



CS49K Programming Languages

Chapter 10: Data Abstraction and Object-Orientation

LECTURE 14: CLASS HIERARCHY

DR. ERIC CHOU

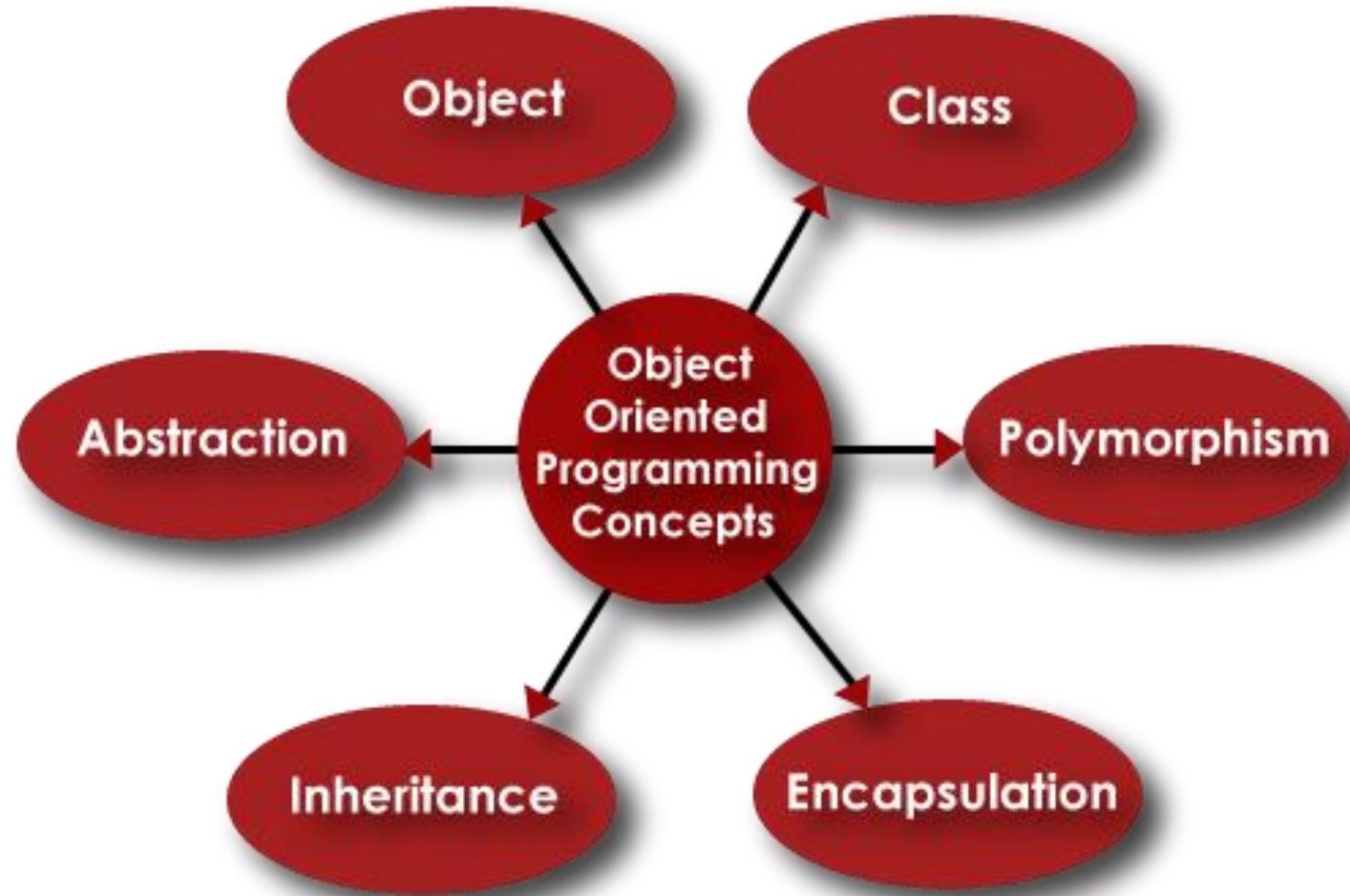
IEEE SENIOR MEMBER

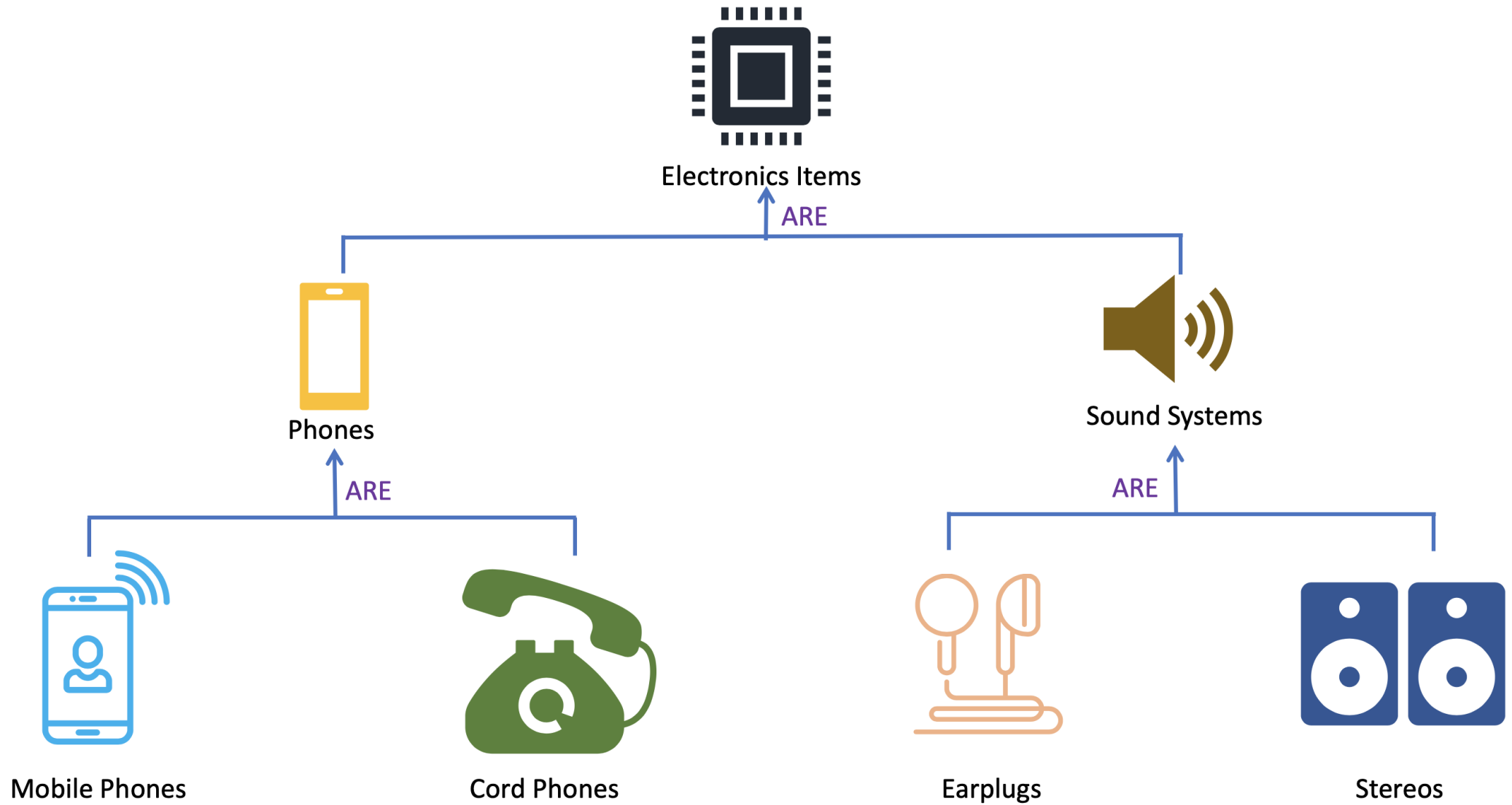
Objectives

- Inheritance
- Polymorphism
- Dynamic Binding
- Mix-In Inheritance
- Multiple Inheritance

Inheritance

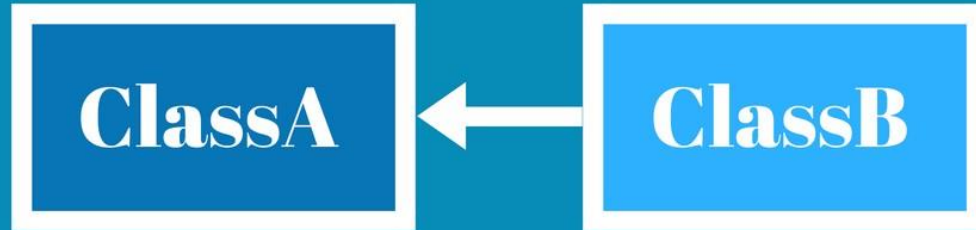
SECTION 1





Inheritance

"IS A"



```
11 class ClassB extends ClassA {
12     constructor() {
13         super();
14         this.propB = 'B';
15     }
16
17     methodB() {
18         return this.propB + super.methodA();
19     }
20 }
```

"HAS A" (composition)



```
12 class ClassB {
13     constructor() {
14         this.propB = 'B';
15         this.propA = new ClassA();
16     }
17
18     methodB() {
19         return this.propB + this.propA.methodA();
20     }
21 }
```

Method

Inheritance

Class

Also called parent class or base class.

When a class has two or more parent classes.

Multiple Inheritance

inherits properties of a parent

An instance of a class.

