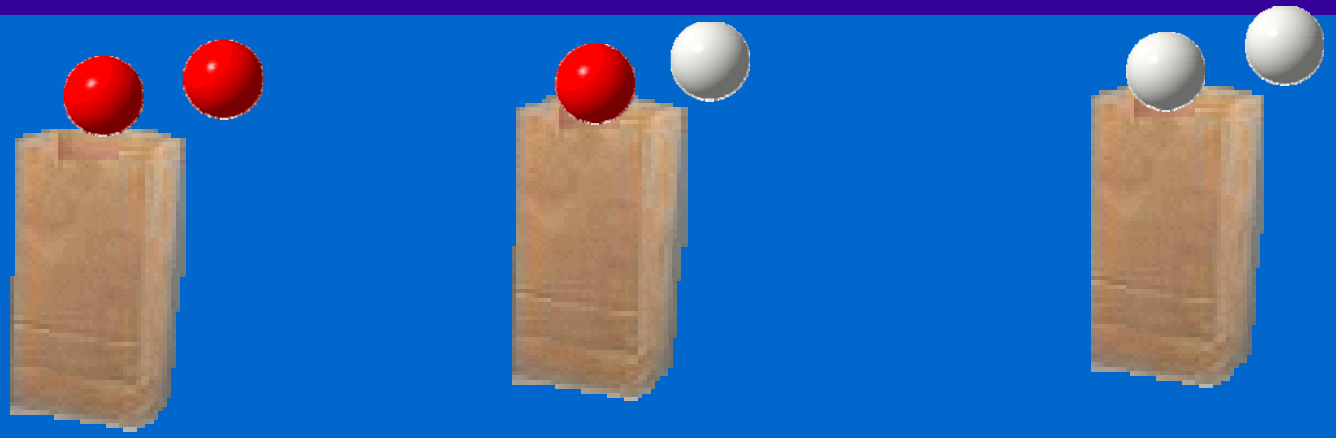
A C++ Program Example: Three Bags

C++ Object Oriented Programming

Pei-yih Ting

NTOU CSE

A Simple Probabilistic Experiment



- Three paper bags, each bag is given two balls with colors shown in the above figure
- We perform the following probabilistic experiment:
  - Step 1: put balls into each bags
  - Step 2: randomly choose a bag
  - Step 3: randomly draw one ball out of the bag
  - Step 4: if the color is red, then take the second ball out of the bag
    otherwise stop the experiment

we want to find out the probability that the **second ball is red** at step 4

A Simple Probabilistic Experiment

Is the remaining ball red or white?

What is the probability of being red again?

$$\Pr\{\text{2nd is red} \mid \text{1st is red}\} = \frac{\Pr\{\text{1st is red and 2nd is red}\}}{\Pr\{\text{1st is red}\}}$$

$$= \frac{\Pr\{\text{1st bag is picked}\}}{\Pr\{\text{1st bag picked and 1st ball is red}\} + \Pr\{\text{2nd bag picked and 1st ball is red}\}}$$

$$= \frac{1/3}{1/3 + 1/3 \times 1/2} = 2/3$$

A Program Written in C (1/3)

- Let's try simulating this experiment and caculating the probability by the so called Monte Carlo method
- Converting the problem specification into C
  - Let's do the experiments 10000 times to estimate the probability
    → a **for** loop
  - Using a random variable in the range {0, 1, 2} to emulate the random choice of a bag at step 2 → variable **draw1**
  - Using another random variable in the range {0, 1} to emulate the random selection of a ball from the chosen bag at step 3
    → variable **draw2**
  - At each run of experiment, keep the count of those experiments with the first selected ball being red → variable **totalCount**
  - At each run of experiment, keep the count of those experiments with both balls being red → variable **redCount**
  - Take the ratio of **redCount** and **totalCount** to be the result

