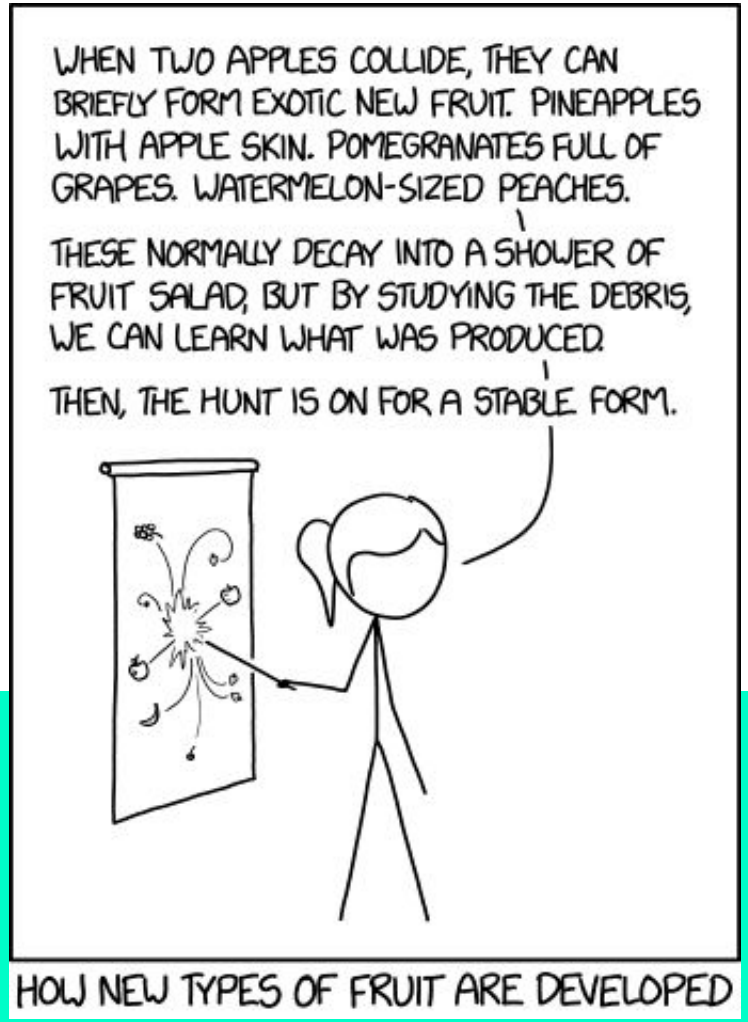


HOW NEW TYPES OF FRUIT ARE DEVELOPED

Lab 5 - C++ Style ADTs

ReadMe

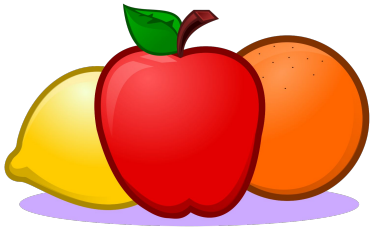
Reminders

- Lab 5 due Sunday, October 14th at 8pm
- Project 3 due Friday, October 26th at 8pm
- NO LAB NEXT WEEK -- Have a great break!

Agenda

- Brief review Polymorphism + Inheritance
- Worksheet
- Exam-style Questions
- Time for Lab Assignment / Project Questions

Today's theme:
Fruit!



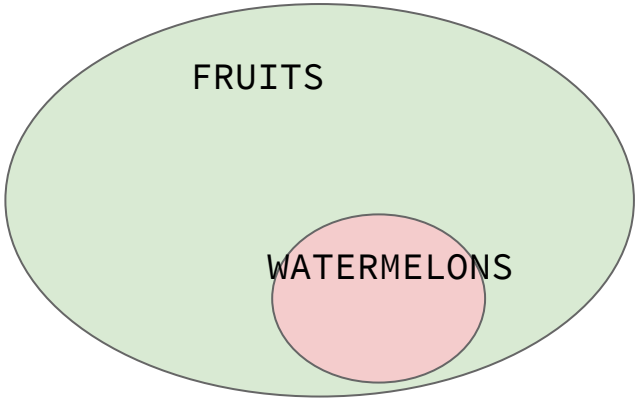
If you want to follow along with today's example:

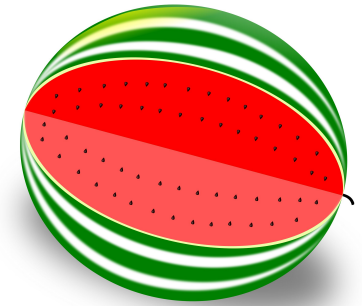
git clone https://github.com/MelGeorge/Fruit.git

Create a project with these files in your IDE, or simply view them in a text editor!

