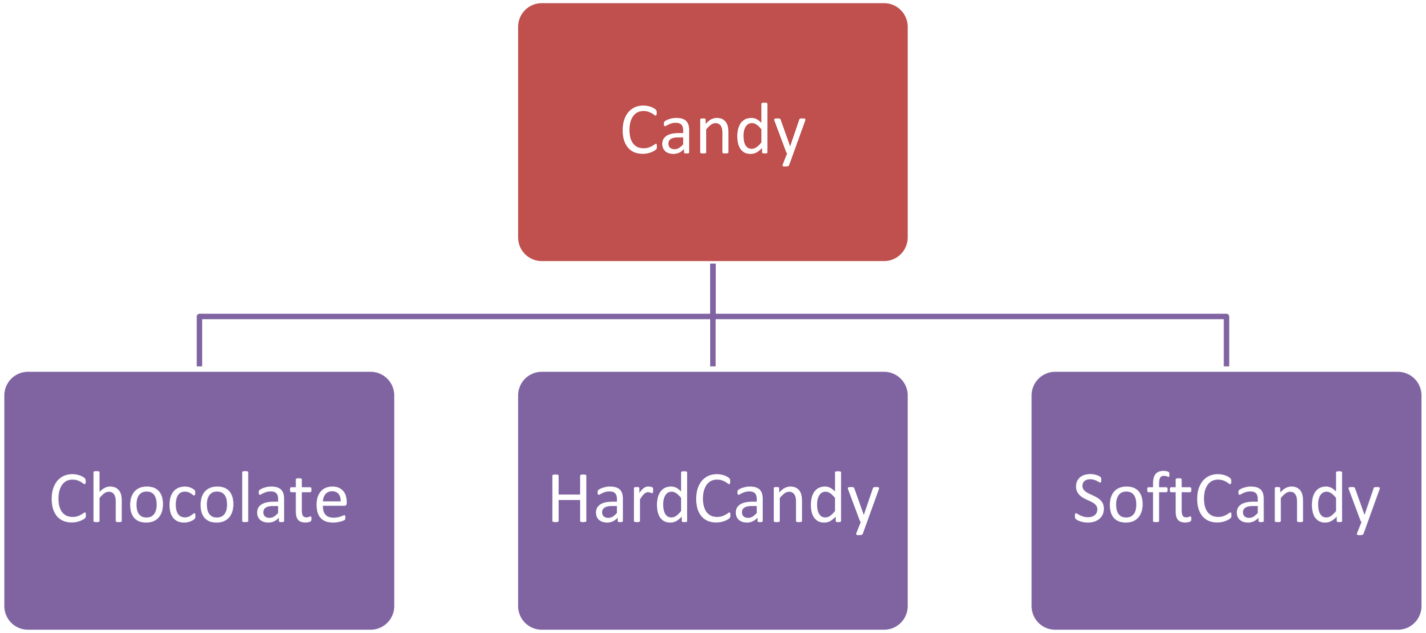
Inheritance; Initialization lists; Polymorphism

ITP 165 – Fall 2015

Week 13, Lecture 2

The Candy hierarchy

- A tasty set of classes



Describing the relationships

- Lets make a `Candy` class and the others so that:
  - A `Chocolate` ***is-a*** type of `Candy`
  - A `HardCandy` ***is-a*** type of `Candy`
  - A `SoftCandy` ***is-a*** type of `Candy`

- Lets put all the common attributes and functionality into `Candy`
  - All `Candy` has a name (`mName`) attribute
  - All `Candy` has a number of calories (`mCalories`) attribute
  - All `Candy` can display its information (`display()`)

Defining the Candy base class

```
// Candy.h
#pragma once
#include <string>

class Candy {
private:
  std::string mName;
  double mCalories;
public:
  Candy(std::string inName, double inCals);
  void display();
};
```

