CSE 303
Concepts and Tools for Software Development

Magdalena Balazinska

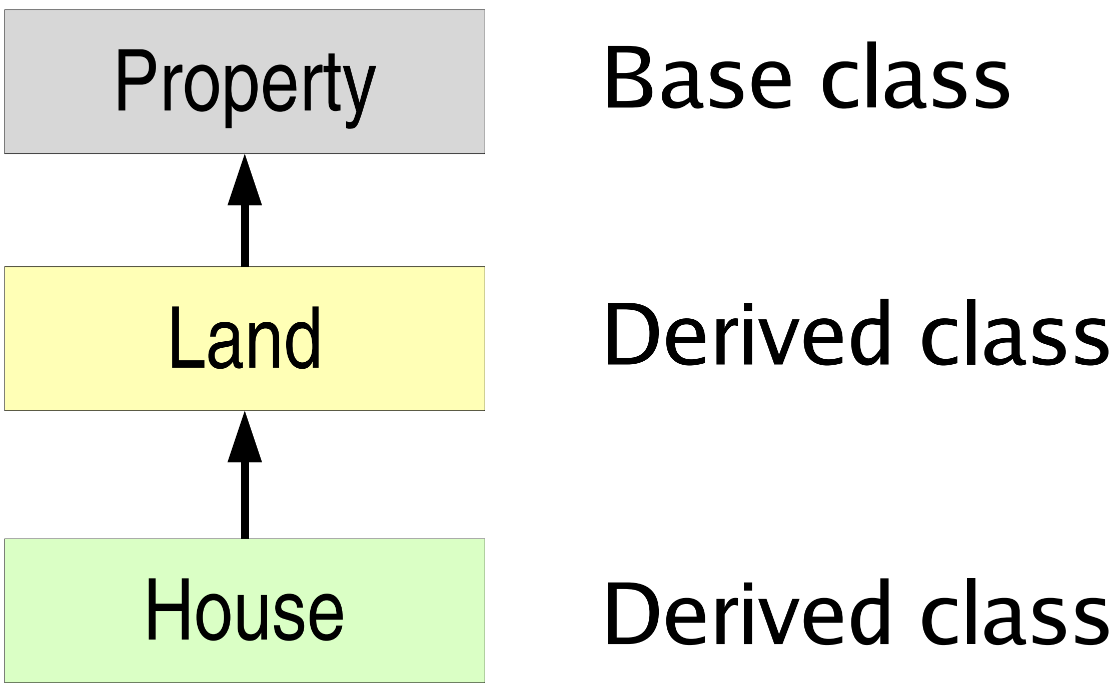
Winter 2007

Lecture 18 – Inheritance (virtual functions and abstract classes) and templates

Where We Are

- We have already covered the introduction to C++
  - Basic syntax (hello world), namespaces
  - Basics of defining and using classes
  - Allocating objects on the stack and on the heap
  - Copy constructors (call-by-value and call-by-reference)
  - Started talking about inheritance
- Today, we will discuss inheritance in greater depth
  - Casting in C++
  - Virtual functions
  - Abstract classes
- We will also start discussing templates

Our Inheritance Example



Last Time

- Last time we examined this example to see
  - Inheritance syntax
  - Access specifiers (public, protected, and private) and what they mean with subclasses
  - What happens when we construct or destroy objects

- Next questions are
  - How to cast pointers
  - What happens when a class overrides a function of its parent class... not always what you think!

C-Style Type Casting

- With inheritance, we often want to cast between pointers to different classes in our class hierarchy
- C-style type casting is dangerous
- Compiler lets you do almost what you want
  - Example: can cast a `void*` to `int`
  - Example2: can cast any `(A*)` to a `(B*)`
    - Even if `A` and `B` are unrelated
- You must be careful
- You must know what you are doing
- Hence, this can be error-prone

New C++ Cast Operators

- Four new cast operators
  - `static_cast`
  - `const_cast`
  - `dynamic_cast`
  - `reinterpret_cast`
- Basic syntax example

```
B b;
A a = static_cast<A>(b);
```

- They make programmer's intent more clear

static_cast and dynamic_cast

- `static_cast`
  - Basic cast operator as we know it (or almost)
  - Can change binary representation of converted expr.
  - For pointers to classes, checks types at compile time
    - Classes must only be related to each other
- `dynamic_cast`
  - Can only be used with pointers
  - Checks object types at runtime
  - Use this operator for casting pointers to objects within a class hierarchy
- Example: `cast_operators()` in `main.cc`

const_cast and reinterpret_cast

- `const_cast`
  - Only removes or adds `const` qualifier
  - We will talk about the `const` qualifier in a few lectures
- `reinterpret_cast`
  - Enables arbitrary pointer casts
  - Unsafe and not portable
  - At least it is clear that cast is dangerous
- No need to know these last two for cse303
- But I encourage you to experiment with them

Function Overriding

- Derived class can override parent member function
- It simply declares a member function with
  - Same name as function in parent class
  - Same parameters
  - Example: `toString`
- To access parent member function from derived class, use the scope resolution operator
  - `Property::toString()`
- What is the difference between overloading and overriding?