Class

Types of Inheritance

1

Single Inheritance

2

Multiple Inheritance

3

Multilevel Inheritance

4

Hierarchal Inheritance

5

Hybrid Inheritance

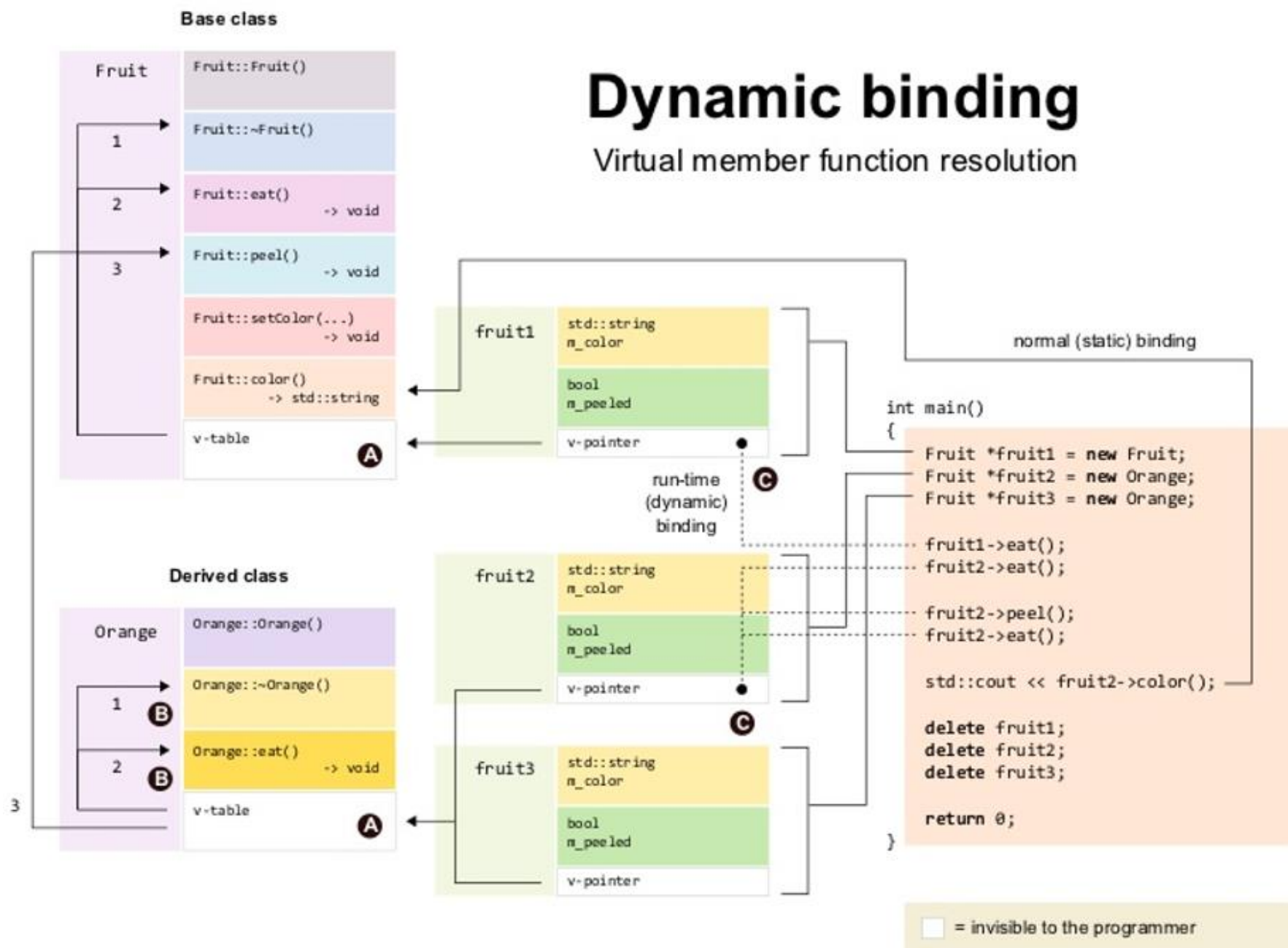
Dynamic Method Binding I

Polymorphic Methods

SECTION 2

Dynamic binding

Virtual member function resolution



Typical implementation

- A v-table is created for every class that declares one or more virtual functions, or overrides a virtual function. Ⓐ
- Such class is called a *polymorphic* class.
- Functions that override virtual functions from a parent class are adjusted in the v-table of the derived class. Ⓑ
- The function call is resolved at run-time, from the matching index in the v-table, pointed to by the actual object's v-pointer. Ⓒ
- Each object of a polymorphic class has its own v-pointer, but there is only one v-table per class.
- The v-tables and v-pointers are added by the compiler, and not visible to the programmer.

C++ Polymorphic Methods

```
class Fruit
{
public:
    Fruit()
        : m_color("fruit-colored"),
          m_peeled(false)
    {}

    virtual ~Fruit() {}
    virtual void eat();
    virtual void peel();
    void setColor(const std::string &color) { m_color = color; }
    const std::string &color() const { return m_color; }

protected:
    std::string m_color;
    bool m_peeled;
};

void Fruit::eat()
{
    std::cout << "Now eating this " << m_color << " fruit!\n";
}

void Fruit::peel()
{
    if (!m_peeled)
        m_peeled = true;
}
```

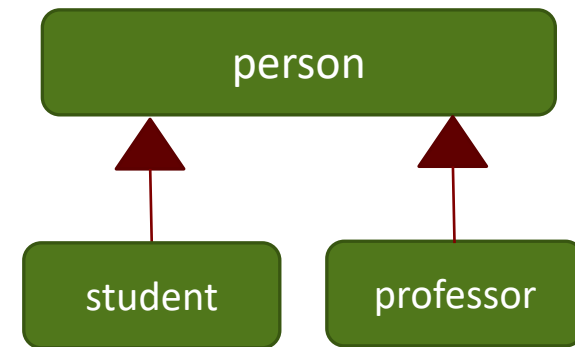
```
class Orange : public Fruit
{
public:
    Orange()
        : Fruit()
    {
        m_color = "orange";
    }
    void eat();
};

void Orange::eat()
{
    if (m_peeled) {
        Fruit::eat();
    } else {
        std::cout << "Can't eat orange before peeling it!\n";
    }
}
```

Subtype Polymorphism

- The ability to use a derived class in a context that expects its base class is called **subtype polymorphism**.

```
class person { ...  
class student : public person { ...  
class professor : public person { ...  
student s;  
professor p;  
...  
person *x = &s;  
person *y = &p;
```



Subtype Polymorphism

Base Method:

```
void person::print_mailing_label() { ...
```

Polymorphic Methods:

```
s.print_mailing_label();    // i.e., print_mailing_label(s)  
p.print_mailing_label();    // i.e., print_mailing_label(p)
```

Static and Dynamic Method Binding:

```
s.print_mailing_label();    // student::print_mailing_label(s)  
p.print_mailing_label();    // professor::print_mailing_label(p)
```

Easy to determining which method to use.

```
x->print_mailing_label();    // ??  
y->print_mailing_label();    // ??
```

x, y are object pointer of person type.

Which print_mailing_label() to be run will have to be determined at run-time. [\[Overriding of Methods\]](#)

Semantics and Performance

- The principal argument against static method binding is that the static approach denies the derived class control over the consistency of its own state.
- Suppose, for example, that we are building an I/O library that contains a `text_file` class:

```
class text_file {  
    char *name;  
    long position;           // file pointer  
public:  
    void seek(long whence);  
    ...  
};
```

Semantics and Performance

- Now suppose we have a derived class **read_ahead_text_file**:

```
class read_ahead_text_file : public text_file {  
    char *upcoming_characters;  
public:  
    void seek(long whence);    // redefinition  
    ...  
};
```

- The code for **read_ahead_text_file::seek** will undoubtedly need to change the value of the cached **upcoming_characters**.
- If the method is not dynamically dispatched, we cannot guarantee that this will happen: if we pass a **read_ahead_text_file** reference to a subroutine that expects a **text_file** reference as argument, and if that subroutine then calls **seek**, we'll get the version of **seek** in the base class.

Virtual and Non-Virtual Methods

A virtual function or virtual method is an inheritable and overridable function or method for which dynamic dispatch is facilitated.

- Calls to virtual methods are dispatched to the appropriate implementation at the run time, based on the class of the object, rather than the type of the reference. [\[dynamic binding using vtable\]](#)
- In C++ and C#, the keyword virtual prefixes the subroutine declaration:

```
class person {  
public:  
    virtual void print_mailing_label();  
    ...  
}
```

Ada 95

Class-Wide Types in Ada 95

- Ada 95 programmer associates it with certain references. In our mailing label example, a formal parameter or an access variable(pointer) can be declared to be of the class-wide type **person'Class**, in which case all calls to all methods of that parameter or variable will be dispatched based
- on the class of the object to which it refers:

Ada 95

Class-Wide Types in Ada 95

```
type person is tagged record ...
type student is new person with ...
type professor is new person with ...

procedure print_mailing_label(r : person) is ...
procedure print_mailing_label(s : student) is ...
procedure print_mailing_label(p : professor) is ...

procedure print_appropriate_label(r : person'Class) is
begin
    print_mailing_label(r);
    -- calls appropriate overloaded version, depending
    -- on type of r at run time
end print_appropriate_label;
```

Abstract Classes

C++ Refers to Abstract Methods as Pure Virtual Methods

Abstract Method in Java and C#:

```
abstract class person {  
    ...  
    public abstract void print_mailing_label();  
    ...  
}
```

Abstract Method in C++:

The notation in C++ is somewhat less intuitive: one follows the subroutine declaration with an “assignment” to zero:

```
class person {  
    ...  
public:  
    virtual void print_mailing_label() = 0;  
    ...  
}
```

Note: In C++, Abstract Method is a method that will be inherited and **MUST** be overridden by some concrete class. Virtual Method is a method that can be inherited and can be overridden. Therefore, abstract method is called **PURE** virtual method.

Abstract Classes

C++ Refers to Abstract Methods as Pure Virtual Methods

Abstract Class:

- No Instantiation.
- Serve as base for concrete classes

Regardless of declaration syntax, a class is said to be abstract if it has at least one abstract method.

It is not possible to declare an object of an abstract class, because it would be missing at least one member.

