IS 0020
Program Design and Software Tools

Introduction to C++ Programming

Operator Overloading, Inheritance

Lecture 5

June 21, 2004

Fundamentals of Operator Overloading

- Use operators with objects (operator overloading)
  - Clearer than function calls for certain classes
  - Operator sensitive to context
- Types
  - Built in (**`int`**, **`char`**) or user-defined
  - Can use existing operators with user-defined types
    - Cannot create new operators
- Overloading operators
  - Create a function for the class
  - Name function **`operator`** followed by symbol
    - **`Operator+`** for the addition operator **`+`**

Fundamentals of Operator Overloading

- Using operators on a class object
  - It must be overloaded for that class
  - Exceptions:
    - Assignment operator, **=**
      - May be used without explicit overloading
      - Memberwise assignment between objects
    - Address operator, **&**
      - May be used on any class without overloading
      - Returns address of object
    - Both can be overloaded

Restrictions on Operator Overloading

- Cannot change
  - How operators act on built-in data types
    - I.e., cannot change integer addition
  - Precedence of operator (order of evaluation)
    - Use parentheses to force order-of-operations
  - Associativity (left-to-right or right-to-left)
  - Number of operands
    - **&** is unitary, only acts on one operand
- Cannot create new operators
- Operators must be overloaded explicitly
  - Overloading **+** does not overload **+=**

Restrictions on Operator Overloading

<table>
<tr><th colspan="8">Operators that can be overloaded</th></tr>
<tr><td>+</td><td>-</td><td>*</td><td>/</td><td>%</td><td>^</td><td>&</td><td>|</td></tr>
<tr><td>~</td><td>!</td><td>=</td><td><</td><td>></td><td>+=</td><td>-=</td><td>*=</td></tr>
<tr><td>/=</td><td>%=</td><td>^=</td><td>&=</td><td>|=</td><td><<</td><td>>></td><td>>>=</td></tr>
<tr><td><<=</td><td>==</td><td>!=</td><td><=</td><td>>=</td><td>&&</td><td>||</td><td>++</td></tr>
<tr><td>--</td><td>->*</td><td>,</td><td>-></td><td>[]</td><td>()</td><td>new</td><td>delete</td></tr>
<tr><td>new[]</td><td>delete[]</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Operators that cannot be overloaded				
.	.*	::	?:	sizeof

Operator Functions As Class Members Vs. As Friend Functions

- Operator functions
  - Member functions
    - Use **`this`** keyword to implicitly get argument
    - Gets left operand for binary operators (like **`+`**)
    - Leftmost object must be of same class as operator
  - Non member functions
    - Need parameters for both operands
    - Can have object of different class than operator
    - Must be a **`friend`** to access **`private`** or **`protected`** data
- Example Overloaded **`<<`** operator
  - Left operand of type **`ostream &`**
    - Such as **`cout`** object in **`cout << classObject`**
  - Similarly, overloaded **`>>`** needs **`istream &`**
  - Thus, both must be non-member functions

Operator Functions As Class Members Vs. As Friend Functions

- Commutative operators
  - May want **`+`** to be commutative
    - So both "**`a + b`**" and "**`b + a`**" work
  - Suppose we have two different classes
  - Overloaded operator can only be member function when its class is on left
    - **`HugeIntClass + Long int`**
    - Can be member function
  - When other way, need a non-member overload function
    - **`Long int + HugeIntClass`**

Overloading Stream-Insertion and Stream-Extraction Operators

- **<<** and **>>**
  - Already overloaded to process each built-in type
  - Can also process a user-defined class
- Example program
  - Class **PhoneNumber**
    - Holds a telephone number
  - Print out formatted number automatically
    - **(123) 456-7890**

Outline

fig08_03.cpp
(1 of 3)

```
// Fig. 8.3: fig08_03.cpp
// Overloading the stream-insertion and
// stream-extraction operators.
#include <iostream>

using std::cout;
using std::cin;
using std::endl;
using std::ostream;
using std::istream;

#include <iomanip>

using std::setw;

// PhoneNumber class definitio
class PhoneNumber {
   friend ostream &operator<<(
   friend istream &operator>>(

private:
   char areaCode[ 4 ];  // 3-d
   char exchange[ 4 ];  // 3-digit exchange and null
   char line[ 5 ];      // 4-digit line and null

}; // end class PhoneNumber
```

