OOP 3200 – Object Oriented Programming II

Week 3 – C++ Class Operators and Conversions

Week 3 Overview

- Class Operators and Conversions
- In-class Exercise 2
- Lab 2
- Quiz 1

Course Outline

Week	Date	Topic	Evaluation	Weight
1	Sep 09, 2020	- Course Orientation - Object-Oriented Programming overview - Partnering for labs		
2	Sep 16, 2020	REVIEW OF CLASSES & OBJECTS in C++ - Encapsulation - Object Attributes and Behaviours - Classes: The Blueprint for Objects - Relationship Between Class and Objects - Static Class Members - Friend Functions	In-Class Exercises 1: (2%) C++ Assignment 1: (6%)	8
3	Sep 23, 2020	CLASS OPERATORS AND DATA TYPE CONVERSIONS in C++: - Creating Class Operators - How Methods Are Shared - Data Type Conversions	In-Class Exercises 2: (2%) C++ Assignment 2: (6%) C++ Quiz 1: (5%)	13
4	Sep 30, 2020	INHERITANCE AND POLYMORPHISM in C++: - Class Inheritance - Polymorphism - Virtual functions - Interfaces / Abstract Classes	In-Class Exercises 3: (2%) C++ Assignment 3: (6%)	8
5	Oct 07, 2020	COLLECTIONS in C++: - Dynamic Object Creation and Deletion - Pointers As Class Members / Destructors - Copy Constructors / Copy Assignment Operators	In-Class Exercises 4: (2%) C++ Assignment 4: (6%)	8
6	Oct 14, 2020	GENERICS in C++ - Method templates - Class Templates	In-Class Exercises 5: (2%)	2
7	Oct 21, 2020	THE C++ STANDARD TEMPLATE LIBRARY (STL): - Vectors and Linked Lists - Stacks and Queues - Maps and Sets	In-Class Exercises 6: (2%) C++ Assignment 5: (6%) C++ Quiz 2: (5%)	13
READING WEEK				

Objectives

- In this lesson, you will learn about:
  - Creating Class Operators
  - How Methods Are Shared
  - Data Type Conversions
  - Two Useful Alternatives: `operator()` and `operator[]`
  - Common Programming Errors
  - Abstraction and Encapsulation
  - Code Extensibility

Creating Class Operators

- C++ operators listed in the table can be redefined for class use

Operator	Description
( )	Function call (see Section 11.4)
[ ]	Array element (see Section 11.4)
->	Structure member pointer reference
new	Dynamic allocation of memory
delete	Dynamic deallocation of memory
++	Increment
--	Decrement
-	Unary minus
!	Logical negation
~	Ones complement
*	Indirection
*	Multiplication
/	Division
%	Modulus (remainder)
+	Addition
-	Subtraction
<<	Left shift
>>	Right shift
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to
&&	Logical AND
||	Logical OR
&	Bit-by-bit AND
^	Bit-by-bit exclusive OR
|	Bit-by-bit inclusive OR
= += -= *= /= %= &= ^= |= <<= >>=	Assignment
,	Comma

Creating Class Operators (cont'd.)

- Providing class with operators requires determination of:
  - Operations that make sense for class
  - How operations should be defined
- Example (`Date` class): define a small meaningful set of operators
  - Addition of two dates is not meaningful
  - Addition of an integer to a date or subtraction of an integer from a date is meaningful
    - Integer defined as number of days to/from a date

Creating Class Operators (cont'd.)

- Defining more `Date` class operators
  - Subtraction of two dates is meaningful
    - Number of days between two dates
  - Comparison of two dates is meaningful
    - Does one date occur before another?

- **Operator functions**
  - Operations on class objects that use C++'s built-in operator symbols

Creating Class Operators (cont'd.)

