Namespaces, Structures and Classes

Namespaces

A ***namespace*** is a collection of name declarations and/or definitions. Your program's interaction with namespaces so far has been limited to using them via the ***using directive***, as in

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

The reason that a program with a line like

```
cout << "Hello world.";
```

fails to compile unless the `using namespace std` directive precedes this statement is that the name `cout`, defined in the header file `<iostream>` has been placed into the namespace `std` within that header file and is unknown outside of that namespace. A namespace defines its own scope and the only way to refer to the members of that namespace outside of it are either to

- put a using namespace directive into the program before the point at which you want to refer to those members, or
- precede each member's name with ***explicit scope resolution***.

The former is what you've done so far, but you have also seen that instead of using the namespace in the program you can write something like

```
std::cout << "Hello world";
```

which tells the compiler to use the name `cout` defined in the namespace `std`. The "`::`" is called the ***scope resolution operator***. The name on the left of it is the name of a scope region such as a namespace, and the name on the right is a member of that scope region. The construct `std::cout` is how you refer to the `cout` member of the namespace `std`.

All of the declarations that you write in a program have scope of one form or another. Recall that block scope is the scope that extends from a point of a declaration within a block to the end of that block, and that file scope is the region of program text from the point of a declaration that is not in any block until the end of the file containing that declaration. All of the names that have file scope in your program actually belong to an unnamed namespace called the global namespace. This does not mean that the namespace is named global! It has no name, but we call it the global namespace.

You can create namespaces of your own. To create a namespace with the name `spacetime` for example, you would write

```
namespace spacetime
{
     /* declarations and definitions here */
}
```

The keyword `namespace` introduces the namespace definition. The name of the namespace follows, in this case `spacetime`, and then what looks like a block: a pair of curly braces with whatever declarations and/or definitions you want to place in them. Notice though that there is no semicolon after the namespace definition. It is not required.

Example

```
namespace spacetime
{

    void printdate( ostream & out, int y, int m, int d )
    {
        out << m <, "/" << d <, "/" << y;
    }

    void printpoint( ostream & out, double x, double y, double z);
}
```

If you then put a directive

```
using namespace spacetime;
```

in your program, you will be able to call the function `printdate()` by writing just the function name itself, but if you do not put this using directive into the program, then you must ***qualify the name*** `printdata()` with the name of its namespace, i.e., you must call `spacetime::printdata()` to use it:

```
spacetime::printdata(2012, 11, 6);
```

The `printpoint()` function is declared in the namespace but its definition is not contained in it. Suppose for some reason that we wanted to write the definition of `printpoint()` outside of the namespace. (Imagine that the namespace has prototypes of functions and only prototypes, and different programmers need to write different implementations of these functions in their own code.) Then you would have to write the function definition as follows:

```
void spacetime::printpoint( ostream & out, double x, double y, double z)
{
    /* code here */
}
```

To summarize, a namespace is a collection of names. To refer to any of those names outside of the namespace itself, you must either insert a `using namespace` directive before the first use of any of those names, or you must qualify those names with the namespace name followed by the scope resolution operator.

Structures

Suppose that we would like to store a student's academic record for processing. It will have string data such as names and id numbers, numeric data such as GPA and the number of credits earned, the courses the student has taken so far, the grades in each, and so on. You could create a separate variable for each data item, and that might be fine for a program that manages one student record, but if you can imagine a program that reads in a list of these records from a text file, perhaps stored one record per line, and has to store each record separately, then you should realize that putting the separate parts of the student records in separate variables is a messy solution. Clearly the data types you have learned about so far, such as strings, scalar types, and arrays, cannot be used very easily to store this complex set of data.

One solution in C and C++ is a type known as a ***structure***. A structure is a ***heterogeneous collection*** of data. It can store data of different types, each with its own name. A structure is introduced with the `struct` keyword, as in the example below.

```
struct User
{
    string firstname;
    string lastname;
    int    score;
};
```

This defines a new data type named `User` that has three members. Each member is a variable that can be assigned data independently of the other members. The first two members of the structure are `string` variables named `firstname` and `lastname`. The third member is an `int` named `score`.