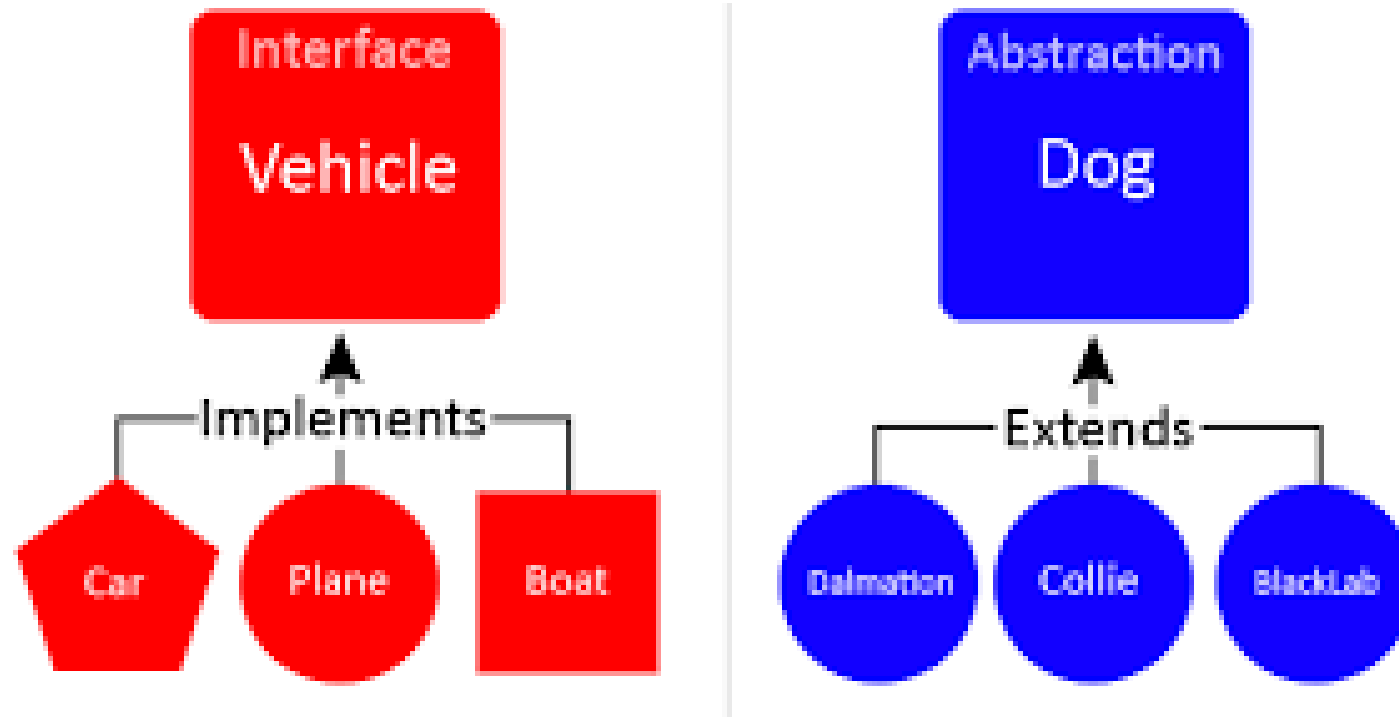
Application of Abstract Class:

Use Interface with Abstract to form Abstract Adapter Class.

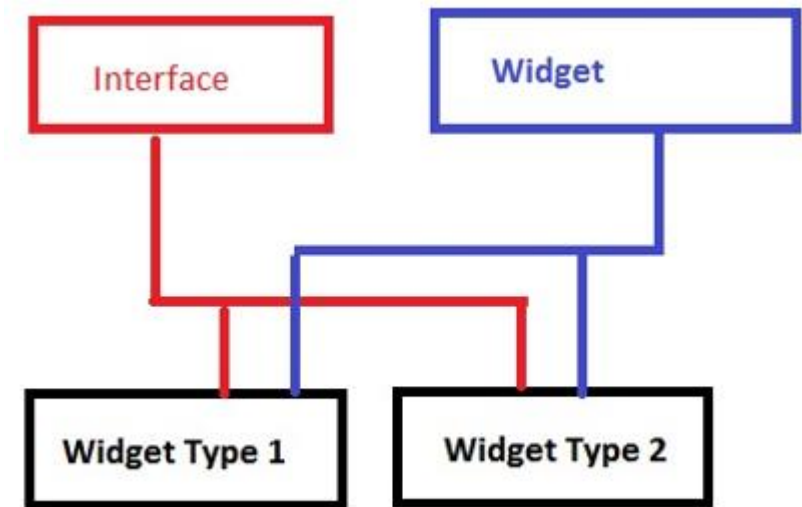
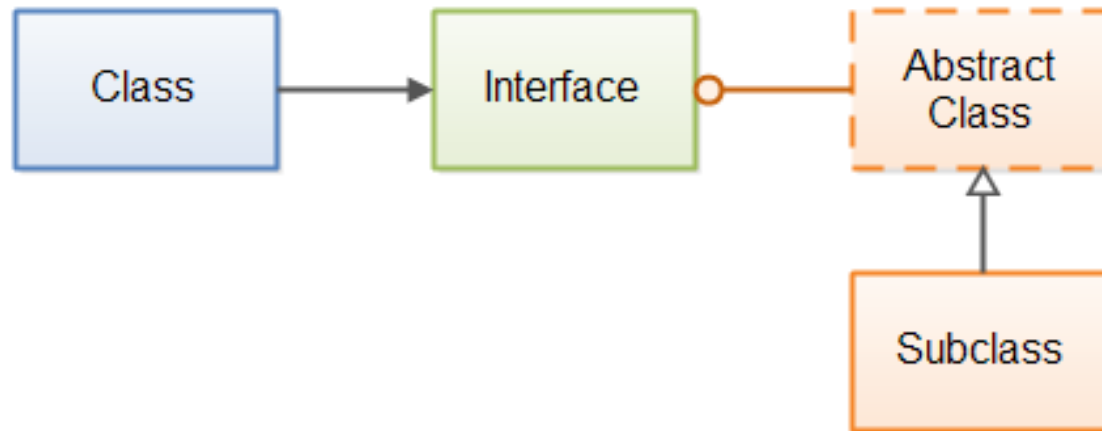
Shield out unneeded methods from interfaces or upper classes.

Abstract Classes

Interfaces vs. Abstract Classes



Abstract Classes



Dynamic Method Binding II

Finding the Method Implementation

SECTION 3

Member Lookup

Memory Organization for Dynamic Binding

- With static method binding (as in **Simula**, **C++**, **C#** or **Ada 95**), the compiler can always tell which version of a method to call, based on the type of the variable being used. [if no virtual method declaration]
- With dynamic method binding, the object referred to by a reference or pointer variable must contain sufficient information to allow the code generated by the compiler to find the right version of method at run time.
- **Technique: vtable** – virtual method table for the object's class.

Implementation of a Virtual Method Call

- Suppose that the this (self) pointer for methods is passed in register r1, that m is the third method of class foo, and that f is a pointer to an object of class foo.
- Then the code to call f->m() looks something like this:

```

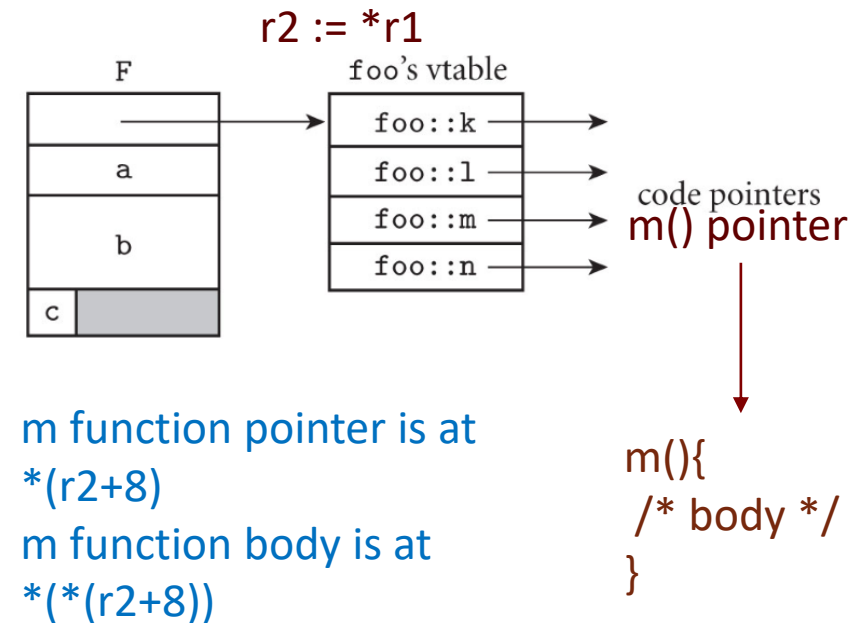
r1 := f
r2 := *r1           -- vtable address
r2 := *(r2 + (3-1) × 4)  -- assuming 4 = sizeof (address)
call *r2

```

```

class foo {
    int a;
    double b;
    char c;
public:
    virtual void k( ...
    virtual int l( ...
    virtual void m();
    virtual double n( ...
    ...
} F;

```



Implementation of Single Inheritance

- If bar is derived from foo, we place its **additional fields** at the end of the “record” that represents it.
- We create a **vtable** for bar by copying the **vtable** for foo, replacing the entries of any virtual methods overridden by bar, and appending entries for any virtual methods declared in bar.
- If we have an object of class **bar** we can assign its address into a variable of type **foo***:

```
class bar : public foo {
    int w;
public:
    void m() override;
    virtual double s( ...
    virtual char *t( ...
    ...
} B;
```

