Week 7 Lecture Notes

Custom classes

These material in these notes can also be found in Chapters 10.2-10.3 and 11 in the CISP 1010‘s textbook, Problem Solving with C++ by Savitch, and in this course’s textbook, Data Structures and Algorithms in C++ by Goodrich et al, Chapters 1.5.1-1.5.3, and 2.1.

Know definitions for all of the **bold, underlined** words that follow in these lecture notes.

We’ll be programming in C++ from now on so create \*.cpp files. We will also use an Integrated Development Enviroment (IDE). There’s a Getting Started guide with a download link in the Getting Started module in our online Course Content.

Contents

[Instance and Class Variables 1](#_Toc45626818)

[Accessor and Mutator Member Functions 3](#_Toc45626819)

[Constructors 8](#_Toc45626820)

[Destructor 12](#_Toc45626821)

[Overloading Operators and Friend Functions 12](#_Toc45626822)

[Arrays of Objects 17](#_Toc45626823)

# Instance and Class Variables

Structures are out and classes are in. Well, structures are still used, but with C++ came many new concepts, the first of which is the **class**. A class has some similarities to a structure. It is a type like a structure is a type, and we can declare variables of those types which are called **objects**. Right now, think of a class as a fancy structure that can have private data (not accessible from other files/classes) and public functions (accessible in other classes/files) inside it. Below is a declaration of Date class.

class Date {

private:

int month;

int day;

int year;

};

First, notice the keyword class. We use this keyword instead of struct, and we don’t have to use typedef anymore when creating a new type for a structure or a class in C++. We just created a new type (a **class**). We don’t have any variables yet (**objects**). The classes are the types, the variables are the objects. Also notice the keyword private. If I had created a C structure of this type, there would be nothing to stop me from setting month to -1 and day to -12 which isn’t a date that makes any sense. Using classes, we can make these variables private so that other functions like main can’t do something like:

// declaring an object variable is also called

// **instantiating** an object

Date d;

// illegal in code outside of the class because

// month is private

d.month = -5;

The keyword private is a **visibility modifier**. A visibility modifier determines access to a variable or a function in code outside of a class where that variable or function is declared. Private access in the Date class means that no code other than code inside the Date class (or friends of Date – covered below) may access this data/function. The keyword public is also a visibility modifier granting access to all code outside of the class. There is one more visibility modifier, protected, which is useful when a class inherits from another class, but we will not cover inheritance in this course.

The variables declared in the Date class, month, day, and year, are called **instance variables**. Instance variables are variables declared inside a class where each object of that class has its own copy of those variables. For example, if I declare two Date objects:

Date myBirthday;

Date yourBirthday;

then myBitrhday.month is a different variable than yourBirthday.month. That makes sense – our birthdays may not be in the same month so the two variables myBirthday and yourBirthday need to have their own copies of month, day, and year. Basically, these variables belong to each separate instance of class Date. Hence, the term instance variable.

Not all variables that we could declare in a class need to be instance variables (one copy per declared/instantiated object). Suppose we want a variable inside our Date class called MIN\_MONTH and we’re going to set it to 0 (January = 0). If I have ten Date objects, how many copies of the value MIN\_MONTH do I need? Well, not ten. I just need one no matter how many Date objects I instantiate. I actually might want this value to exist even if I don’t have any Date objects at all. So, I’d like this variable to belong to the class, not any particular Date object. I want a **class variable** (a variable declared inside a class that does not belong to any particular object, and there will only be one copy of it not matter how many objects of this class are instantiated).

In the example code of date.h below, I’ve added a constant class variable (not all class variables have to be constant) by using the keyword static. I made it public because it’s constant and can’t be altered anyway, and it might be useful outside of this class. All variables declared inside a class with the keyword static are class variables.

class Date {

private:

int month;

int day;

int year;

public:

static const int MIN\_MONTH;

};

We have to initialize our class variable in date.cpp with the statement:

const int Date::MIN\_MONTH = JANUARY;

Where did I get JANUARY? I added an enumerated type to the date.h file whose contents so far is shown below.