The syntax of the structure definition is

```
struct structurename { list_of_member_declarations };
```

where *`structurename`* is a valid identifier and *`list_of_member_declarations`* is a list of variable declarations. ***The structure definition must be terminated by a semicolon.***

The structure does not have to contain different types of data. Here is a useful structure whose members are all integers:

```
struct Date
{
    int year;
    int month;
    int day;
};
```

Because a structure definition defines a type and not a variable, it can be placed into the global scope so that all functions have access to the definition.

Structures can be initialized when they are declared using the same form of initialization as arrays:

```
User default_user = {"Jody", "Jones", 0 };
```

The values are assigned to the members in the same order as they appear in the declaration. I.e.., it is ***positional assignment***. Variables whose type is a structure are declared the same way as ordinary variables:

```
User current_user;
User default_user;
```

Having declared a variable of the given structure type, the individual members of that variable are accessed using the "dot operator". The syntax is

```
variable_name.member_name
```

as in

```
cin  >> current_user.firstname;
cin  >> current_user.lastname;
current_user.score = 0;
```

Two variables of the same structure type can be assigned one to the other, and the entire structure is copied from one to the other:

```
current_user = default_user;
```

is the same as

```
current_user.firstname = default_user.firstname;
current_user.lastname  = default_user.lastname;
current_user.score     = default_user.score = 0;
```

Suppose a text file has lines consisting of a first name, last name, and a score, such as

```
Elmer Fudd 82
```

A program fragment like the following could be used to read the data from the file so that it can be sorted or processed in other ways.

```
int main()
{
    User    scoredata[MAXUSERS];  //  an array of 100 User structures
    User    temp;                 // temp variable for input
    fstream fin;
    int     length = 0;

    /* open a file and associate with the ifstream fin */
    /* skipping this part ... */

    fin >> temp.firstname >> temp.lastname >> temp.score;
    while ( length < MAXUSERS && !fin.eof() ) {
        scoredata[length] = temp;  // struct-to-struct assignment!!
        length++;
        fin >> temp.firstname >> temp.lastname >> temp.score;
    }

    /* now process the data */
}
```

The following function could be used to sort the data by score in ascending order. It uses a simple selection sort.

```
void sortByScore ( User list[], int len)
{
    int smallest; // index of smallest item in sorted part
    int first;    // index of first item in unsorted part
    User temp;

    for (first = 0; first < len-1; first++) {
        // find index of smallest element in range first to N-1
        smallest = first;
        for (int k = first+1; k <= len-1; k++)
            if ( list[k].score < list[smallest].score)
                smallest = k;

        temp = list[smallest];
        list[smallest] = list[first];
        list[first] = temp;
```

```
    }

}
```

Structures can contain other structures. There is no restriction on the type of the members of a structure. They can contain arrays, structures, arrays of structures, and so on.

Example:

```
struct Date
{
    unsigned int year;
    unsigned int month;
    unsigned int day;
};

struct User
{
    string firstname;
    string lastname;
    int    scores[5];
    Date   birthday;
};
```

Note: ***If an array is a member of a structure, it should be declared with constant size.***

To access the members of a structure that is a member of another structure, you need the dot operator twice. The following example illustrates how the array elements and substructure members are accessed. This function reads from an input stream into a call by reference parameter of type `User`, assuming the data is stored in the stream in the order

```
firstname lastname birthmonth birthday birthyear score1 score 2 score3 score4 score5
```

```
void get_one_record( istream & instream,
                     User & record )
{
    instream >> record.firstname >> record.lastname;
    instream >> record.birthday.month
             >> record.birthday.day
             >> record.birthday.year;
    for ( int i = 0; i < 5; i++ )
        instream >> record.scores[i];
}
```

A program fragment to read a file of such records into an array of `User` structures would look like the following:

```
int main()
{
    User    scoredata[MAXUSERS];
    User    temp;
    fstream fin;
    int     length = 0;
```

```
    /* open a file and associate with the ifstream fin */
    /* skipping this part ... */

    // This code looks  the same as reading a single value. The details
    // are hidden in the function calls.
    get_one_record(fin, temp);
    while ( length < MAXUSERS && !fin.eof() ) {
        scoredata[length] = temp;  // struct-to-struct assignment!!
        length++;
        get_one_record(fin, temp);
    }

    /* now process the data */

}
```

