C++

Polymorphism
C++ Polymorphism

- Polymorphism Examples
- Relationships Among Objects in an Inheritance Hierarchy
  - Invoking Base-Class Functions from Derived-Class Objects
  - Aiming Derived-Class Pointers at Base-Class Objects
  - Derived-Class Member-Function Calls via Base-Class Pointers
  - Virtual Functions
- Summary of the Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers
- Type Fields and `switch` Statements
- Abstract Classes and Pure `virtual` Functions
- Polymorphism Case Study
24.1 Introduction

- Polymorphism with inheritance hierarchies
  - “Program in the general” vs. “program in the specific”
  - Process objects of classes that are part of the same hierarchy as if they are all objects of the base class.
  - Each object performs the correct tasks for that object’s type
    - Different actions occur depending on the type of object.
  - New classes can be added with little or not modification to existing code.
Polymorphism Examples

Example Animal hierarchy

- Animal base class – every derived class has a function move.
- Different animal objects are maintained as a vector of Animal pointers.
- Program issues same message (move) to each animal generically.
- Proper function gets called
  - A Fish will move by swimming.
  - A Frog will move by jumping.
  - A Bird will move by flying.
24.2 Polymorphism Examples

Polymorphism occurs when a program invokes a virtual function through a base-class pointer or reference.

- C++ dynamically chooses the correct function for the class from which the object was instantiated.

Example: SpaceObject

- Video game manipulates objects of types that inherit from SpaceObject, which contains member function draw.
- Function draw implemented appropriately for the different derived classes.
- A screen-manager program maintains a container of SpaceObject pointers.
- Call draw on each object using SpaceObject pointers.
  - The proper draw function is called based on object’s type.
- A new class derived from SpaceObject can be added without affecting the screen manager.
24.3 Relationships among Objects in an Inheritance Hierarchy

- Aim base-class pointer at base-class object
  - Invoke base-class functionality
- Aim derived-class pointer at derived-class object
  - Invoke derived-class functionality
- Aim base-class pointer at derived-class object
  - Because derived-class object is an object of base class
  - Invoke base-class functionality
  - Invoked functionality depends on the type of the handle used to invoke the function, not on the type of the object to which the handle points.

- virtual functions
  - Make it possible to invoke the object type's functionality, rather than invoke the handle type's functionality.
  - Crucial to implementing polymorphic behavior.
Invoking Base-Class Functions from Derived-Class Objects

// Fig. 24.1: CommissionEmployee.h
// CommissionEmployee class definition represents a commission employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using std::string;

class CommissionEmployee
{
    public:
        CommissionEmployee( const string &, const string &, const string &, double = 0.0, double = 0.0 );

        void setFirstName( const string & ); // set first name
        string getFirstName() const; // return first name

        void setLastName( const string & ); // set last name
        string getLastName() const; // return last name

        void setSocialSecurityNumber( const string & ); // set SSN
        string getSocialSecurityNumber() const; // return SSN

        void setGrossSales( double ); // set gross sales amount
        double getGrossSales() const; // return gross sales amount

    private:

    CommissionEmployee( const string &, const string &, const string &, double, double );

    double commissionRate; // sales commission rate
    double grossSales; // gross sales amount

};
void setCommissionRate( double ); // set commission rate
double getCommissionRate( ) const; // return commission rate
double earnings( ) const; // calculate earnings
void print( ) const; // print CommissionEmployee object

private:
  string firstName;
  string lastName;
  string socialSecurityNumber;
  double grossSales; // gross weekly sales
  double commissionRate; // commission percentage
}; // end class CommissionEmployee

Function **earnings** will be redefined in derived classes to calculate the employee’s earnings

Function **print** will be redefined in derived class to print the employee’s information
// Fig. 24.2: CommissionEmployee.cpp
// Class CommissionEmployee member-function definitions.
#include <iostream>
#include <iostream>
#include <iostream>
#include "CommissionEmployee.h" // CommissionEmployee class definition