|  |
| --- |
| #ifndef DATE\_H  #define DATE\_H  #include <iostream>  using namespace std;  enum month\_t { JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER };  class Date  {  public:  static const int MIN\_MONTH;  static const int MAX\_MONTH;  static const int MIN\_YEAR;  static const int MIN\_DAY;  private:  int month; // 0 = January  int day;  int year;  };  #endif |

We can use public class variables in other files. For example, in main.cpp, I can use the following statement to print out the minimum year for any Date object without instantiating (declaring) any dates at all.

cout << "Minimum year: " << Date::MIN\_YEAR << endl;

Note that if you omit the public or private visibility modifiers in a class, everything will be private. If you omit them in a C++ structure, everything will be public. It is a best practice to write these visibility modifiers explicitly to make your code as clear as possible.

**Pair Programming 7a:** Using the IDE, create a workspace/project called pp7a. Copy the contents of the main.cpp file for pair programming 7 in our online Course Content and paste it into the main.cpp file in the project. Add two new files to the project, date.h and date.cpp. **MAKE SURE  YOU BROWSE TO THE DRIVE YOU WANT TO USE IN THE IDE!** Remember, we’re programming in C++ now so we’ll create “cpp” files.

* Inside date.h, add the definition above of the Date class. Don’t forget #ifndef, etc. We still need those preprocessor directives.
* In date.cpp, add statements to initialize all four class variables, MIN\_MONTH to JANUARY, MAX\_MONTH to DECEMBER, MIN\_YEAR to 1, and MIN\_DAY to 1.
* Read through main.cpp. Compile and execute the program. Uncomment the statement  
  d.month = -5;  
  recompile and execute to see the error this statement causes because month is private. Re-comment this statement (so it won’t interfere with tests below).

# Accessor and Mutator Member Functions

Since the Date class data members month, day, and year are private, how do we access them? Right now, we can’t set them to anything or even print them! We have to write public member functions that allow use to “set” and “get” these values. Sometimes these functions are called set and get member functions and sometimes they are called **mutator** and **accessor** member functions, respectively. A mutator member function is any function that changes/mutates a private data member in a class. An accessor member function is one that allows read access to a private data member (i.e., returns its value so we can print it, use it in a formula to compute something, etc.). It is a best practice in some languages to begin the names of accessor and mutator member functions with get and set, respectively. That’s what I want you to do in this class.

Since we’re using a class and not a structure, and we’re doing some object-oriented programming, we can use another object-oriented concept called **encapsulation**. Encapsulation is the idea that everything we need to have to work with a date is encapsulated in the Date class. In a strict object-oriented language, there would be nothing we might need to do with a Date that would not be encapsulated in the class. In a date class more encompassing than ours we might have additional data members such as the day of the week, the name of the day of the week, the name of the month, the day of the year, and even the epoch (i.e., AD). In C++, not a strictly object-oriented language, most, but not all, of the functions we need to manipulate our C++ Date objects will be member functions. They will be encapsulated in the class. We will even put the function declarations inside the class definition itself. For example, consider the new definition for the Date class below. It has a public section now that has prototypes for two member functions: a setYear member function that will set the Date object’s year data member and a getYear member function that will return the year data member so we can print it, for example.

|  |
| --- |
| class Date {     public:  etc.        void setYear( int );        int getYear() const;     private:        int month;        int day;        int year;  }; |

Note the const keyword at the end of the getYear member function. This indicates that getYear can’t change the data of the object month, day, and year. An accessor member function like getYear shouldn’t be able to change an object’s data, so using the const keyword to make the function constant is a best practice with accessor member function. Here’s some code to exemplify the use of the constant function

cout << myBirthday.getYear() << endl;

In this call to getYear() the **invoking object**, the object that called getYear, myBirthday, should not be allowed to alter its month, day, or year values, and it can’t because the function is constant.