This program fragment demonstrates a subtle point. Although you cannot assign one array to another, as in this code fragment:

```
int a[10]={0}, b[10];
b = a;
```

because it is a syntax error, you ***can*** assign structures containing arrays to each other, as in this fragment:

```
struct S {
    int list[10]
};
S a, b;
for ( int i = 0; i < 10; i++)
    a.list[i] = i;
b = a;
```

and it will do a copy of the entire `a` structure to the entire `b` structure including all elements of their arrays. In the listing above, the variable `temp` is assigned to `scoredata[length]` and this is a structure to structure assignment of a structure type containing an array.

Classes

In the preceding example, the `get_one_record()` function is intimately connected to the definition of the `User` structure above. If that structure is changed, this function must also be changed. In order to access the members of the `User` structure, that structure is an explicit call-by-reference parameter of the function.

The connection between the function and the data definition is a strong one, but the programmer is free to screw everything up because, as the program is currently designed, there is no language mechanism to prevent the programmer from breaking the program by failing to make modifications correctly.

In general, the connection between structure definition and the functions that act on them is weak and prone to error. If the definition changes, so must every function that acts on instances of it. To reduce this chance, one should put all functions and the definition into a single file, with clear instructions. The programmer is still able to foul everything up because there is no language mechanism to prevent the programmer from breaking the program. This is one reason for the invention of an alternative language model known as ***classes***.

A ***class*** is a collection of data and functions. It is like a structure with functions in it, except that it is much more than that. The data and the functions are part of a single entity. When you declare a variable whose type is a class, it has a special name. It is called an ***object***. *Do not get confused between objects and members.* The members of a class are the things defined between the curly braces, its elements. They are like members of a structure. An object of a class is an instance of that class, i.e., a variable whose type is that class.

Access to the members of a class may be restricted by the class's designer. Every member of a class, whether a function or data, has either

- public,
- protected, or
- private

access. We will ignore protected access for now and focus on public and private access.

- A ***public*** member is one that can be accessed by any function in the program. It is just like a member of a struct.
- A ***private*** member cannot be accessed by any functions in the program except those that are members of that class (and some others, to be explained later).

In object-oriented programming, we ***hide*** the data of our class so that no code can see it except the functions of the class to which it belongs. This is called ***data-hiding***. We put all related data together into the class's private data part. This is ***data encapsulation***. The functions that act on this data are the only things that are made public. These functions are the only means by which other parts of the program can access or change the data of the class. Only the function prototypes are put in the public part of the class. The actual definitions will be written someplace else. This is because it is no one's business ***how they work***, but just ***what they do***. This concept of separating how functions work from what they do is called ***procedural abstraction***. The idea of encapsulating the data and functions that act on it into a single entity is called ***data abstraction***.

We will introduce class definitions by example rather than by formal syntax rules.

In the simplest case, the syntax of a class definition is similar to that of a structure definition, except that it includes function prototypes and has access qualifiers:

```
class User
{
// The public interface:
public:
    void get_one_record  ( istream & instream ); // note no User parameter
    void print_one_record( ostream & outstream); // note no User parameter

// The private stuff:
private:
    string firstname;
    string lastname;
    int    scores[5];   // an array of 5 scores
    Date   birthday;    // This member is also a struct. Members can be ANY
                        // previously DECLARED type.
};
```

Points to Remember:

- The keyword `class` introduces the definition.
- The definition ends with a semicolon (just like a `struct`).
- `public:` (don't forget the colon) introduces the list of members that have ***public access***. Until another qualifier is reached, all following members, whether functions or data, have public access.
- The phrase `private:` (again there is a colon) introduces the list of members that have ***private access***s. Until another qualifier is reached, all following members, whether functions or data, have private access.
- If the class has **no access qualifier**, then all members default to ***private access*** until another access qualifier is reached. So the above could also have been written as

```
class User
{
    string firstname;
    string lastname;
    int    scores[5];   // an array of 5 scores
    Date   birthday;    // This member is also a struct. Members can be ANY
                        // previously DECLARED type.
// The public interface:
public:
    void get_one_record  ( istream & instream ); // note no User parameter
    void print_one_record( ostream & outstream); // note no User parameter
};
```

  because the members `firstname`, `lastname`, `scores`, and `birthday` precede the `public:` section.