Notice function prototypes for overloaded operators **>>** and **<<**

They must be non-member **friend** functions, since the object of class **Phonenumber** appears on the right of the operator.

cin << object
cout >> object

```
// overloaded stream-insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<( ostream &output, const PhoneNumber &num )
{
   output << "(" << num.areaCode << ") "
          << num.exchange << "-" << num.line;

   return output;     // enables cout << a << b << c;

} // end function operator<<

// overloaded stream-extraction operator; cannot be
// a member function if we would like to invoke it
// cin >> somePhoneNumber;
istream &operator>>( istream &input, PhoneNumber &n
{
   input.ignore();                         // skip (
   input >> setw( 4 ) >> num.areaCode; // input are
   input.ignore( 2 );                      // skip ) a
   input >> setw( 4 ) >> num.exchange;
   input.ignore();
   input >> setw( 5 ) >> num.line;

   return input;      // enables cin >
```

The expression:
`cout << phone;`
is interpreted as the function call:
`operator<<(cout, phone);`

`output` is an alias for **`cout.`**

This allows objects to be cascaded.
`cout << phone1 << phone2;`
first calls
`operator<<(cout, phone1)`, and returns **`cout`**.

Next, **`cout << phone2`** executes.

Stream manipulator **`setw`** restricts number of characters read. **`setw(4)`** allows 3 characters to be read, leaving room for the null character.

```
} // end function operator>>

int main()
{
   PhoneNumber phone; // create object phone

   cout << "Enter phone number in the form (123) 456-7890:\n";

   // cin >> phone invokes operator>> by implicitly issuing
   // the non-member function call operator>>( cin, phone )
   cin >> phone;

   cout << "The phone number entered was: " ;

   // cout << phone invokes operator<< by implicitly issuing
   // the non-member function call operator<<( cout, phone )
   cout << phone << endl;

   return 0;

} // end main
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

Overloading Unary Operators

- Overloading unary operators
  - Non-**`static`** member function, no arguments
  - Non-member function, one argument
    - Argument must be class object or reference to class object
  - Remember, **`static`** functions only access **`static`** data

Overloading Operators

- Overloading unary operators (**`!`** to test for empty string)
  - Non-**`static`** member function: **`!s`** becomes **`s.operator!()`**

    ```
    bool operator!() const;
    ```
  - Non-member function: **`s!`** becomes **`operator!(s)`**

    ```
    friend bool operator!( const String & )
    ```
- Overloading binary operators
  - Non-member function (arg. must be class object or reference)

    ```
    friend const String &operator+=(String &, const
    String & );
    ```
  - Non-**`static`** member function:

    ```
    const String &operator+=( const String & );
    ```

  - **`y += z`** equivalent to **`y.operator+=( z )`**

Case Study: Array class

- Arrays in C++
  - No range checking
  - Cannot be compared meaningfully with **`==`**
  - No array assignment (array names **`const`** pointers)
  - Cannot input/output entire arrays at once
- Example:Implement an **`Array`** class with
  - Range checking
  - Array assignment
  - Arrays that know their size
  - Outputting/inputting entire arrays with **`<<`** and **`>>`**
  - Array comparisons with **`==`** and **`!=`**

Case Study: Array class

- Copy constructor
  - Used whenever copy of object needed
    - Passing by value (return value or parameter)
    - Initializing an object with a copy of another
      - **`Array newArray( oldArray );`**
      - **`newArray`** copy of **`oldArray`**
  - Prototype for class **`Array`**
    - **`Array( const Array & );`**
    - *Must* take reference
      - Otherwise, pass by value
      - Tries to make copy by calling copy constructor…
      - Infinite loop

```
// Fig. 8.4: array1.h
// Array class for storing arrays of integers.
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream>

using std::ostream;
using std::istream;

class Array {
   friend ostream &operator<<( ostream &, const Array & );
   friend istream &operator>>( istream &, Array & );

public:
   Array( int = 10 );       // default con
   Array( const Array & );  // copy constru
   ~Array();                // destructor
   int getSize() const;     // return size

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

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

```

Most operators overloaded as member functions (except **<<** and **>>**, which must be non-member functions).

Prototype for copy constructor.