Where do we actually write these functions? We have two choices. Our first choice is to write them in date.cpp as shown below for the setYear member function. An example of a good comment header is also shown.

|  |
| --- |
| /\* setYear: sets year   \* Parameter y year   \* Returns: nothing   \*/  void Date::setYear ( int y )  {    if ( y < MIN\_YEAR ) year = MIN\_YEAR;    else year = y;  } |

Notice several differences from the non-object-oriented programming we’ve been doing. First, we have to state somehow that these two functions are member functions inside the Date class. We do that with the scope resolution operator, **::**

void Date::setYear( int y )

If you forget to use the class name followed by the scope resolution operator, Date::, then you haven’t written the Date class’ setYear member function. You’ve written some other setYear function that isn’t a member of any class. You could get several compilation errors: 1) year is undeclared, and 2) the function doesn’t exist when you try to call it. Don’t try to fix the first error by declaring year inside the function! You want to be accessing the year variable that’s already declared in the Date class, not some local variable that will just go out of scope as soon as the function returns.

Next, notice the use of the variable year. Where did this variable come from? It wasn’t declared as a local variable inside the setYear function. Well, it is the private year variable inside the Date class definition. How does the compiler know to look in the Date class? Because, you’ve specified that the setYear function is a Date class member function by using the scope resolution operator with the function name, Date::setYear. Again, it would be a mistake to redeclare year. In the set function, you don’t want to set a local variable, you want to set the private year variable in the class. Novice programmers often make the logic error of declaring a local variable with the same name as an instance variable inside a member function like setYear.

How can setYear access year if it’s private? Since setYear is a member function of the Date class, it can access the private year variable. Also note that the code in setYear has protected the year data member because it will never be set to something crazy like a negative number.

We haven’t written the getYear function yet. It is a tiny function with just one statement:

return year;

We could write it, also in date.cpp just like setYear. But writing a function for such a small amount of code is inefficient. Function calls add to execution time. It is more efficient to make really short functions like getYear an **inline** function by putting its entire definition in date.h inside the class definition. You may not have thought about it yet, but in object-oriented programming, there are a lot of function calls. We already added two that we didn’t have before. In order to set the year value or get it for printing, for example, we have to call functions. We didn’t have to do that before using the non-object-oriented programming structures that we created and used in C. So, for added correctness (year will never be negative) we have added inefficiency (extra function calls). C++ uses inline functions to reduce this inefficiency. If a function is inline, then the compiler actually puts the code of the function in place of the function call. When we execute, the function won’t actually be called. To make a function inline, put its definition in the class definition in the date.h header file as shown below, and do not put it in the date.cpp file.

|  |
| --- |
| class Date  {     public:        int getYear() const { return year; }  etc.     private:        int month;        int day;        int year;  }; |

inline is also a keyword in C++ that you can use to make a function an inline function. However, with many compilers, the function prototype and definition must be in the same file or the linker won’t find it.

Examples of using setYear and getYear are below. Notice how we use the dot operator to call the member functions just like we use it to access data members of a structure or class.

Date d;

// the object d is the **invoking object**

// the object/variable “calling” the member function

d.setYear( 1999 );

cout << d.getYear() << endl;

d.setYear( -5 );

// prints 1 (MIN\_YEAR) which is what the setYear sets

// the year to when the parameter is less than MIN\_YEAR

cout << d.getYear() << endl;

**Pair Programming 7a:** add the setYear declaration and an inline getYear member function to date.h. Add the setYear definition to date.cpp. Uncomment, compile and execute statements in main to test these functions.

One idea in object-oriented programming is **information hiding**. Information hiding is hiding implementation details from the other programmers using your class. I could write a Date class with public member functions such as setYear, getYear, setMonth, getMonth, etc. But, I don’t have to store the date in three integers, month, day, and year. I could store the date as the number of seconds from midnight, January 1, 1970 GMT (which is what Linux actually does!). Would you, as a programmer using my Date class, really care? No! As long as the public interface to the class, those “set” and “get” functions, don’t change, it doesn’t affect how you write your code that uses that public interface.

In conjunction with information hiding is the idea of an **abstract data type (ADT)**. An ADT is a semantic model – the public interface that defines how the ADT is used. It is not the implementation details. If you are a programmer using my Date class, you don’t need to know how I stored the date. Did I use 3 integers, one for month, day and year? Did I use a three element array for the integers month, day, and year or three separate variables? Did I store the date in an unsigned long that holds the number of days since January 1, 1970, GMT? You don’t know, and you don’t care. You just use the public interface (the public member functions) to create and manipulate Date objects. I could even change the internal implementation for the Date class and you wouldn’t care. As long as I didn’t remove anything from the public interface, your code would still work. We will look at the idea of an ADT in more detail when we look at other data structures such as stacks, queues, and heaps.