Virtual Functions

- Gotcha with method overriding
  - By default, the **invoked function is selected statically, at compile time based on pointer type**
- **To enable dynamic binding and dispatching, must declare a function to be virtual**
  - `virtual void toString2();`
  - Once a function is virtual, it remains virtual all the way down the class hierarchy
  - Nevertheless, declare it as virtual in all classes
- Examples: `overriding_catch()`

Virtual Destructor

- Make all destructors virtual
- Problem illustration (Y derives from X)

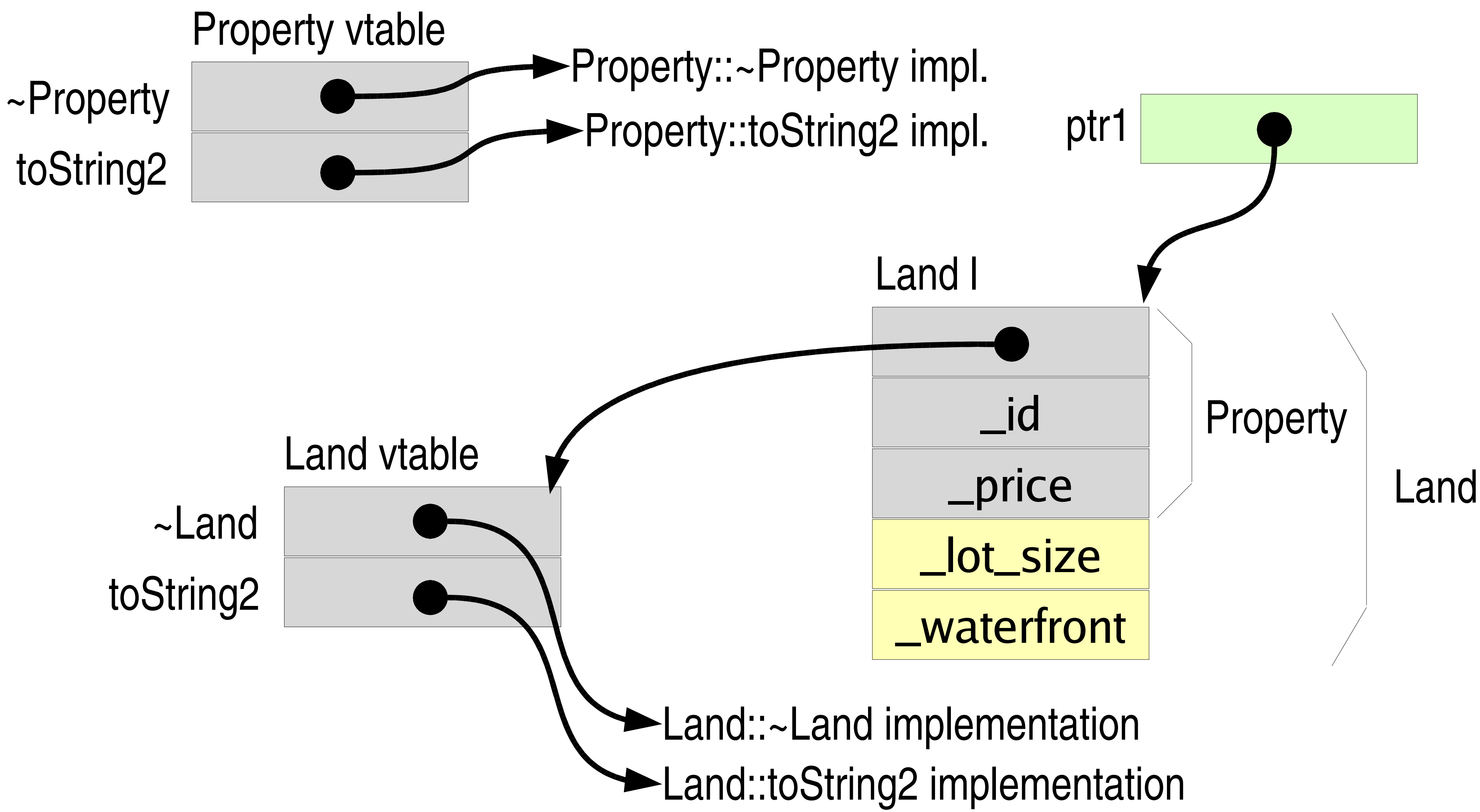
```
Y *ptrY = new Y();
X *ptrX = ptrY; // Implicit cast
delete ptrX;
```

- Without a virtual destructor, call to `delete ptrX` calls destructor for X, even if `ptrX` points to a subtype Y
- A virtual destructor solves this problem

Polymorphism

- Virtual member functions enable polymorphism
  - Accessing a virtual member function through a base-class pointer produces different results depending on runtime type of object
- To support polymorphism at runtime (i.e., dynamic binding), the C++ compiler builds several data structures at compile time
  - For each class that has at least one virtual function, it builds a virtual function table (vtable)