// constructor
CommissionEmployee::CommissionEmployee( 
    const string &first, const string &last, const string &ssn, 
    double sales, double rate )
    : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
{
    setGrossSales( sales ); // validate and store gross sales
    setCommissionRate( rate ); // validate and store commission rate
} // end CommissionEmployee constructor

// set first name
void CommissionEmployee::setFirstName( const string &first )
{
    firstName = first; // should validate
} // end function setFirstName

// return first name
string CommissionEmployee::getFirstName() const
{
    return firstName;
} // end function getFirstName
29 // set last name
30 void CommissionEmployee::setLastName(const string &last)
31 {
32     lastName = last; // should validate
33 } // end function setLastName
34
35 // return last name
36 string CommissionEmployee::getLastName() const
37 {
38     return lastName;
39 } // end function getLastName
40
41 // set social security number
42 void CommissionEmployee::setSocialSecurityNumber(const string &ssn)
43 {
44     socialSecurityNumber = ssn; // should validate
45 } // end function setSocialSecurityNumber
46
47 // return social security number
48 string CommissionEmployee::getSocialSecurityNumber() const
49 {
50     return socialSecurityNumber;
51 } // end function getSocialSecurityNumber
52
53 // set gross sales amount
54 void CommissionEmployee::setGrossSales(double sales)
55 {
56     grossSales = (sales < 0.0) ? 0.0 : sales;
57 } // end function setGrossSales
Invoking Base-Class Functions from Derived-Class Objects

// return gross sales amount
double CommissionEmployee::getGrossSales() const
{
    return grossSales;
} // end function getGrossSales

// set commission rate
void CommissionEmployee::setCommissionRate( double rate )
{
    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
} // end function setCommissionRate

// return commission rate
double CommissionEmployee::getCommissionRate() const
{
    return commissionRate;
} // end function getCommissionRate

// calculate earnings
double CommissionEmployee::earnings() const
{
    return getCommissionRate() * getGrossSales();
} // end function earnings

Calculate earnings based on commission rate and gross sales
Invoking Base-Class Functions from Derived-Class Objects

```cpp
// print CommissionEmployee object
void CommissionEmployee::print() const
{
    cout << "commission employee: " << getFirstName() << ' ' << getLastName() << 
         "social security number: " << getSocialSecurityNumber() << 
         "gross sales: " << getGrossSales() << 
         "commission rate: " << getCommissionRate();
}
```

Display name, social security number, gross sales and commission rate
Invoking Base-Class Functions from Derived-Class Objects

---

```cpp
// Fig. 24.3: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class CommissionEmployee.
#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
using std::string;

#include "CommissionEmployee.h" // CommissionEmployee class declaration

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
    BasePlusCommissionEmployee( const string &, const string &, const string &,
        double = 0.0, double = 0.0, double = 0.0 );

    void setBaseSalary( double ); // set base salary
double getBaseSalary() const; // return base salary
double earnings() const; // calculate earnings
void print() const; // print BasePlusCommissionEmployee object

private:
    double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif
```

Redefine functions *earnings* and *print*
Invoking Base-Class Functions from Derived-Class Objects

// Fig. 24.4: BasePlusCommissionEmployee.cpp
// Class BasePlusCommissionEmployee member-function definitions.
#include <iostream>
using std::cout;

// BasePlusCommissionEmployee class definition
#include "BasePlusCommissionEmployee.h"

// constructor
BasePlusCommissionEmployee::BasePlusCommissionEmployee(const string &first, const string &last, const string &ssn, double sales, double rate, double salary )
// explicitly call base-class constructor
: CommissionEmployee( first, last, ssn, sales, rate )
{
  setBaseSalary( salary ); // validate and store base salary
} // end BasePlusCommissionEmployee constructor

// set base salary
void BasePlusCommissionEmployee::setBaseSalary( double salary )
{
  baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
} // end function setBaseSalary