A Program Written in C (2/3)

```
04 #include <stdio.h>
05 #include <stdlib.h>
06 #include <time.h>
07
08 void main()
09 {
10     long i;
11     int draw1, draw2, choice, tmp;
12     long totalCount=0L,
               redCount=0L;
13
14     srand(time(NULL));
15     for (i=0; i<100000L; i++)
16     {
17         draw1 = rand() % 3; // pick a bag out of the three
18
19         if (draw1 == 0) // (Red, Red)
20         {
21             totalCount++;
22             redCount++;
23         }
24         else if (draw1 == 1) // (Red, White)
25         {
26             draw2 = rand() % 2;
27             if (draw2 == 0) // the first is Red
28                 totalCount++;
29             else // the first is White
30                 /* do nothing */;
31         }
32     }
33
34     printf("Pr(2nd is red | 1st is red)=%lf\n",
35       (double)redCount / (double)totalCount);
36 }
```

Output:
Pr(2nd is red | 1st is red)=0.665299

A Program Written in C (3/3)

- Is the conversion process from the problem specification to a C program direct and trivial? NO
- If you just read the C program alone, can you reconstruct the problem easily and exactly? NO
- There are many missing pieces of the original problem specification in the above C program.
  - 100000 experiments mixed together (without my explanations, some might have a wrong picture of what the program actually does) Variables totalCount and redCount are something not in the original problem specification.
  - Meaning of variables draw1 and draw2 are a little bit intriguing.
  - There is no bag appearing in the program.
  - No codes are associated with the case that the bag with two white balls is selected.

The Same Program Written in C++

- Model the problem *in the application domain* (*the problem domain*) with minimal transformation to the computer technical domain
- Identify all objects, describe their functionalities and inter-relationships, categorize them, extract common characteristics
  - Experiment (Game)
    - contain three bags
    - random selection of a bag
  - Bag
    - contain zero, one, or two balls
    - random selection of a ball inside
  - Ball
    - color

The Same Program Written in C++

- Characterize the usages of the overall system: these usages would integrate the functionalities of the above designed set of objects (classes) (Use cases, Scenarios)
  - Perform an experiment: requires the participation of three bags, each bag has two balls with color as specified, select a bag, then select a ball, check its color, if red, check the color of the second ball
  - Perform the above experiment for 100000 times and keep the statistics

bottom-up programming methodology

- Use existing/common OO architecture or components to implement the designed architecture.
- Move on to customized OO programming.

Game Class

```
041 ---------------- 2:Game.h ----------------
042
043
044 #ifndef game_h
045 #define game_h
046
047 #include "Bag.h"
048
049 class Game
050 {
051 public:
052     Bag *getABag();
053     Game();
054     ~Game();
055 private:
056     Bag *m_bags[3];
057 };
058
059 #endif
```

```
062 ---------------- 3:Game.cpp ------------
063
064
065 #include "Game.h"
066 #include "Bag.h"
067 #include <stdlib.h> // rand()
068
069 Game::Game()
070 {
071     m_bags[0] = new Bag(0,0);
072     m_bags[1] = new Bag(0,1);
073     m_bags[2] = new Bag(1,1);
074 }
075
076 Game::~Game()
077 {
078     int i;
079     for (i=0; i<3; i++)
080         delete m_bags[i];
081 }
082
083 Bag *Game::getABag()
084 {
085     return m_bags[rand()%3];
086 }
```

Bag Class

```
089 ---------- 4:Bag.h ----------
090
091
092 #ifndef BAG_H
093 #define BAG_H
094
095 class Ball;
096
097 class Bag
098 {
099 public:
100     Ball *getABall();
101     void putBallsBack();
102     Bag(int color1, int color2);
103     ~Bag();
104 private:
105     Ball *m_balls[2];
106     int m_numberOfBalls;
107 };
108
109 #endif
```

```
112 ---------- 5:Bag.cpp ----------
113
114
115 #include "Bag.h"
116 #include "Ball.h"
117 #include <stdlib.h> // rand()
118
119 Bag::Bag(int color1, int color2)
120     : m_numberOfBalls(2)
121 {
122     m_balls[0] = new Ball(color1);
123     m_balls[1] = new Ball(color2);
124 }
125
126 Bag::~Bag()
127 {
128     delete m_balls[0];
129     delete m_balls[1];
130 }
131
```