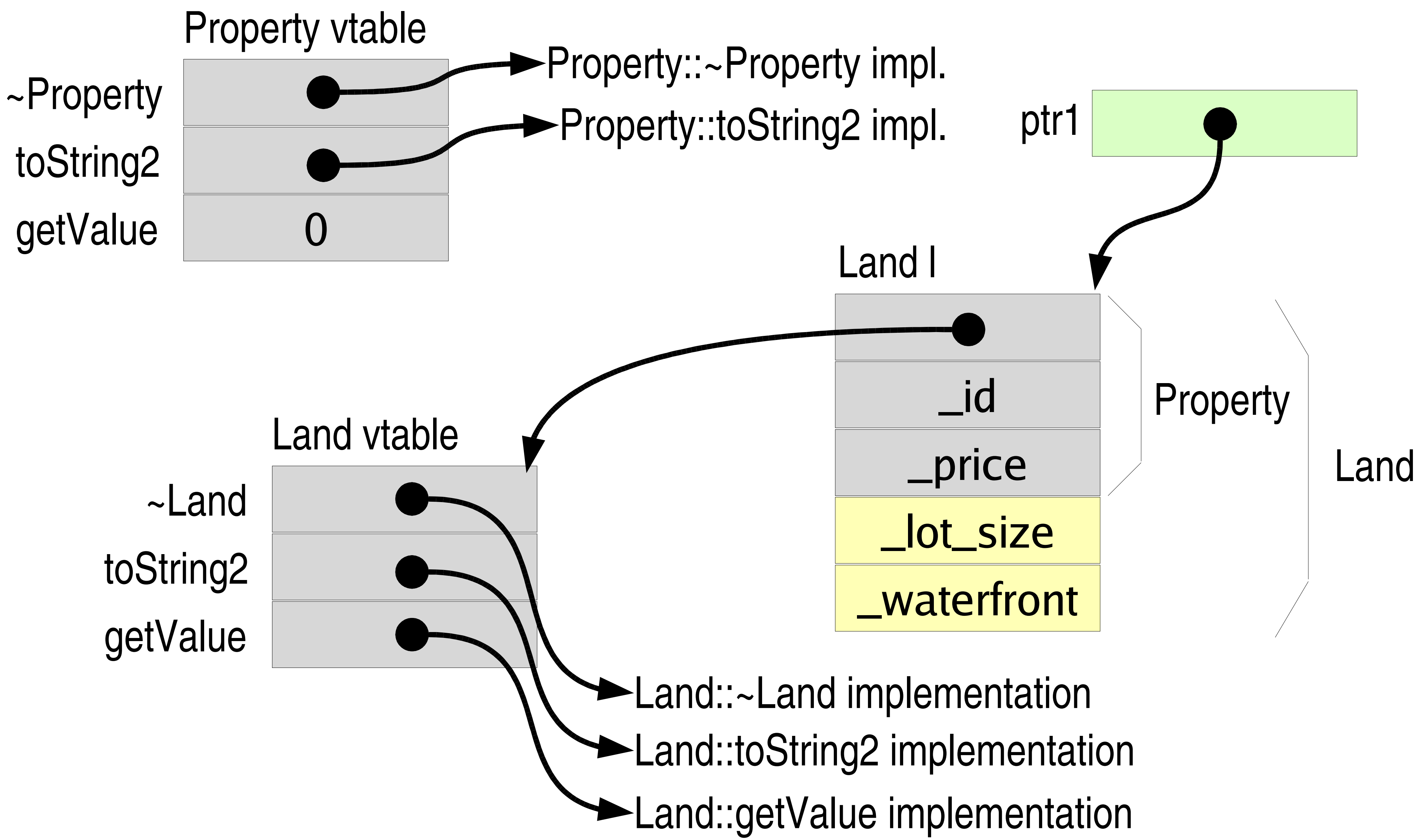
Virtual Function Table (vtable)



Abstract Classes

- In C++, there is no notion of interfaces
- Instead, we must use **abstract classes**
  - An abstract class cannot be instantiated
  - To make a class **abstract**, declare one member function as **pure virtual**
  - `virtual float getValue() = 0;`
- An abstract class can provide a partial implementation (ex: `Property` class)
- A class with **only pure virtual member functions** is called a **pure abstract class** (ex: Element class)
  - A pure abstract class constitutes a true interface

Virtual Function Table (vtable)



Pure Abstract Class Example

```
class Element {  // Pure abstract class
 public:
  virtual int compare(const Element& other) = 0;
  virtual void print() = 0;
};
// Using multiple inheritance
class House: public Property, public Element {
...
virtual int compare(const Element& other) { ... }
virtual void print() { ... }
...
};
```

C++ Inheritance Summary

- C++ distinguishes between
  - Static binding by default
  - Dynamic binding for virtual member functions
- C++ allows multiple inheritance
- No notion of interface
- Instead (pure) abstract classes
- Explicit casting with four types of operators

Introduction to Templates

- Motivation: often want to perform the same operations on different data types
- Example: storing data in a linked list
  - Solution 1: Create a new list class for each data type we want to store in a list
  - Solution 2: Force all data types to have a common ancestor X and create a list of X
  - Solution 3: Create a generic list class, and have the compiler use that generic class as a *template* to generate code for all the list classes we need

Readings

- Carefully study the code that accompanies today's lecture