

**605.404 Object-Oriented Programming with C++**  
**Summer 2007**  
**Lectures 08**

8	Classes and Operator Overloading	S: Chaps 10-11
---	----------------------------------	----------------

Additional Resources Related to This Lecture:

OMG Unified Modeling Language Specification, v2.0  
( <http://www.uml.org/#UML2.0> )

OMG Unified Modeling Language Specification, v1.5  
([http://www.omg.org/technology/documents/modeling\\_spec\\_catalog.htm#UML](http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML) )

Coplien, J. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992 (reprinted with corrections April 1992).

Lieberher, K. and Holland, I. "Assuring Good Style for Object-oriented Programs", IEEE Software September 1989, pp 38-48.

Lippman, Stanley B. *A C++ Primer*. Reading, Mass: Addison-Wesley, 1989.

Pohl, Ira. *Object-Oriented Programming Using C++*. Reading, Mass: Benjamin/Cummings, 1993.

Class is a design concept (defines new types)

Object is a run-time concept (specific instances of a type)

#### Chapter 10 Classes

- I. 10.1 Introduction
  - A. a type is a concrete representation of a concept
  - B. "a program that provides types that closely match the concepts of the application tends to be easier to understand and easier to modify than a program that does not"
  - C. enables compiler to detect illegal uses of objects that would otherwise remain undetected until the program is thoroughly tested
  - D. fundamental idea in defining a new type is to separate
    - 1. the incidental details of the implementation of the type
    - 2. properties essential to the correct use of the type
  - E. Chapter 10 focuses on relatively simple concrete user-defined types that logically don't differ much from built-in types
- II. 10.2 Classes
  - A. a class is a user-defined type
  - B. Booch (*Object-Oriented Analysis and Design with Application, Second Edition*) says the *object model* has four major elements (i.e., a model without any one of these elements is not object-oriented)
    - 1. abstraction: denotes the essential characteristics of an object that distinguishes it from all other kinds of objects and thus provides crisply

- defined conceptual boundaries, relative to the perspective viewer
- a) entity: an object represents a useful model of a problem-domain or solution domain entity
  - b) action: an object that provides a generalized set of operations, all of which perform some kind of function
  - c) virtual machine: an object that groups together operations that are all used by some superior level of control, or operations that all use some junior-level set of operations
  - d) coincidental: an object that packages a set of operations that have no relation to each other
2. encapsulation: the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation
  3. modularity: the property of a system that has been decomposed into a set of cohesive and loosely coupled modules
  4. hierarchy: a ranking or ordering of abstractions
- C. and three minor elements (i.e., these elements are useful but not essential parts of the object model)
1. typing: the enforcement of the class of an object, such that objects of different types may not be interchanged, or at most, they may be interchanged only in restricted ways
  2. concurrency: the property that distinguishes an active object from one that is not active
  3. persistence: the property of an object through which its existence transcends time (i.e., the object continues to exist after its creator ceases to exist) and/or space (i.e., the object's location moves from the address space in which it was created)

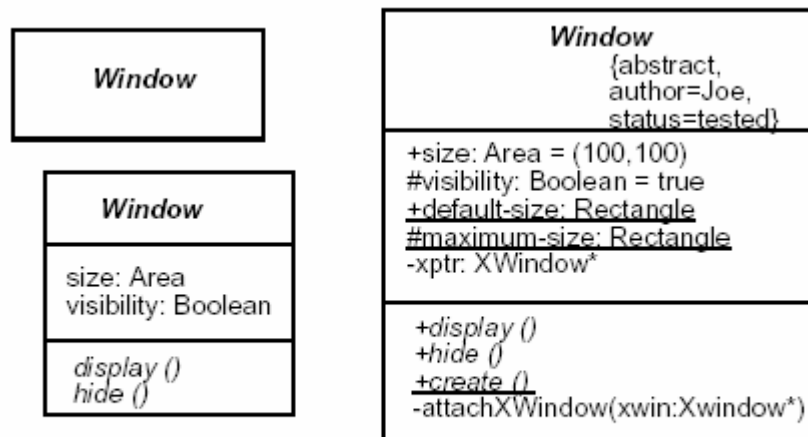


Figure 3-20 Class Notation: Details Suppressed, Analysis-level Details, Implementation-level Details

