Basic OOP in C++

Overview

- Object Models, Classes, Inheritance, and Polymorphism
- Interface-based Design: Abstract Classes

References

- Gary J. Bronson: C++ for Engineers and Scientists. 3rd Edition. Thomson (2010)
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: C++ Primer. 5th Edition. Addison-Wesley (2013)

C++ Object Models

- C++ supports:
  - A value-based object model
  - A reference-based object model

The Value-based Object Model

- Value-based object model:
  - Objects are stored on the stack.
  - Object are accessed through object variables.
  - An object's memory is implicitly released.

Valued-based Objects

- Value-based objects look and feel like records (or structs):

```
Card AceOfDiamond( Diamond, 14 );

Card TestCard( Diamond, 14 );


cout << "The test card is " << TestCard.getName() << endl;
```

The Reference-based Object Model

- Reference-based object model:
  - Objects are stored on the heap.
  - Objects are accessed through pointer variables.
  - An object's memory must be explicitly released.

Reference-based Objects

- Reference-based objects require pointer variables and an explicit new and delete:

```
Card* AceOfDiamond = new Card( Diamond, 14 );
Card* TestCard = new Card( Diamond, 14 );

cout << "The test card is " << TestCard->getName() << endl;

delete AceOfDiamond;
delete TestCard;
```

Member Dereference

release memory

Inheritance

- Inheritance lets us define classes that model relationships among classes, sharing what is common, and specializing only that which is inherently different.
- Inheritance is
  - A mechanism for specialization
  - A mechanism for reuse
  - Fundamental to supporting polymorphism

An Account Class

Account.h

```
4   class Account
5   {
6   private:
7       unsigned long long fNumber;
8       double fBalance;
9
10  public:
11      Account( unsigned long long aNumber, double aBalance = 0.0 );
12      virtual ~Account() {}

14      bool deposit( double aAmount );

16      virtual bool withdraw( double aAmount );
17
18      double getBalance() const;
19  };
```

Line: 1 Column: 1 C++ Tab Size: 4

destructor (virtual)

virtual method (can be overridden)

- An account has a number(2^{64}- 1 values) and a balance.
- To create an account we need a number. Funds can be credited to an account once it has been created.

Virtual Member Functions

- To give a member function from a base class new behavior in a derived class, one overrides it.
- To allow a member function in a base class to be overridden, one must declare the member function virtual.
- Note, non-virtual member functions are resolved with respect to the declared type of the object.
- In Java all member functions are virtual.
- Explicitly defining a (virtual) destructor affects which special member functions the compiler synthesizes (C++-11).

Method Overriding

- Method overriding is an object-oriented programming mechanism that allows a subclass to provide a more specific implementation of a method, which is also present in one of its superclasses.
- The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has the same name, parameter signature, and return type as the method in the parent class. We say these methods belong to the same method family.
- Which (overridden) method is selected at runtime is determined by the receiver object used to invoke it. In general, the most recent definition is chosen, if possible.

Method Family

- A member function of a class always belongs to a specific set, called method family. If the elements of this set are virtual, then their invocation is governed by dynamic binding, a technique that makes polymorphism real.

Virtual withdraw Method

BankAccount.h

```
6  class BankAccount : public Account
7  {
8  private:
9      double fInterestRate;
10
11 public:
12     BankAccount( unsigned long long aNumber, double aBalance = 0.0, double aInterestRate = 0.0 );
13     ~BankAccount() override {}
14
15     bool withdraw( double aAmount ) override;
16
17     void setInterestRate( double aInterestRate );
18     double getInterestRate() const;
19
20     bool creditInterest();
21     bool applyServiceFee( double aFee );
22
23     // inherited from Account:
24     //  bool deposit( double aAmount );
25     //  bool withdraw( double aAmount );
26     //  double getBalance() const;
27 };
```

Line: 1 Column: 1 C++ Tab Size: 4

Overriding the withdraw Method

Using this-> is optional here as getBalance() is not virtual.

