CS32 Discussion 1F

Week 5

TA: Manoj Reddy

LA: Katherine Zhang

Credit: Prof. Carey Nachenberg

Outline

- Object Oriented Programming
  - Inheritance
  - Polymorphism
- Recursion

Inheritance

- Basis of all Object-Oriented Programming
- Models the “is a” relationship
  - Different from “has a”
- Terminology:
  - Person: Base class or Superclass
  - Student: Derived class or Subclass
- Inheritance can be multiple levels deep
- 3 main uses:
  - Reuse
  - Extension
  - Specialization
- Reduces duplicate code

```
class Person
{
public:
   string getName(void);
   ...

private:
   string m_sName;
   int    m_nAge;
};
```

```
class Student :
public Person
{
public:
   // new stuff:
   int GetStudentID();
   ...
private:
   // new stuff:
   int m_studentID;
   ...
};
```

Reuse

- Only public members in base class can be reused in derived class
- Private members in the base class are hidden from derived class(es)
- protected in base class:
  - Allows all it's derived classes to reuse the protected members from the base class
  - Still prevents rest of program to access protected members

Extension

- Allows creation of additional methods or data to a derived class
- These extensions are unknown to the base class
- Base class only know about itself
  - It has not idea about classes derived from it!

Specialization

- Override existing functions from base class in derived class
- Insert ‘virtual’ keyword in front of both original and replacement functions
  - Except when defining member functions outside the class
  - Only use ‘virtual’ keyword within your class definition
- By default, most derived version of virtual method will be called
- To call the base class’s version

baseclass::method()

```
class Student
{
public:
  virtual void WhatDoISay()
  {
    cout << "Go bruins!";
  }
   ...
};
```

```
class NerdyStudent: public Student
{
public:
   virtual void WhatDoISay()
   {
     cout << "I love circuits!";
   }
   ...
};
```

Inheritance: Construction & Destruction

- Constructor:
  - Construct base part first and then derived part
  - Constructs member variables first (in order) and then run the constructor
- Destructor:
  - Reverse order
  - Destructs the derived part first then the base part
  - Runs the destructor for class and then destructs the member variables (reverse order)
- Note: Inheritance might require the use of initializer lists

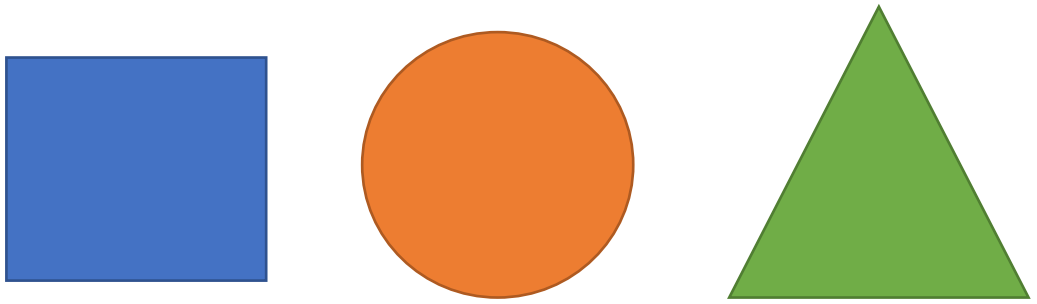
Polymorphism

- Any time we use a base pointer or a base reference to access a derived object
- Only works when you use a ***reference*** or ***pointer*** to pass an object (same underlying implementation)
- Example: Shape, Square, Circle
  - *void PrintPrice(Shape &s)*
- Superclass pointer can point to a subclassed variable (not the other way around)

```
class Person
{
public:
  string getName();
  ...

private:
  string
  int
};
```

```
class Student :
        public Person
{
public:
  // new stuff:
  int getStudentID();
private:
  // new stuff:
  int m_nStudentID;
};
```

```
void SayHi(Person *p)
{
    cout << "Hello " <<
      p->getName();
}


int main()
{
   Student s("Carey",38,3.9);

   SayHi(&s);
}
```