A **utility function** is a private function for use inside the class only and not outside the class. A useful utility function for the Date class is shown below. It returns the maximum day of a date with a precondition that the month and year have already been set.

|  |
| --- |
| /\* daysInMonth: returns number of days in month based upon year  \* Pre-condition object month and year have been set  \* Returns maximum valid day given a particular month/year  \*/  int Date::daysInMonth() const {  int max; // max number of days in month  switch( month ) {  case JANUARY: case MARCH: case MAY: case JULY:  case AUGUST: case OCTOBER: case DECEMBER:  max = 31; break;  case APRIL: case JUNE: case SEPTEMBER: case NOVEMBER:  max = 30; break;  case FEBRUARY: // FEB  // leap year?  if ( (year % 4 == 0 && year % 100 != 0) ||  ( year %400 == 0) )  max = 29;  else  max = 28;  }  return max;  } |

It would be nice to be able to print the date in a format we all recognize. The printDate function below is a public function that does just that. We will remove this function from the class when we learn how to overload the << operator below, but, for now, it allows us to print the date easily in a nice format with a statement such as

d1.printDate();

|  |
| --- |
| /\* printDate: prints date in form mm/dd/yyyy  \*/  void Date::printDate() const {  cout << setfill( '0' ) << setw(2) <<  (month+1) << '/' << day << '/' << setw(4) << year;  } |

**Pair Programming 7a:** add code for printDate, setDay, getDay, setMonth, and getMonth. The setDay function should use the daysInMonth utility function to determine the maximum valid day for the particular month and year. Function declarations in date.h are below. Add code in date.h to make the “get” member functions inline. Also write a function called setDate that has three parameters: the month, day and year. It can call setYear, setMonth and setDay. Make sure to always set the day after setting the month and year since the program needs to know these value to determine a valid day. Uncomment the code to test these functions in main.

|  |
| --- |
| class Date {     public:  static const int MIN\_MONTH;  static const int MAX\_MONTH;  static const int MIN\_YEAR;  static const int MIN\_DAY;        void setYear( int );        int getYear() const { return year; }        void setMonth( int );        int getMonth() const { return month; }        void setDay( int );        int getDay() const { return day; }        void setDate( int, int, int );     private:        int daysInMonth() const;  int month;        int day;        int year;  }; |

# Constructors

When we created a Date object, what were its month, day and year values before we set them with those set member functions? Wouldn’t it be nice if we could set them to some initial value when we first create/**instantiate** the Date object? (Instantiate is another object-oriented vocabulary word. It means to create/declare the object/variable). There are special member functions called **constructors** that are called only when an object is created. Some constructor rules:

* The name of a constructor MUST be the same as the class name.
* You can overload the constructor (e.g., write multiple versions of the member function by the same name but different parameter types)
* Constructors are ONLY called when an object is created.
* Constructors don’t have return values. I don’t mean the return type is void. I mean that there IS NO RETURN TYPE.
* Always write a default constructor
* ALL constructors should set ALL instance variables

Here are some sample constructor prototypes added to date.h

|  |
| --- |
| class Date {     public:  etc.  // **default constructor** is the constructor  // with no parameters        Date();        Date( int, int, int ); // month, day, year        Date( int, int ); // month, year parameters        Date( int ); // year parameter  etc.  }; |

How do we write these constructors? Here are some examples of two of them that go in date.cpp. You will need to #include <ctime> for the time\_t and tm types and the localtime and time functions. Notice how we don’t rewrite the code that’s in the setDay, SetMonth, and setYear functions. We just call the function we already wrote. This is another concept of object-oriented programming: don’t write the same code twice. In general, if you find yourself writing the same code again, STOP. Put it in a function. We get more correct code and more maintainable/reusable code. We only have to make a change in one place. We are still, however, giving up some efficiency due to more function calls.

|  |
| --- |
| /\* Date: default constructor sets date to today   \*/  Date::Date()  {     time\_t t = time(NULL);     tm\* tPtr = localtime( &t );     month = tPtr->tm\_mon;     day = tPtr->tm\_mday;     year = tPtr->tm\_year + 1900;  }  /\* 3-arg constructor   \* Parameters:   \*    mm month, 1 - 12   \*    dd day in month   \*    yyyy year   \*/  Date::Date( int mm, int dd, int yyyy ) {     setMonth( mm );  setYear( yyyy );  setDay( dd );  } |

How do we call these constructors in main? Note the calls using pointers to the Date objects in the examples below.