- Declaring and implementing operator functions: same as for all C++ member functions except:
  - Function's name connects **appropriate operator symbol** to operator defined by the function
  - Format: `operator <symbol>`

    where symbol is an operator listed in the earlier table

- Examples:

  `operator+` is the name of addition function

  `operator==` is the name of comparison function

Creating Class Operators (cont'd.)

- Writing operator function: function accepts desired inputs and produces correct return value

- Creating comparison function for `Date` class: compare two dates for equality
  - Select C++'s equality operator

  - Function name is `operator==`
    - Accepts two `Date` objects
    - Compares objects
    - Returns a Boolean value indicating result
      - `True` for equality; `False` for inequality

Creating Class Operators (cont'd.)

- Prototype of comparison function

```
bool operator==(const Date&);
```

  - Indicates that function is named `operator==`
  - Function returns Boolean value
  - Accepts a reference to `Date` object
  - Second `Date` object calls the function

Creating Class Operators (cont'd.)

- Writing comparison function:

```
bool Date::operator==(const Date& date2)
{
  if(day == date2.day && month == date2.month && year == date2.year)
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

  - `const` keyword ensures that the dereferenced `Date` object cannot be changed in the function

Creating Class Operators (cont'd.)

- Calling operator function: same syntax as calling C++ built-in types

- Example: if `a` and `b` are objects of type `Date`
  - the expression `if (a == b)` is valid
  - `if (a.operator == b)` is also valid

- The Program on the following slide is complete including:
  - **Declaration** and **definition** of operator function
  - `if` statement with comparison operator function

Program 11.1

```
#include <iostream>
using namespace std;

// class declaration section
class Date
{
  private:
    int month;
    int day;
    int year;
  public:
    Date(int = 7, int = 4, int = 2012);  // constructor
    bool operator==(Date&);  // prototype for the operator== function
};

// class implementation section
Date::Date(int mm, int dd, int yyyy)
{
  month = mm;
  day = dd;
  year = yyyy;
}

bool Date::operator==(Date& date2)
{
  if(day == date2.day && month == date2.month && year == date2.year)
    return true;
  else
    return false;
}

int main()
{
  Date a(4,1,2012), b(12,18,2010), c(4,1,2012); // declare 3 objects
```

Creating Class Operators (cont'd.)

```
    if (a == b)
      cout << "Dates a and b are the same." << endl;
    else
      cout << "Dates a and b are not the same." << endl;
    if (a == c)
      cout << "Dates a and c are the same." << endl;
    else
      cout << "Dates a and c are not the same." << endl;
  return 0;
}
```

Creating Class Operators (cont'd.)

- Creating an addition function for `Date` class
  - Select C++'s addition operator
  - Function name is `operator+`
    - Accepts `Date` object and an integer object
    - Adds integer to `Date` to return a `Date`
  - Suitable prototype: `Date operator+(int)`
  - Data conversion for `Date` class required
    - Resulting day returned must be in range 1 – 30
    - Resulting month returned must be in range 1 – 12

Creating Class Operators (cont'd.)

- Suitable **addition** function implementation