Inheritance

- Inheritance is when we have base classes and derived classes. A base class is usually a broad category, whereas a derived class is usually a specific subtype within that category.
- For example, we might have a base class called “Fruit” (a large category) and a derived class called “Watermelon” (a type of fruit). A Watermelon “is a” fruit.



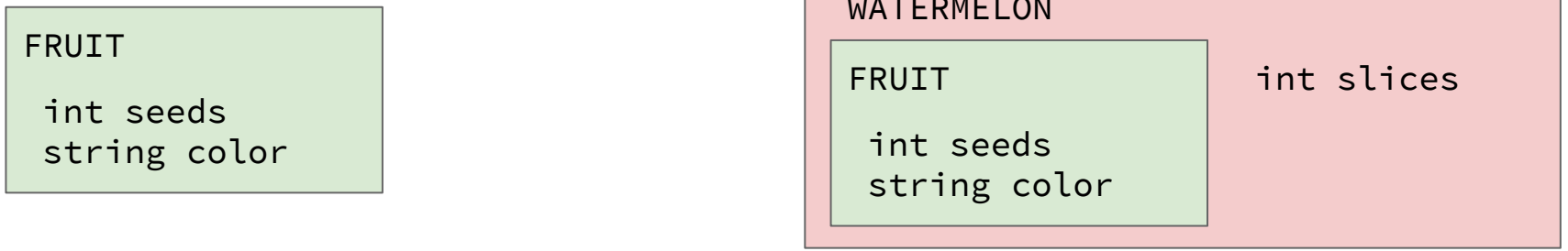


Inheritance - Fruits & Watermelons Example

```
class Fruit {
public:
    Fruit();
    Fruit(int seeds_in, std::string color_in);
    virtual void eat();
    int get_num_seeds();
    void set_num_seeds(int num);
    std::string get_color();
    std::string set_color(std::string);

private:
    int seeds;
    std::string color;
};
```

```
// Watermelon inherits from Fruit
class Watermelon : public Fruit {
public:
    Watermelon();
    virtual void eat() override;
    void slice(int num_slices);
    int get_num_slices();

private:
    int slices;
    // Since Watermelon inherits from Fruit,
    // it secretly has these here too:
    // int seeds;
    // std::string color;
};
```

In memory, these objects look like this:



Inheritance

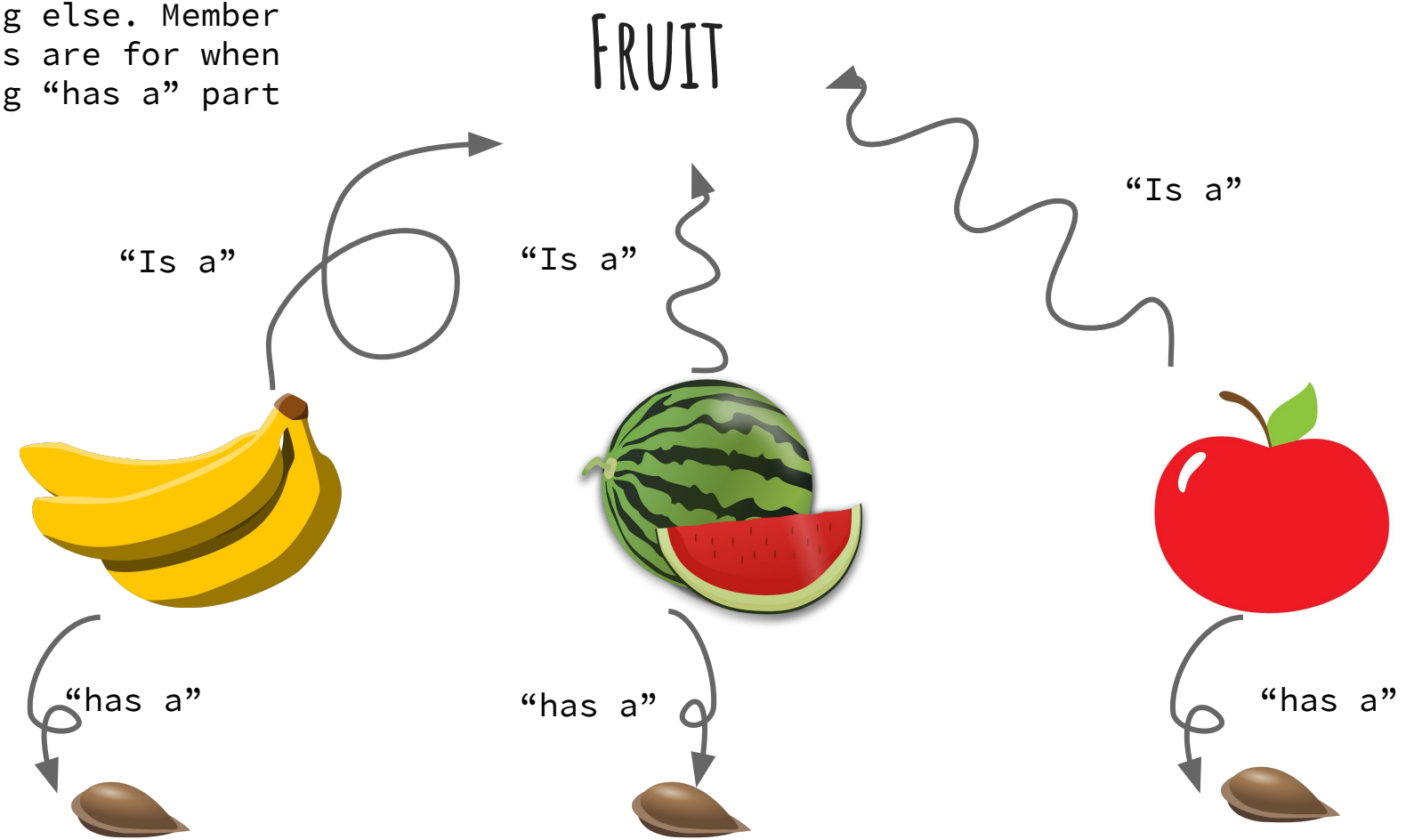
```cpp
class Fruit {
public:
    Fruit();
    Fruit(int seeds_in, std::string color_in);
    virtual void eat();
    int get_num_seeds();
    void set_num_seeds(int num);
    std::string get_color();
    std::string set_color(std::string);

private:
    int seeds;
    std::string color;
};
```

```cpp
// Watermelon inherits from Fruit
class Watermelon : public Fruit {
public:
    Watermelon();
    virtual void eat() override;
    void slice(int num_slices);
    int get_num_slices();

private:
    int slices;
    // Since Watermelon inherits from Fruit,
    // it secretly has these here too:
    // int seeds;
    // std::string color;
};
```

```cpp
// Make a watermelon
Watermelon w1;

// Watermelons can call Fruit functions like get
// color, set color, and get_num_seeds
std::cout << w1.get_color() << std::endl;
w1.set_color("purple");
std::cout << w1.get_num_seeds() << std::endl;

// But they can also call Watermelon functions
// like get_num_slices
std::cout << w1.get_num_slices() << std::endl;

// They can even call functions defined in BOTH
// Fruit and Watermelon
w1.eat();
```

Inheritance is BOTTOM UP-- if both classes have defined the same function, look in the derived class first.

Inheritance is for when something “is a” type of something else. Member variables are for when something “has a” part



Polymorphism

- Basically, when objects can behave in many different ways, depending on the conditions.
- 3 types:
  - Ad-Hoc Polymorphism / Function Overloading : many functions with the SAME NAME but different SIGNATURES
  - Subtype Polymorphism : a variable of a base type can hold an object of a derived type
    - Upcasts are safe, downcasts are not -- use the “is a” construct to check!
    - Static vs. Dynamic Type and Static vs. Dynamic Binding
  - Parametric Polymorphism : templating! Later <3