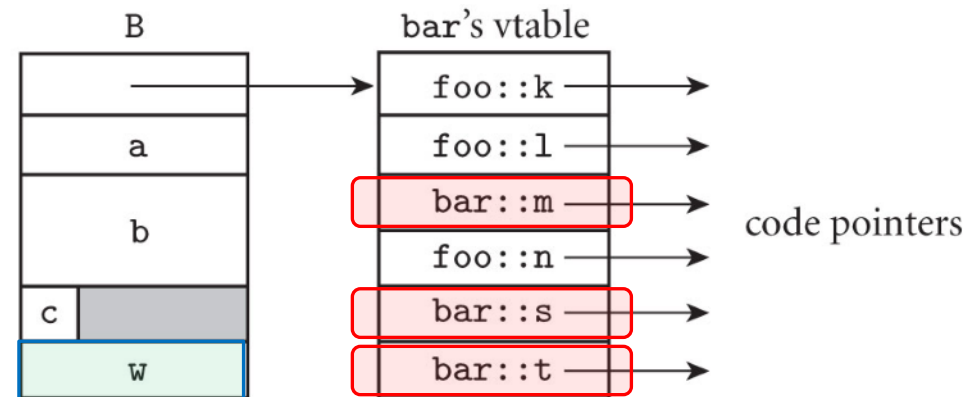


Figure 10.4 Implementation of single inheritance. As in Figure 10.3, the representation of object B begins with the address of its class’s **vtable**. The first four entries in the table represent the same members as they do for foo, except that one—**m**—has been overridden and now contains the address of the code for a different subroutine. Additional fields of bar follow the ones inherited from foo in the representation of B; additional virtual methods follow the ones inherited from foo in the **vtable** of class bar.

Dynamic Binding for Downward Assignment and Upward Assignment

- In C++ (as in all statically typed object-oriented languages), the compiler can verify the type correctness of this code statically.
- It does not know what the class of the object referred to by **q** will be at run time, but it knows that it will either be **foo** or something derived (directly or indirectly) from **foo**
- This ensures that it will have all the members that may be accessed by foo-specific code.

```
class foo { ...  
class bar : public foo { ...  
...  
foo F;  
bar B;  
foo* q;  
bar* s;  
...  
q = &B;    // ok; references through q will use prefixes  
           // of B's data space and vtable  
s = &F;    // static semantic error; F lacks the additional  
           // data and vtable entries of a bar
```

Casts in C++

- C++ allows “backward” assignments by means of a **dynamic_cast** operator:

```
s = dynamic_cast<bar*>(q);      // performs a run-time check
```

- If the run-time check fails, s is assigned a null pointer.
- For backward compatibility C++ also supports traditional C-style casts of object pointers and references:

```
s = (bar*) q;                  // permitted, but risky
```

- With a **C**-style cast it is up to the programmer to ensure that the actual object involved is of an appropriate type: no dynamic semantic check is performed.

Dynamic Binding in Java and C#

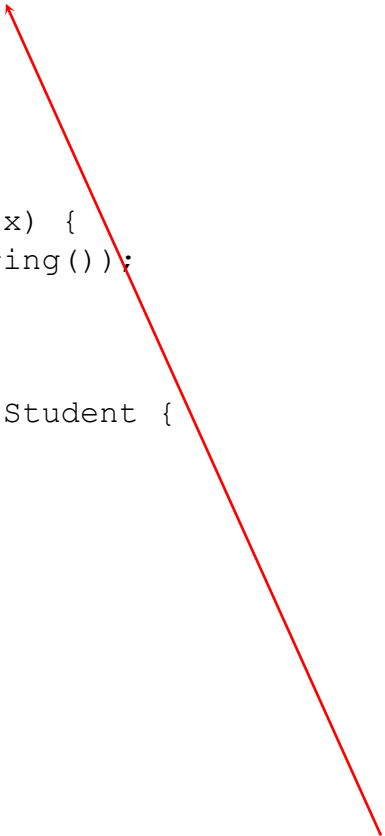
```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```



An object of a subtype can be used wherever its supertype value is required. This feature is known as **polymorphism**.

When the method **m(Object x)** is executed, the argument **x**'s **toString** method is invoked. **x** may be an instance of **GraduateStudent**, **Student**, **Person**, or **Object**. Classes **GraduateStudent**, **Student**, **Person**, and **Object** have their own implementation of the **toString** method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as **dynamic binding**.

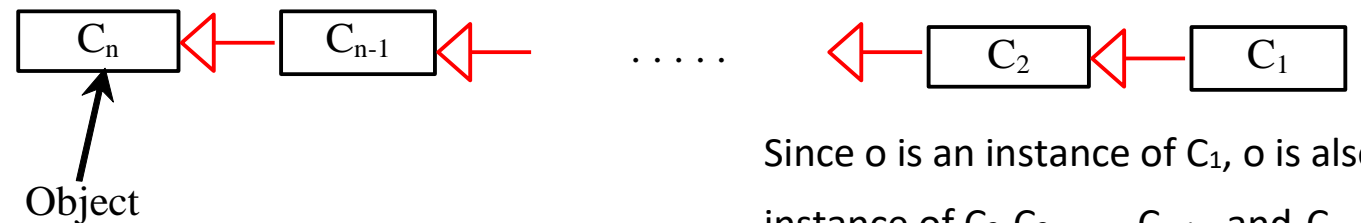
Note: Method **m** takes a parameter of the **Object** type. You can invoke it with any object.

Dynamic Binding in Java and C#

Dynamic binding works as follows:

- Suppose an object o is an instance of classes C_1, C_2, \dots, C_{n-1} , and C_n , where C_1 is a subclass of C_2 , C_2 is a subclass of C_3 , ..., and C_{n-1} is a subclass of C_n .
- That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the Object class.
- If o invokes a method p , the JVM searches the implementation for the method p in **C_1, C_2, \dots, C_{n-1} and C_n , in this order**, until it is found.
- Once an implementation is found, the search stops and the first-found implementation is invoked.

Java takes Downward cast in C-style:
`s = (bar) q;`



Since o is an instance of C_1 , o is also an instance of C_2, C_3, \dots, C_{n-1} , and C_n

Reverse Assignment in Eiffel and C#

? dynamic, = assignment

Eiffel has a reverse assignment operator, `?=`, which (like the C++ `dynamic_cast`) assigns an object reference into a variable if and only if the type at run time is acceptable:

```
class foo ...
class bar inherit foo ...
...
f : foo
b : bar
...
f := b      -- always ok
b ?= f      -- reverse assignment: b gets f if f refers to a bar object
              -- at run time; otherwise b gets void
```


Dynamic Method Binding III

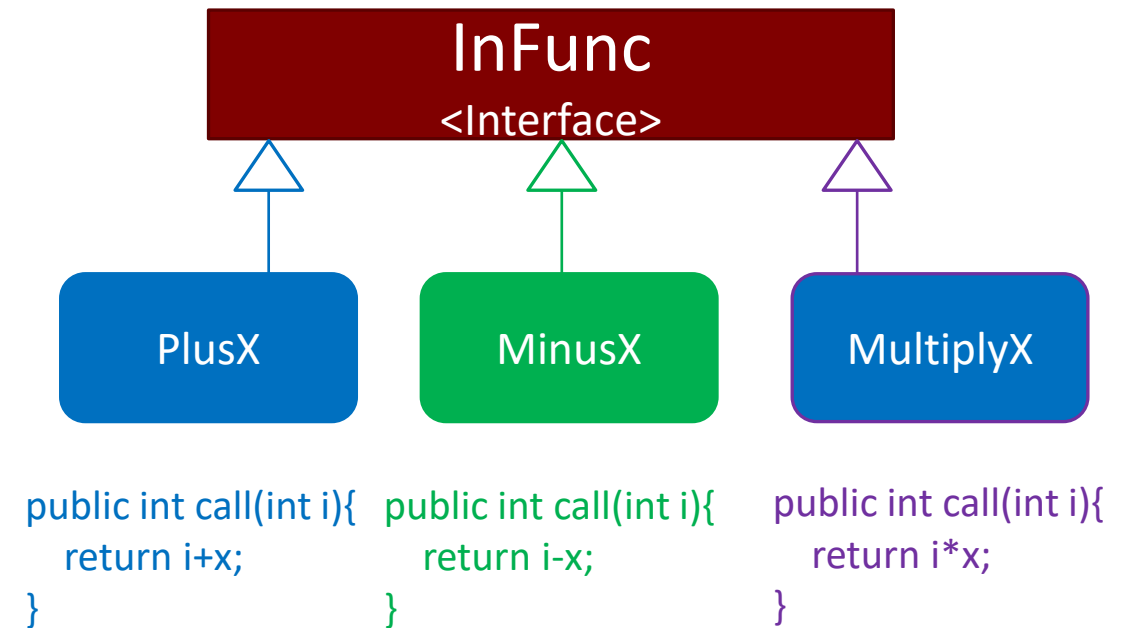
Object Closures

SECTION 4

Review Object Closures [Sec. 3.6.3]

Object Closures in Java

```
interface IntFunc {  
    public int call(int i);  
}  
  
class PlusX implements IntFunc {  
    final int x;  
    PlusX(int n) { x = n; }  
    public int call(int i) { return i + x; }  
}  
  
...  
IntFunc f = new PlusX(2);  
System.out.println(f.call(3));           // prints 5
```



Object Closures

Virtual Methods in an Object Closure [Using Polymorphism to pass Methods]

Motivation: Object closures can be used in an object-oriented language to achieve the same effect as subroutine closures in a language with nested subroutines: namely, to encapsulate a method with context for later execution. [Sec. 3.6.3, 9.3]

It should be noted that this mechanism relies on **dynamic method binding**.

Note: Any object derived from `un_op<int>` can be passed to `apply_to_A`. The “right” function will always be called because `operator()` is virtual.

