

Virtual Functions

- A virtual function is a class member function that is declared within a base class and **redefined** by a derived class. The term "**overriding**" is used to describe virtual function redefinition.
- To create a virtual function, precede the function's declaration with the keyword "**virtual**" such as

virtual void func();

Virtual Functions (2)

- Implement the "**one interface, multiple methods**" philosophy that underlies "**polymorphism**".
- When redefined by a derived class, the keyword "virtual" is **not** needed.
- A redefined virtual function must have precisely the same **type** and **number of parameters** and the same **return type**.

Virtual Functions (3)

```
class Base {  
    public:  
        virtual void func();  
}
```

```
class Derive1 : public Base {  
    public:  
        void func();  
}
```

```
class Derive2 : public Base {  
    public:  
        void func();  
}
```

Virtual Functions (4)

```
main() {  
    Base *p;  
    Base ob;  
    Derived1 ob1;  
    Derived2 ob2;  
  
    p = &ob; p->func();    // use base's func  
    p = &ob1; p->func();  // use Derived1's func  
    p = &ob2; p->func();  // use Derived2's func  
}
```

Virtual Functions (5)

- Destructor functions may be virtual, constructors may **not**.
- Virtual functions are accessed **only** via a **base class pointer** (or **reference**).
- The type of the object being pointed to determines which version of an overridden virtual function will be executed when accessed via a base class pointer, and that this decision is made at **run time** (this is why it is called "**dynamic binding**").

```

class Base
{
public:
    Base()          { cout << "Constructor: Base" << endl; }
    ~Base()         { cout << "Destructor: Base" << endl; }
};

class Derived: public Base
{
public:
    Derived()       { cout << "Constructor: Derived" << endl; }
    ~Derived()      { cout << "Destructor: Derived" << endl; }
};

void main()
{
    Base *Var = new Derived();
    delete Var;
}

```

- The destructor of the derived class was not called at all.

```

class Base
{
public:
    Base()                { cout << "Constructor: Base" << endl; }
    virtual ~Base()       { cout << "Destructor: Base" << endl; }
};

```

```

class Derived: public Base
{
public:
    Derived()             { cout << "Constructor: Derived" << endl; }
    ~Derived()            { cout << "Destructor: Derived" << endl; }
};

```

```

void main() {
    Base *Var = new Derived();
    delete Var; }

```

- We cannot declare **pure** virtual destructor. Even if a virtual destructor is declared as pure, it will have to implement an **empty** body (at least) for the destructor.

Virtual Functions (6)

- When a virtual function is called by a specific object by **name** and using the **dot** member selection operator, the call is resolved at **compile time** (this is called "**static binding**").
- A virtual function is resolved "**statically**" when it is called with an explicitly **scope operator**, even if a pointer or reference is used. For example,

p->Base::func();

// not virtual; call base-class version

Virtual Functions (7)

- Do **not** make a function virtual unless you want the derived class to be able to get control of it.

Pure Virtual Functions

- Has no definition relative to the **base** class. Only the function's prototype is included. For example,

virtual void func() = 0;

- Forces the derived class to override it. If a derived class does not, a **compile-time** error results.

Pure Virtual Functions (cont)

- Defer the implementation decision of the function. In OOP terminology, it is called a "deferred method".

Abstract Class

- A class that has at least one pure virtual function.
- Technically it is an incomplete type, and no objects of an abstract base class can be created.
- However, pointers and references to abstract base class are okay.