// return base salary
double BasePlusCommissionEmployee::getBaseSalary() const
{
  return baseSalary;
} // end function getBaseSalary
Invoking Base-Class Functions from Derived-Class Objects

```cpp
30 // calculate earnings
31 double BasePlusCommissionEmployee::earnings() const
32 {
33     return getBaseSalary() + CommissionEmployee::earnings();
34 } // end function earnings

// print BasePlusCommissionEmployee object
37 void BasePlusCommissionEmployee::print() const
38 {
39     cout << "base-salaried ";
40     // invoke CommissionEmployee's print function
41     CommissionEmployee::print();
42     cout << \nbase salary: " << getBaseSalary();
43 } // end function print
```

Redefined earnings function incorporates base salary

Redefined print function displays additional BasePlusCommissionEmployee details
// Fig. 24.5: fig24_05.cpp
// Aiming base-class and derived-class pointers at base-class
// and derived-class objects, respectively.
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

// include class definitions
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
    // create base-class object
    CommissionEmployee commissionEmployee("Sue", "Jones", "222-22-2222", 10000, .06);

    // create base-class pointer
    CommissionEmployee *commissionEmployeePtr = 0;
Invoking Base-Class Functions from Derived-Class Objects

```cpp
24 // create derived-class object
25 BasePlusCommissionEmployee basePlusCommissionEmployee(
26     "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
27
28 // create derived-class pointer
29 BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
30
31 // set floating-point output formatting
32 cout << fixed << setprecision( 2 );
33
34 // output objects commissionEmployee and basePlusCommissionEmployee
35 cout << "Print base-class and derived-class objects:
      
commissionEmployee.print();  // invokes base-class print
36         cout << "\n";
37         basePlusCommissionEmployee.print();  // invokes derived-class print
38         cout << "\n";
39
40 // aim base-class pointer at base-class object and print
41 commissionEmployeePtr = &commissionEmployee;  // perfectly natural
42         cout << "\n\nCalling print with base-class pointer to "
43                 << "\nbase-class object invokes base-class print function:\n      "
44                 << "\nbase-class object invokes base-class print function:\n      ";
45         commissionEmployeePtr->print();  // invokes base-class print
```

Aiming base-class pointer at base-class object and invoking base-class functionality
Invoking Base-Class Functions from Derived-Class Objects

// aim derived-class pointer at derived-class object and print
basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
cout << "\n\nCalling print with derived-class pointer to "
<< "derived-class object invokes derived-class "
<< "print function:\n\n";
basePlusCommissionEmployeePtr->print(); // invokes derived-class print

// aim base-class pointer at derived-class object and print
commissionEmployeePtr = &basePlusCommissionEmployee;
cout << "\n\nCalling print with base-class pointer to "
<< "derived-class object invokes base-class print"
<< "function on that derived-class object:\n\n";
commissionEmployeePtr->print(); // invokes base-class print

cout << endl;
return 0;
} // end main

Aiming derived-class pointer at derived-class object and invoking derived-class functionality

Aiming base-class pointer at derived-class object and invoking base-class functionality
Invoking Base-Class Functions from Derived-Class Objects

Print base-class and derived-class objects:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Calling print with base-class pointer to base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

(Continued at top of next slide...)
Calling `print` with derived-class pointer to
derived-class object invokes derived-class `print` function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Calling `print` with base-class pointer to derived-class object
invokes base-class `print` function on that derived-class object:

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
24.3.2 Aiming Derived-Class Pointers at Base-Class Objects

- Aim a derived-class pointer at a base-class object.
  - C++ compiler generates error.
    - CommissionEmployee (base-class object) is not a BasePlusCommissionEmployee (derived-class object)
  - If this were to be allowed, programmer could then attempt to access derived-class members which do not exist.
    - Could modify memory being used for other data.
Aiming Derived-Class Pointers at Base-Class Objects

```
// Fig. 24.6: fig24_06.cpp
// Aiming a derived-class pointer at a base-class object.
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
    CommissionEmployee commissionEmployee("Sue", "Jones", "222-22-2222", 10000, .06);
    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;

    // aim derived-class pointer at base-class object
    // Error: a CommissionEmployee is not a BasePlusCommissionEmployee
    basePlusCommissionEmployeePtr = &commissionEmployee;
    return 0;
} // end main
```