BankAccount.cpp

```
9   bool BankAccount::withdraw( double aAmount )
10  {
11      if ( (this->getBalance() - aAmount) > 0.0 )
12      {
13          return Account::withdraw( aAmount );
14      }
15      else
16          return false;
17  }
```

Line: 1 Column: 1 C++ Tab Size:

Call overridden method

Calling a Virtual Method

Main.cpp

```
8  int main()
9  {
10     BankAccount lAccount( 12345, 0.0, 0.5 );
11
12     cout << "Balance: " << lAccount.getBalance() << endl;
13
14     // dynamic method invocation
15
16     Account& lBankAccountReference = lAccount;
17
18     if ( lBankAccountReference.withdraw( 50.0 ) )
19     {
20         cout << "We got instant credit. Wow!" << endl;
21     }
22     else
23     {
24         cout << "The bank refused to give us money." << endl;
25     }
26
27     return 0;
28 }
```

Line: 1 Column: 1 C++ Tab Size: 4

Calls BankAccount::withdraw

Debug

```
Kamala:Debug Markus$ ./Inheritance
Balance: 0
The bank refused to give us money.
Kamala:Debug Markus$
```

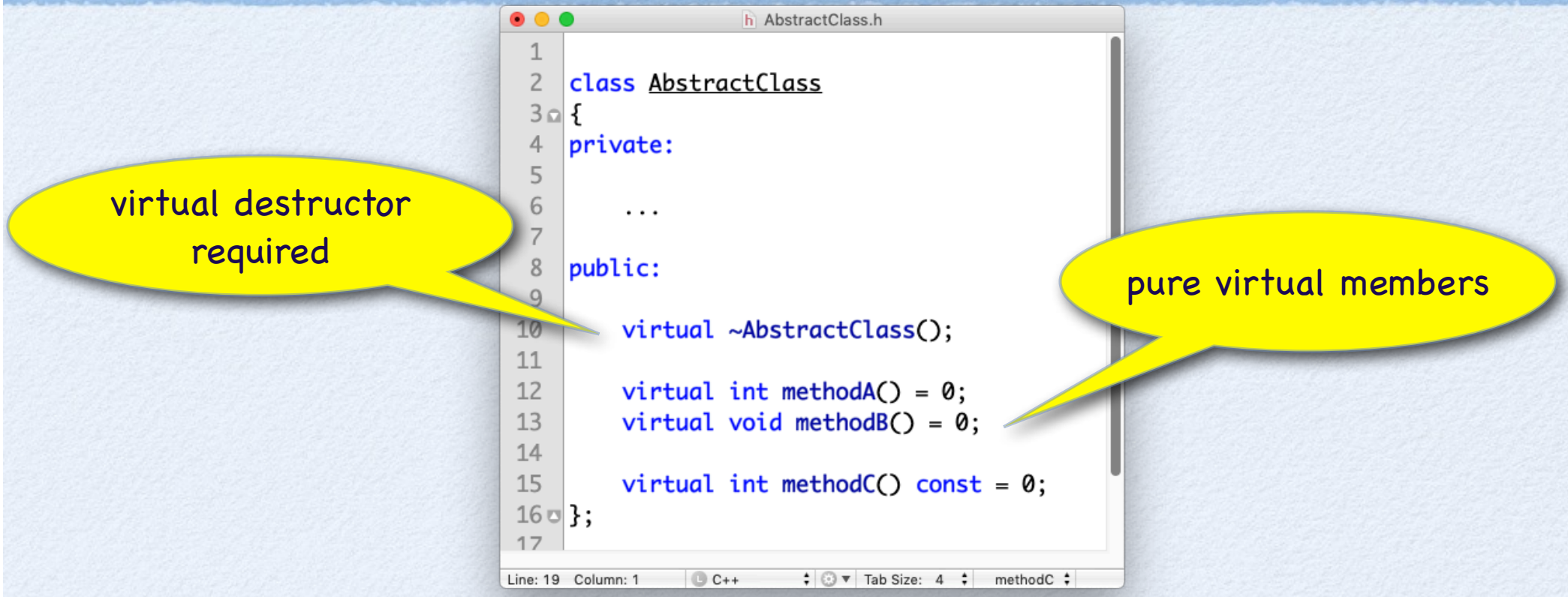
Interface-based Design

- The notion of interface-based design is an architectural pattern for implementing modular programming at the component level in an object-oriented programming language.
- From a conceptual standpoint, an interface is a contractual specification (i.e., protocol) between two parties. Ideally, if both parties adhere to the specification of the same interface, then correct interaction is guaranteed.
- In C++, we can use abstract classes as a means to create interfaces.
- Note, languages likes C# or Java support natively interfaces as built-in abstraction. Interfaces and abstract classes coexist in these languages.
- An abstract class can define behavior, whereas an interface cannot (except for class-level features in C# and Java).