- D. Notation from UML v1.4 spec section 3.22.5
- E. 10.2.1 Member Functions
  1. 10.2.1 Member Functions
    - a) *my\_birthday* is an instance of class (struct) *Date*

- b) in function *f()*, *today* is an instance of class (struct) *Date*
- c) note that declaration of class (struct) has scope between { }
- d) need to use scope resolution operator in definition of functions (when not *inline* because different classes (structs) may contain similarly signed functions)
- e) in a member function, member names can be used without explicit reference to an object
  - (1) *today.init(16,10,1996)* has all the member names in *Date::init(int dd, int mm, int yy)* referring to the object that called the *init* function
  - (2) *this* will be presented in section 10.2.7 Self-Reference
- f) see 224a.cpp demonstration
- g) class definition
  - (1) also called a *class declaration*
  - (2) defines a new type
  - (3) can be replicated in different source files using *#include* without violating the one-definition rule (see section 9.2.3 in the text)

## 2. 10.2.2 Access Control

- a) *struct* does not specify that the functions it contains should be the only ones to
  - (1) depend directly on the *struct's* representation
  - (2) the only ones that directly access objects of the type
- b) *class* has these restrictions
- c) a *struct* is simply a *class* whose members are public by default
- d) see 225a.cpp demonstration (produces compilation errors as expected)
- e) advantages
  - (1) errors in data values for member variables limited to member functions
  - (2) user code depends only on the public interface and need not be rewritten (though may need recompilation) if representation of a class is changed
  - (3) potential users need only examine the definition of public (and protected) member functions in order to learn to use a class

f) importance of having the *init()* function

## 3. 10.2.3 Constructors

- a) programmer might forget to call *init()* or call it more than once accidentally
- b) *constructor* is a function with explicit purpose of initializing objects
- c) should provide several ways of initializing an object (must obey overloading rules as other functions)
- d) importance of using default arguments to reduce the number of related functions (overloaded functions)
  - (1) example bottom page 227 assumes that *today* an object of type *Date* has been initialized already!
  - (2) Initialization of *today* would use system calls to time (see section D.4.4)

4. 10.2.4 Static Members
  - a) *Date* class dependent on global variable *today* in examples thus far
  - b) Need a variable that is part of a class, yet not part of any object of that class, is called a *static* member of the class
    - (1) Called class variables in Smalltalk
  - c) There is exactly one copy of each static member for the entire class
  - d) A function that needs access to members of a class, yet doesn't need to be invoked by a particular object is a static member function
  - e) See 228a.cpp demonstration
5. 10.2.5 Copying Class Objects
  - a) a class object can be initialized with a copy of an object of its class; this can be done even where constructors have been declared
  - b) default behavior is that the copy is a copy of each member
  - c) you can override default behavior by defining a copy constructor (urged to do this by Coplien)
  - d) can copy by assignment (memberwise copy also)
  - e) see 229a.cpp demonstration
6. 10.2.6 Constant Member Functions
  - a) need methods for examining member values and not changing them
  - b) a *const* member function can be invoked for both *const* and non-*const* objects
  - c) non-*const* member functions can be invoked only for *const* objects
7. 10.2.7 Self-Reference
  - a) *\*this* refers to the object for which a member function is invoked
    - (1) in nonstatic member functions it is a pointer to the object for which the function was invoked
    - (2) in a non-*const* member function of class *X*, the type of *this* is *X\**
      - (a) you can take the address of *this* or assign to *this*
    - (3) in a *const* member function of class *X* the type of *this* is *const X\** to prevent modification of the object itself
  - b) see example 230a.cpp (still needs more work on date correctness issues)
  - c) 10.2.7.1 Physical and Logical Constness
    - (1) what if a member is logically *const* but still needs to change the value of a member
    - (2) to the user, the function appears not to change the state of its object; some detail that the user cannot directly observe is updated (*logical constness*)
    - (3) see 232a.cpp demonstration
      - (a) made up *compute\_cache\_value()* method contents for the time being
  - d) 10.2.7.2 Mutable
    - (1) "casting away *const*" has implementation-dependent behavior that can be avoided (use *mutable*)
    - (2) *mutable* is a storage specifier (like *register*) that specifies that a member should be stored in a way that allows updating – even when it is a member of a *const* object
    - (3) *mutable* means "can never be *const*"
    - (4) see 233a.cpp demonstration

