Cpt S 122 – Data Structures

Course Review
Midterm Exam # 2

Nirmalya Roy

School of Electrical Engineering and Computer Science
Washington State University

Midterm Exam 2

- When: Monday (11/05) 12:10 pm -1pm
- Where: In Class

- Closed book, Closed notes
- Comprehensive

- Material for preparation:
  - Lecture Slides
  - Quizzes, Labs and Programming assignments
  - Deitel & Deitel book (Read and re-read Chapter 15 to 22 and Chapter 24)

Course Overview

- C++ as a better C; Introducing Object Technology (Chapter 15)
  - Inline Function
  - Function Overloading and Function Templates
  - Pass-by-value and Pass-by-reference
- Introduction to Classes, Objects & Strings (Chapter 16)
  - Data members, Members functions, `set` and `get` functions
  - Constructors
- Classes: A Deeper Look, Part I (Chapter 17)
  - Separating interface from implementation
  - Destructors

Course Overview

- Classes: A Deeper Look, Part 2 (Chapter 18)
  - `const` Objects and `const` Member functions
  - Composition: Objects as members of class
  - `friend` function and `friend` class
  - `this` pointer
- Operator Overloading; Class String (Chapter 19)
  - Implementation of operator overloading
  - Dynamic memory management using `new` operator
  - Explicit constructor

Course Overview

- Object Oriented Programming: Inheritance (Chapter 20)
  - Base Classes & Derived Classes
  - `public`, `protected`, and `private` Inheritance
- Object Oriented Programming: Polymorphism (Chap. 21)
  - Abstract Classes & pure `virtual` Functions
  - `virtual` Functions & Dynamic Binding
  - Polymorphism & RunTime Type Information (RTTI)
    - downcasting, dynamic_cast
  - `virtual` Destructors

Course Overview

- Templates (Chapter 22)
  - Function Template
  - Class Templates
  - STL Containers: example of container class template such as stack
- Exception Handling (Chapter 24)
  - Use of *try, catch* and *throw* to

    detect, handle and *indicate* exceptions, respectively.
  - Exception handling with constructors & destructors
  - Processing `new` failures

Constructor & Destructor

- Constructor is a special member function which enables an object to initialize itself when it is created
  - Name is same as the class name
  - Invoked whenever an object of its associated class is created
  - Constructs the values of the data members of the class
- Destructor is a special member function that destroys the objects when they are no longer required

Constructor (cont.)

```
class integer{
    int m,n;
    public:
    integer (void); //constructor
    …….
    };

    integer :: integer(void){ //constructor defined
    m = 0; n = 0;
    }
    integer int1; // object int1 created
```

Constructors (cont.)

- Not only creates the object int1 of type integer
  - But also initializes its data members m and n to zero.
  - No need to invoke the constructor function.
- A constructor that accepts no parameters is called a *default constructor*
  - The default constructor for class integer is
    - class integer :: integer();
  - If no such constructor is defined then compiler supplies a default constructor.

Parameterized Constructors

```
class integer{
    int m,n;
    public:
    integer (int x, int y); //parameterized constructor
    …….
    };

    integer :: integer(int x, int y){ //constructor defined
    m = x; n = y;
    }

    integer int1 (10, 100); //must pass the initial values
    when object int1 is declared; implicit call
     integer int1 = integer (10, 100); //explicit call
```

Multiple Constructors in a Class

- integer (); // No arguments
- integer (int, int); // with arguments

```
class integer{
    int m, n;
    public:
    integer (){ m = 0; n = 0;}  //constructor 1
    integer (int a; int b){ m = a; n = b;} //constructor 2
    integer (integer & i){ m = i.m; n = i.n;} //constructor 3
    };
```

```
integer  I1; // object I1 created
integer  I2 (20, 40); // object I2 created
integer  I3 (I2); // object I3 created
```

- copies the value of I2 into I3
- sets the value of every data element of I3 to value of corresponding data elements of I2.
- copy constructor

Copy Constructor

- A *copy constructor* is used to declare and initialize an object from another object
  - integer I3 (I2)
  - define object I3 and at the same time initialize it to the values of I2
  - Another form is: integer I3 = I2;
    - This process of initializing through a copy constructor is known as *copy initialization*
  - I3 = I2 ??
    - Will not invoke the copy constructor
    - However I3 and I2 are objects; the statement is legal and simply assign the values of I2 to I3; member by member.
    - This is the task of overloaded assignment operator (=)

Function Overloading

- C++ enables several functions of the same name to be defined, as long as they have different signatures.
  - This is called function overloading.
- The C++ compiler selects the proper function to call
  - examining the number, types and order of the arguments in the call.
- Overloaded functions are distinguished by their signatures.
  - A signature is a combination of a function's name and its parameter types (in order).
- Function overloading is used to create several functions of the same name
  - perform similar tasks, but on different data types.