```
   // 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-con
   int &operator[]( int );

   // subscript operator for const objects returns rvalue
   const int &operator[]( int ) const;

private:
   int size; // array size
   int *ptr; // pointer to first element of array

}; // end class Array

#endif
```

!= operator simply returns opposite of **==** operator. Thus, only need to define the **==** operator.

array1.h (2 of 2)

```
// Fig 8.5: array1.cpp
// Member function definitions for class Array
#include <iostream>

using std::cout;
using std::cin;
using std::endl;

#include <iomanip>

using std::setw;

#include <new>       // C++ standard "new" operator

#include <cstdlib>   // exit function prototype

#include "array1.h"  // Array class definition

// default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   // validate arraySize
   size = ( arraySize > 0 ? arraySize : 10 );

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

```

array1.cpp (1 of 7)

```
   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0;          // initialize array

} // end Array default constructor

// copy constructor for class Array;
// must receive a reference to pre
Array::Array( const Array &arrayTo
   : size( arrayToCopy.size )
{
   ptr = new int[ size ]; // create space for array

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

} // end Array copy constructor

// destructor for class Array
Array::~Array()
{
   delete [] ptr;  // reclaim array space

} // end destructor
```

We must declare a new integer array so the objects do not point to the same memory.

```
// return size of array
int Array::getSize() const
{
   return size;

} // end function getSize

// overloaded assignment operator
// const return avoids: ( a1 = a2 ) = a3
const Array &Array::operator=( const Array &right )
{
   if ( &right != this ) {  // check for self-assignment

      // for arrays of different sizes, deallocate original
      // left-side array, then allocate new left-side array
      if ( size != right.size ) {
         delete [] ptr;         // reclaim 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
```

Want to avoid self-assignment.

```

   return *this;   // enables x = y = z, for example

} // end function operator=

// 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 sizes

   for ( int i = 0; i < size; i++ )

      if ( ptr[ i ] != right.ptr[ i ] )
         return false; // arrays are not equal

   return true;        // arrays are equal

} // end function operator==

```

Outline

array1.cpp (5 of 7)

```
// overloaded subscript operator for non-const Arrays
// reference return creates an lvalue
int &Array::operator[]( int subscript )
{
   // check for subscript out of range error
   if ( subscript < 0 || subscript >= size
      cout << "\nError: Subscript " << subs
           << " out of range" << endl;

      exit( 1 );  // terminate program; subscript out of range

   } // end if

   return ptr[ subscript ]; // reference return

} // end function operator[]

```

`integers1[5]` calls **`integers1.operator[]( 5 )`**

`exit()` (header **`<cstdlib>`**) ends the program.

```
// overloaded subscript operator for const Arrays
// const reference return creates an rvalue
const int &Array::operator[]( int subscript ) const
{
   // check for subscript out of range error
   if ( subscript < 0 || subscript >= size ) {
      cout << "\nError: Subscript " << subscript
           << " out of range" << endl;

      exit( 1 );  // terminate program; subscript out of range

   } // end if

   return ptr[ subscript ]; // const reference return

} // end function operator[]

// overloaded input operator for class Array;
// inputs values for entire array
istream &operator>>( istream &input, Array &a )
{
   for ( int i = 0; i < a.size; i++ )
      input >> a.ptr[ i ];

   return input;   // enables cin >> x >> y;

} // end function
```

array1.cpp (7 of 7)

```

// overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;

   // output private ptr-based array
   for ( i = 0; i < a.size; i++ ) {
      output << setw( 12 ) << a.ptr[ i ];

      if ( ( i + 1 ) % 4 == 0 ) // 4 numbers per row of output
         output << endl;

   } // end for

   if ( i % 4 != 0 )  // end last line of output
      output << endl;

   return output;   // enables cout << x << y;

} // end function operator<<
```

Converting between Types

- Cast operator (conversion operator)
  - Convert from One class to another built-in type
  - Must be non-**`static`** member function -
    - Cannot be **`friend`**
  - Do not specify return type
    - Implicitly returns type to which you are converting
  - Example: **`A::operator char *() const;`**
    - Casts class **`A`** to a temporary **`char *`**
    - **`(char *)s`** calls **`s.operator char*()`**

