



CS 11 C++ track: lecture 8

- Today:
 - Inheritance



Inheritance (1)

- In C++ we create classes and instantiate objects
- An object of class Fruit "is-a" Fruit
 - with methods to do things that Fruits do
- An object of class Banana is also a Fruit
 - probably does a lot of the same things that Fruits do
 - but maybe in a slightly different way
 - as well as other things
- Want to make this relationship explicit
 - and re-use code that is common to both classes



Inheritance (2)

```
class Fruit {  
    private:  
        Color *color;  
        int calories;  
        int seeds;  
    public:  
        Fruit();  
        ~Fruit();  
        int seedsRemaining() const;  
        int numCalories() const;  
}
```



Inheritance (3)

```
#include "Fruit.hh"

class Banana : public Fruit {
private:
    bool peelExists;
public:
    Banana();
    ~Banana();
    bool stillHasPeel() const;
}
```



Inheritance (4)

- Now Banana is a "subclass" of Fruit
- Has its own methods, plus methods of Fruit
- Any function that accepts a Fruit argument can also take a Banana
 - because a Banana "is-a" Fruit too!
 - works because all Fruit methods also defined in Bananas



Inheritance (5)

```
void printCalories(Fruit *f) {  
    cout << f->numCalories << endl;  
}
```

```
// later...
```

```
Fruit *f = new Fruit();
```

```
Banana *b = new Banana();
```

```
printCalories(f); // OK
```

```
printCalories(b); // also OK
```



Inheritance (6)

```
void printPeelStatus(Banana *b) {  
    cout << b->stillHasPeel() << endl;  
}
```

```
// later...
```

```
Fruit *f = new Fruit();
```

```
Banana *b = new Banana();
```

```
printPeelStatus(b); // OK
```

```
printPeelStatus(f); // not OK!
```

```
// a Fruit is not a Banana!
```



Inheritance (7)

```
// Add some definitions:  
Fruit::seedsRemaining() {  
    return 10; // or whatever..  
}  
  
// Want to override for Bananas:  
Banana::seedsRemaining() {  
    return 0;  
}
```



Inheritance (8)

```
void printSeedsRemaining(Fruit *f) {  
    cout << f->seedsRemaining() << endl;  
}
```

// Later:

```
Fruit *f = new Fruit();  
Banana *b = new Banana();  
printSeedsRemaining(f); // prints?  
printSeedsRemaining(b); // prints?
```



Inheritance (9)

- Problem!
- When Banana is treated like a generic Fruit in `printSeedsRemaining()`, Fruit's `seedsRemaining()` method is called
 - not what we want
- Want Banana's `seedsRemaining()` method to be called
- How do we achieve this?



Virtual methods (1)

```
class Fruit {  
    private:  
        Color *color;  
        int calories;  
        int seeds;  
    public:  
        Fruit();  
        ~Fruit();  
        virtual int seedsRemaining() const;  
        int numCalories() const;  
}
```



Virtual methods (2)

- That's all we need to change!
- We have made `seedsRemaining()` a **virtual** method
- That means that even when a Banana is being treated like a generic Fruit, the Banana version of `seedsRemaining()` will be called
- NOTE: When Banana treated like a generic Fruit, can't call Banana-specific methods



Virtual methods (3)

- Virtual methods are what we want in this case
- Why not make ALL methods virtual?
 - because there is a cost associated with it
- Only make methods virtual if you expect that you will need to override them



Pure virtual methods (1)

- Still something strange
- Does the concept of a "generic Fruit" mean anything?
 - no.
- But we can create generic Fruits and call methods on them
- What if we want to say "this is what a Fruit can do" but not specify behavior
 - leave to subclasses to do that



Pure virtual methods (2)

```
class Fruit {
    private:
        Color *color;
        int calories;
        int seeds;
    public:
        Fruit();
        ~Fruit();
        virtual int seedsRemaining() const = 0;
        virtual int numCalories() const = 0;
}
```



Pure virtual methods (3)

- Now can't define Fruit instances
 - because there is no definition for methods `seedsRemaining()` and `numCalories()`
- But can still use it as base class for Banana
 - which does define those methods
- The `= 0` syntax (along with the **virtual** keyword) indicates that a method is a pure virtual method
- Must be defined in subclasses to get an object you can instantiate



Protected fields (1)

- Fields that are private are *really* private!
- Only objects of that class can access them directly
- Objects of subclasses can't access those fields
 - even though they "own" the fields in a sense
- Sometimes this is what you want
 - and sometimes it isn't
- How do we say "I want this field to be directly accessible even in subclasses?"



Protected fields (2)

```
class Fruit {  
    protected:  
        Color *color;  
        int calories;  
        int seeds;  
    public:  
        Fruit();  
        ~Fruit();  
        virtual int seedsRemaining() const = 0;  
        virtual int numCalories() const = 0;  
}
```



Protected fields (3)

- Define fields to be **protected**
- Now, subclasses will also be able to access the field directly
- Which is better, **private** or **protected**?
 - beats me
 - much controversy in OO theory about this
- **private** is safer
- **protected** can be more efficient
 - don't need extra accessors
- and may be more "conceptually correct"



The STL

- STL stands for Standard Template Library
- Lots of useful classes for various uses
- `vector<int>` → "vector" (array) of integers
- `list<Fruit *>` → linked list of `Fruit*`
- `map<string, Fruit*>` → "map" (association) between string and `Fruit*`
- Need to use for lab 7
- More on web page



Lab 7

- Very easy exercise on inheritance
- Shouldn't take long
- Can use extra time to get caught up on older labs