- (5) see 233b.cpp for demonstration of indirect access to object that can change
- 8. 10.2.8 Structures and Classes
  - (1) Stroustrup prefers *struct* for classes that have all public data; just “data structures”
  - (2) Even with *struct* constructors and access functions are useful
- 9. 10.2.9 In-Class Function Definitions
  - a) a member function defined within the class definition – rather than simply declared there – is taken to be an inline member function
  - b) remember that elements must be defined before use so it is important in what order definitions appear in the class
- F. 10.3 Efficient User-Defined Types
  - 1. page 237 – the set of operations for a fairly typical user-defined type
    - a) constructor
    - b) accessor
    - c) manipulators
    - d) copy
    - e) exception handling and reporting
  - 2. (requires operator overloading in this example section 10.3.3)
  - 3. 10.3.1 Member Functions
    - a) initialization may be complicated because it establishes the invariant for the class
  - 4. 10.3.2 Helper Functions
    - a) functions associated with the class that need not be defined in the class itself because they don’t need direct access to the representation
    - b) use of .h file or namespaces to enclose the helper functions
  - 5. 10.3.3 Overloaded Operators
    - a) details in Chapter 11; examples on page 241 for declarations
  - 6. 10.3.4 The Significance of Concrete Classes
    - a) concrete types are user-defined types different than built-in types
    - b) they are not abstract classes or class hierarchies
    - c) (also called value types and their use is value-oriented programming)
    - d) intent of a concrete type is to do a single, relatively small thing well and efficiently
    - e) building a new type based on concrete types illustrated page 242
- G. 10.4 Objects
  - 1. 10.4.1 Destructors
    - a) some constructors allocate resources that must be released after use
    - b) destructors clean up and release resources
    - c) called automatically when an automatic variable goes out of scope, or when an object on the free store is deleted
    - d) most common use is to release memory acquired in a constructor
    - e) see example 243a.cpp
    - f) usually used for variably-sized objects in C++
  - 2. 10.4.2 Default Constructors
    - a) most types can be considered to have a default constructor
    - b) called without supplying an argument
      - (1) in the example page 245 the constructor has a default

argument value of 15, so the destructor must be  
Table::~Table(size\_t)

- c) a compiler generated default constructor implicitly calls the default constructor for a class' members of class type and bases
  - d) because *const* and references must be initialized (see section 5.5 and 5.4) a class containing *const* or reference members cannot be default-constructed unless the programmer explicitly supplies a constructor (see 10.4.6.1)
  - e) default constructors can be invoked explicitly
  - f) built-in types have default constructors (see section 6.2.8)
3. 10.4.3 Construction and Destruction
- a) list is important page 244
4. 10.4.4 Local Variables
- a) constructor for local variable executed each time the thread of control passes through the declaration of the local variable
  - b) destructor for local variable executed each time the local variable's block is exited (in reverse order of construction\_
- c) 10.4.4.1 Copying Objects
- (1) Lippman Notes:
    - (a) "there is one instance in which the constructors provided by the class designer are not invoked to initialize a newly defined class object – when a class object is initialized with another object of its class"
    - (b) String vowel ("a");
    - (c) String article = vowel;
    - (d) The initialization of *article* is accomplished by copying in turn each member of *vowel* into the corresponding member of *article*
    - (e) This is called memberwise initialization
    - (f) The compiler accomplishes memberwise initialization by internally defining a special constructor `X::X(const X& );`
    - (g) The initialization of a class object with another object of its class occurs in three program situations:
      - (i) The explicit initialization of one object with another
      - (ii) The passing of a class object as an argument to a function
      - (iii) The return of a class object as the return value of a function
  - (2) Memberwise copy is usually the wrong semantics for copying objects containing resources managed by constructor/destructor pairs
  - (3) Defining the copy constructor page 246 and copy assignment operator
    - (a) The copy constructor initializes uninitialized memory
    - (b) The copy assignment operator must correctly deal with a self-constructed object

5. 10.4.5 Free Store
  - a) an object created on the free store has its constructor invoked by the new operator and exists until the delete operator is applied to a pointer to it
  - b) example of memory leak page 246-247
  - c) after delete has been applied to an object it is an error to access that object in any way (implementations cannot reliably detect such errors)