Function Overloading

```cpp
// Fig. 6.24: fig06_24.cpp
// Overloaded functions.
#include <iostream>
using namespace std;

// function square for int values
int square( int x )
{
   cout << "square of integer " << x << " is ";
   return x * x;
} // end function square with int argument

// function square for double values
double square( double y )
{
   cout << "square of double " << y << " is ";
   return y * y;
} // end function square with double argument

```

Fig. 6.24 | Overloaded square functions. (Part 1 of 2.)

Example: Function Overloading

```
int main()
{
   cout << square( 7 ); // calls int version
   cout << endl;
   cout << square( 7.5 ); // calls double version
   cout << endl;
} // end main
```

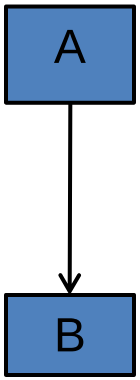
```
square of integer 7 is 49
square of double 7.5 is 56.25
```

Fig. 6.24 | Overloaded square functions. (Part 2 of 2.)

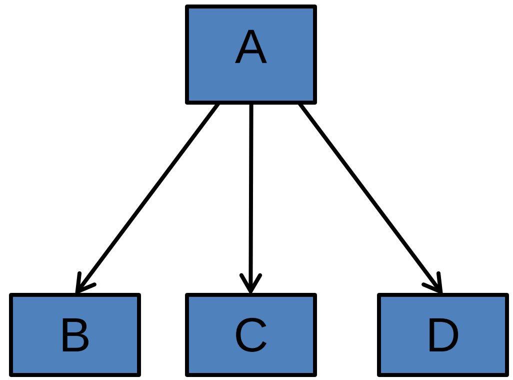
Inheritance

- With object-oriented programming, we focus on the commonalities among objects in the system rather than on the special cases.
- We distinguish between the is-a relationship and the *has-a* relationship.
- The *is-a* relationship represents inheritance.
  - In an *is-a* relationship, *an object of a derived class* also can be treated as *an object of its base class*.
- By contrast, the *has-a* relationship represents composition.