```
Date Date::operator+(int days)
{
  Date temp;               // a temporary Date to store the result
  temp.day = day + days;   // add the days
  temp.month = month;
  temp.year = year;
  while (temp.day > 30)    // now adjust the months
  {
    temp.month++;
    temp.day -= 30;
  }
  while (temp.month > 12) // adjust the years
  {
    temp.year++;
    temp.month -= 12;
  }
  return temp;        // the values in temp are returned
}
```

Creating Class Operators (cont'd.)

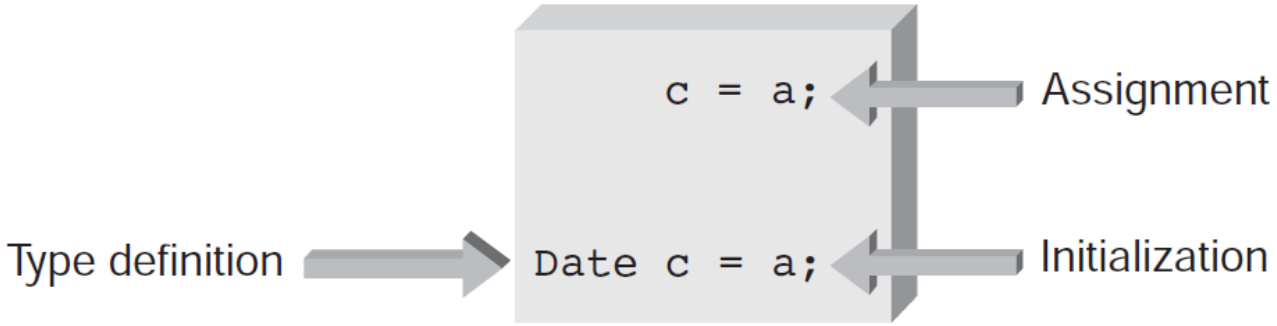
- Restrictions on redefining C++ operators for class use
  - An operator's syntax can't be changed
  - Symbols not in the table shown above cannot be redefined
    - Examples: `.`, `::`, and `?:`
  - New operator symbols cannot be created
  - Neither the precedence nor the associativity of C++'s operators can be modified
  - Operators cannot be redefined for built-in types
  - Operator must be member of a class or be defined to take at least one class member as an operand

Assignment Operator

- **Assignment operator**, =
  - One operator that works with all classes without requiring an operator function
  - **Memberwise assignment** (`a = b;`)
- C++ compiler builds a memberwise assignment operator as the default assignment operator for each class
- Simple assignment operator declaration form:
  - `void operator=(const ClassName&);`
  - Won't work with `a = b = c;`

Copy Constructors

- Assignments are entirely different operations than initializations



- **Copy constructor**
  - Initializes an object by using another object of the same class
  - If you don't declare one, the compiler creates it for you
  - Works just fine unless the class contains pointer data members
  - Format: *ClassName*(const *ClassName*&);

Base/Member Initialization

- True initialization has no reliance on assignment
  - Possible in C++ by using a **base/member initialization list**

- Option 1:
  - *ClassName*(*argument list*):*list of data members*(*initializing values*) {}

- Option 2:
  - Declare a function prototype with defaults in the class declaration section followed by the initialization list in the class implementation section

Operator Functions as Friends

- Functions in Programs 11.1 and 11.2 constructed as class members
- All operator functions (except for =, `()`, `[]`, `->`) may be written as friend functions
- Example: `operator+()` function from Program 11.2
  - If written as friend, suitable prototype would be:

```
friend Date operator+ (Date&, int);
```

  - Friend version contains reference to `Date` object that is not contained in member version
- Table 11.2 shows equivalence of functions and arguments required

Table 11.2 Operator Function Argument Requirements

Operator	Member Function	Friend Function
Unary	1 implicit	1 explicit
Binary	1 implicit and 1 explicit	2 explicit

Operator Functions as Friends (cont'd.)

- Program 11.2's `operator+()` function, written as a friend

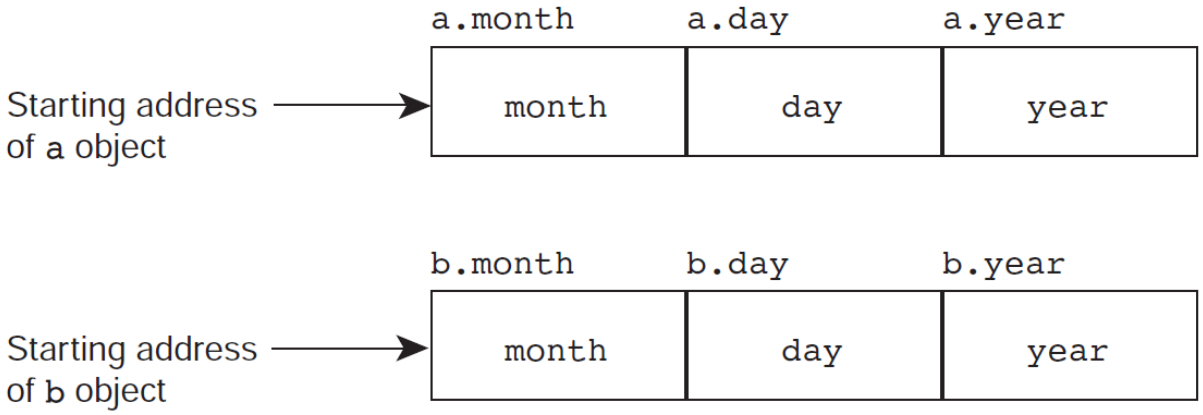
```
Date operator+(Date& op1, int days)
{
  Date temp; // a temporary Date to store the result
  temp.day = op1.day + days; // add the days
  temp.month = op1.month;
  temp.year = op1.year;
  while (temp.day > 30) // now adjust the months
  {
    temp.month++;
    temp.day -= 30;
  }
  while (temp.month > 12) // adjust the years
  {
    temp.year++;
    temp.month -= 12;
  }
  return temp; // the values in temp are returned
}
```

Operator Functions as Friends (cont'd.)

- Difference between the member and friend versions of `operator+()` function: explicit use of additional `Date` parameter (`op1`)

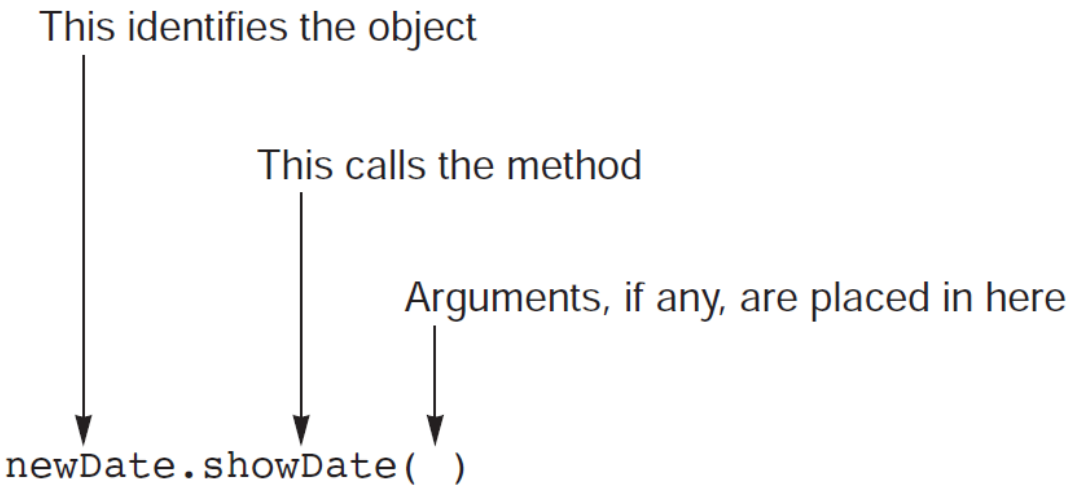
- Convention for choosing between friend or member implementation:
  - Member function: better for binary functions that modify neither operand (such as ==, +, -)
  - `Friend` function: better for binary functions that modify one of their operands

How Methods Are Shared



- Each time an object is created
  - Distinct area of memory is set aside for its **data members**
  - Replication of data storage isn't implemented for member methods

How Methods Are Shared (continued)



- Sharing member methods requires providing a means of identifying which specific object a member method should be operating on
  - Accomplished by the name of the object preceding the method call
  - Method gets direct access to the object used in calling it

The `this` Pointer

- Added automatically to each nonstatic class method as a hidden argument
- When a method is called, the calling object's address is passed to it
  - Stored in the method's `this` pointer
- Address in `this` pointer can be used explicitly in the called method
  - `(*this).month`

`(*this).month` accesses `oldDate`'s `month` member.
`(*this).day` accesses `oldDate`'s `day` member.
`(*this).year` accesses `oldDate`'s `year` member.

The Assignment Operator Revisited

- Drawback of simple assignment operator function: it returns no value
  - Multiple assignments (`a = b = c`) not possible
- Can use operator function that has return type together with `this` pointer
- Prototype and implementation of modified function illustrated on next slide
  - Function in Program 11.4 allows multiple assignment

The Assignment Operator Revisited (cont'd.)

- Prototype for modified assignment operator