Cannot assign base-class object to derived-class pointer because is-a relationship does not apply
**Borland C++ command-line compiler error messages:**

Error E2034 Fig24_06\fig24_06.cpp 14: Cannot convert 'CommissionEmployee *' to 'BasePlusCommissionEmployee *' in function main()

**GNU C++ compiler error messages:**

Fig24_06.cpp:14: error: invalid conversion from `CommissionEmployee*' to `BasePlusCommissionEmployee*'

**Microsoft Visual C++,NET compiler error messages:**

C:\examples\ch24\Fig24_06\fig24_06.cpp(14) : error C2440:
'=' : cannot convert from 'CommissionEmployee ___w64' to
'BasePlusCommissionEmployee *'
Cast from base to derived requires dynamic_cast or static_cast
24.3.3 Derived-Class Member-Function Calls via Base-Class Pointers

- Aiming base-class pointer at derived-class object.
  - Calling functions that exist in base class causes base-class functionality to be invoked.
  - Calling functions that do not exist in base class (may exist in derived class) will result in error.
    - Derived-class members cannot be accessed from base-class pointers.
    - However, this can be accomplished using downcasting (Section 13.8).
Aiming base-class pointer at derived-class object

```cpp
// Fig. 24.7: fig24_07.cpp
// Attempting to invoke derived-class-only member functions through a base-class pointer.
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
    CommissionEmployee *commissionEmployeePtr = 0; // base class
    BasePlusCommissionEmployee basePlusCommissionEmployee("Bob", "Lewis", "333-33-3333", 5000, .04, 300); // derived class

    // aim base-class pointer at derived-class object
    commissionEmployeePtr = &basePlusCommissionEmployee;

    // invoke base-class member functions on derived-class
    // object through base-class pointer
    string firstName = commissionEmployeePtr->getFirstName();
    string lastName = commissionEmployeePtr->getLastName();
    string ssn = commissionEmployeePtr->getSocialSecurityNumber();
    double grossSales = commissionEmployeePtr->getGrossSales();
    double commissionRate = commissionEmployeePtr->getCommissionRate();

    // attempt to invoke derived-class-only member functions
    // on derived-class object through base-class pointer
    double baseSalary = commissionEmployeePtr->getBaseSalary();
    commissionEmployeePtr->setBaseSalary(500);

    return 0;
} // end main
```

© 2007 Pearson Ed - All rights reserved.
Borland C++ command-line compiler error messages:

Error E2316 Fig24_07\fig24_07.cpp 26: 'getBaseSalary' is not a member of 'CommissionEmployee' in function main()
Error E2316 Fig24_07\fig24_07.cpp 27: 'setBaseSalary' is not a member of 'CommissionEmployee' in function main()

Microsoft Visual C++.NET compiler error messages:

C:\examples\ch24\Fig24_07\fig24_07.cpp(26) : error C2039: 'getBaseSalary' : is not a member of 'CommissionEmployee'
    C:\cpphtp5_examples\ch24\Fig24_07\CommissionEmployee.h(10) : see declaration of 'CommissionEmployee'
C:\examples\ch24\Fig24_07\fig24_07.cpp(27) : error C2039: 'setBaseSalary' : is not a member of 'CommissionEmployee'
    C:\examples\ch24\Fig24_07\CommissionEmployee.h(10) : see declaration of 'CommissionEmployee'

GNU C++ compiler error messages:

Fig24_07.cpp:26: error: `getBaseSalary' undeclared (first use this function)
Fig24_07.cpp:26: error: (Each undeclared identifier is reported only once for each function it appears in.)
Fig24_07.cpp:27: error: `setBaseSalary' undeclared (first use this function)
24.3.4 Virtual Functions

- Normally the handle determines which class's functionality to invoke.
- With virtual functions
  - The type of the object being pointed to, not the type of the handle, determines which version of a virtual function to invoke.
  - This allows a program to dynamically (at runtime rather than compile time) determine which function to use.
    - Referred to as dynamic binding or late binding.
24.3.4 Virtual Functions

- Declared by preceding the function's prototype with the keyword `virtual` in the base class.