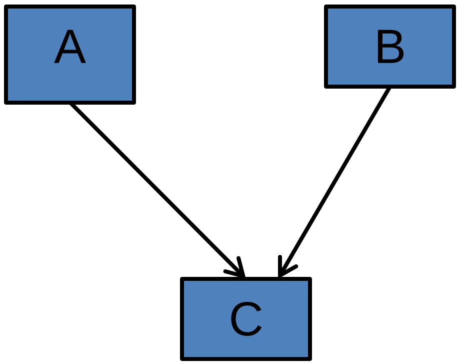
Variety of Inheritance



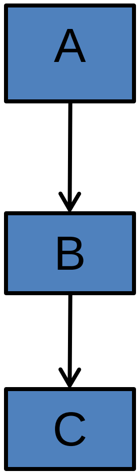
Single Inheritance



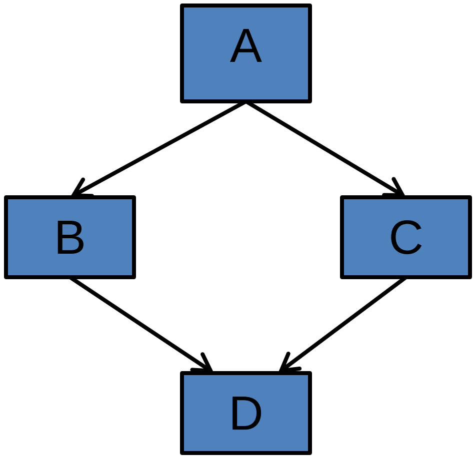
Hierarchical Inheritance



Multiple Inheritance



Multilevel Inheritance



Hybrid Inheritance

Derived class cannot access Base class private data directly but can access it through inherited member function

```
// calculate earnings
double BasePlusCommissionEmployee::earnings() const
{
   // derived class cannot access the base class's private data
   return baseSalary + ( commissionRate * grossSales );
} // end function earnings

// print BasePlusCommissionEmployee object
void BasePlusCommissionEmployee::print() const
{
   // derived class cannot access the base class's private data
   cout << "base-salaried commission employee: " << firstName << ' '
      << lastName << "\nsocial security number: " << socialSecurityNumber
      << "\ngross sales: " << grossSales
      << "\ncommission rate: " << commissionRate
      << "\nbase salary: " << baseSalary;
} // end function print
```

Fig. 12.11 | BasePlusCommissionEmployee implementation file: private base-class data cannot be accessed from derived class. (Part 3 of 5.)

Accessing private data in base-class using base-class member function

- The errors in `BasePlusCommissionEmployee` *could have been prevented* by using
  - the *get* member functions inherited from base class `CommissionEmployee`.
- For example, we could have invoked `getCommissionRate` and `getGrossSales` to access
  - `CommissionEmployee`'s `private` data members `commissionRate` and `grossSales`, respectively.

Dynamic Memory Management

- Control the allocation and deallocation of memory in a program
  - for objects and for arrays of any built-in or user-defined type.
  - known as dynamic memory management.
  - performed with `new` and `delete`.
- You can use the `new` operator to dynamically allocate (i.e., reserve) the exact amount of memory required to hold an object or array at execution time.
- The object or array is created in the free store (also called the heap)
  - a region of memory assigned to each program for storing dynamically allocated objects.
- Once memory is allocated in the free store, you can access it via the pointer that operator `new` returns.
- You can return memory to the free store by using the `delete` operator to deallocate it.

Dynamic Memory Management (cont.)

- To destroy a dynamically allocated object, use the `delete` operator as follows:
  - `delete ptr;`
- To deallocate a dynamically allocated array, use the statement
  - `delete [] ptr;`

What is `this` pointer?

- Every object has a special pointer "this" which points to the object itself.
- This pointer is accessible to *all members of the class but not to any static members* of the class.
- Can be used to find the *address of the object* in which the function is a member.
- Presence of this pointer is not included in the `sizeof` calculations.

Rule of Three (the Law of The Big Three or The Big Three)

- Rule of three is a Rule of thumb in C++ that claims that if a class defines one of the following
  - it should probably explicitly define all three.
- A copy constructor, a destructor, and an overloaded assignment operator
  - provided as a group for any class that uses dynamically allocated memory.
- Not providing a copy constructor, and an overloaded assignment operator for a class when objects of that class contain pointers to dynamically allocated memory is a logic error.

Implementation of Operator Overloading: Example: Array Class

```
// Fig. 11.10: Array.h
// Array class definition with overloaded operators.
#ifndef ARRAY_H
#define ARRAY_H

#include <iostream>
using namespace std;

class Array
{
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );
public:
   Array( int = 10 ); // default constructor
   Array( const Array & ); // copy constructor
   ~Array(); // destructor
   int getSize() const; // return size

   const Array &operator=( const Array & ); // assignment operator
   bool operator==( const Array & ) const; // equality operator

```

Fig. 11.10 | Array class definition with overloaded operators. (Part 1 of 2.)

Case Study: Array Class (cont.)

```
   // inequality operator; returns opposite of == operator
   bool operator!=( const Array &right ) const
   {
      return ! ( *this == right ); // invokes Array::operator==
   } // end function operator!=

   // subscript operator for non-const objects returns modifiable lvalue
   int &operator[]( int );

   // subscript operator for const objects returns rvalue
   int operator[]( int ) const;
private:
   int size; // pointer-based array size
   int *ptr; // pointer to first element of pointer-based array
}; // end class Array

#endif
```

Fig. 11.10 | Array class definition with overloaded operators. (Part 2 of 2.)

Default Constructor

```cpp
// Fig 11.11: Array.cpp
// Array class member- and friend-function definitions.
#include <iostream>
#include <iomanip>
#include <cstdlib> // exit function prototype
#include "Array.h" // Array class definition
using namespace std;

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   // validate arraySize
   if ( arraySize > 0 )
      size = arraySize;
   else
      throw invalid_argument( "Array size must be greater than 0" );

   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; ++i )
      ptr[ i ] = 0; // set pointer-based array element
} // end Array default constructor
```

Fig. 11.11 | Array class member- and friend-function definitions. (Part 1 of 8.)

