C++

Polymorphism

Systems Programming
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 {No time for this!!}
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 function 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.
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

1. Aim base-class pointer at base-class object
   - Invoke base-class functionality

2. Aim derived-class pointer at derived-class object
   - Invoke derived-class functionality

3. 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.

* This is crucial to implementing polymorphic behavior.
// 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
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
Invoking Base-Class Functions from Derived-Class Objects

// Fig. 24.2: CommissionEmployee.cpp
// Class CommissionEmployee member-function definitions.
#include <iostream>
using std::cout;

#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
Invoking Base-Class Functions from Derived-Class Objects

```cpp
// set last name
void CommissionEmployee::setLastName( const string &last )
{
    lastName = last; // should validate
} // end function setLastName

// return last name
string CommissionEmployee::getLastName() const
{
    return lastName;
} // end function getLastName

// set social security number
void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
{
    socialSecurityNumber = ssn; // should validate
} // end function setSocialSecurityNumber

// return social security number
string CommissionEmployee::getSocialSecurityNumber() const
{
    return socialSecurityNumber;
} // end function getSocialSecurityNumber

// set gross sales amount
void CommissionEmployee::setGrossSales( double sales )
{
    grossSales = ( sales < 0.0 ) ? 0.0 : sales;
} // end function setGrossSales
```
Invoking Base-Class Functions from Derived-Class Objects

59  // return gross sales amount
60  double CommissionEmployee::getGrossSales() const
61  {
62    return grossSales;
63  } // end function getGrossSales
64
65  // set commission rate
66  void CommissionEmployee::setCommissionRate( double rate )
67  {
68    commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
69  } // end function setCommissionRate
70
71  // return commission rate
72  double CommissionEmployee::getCommissionRate() const
73  {
74    return commissionRate;
75  } // end function getCommissionRate
76
77  // calculate earnings
78  double CommissionEmployee::earnings() const
79  {
80    return getCommissionRate() * getGrossSales();
81  } // end function earnings

Calculate earnings based on commission rate and gross sales
// 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

// 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

© 2007 Pearson Ed - All rights reserved.
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
// calculate earnings
double BasePlusCommissionEmployee::earnings() const
{
    return getBaseSalary() + CommissionEmployee::earnings();
} // end function earnings

// print BasePlusCommissionEmployee object
void BasePlusCommissionEmployee::print() const
{
    cout << "base-salaried ";
    // invoke CommissionEmployee's print function
    CommissionEmployee::print();
    cout << "\nbase salary: " << getBaseSalary();
} // 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>
#include <iomanip>
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

// 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 commissionEmployee and basePlusCommissionEmployee
cout << "Print base-class and derived-class objects:

";
commissionEmployee.print(); // invokes base-class print
cout << "\n\n";
basePlusCommissionEmployee.print(); // invokes derived-class print

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

Utilizing pointers now!

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

© 2007 Pearson Ed -All rights reserved.
Invoking Base-Class Functions from Derived-Class Objects

```cpp
// aim derived-class pointer at derived-class object and print
basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
cout << "\n\nCalling print with derived-class pointer to "
  << "\nderived-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 "
  << "\nderived-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...)
Invoking Base-Class Functions from Derived-Class Objects

(...Continued from bottom of previous 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.
// 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:

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

### GNU C++ compiler error messages:

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

### Microsoft Visual C++.NET compiler error messages:

```plaintext
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).
// 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
    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
**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_exampl es\ch24\Fi g24_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:\exampl es\ch24\Fi g24_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**.

Systems Programming: Polymorphism

© 2007 Pearson Ed - All rights reserved.
24.3.4 Virtual Functions

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

Example

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

would appear in the base class `Shape`.

- 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.
Virtual Functions

// 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
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

#if defined
#endif
```

Declaring `earnings` and `print` as `virtual` allows them to be overridden, not redefined.
Virtual Functions

```cpp
// 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.
// Fig. 24.10: fig24_10.cpp
// Introducing polymorphism, virtual functions and dynamic binding.
#include <iostream>
#include <iomanip>
#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;
Virtual Functions

30 // set floating-point output formatting
31 cout << fixed << setprecision( 2 );
32
33 // output objects using static binding
34 cout << "Invoking print function on base-class and derived-class "
35 << "\n objects with static binding\n\n";
36 commissionEmployee.print(); // static binding
37 cout << "\n\n";
38 basePlusCommissionEmployee.print(); // static binding
39
40 // output objects using dynamic binding
41 cout << "\n\n Invoking print function on base-class and "
42 << " derived-class \n objects with dynamic binding";
43
44 // aim base-class pointer at base-class object and print
45 commissionEmployeePtr = &commissionEmployee;
46 cout << "\n\n Calling virtual function print with base-class "
47 << " into base-class object invokes base-class "
48 << " print function: \n\n";
49 commissionEmployeePtr->print(); // invokes base-class print

© 2007 Pearson Ed - All rights reserved.
Virtual Functions

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

// polymorphism invokes BasePlusCommissionEmployee's print;
// base-class pointer to derived-class object
commissionEmployeePtr->print();
cout << endl;
return 0;
} // end main

© 2007 Pearson Ed - All rights reserved.
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 …)
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

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

© 2007 Pearson Ed - All rights reserved.
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.
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.
- Does not provide implementation.
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.
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.