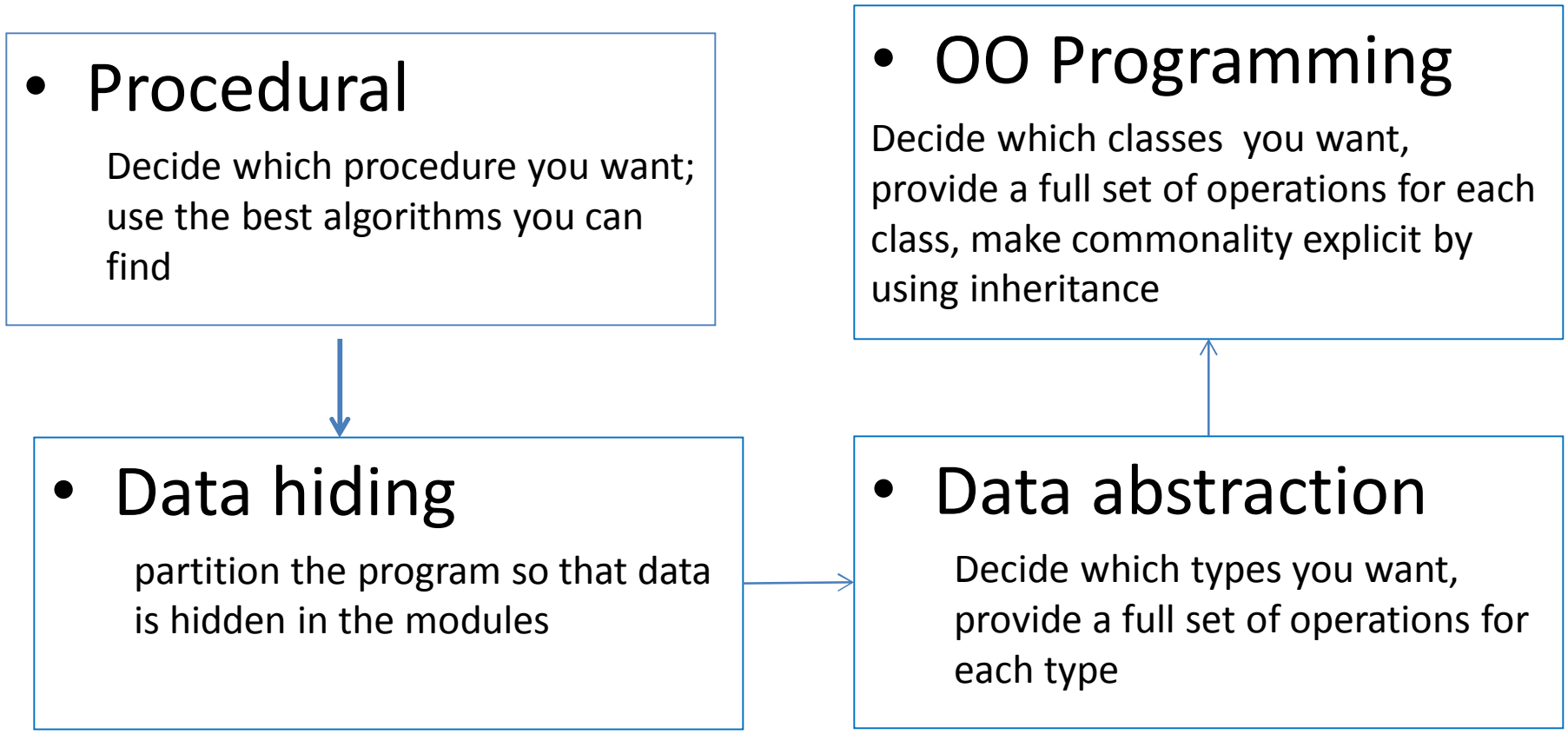
Object-oriented Programming Paradigm

Programming Paradigm

- Programming paradigm
  - A pattern that serves as a *school of thoughts* for programming of computers
- Programming technique
  - Related to an algorithmic idea for solving a particular class of problems
- Programming style
  - The way we express ourselves in a computer program, related to elegance or lack of elegance

The big picture in Stroustrup's view



Object-oriented Programming

Send messages between objects to simulate the temporal evolution of a set of real world phenomena

Three major principals are abstract data types, inheritance, and polymorphism.

The object-oriented paradigm has gained great popularity in the recent decade. The primary and most direct reason is undoubtedly the strong support of encapsulation and the logical grouping of program aspects. These properties are very important when programs become larger and larger.