Abstract Classes

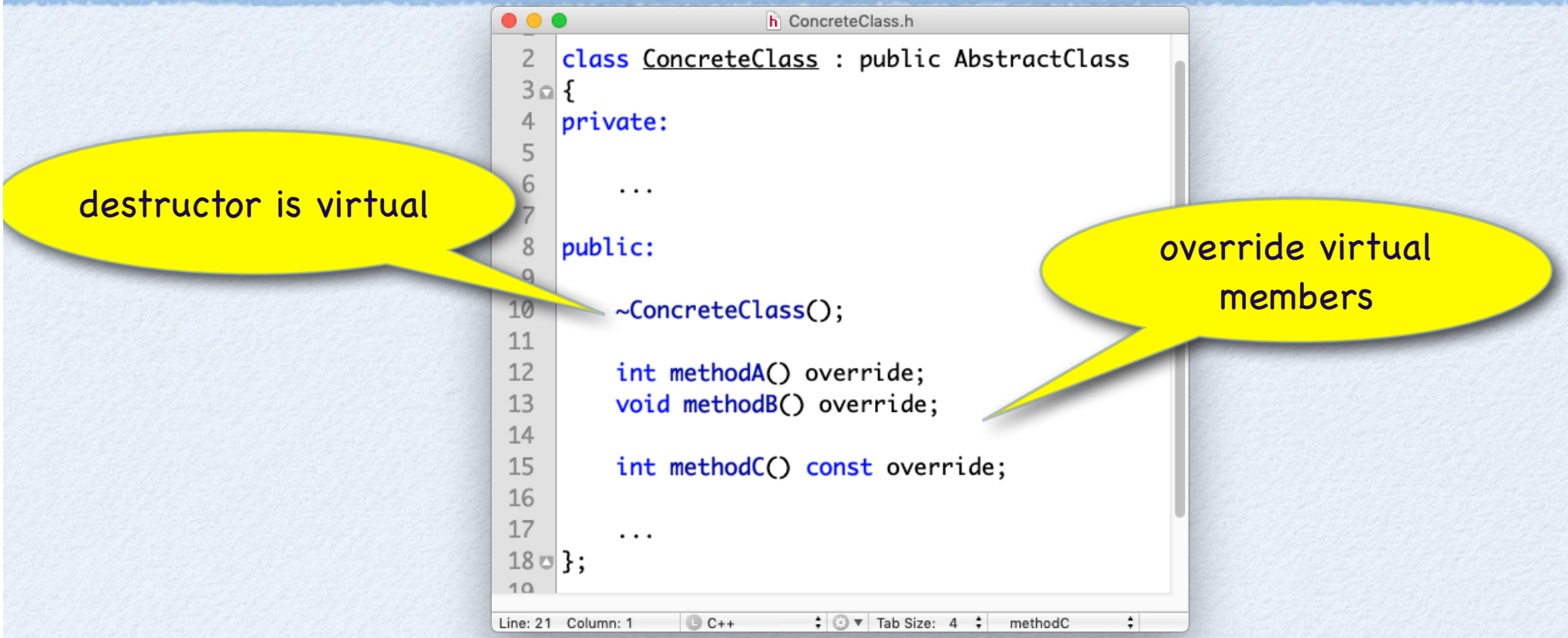
- A class is abstract if it contains one or more pure virtual member functions.
- An abstract class cannot be instantiated.
- Derived classes must provide definitions for the pure virtual member functions, or declare them as pure virtual itself.
- An abstract class requires a virtual destructor. If one declares a pure virtual destructor, a definition for it must be given in a subclass eventually.