Bag Class (cont'd)

```
132 Ball *Bag::getABall()
133 {
134     if (m_numberOfBalls == 0)
135         return 0;
136     else if (m_numberOfBalls == 1)
137     {
138         m_numberOfBalls = 0;
139         return m_balls[0];
140     }
141     else
142     {
143         int iPicked = rand()%2;
144         Ball *pickedBall = m_balls[iPicked];
145         if (iPicked == 0)
146         {
147             m_balls[0] = m_balls[1];
148             m_balls[1] = pickedBall;
149         }
150         m_numberOfBalls = 1;
151         return pickedBall;
152     }
153 }
```

```
154
155 void Bag::putBallsBack()
156 {
157     m_numberOfBalls = 2;
158 }
```

This design and implementation are problematic. When you get a ball from a bag, the ownership of the ball is better naturally transferred.

Ball Class

```
161 ---------- 6:Ball.h ----------
162
163
164 #ifndef BALL_H
165 #define BALL_H
166
167 class Ball
168 {
169 public:
170     bool IsRed();
171     Ball(int color);
172 private:
173     int m_redWhite;
174 };
175
176 #endif
```

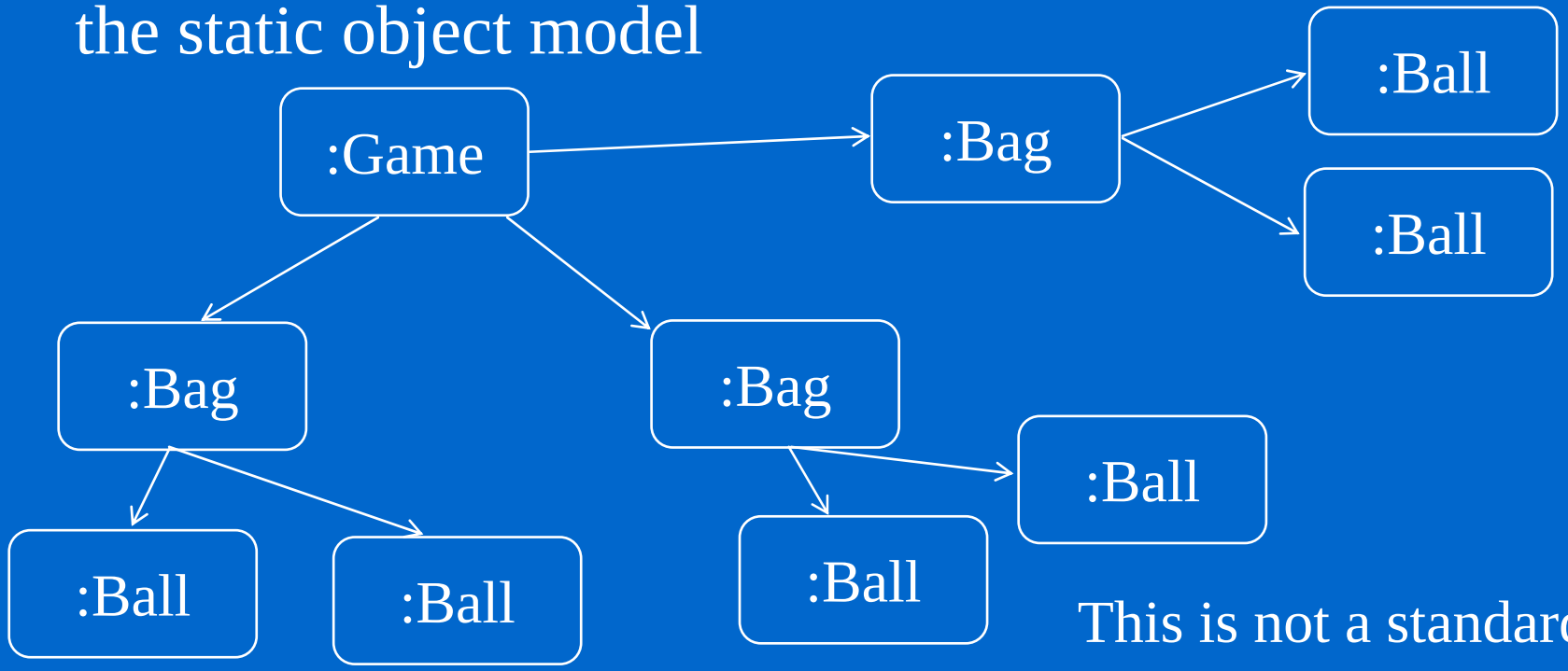
```
179 ---------- 7:Ball.cpp ----------
180
181
182 #include "Ball.h"
183
184 Ball::Ball(int color)
185 : m_redWhite(color)
186 {
187 }
188
189 bool Ball::IsRed()
190 {
191     if (m_redWhite == 0)
192         return true;
193     else
194         return false;
195 }
```