```
A::operator int() const;
A::operator OtherClass() const;
```

- Casting can prevent need for overloading
  - Suppose class **`String`** can be cast to **`char *`**
  - **`cout << s;`** // **`cout`** expects **`char *`**; **`s`** is a **`String`**
    - Compiler implicitly calls the function to convert **`s`** to **`char *`**
    - Do not have to overload **`<< for String`**

Case Study: A `String` Class

- Build class **`String`**
  - String creation, manipulation
  - Class **`string`** in standard library (more Chapter 15)
- Conversion constructor
  - Single-argument constructor
  - Turns objects of other types into class objects
    - **`String s1(“hi”);`**
    - Creates a **`String`** from a **`char *`**
  - Any single-argument constructor is a conversion constructor

Overloading ++ and --

- Increment/decrement operators can be overloaded
  - Add 1 to a **`Date`** object, **`d1`**
  - Prototype (member function)
    - **`Date &operator++();`**
    - **`++d1`** same as **`d1.operator++()`**
  - Prototype (non-member)
    - **`Friend Date &operator++( Date &);`**
    - **`++d1`** same as **`operator++( d1 )`**

Overloading ++ and - -

- To distinguish pre/post increment
  - Post increment has a dummy parameter
    - **`int`** of **`0`**
  - Prototype (member function)
    - **`Date operator++( int );`**
    - **`d1++`** same as **`d1.operator++( 0 )`**
  - Prototype (non-member)
    - **`friend Date operator++( Data &, int );`**
    - **`d1++`** same as **`operator++( d1, 0 )`**
  - Integer parameter does not have a name
    - Not even in function definition

Overloading ++ and --

- Return values
  - Preincrement
    - Returns by reference (**`Date &`**)
    - lvalue (can be assigned)
  - Postincrement
    - Returns by value
    - Returns temporary object with old value
    - rvalue (cannot be on left side of assignment)
- Example **`Date`** class
  - Overloaded increment operator
    - Change day, month and year
  - Overloaded **`+=`** operator
  - Function to test for leap years
  - Function to determine if day is last of month

date1.h (1 of 2)

```
// Fig. 8.10: date1.h
// Date class definition.
#ifndef DATE1_H
#define DATE1_H
#include <iostream>

using std::ostream;

class Date {
   friend ostream &operator<<( ostream &, const Date & );

public:
   Date( int m = 1, int d = 1, int y =
   void setDate( int, int, int ); // s

   Date &operator++();              // preincrement operator
   Date operator++( int );          // postincrement operator

   const Date &operator+=( int ); // add days, modify object

   bool leapYear( int ) const;    // is this a leap year?
   bool endOfMonth( int ) const;  // is this end of month?
```

Note difference between pre and post increment.

date1.h (2 of 2)

```

private:
   int month;
   int day;
   int year;

   static const int days[];       // array of days per month
   void helpIncrement();          // utility function

}; // end class Date

#endif
```

```
Date &Date::operator++()
 {
    helpIncrement();
   return *this;  // reference return to create an lvalue
} // end function operator++