Defining the Candy base class

```
// Candy.h
#pragma once
#include <string>

class Candy {
private:
  std::string mName;
  double mCalories;
public:
  Candy(std::string inName, double inCals);
  void display();
};
```

These are unavailable to `Chocolate` etc. `private` means only `Candy` members have access to them

The protected keyword

- The `private` member variables are only available to class functions, not to the children's functions
  - In other words, `Chocolate` could not access `Candy`'s `mName` variable

- The `protected` keyword is like `public` and `private` – it changes accessibility for parts of a class

- The `protected` keyword grants access to the class and all its children
  - Making `mName` a `protected` variable would allow `Candy` AND `Chocolate` access, but not `main`

A better Candy class

```
// Candy.h
#pragma once
#include <string>

class Candy {
protected:
  std::string mName;
  double mCalories;
public:
  Candy(std::string inName, double inCals);
  void display();
};
```

The Candy class functions

```cpp
// Candy.cpp
#include "Candy.h"
#include <iostream>

Candy::Candy(std::string inName, double inCals)
{
  mName = inName;
  mCalories = inCals;
}

void Candy::display()
{
  std::cout << mName << " : " << mCalories << std::endl;
}
```

Making Candy!

```
// Driver.cpp
#include "Candy.h"

int main()
{
   Candy myCandy("Snickers", 100);
   myCandy.display();

   return 0;
}
```

Making Candy!

C:\Windows\system32\cmd.exe

```
Snickers : 100
Press any key to continue . . .
```

Describing Chocolate

- Because `Chocolate` ***is-a*** `Candy` we know…
  - That `Chocolate` has a name (`mName`)
  - That `Chocolate` has calories (`mCalories`)
  - That `Chocolate` can display information (`display()`)

- But `Chocolate` is more than just `Candy`
  - It has a % of cocoa (`mPerCocoa`)
  - It has its own display information (`display()`) that includes the % cocoa

Defining Chocolate

```
// Chocolate.h
#pragma once
#include "Candy.h"

class Chocolate : public Candy {
private:
  double mPerCocoa;
public:
  Chocolate(std::string inName,
    double inCals, double inPerC);
  void display();
};
```

Defining Chocolate

```
// Chocolate.h
#pragma once
#include "Candy.h"

class Chocolate : public Candy {
private:
  double mPerCocoa;
public:
  Chocolate(std::string inName,
    double inCals, double inPerC);
  void display();
};
```

This means that Chocolate is now defined as a child of Candy.

Defining Chocolate

```cpp
// Chocolate.cpp
#include "Chocolate.h"
#include <iostream>

Chocolate::Chocolate(std::string inName, double inCals, double inPerC)
{
  mName = inName;
  mCalories = inCals;
  mPerCocoa = inPerC;
}

void Chocolate::display()
{
  std::cout << mName << " : " << mCalories << std::endl;
  std::cout << "\t% cocoa: " << mPerCocoa << std::endl;
}
```

Defining Chocolate

```
// Chocolate.cpp
#include "Chocolate.h"
#include <iostream>

Chocolate::Chocolate(std::string inName, double inCals, double inPerC)
{
  mName = inName;
  mCalories = inCals;
  mPerCocoa = inPerC;
}

void Chocolate::display()
{
  std::cout << mName << " : " << mCalories << std::endl;
  std::cout << "\t% cocoa: " << mPerCocoa << std::endl;
}
```

C++ reports an error here “Error: no default constructor exists for class Candy”

The need for initialization lists

- When making a `Chocolate` C++ first makes a `Candy`
  - It calls the `Candy` constructor first, then the `Chocolate` one

- When making `Candy` before `Chocolate` C++ calls the `Candy` default constructor

- To make C++ use a different constructor, we'll need to use an ***initialization list***