Default Constructor Explanation

- Declares the *default constructor* for the class and specifies a default size of 10 elements.
- The default constructor validates and assigns the argument to data member `size`,
  - uses `new` to obtain the memory for the internal pointer-based representation of this array
  - assigns the pointer returned by `new` to data member `ptr`.
- Then the constructor uses a `for` statement to set all the elements of the array to zero.

Copy Constructor for class Array

```
// copy constructor for class Array;
// must receive a reference to prevent infinite recursion
Array::Array( const Array &arrayToCopy )
   : size( arrayToCopy.size )
{
   ptr = new int[ size ]; // create space for pointer-based array

   for ( int i = 0; i < size; ++i )
      ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
} // end Array copy constructor

```

Fig. 11.11 | Array class member- and friend-function definitions. (Part 2 of 8.)

Copy Constructor Explanation

- Declares a *copy constructor* that initializes an `Array` by making a copy of an existing `Array` object.
- *Such copying must be done carefully to avoid the pitfall of leaving both `Array` objects pointing to the same dynamically allocated memory.*
- Copy constructors are *invoked* whenever a copy of an object is needed
  - such as in passing an object by value to a function,
  - returning an object by value from a function or
  - initializing an object with a copy of another object of the same class.

Copy Constructor Explanation

- The copy constructor for `Array` uses *a member initializer to copy the `size` of the initializer `Array` into data member `size`,*
  - uses `new` to obtain the memory for the internal pointer-based representation of this `Array`
  - assigns the pointer returned by `new` to data member `ptr`.
- Then the copy constructor uses a `for` statement to copy all the elements of the initializer `Array` into the new `Array` object.
- An object of a class can look at the `private` data of any other object of that class (using a handle that indicates which object to access).

Infinite Recursion of Copy Constructor

- A copy constructor must receive its argument *by reference, not by value*.
- Otherwise the copy constructor call results in infinite recursion
  - Receiving an object by value requires a copy constructor to make a copy of the argument object.
  - Recall that any time a copy of an object is required, the class's copy constructor is called.
  - If the copy constructor received its argument by value, the copy constructor would call itself recursively to make a copy of its argument!

Destructor for class Array

```
// destructor for class Array
Array::~Array()
{
   delete [] ptr; // release pointer-based array space
} // end destructor

// return number of elements of Array
int Array::getSize() const
{
   return size; // number of elements in Array
} // end function getSize

```

Fig. 11.11 | Array class member- and friend-function definitions. (Part 3 of 8.)

Destructor Explanation

- The destructor uses `delete []` to release the memory allocated dynamically by `new` in the constructor.

Equality Operator for class Array

```
// determine if two Arrays are equal and
// return true, otherwise return false
bool Array::operator==( const Array &right ) const
{
   if ( size != right.size )
      return false; // arrays of different number of elements

   for ( int i = 0; i < size; ++i )
      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // Array contents are not equal

   return true; // Arrays are equal
} // end function operator==

```

Fig. 11.11 | Array class member- and friend-function definitions. (Part 5 of 8.)

Explanation for Equality Operator

- Overloaded equality operator (==) for the class.
- When the compiler sees the expression `integers1 == integers2`, the compiler invokes member function `operator==` with the call
  - `integers1.operator==( integers2 )`
- Member function `operator==` immediately returns `false` if the `size` members of the arrays are not equal.
- Otherwise, `operator==` compares each pair of elements.
  - If they're all equal, the function returns `true`.
  - The first pair of elements to differ causes the function to return `false` immediately.

Overloaded Assignment Operator

```
// overloaded assignment operator;
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
   if ( &right != this ) // avoid self-assignment
   {
      // for Arrays of different sizes, deallocate original
      // left-side array, then allocate new left-side array
      if ( size != right.size )
      {
         delete [] ptr; // release space
         size = right.size; // resize this object
         ptr = new int[ size ]; // create space for array copy
      } // end inner if

      for ( int i = 0; i < size; ++i )
         ptr[ i ] = right.ptr[ i ]; // copy array into object
   } // end outer if

   return *this; // enables x = y = z, for example
} // end function operator=

```

Fig. 11.11 | Array class member- and friend-function definitions. (Part 4 of 8.)

