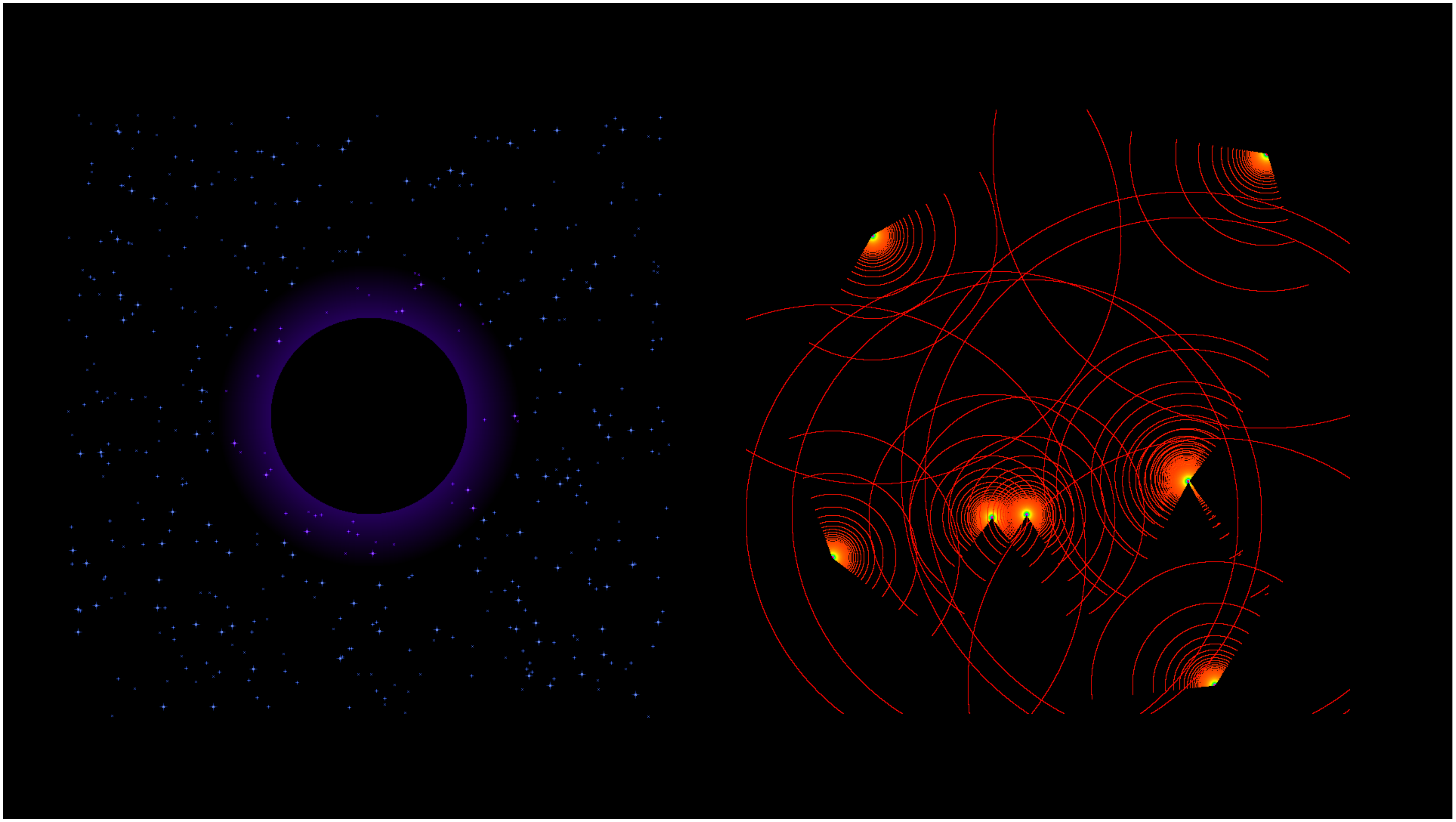
```cpp
class Animal {
  public:
    void speak() {                                    }
};

class Dog : public Animal {
  public:
     void speak() {                                    }
};

class Cat : public Animal {
  public:
     void speak() {                                    }
};
```