// calls default constructor. Doesn’t use parentheses

// because the compiler thinks “Date d1();”  is a

// function prototype/declaration

Date d1;

d1 = Date( ); // also calls default constructor

Date \*d2Ptr = new Date(); // default constructor

d1.printDate(); // d1 is the **invoking object**

cout << endl;

d2Ptr->printDate(); // note the use of ->

cout << endl;

// don’t forget to deallocate memory when you’re done with

// it! Note: C++ delete operator instead of C’s free

// function

delete d2Ptr;

// calls 3-arg constructor

Date d3( AUGUST, 13, 2014);

// calls 3-arg constructor but date set to 1/1/1

Date\* d4Ptr = new Date( -1, -1, -1 );

d3.printDate();

cout << endl;

d4Ptr->printDate();

cout << endl;

delete d4Ptr;

In the code above, note the use of the arrow dereferencing operator, ->, when using a pointer to an object just as with a pointer to a structure. Don’t use the dot operator. Also note the C++ new and delete operators which are used instead of C’s malloc and free functions, respectively. Do NOT mix and match malloc/free and new/delete. The new operator does something the malloc function does not: it calls a constructor. The delete operator does something the free function does not: it calls a destructor (covered below). So, the C functions and the C++ operators are not the same.

Constructors can have an initialization section such as below which sets the instance variables month, day and year to the values in parentheses. The variables in the initialization section must be listed in the order they were declared in the class definition.

|  |
| --- |
| Date::Date( int mm, int dd, int yy ) **: month(mm), day(dd), year(yy)** {  // then code inside the constructor executes after the  // initialization list  } |

As a general rule, always write a default constructor. If you do not write any constructors, the compiler adds an empty default constructor to the program. If you write some constructor other than the default, then the compiler does not add the default constructor which means your code doesn’t have one and declaring an object such as

Date d1;

won’t work as this calls the default constructor. Also, code in the C++ Standard Templates Library, STL, which we will use later in the semester often calls class default constructors. So, if you want to use anything in the STL in your code, you will have to make sure your classes have a default constructor.

C++11 supports initializing class data members to set default values as shown below. The order of execution is: initialization at variable declaration, the initialization section in the constructor heading and, finally, the code inside the constructor body.

|  |
| --- |
| class Date  {     public:  etc.  private:  **int month = 0; // only in C++11**  **int day = 1;**  **int year = 2000;** |

C++11 also supports a constructor calling another constructor (called constructor delegation) in the initialization section. Below, the default constructor calls the 3-argument constructor. For example:

|  |
| --- |
| Date::Date( int mm, int yyyy ): Date(mm, 1, yyyy)  { } |

Constructors are used for automatic type conversion. Consider the statement:

d1 = d2 + 2000;

It’s actually valid. What happens? The overloaded operator function   
operator+(d2, Date(2000) ) is called and since there’s a Date constructor that takes one int argument, this works. We will cover overloading operators below.

Note: just as with structures, one date object can be assigned another:

d1 = d2;

Each data member from d2 is copied to d1. This can cause problems if a class dynamically allocates memory, though. We will address this later.

A special kind of constructor is called a **copy constructor**. A copy constructor for class A is one that makes a copy of an A object from another A object. So, a copy constructor always has exactly one parameter and that parameter is the same as the class. For example, a Date class copy constructor might look like:

|  |
| --- |
| Date::Date( const Date& d ) {     month = d.month;     day = d.day;     year = d.year;  } |

A class copy constructor will ALWAYS have one parameter that is a constant reference to an object of that very class.

**Pair Programmign 7a:** add code for the 5 Date constructors listed above in date.h and date.cpp. Then uncomment the code in main to test each one of these constructors. Note this code uses some pointers to objects.