```
Date operator=(const Date&);
```

- Implementation of prototype

```
Date Date::operator=(const Date &newdate)
{
  day = newdate.day;      // assign the day
  month = newdate.month; // assign the month
  year = newdate.year;   // assign the year

  return *this;
}
```

Objects as Arguments

Table 11.3 Examples of Object Arguments

	Passing an Object	Passing a Reference	Passing an Address
Method call	`newDate.swap(oldDate)`	`newDate.swap(oldDate)`	`newDate.swap(&oldDate)`
Method prototype	`void swap(Date)`	`void swap(Date&)`	`void swap(Date *)`
Method header	`void swap(Date temp)`	`void swap(Date& temp)`	`void swap(Date *temp)`
Comments	A copy of `oldDate` is passed; `temp` is an object, and `newDate` is passed to the `this` pointer.	The address of `oldDate` is passed; `temp` is a reference, and `newDate` is passed to the `this` pointer.	The address of `oldDate` is passed; `temp` is a pointer, and `newDate` is passed to the `this` pointer.

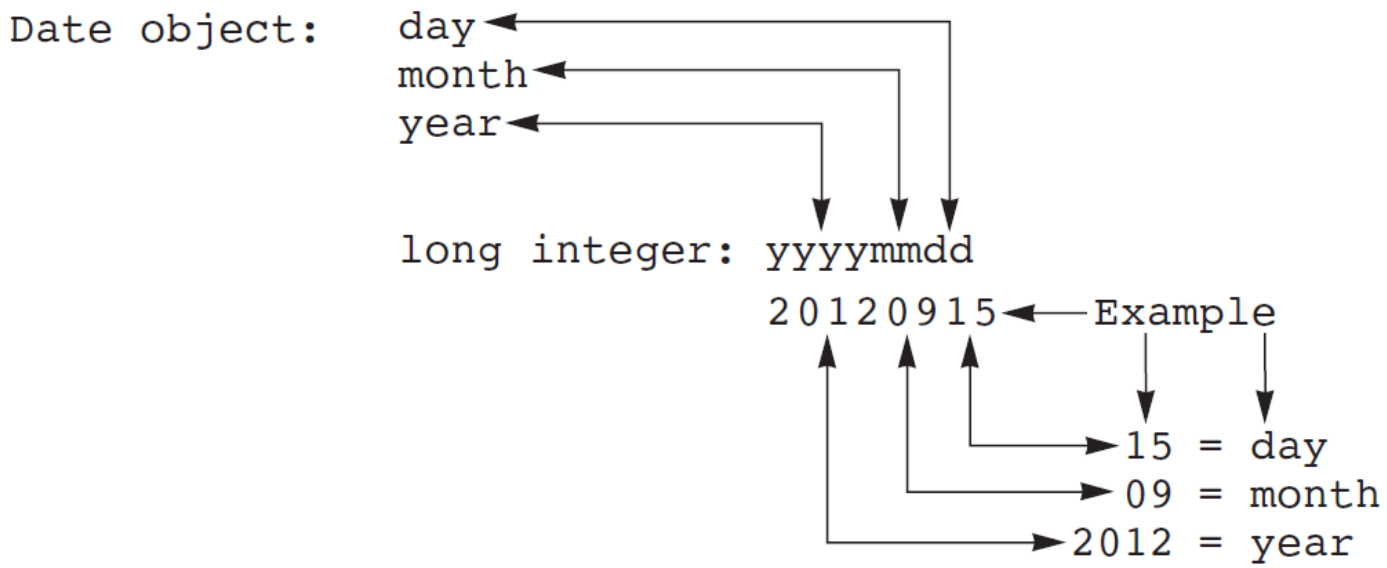
DURHAM COLLEGE
SUCCESS MATTERS

Notation

- In using a reference or pointer
  - Pay attention to using the passed address correctly
- For pointers, the notation is the same as that used for the `this` pointer
- One of the main reasons for including references in C++ is for their use as function arguments
  - In some applications, pointers must be used

Data Type Conversions

- Introduction of classes expands possibilities to following cases
  - Conversion from class type to built-in type
  - Conversion from built-in type to class type
  - Conversion from class type to class type



Built-in to Built-in Conversion

- Conversion from built-in to built-in:
  - Implicit conversion: occurs in context of a C++ operation
    - Example: When double precision number is assigned to integer variable, only integer portion stored
  - Explicit conversion: occurs when cast is used
    - Format `(dataType) expression`

Class to Built-in Conversion

- Conversion from class to built-in done by using:
  - **Conversion operator function**: member operator function that has name of built-in data type or class
  - When operator function has built-in data type name, it is used to convert from class to built-in data type