```
template<class T>
class un_op {
public:
    virtual T operator()(T i) const = 0;
};

class plus_x : public un_op<int> {
    const int x;
public:
    plus_x(int n) : x(n) { }
    virtual int operator()(int i) const { return i + x; }
};

void apply_to_A(const un_op<int>& f, int A[], int A_size) {
    int i;
    for (i = 0; i < A_size; i++) A[i] = f(A[i]);
}

...
int A[10];
apply_to_A(plus_x(2), A, 10);
```

Diagram Annotations:

- abstract**: Points to the `virtual T operator()(T i) const = 0;` line.
- An Closure**: Points to the `plus_x(2)` argument in the `apply_to_A` call.
- Each A[i] increased by 2**: Points to the `A[i] = f(A[i]);` line, indicating the result of the closure.

Encapsulating Arguments

A particularly useful idiom for many applications is to encapsulate a method and its arguments in an object closure for later execution.

- Suppose, for example, that we are writing a discrete event simulation, as described in Section C-9.5.4.
- We might like a general mechanism that allows us to schedule a call to an arbitrary subroutine, with an arbitrary set of parameters, to occur at some future point in time.
- If the subroutines we want to have called vary in their numbers and types of parameters, we won't be able to pass them to a general-purpose **schedule_at** routine.
- We can solve the problem with object closures, as shown in **Figure 10.5 [Next Slides]**.

```
class fn_call {
```

Polymorphic method

```
public:
    virtual void operator()() = 0;
```

```
};
```

```
void schedule_at(fn_call& fc, time t) {
```

Connector function

```
}
```

```
...
```

```
void foo(int a, double b, char c) {
```

Implementation Body function

```
}
```

```
class call_foo : public fn_call {
```

```
    int arg1;
```

```
    double arg2;
```

```
    char arg3;
```

```
public:
```

```
    call_foo(int a, double b, char c) :    // constructor
```

```
        arg1(a), arg2(b), arg3(c) {
```

```
        // member initialization is all that is required
```

```
    }
```

```
    void operator()() {
```

```
        foo(arg1, arg2, arg3);
```

```
    }
```

```
};
```

```
...
```

```
call_foo cf(3, 3.14, 'x');           // declaration/constructor call
```

```
schedule_at(cf, now() + delay);
```

```
    // at some point in the future, the discrete event system
```

```
    // will call cf.operator()(), which will cause a call to
```

```
    // foo(3, 3.14, 'x')
```

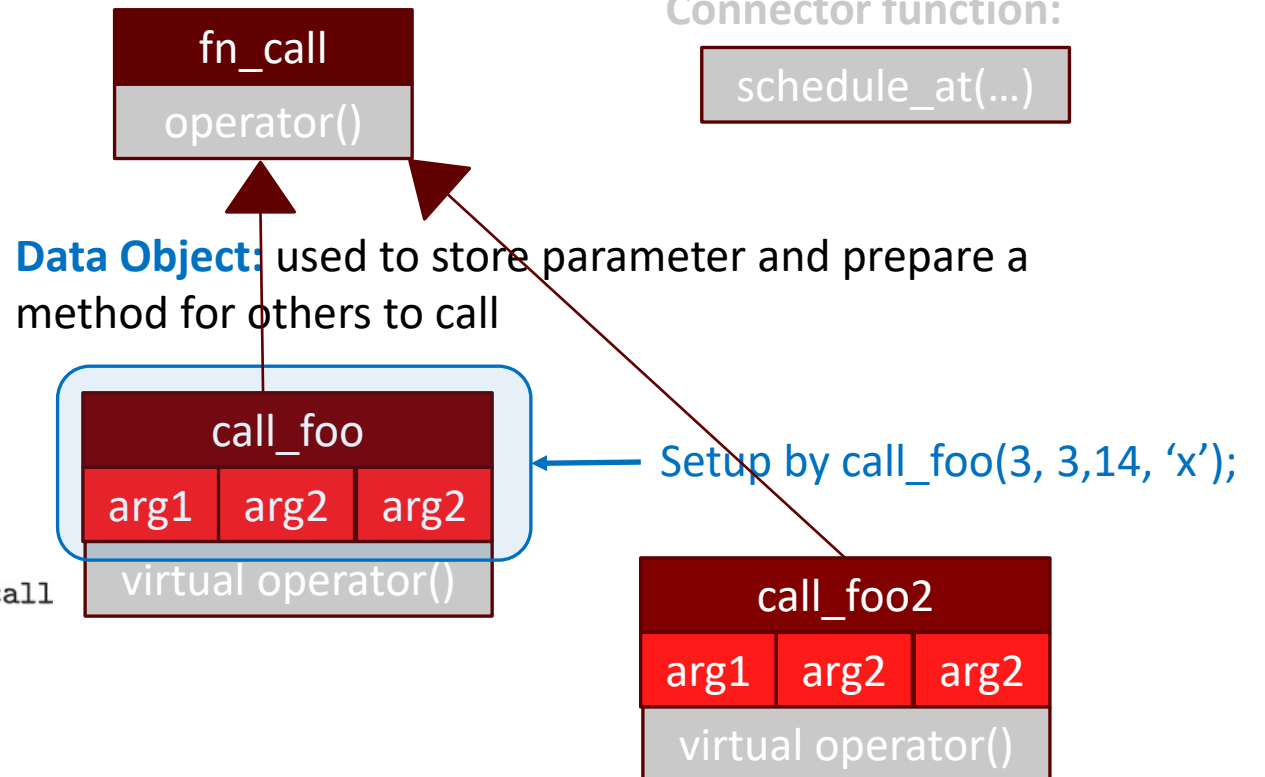
Figure 10.5 Subroutine pointers and virtual methods. Class `call_foo` encapsulates a subroutine pointer and values to be passed to the subroutine. It exports a parameter-less subroutine that can be used to trigger the encapsulated call.

Super class `fn_call` for polymorphism:

Using polymorphic method to allow different functional calls.

Connector function:

`schedule_at(...)`



C++11 Standard Object Closure Encapsulating Arguments

- Function **schedule_at** would then be defined to take an object of **class std::function<void()>** (function object encapsulating a function to be called with zero argument) at its first parameter.
- Object **cf**, which Figure 10.5 passes in that first parameter position would be declared as

```
std::function<void()> cf = std::bind(foo, 3, 3.14, 'x');
```

- Figure 10.5 example can be rewritten as:

```
std::function<void()> cf = std::bind(foo, 3, 3.14, 'x');  
schedule_at(cf, now() + delay);
```

Function Object for Table of Functions

Class `std::function<>`, declared in `<functional>`, provides polymorphic wrappers that generalize the notion of a function pointer. This class allows you to use *callable objects* (functions, member functions, function objects, and lambdas; see Section 4.4, page 54) as first-class objects.

For example:

```
void func (int x, int y);

// initialize collections of tasks:
std::vector<std::function<void(int,int)>> tasks;
tasks.push_back(func);
tasks.push_back([] (int x, int y) {
    ...
});
```

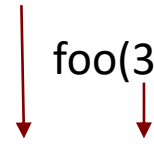
```
// call each task:
for (std::function<void(int,int)> f : tasks) {
    f(33,66);
}
```

To call each task, you could also simply call:

```
// call each task:
for (auto f : tasks) {
    f(33,66);
}
```

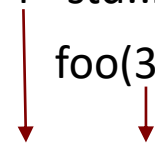
Function Wrapper Class(Object):

1. Using binding method to encapsulate arguments
`std::function<void()> f= std::bind(foo, 3, 3.14, 'x');`


foo(3, 3.14, 'x');
Call by f(3, 3.14, 'x');

Function Wrapper Class(Object) with Parameters:

2. `std::function<void(int, int, char)> f= std::bind(foo, 3, 3.14, 'x');`


foo(3, 3.14, 'x');
Call by f(3, 3.14, 'x');

Note: Two ways are equivalent

Using Functional Wrapper Class std::<function>

Demo Program: testfunc.zip (Download, unzip and import into IDE)
testfunc2.zip (Not shown here, Simpler)