# Destructor

There can be many constructor member functions, but there can only be one **destructor** member function. Destructor rules:

* A destructor is a member function that gets called automatically when the object passes out of scope or is deallocated from memory. Do NOT write code yourself to call a destructor.
* The name of the destructor is: ~*nameOfClass* (a tilde followed by the name of the class). A destructor may not have any parameters.
* The classes that need destructors are the ones that have allocated resources that need to be deallocated. Our Date class didn’t allocate any resources. But, if we had written a LinkedList class, which we will do next week, we would have memory to deallocate when our LinkedList went out of scope or we deallocated a pointer to it. Other resources to deallocate might include closing a file or a connection to a database.
* Destructors should all be virtual functions. Virtual functions are beyond the scope of the course, but the keyword virtual is used to ensure that the correct function is called when using inheritance which is also beyond the scope of the course. We still need to write destructors properly, so we will use the virtual keyword.

We will write a destructor for our Date class just to practice writing a destructor. The Date class doesn’t really need an explicitly written destructor since it doesn’t need to release any resources. Just put the following code in Date’s destructor so we can see it is called. Add the destructor declaration n date.h:

|  |
| --- |
| class Date {  public:  etc.        virtual ~Date(); |

Then write the destructor definition in date.cpp

|  |
| --- |
| Date::~Date() {     printDate();     cout << " destroyed\n";  } |

**Pair Programming 7b:** Add another project to the pp7 workspace called pp7b. Copy the code in pp7a’s main.cpp into this project’s main.cpp. Add two new files to the project (DON’T ADD EXISTING FILES) date.h and date.cpp and copy the code from these pp7a files to the pp7b files. Add this destructor to your Date class, recompile and execute. You should see messages from the destructor for every Date object. After you’re done, go ahead and comment out the printing code (se we don’t have to see it anymore).

# Overloading operators and friend functions

What if we would like to be able to add some days to a date object with the + operator (as opposed to writing a function called add in our class) such as:

Date d1( DECEMBER, 25, 1999);

Date d2;

d2 = d1 + 1;

In C++, we can not only overload functions, we can overload operators. Some operators are already overloaded. Take the minus sign, -. With two operands it is subtraction. With one operand, it is negation as shown below.

x = y – z;

x = -y;

To overload an operator in C++, we write a function. If the function operates on only one object (the invoking object), we usually write it as a member function. If the function operates on two objects, we usually write it outside of the class – not a member function. A function that overloads an operator must be named operator*X* where *X* is the operator we want to overload. For example, below is an overloaded + operator for the Date class.

|  |
| --- |
| Date operator+( const Date& d1, const Date& d2 ) {    int month = d1.month, day = d1.day, year = d1.year;    // The code here just adds the days. It doesn’t  // check to see if it’s a valid day. This is fine  // for the purpose of the pair programming.  day += d2.day;    return Date( month, day, year );  } |

Note that this function that is not a member of the class Date is accessing private data month, day, and year. How is that possible? Well, it’s not unless we make this function a friend of the Date class in date.h as shown below. **Friend functions** are functions that can access private data members of a class but are not member functions inside the class.

|  |
| --- |
| class Date {     public:     friend Date operator+( const Date& d1, const Date& d2 );  … etc.  }; |

Note several other features about this function. First, its two parameters are both constant reference variables. Second, it returns a Date object by calling the 3-argument Date constructor to construct an unnamed Date object “on the fly” and returns that unnamed Date object. This unnamed Date object ceases to exist after the function returns.

Let’s look at the function call. When we write:

d3 = d1 + d2;

we are really making the following function call:

d3 = operator+( d1, d2 );

As you can see, the operator+ function is not a member function inside the Date class because there is not invoking object – the function call isn’t d1.operator+(d2) where d1 is the invoking object. Instead, the function has two Date object parameters where the actual parameters are constant reference variables: reference because it’s more efficient to pass a reference than a copy of an entire object which would happen if we passed the parameter by value and constant because the operator+ function shouldn’t be allowed to change these parameters that we have passed by reference. The operator+ function returns a Date object because we set “d3 = “ the return value of the function which is the unnamed Date object created by calling the 3-argument constructor. So, the unnamed Date object is copied to d3.