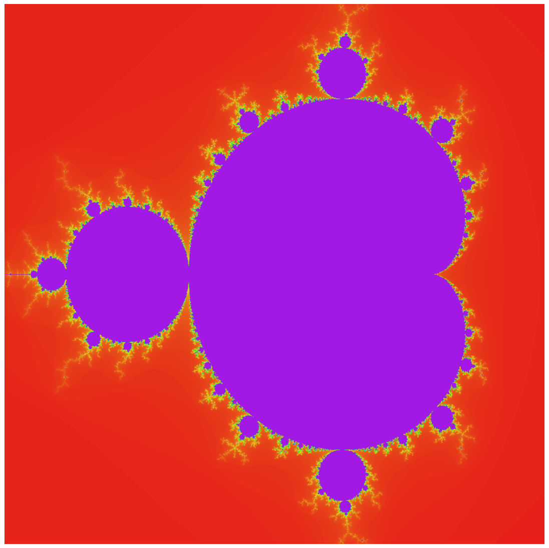
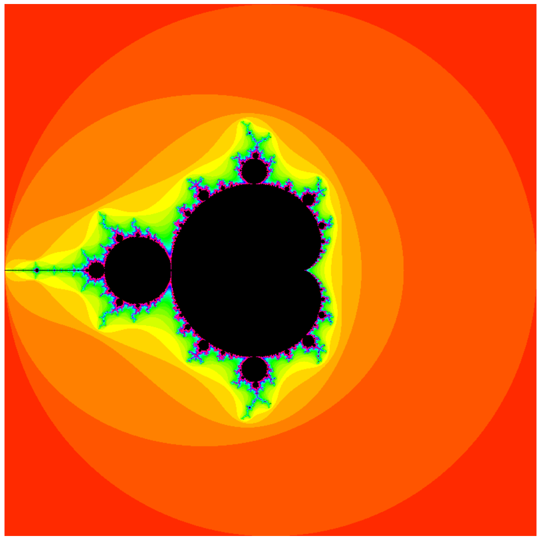
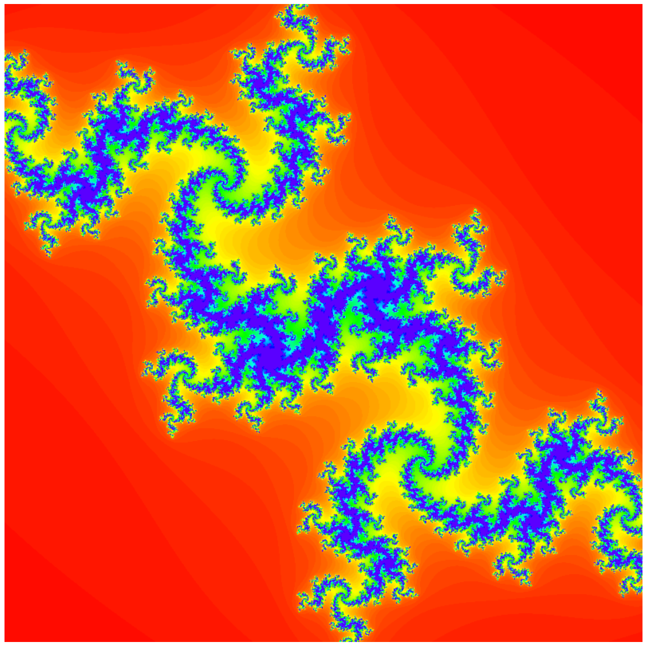
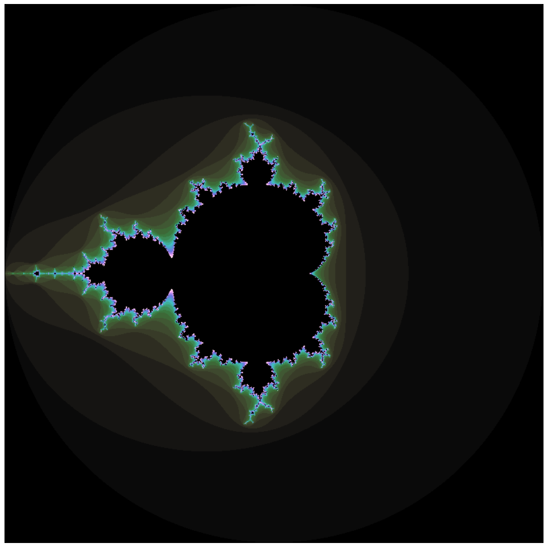
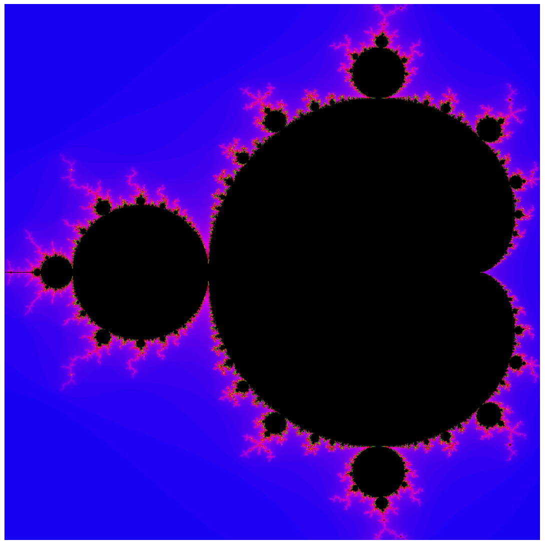
## MP1 Artwork