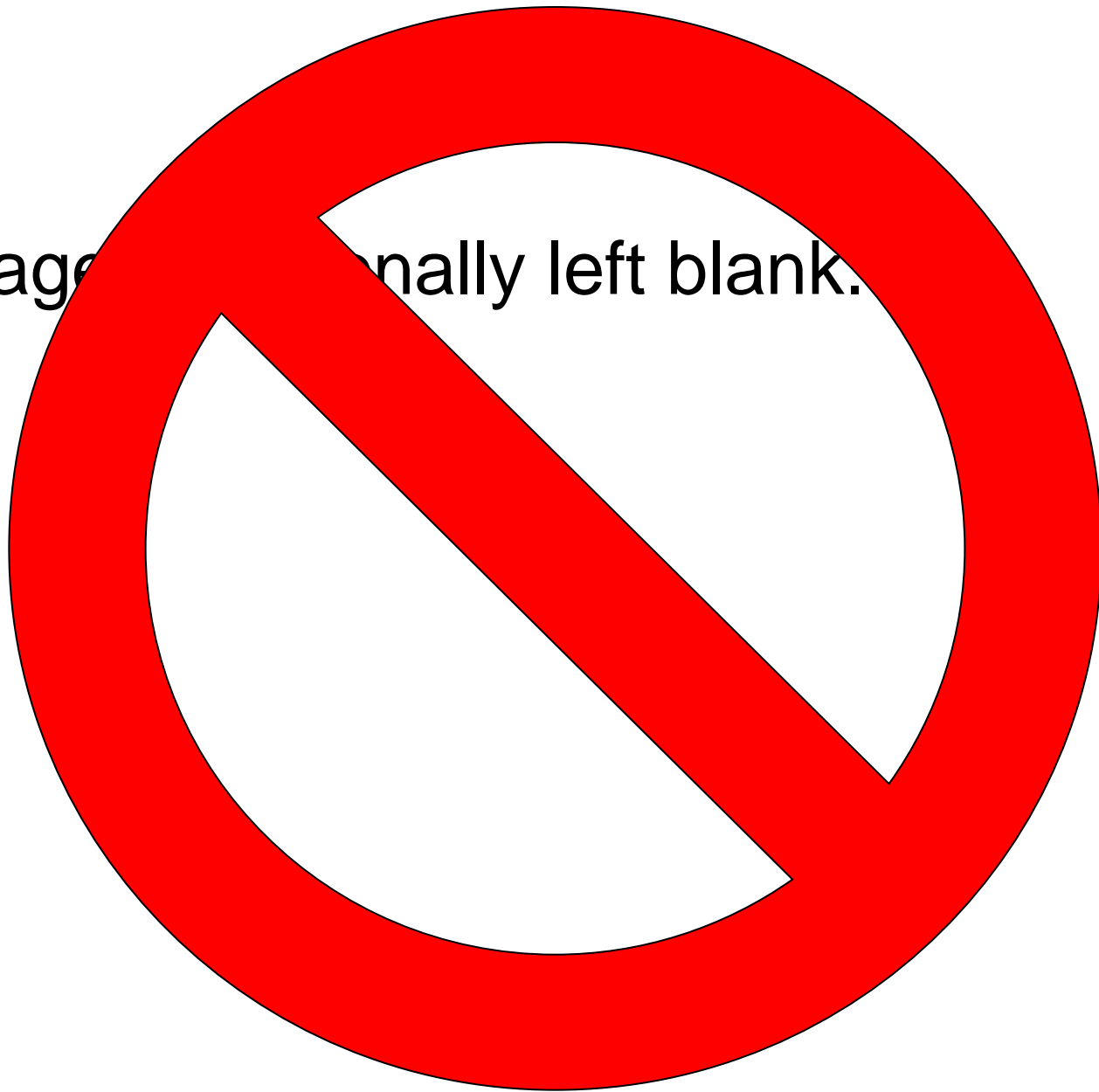
Early (or Static) Binding

- Early binding refers to those events that can be known at compile time, which include "normal" functions, **overloaded functions**, and non-virtual member and friend functions.
 1. advantages: efficient.
 2. disadvantages: lack of flexibility.

Late (or Dynamic) Binding

- Late binding may have more overhead associated with a function call, thus is generally slower.

This page intentionally left blank.



Chapter 10- Virtual Functions and Polymorphism

Outline

- 10.1 Introduction
- 10.2 Type Fields and `switch` Statements
- 10.3 Virtual Functions
- 10.4 Abstract Base Classes and Concrete Classes
- 10.5 Polymorphism
- 10.6 Case Study: A Payroll System Using Polymorphism
- 10.7 New Classes and Dynamic Binding
- 10.8 Virtual Destructors
- 10.9 Case Study: Inheriting Interface and Implementation
- 10.10 Polymorphism, `virtual` Functions and Dynamic Binding
“Under the Hood”

10.2 Type Fields and `switch` Statements

- `switch` statement
 - Take an action on a object based on its type
 - `virtual` functions and polymorphic programming can eliminate the need for `switch` logic

10.3 virtual Functions

- **virtual** functions
 - Suppose a set of shape classes such as **Circle**, **Triangle**, etc.
 - Every shape has own unique draw function but possible to call them by calling the **draw** function of base class **Shape**
 - Compiler determines dynamically (i.e., at run time) which to call
 - In base-class declare **draw** to be **virtual**
 - Override **draw** in each of the derived classes
 - **virtual** declaration:
 - Keyword **virtual** before function prototype in base-class

```
virtual void draw() const;
```

10.3 Virtual Functions

`ShapePtr->Draw() ;`

- Compiler implements **dynamic** binding
- Function determined during execution time

`ShapeObject.Draw() ;`

- Compiler implements **static** binding
- Function determined during compile-time

10.4 Abstract and Concrete Classes

- Abstract classes
 - Sole purpose is to provide a base class for other classes
 - **No** objects of an abstract base class can be instantiated
 - Too generic to define real objects
 - Can have pointers and references
- Making abstract classes
 - Declare one or more **virtual** functions as “pure” by initializing the function to zero
 - Example of a pure **virtual** function:

```
virtual double earnings() const = 0;
```

10.5 Polymorphism

- Polymorphism

- If non-`virtual` member function defined in multiple classes and called from base-class pointer then the base-class version is used
- Suppose `print` is **not** a `virtual` function

```
Employee e, *ePtr = &e;
HourlyWorker h, *hPtr = &h;
ePtr->print();    // call base-class print function
hPtr->print();    // call derived-class print
function
ePtr=&h;          // allowable implicit conversion
ePtr->print();    // still calls base-class print
```

10.8 Virtual Destructors

- Problem:
 - If a base-class pointer to a derived object is deleted, the base-class destructor will act on the object
- Solution:
 - declare a `virtual` base-class destructor to ensure that the appropriate destructor will be called

10.10 Polymorphism, virtual Functions and Dynamic Binding

“Under the Hood”

- When to use polymorphism
 - Polymorphism requires a lot of overhead
 - Polymorphism is **not** used in STL (Standard Template Library) to optimize performance
- **virtual** function table (vtable)
 - Every class with a **virtual** function has a vtable
 - For every **virtual** function, its vtable has a pointer to the proper function
 - If a derived class has the same function as a base₂₃ class, then the function pointer points to the base-