```
class Date
{
  private:
    int month, day, year;
  public:
    Date(int = 7, int = 4, int = 2012);    // constructor
    operator long();     // conversion operator prototype
    void showDate();
};
```

```
// conversion operator definition for converting a Date to a long int
Date::operator long()   // must return a long, as its name implies
{
  long yyyymmdd;
  yyyymmdd = year * 10000 + month * 100 + day;
  return(yyyymmdd);
}
```

Built-in to Class Conversion

- Conversion from built-in to class done by using constructor functions
  - **Type conversion constructor**: constructor whose first argument is not a member of its class and whose remaining arguments, if any, have **default values**
  - If the first argument is built-in data type, the constructor can be used to cast built-in data type to class object

```
// constructor for converting from long to Date
Date::Date(long findate)
{
  year = int(findate/10000.0);
  month = int((findate - year * 10000.0)/100.0);
  day = int(findate - year * 10000.0 - month * 100.0);
}
```

Class to Class Conversion

- Conversion from class to class: done using **member conversion operator** function
  - Same as cast from class to built-in data type
  - In this case, operator function uses class name being converted to rather than built-in data name
  - Demonstrated in Program 11.8 (over the next few slides)

Class to Class Conversion (continued)

Program 11.8

```
#include <iostream>
#include <iomanip>
using namespace std;

// forward declaration of class Intdate
class Intdate;

// class declaration section for Date
class Date
{
  private:
    int month, day, year;
  public:
    Date(int = 7, int = 4, int = 2012);  // constructor
    operator Intdate();  // conversion operator from Date to Intdate
    void showDate();
};

// class declaration section for Intdate
class Intdate
```

Class to Class Conversion (continued)

```
{
  private:
    long yyyymmdd;
  public:
    Intdate(long = 0);     // constructor
    operator Date();  // conversion operator from Intdate to Date
    void showint();
};

// class implementation section for Date
Date::Date(int mm, int dd, int yyyy)  // constructor
{
  month = mm;
  day = dd;
  year = yyyy;
}

// conversion operator function converting from Date to Intdate class
Date::operator Intdate()   // must return an Intdate object
{
  long temp;
  temp = year * 10000 + month * 100 + day;
  return(Intdate(temp));
}
```

Class to Class Conversion (continued)

```
// member function to display a Date
void Date::showDate()
{
  cout << setfill('0')
       << setw(2) << month << '/'
       << setw(2) << day << '/'
       << setw(2) << year % 100;
  return;
}

// class implementation section for Intdate
Intdate::Intdate(long ymd)  // constructor
{
  yyyymmdd = ymd;
}