**Pair Programming 7b:** implement the overloaded + operator as shown above. When adding two dates, add the days only. Keep the month and year of the first parameter for the year of the new date. For example, adding 5/1/2000 + 4/2/2001 = 5/3/2000. Make sure you add a friend statement for each function inside the Date class definition in date.h. Uncomment code in main to test.

Let’s look at another example, overloading the == operator which compares two Date objects. We get to decide what it means for two Date objects to be identical. I’m going to say they are identical if they have the same month, day and year. So, we want to be able to write code such as:

if ( d1 == d2 )

Since the function operator== will operate on two objects, we will make it a non-member function, and the function call will be as if we wrote if ( operator==( d1, d2 ) ) where d1 and d2 are both parameters and there is not invoking object. This means writing a non-member function that has two Date objects as its parameters and returns a bool. The formal parameters should be const Date&. Reference because it’s more efficient and constant because the operator== function shouldn’t be able to change its operands.

|  |
| --- |
| bool operator==( const Date& d1, const Date& d2 ) {     if ( d1.day == d2.day && d1.month == d2.month &&        d1.year == d2.year ) return true;     else return false;  } |

**Pair Programming 7b:** implement operator== (it will need to be a friend of the Date class) and update the main function and test the overloaded == operator.

Suppose we want to overload the unary - (negation) operator so we can write code to negate a Date object such as:

Date d1, d2;

d1 = -d2;

I’m not really sure what negating a date means … maybe it’s like an “unbirthday” when a kid has their birthday around a major holiday so you celebrate it 6 months from the real birthday. But, let’s look at how we would implement it, anyway. Since this function operates on only one object, we’ll make it a member function. The function call has an invoking object such as   
d1 = d2.operator-( );

Note that since we will write this function as a member function of the class, there is an invoking object, d2 in the call above. Since the function is a member of the date class, we put its declaration in date.h as shown below. The function itself is constant because it shouldn’t be able to change the invoking object.

Date operator-( ) const;

We write the function’s definition in date.cpp as shown here.

|  |
| --- |
| Date Date::operator-( ) const {  return Date( abs(this->month-6), this->day, this->year );  } |

Now is a good time to introduce the keyword this. You don’t have to declare this. It exists in any member function (not in a non-member function) with an invoking object. It is a pointer, that points to the invoking object. So, writing

this->day

when the function call was

d1 = -d2;

means, in the execution of operator-(), the this keyword points to d2, the invoking object.

We could have written the overloaded operator+, operator- and operator== as member functions. However, two examples of overloaded operators that cannot be written as member functions are >> (extraction operator) and << (insertion operator) for input and output, respectively. Suppose we would like to write (or read) a Date object using the << operator and an ofstream object such as:

cout << d1 << endl;

That’s really:

operator<<( cout, d1 ) << endl;

Looks like we need a function called operator<< that has two parameters: cout and a Date object. But, what does it have to return? Well, it has to return/evaluate to cout because the next thing to happen needs to be

cout << endl;

operator<< must return its cout object parameter so the next << operator that prints the endl has the same cout parameter. The object cout is an ostream (an ofstream is also an ostream, by the way). This function isn’t a member function of class Date because it is not invoked by a Date object. We need to make it a friend of the Date class, though, so it can access the Date class private data directly, without using the Date class public get methods.

|  |
| --- |
| ostream& operator<<( ostream& outStream, const Date& d ) {    outStream << setfill( '0' ) << setw(2) <<  (d.month+1) << '/' << d.day << '/' << setw(4) << d.year;     return outStream;  } |

The function’s parameters are both reference variables. The ostream must be a reference variable just like ofstream or ifstream. The actual parameter for this ostream formal parameter is altered by the function as data is written to the stream. The Date object is passed by reference for efficiency and as a constant to keep this function from altering the actual parameter since it shouldn’t ever do that. Since the return value is the very ostream parameter, it has to be the same type as that parameter: ostream&. Remember, this function must be declared a friend function in the class definition in date.h so it can access private data d.month, d.day and d.year.