6. 10.4.6 Class Objects as Members

- a) specification of arguments for the member's constructor are in a member initializer list
- b) Borland C++ Programmer's Guide (Class Initialization)
  - (1) an object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list
  - (2) If a class has a constructor its objects must be either initialized or have a default constructor (latter used for objects not explicitly initialized)
  - (3) Objects of classes with constructors can be initialized with an expression list in parentheses. This list is used as an argument list to the constructor (as in *X two(1)* or *X three=1*)
  - (4) The constructor can assign values to its members in two ways
    - (a) It can accept values as parameters and make assignments to the member variables within the function body of the constructor

```
class X
{
    int a,b;
public:
    X(int i, int j) {a = i; b = j}
};
```

- (b)
- (c) An initializer list can be used prior to the function body; provides a method for passing values along to base class constructors

```

class X
{
    int a, b, &c; //Note the reference variable
public:
    X(int i, int j) : a(i), b(j), c(a) { }
};

```

(d)

(e) The initializer list is the only place to initialize a reference variable

c) ARM section 12.6.2 Initializing Bases and Members

(1) Initializers for immediate base classes and for members not inherited from a base class may be specified in the definition of a constructor. This is most useful for class objects, constants, and references where the semantics of initialization and assignment differ

(2) The argument list is used to initialize the named nonstatic member of base class object. This is the only way to initialize nonstatic *const* and reference members

d) Members' constructors are called before the body of the containing class' own constructor is executed. The constructors are called in the order in which the members are declared in the class

e) When a class object containing class objects is destroyed, the body of that object's own destructor is executed first and then the members' destructors are executed in reverse order of declaration

f) 10.4.6.1 Necessary Member Initialization

(1) essential for types for which initialization differs from assignment (member objects of classes without default constructors, *const* members, and reference members)

g) 10.4.6.2 Member Constants

(1) possible to initialize a static integral constant member by adding a *const-expression* initializer to its member declaration

(2) can be done – should use *enum*

h) 10.4.6.3 Copying Members

(1) a default copy constructor or default copy assignment copies all elements of a class

(2) if this copy cannot be done, it is an error to try to copy an object of such a class

7. 10.4.7 Arrays

a) if an object of a class can be constructed without supplying an explicit initializer, then arrays of that class can be defined

b) except by using an initializer list there is no way to specify explicit arguments for a constructor in an array declaration

c) use *vector* if you can

8. 10.4.8 Local Static Store

9. 10.4.9 Nonlocal Store
    - a) a variable defined outside any function is initialized before *main()* is invoked and its destructor is invoked after exit from *main()*
    - b) cautions on invocation and libraries
  10. 10.4.10 Temporary Objects
    - a) most often result of arithmetic operations
  11. 10.4.11 Placement of Objects
    - a) what if you want to allocate the object somewhere other than free store?
    - b) Can place objects anywhere by providing an allocator function with extra arguments and then supplying those extra arguments when using *new*
  12. 10.4.12 Unions
    - a) (best used for low level code from this text perspective)
- III. *Advanced C++ Programming Styles and Idioms*, Coplien
- A. 2.1 Classes
    1. Unlike C, C++ introduces a new type into a program for each tagged structure or class; it is as though a typedef were inserted after the declaration. In C++, the tag and type are synonymous, whereas in C they are separate names
    2. Analogy between C structs and C++ classes. C++ classes offer two things beyond C structs: typing and abstraction
  - B. 2.2 Object Inversion
    1. In C++ view the functions as belonging to the structure because of their close coupling to the structure and its data
    2. (some) C++ member functions do not dereference a structure pointer, and do not show any explicit access to the class object that would correspond to the struct in C
      - a) C++ creates another level of scope for classes and functions can be thought of as living inside instances of structures
      - b) implicit parameter **this**
      - c) C and C++ Code for Stack Implementation
  - C. 2.3 Constructors and Destructors
    1. C++ variables should always be automatically initialized (built-in types are not initialized, though to be consistent with C)
    2. classes should control their own memory allocation
    3. constructors control class instance initializations
    4. Stack example with Constructors and Destructors
  - D. 2.11 Program Organization Conventions
    1. Class declarations are kept in a header file; declarations refer to those things that are of interest to a user of a class, not just to the person implementing the class
  - E. class implementations kept mostly in .c files

```

/* Copyright (c) 1992 by      */
/* AT&T Bell Laboratories.    */
/* Advanced C++ Programming   */
/* Styles and Idioms          */
/* James O. Coplien           */
/* All rights reserved.       */

```

```
#define STACK_SIZE 10
```

```

struct Stack {
    long items[STACK_SIZE];
    int sp;
};