// overloaded postincrement operator; note that the dummy
// integer parameter does not have a parameter name
Date Date::operator++( int )
{
   Date temp = *this;  // hold current state of object
   helpIncrement();
   // return unincremented, saved, temporary object
   return temp;   // value return; not a reference return
} // end function operator++
```

Inheritance

- Inheritance
  - Software reusability
  - Create new class from existing class
    - Absorb existing class's data and behaviors
    - Enhance with new capabilities
  - Derived class inherits from base class
    - Derived class
      - More specialized group of objects
      - Behaviors inherited from base class
        - Can customize
      - Additional behaviors

Inheritance

- Class hierarchy
  - Direct base class
    - Inherited explicitly (one level up hierarchy)
  - Indirect base class
    - Inherited two or more levels up hierarchy
  - Single inheritance
    - Inherits from one base class
  - Multiple inheritance
    - Inherits from multiple base classes
      - Base classes possibly unrelated
    - Chapter 22

Inheritance

- Three types of inheritance
  - **`public`**
    - Every object of derived class also object of base class
      - Base-class objects not objects of derived classes
      - Example: All cars vehicles, but not all vehicles cars
    - Can access non-**`private`** members of base class
      - Derived class can effect change to **`private`** base-class members
        - Through inherited non-**`private`** member functions
  - **`private`**
    - Alternative to composition
    - Chapter 17
  - **`protected`**
    - Rarely used

Inheritance

- Abstraction
  - Focus on commonalities among objects in system
- "is-a" vs. "has-a"
  - "is-a"
    - Inheritance
    - Derived class object treated as base class object
    - Example: Car *is a* vehicle
      - Vehicle properties/behaviors also car properties/behaviors
  - "has-a"
    - Composition
    - Object contains one or more objects of other classes as members
    - Example: Car *has a* steering wheel

Base Classes and Derived Classes

- Base classes and derived classes
  - Object of one class "is an" object of another class
    - Example: Rectangle is quadrilateral.
  - Base class typically represents larger set of objects than derived classes
    - Example:
      - Base class: **`Vehicle`**
        - Cars, trucks, boats, bicycles, …
      - Derived class: **`Car`**
        - Smaller, more-specific subset of vehicles

Base Classes and Derived Classes

- Inheritance examples

Base class	Derived classes
`Student`	`GraduateStudent` `UndergraduateStudent`