Overloading the extraction operator for input is similar. We want to be able to use the extraction operator such as:

cout << “Enter month, day, and year: “;

cin >> d1;

We need a function whose declaration is:   
istream& operator>>( istream&, Date& ).   
Note that the second parameter, the Date reference is not constant. The functionality of input is to change the values in the actual parameter so the formal parameter can’t be constant.

|  |
| --- |
| istream& operator>>( istream& inStream, Date& d ) {     inStream >> d.month >> d.day >> d.year;  d.setMonth( d.month-1 ); // since user enters 1 for JAN     return inStream;  } |

If the insertion or extraction operands were member functions, what class would they be a member of? The call would be:

cout << d1;

which means it would look like cout.operator<<( d1 ) since the first operand is always the invoking object. So, the invoking object wouldn’t even be a Date object. It would be cout. And we can’t change cout’s class.

**Pair Programming 7b:** add overloaded – (unary negation), >> and << to date.h and date.cpp then uncomment code in main to test these. You will have to add #include <iostream> and using namespace statements in date.h for ostream and istream.

What if we want to overload the assignment operator? We can already us the assignment operator with Date objects just as we can with structures as shown here:

d1 = d2;

Or here:

d1 = d2 = d3;

The assignment operator just copies all data members. However, if we copied one node to another node, we probably would not want to copy the pointers. Or, if we copied one list to another list, we wouldn’t just want to copy the head and tail pointers. If we only copied the data in the list class, that’s what would happen, and we’d end up with two different list variables whose head and tail pointers both point to the same list. This is called a shallow copy. Instead, if we assign one list to another list such as

list2 = list1;

we want list2 to be its own list, a duplicate of list1. And, if we made changes to list2, those changes would not happen to list1. This would be a deep copy. So, in many instances, we do need to overload the assignment operator, =, to get a different copy (in the case f=of a node) or a deep copy (in the case of a list). We will practice writing an overloaded operator= in the Date class even though it’s not necessary.

The assignment operator MUST be a class member function, which means its call for the statement

d1 = d2 = d3;

would look like

d1.operator=( d2.operator=(d3) );

So, the member function has one parameter (e.g., d3 ) and must return a Date& object. Actually, it must return the very Date object that invoked it. In the case of   
d2.operator( d3 ), a reference to d2 should be returned. If you return a non-reference, the function operator= will call itself when it assigns its return value to d2 and there will be infinite recursion. OPERATOR == SHOULD ALWAYS RETURN A REFERENCE. Note the return value of \*this – the thing that this points to which is the invoking object which in the first call is d2.

|  |
| --- |
| Date& Date::operator=( const Date& d1 ) {     this->day = d1.day;     this->month = d1.month;     this->year = d1.year;     return \*this;  } |

**Pair Programming 7b:** update main.cpp, date.h and date.cpp to write/use an overloaded assignment operator.

General rules for overloading operators in C++:

1. Make a member function if there’s only one object operand and a non-member function if there are two operands.
2. The overloaded assignment operator has to be a member of the class
3. The overloaded << and >> operators have to be friends
4. If it’s a non-member function, make it a friend of the class
5. You can’t create a new operator
6. You can’t change operator precedence
7. You can’t overload . (dot operator), ?: (tertiary if/else operator),  .\* (pointer to a data member) or :: (scope resolution operator)
8. When overloading the assignment operator, =, in a class that dynamically allocates memory, don’t just copy pointers (see below for more information on writing an overloaded assignment operator that performs a deep copy).

General rules for a class if it is to be considered an Abstract Data Type (ADT):

1. All data members are private
2. Every basic operation on the object has a public interface (a function) whose use is independent of the way the object data is stored. For example, I could have written my Date class to store the number of days since January 1, 1970 in one long variable. As long as I still have public get/set member functions for month, day and year, it doesn’t matter.
3. Utility functions are private

# Arrays of objects

An array of objects is just like an array of structures. Some code examples:

// default constructor called for all objects so your

// class better have one!

Date dates[3];

dates[1].setYear( 1999 );

for( i = 0; i < 3; i++ ) {

cout << dates[i] << endl;

}

Declaring an array of objects such as

Date dates[3];

calls the default constructor FOR EVERY OBJECT IN THE ARRAY, in this case, 3 times.

**Pair Programming 7b:** Uncomment the code above in the main function to ensure your class can be used in an array.