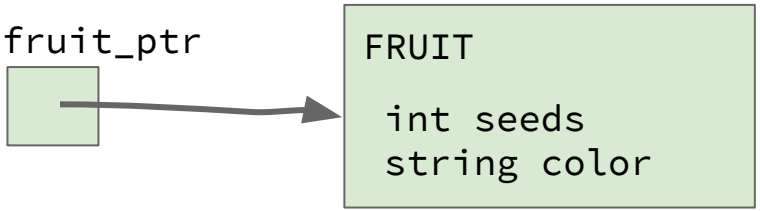
Polymorphism

```
Watermelon watermelon;
Fruit fruit;
Fruit * fruit_ptr = &fruit;
fruit_ptr->eat();
fruit_ptr = &watermelon;
fruit_ptr->eat();
```

Polymorphism comes into play when we have pointers to objects. When we access those objects and call their member functions through the pointers, we will see polymorphic behavior.

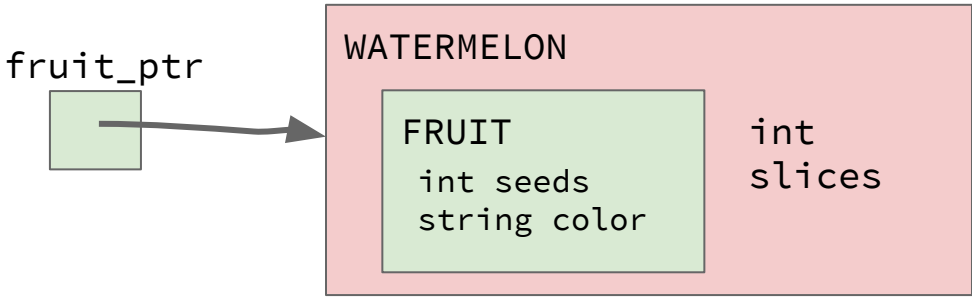
A Fruit pointer can point to a Watermelon, because all watermelons are fruits.

Fruit ptr will “act more like a fruit”



Polymorphism is more TOP DOWN-- look at static type (base class) first

Fruit ptr will “act more like a watermelon”



Static Type vs. Dynamic Type

- Static type is the type of an object at the beginning / throughout the program.
- Dynamic type is the type of an object during run time.
  - Comes into play when we have virtual functions.

What are the static and dynamic types of fruit_ptr in these two code snippets?

```
Watermelon watermelon;
Fruit fruit;
Fruit * fruit_ptr = &watermelon;
```

```
Watermelon watermelon;
Fruit fruit;
Fruit * fruit_ptr = &fruit;
```

```
Watermelon watermelon;
Fruit fruit;
Watermelon * watermelon_ptr = &fruit;
```

This won't work. This is a downcast. They're not safe and they won't compile.
Why do you think this is?

Polymorphism - which function do I choose?

Decide which function to call by doing the following:

1. Evaluate static and dynamic types
2. Look at static type's functions. If the static type has a match, call it. Unless it says virtual - go to step 3
3. Look at the object's dynamic type. If there is a function that matches the signature we're looking for exactly, call that.

Code:

```
Watermelon watermelon;
Fruit fruit;
Fruit * fruit_ptr = &fruit;
fruit_ptr->eat();
fruit_ptr = &watermelon;
fruit_ptr->eat();
```

Result:

```
nom nom
slicing watermelon
slicing watermelon into 10 pieces
eating watermelon
spitting out seeds
Program ended with exit code: 0
```

What happens?

```cpp
int main(int argc, const char * argv[]) {

    Watermelon watermelon;
    Fruit fruit;
    Fruit * fruit_ptr = &fruit;
    fruit_ptr->eat();
    fruit_ptr = &watermelon;
    fruit_ptr->eat();

    watermelon.eat();
    fruit.eat();

    std::cout << watermelon.get_color() << std::endl;
    std::cout << fruit_ptr->get_color() << std::endl;

    return 0;
}
```

```cpp
class Fruit {
public:
    Fruit();
    Fruit(int seeds_in, std::string color_in);
    virtual void eat();
    int get_num_seeds();
    void set_num_seeds(int num);
    std::string get_color();
    std::string set_color(std::string);

private:
    int seeds;
    std::string color;
};
```

```cpp
// Watermelon inherits from Fruit
class Watermelon : public Fruit {
public:
    Watermelon();
    virtual void eat() override;
    void slice(int num_slices);
    int get_num_slices();

private:
    int slices;
    // Since Watermelon inherits from Fruit,
    // it secretly has these here too:
    // int seeds;
    // std::string color;
};
```