Example

```cpp
class Shape {  
public:  
    virtual void draw() const;  
};  
```

- If the program invokes a virtual function through a base-class pointer to a derived-class object (e.g., `shapePtr->draw()`), the program will choose the correct derived-class `draw` function dynamically based on the object type.

- Derived classes override virtual functions to enable polymorphic behavior.
24.3.4 Virtual Functions

- Once declared `virtual`, a function remains `virtual` all the way down the hierarchy.

- When a `virtual` function is called by referencing a specific object by name using the dot member-selection operator (e.g., `squareObject.draw()`), the function invocation is resolved at compile time. {This is static binding and this is Not polymorphic behavior!}

- Dynamic binding with `virtual` functions only occurs off pointer and reference handles.
// Fig. 24.8: CommissionEmployee.h

// CommissionEmployee class definition represents a commission employee.
#ifndef COMMISSION_H
#define COMMISSION_H

#include <string> // C++ standard string class
using std::string;

class CommissionEmployee
{
  public:
    CommissionEmployee( const string &, const string &, const string &, double = 0.0, double = 0.0 );

    void setFirstName( const string & ); // set first name
    string getFirstName() const; // return first name

    void setLastName( const string & ); // set last name
    string getLastName() const; // return last name

    void setSocialSecurityNumber( const string & ); // set SSN
    string getSocialSecurityNumber() const; // return SSN

    void setGrossSales( double ); // set gross sales amount
    double getGrossSales() const; // return gross sales amount
Virtual Functions

```cpp
template <typename ... T>
void setCommissionRate( double ); // set commission rate
double getCommissionRate() const; // return commission rate

virtual double earnings() const; // calculate earnings
virtual void print() const; // print CommissionEmployee object

private:
    string firstName;
    string lastName;
    string socialSecurityNumber;
    double grossSales; // gross weekly sales
    double commissionRate; // commission percentage
}; // end class CommissionEmployee
```

Declaring `earnings` and `print` as `virtual` allows them to be overridden, not redefined.
Systems Programming: Polymorphism

Virtual Functions

`// Fig. 24.9: BasePlusCommissionEmployee.h
// BasePlusCommissionEmployee class derived from class
// CommissionEmployee.

#ifndef BASEPLUS_H
#define BASEPLUS_H

#include <string> // C++ standard string class
using std::string;

#include "CommissionEmployee.h" // CommissionEmployee class declaration

class BasePlusCommissionEmployee : public CommissionEmployee
{
public:
    BasePlusCommissionEmployee( const string &, const string &, const string &, double = 0.0, double = 0.0, double = 0.0 );

    void setBaseSalary( double ); // set base salary
    double getBaseSalary() const; // return base salary

    virtual double earnings() const; // calculate earnings
    virtual void print() const; // print BasePlusCommissionEmployee object

private:
    double baseSalary; // base salary
}; // end class BasePlusCommissionEmployee

#endif

Functions earnings and print are already virtual – good practice to declare virtual even when overriding function

© 2007 Pearson Ed -All rights reserved.
// Fig. 24.10: fig24_10.cpp
// Introducing polymorphism, virtual functions and dynamic binding.
#include <iostream>
using std::cout;
using std::endl;
using std::fixed;

#include <iomanip>
using std::setprecision;

// include class definitions
#include "CommissionEmployee.h"
#include "BasePlusCommissionEmployee.h"

int main()
{
    // create base-class object
    CommissionEmployee commissionEmployee( "Sue", "Jones", "222-22-2222", 10000, .06 );

    // create base-class pointer
    CommissionEmployee *commissionEmployeePtr = 0;

    // create derived-class object
    BasePlusCommissionEmployee basePlusCommissionEmployee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