`Shape`	`Circle`
	`Triangle`
	`Rectangle`

`Loan`	`CarLoan`
	`HomeImprovementLoan`
	`MortgageLoan`

`Employee`	`FacultyMember` `StaffMember`
`Account`	`CheckingAccount` `SavingsAccount`

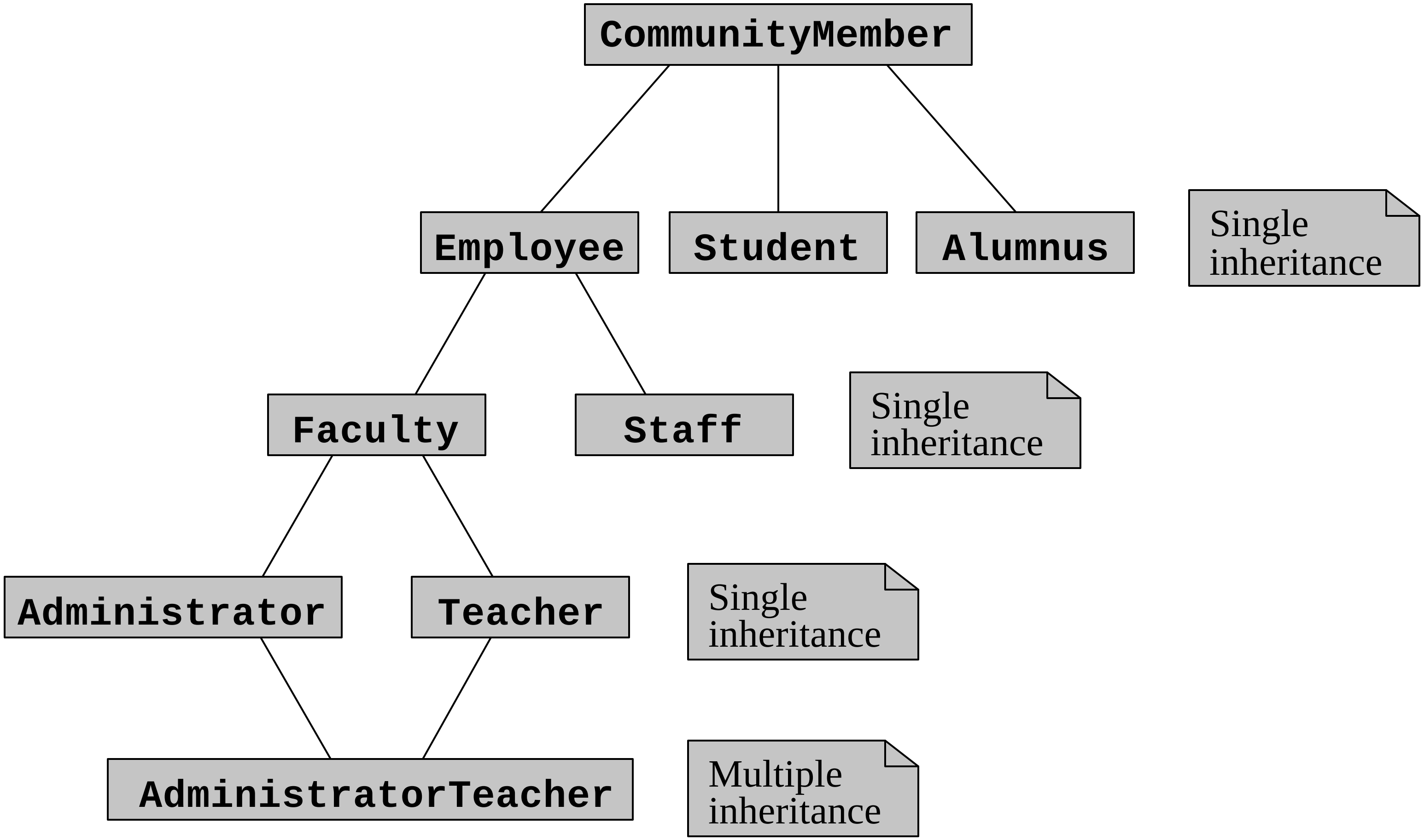
Base Classes and Derived Classes

- Inheritance hierarchy
  - Inheritance relationships: tree-like hierarchy structure
  - Each class becomes
    - Base class
      - Supply data/behaviors to other classes

    OR
    - Derived class
      - Inherit data/behaviors from other classes

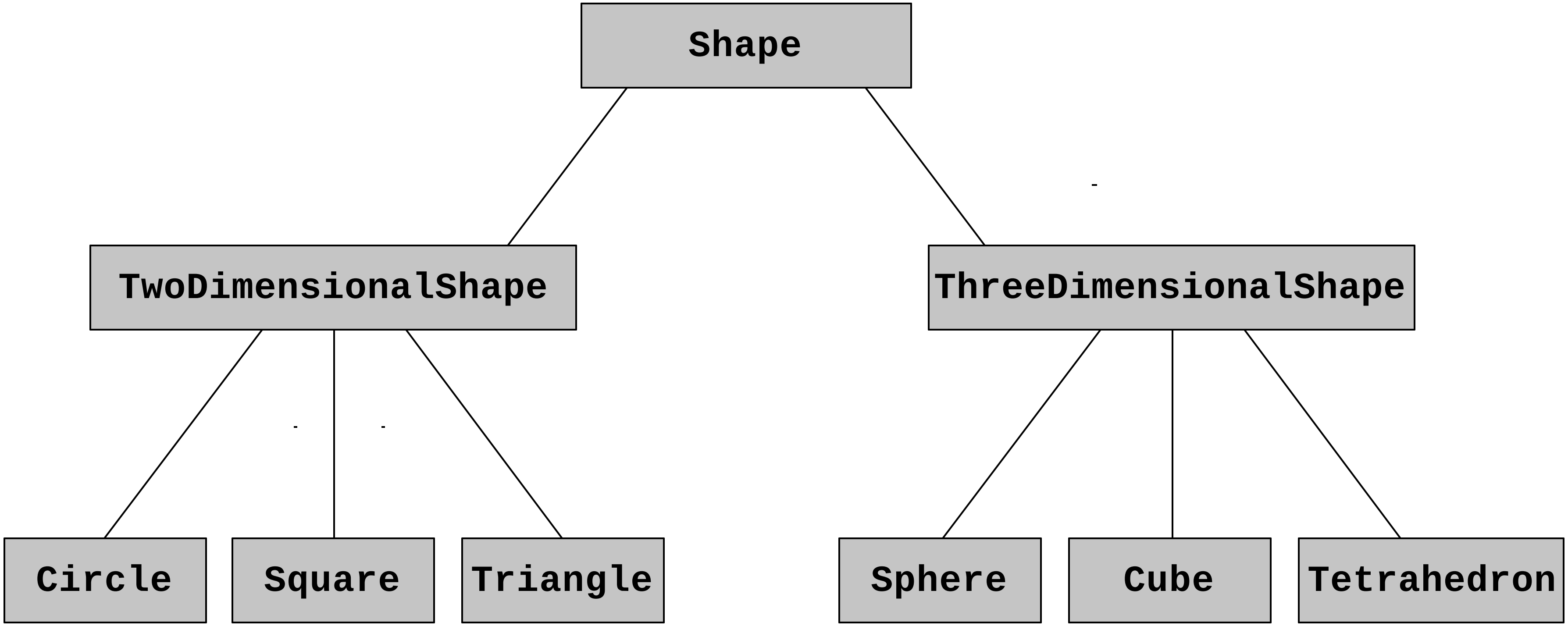
Inheritance hierarchy

Fig. 9.2 Inheritance hierarchy for university **CommunityMember**s.



Inheritance hierarchy

Fig. 9.3 Inheritance hierarchy for Shapes.



Base Classes and Derived Classes

- **`public`** inheritance
  - Specify with:

    `Class TwoDimensionalShape : public Shape`
    - Class **`TwoDimensionalShape`** inherits from class **`Shape`**
  - Base class **`private`** members
    - Not accessible directly
    - Still inherited - manipulate through inherited member functions
  - Base class **`public`** and **`protected`** members
    - Inherited with original member access
  - **`friend`** functions
    - Not inherited

`protected` Members

- **`protected`** access
  - Intermediate level of protection between **`public`** and **`private`**
  - **`protected`** members accessible to
    - Base class members
    - Base class **`friend`**s
    - Derived class members
    - Derived class **`friend`**s
  - Derived-class members
    - Refer to **`public`** and **`protected`** members of base class
      - Simply use member names

Relationship between Base Classes and Derived Classes

- Base class and derived class relationship
  - Example: Point/circle inheritance hierarchy
    - Point
      - x-y coordinate pair
    - Circle
      - x-y coordinate pair
      - Radius

Relationship between Base Classes and Derived Classes

- Using `protected` data members
  - Advantages
    - Derived classes can modify values directly
    - Slight increase in performance
      - Avoid set/get function call overhead
  - Disadvantages
    - No validity checking
      - Derived class can assign illegal value
    - Implementation dependent
      - Derived class member functions more likely dependent on base class implementation
      - Base class implementation changes may result in derived class modifications
        - Fragile (brittle) software

Case Study: Three-Level Inheritance Hierarchy

- Three level point/circle/cylinder hierarchy
  - Point
    - x-y coordinate pair
  - Circle
    - x-y coordinate pair
    - Radius
  - Cylinder
    - x-y coordinate pair