When is polymorphism useful?

```
class Fruit {
    …
}
```

```
class Banana : public Fruit
{
    …
}
```

```
class Watermelon : public Fruit
{
    …
}
```

```
class Apple: public Fruit
{
    …
}
```

```
Banana banana1();
Banana banana2();
Watermelon watermelon1();
Apple apple1();

Fruit* groceries[4] = {&banana1, &banana2, &watermelon1, &apple1};

for(int i = 0; i < 4; ++i) {
    groceries[i]->eat();
    //we automatically call the right “eat” function!
}
```

Overriding vs. Overloading

- Function OVERLOADING is when we have several functions that may have the same name, but different SIGNATURES

When I call this code with different argument types, different constructors will run!

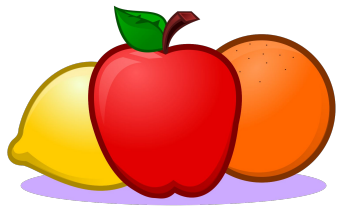
```
Fruit::Fruit(int seeds_in, std::string color_in)
    : seeds(seeds_in), color(color_in) {}

Fruit::Fruit() : seeds(0), color("green") {}
```

What happens when I run ...

```
Fruit grapefruit(10, "pink");
```

What about when I run ...

```
Fruit grapefruit;
```



Overriding vs. Overloading

- Function OVERRIDING is when a derived class has a function with the same signature as a virtual function in the base class.

```
class Fruit {
public:
    Fruit();
    Fruit(int seeds_in, std::string color_in);
    virtual void eat();
    int get_num_seeds();
    void set_num_seeds(int num);
    std::string get_color();
    std::string set_color(std::string);

private:
    int seeds;
    std::string color;
};
```

```
void Fruit::eat() {
    std::cout << "nom nom" << std::endl;
}
```

```
// Watermelon inherits from Fruit
class Watermelon : public Fruit {
public:
    Watermelon();
    virtual void eat() override;
    void slice(int num_slices);
    int get_num_slices();

private:
    int slices;
    // Since Watermelon inherits from Fruit,
    // it secretly has these here too:
    // int seeds;
    // std::string color;
};
```

```
void Watermelon::eat() {
    std::cout << "slicing watermelon\n";
    slice(10);
    std::cout << "eating watermelon\n";
    std::cout << "spitting out seeds\n";
    set_num_seeds(get_num_seeds() - 2);
}
```

Overriding & the override keyword

- The override keyword is used when we are writing a function for a derived class that we want to OVERRIDE a function in the base class.
- We use the override keyword because we want the compiler to yell at us if the new function we write doesn't actually override any base class functions.

```
15 class Fruit {
16 public:
17     Fruit();
18     Fruit(int seeds_in, std::string color_in);
19     virtual void eat();
```

```
15 // Watermelon inherits from Fruit
16 class Watermelon : public Fruit {
17 public:
18     Watermelon();
19     virtual void eat() const;
20     void slice(int num_slices);
21     int get_num_slices();
```

Compiler is happy to let us write this new "eat()" function in Watermelon, but it doesn't override the function in Fruit!

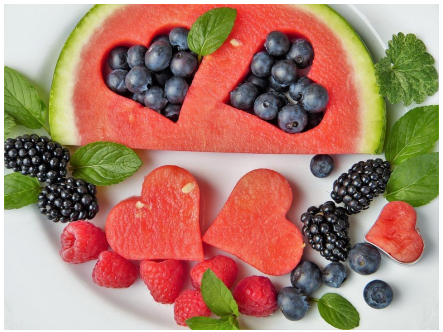
Overriding & the override keyword

- The override keyword is used when we are writing a function for a derived class that we want to OVERRIDE a function in the base class.
- We use the override keyword because we want the compiler to yell at us if the new function we write doesn't actually override any base class functions.

```
15 class Fruit {
16 public:
17     Fruit();
18     Fruit(int seeds_in, std::str
19     virtual void eat();
```

Compiler/ IDE yells at us and says we're not doing what we've intended!

```
16 class Watermelon : public Fruit {
17 public:
18     Watermelon();
19     virtual void eat() const override;   'eat' marked 'override' but does not override any member functions
```



Worksheet
Exam style Q's
Lab Assignment