Redefining Chocolate

```
// Chocolate.cpp
#include "Chocolate.h"
#include <iostream>

Chocolate::Chocolate(std::string inName, double inCals,
   double inPerC) : Candy(inName, inCals)
{
   mPerCocoa = inPerC;
}

void Chocolate::display()
{
   std::cout << mName << " : " << mCalories << std::endl;
   std::cout << "\t% cocoa: " << mPerCocoa << std::endl;
}
```

Redefining Chocolate

```cpp
// Chocolate.cpp
#include "Chocolate.h"
#include <iostream>

Chocolate::Chocolate(std::string inName, double inCals,
   double inPerC) : Candy(inName, inCals)
{
   mPerCocoa = inPerC;
}

void Chocolate::display()
{
   std::cout << mName << " : " << mCalories << std::endl;
   std::cout << "\t% cocoa: " << mPerCocoa << std::endl;
}
```

Explicitly calls the 2 parameter constructor of Candy

Also legal!

```cpp
// Chocolate.cpp
#include "Chocolate.h"
#include <iostream>

Chocolate::Chocolate(std::string inName, double inCals,
   double inPerC) : Candy(inName, inCals), mPerCocoa(inPerC)
{
  // Nothing needed here!
}

void Chocolate::display()
{
   std::cout << mName << " : " << mCalories << std::endl;
   std::cout << "\t% cocoa: " << mPerCocoa << std::endl;
}
```

Initialization List

```
Chocolate::Chocolate(std::string inName, double inCals,
   double inPerC) : Candy(inName, inCals), mPerCocoa(inPerC)
{
  // Nothing needed here!
}
```

Single colon to start initialization list

Parameterized constructor from parent

Commas separate list items

Regular member variables can be set the same way!

About initialization lists

- Initialization lists are good to use for constructors
  - Often, there's no code to write in the body

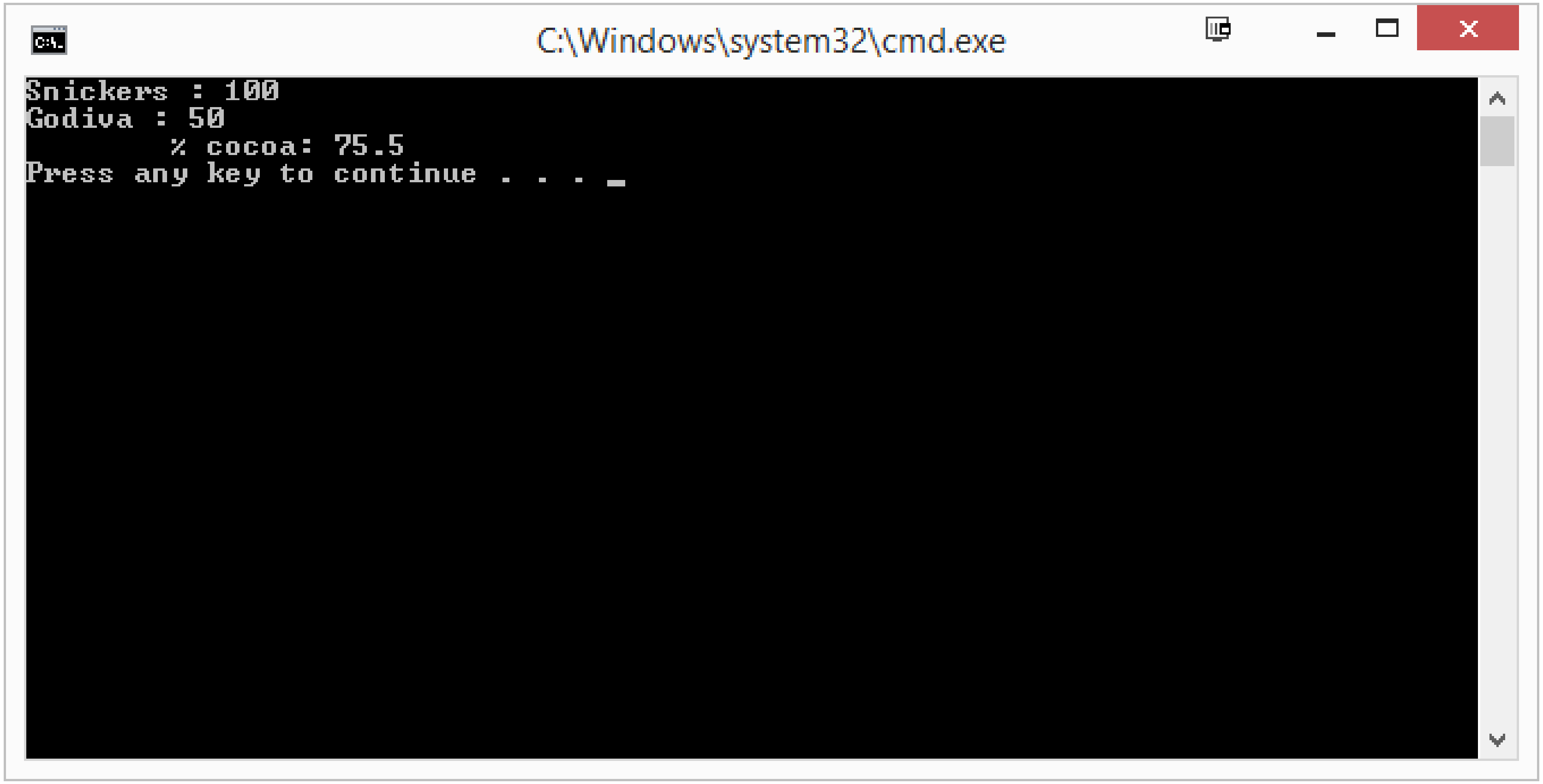
- Initialization lists are **REQUIRED** when you have no default constructor for the base class, but have parameterized constructors