Explanation for Overloaded Assignment Operator

- Overloaded assignment operator function for the Array class.
- When the compiler sees the expression `integers1 = integers2`, the compiler invokes member function `operator=` with the call
  - `integers1.operator=( integers2 )`
- Member function `operator=`'s implementation tests for self-assignment in which an `Array` object is being assigned to itself.
  - if `this` is equal to the `right` operand's address, a self-assignment is being attempted, so the assignment is skipped.

Explanation of Overloaded Assignment Operator (cont.)

- `operator=` determines whether the sizes of the two arrays are identical
  - the original array of integers in the left-side `Array` object is not reallocated.
- Otherwise, `operator=` uses `delete`
  - to release the memory,
  - copies the `size` of the source array to the `size` of the target array,
  - uses `new` to allocate memory for the target array and
  - places the pointer returned by `new` into the array's `ptr` member.
- Regardless of whether this is a self-assignment, the member function returns the current object (i.e., `*this`) as a constant reference;
  - this enables cascaded `Array` assignments such as `x = y = z`,
  - prevents ones like `(x = y) = z` because `z` cannot be assigned to the `const Array-` reference that is returned by `(x = y)`.

Overloaded Inequality Operator

```
// inequality operator; returns opposite of == operator
bool operator!=( const Array &right ) const
{
   return ! ( *this == right ); // invokes Array::operator==
} // end function operator!=
```

Explanation of Overloaded Inequality Operator

- Overloaded inequality operator (`!=`).
- Member function `operator!=` uses the overloaded `operator==` function to determine whether one `Array` is equal to another, then returns the opposite of that result.
- Writing `operator!=` in this manner enables you to reuse `operator==`, which *reduces the amount of code that must be written in the class*.
- Full function definition for `operator!=` allows the compiler to inline the definition.

explicit Constructors

- Any single-argument constructor can be used by the compiler to perform an implicit conversion.
  - The constructor's argument is converted to an object of the class in which the constructor is defined.
- The conversion is automatic and you need not use a cast operator.
- *In some situations, implicit conversions are undesirable or error-prone.*
  - For example, our `Array` class defines a constructor that takes a single `int` argument.
  - The intent of this constructor is to create an *`Array` object* containing the number of elements specified by the `int` argument.
  - However, this constructor can be misused by the compiler to perform an *implicit* conversion.

Polymorphism

- One name, multiple forms
  - Overloaded function, overloaded operators
  - Overloaded member functions are selected for invoking by matching argument, both *type and number*
  - Information is known to the compiler at *compile time*
    - Compiler is able to select the appropriate function at the compile time
  - This is called *early binding, or static binding, or static linking*
    - An object is bound to its function call at compile time
  - This is also known as *compile time polymorphism*

Polymorphism (cont.)

- Consider the following class definition where the function name and prototype is same in both the base and derived classes.

```
class A{
        int x;
        public:
          void show() {…} //show() in base class
};
class B: public A{
        int y;
        public:
          void show() {…} //show() in derived class
};
```