- Function members are not defined in the class, but only declared. This means only their prototypes are written in the class definition.
- As with structures, array members should have constant size.

The public members of the class are accessed the same way that the members of a structure are accessed, using the dot operator. For example, a main program could contain the lines

```
User user1, user2;
user1.get_one_record( cin );
```

Two instances of the `User` class named `user1` and `user2` are declared, and the member function `get_one_record()` is called on the `user1` instance. We use the language "called on an object" or "called on an instance" to emphasize that the member function is a member of the class but that when it runs, it is acting on the data contained in a specific instance of that class. In this case `get_one_record()` will run on `user1`.

Now we turn to the function definitions. Since the functions are declared within the class but not defined there, their definitions are written outside of the class definition. Because their definitions are outside of the class definition, their form is a bit different than that of a function which is not a member function of a class. This is because a class acts like a namespace; the names declared within the class definition have scope that is limited to that definition. In order to use those names outside of the class definition, they must be qualified in the same way that names in a namespace must be qualified. Thus, the definitions of the `get_one_record()` and `print_one_record()` functions would be as shown in the listing below.

```
// Read a user record from the input stream into a User structure
void User::get_one_record( istream & instream)
{
```

```
    instream >> firstname >> lastname;
    instream >> birthday.month   // birthday is the class member, but it is a
             >> birthday.day     // struct and the dot is needed to get to its
             >> birthday.year;   // month, day, and year members
    for ( int i = 0; i < 5; i++ )
        instream >> scores[i];   // scores is the class member too.
}

// Write a User structure to the output stream, with some formatting
void User::print_one_record( ostream & outstream)
{
    outstream << lastname
              << ", " << firstname
              << "\t" << birthday.month
              << "/" << birthday.day
              << "/" << birthday.year
              << "\t" ;
    for ( int i = 0; i < 5; i++ )
        outstream << scores[i] << " ";
    outstream << endl;
}
```

It is only the name of the function that gets qualified, not the entire prototype. It is incorrect to write the definition like this:

```
User::void get_one_record( istream & instream)
{
    ... // INCORRECT
}
```

because this would mean that `void` is a name defined in the namespace `User` and `get_one_record` is a name defined in the global scope.

Observe that the functions do not have a parameter that is an instance of the class, but they access the private members such as `firstname` and `lastname`. This may seem confusing because `firstname` and `lastname` are not written using the dot operator. Think about it though. If you were to try to use the dot operator, what would be on its left side within this function?

When the function is called, it is called like this:

```
user1.get_one_record( cin );
```

In other words, it is called on a particular instance of the `User` class named `user1`, and so when it runs, the member variables `firstname`, `lastname`, `score` and so on are the members of the object `user1`. This is why they are written without the dot operator.

Separating Class Interface and Implementation

A class interface, or definition, should always be written separately from its implementation and they should be placed into separate files. The interface should be placed in a `.h` file (a header file) and its implementation in a `.cpp` file. This is in the spirit of procedural abstraction. The client code, and the people who write it, do not need to see how the functions are written; they only need to see the prototypes. Because the implementation file has to have access to the class definition, it needs to include the interface file usually, so you should put an `#include` directive in the `.cpp` file.

Imagine that you are writing some class that you would like to distribute to many users. You want to protect your code because you think it may have value. The way that you do this is by compiling the implementation file into an object file and then distributing the header file and the the object file (either in a `.o` file or compiled into a library file.) The header file should be thoroughly documented.

Header Guards

As with any header file, you need to enclose the code in a ***header guard***. The `#ifndef` directive is used to prevent multiple includes of the same file, which would cause compiler errors. #ifndef X is evaluated to true by the preprocessor if the symbol `X` is not defined at that point. X can be defined by either a `#define` directive, or by a `-DX` in the compiler's command-line options. The convention is

```
#ifndef __HEADERNAME_H #define __HEADERNAME_H

//  interface definitions appear here

#endif // __HEADERNAME_H
```

For those wondering why we need this, remember that the `#include` incorporates the named file into the current file at the point of the directive. If we do not enclose the header file in this pair of directives, and two or more included files contain an include directive for the header file, then multiple definitions of the same class (or anything else declared in the header file) will occur and this is a syntax error.