- Initialization lists only set member variables of the class for which the constructor is a member function AND/OR call constructors for the base class
  - Cannot set `mName(inName)` in constructor for `Chocolate`, you must call constructor for `Candy` class

The new Driver

```cpp
// Driver.cpp
#include "Candy.h"
#include "Chocolate.h"

int main()
{
  Candy myCandy("Snickers", 100);
  myCandy.display();

  Chocolate myChocolate("Godiva", 50, 75.5);
  myChocolate.display();

  return 0;
}
```

Making more Candy!



This works, kinda... why?

```
// Driver.cpp
#include "Candy.h"
#include "Chocolate.h"

int main()
{
  Candy* myCandy;

  myCandy = new Candy("Snickers", 100);
  myCandy->display();
  delete myCandy;
  myCandy = nullptr;

  myCandy = new Chocolate("Godiva", 50, 75.5);
  myCandy->display();
  delete myCandy;
  myCandy = nullptr;

  return 0;
}
```

This works, kinda… why?

```cpp
// Driver.cpp
#include "Candy.h"
#include "Chocolate.h"

int main()
{
  Candy* myCandy;

  myCandy = new Candy("Snickers", 100);
  myCandy->display();
  delete myCandy;
  myCandy = nullptr;

  myCandy = new Chocolate("Godiva", 50, 75.5);
  myCandy->display();
  delete myCandy;
  myCandy = nullptr;

  return 0;
}
```