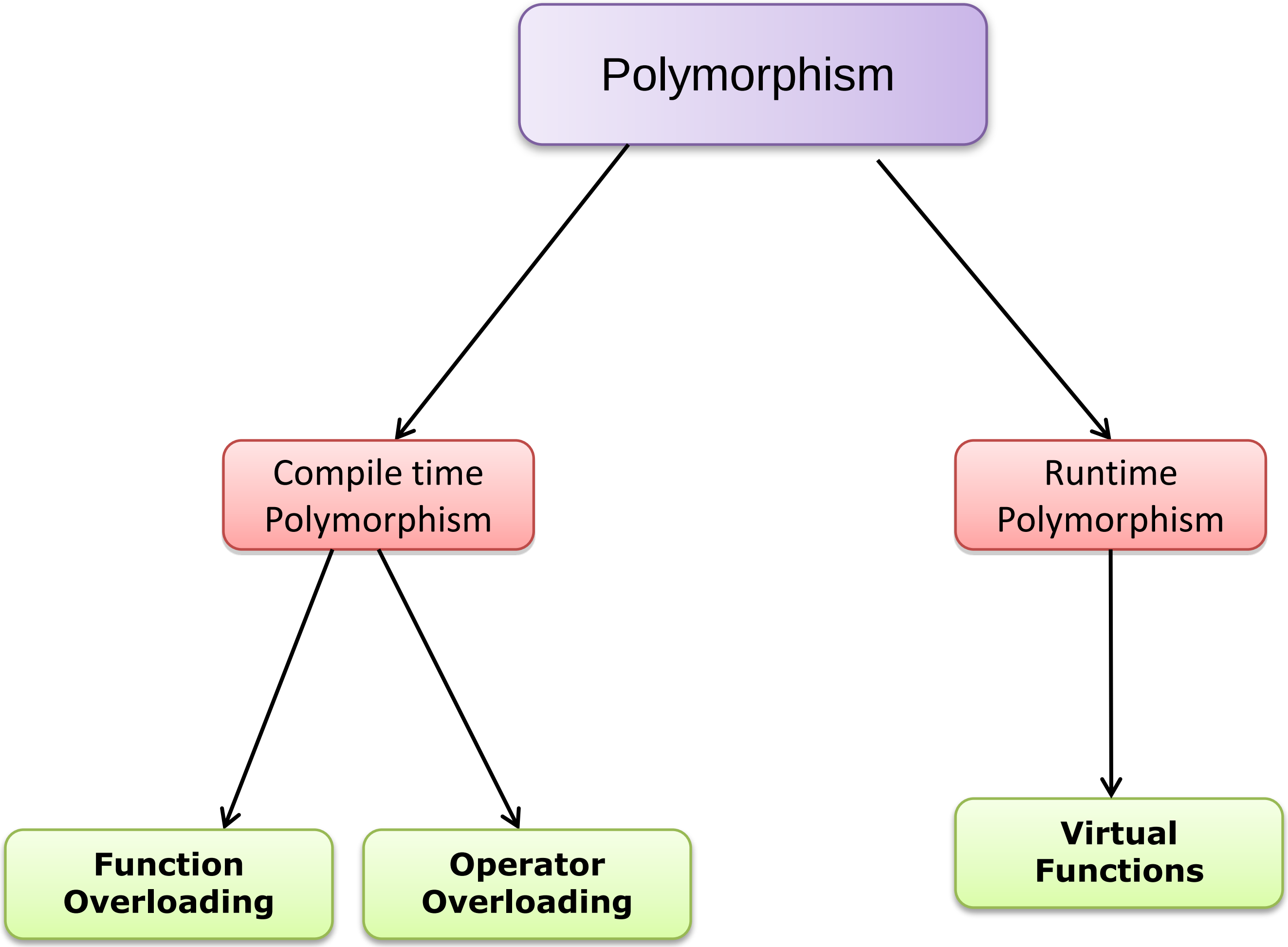
Polymorphism (cont.)

- How do we use the member function show() to print the values of objects of both the classes A and B?
  - prototype show() is same in both the places.
  - The function is not overloaded and therefore static binding does not apply.
- It would be nice if appropriate member function could be selected while the program is running
  - This is known as runtime polymorphism
  - How could it happen?
    - C++ supports a mechanism known as virtual function to achieve runtime polymorphism
    - At run time, when it is known what class objects are under consideration, the appropriate version of the function is called.

Polymorphsim (cont.)

- Function is linked with a particular class much later after the compilation, this processed is termed as *late binding*
  - It is also known as *dynamic binding* because the selection of the appropriate function is done dynamically at runtime.
- Dynamic binding is one of the powerful feature in C++
  - Requires the use of pointers to objects
  - Object pointers and virtual functions are used to implement dynamic binding or runtime polymorphism

Polymorphism



Relationships Among Objects in an Inheritance Hierarchy

- Demonstrate how base-class and derived-class pointers can be aimed at base-class and derived-class objects
  - how those pointers can be used to invoke member functions that manipulate those objects.
- A key concept
  - an object of a derived class can be treated as an object of its base class.
  - the compiler allows this because each derived-class object *is an* object of its base class.
- However, we cannot treat a base-class object as an object of any of its derived classes.
- The *is-a* relationship applies only from a derived class to its direct and indirect base classes.