Constructors

When you declare an object of a class, such as in this declaration of `user1`:

```
User user1;
```

you are telling the compiler to create an instance of the `User` class named `user1`. For this particular example, it may not seem too hard for the compiler to figure out how to create a variable of this type. But there are questions. Should the compiler initialize the elements of the `scores` array to some particular value? Should the last and first names be given a particular initial value?

C++ provides a way to initialize the members of a class object when the object is instantiated. When you define a class, you have the option to define a special member function called a ***constructor***. A constructor is a member function that is executed when the object is initialized. Its definition is different from that of an ordinary function in that:

- Constructors have the same name as the class.
- They have no return values and cannot contain a return statement.

As an example prototype for a constructor for our `User` class:

```
User( string fname, string lname, Date bday );
```

defines a constructor with three arguments. Notice that there is no return type. Like ordinary functions, constructors can have arguments. Like ordinary functions, they can be overloaded. Unlike ordinary functions,

- Constructors can have a special feature called an ***initializer list***.

- Whether or not you create a constructor for a class, it does have one. *If you do not provide a constructor for a class, the compiler will generate one for it.* This automatically generated constructor will create an uninitialized object of that class. *If you do provide any constructor for your class, the compiler will not generate one for it.* A constructor that you define is called a ***user-defined constructor***.
- If a class must initialize its data members, then the class needs a user-defined constructor because the compiler-generated constructor will not be able to do this.
- Your program cannot call a constructor! It is called by the run-time library when the object must be initialized.

We illustrate with an example. We enhance the `User` class to contain two overloaded constructors.

Example

```
class User
{
public:
    User();
    User( string fname, string lname, Date bday );
    void get_one_record  ( istream & instream ); // note no User parameter
    void print_one_record( ostream & outstream); // note no User parameter
private:
    string firstname;
    string lastname;
    int    scores[5];
    Date   birthday;
};
```

The first constructor has no parameters. It is an example of a ***default constructor***. *A default constructor is a constructor that can be called with no arguments.*

The second has three arguments. It can be called with a first and last name and a date to be assigned to the birthday member, and it could initialize the scores array to all zeros.

Your program cannot call the constructor. It is called automatically during program execution. When it is called depends upon the type of variable that it is. For now you can think of variables with block scope as being instantiated when the declaration within the block is reached.

The above constructors could be used to declare `User` objects as follows:

```
User user1;                              // default constructor invoked
Date bday = {1950,12,30};                // Create a date object
User user2("Bjarne", "Stroustrup", bday); // second constructor invoked
User user3, user4;                       // default constructor invoked twice
```

- The default constructor is called when you declare an object of the class in the ordinary way, just by writing the variable name after the class name, as in the first line.
- The non-default constructor is called by writing the class name, then the variable name, and after that the parenthesized argument list, as in the third line. This is very different from ana ordinary function call!

The following are illegal ways to declare objects that have user-defined constructors:

```
User user5();  // ILLEGAL - no parentheses allowed
User user6;
const Date bday = {1900,1,1};
user6.User("Sam", "Spade", bday); // ILLEGAL - user6 already exists and
                                  // cannot call by name on an existing object
User user7 = {"Sam", "Spade", bday }; // Cannot initialize using ={}
```

- You cannot put empty parentheses after the variable name. The compiler sees this as a prototype for a function. The first line for example looks like a function named `user5` that returns a `User` object and has no parameters.
- You cannot use a constructor to give values to an object that already exists. The object `user6` was created in line 2, and then a constructor was called on it in line 4, but it was already constructed. That is an error.
- Although you can initialize structures using the curly brace initializer, in general you cannot do this with classes. The only time you can do this with classes is if they have no private section, roughly. Just avoid it.