Pure Virtual Functions & Abstract Base Class

- Pure virtual function
  - Has no actual code
  - Indicating that the base version of the function will never be called
  - Syntax: virtual float getArea() = 0;
- Abstract Base Class
  - Atleast one pure virtual function
  - Cannot instance an instance of ABC
- Benefits:
  - Avoids writing “dummy” logic in base class
  - Ensure the programmer to implement the correct functionality in derived class

Recursion

- Function calls itself
- Two rules:
  - Must have a stopping condition (aka Base Case)
    - Your recursive function must be able to solve the simplest, most basic problem without using recursion
    - Every recursive function must have some mechanism to allow it to stop calling itself
  - Must have a simplifying step
    - Every time a recursive function calls itself, it must pass in a smaller sub-problem that ensures the algorithm will eventually reach its stopping condition
    - A recursive function must eventually reach its stopping condition or it'll run forever
- Recursive functions should never use global, static or member variables
- They should only use local variables and parameters!

Example: Factorial

```
int fact(int n)
{
  if (n == 0)
     return 1;

  return  n * fact(n-1);
}
```

Recursion Problem

- What is the output of the following program?

```
int fun(int a, int b)
{
  if (b == 0)
     return 0;
  if (b % 2 == 0)
     return fun(a+a, b/2);

  return fun(a+a, b/2) + a;
}
```

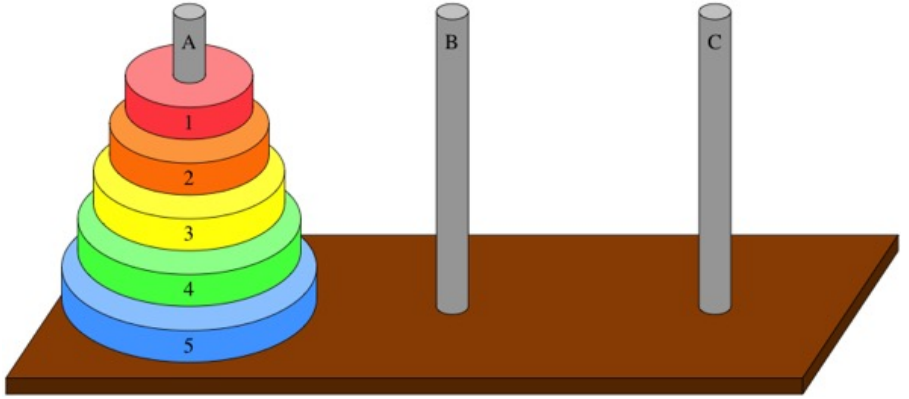
```
int main()
{
  cout << fun(6,8);
  return 0;
}
```

Practice

```
bool fun_a(int n) {
    if (n == 0)
        return true;
    else
        return fun_b(n - 1);
}

bool fun_b(int n) {
    if (n == 0)
        return false;
    else
        return fun_a(n - 1);
}
```