Data Abstraction

- An abstract data type should satisfy the following conditions:
  - The representation of, and operations on, objects of the type are defined in a single syntactic unit.
  - The implementation details is hidden from the program units that use these objects.

Advantages of Data Abstraction

- Provide a way of organizing programs
- Improve the reliability
- Allow the implementation to be changed without affecting user code
- Promote two important design goals: low coupling and high cohesion.
  - Coupling refers to the degree of dependency between two modules. Cohesion refers to the degree to which a single module forms a meaningful unit.

Issues

- What is form of encapsulation?
- How access controls are provided?
- What kind of operations should be supported?
- Can ADT be parameterized?

Inheritance

- Inheritance allows new classes defined in terms of existing ones, i.e., by allowing them to inherit common parts

- Inheritance addresses two issues
  - Code reuse
    - All ADTs are independent, thus, ADTs are difficult to reuse
  - Program organization
    - All ADTs are at the same level, thus, it is impossible to organize a program to match the problem space.

- One disadvantage of inheritance for reuse:
  - Creates interdependencies among classes that complicate maintenance

Method overriding

- The process of redefining one or more methods defined in the parent class
- Changing the method implementation, but not the method signatures, including names, parameter lists, and return type
- Provide an operation that is specific for objects of the derived class

Method overloading

- The process of redefining one or more methods, not necessarily of parent class.
- Change method implementation and parameter lists, but not the method name and return type.
- **Restrict your use of overloaded method names on situations where the methods really are performing the same basic function with different data.**

Dynamic Binding and polymorphism

- A *polymorphic variable* can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants.

- When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method will be dynamic.

- Allows software systems to be more easily extended during both development and maintenance

Extensible

- A program is *extensible* if you can add new functionality by inheriting new data types from the common base class. The functions that manipulate the base-class interface will not need to be changed at all to accommodate the new classes.

An extended program

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
  virtual char* what() const {
    return "Instrument";
  }
  // Assume this will modify the object:
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identical function from before:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }
```

```
// Upcasting during array initialization:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

- The **adjust( )** function is *not* overridden for **Brass** and **Woodwind**. When this happens, the “closest” definition in the inheritance hierarchy is automatically used.
- The array **A[ ]** contains pointers to the base class **Instrument**, so upcasting occurs during the process of array initialization.
- In the call to **tune( )**, upcasting is performed on each different type of object, yet the desired behavior always takes place.

Abstract class

- An *abstract method* is one that does not include a definition (it only defines a protocol)
- Generally, an *abstract class* includes at least one abstract/virtual method
- An abstract class cannot be instantiated

Initialization: Constructors

```
 //: Cartoon.java
// Constructor calls during inheritance

class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing
constructor");
  }
}

public class Cartoon extends Drawing {
  Cartoon() {
    System.out.println("Cartoon
constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
} ///:~
```

The output for this program shows the automatic calls:

```
Art constructor
Drawing constructor
Cartoon constructor
```

```
//: Chess.java
// Inheritance, constructors and arguments

class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}

public class Chess extends BoardGame {
  Chess() {
    super(11);
    System.out.println("Chess constructor");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
  }
} ///:~
```

The output for this program

??

The final keyword

- Final data
  - With a primitive, **final** makes the *value* a constant, but with an object handle, **final** makes the handle a constant.
  - A field that is both **static** and **final** has only one piece of storage that cannot be changed.

The final keyword

- Final methods
  - Prevent methods to be overridden
  - Allow the compiler to turn any calls to that method into *inline* calls

The final keyword

- Final class
  - The class cannot be inherited
  - All methods in a **final** class are implicitly **final**, since there's no way to override them.

Public vs private inheritance

- public keyword → the public and protected members inherited from the base class are also public and protected in the derived class

- private keyword → public and protected members inherited from the base class become *private* in the in the derived class

```
class Single_linked_list{
  private:
     class Node{
         public:
         node *link;
         int contents;
     };
     node *head;
  public:
     Single_linked_list(){head = 0};
     void insert_at_head(int);
     void insert_at_end(int);
     int remove_at_head();
     int empty();
 };
```

```
class Stack2: private Single_linked_list{
    public stack2(){}
    void push(int value){
        Single_linked_list::insert_at_head(value);
    }

    int pop(){
        return Single_linked_list::remove_at_head();}

   Single_linked_list:: empty;
};
```