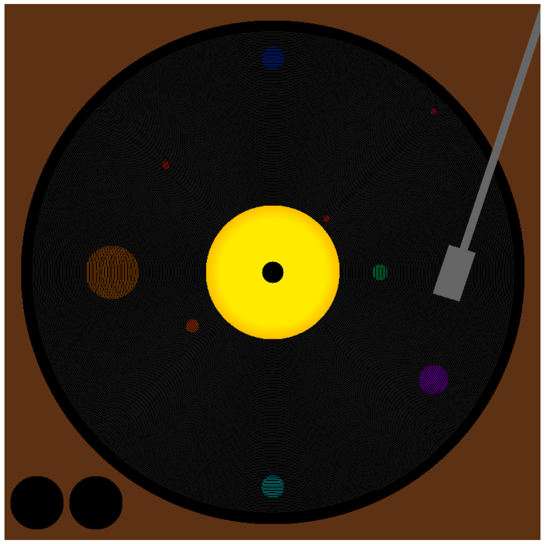






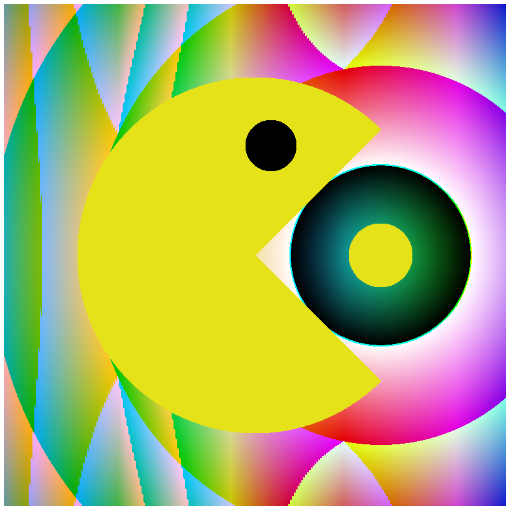
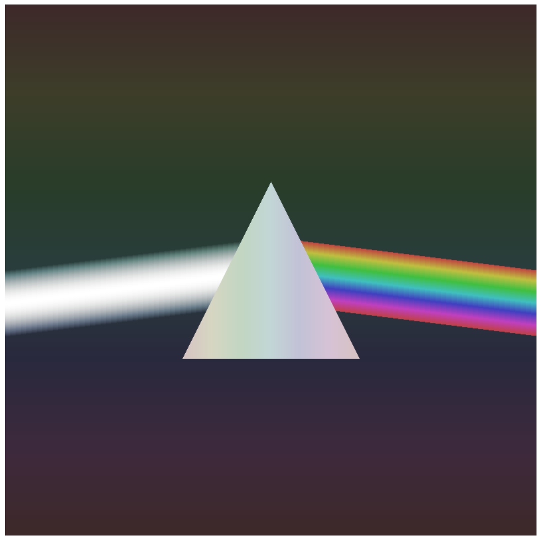
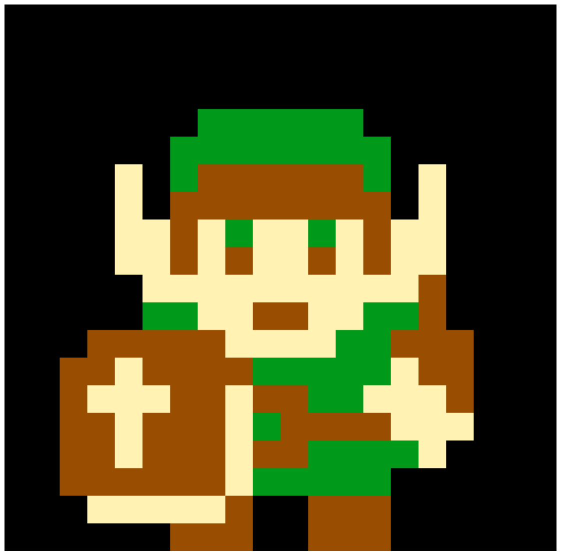
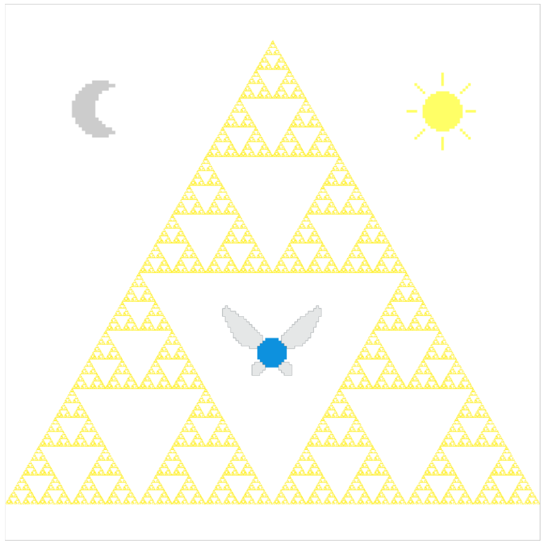