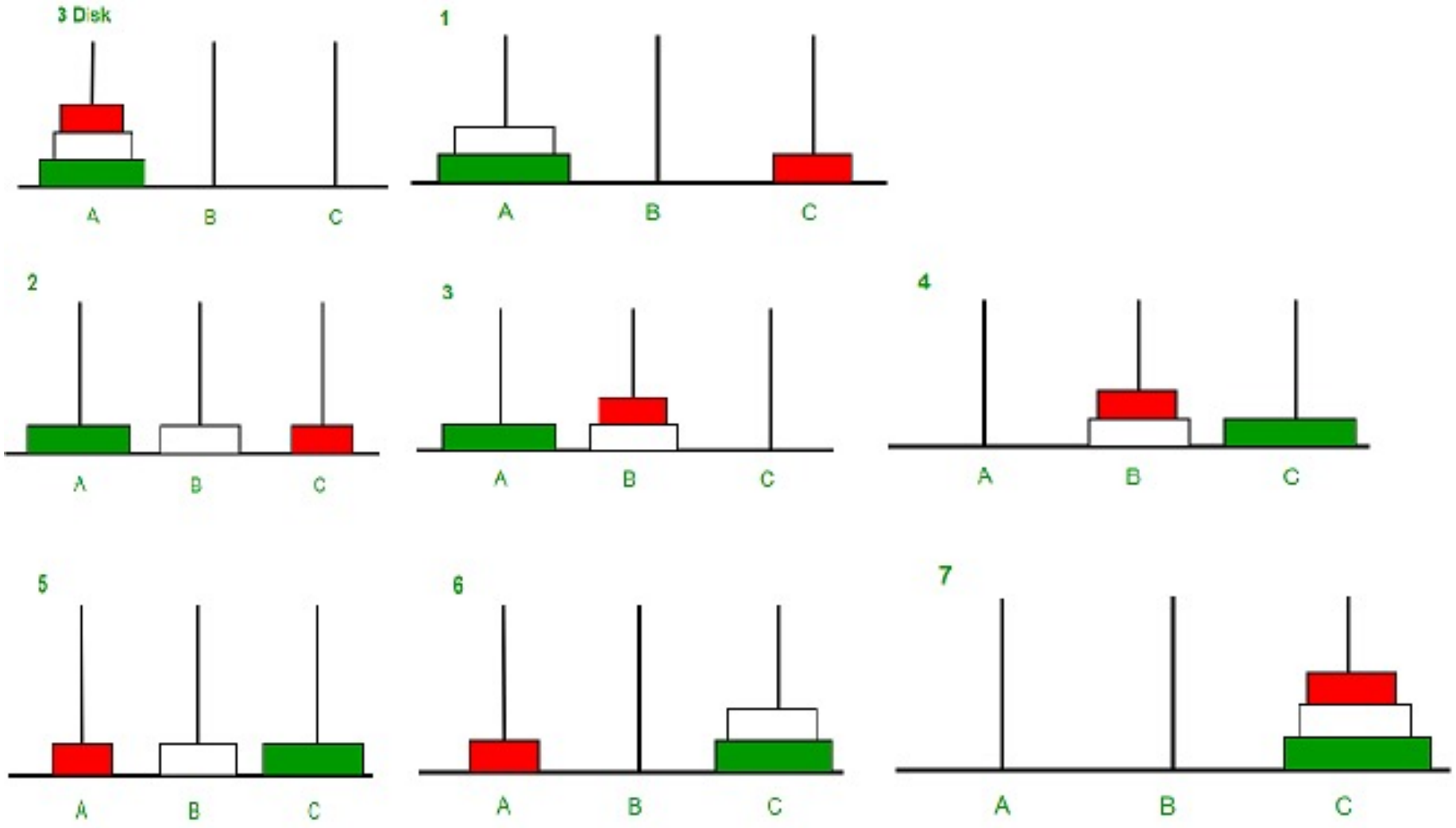
```
int main(){
    cout << fun_a(22) << endl;
}
```

Tower of Hanoi



- 3 rods and n disks
- Goal: Move n disks from rod A to rod C
- Rules:
  - Only one disk can be moved at a time
  - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack
  - No disk may be placed on top of a smaller disk
- How to solve for
  - n=1
  - n=2
  - n=3
  - n=5
  - …

Tower of Hanoi



Tower of Hanoi

```cpp
#include <iostream>
using namespace std;

// C recursive function to solve tower of hanoi puzzle
void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
     if (n == 1)
     {
     cout << "\n Move disk 1 from rod " << from_rod << " to rod " << to_rod << endl ;
     return;
     }
     towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
     cout << "\n Move disk " << n << " from rod " << from_rod << " to rod " << to_rod;
     towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

int main()
{
     int n = 4; // Number of disks
     towerOfHanoi(n, 'A', 'C', 'B');  // A, B and C are names of rods
     return 0;
}
```