Public vs private inheritance example

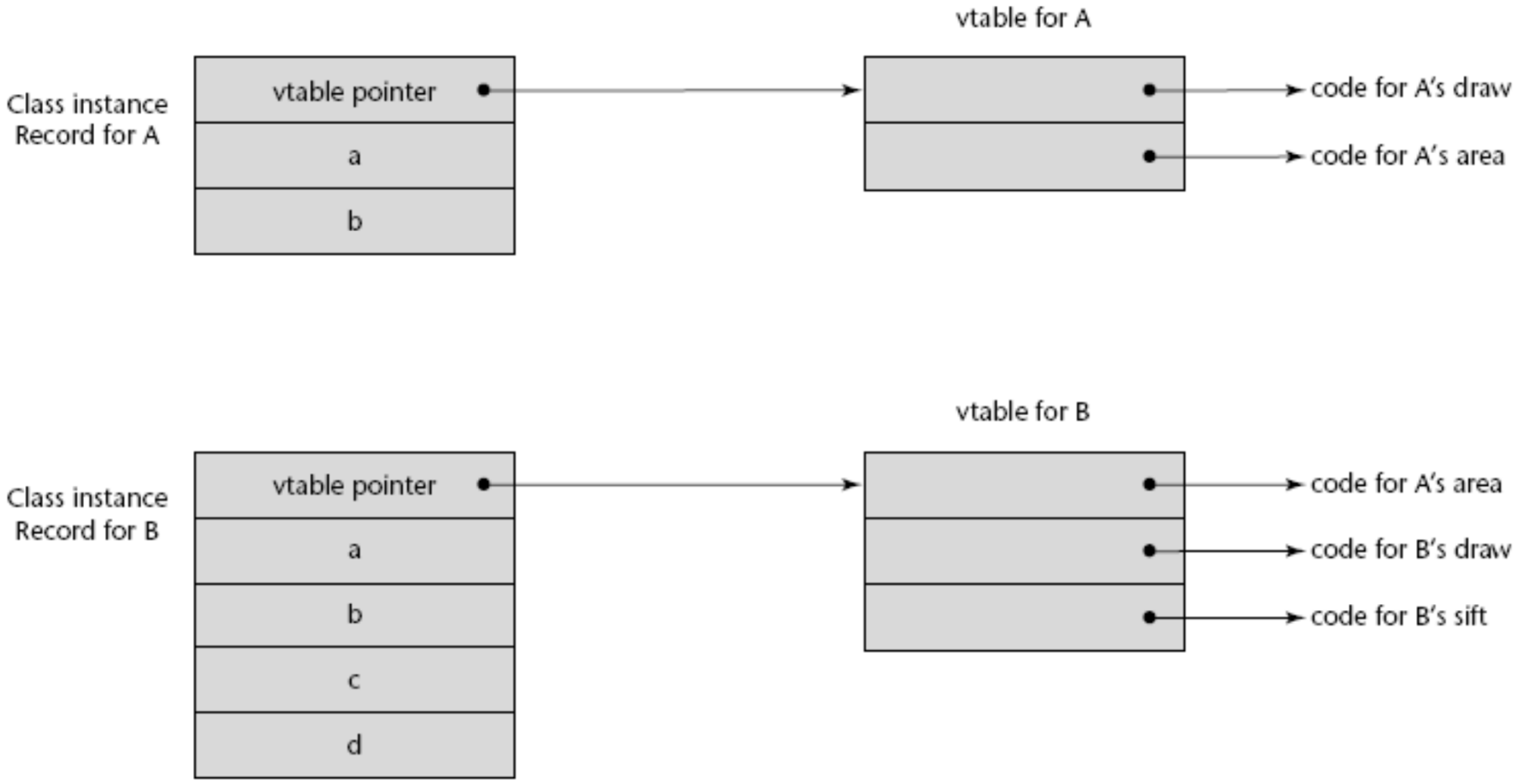


Figure 12.2

An example of the CIRs with single inheritance

How dynamic binding work?

Criticisms

"This semester Dan Licata and I are co-teaching a new course on functional programming for first-year prospective CS majors... Object-oriented programming is eliminated entirely from the introductory curriculum, because it is both anti-modular and anti-parallel by its very nature, and hence unsuitable for a modern CS curriculum. A proposed new course on object-oriented design methodology will be offered at the sophomore level for those students who wish to study this topic."

from ***Robert Harper, March 2011***