```
1 #include <iostream>
2 #include <functional>
3
4 using namespace std;
5
6 void func(){
7     cout << "I am using Binding" << endl;
8 }
9
10 int func2(int x, int y){
11     return x+y;
12 }
13 class F {
14 public:
15     int func3(int x, int y);
16     int func4(int x, int y);
17 };
18 int F::func3(int x, int y){
19     return x+y;
20 }
21 int F::func4(int x, int y){
22     return x-y;
23 }
24
25 int main()
26 {
27     std::function<void()> f1 = std::bind(func);
28     f1();
29     std::function<int(int, int)> f2 = std::bind(func2, 3, 5);
30     cout << "3 + 5 = " << f2(3, 5) << endl;
31     F ff;
32     using namespace std::placeholders;
33     std::function<int(int, int)> f3 = std::bind(&F::func3, ff, _1, _2);
34     cout << "6 + 8 = " << f3(6, 8) << endl;
35     f3 = std::bind(&F::func4, ff, _1, _2); // rebinding
36     cout << "6 - 8 = " << f3(6, 8) << endl;
37     return 0;
38 }
39
```

func1: function without return value and parameter
func2: function with return value and parameter
func3: member function in a class and object
func4: rebinding member function for the same wrapper.

_1, _2: parameter 1 and parameter 2

"C:\Eric_Chou\CppDev\ProgrammingLanguages\chapter 10\testfunc\testfunc\bin\Debug\testfunc.exe"

I am using Binding

3 + 5 = 8

6 + 8 = 14

6 - 8 = -2

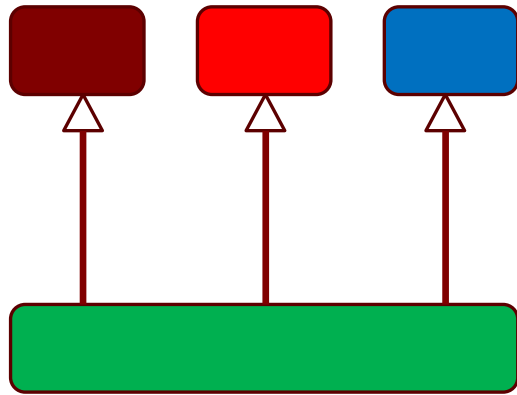
Process returned 0 (0x0) execution time : 0.014 s

Press any key to continue.

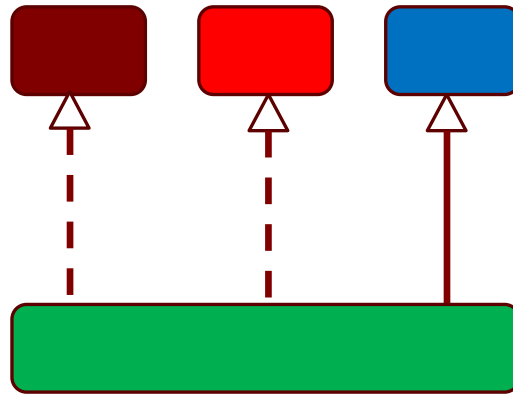
Mix-In Inheritance

SECTION 5

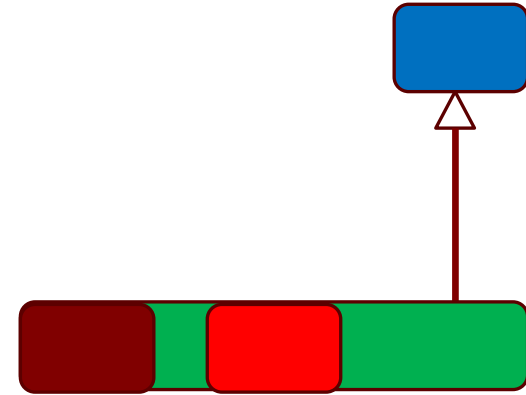
Multiple Inheritance VS Mix-In Inheritance



Multiple Inheritance



Mix-In Inheritance



Single Inheritance

Mix-In Inheritance

- Classes can inherit from only one **real** parent
- Can **mix in** any number of interfaces, simulating multiple inheritance
- Interfaces appear in Java, C#, Go, Ruby, Scala, Objective-C, Swift, Ada 2005, and etc.
 - contain only abstract methods, no method bodies or fields
- Has become dominant approach, superseding true multiple inheritance

WIN10 WIDGETS

FOR RAINMETER BY TJ MARKHAM

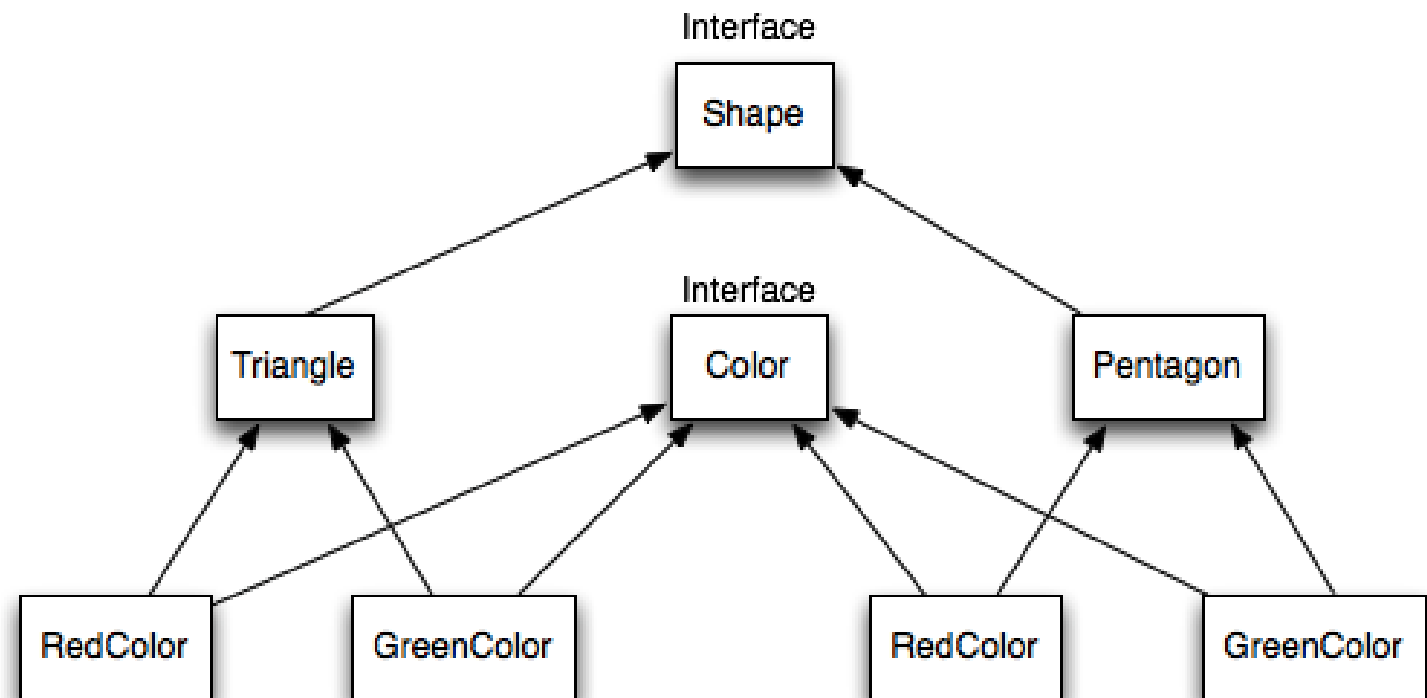
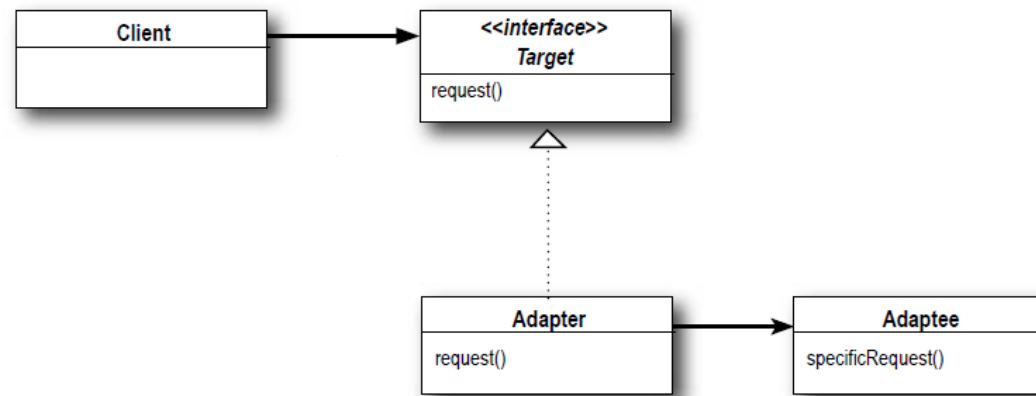
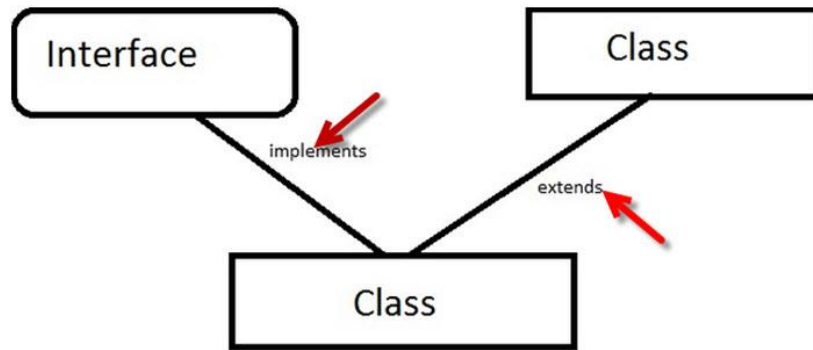


win10widgets.com

The Motivation of Interfaces

- To capture the requirements of the windowing system, a language with mix-in inheritance allows the programmer to define the interface that a class must provide in order for its objects to be used in certain contexts.
- For Widgets, the reporting system define a **sortable_object** interface; the window system might define a **graphable_object** interface; the file system might define a **storable_object** interface.
- No actual functionality would be provided by any of the interfaces: the designer of the widget class would need to provide appropriate implementation.
- An interface is a class containing only abstract methods.
- Widgets implements the same interface can perform the same method of different implementation in a same way.

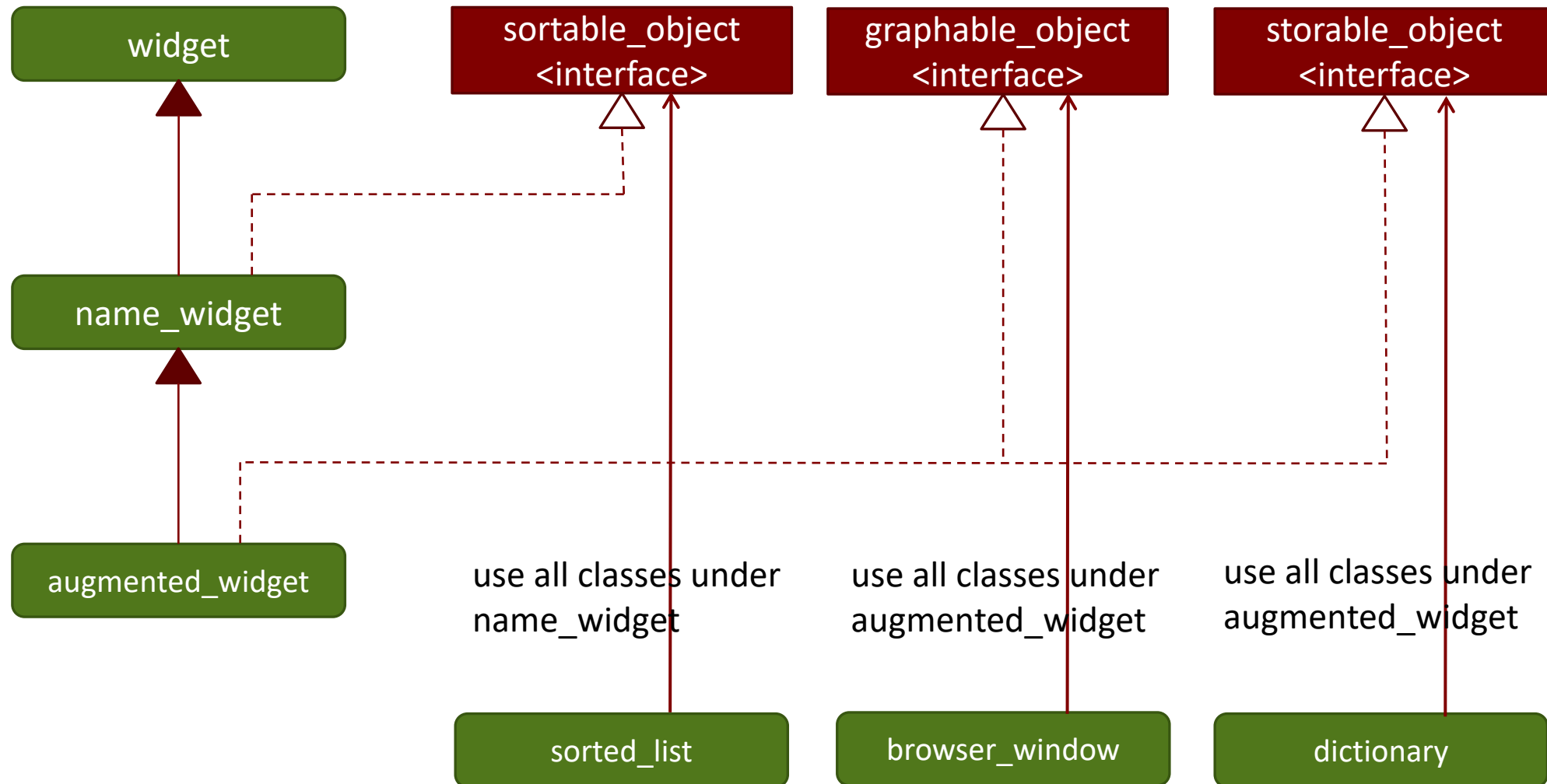
Mixing Interfaces into a Derived Class



Implementation of Interfaces

- In a language like **Ruby**, **Objective-C**, or **Swift**, which uses dynamic method lookup, the methods of an interface can be added to the method dictionary of any class that implements the interface.
- In any context that requires the interface type, the usual lookup mechanism will find the proper methods. In a language with fully static typing, in which pointers to methods are expected to be at known **vtable offsets**, new machinery is required.
- The challenge boils down to a need for multiple views of an object.

UML Class Diagram for Figure 10.6



Interface Classes in Java