// conversion operator function converting from Intdate to Date class
Intdate::operator Date()   // must return a Date object
```

Class to Class Conversion (continued)

```
{
  int mo, da, yr;
  yr = int(yyyymmdd/10000.0);
  mo = int((yyyymmdd - yr * 10000.0)/100.0);
  da = int(yyyymmdd - yr * 10000.0 - mo * 100.0);
  return(Date(mo,da,yr));
}

// member function to display an Intdate
void Intdate::showint()
{
  cout << yyyymmdd;
  return;
}

int main()
{
  Date a(4,1,2011), b;       // declare two Date objects
  Intdate c(20121215L), d;  // declare two Intdate objects
  b = Date(c);       // cast c into a Date object
  d = Intdate(a);    // cast a into an Intdate object
```

Class to Class Conversion (continued)

```
    cout << " a's date is ";
    a.showDate();
    cout << "\n   as an Intdate object this date is ";
    d.showint();

    cout << "\n c's date is ";
    c.showint();
    cout << "\n   as a Date object this date is ";
    b.showDate();
    cout << endl;

    return 0;
}
```

Two Useful Alternatives: `operator()` and `operator[ ]`

- Sometimes operations are needed that have more than two arguments
  - Limit of two arguments on **binary operator functions**
  - Example: `Date` objects contain three integer data members: `month`, `day`, and `year`
    - Might want to add an integer to any of these three members (instead of just the day member as done in Program 11.2)

Two Useful Alternatives: `operator()` and `operator[ ]` (cont'd.)

- `operator[ ]` (subscript operator): used where single non-object argument is required
- `operator()` (parentheses operator): no limit on number of arguments that can be passed to it
  - Addresses problem of passing three integers to `Date` addition operator function
- Parentheses and subscript operators must be defined as member (not friend) functions

Two Useful Alternatives: `operator()` and `operator[ ]` (cont'd.)

- Declaration

```
class Date
{
  private:
    int month;
    int day;
    int year;
  public:
    Date(int = 7, int = 4, int = 2012);    // constructor
    Date operator[](int);   // function prototype
    void showDate();        // member function to display a Date
};
```

Two Useful Alternatives: `operator()` and `operator[ ]` (cont'd.)

❖ Implementation

```
Date Date::operator[](int days)
{
  Date temp;    // a temporary Date to store the result

  temp.day = day + days;    // add the days
  temp.month = month;
  temp.year = year;
  while (temp.day > 30)     // now adjust the months
  {
    temp.month++;
    temp.day -= 30;
  }
  while (temp.month > 12)   // adjust the years
  {
    temp.year++;
    temp.month -= 12;
  }
  return temp;    // the values in temp are returned
}
```

Two Useful Alternatives: `operator()` and `operator[ ]` (cont'd.)

❖ Usage

```
int main()
{
  Date oldDate(7,4,2011), newDate; // declare two objects

  cout << "The initial Date is ";
  oldDate.showDate();

  newDate = oldDate[284]; // add in 284 days = 9 months and 14 days

  cout << "\nThe new Date is ";
  newDate.showDate();
  cout << endl;