```

```
void Stack_initialize(s)
```

```

struct Stack *s;
{
    s->sp = -1;
}

```

```
long Stack_top(s)
```

```

struct Stack *s;
{
    return s->items[s->sp];
}

```

```
long Stack_pop(s)
```

```

struct Stack *s;
{
    return s->items[s->sp--];
}

```

```
void Stack_push(s, i)
```

```

struct Stack *s; long i;
{
    s->items[++s->sp] = i;
}

```

```
int main()
```

```

{
    struct Stack q;
    int i;
    Stack_initialize(&q);
    Stack_push(&q,1);
    i = Stack_top(&q);
    Stack_pop(&q);
}

```

```

/* Copyright (c) 1992 by      */
/* AT&T Bell Laboratories.    */
/* Advanced C++ Programming   */
/* Styles and Idioms          */
/* James O. Coplien           */
/* All rights reserved.       */

```

```
const int STACK_SIZE = 10;
```

```

class Stack {
private:
    long items[STACK_SIZE];
    int sp;
public:
    void initialize();
    long top() const;
    long pop();
    void push(long);
};

```

```

void Stack::initialize() {
    sp = -1;
}

```

```

long Stack::top() const {
    return items[sp];
}

```

```

long Stack::pop() {
    return items[sp--];
}

```

```

void Stack::push(long i) {
    items[++sp] = i;
}

```

```
int main()
```

```

{
    Stack q;
    q.initialize();
    q.push(1);
    int i = q.top();
    q.pop();
}

```

```

/* Copyright (c) 1992 by AT&T Bell Laboratories. */
/* Advanced C++ Programming Styles and Idioms, */
/* James O. Coplien All rights reserved. */

const int STACK_SIZE = 10;
class Stack {
public:
    Stack();
    Stack(int);
    ~Stack();
    long top() const;
    long pop();
    void push(long);
private:
    long *items;
    int sp;
};
Stack::Stack() {
    items = new long[STACK_SIZE];
    sp = -1;
}
Stack::Stack(int size) {
    items = new long[size]; // like a typed sbrk or malloc call,
                           // except constructor is called
                           // if present

    sp = -1;
}
Stack::~~Stack() {
    delete[] items; // like free, except destructor
                  // is called
}
long Stack::top() const {
    return items[sp];
}
long Stack::pop() {
    return items[sp--];
}
void Stack::push(long i) {
    items[++sp] = i;
}
int main()
{
    Stack q; // call Stack::Stack()
    Stack r(15); // call Stack::Stack(int)
    q.push(1);
    long i = q.top();
    q.pop();
}

```

- G. Chapter 3 Concrete Data Types (introduction)
  - 1. concrete data types follow a specific form using class members to augment the C++ compiler's type system so that compiler can generate efficient and safe code for arbitrarily complex abstractions
  - 2. orthodox canonical class form
    - a) canonical meaning it gives a framework of rules for the compiler to follow in code generation
    - b) orthodox meaning that it is the form most directly understood and supported by the language itself
- H. 3.1 The Orthodox Canonical Class Form
  - 1. "When to use this idiom: In general, you *must* use the orthodox canonical form if:
    - a) you want to support assignment of objects of the class, or want to pass those objects as call-by-value parameters to a function, *and*
    - b) the object contains pointers to objects that are reference counted, or the class destructor performs a **delete** on a data member of the object.
  - 2. You *should* use the orthodox canonical form for any nontrivial class in a program, for the sake of uniformity across classes and to manage the increasing complexity of each class over the course of program evolution."  
p. 44
  - 3. Coplien's example of a canonical class definition

```

/* Copyright (c) 1992 by AT&T Bell Laboratories.*/
/* Advanced C++ Programming Styles and Idioms   */
/* James O. Coplien                             */
/* All rights reserved.                         */

class String {
public:
    // the public user interface to a String:

    // redefine "+" to mean catenation, two cases:
    friend String operator+(const char*, const String&);
    String operator+(const String&) const;
    int length() const;           // length of string in characters
    // . . . .                   // other interesting operations

    // boilerplate member functions:

    String();                     // default constructor
    String(const String&);        // constructor to initialize a new
                                // string from an existing one
    String& operator=(const String&); // assignment
    ~String();                   // destructor

    /*
     * These operators are typical of the kinds of customized
     * behaviors a user can define for a type. These are
     * examples suitable for a String class.
     */

    String(const char *);        // initialize from a "C string"
private:
    char *rep;                  // implementation data and
                                // internal functions
                                // (here, represent internals
                                // as a good old C string)
};