Defining Constructors

The two constructors above could be implemented as follows.

```
User::User()
{
    firstname = "";
    lastname  = "";
    for ( int i = 0; i < 5;  i++) scores[i] = 0;
    birthday.year = birthday.month = birthday.day = 0;
}

User::User (string fname, string lname, Date bday )
{
    firstname = fname;
    lastname  = lname;
    for ( int i = 0; i < 5;  i++) scores[i] = 0;
    birthday = bday;
}
```

Notice that the constructor name is qualified since its definition is outside of the class definition. The default constructor assigns empty strings to string variables and zeros to integer variables. The non-default constructor assigns the string arguments to the string member variables, and the `Date bday` to the birthday member of the object, using a structure assignment.

We can also define a default constructor in another way. Instead of writing it with no arguments, we can give default arguments to all of its parameters, so that it can be called with no arguments. For example:

```
class User
{
public:
    User( string fname = "", string lname = "" );
    User( string fname, string lname, Date bday );
    void get_one_record  ( istream & instream );
    void print_one_record( ostream & outstream);
private:
    string firstname;
```

```
    string lastname;
    int    scores[5];
    Date   birthday;
};
```

The first constructor can be called with no arguments, in which case `lname` and `fname` will be assigned empty strings.

The following class interface is illegal because there are two constructors that can be called with no arguments, i.e., two default constructors.

```
class User
{  // ILLEGAL CLASS DEFINITION
public:
    User();                                          // default constructor
    User( string fname = "", string lname = "" );    // default constructor
    User( string fname, string lname, Date bday );
    void get_one_record  ( istream & instream );
    void print_one_record( ostream & outstream);
private:
    string firstname;
    string lastname;
    int    scores[5];
    Date   birthday;
};
```

An important rule to remember is that a class cannot have more than one default constructor.

Initializer Lists

Constructors are different from ordinary functions in another way: they can have a special section called an ***initializer section***, or ***initializer list***, in their definitions. A simple one looks like this:

```
User::User (string fname, string lname, Date bday ):
           birthday(bday), firstname(fname), lastname(lname)
{
    for ( int i = 0; i < 5;  i++)
        scores[i] = 0;
}
```

An initializer list is a comma-separated list of ***initializations*** introduced by a colon, immediately after the function header and before the function block. *An initialization is not an assignment operation.* It looks a bit like a function call. You write the name of a member variable from the class, followed by a pair of parentheses, between which is the expression that initializes that variable. Its form is

member-variable-name(initial-value)

In the above example

`firstname(fname)`

is an initialization. Its effect is to copy the value of `fname` into `firstname`, as if it were an assignment operation, but in fact it is more generally a call to a special initializer for the variable. You can think of it like an assignment operation, but it is not. In general, the members of a class can be objects of other classes, as we will see later, and the initializer will invoke their constructors when possible. Note that the `birthday` member, which is a struct, has no "constructor" but we can initialize it just the same using

```
birthday(bday)
```

because the compiler creates what is effectively a constructor for it.

Two important thing to remember about initializer lists are that

- they are executed before the body of the constructor,
- the order in which the individual initializations take place is not the order in which they are listed but the order in which they appear in the class's declaration, which is called their ***declaration order***.

The member variables will be assigned initial values in the following order:

- `firstname` gets a copy of `fname`
- `lastname` gets a copy of `lname`
- `birthday` gets a copy of `bday`

We will say more about initializer lists later.

Explicit Constructor Calls

There is another way for constructors to be invoked, using an explicit constructor call. This is something you may occasionally have to do, but it is an advanced idea. For now we just note it in passing. The following is an alternative way to declare and initialize an object of the `User` class:

```
User newuser;
newuser = User( "Sam", "Spade", bday);
```

Let us break down what this means. The object `newuser` is declared in the first line. The default constructor is used to create `newuser`. It has whatever values are given to it by the default constructor. In the second line, the non-default constructor for the `User` class is invoked explicitly, with what looks like a function call (even though I said that you can never do this earlier). *But this explicit call to the constructor is not on any object.* It creates an object with no name, called an ***anonymous object***. That anonymous object is then copied into the `newuser` object.