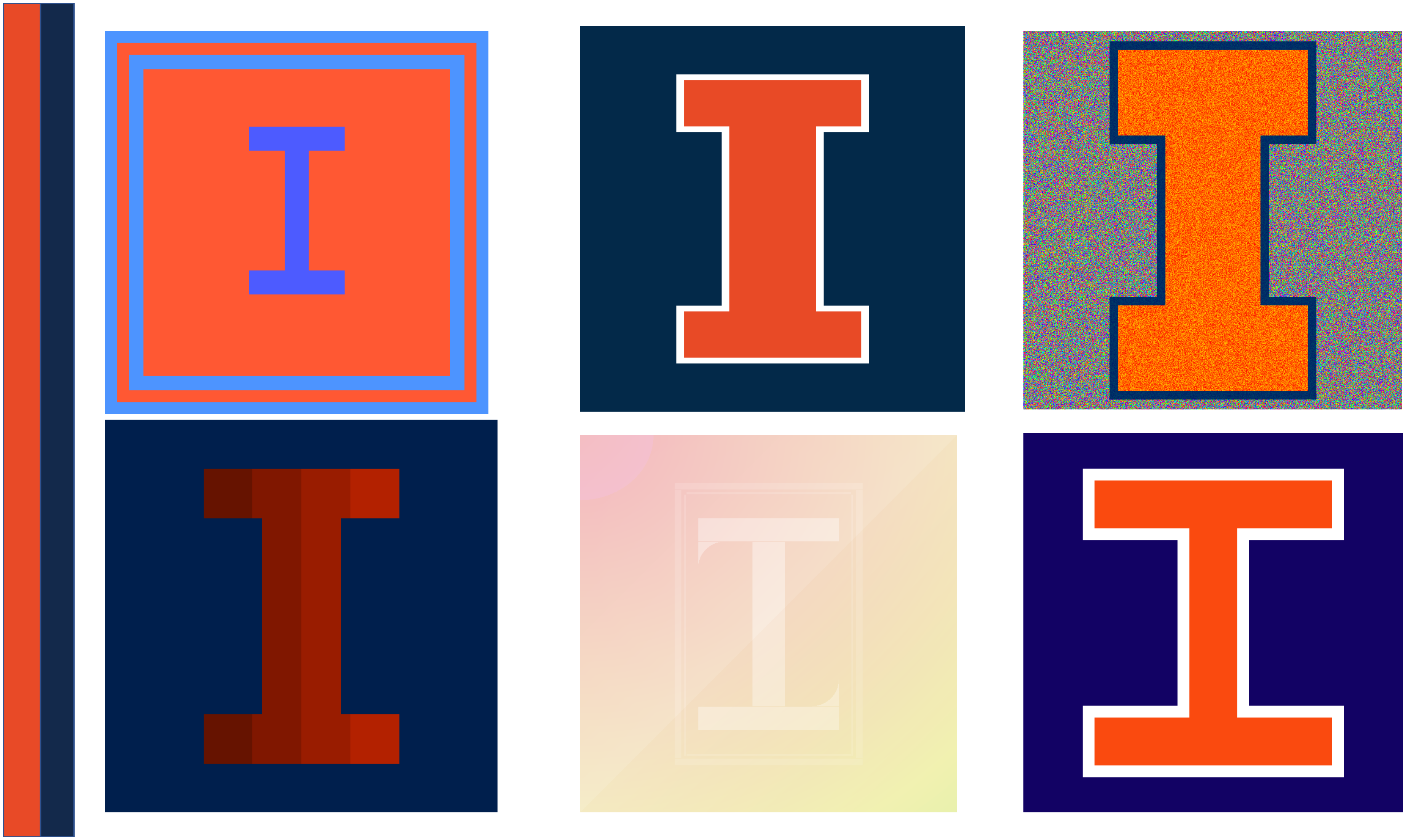







I
I
I
I
I
I

## Abstract Class:

**[Requirement]:**

**[Syntax]:**

**[As a result]:**

## virtual-dtor.cpp

```cpp
class Cube {
  public:
    ~Cube();
};

class RubikCube : public Cube {
  public:
    ~RubikCube();
};
```

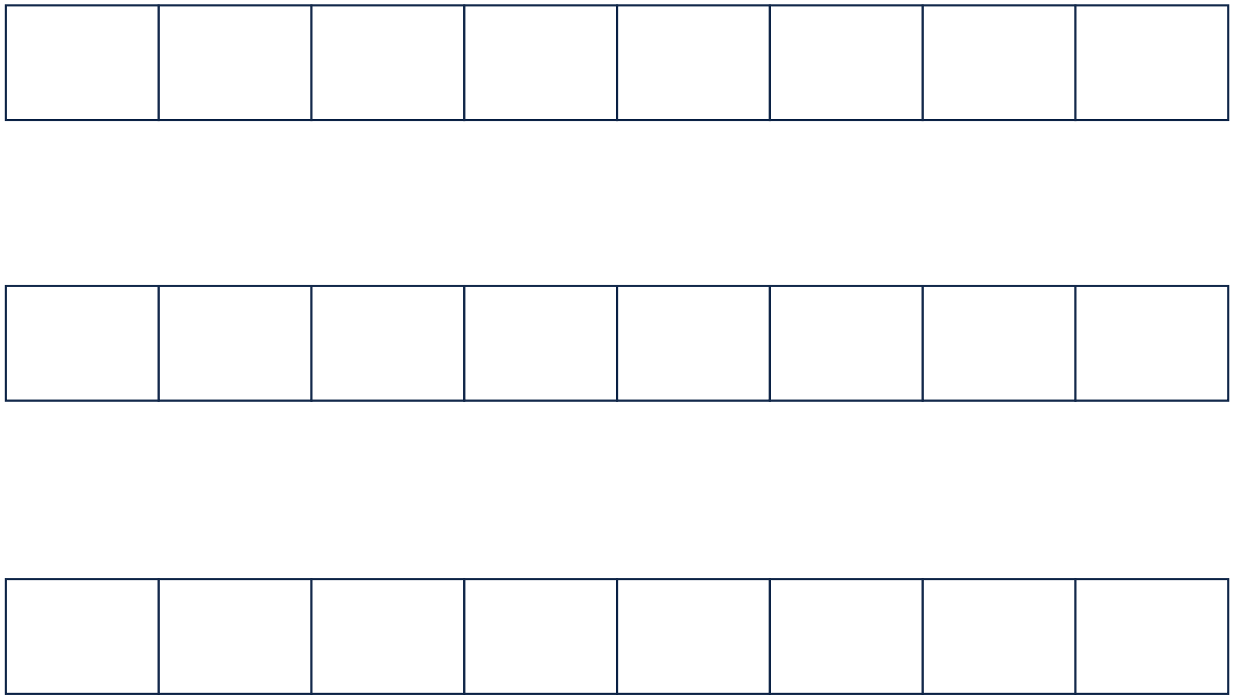
## MP2: cs225/PNG.h

```
class PNG {
  public:
    PNG();
    PNG(unsigned int width, unsigned int height);
    PNG(PNG const & other);
    ~PNG();

    PNG & operator= (PNG const & other);
    bool operator== (PNG const & other) const;

    bool readFromFile(string const & fileName);
    bool writeToFile(string const & fileName);
    HSLAPixel & getPixel(unsigned int x, unsigned int y) const;
    unsigned int width() const;
    // ...

  private:
    unsigned int width_;
    unsigned int height_;
    HSLAPixel *imageData_;
    void _copy(PNG const & other);
};
```

## Abstract Data Type

## List ADT

## What types of “stuff” do we want in our list?



## Templates

## template1.cpp

```


T maximum(T a, T b) {
  T result;
  result = (a > b) ? a : b;
  return result;
}
```

**List.h**

```
#pragma once



class List {
  public:








  private:



};

#endif

```

**List.cpp**