Virtual Function

- `virtual` function invocation through
  - a base-class pointer to a derived-class object
  - a base-class reference to a derived-class object
  - the program will choose the correct derived-class function dynamically (i.e., at execution time) ***based on the object type***
    - *not the pointer or reference type.*
  - This is known as dynamic binding or late binding.

Abstract Classes and pure virtual Functions

- A class is made abstract by declaring one or more of its `virtual` functions to be "pure."
  - A pure `virtual` function is specified by placing "= 0" in its declaration, as in

```
virtual void draw() const = 0; // pure virtual function
```

- The "= 0" is a pure specifier.
- Pure `virtual` functions *do not provide implementations*.

Abstract Classes and pure virtual Functions

- There are cases in which it’s useful to define *classes from which you never intend to instantiate any objects*.
  - Such classes are called **abstract classes**.
  - These classes normally are used as base classes in inheritance hierarchies
- These classes *cannot be used to instantiate objects*, because, abstract classes are *incomplete*
  - derived classes must define the “missing pieces.”
- An abstract class provides a base class from which other classes can inherit.
- Classes that can be used to instantiate objects are called **concrete classes**.
  - Such classes *define every member function* they declare.

Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

- Internal implementation of polymorphism, `virtual` functions and dynamic binding.
- Appreciate the overhead of polymorphism due to its elegant data structure
- Polymorphism is accomplished through three levels of pointers (i.e., "triple indirection").
- C++ compiles a class that has one or more `virtual` functions
  - builds a virtual function table (*vtable*) for that class.
- An executing program uses the *vtable* to select the proper function implementation each time a `virtual` function of that class is called.

Virtual function working mechanism

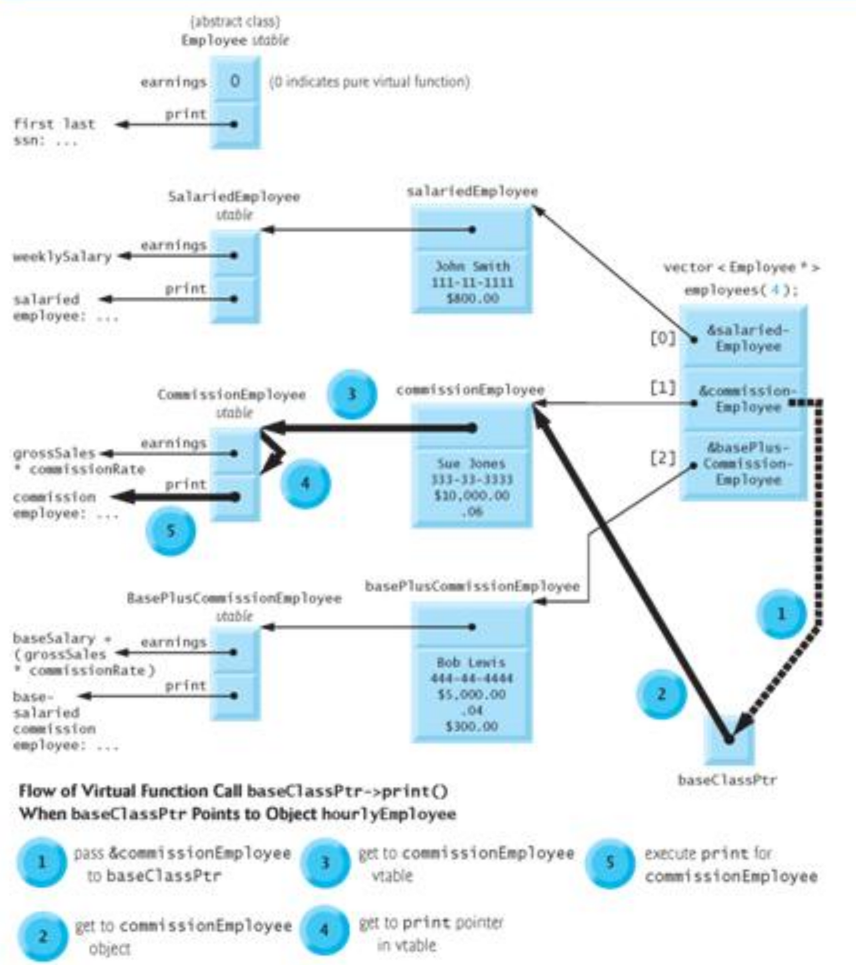


Fig. 13.18 | How virtual function calls work.