```
public class widget { ...
}

interface sortable_object {
    String get_sort_name();
    bool less_than(sortable_object o);
    // All methods of an interface are automatically public.
}

interface graphable_object {
    void display_at(Graphics g, int x, int y);
    // Graphics is a standard library class that provides a context
    // in which to render graphical objects.
}

interface storable_object {
    String get_stored_name();
}

class named_widget extends widget implements sortable_object {
    public String name;
    public String get_sort_name() {return name;}
    public bool less_than(sortable_object o) {
        return (name.compareTo(o.get_sort_name()) < 0);
        // compareTo is a method of the standard library class String.
    }
}

class augmented_widget extends named_widget
    implements graphable_object, storable_object {
    ... // more data members
    public void display_at(Graphics g, int x, int y) {
        ... // series of calls to methods of g
    }
    public String get_stored_name() {return name;}
}

...
class sorted_list {
    public void insert(sortable_object o) { ...
    public sortable_object first() { ...
    ...
}

class browser_window extends Frame {
    // Frame is the standard library class for windows.
    public void add_to_window(graphable_object o) { ...
    ...
}

class dictionary {
    public void insert(storable_object o) { ...
    public storable_object lookup(String name) { ...
    ...
}
```

- By Implementing the **sortable_object** interface in **name_widget** and the **graphable_object** and **storable_object** interfaces in **augmented_widget**, we obtain the ability to pass objects of those classes to and from such routines as
 - **sorted_list.insert**,
 - **browser_window.add_to_window**, and
 - **dictionary.insert**.
- **dictionary.insert** expects a **storable_object** view of its parameter – a way to find the parameter's **get_stored_name** method, however, is implemented by **augmented_widget**, and will expect an **augmented_widget** view of its this parameter – a way to find the object's fields and other methods.
- Given that **augmented_widget** implements three different interfaces, there is no way that a single **vtable** can suffice: its first entry can't be the first method of **sortable_object**, **graphable-object**, and **storable_object** simultaneously.

Compile-Time Implementation of mix-in Inheritance

- The most common solution is to include three extra **vtable** pointers in each **augmented_widget** objects – one for each of the implemented interfaces.
- For each interface view, we can use pointer to the place within the object where the corresponding **vtable** pointer appears.
[this correction]
- If we call **dictionary.insert** on an **augmented_widget** object **w**, whose address is currently in **r1**. The compiler, which knows the offset **c** of **w's storable_object vtable** pointer, will add **c** to **r1** before passing it to insert.

Compile-Time Implementation of mix-in Inheritance

- What happens when insert calls **storable_object.get_stored_name**?
- Assume that the **storable_object** view of **w** is available in, register **r1**, the compiler will generate code that looks something like this:

r2 := *r1	- - vtable addresses
r3 := *r2	- - this correction (indirect addressing)
r3 += r1	- - address of w
call *(r2+4)	- - method addresses

Note: assume that [\[this correction\]](#) occupies the first four bytes of the **vtable**, and that the addresses of **get_stored_name** lies after it, in the table's first regular slot. We assumed that [this](#) should be passed to **r3** and that there are no other arguments.

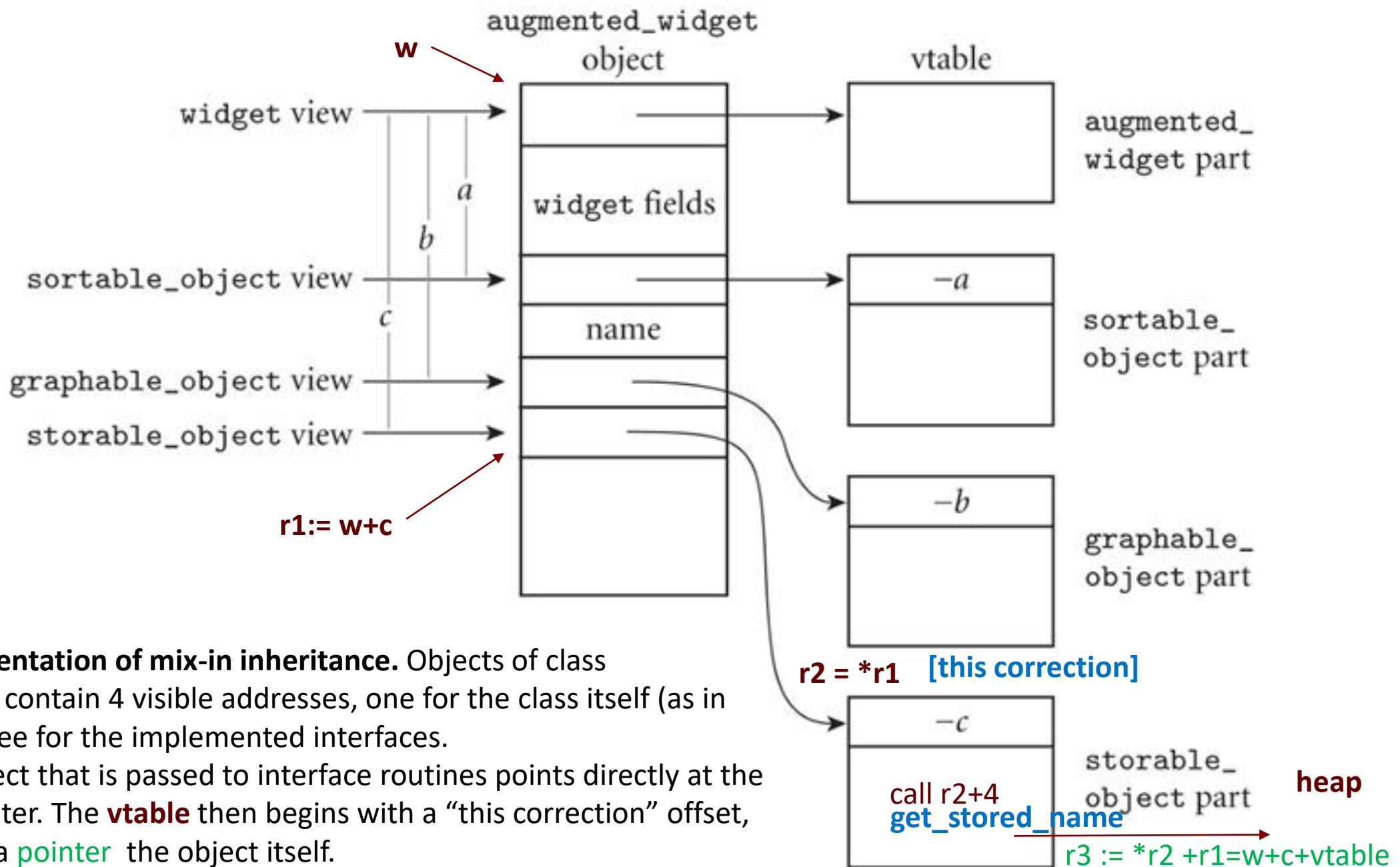


Figure 10.7 Implementation of mix-in inheritance. Objects of class `augmented_widget` contain 4 visible addresses, one for the class itself (as in Figure 10.3) and three for the implemented interfaces. The view of the object that is passed to interface routines points directly at the relevant `vtable` pointer. The `vtable` then begins with a “this correction” offset, used to regenerate a `pointer` the object itself.

Multiple Inheritance

SECTION 6

True Multiple Inheritance

- In C++, you can say
class professor : public teacher, public researcher {
 ...
}

Here you get all the members of teacher ***and*** all the members of researcher

- If there's anything that's in both (same name and argument types), then calls to the member are ambiguous; the compiler disallows them

True Multiple Inheritance

- You can of course create your own member in the merged class