To confuse things even further, you can invoke a default constructor explicitly, as in

```
User newuser;
newuser = User();
```

This is different from the illegal

```
User newuser(); // remember that this is illegal!!
```

because the empty parentheses follow the class name, not the object name.

Classes with Class Member Variables

Classes can have members of any type. The members can be members of another class. We can modify our example by turning the `Date` structure into a class to demonstrate.

```
class Date
{
public:
    Date ( int year = 1972, int month = 1, int day = 1 );
    void set     ( unsigned int y, unsigned int m, unsigned int d);
    int  getday  ( );
    int  getmonth( );
    int  getyear ( );

private:
    unsigned int year;
    unsigned int month;
    unsigned int day;
};
```

Now `Date` is a class with a single constructor that can be called with no arguments. If it is called with no arguments, the date is set to January 1, 1972. Assume that our `User` class remains the same. It now has a member that is a member of the `Date` class. We are forced to rewrite our member functions because they access the members of the `Date` class, which are now private. The `User` class is not allowed to access the private members of the `Date` class. Just because it contains a `Date` member does not give it this privilege. Its functions are not member functions of the `Date` class.

It has to use the public member functions of the `Date` class to do its work. They now become

```
User::User()
{
    firstname = "";
    lastname  = "";
    for ( int i = 0; i < 5;  i++)
        scores[i] = 0;
    birthday.set(0,0,0);
}

User::User (string fname, string lname, Date bday ):
           birthday(bday), firstname(fname), lastname(lname)
{
    for ( int i = 0; i < 5;  i++)
        scores[i] = 0;
}
```

The first constructor calls the `set()` member function of the `Date` class to set the date of its `birthday` member. The second constructor did not change. A very subtle thing is going on in the second constructor. The initializer list

```
birthday(bday), firstname(fname), lastname(lname)
```

has an initialization of the `birthday` member. This has to invoke an initializer for the `Date` class. There is only one user-defined constructor for this class and its prototype looks like

```
Date ( int year = 1972, int month = 1, int day = 1 );
```

If we were calling this constructor, we would have to ahve three integer arguments, but we do not. We just have a Date argument, so this is not the constructor that is used to perform the initialization. What is? The compiler generates an initializer that copies the members of `bday` into `birthday`.

We could also create a third `User` class constructor with the prototype

```
User( string firstname, string lastname, int y, int m, int d );
```

and implement it like this:

```
User::User( string firstname, string lastname, int y, int m, int d ):
      birthday(y,m,d), fname(firstname), lname(lastname)
{
    for ( int i = 0; i < 5;  i++)
        scores[i] = 0;
}
```

explicitly calling the constructor of the `Date` class on the `birthday` member of the `User` class. Although this makes the `User` class a bit more flexible, it breaks the data abstraction because the `User` class is given values for the private member variables of the `Date` class in its own constructor.

const Member Functions

Every member function of a class falls into one of two categories: either it modifies some data member of the class or it does not. If it does not modify any member of the class, it is called an ***accessor*** function – it retrieves data from the class object but does not modify it. Usually a class will have several accessor functions. If it modifies any part of the class object, it is called a ***mutator*** function.

You can tell the compiler and the reader of the code that a function is an accessor by putting the `const` keyword after the parameter list in the function's prototype as well as in its definition. The `const` keyword used in this way says that the function is guaranteed not to modify the object on which it is called. To illustrate, we will supplement our `Date` class to use the `const` modifier wherever we can:

```
class Date
{
public:
    Date ( int y = 1972, int m = 1, int d = 1 );
    void set     ( unsigned int y, unsigned int m, unsigned int d);
    int  getday  ( ) const;
    int  getmonth( ) const;
    int  getyear ( ) const;

private:
    unsigned int year;
    unsigned int month;
    unsigned int day;
};
```

The three functions that do not change the object are the *getX()* type functions: `getday()`, `getmonth()`, and `getyear()`. They are marked as `const`. We can now modify the `User` class interface as well:

```
class User
{
// The public interface:
public:
```

```
    User();
    User( string firstname, string lastname, Date bday );
    User( string firstname, string lastname, int y, int m, int d );
    string firstname() const;
    string lastname()  const;
    int    highscore() const;
    int    lowscore()  const;
    void   getbirthday( unsigned int &m, unsigned int &d) const;
    void   read_record  ( istream & instream );
    void   write_record ( ostream & outstream) const;

// The private stuff:
private:
    string fname;
    string lname;
    int    scores[5];
    Date   birthday;
};
```

Remember to put the `const` keyword after the header in the function definitions as well.