main()

```
001
002 ------------ 1:main.cpp ------------
003
004
005 #include "Game.h"
006 #include "Bag.h"
007 #include "Ball.h"
008 #include <stdlib.h> // srand()
009 #include <time.h> // time()
010 #include <iostream.h>
011
012 void main()
013 {
014     int   i;
015     Game  theGame;
016     Bag  *pickedBag;
017     Ball *pickedBall;
018     int   totalCount = 0;
019     int   secondIsAlsoRed = 0;
020
021     srand(time(0));
022
023     for (i=0; i<100000; i++)
024     {
025         pickedBag = theGame.getABag();
026         pickedBall = pickedBag->getABall();
027         if (pickedBall->IsRed())
028         {
029             totalCount++;
030             if (pickedBag->getABall()->IsRed())
031                 secondIsAlsoRed++;
032         }
033         pickedBag->putBallsBack();
034     }
035
036     cout << "The probability that remaining
        ball is red = "
037          << ((double)secondIsAlsoRed/totalCount)
        << "\n";
038 }
039
040
```

Some Observations

- Lengthier codes
- More functions
- Slower (maybe)
- There is a clear conceptual architecture for the program: the static object model



This is not a standard graph.

More Observations

- Bottom-up design: some of the functions of an object might not even be used in this particular application. Ex. the Complex class in the lab
- The functions and data of each class/object are self-contained.
- The *data coupling* and *control coupling* between an object and other objects are designed to be minimal. Objects interact with each other through constrained interface functions.
- Software operations mimic the physical functions of the original real world problem.
- The overall program functionalities are provided by a set of cooperating objects.

Even More

- Many consumer products are designed with cooperating parts: e.g.
  - Car: engine, fuel system, wheels, transmission, steeling, bucket seats, …
  - Computer: CPU, MB, RAM, HD, display interface, keyboard/mouse, screen
- ++ Just a little engineering common sense would tell you how to maintain or repair a car/computer when it breaks down – find out which part is not functioning well and replace it with a good one.
- ++ The quality control of manufacturing each part is much easier.
- — The design of such a product with many replaceable parts are not trivial. It certainly increases the design/manufacturing cost and thus the price/competitive capability of the product.
- ++ However, you can see that this is a cost efficient strategy to make a product work for a few years and your customers satisfied.
- Ask yourself a question: Is the technology not good to glue everything together as a whole? to make the product more monolithic, more tasteful, more handy, more style of future

Summary

- There are many OOA / OOD methodologies since ’80s.
- After a major unification of *Jacobson*, *Booch*, and *Rumbaugh* in the ’90s, we have the UML, Unified Modeling Language for describing the OO design artifacts and the design process (the methodology) associated with it.
- In this course, we will focus on OOP, especially on how C++ provides features for implementing your OO design.
- We will try to elaborate those OO concepts provided by the implementation language: namely, *objects*, *abstraction*, *interface*, *encapsulation*, *inheritance*, *polymorphism*, generic programming (the *templates*), and *exceptions*.
- You are encouraged to browse the OOA, OOD stuffs.