    - Radius
    - Height

Constructors and Destructors in Derived Classes

- Instantiating derived-class object
  - Chain of constructor calls
    - Derived-class constructor invokes base class constructor
      - Implicitly or explicitly
    - Base of inheritance hierarchy
      - Last constructor called in chain
      - First constructor body to finish executing
      - Example: **`Point3`**/**`Circle4`**/**`Cylinder`** hierarchy
        - **`Point3`** constructor called last
        - **`Point3`** constructor body finishes execution first
    - Initializing data members
      - Each base-class constructor initializes data members inherited by derived class

Constructors and Destructors in Derived Classes

- Destroying derived-class object
  - Chain of destructor calls
    - Reverse order of constructor chain
    - Destructor of derived-class called first
    - Destructor of next base class up hierarchy next
      - Continue up hierarchy until final base reached
        - After final base-class destructor, object removed from memory
- Base-class constructors, destructors, assignment operators
  - Not inherited by derived classes
  - Derived class constructors, assignment operators can call
    - Constructors
    - Assignment operators

public, protected and private Inheritance

Base class member access specifier	Type of inheritance		
	public inheritance	**protected** inheritance	**private** inheritance
Public	**public** in derived class. Can be accessed directly by any non-**static** member functions, **friend** functions and non-member functions.	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**private** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.
Protected	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**protected** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.	**private** in derived class. Can be accessed directly by all non-**static** member functions and **friend** functions.
Private	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.	Hidden in derived class. Can be accessed by non-**static** member functions and **friend** functions through **public** or **protected** member functions of the base class.

Software Engineering with Inheritance

- Customizing existing software
  - Inherit from existing classes
    - Include additional members
    - Redefine base-class members
    - No direct access to base class's source code
      - Link to object code
  - Independent software vendors (ISVs)
    - Develop proprietary code for sale/license
      - Available in object-code format
    - Users derive new classes
      - Without accessing ISV proprietary source code