If you tell the compiler that the `getbirthday()` member function of the `User` class is `const`, but you do not mark the functions that it calls on that `birthday` member as `const`, then the compiler will issue an error message. This is because the compiler assumes that any function not marked `const` may modify the object on which it is called. The `getbirthday()` function looks like this

```
void   User::getbirthday( unsigned int &m, unsigned int &d) const
{
    m = birthday.getmonth();
    d   = birthday.getday();
}
```

If `getmonth()` and `getday()` are not `const` member functions of the `Date` class, the compiler will assume that they modify the `Date` object, `birthday`, on which they are called. Since `getbirthday()` is supposed to be a `const` function, this contradicts the requirement, so the compiler issues an error message.

You should get into the habit of marking every accessor as a const member function.

Static Member Variables

A member of a class, whether a data member or a member function, can be designated as ***static*** by preceding its declaration with the keyword `static`. Static data members are a bit easier to understand so we will start with them. A ***static data member*** of a class is one that is not a part of any of the objects of the class, but is instead a single member that they all share. It is like a global variable in a program except that it is restricted to be "global" among the objects of the class.

To declare that a data member is static, you have to do two things:

1. You must precede its declaration within the class with the `static` keyword (before the type and any other qualifiers such as `const`), and
2. You must put its definition outside of the class definition, in a surrounding namespace scope, without the `static` keyword but with its name qualified by the :: operator.

As this is surely confusing, here is an example.

Example

```
class Myclass
{
public:
    static int counter;             // declares counter to be static
    static const int maxvalue = 10; // a static constant member
    int sum;
};

int Myclass::counter;        // defines Myclass::counter
const int Myclass::maxvalue; // defines Myclass::maxvalue
```

In this example, the variables `counter` and `maxvalue` are static. The declarations within the class are just declarations, not a definitions. (In this example, you cannot see the difference; trust me for now that there is a difference.) Each must be defined outside of the class but within a surrounding namespace scope. The simplest thing to do is to put its definition in the same file, so that `Myclass` and `Myclass::counter` and `Myclass::maxvalue` are defined in file scope (known as the global namespace.)

If we declare a couple of objects of class `Myclass`,

```
Myclass obj1, obj2;
```

and each will have its own copy of `sum`, but they will share `counter`.

```
obj1.sum = 1;
obj2.sum = 2;
obj1.counter = 0;
obj2.counter = 20;
cout << obj1.counter << endl;  // outputs 20
```

Notice that we accessed `counter` using the member access syntax (i.e., using the dot operator) on specific classes. Because counter does not belong to any one object but is really like a part of the class itself, we can also access it using the syntax `Myclass::counter`. So we can also write

```
Myclass::counter = 20;
obj2.counter    += 10;
cout << Myclass::counter << endl;  // outputs 30
```

Does this look a bit familiar? Recall how, when formatting output on an output stream, you used the instructions

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
```

to display numbers as fixed decimals. The `ios::fixed` argument is a static member of the class known as `ios`.

The `maxvalue` member is a constant member. We are allowed to initialize it within the class declaration. We could have also initialized it in the definition ouside of the class definition.

A ***static member function*** is a member function that cannot access the non-static parts of the class. This implies that it can only access the static data members of the class. If multiple objects share a static data member, they each have the ability to update that member. Although this is possible, it is better if they all call a function that is not called on any particular object. This is one use of a static member function. In the above example, we could define a static member function that resets the counter:

```
class Myclass
{
public:
    static int counter;             // declares counter to be static
    static const int maxvalue = 10; // a static constant member
    static void reset();            // a static function that resets counter
    int sum;
};

int Myclass::counter;        // defines Myclass::counter
const int Myclass::maxvalue; // defines Myclass::maxvalue

void Myclass::reset() : counter(0) {}
```

The keyword `static` is not repeated when the function is defined.