Polymorphism and Runtime Type Information with Downcasting, dynamic_cast, typeid and type_info

- Demonstrate the powerful capabilities of runtime type information (RTTI) and dynamic casting,
  - enable a program to determine the type of an object at execution time and act on that object accordingly.
- To accomplish this, we use operator **dynamic_cast** to determine whether the type of each object is derived class i.e; `BasePlusCommissionEmployee`.
  - This is the ***downcast*** operation.
  - Dynamically downcast `base-class or abstract base-class` pointer/reference `i.e;`
    - `employees[i]` from type `Employee *` to type `BasePlusCommissionEmployee *`.

Observations

- If a class has a virtual function; provide *a virtual destructor*, even if one is not required for the class.
  - ensure that a custom derived-class destructor will be invoked (if there is one) when a derived-class object is deleted via a base class pointer
- *Constructor cannot be virtual*
  - Declaring a constructor virtual is a compilation error.

Templates

- Function templates and class templates enable to specify, with a single code segment,
  - an entire range of related (overloaded) functions
    - function-template specializations
  - an entire range of related classes
    - class-template specializations.
- This technique is called generic programming.
- Note the distinction between *templates* and *template specializations*:
  - *Function templates* and *class templates* are like stencils out of which we trace shapes.
  - *Function-template specializations* and *class-template specializations* are like the separate tracings that all have the *same shape*, but could, for example, be drawn in *different colors.*

What is Function Template?

- All function template definitions begin with the `template` keyword followed by
  - a template parameter list to the function template enclosed in angle brackets (< and >).
- Every parameter in the template parameter list is preceded by keyword `typename` or keyword `class`.
- The formal type parameters are placeholders for fundamental types or user-defined types.
- These placeholders are used to specify the types of the function's parameters,
  - to specify the function's return type and
  - to declare variables within the body of the function definition.

Example: Function Templates

```
// Fig. 6.26: maximum.h
// Definition of function template maximum.
template < typename T >  // or template< typename T >
T maximum( T value1, T value2, T value3 )
{
   T maximumValue = value1; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if ( value2 > maximumValue )
      maximumValue = value2;

   // determine whether value3 is greater than maximumValue
   if ( value3 > maximumValue )
      maximumValue = value3;

   return maximumValue;
} // end function template maximum
```

Fig. 6.26 | Function template maximum header.

Why Function Templates & How it works

- If the program logic and operations are identical for each data type
  - overloading may be performed more compactly and conveniently by using function templates.
- When the compiler detects a `templated` function invocation in the client program,
  - the compiler uses its ***overload resolution capabilities*** to find a definition of function that best matches the function call.

STL: Containers

- Standard Template Library: Containers
  - *A container is a holder object that stores a collection of other objects (its elements).*
  - Implemented as *class templates*, which allows a great flexibility in the types supported as elements.
- Containers replicate structures very commonly used in programming:
  - *dynamic arrays (vector), queues (queue), stacks (stack), heaps (priority_queue), linked lists (list), trees (set), associative arrays (map) etc*
- The container manages the storage space for its elements
  - provides member functions to access them, either directly or through iterators (reference objects like pointers).

Exception Handling

- What is exception handling?
  - Example: Handling an attempt to divide by zero
  - Use *try, catch* and *throw* to *detect, handle* and *indicate* exceptions, respectively.
  - Rethrowing an exception
- Exception Specifications
  - Processing unexpected and uncaught exceptions
- Stack unwinding
  - enables exceptions not caught in one scope to be caught in another

Exception Handling

- Constructors, destructors & exception handling
- Processing `new` failures
  - Dynamic memory allocation
  - Use `unique_ptr` to prevent memory leak
- Exception & Inheritance
  - Understand the exception inheritance hierarchy

Tentative Midterm Exam#2 Structure

- Part I: Conceptual Questions
  - Short answer, Fill-in-the-blank, and True/False (30 pts)
  - Go though the self-review exercises at the end of each chapter
- Part II: Programming Questions
  - Write C++ code (70 pts)
  - Programming questions
    - Retake Quiz 3 and Quiz 4
    - Inheritance, Operator overloading, Polymorphism, and Templates

- Special office hours on Monday (11/05) morning for the exam
  - From 9 am to 12:00 pm, EME 127

Good Luck !