```

#### 4.

### IV. Class Initialization

- A. "An object of a class with only public members and no constructors or base classes (typically a structure) can be initialized with an initializer list. If a class has a constructor, its objects must be either initialized or have a default constructor. The latter is used for objects not explicitly initialized." [Borland C++ 4.0 Programmer's Guide pg 145]
- B. "The constructor can assign values to its members in two ways:
  1. It can accept the values as parameters and make assignments to the member variables within the function body of the constructor.
  2. An initializer list can be used prior to the function body:

```

class X
{
    int a,b,&c;
public:
    X(int i, int j) : a(i) b(j), c(a) {}
};

```

- a)
    - b) The initializer list is the only place to initialize a reference variable." [Borland C++ 4.0 Programmer's Guide pp 144-145]
- C. ARM section 12.6.2 pp 290 ff Initializing Bases and Members
  - 1. "Initializers for immediate base classes and for members not inherited from a base class may be specified in the definition of a constructor. This is most useful for class objects, constants, and references where the semantics of initialization and assignment differ."
- V. ARM section 9.3.1 The *this* pointer pp 176-177
  - A. "The type of *this* in a member function of class X is X \*const unless the member function is declared const or volatile; in those cases the type of *this* is const X \*const and volatile X \*const, respectively."
 

```

          struct s {
              int a;
              int f() const { return a; }
              int g() { return a++ ; }
              int h() const { return a++;
          } //error
          
```
  - B.
- VI. The Law Of Demeter (Based on "Assuring Good Style for Object-oriented Programs")
  - A. The Demeter system provides a high-level interface to class-based, object-oriented systems. "One of Demeter's primary goals is to develop an environment that eases the evolution of a class hierarchy."
  - B. Focus on style rules that restrict how methods are written for a set of class definitions
  - C. Definitions used in the Law of Demeter
    - 1. Client. Method *M* is a client of method *f* attached to class *C* if inside *M* message *f* is sent to an object of class *C* or to *C*. The exception is that if *f* is specialized in one or more subclasses then *M* is only a client of *f* attached to the highest class in the subclass hierarchy. Method *M* is a client of class *C* if *M* is a client of some method attached to class *C*.
    - 2. Supplier. If method *M* is a client of class *C* as just described then *C* is a supplier of *M*. (a supplier class of method *M* is a class whose methods are called in *M*)
    - 3. Acquaintance class. Class *C*<sub>1</sub> is an acquaintance class of method *M* attached to class *C*<sub>2</sub> if *C*<sub>1</sub> is a supplier to *M* and *C*<sub>1</sub> is not
      - a) an argument class of *M*, including *C*<sub>2</sub>, nor
      - b) an instance variable class of *C*<sub>2</sub>, nor
      - c) a superclass of the above classes.
      - d) (an acquaintance class of method *M* is a supplier class that is not an argument class of *M* nor an instance-variable class of the class to which *M* is attached)
    - 4. Preferred-acquaintance class. A preferred acquaintance class of method *M* is either a class of objects created directly in *M* or a class of global variable used in *M*.
    - 5. Preferred-supplier class. Class *B* is a preferred supplier of method *M* (attached to class *C*) if *B* is a supplier of *M* and one of the following

conditions holds:

- a) *B* is an instance-variable class of *C* or a superclass of such a class,
- b) *B* is an argument class of *M*, including *C* or a superclass of such a class, or
- c) *B* is the preferred acquaintance class of *M*
- d) (the preferred supplier classes are made up of method's preferred acquaintance classes and its instance-variable and argument classes.

D. Law of Demeter, Class Form, Minimization Version

1. *Minimize the number of acquaintance classes over all methods.*
2. suggests checking at compile time or design time based on the following information for each method:
  - a) types of each of the arguments and the result
  - b) the acquaintance class
3. would be nice to extend C++ to declare acquaintance classes

E. The Law of Demeter, Class Form, Strict Version

1. *All methods may have only preferred-supplier classes.*
2. helps control complexity because preferred suppliers are usually a small subset of all classes in an application

F. The Law of Demeter, Object Form

1. *All methods may have only preferred-supplier objects*
2. this cannot be enforced at compile time
3. a method's preferred objects are
  - a) the immediate parts of the pseudovisible itself
  - b) the method's argument objects, or
  - c) the objects that are either objects created directly in the method or objects in global variables

G. Law of Demeter, Class Form, Strict Version customized for C++

1. In all member functions *M* of class *C*, you may use only members (function and data) of the following classes and their base classes:
  - a) *C*
  - b) data-member classes of *C*,
  - c) argument classes of *M*,
  - d) classes whose constructor functions are called in *M*, or
  - e) the classes of global variables used in *M*.