    // create derived-class pointer
    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
// set floating-point output formatting
cout << fixed << setprecision(2);

// output objects using static binding
cout << "Invoking print function on base-class and derived-class "
    << "\nobjects with static binding\n\n";
commissionEmployee.print(); // static binding
cout << "\n";
basePlusCommissionEmployee.print(); // static binding

// output objects using dynamic binding
cout << "\n\nInvoking print function on base-class and "
    << "derived-class \nobjects with dynamic binding";

// aim base-class pointer at base-class object and print
commissionEmployeePtr = &commissionEmployee;
cout << "\nCalling virtual function print with base-class pointer"
    << "\nto base-class object invokes base-class "
    << "print function:\n\n";
commissionEmployeePtr->print(); // invokes base-class print

Aiming base-class pointer at base-class object and invoking base-class functionality

© 2007 Pearson Ed - All rights reserved.
Aiming derived-class pointer at derived-class object and invoking derived-class functionality

Aiming base-class pointer at derived-class object and invoking derived-class functionality via polymorphism and *virtual* functions
Virtual Functions

Invoking print function on base-class and derived-class objects with static binding

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Invoking print function on base-class and derived-class objects with dynamic binding

Calling virtual function print with base-class pointer to base-class object invokes base-class print function:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Calling virtual function print with derived-class pointer to derived-class object invokes derived-class print function:

(Continued at the top of next slide …)
Virtual Functions

(...Continued from the bottom of previous slide)

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Calling virtual function print with base-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
Summarizing Allowed Assignments Between Base-Class and Derived-Class Objects and Pointers

- Four ways to aim base-class and derived-class pointers at base-class and derived-class objects
  - Aiming a base-class pointer at a base-class object
    • Is straightforward.
  - Aiming a derived-class pointer at a derived-class object
    • Is straightforward.
  - Aiming a base-class pointer at a derived-class object
    • Is safe, but can be used to invoke only member functions that base-class declares (unless downcasting is used).
    • Can achieve polymorphism with virtual functions
  - Aiming a derived-class pointer at a base-class object
    • Generates a compilation error.
24.4 Type Fields and `switch` Statements

- A `switch` statement can be used to determine the type of an object at runtime.
  - Include a type field as a data member in the base class.
  - This enables the programmer to invoke appropriate action for a particular object.
- Causes problems
  - A type test may be forgotten.
  - May forget to add new types.
24.5 Abstract Classes and Pure Virtual Functions

- **Abstract classes**
  - Classes from which the programmer never intends to instantiate any objects.
    - Incomplete—derived classes must define the “missing pieces”.
    - Too generic to define real objects.
  - Normally used as base classes and called **abstract base classes**.
    - Provides an appropriate base class from which other classes can inherit.
- **Classes used to instantiate objects are called **concrete classes**.
  - Must provide implementation for every member function they define.
Abstract Classes and Pure virtual Functions

- **Pure virtual function**: A class is made abstract by declaring one or more of its virtual functions to be "pure" by placing " = 0" in its declaration.

Example

```cpp
virtual void draw() const = 0;
```

- " = 0" is known as a **pure specifier**.
- Do not provide implementations.
Abstract Classes and Pure virtual Functions

- Every **concrete derived class** must override all base-class pure **virtual** functions with concrete implementations.
- If not overridden, the derived-class will also be abstract.
- Used when it does not make sense for base class to have an implementation of a function, but the programmer wants all concrete derived classes to implement the function.
Software Engineering Observation 24.8

- An **abstract class** defines a common public interface for the various classes in a class hierarchy.
- An **abstract class** contains one or more pure **virtual** functions that concrete derived classes must override.
Abstract Classes and Pure virtual Functions

- The **abstract base class** can be used to declare pointers and references that can refer to objects of any concrete class derived from the abstract class.

- Programs typically use such pointers and references to manipulate derived-class objects polymorphically.

- Polymorphism is particularly effective for implementing layered software systems.

**Examples:**

1. Reading or writing data from and to devices.
2. An **iterator class** that can traverse all the objects in a container.