Pure Virtual Member Functions



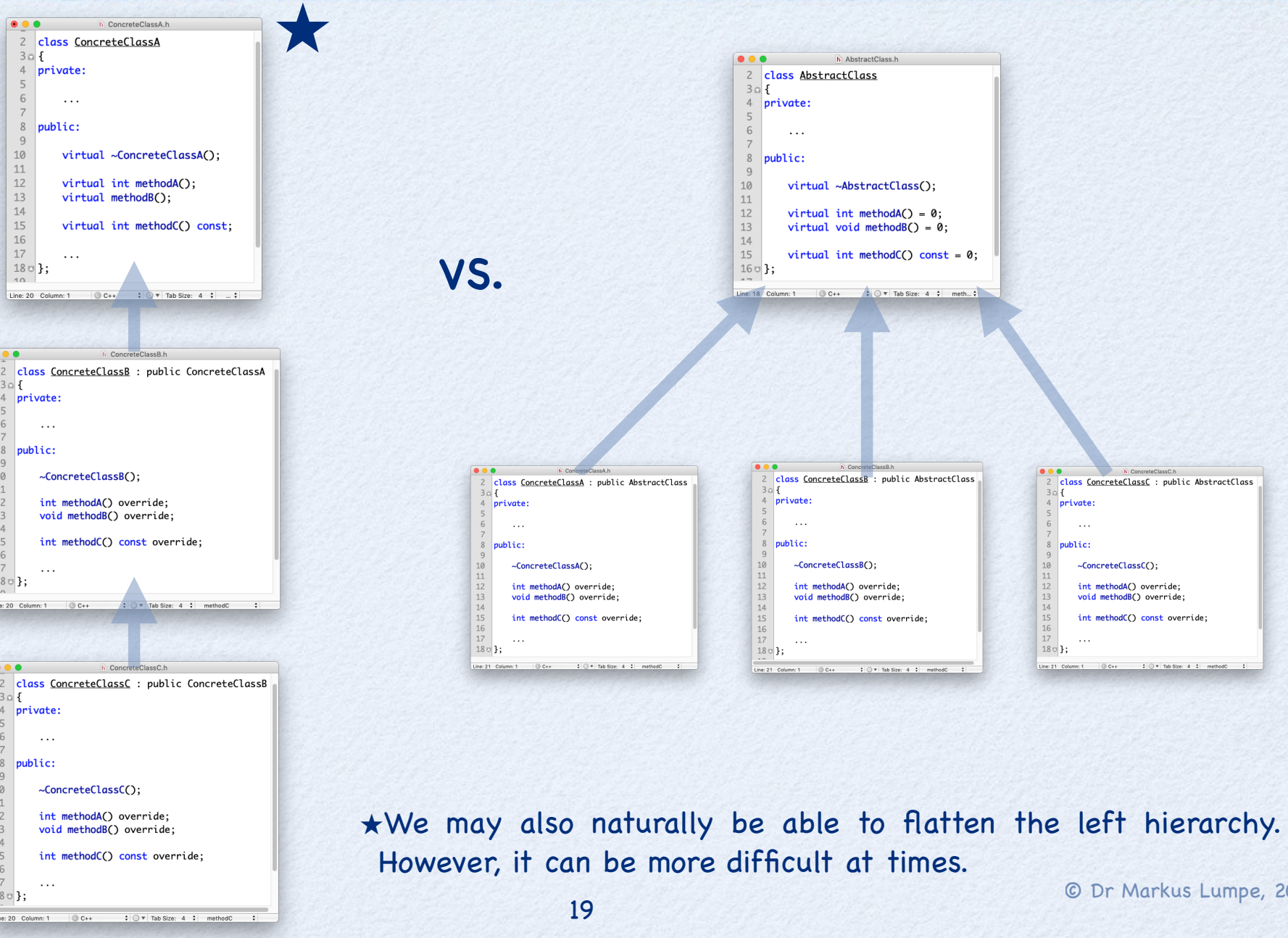
- Pure virtual member functions declare what a class provides, but defer the implementation of that behavior to subclasses.
- In C++, we use = 0 to denote that a member function is abstract, that is, it does not have an implementation.

Implementing Pure Virtual Member Functions



- In a subclass we override the pure virtual members.
- If all pure virtual members have been overridden, then we can create objects of this class.

Flatten Class Hierarchies



★We may also naturally be able to flatten the left hierarchy. However, it can be more difficult at times.