## Chapter 11 Overloading

### VII. 11.1 Introduction

- A. defining operators for types other than built-in types allows use of convenient notation in other parts of the program
- B. *b+d* means *b.operator+(d)* in the example on page 262
- C. usual precedence of operators continues to hold (see pages 120-121)

### VIII. 11.2 Operator Functions

- A. list page 262 of which operators can be declared
- B. the following cannot be overloaded (scope resolution operator, member selection, member selection through pointer to member) and (*sizeof* and *typeid*)
- C. 11.2.1 Binary and Unary operators
  1. **binary operator** either a nonstatic member function taking one argument or a nonmember function taking two arguments

- a) operator << example
        - (1) aa<<bb can be interpreted as either
          - (a) aa.operator<<(bb)
          - (b) operator<<(aa,bb)
        - b) if both are defined, overload resolution determines which if any interpretation is used
    - 2. **unary operator**, whether prefix or postfix, can be defined by either a nonstatic member function taking no arguments or a nonmember function taking one argument
      - a) for any prefix operator (<< example), <<aa can be interpreted as
        - (1) aa.operator<<() or operator<<(aa)
        - (2) if both are defined, overload resolution determines which if any interpretation is used
      - b) for any postfix operator (<< example), aa<< can be interpreted as
        - (1) aa.operator<<(int) or operator<<(aa,int) [SEE SECTION 11.11 later]
      - c) if both are defined, overload resolution determines which if any interpretation is used
    - 3. an operator can be declared only for the syntax defined for it in the grammar (e.g., cannot force a unary % or a ternary +)
  - D. 11.2.2 Predefined Meaning for Operators
    - 1. operator=, operator[], operator(), and operator-> must be nonstatic member functions (ensures first operands are lvalues)
    - 2. recognize that flexibility of equivalent combinations of other operators for built-in operators is not automatic for user-defined operators
    - 3. operators = & , have predefined meanings for class objects that can be made private
  - E. 11.2.3 Operators and User-Defined Types
    - 1. an operator function must either be a member or take at least one argument of a user-defined type (except when redefining *new* and *delete*)
    - 2. operators cannot be defined to operate exclusively on pointers
    - 3. an operator function intended to accept a built-in type as its first operand cannot be a member function
    - 4. operations can be defined for enumerations (because they are user defined types)
  - F. 11.2.4 Operators in Namespaces
    - 1. an operator is either a member of a class or defined in some namespace
    - 2. rules for resolution page 266
    - 3. note that in operator lookup no preference is given to members over non-members (differs from lookup of named functions)
- IX. 11.3 A Complex Number Type
  - A. basic structure
  - B. 11.3.1 Member and Nonmember Operators
    - 1. operators that inherently modify the value of their first argument are defined in the class itself
    - 2. operators that produce a new value based on the values of its arguments are defined outside the class and use the essential operators in their implementation
  - C. 11.3.2 Mixed-mode Arithmetic