  return 0;
}
```

Common Programming Errors

- Attempting to redefine an operator's meaning as it applies to C++'s built-in data types
- Redefining an overloaded operator to perform a function not indicated by its conventional meaning
  - Will work but is bad programming practice
- Using a user-defined assignment operator in a multiple assignment expression when the operator hasn't been defined to return an object

Common Programming Errors (cont'd.)

- Attempting to make conversion operator function a friend, rather than a member function
- Attempting to specify return type for a conversion operator function
- Forgetting that `this` is a pointer that must be declared using `*this` or `this->`

Summary

- User-defined operators can be constructed for classes by using operator functions
- User-defined operators may be called as a conventional function with arguments or as an operator expression
- Operator functions can also be written as friend functions
- Three categories of data type conversions:
  - Built-in types to class types
  - Class types to built-in types
  - Class types to class types

Summary (cont'd.)

- Type conversion constructor: constructor whose first argument is not a member of its class and whose remaining arguments have default values
- Conversion operator function is member operator function that has the name of built-in data type or class
- An object can be used as a method's argument
- The address of an object can also be passed as an argument

Summary (cont'd.)

- For each class object, a separate set of memory locations is reserved for all data members, except those declared as static
- For each class, only one copy of the member methods is retained in memory, and each object uses the same function
- Subscript operator function, `operator[]`, permits a maximum of one non-class argument
- Parentheses operator function, `operator()`, has no limits on number of arguments

Lesson Supplement: Insides and Outsides

- Concept of encapsulation is central to objects
- In programming terms, an object's attributes are described by data
- Useful visualization is comparing an object with a boiled egg

Lesson Supplement: Insides and Outsides (cont'd.)

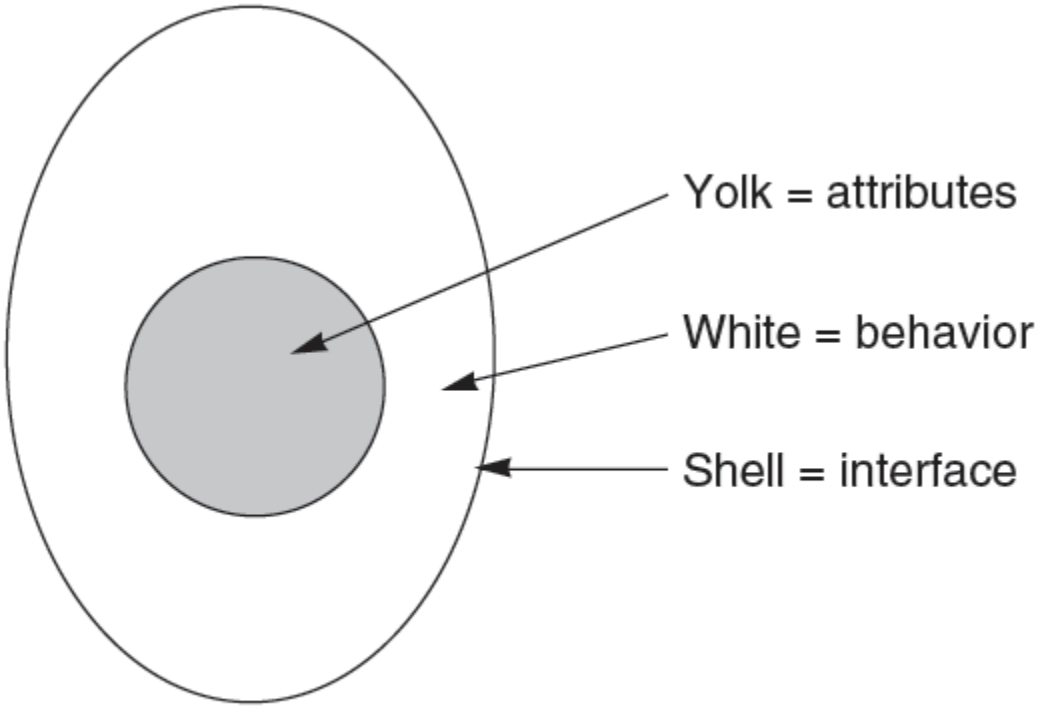


Figure 11.6 The boiled egg object model

Abstraction and Encapsulation

- **Abstraction** means concentrating on what an object is and does before making any decisions about how to implement the object

- **Encapsulation** generally means separating the implementation details of the abstract attributes and behavior and hiding them from the object's outside users

Code Extensibility

- An advantage of the inside-outside object approach
  - It encourages extending existing code without needing to completely rewrite it
- All interactions between objects are centered on the outside interface
  - All implementation details are hidden in the object's inside
- As long as the interface to existing operations isn't changed, new operations can be added as needed