```
professor::print () {  
    teacher::print ();  
    researcher::print (); ...  
}
```

Or you could get both:

```
professor::tprint () {  
    teacher::print ();  
}  
professor::rprint () {  
    researcher::print ();  
}
```

True Multiple Inheritance

- Virtual base classes: In the usual case if you inherit from two classes that are both derived from some other class B, your implementation includes two copies of B's data members
- That's often fine, but other times you want a **single** copy of B
 - For that you make B a virtual base class

The Conclusion of Object-oriented Programming

SECTION 7

Object-Oriented Programming

Everything is an Object that Can be Inherited and Generalized

- Analogy to the real world is central to the Object-Oriented paradigm - you think in terms of **real-world** objects that interact to get things done
- Many Object-Oriented languages are strictly sequential, but the model adapts well to parallelism as well
- Strict interpretation of the term
 - uniform data abstraction - everything is an object
 - inheritance
 - dynamic method binding (polymorphism)

Object-Oriented Programming

What are the benefits of Object-Oriented Programming?

- Modularized
- Support event-driven programming better
- Support concurrent programming better
- Support distributed computing better
- Support graphic user interface better
- Support development of application programming interface (API), foundation class (FC), and design framework better (Uniform Object Model)

Object-Oriented Programming

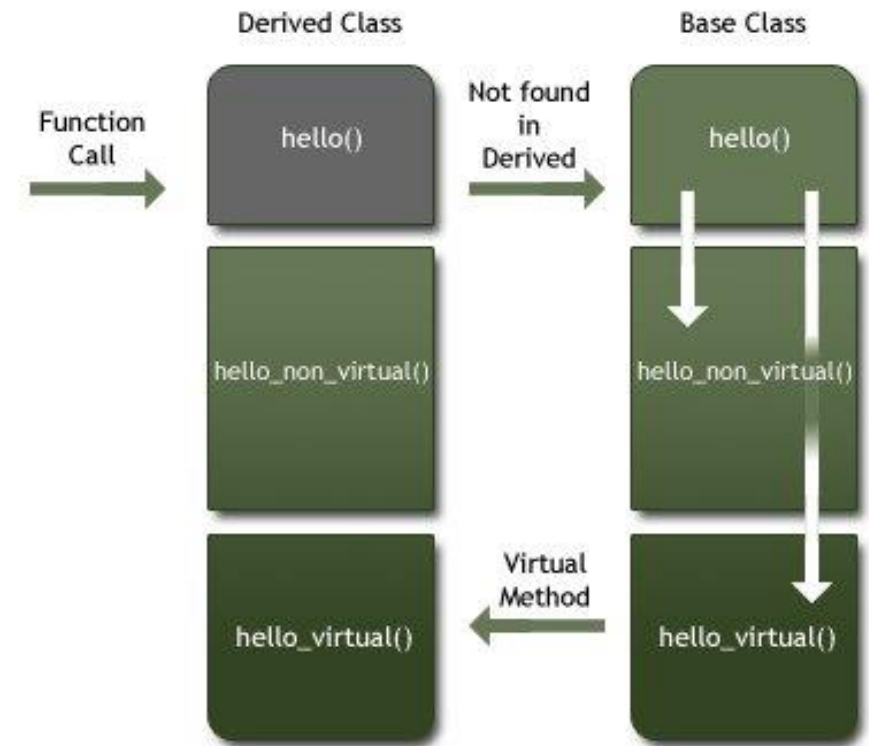
- Lots of conflicting uses of the term out there object-oriented **style** available in many languages
 - data abstraction **crucial**
 - inheritance **required** by most users of the term Object-Oriented
 - centrality of dynamic method binding a matter of dispute

Object-Oriented Programming

- Smalltalk and Ruby are closer to ideal object-oriented programming language.
- **Smalltalk** is, historically, the canonical object-oriented language
 - It has all three of the characteristics listed above
 - It's based on the thesis work of Alan Kay at Utah in the late 1960's
 - It went through 5 generations at Xerox PARC, where Kay worked after graduating
 - Smalltalk-80 is the current standard

Object-Oriented Programming

- Other languages are described in what follows:
- Modula-3
 - single inheritance
 - all methods virtual
 - no constructors or destructors
- Java
 - interfaces, **mix-in** inheritance
 - all methods virtual



Object-Oriented Programming

Ada 95

- **tagged** types
- single inheritance
- no constructors or destructors
- **class-wide** parameters:
 - methods static by default
 - can define a parameter or pointer that grabs the object-specific version of all methods
 - base class doesn't have to decide what will be virtual
- notion of child packages as an alternative to friends

Object-Oriented Programming

Is C++ object-oriented?

- Uses all the right buzzwords
- Has (multiple) inheritance and generics (templates)
- Allows creation of user-defined classes that look just like built-in ones
- Has all the low-level C stuff to escape the paradigm
- Has friends
- Has static type checking

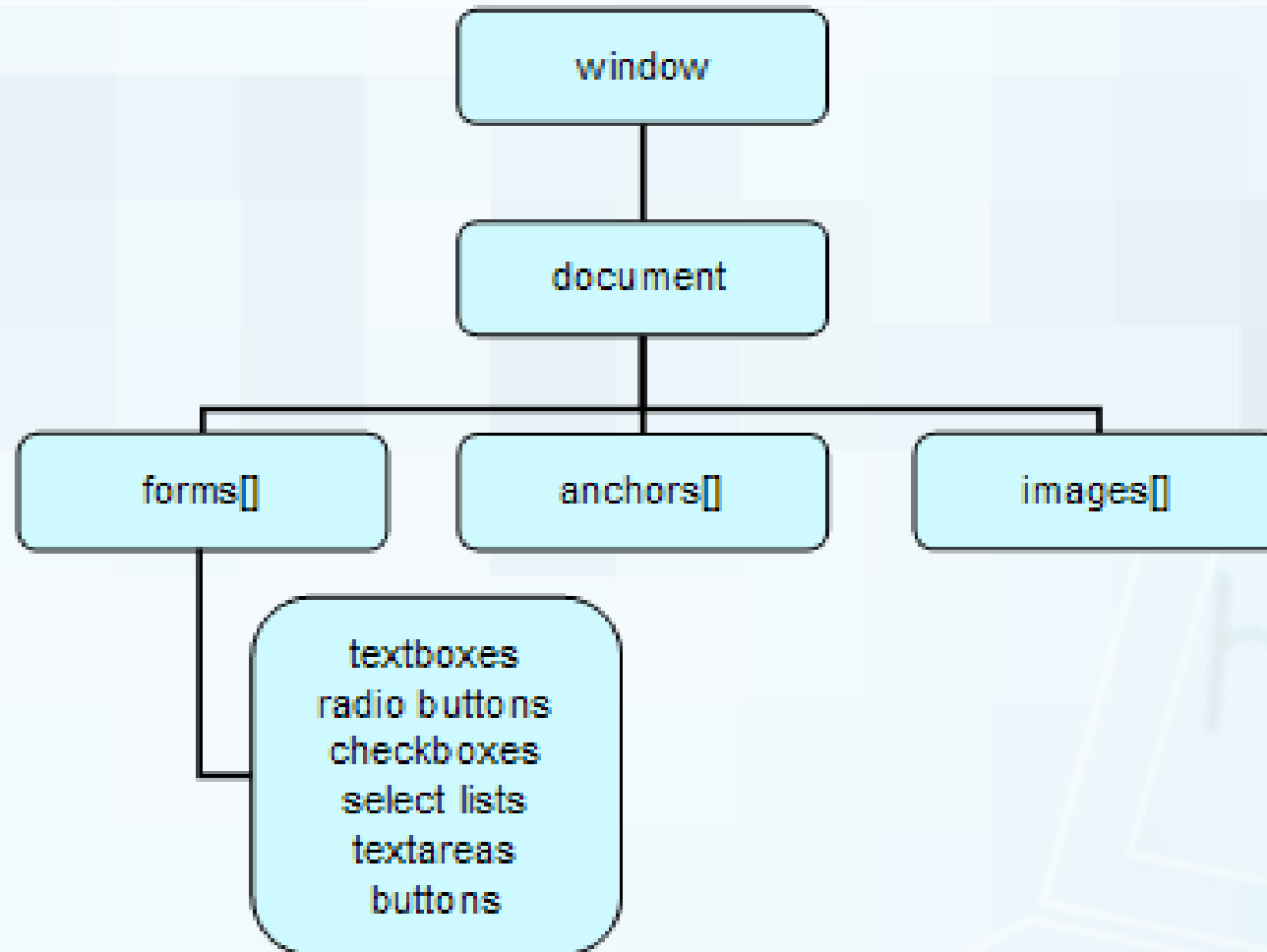
Object-Oriented Programming

Is JavaScript object-oriented?

- JavaScript is a functional, dynamic, interpreted and not out-of-the-box Object-Oriented Programming Package
- Key Features of Object-Oriented Programming:
 - Encapsulation (One way to support is through Closure)
 - Inheritance (One way to support is through Prototypal Inheritance)
 - Polymorphism
- The main difference between classical Object-Oriented languages (Java, C++, .NET) and JavaScript is that the Object-Oriented Programming Features are not provided out-of-the-box by the language.



The HTML DOM



Object-Oriented Programming

Other Programming Paradigms will Not be Covered in an 8-Week Course

- In the same category of questions:
 - Is Prolog a logic language?
 - Is Common Lisp functional?
- However, to be more precise:
 - Smalltalk is really pretty purely object-oriented
 - Prolog is primarily logic-based
 - Common Lisp is largely functional
 - C++ can be used in an object-oriented style