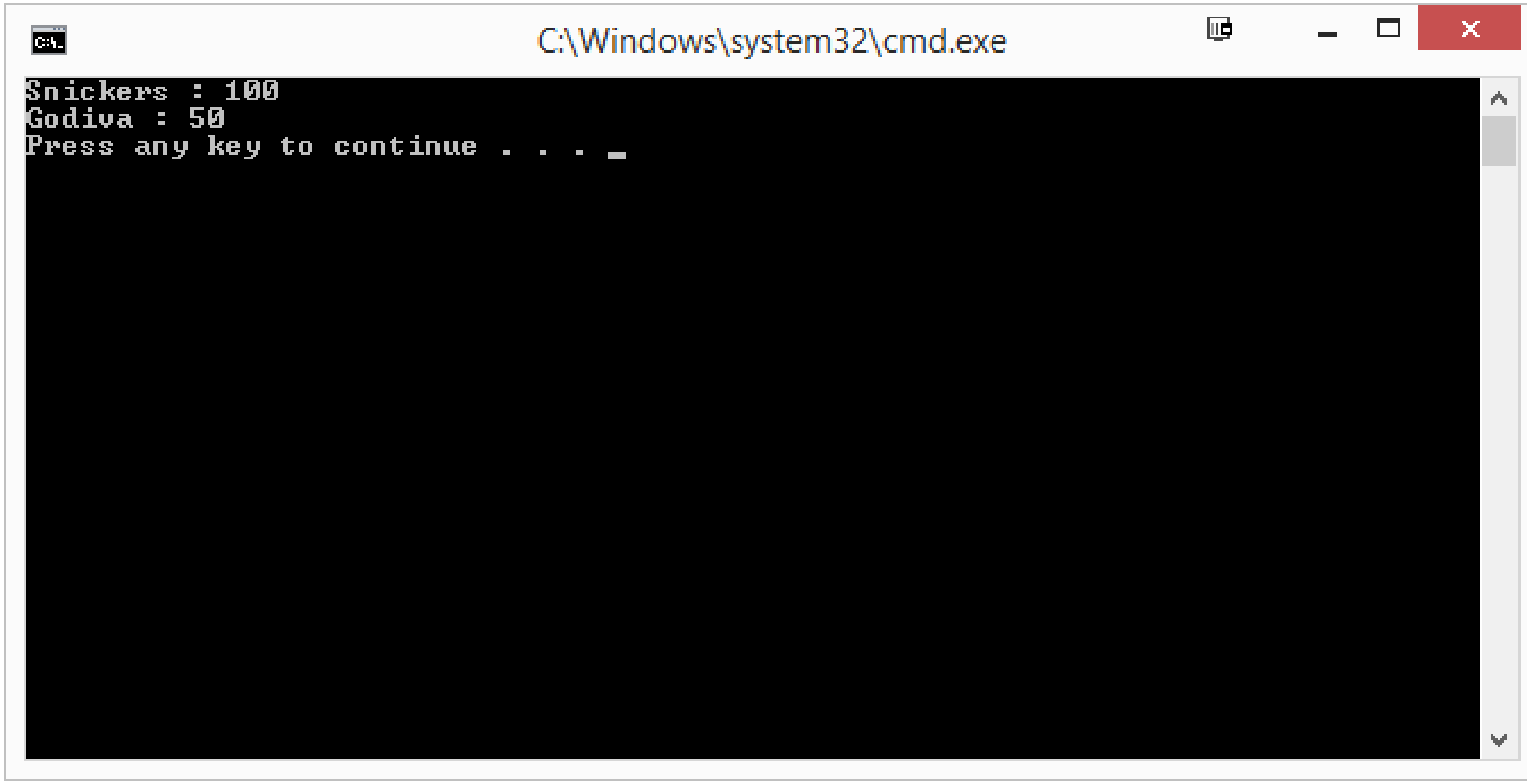
Chocolate ***is-a*** Candy so C++ will assign the pointer (only works with pointers)

This works, kinda... why?

```
// Driver.cpp
#include "Candy.h"
#include "Chocolate.h"

int main()
{
  Candy* myCandy;

  myCandy = new Candy("Snickers", 100);
  myCandy->display();
  delete myCandy;
  myCandy = nullptr;

  myCandy = new Chocolate("Godiva", 50, 75.5);
  myCandy->display();
  delete myCandy;
  myCandy = nullptr;

  return 0;
}
```

Candy has a display function, so does Chocolate who's will be called?