- D. 11.3.3 Initialization
- E. 11.3.4 Copying
  - 1. explicit copy constructor page 271
- F. 11.3.5 Constructors and Conversions
  - 1. no implicit user-defined conversions are applied to the left-hand side of a . or a -> this is the case even when the . is implicit
- G. 11.3.6 Literals
  - 1. (think of constructor invocations with literal arguments as literals)
- H. 11.3.7 Additional Member Functions
- I. 11.3.8 Helper Functions
- X. 11.4 Conversion Operators
  - A. using a constructor to specify type conversion is convenient but has implications that can be undesirable. A constructor cannot specify
    - 1. an implicit conversion from a user-defined type to a built-in type (because the built-in types are not classes)
    - 2. a conversion from a new class to a previously defined class (without modifying the declaration for the old class)
  - B. use a conversion operator
    - 1.  $X::operator T()$ , where  $T$  is a type name, defines a conversion from  $X$  to  $T$
    - 2. a conversion operator resembles a constructor
  - C. best to use named conversion functions to start as in  $X::make\_int()$
  - D. 11.4.1 Ambiguities
    - 1. in some cases the desired type can be constructed by repeated use of constructors or conversion operators. This must be handled by explicit conversion – only one level of user-defined implicit conversion is legal
- XI. 11.5 Friends
  - A. an ordinary member function declaration specifies three logically distinct things
    - 1. the function can access the private part of the class declaration and
    - 2. the function is in the scope of the class and
    - 3. the function must be invoked on an object (has a *this* pointer)
  - B. *static* member functions have only the first two properties
  - C. *friend* functions have only the first property
  - D. note order of declaration on page 278 example (*Matrix* declared before first use in *friend* function operator\*)
  - E. *friend* can be in either the private or public part of a class declaration
  - F. *friend* function is explicitly declared in the declaration of the class of which it is a friend; it is as much a part of the interface as is a member function
  - G. a member function of one class can be a friend of another
  - H. all functions of one class can be friends of another (see page 279 example)
  - I. 11.5.1 Finding Friends
    - 1. a friend declaration does not introduce a name into an enclosing scope
    - 2. a friend class must be previously declared in an enclosing scope or defined in the non-class scope immediately enclosing the class that is declaring it a friend; scopes outside the innermost enclosing namespace scope are not considered
    - 3. a friend function should be explicitly declared in an enclosing scope or take an argument of its class derived from that
  - J. 11.5.2 Friends and Members
    - 1. try to minimize the number of functions that access the representation of a

- class and try to make the set of access functions as appropriate as possible
  - 2. an operation modifying the state of a class object should therefore be a member or a global function taking a non-const reference argument (or a non-const pointer argument)
  - 3. operators that require lvalue operands for fundamental types are most naturally defined as members for user-defined types
  - 4. if implicit type conversion is desired for all operands of an operation, the function implementing it must be a nonmember function taking a const reference argument or a non-reference argument
    - a) often the case for functions implementing operators that do not require lvalue operands when applied to fundamental types
    - b) consequently binary operators are the most common source of friend functions
  - 5. if no type conversions are defined, there appears no compelling reason to choose a member over a friend taking a reference argument or vice versa
  - 6. all else equal choose a member
- XII. 11.6 Large Objects
  - A. reduce copying through use of reference objects
  - B. (pointers cannot be used because it is not possible to redefine the meaning of an operator applied to a pointer)
  - C. may need to buffer results to avoid copying results
- XIII. 11.7 Essential Operators
  - A. copy constructor  $X(const X\&)$
  - B. remember that assignment and initialization are different operations
  - C. see page 284
    - 1. remember that objects are copied when passed as function argument, as function return value, and as exception parameters
    - 2. copy operations are not inherited (see section 12.2.3)
  - D. 11.7.1 Explicit Constructor
    - 1. by default, a single argument constructor also defines an implicit conversion
      - a) can be suppressed by declaring a constructor *explicit* (will not implicitly be invoked)
    - 2. use of subrange types (Date with Year type for example page 285) (wrapper around *int*)
- XIV. 11.8 Subscripting
  - A. operator[] function can be used to give new subscript meaning to class objects
  - B. operator[]() must be a member function
- XV. 11.9 Function Call
  - A. *expression(expression-list)* can be thought of as a binary operator with the *expression* as the left-hand operand and *expression-list* as the right-hand operand
  - B. the call operator () can be overloaded in the same way as other operators can
    - 1. an argument list for an *operator()* () is evaluated and checked according to the usual argument passing rules
  - C. call operator also known as the application operator
  - D. important for objects that behave like functions (*function-like object* or *function object*)
  - E. operator() () must be a member function

XVI. 11.10 Dereferencing

- A. overloading -> used to create “smart pointers” (objects that act like pointers and in addition perform some action whenever an object is accessed through them)
- B. remember that user-defined operators do not have the built-in flexible behaviors of ordinary operators (see page 290); programmer needs to provide equivalence if needed
- C. overloading -> is important for indirection and delegation
- D. operator -> must be a member function; its return type must be a pointer or an object of a class to which you can apply ->

XVII. 11.11 Increment and Decrement

- A. how to tell which version is prefix and which is postfix (see examples page 292)  
(version without the dummy argument is the prefix operator)

XVIII. 11.12 A String Class

- A. (see text for examples)