Previous code's wrong output



The virtual keyword

- To make the `Candy` pointer call the `display` of the `Chocolate` class and not the `Candy` version we need a new keyword
- The `virtual` keyword tells C++: "Call the most derived version of this function with the same signature."
- Remember: A function's signature is:
  - It's return type
  - It's name
  - It's parameters
- All 3 must match for `virtual` to work properly

The final Candy class

```
// Candy.h
#pragma once
#include <string>

class Candy {
protected:
  std::string mName;
  double mCalories;
public:
  Candy(std::string inName, double inCals);
  virtual void display();
};
```

The final Candy class

```
// Candy.h
#pragma once
#include <string>

class Candy {
protected:
  std::string mName;
  double mCalories;
public:
  Candy(std::string inName, double inCals);
  virtual void display();
};
```

Now the child classes (like Chocolate) can override the display function from a Candy pointer

One keyword makes a big difference!

C:\Windows\system32\cmd.exe

```
Snickers : 100
Godiva : 50
        % cocoa: 75.5
Press any key to continue . . .
```

Why Candy is bad for you

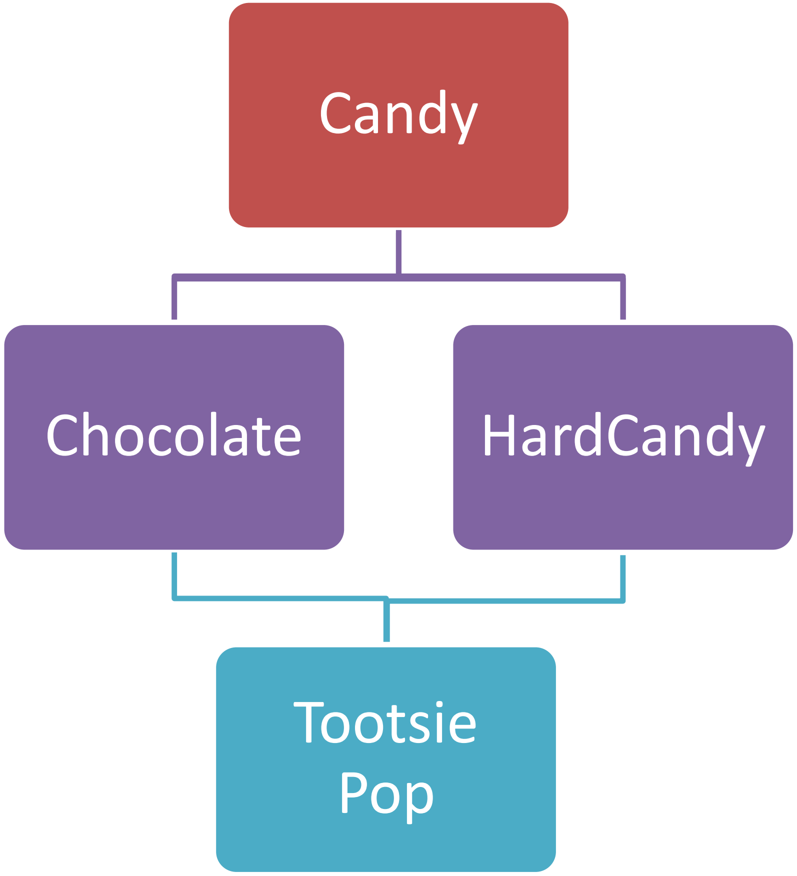
- Candy is a bad example of inheritance

- Consider a Tootsie Pop
  - Hard candy lollypop with a chocolate-chewy inside
  - It's both `Chocolate` and `HardCandy`

- Problem: Both `Chocolate` and `HardCandy` inherit from `Candy` so they BOTH have a variable called `mName`

"The deadly diamond of death"

- Multiple inheritance with poorly defined classes can lead to problems like this



Lab Practical #23