How to Survive and Prosper in the C++ Laboratory

Part II

A Lab Manual
for
Intermediate C++ Programmers

Porter Scobey
Department of Mathematics and Computing Science
Saint Mary's University
Halifax, Nova Scotia, Canada

Latest Revised Printing: January 7, 2004

Table of Contents

Preface

How to Survive and Prosper in the C++ Laboratory (Part II) is a continuation of **Part I** of the C++ Lab Manual of the same title, picks up where that text left off, and has the same underlying philosophy and Module format, for the most part. Sometimes in a "**Sample Programs**" section the subsections "**What you see for the first time in ...**" and "**Additional notes and discussion on ...**" are merged into a section called "**Notes and discussion on ...**" in cases where there really isn't a new sample program with source code, but instead just an executable or a previous sample program being used in a new context. And for the same reason, sometimes the "**Sample Programs**" section itself morphs into "**Sample Programs and Other Files**".

Under the assumption that all or most students using this manual will also have worked through the preceding one, the directions given here will often be less specific, in keeping with a more mature audience. In particular, the specific "bugging" activities of Part I will now be relatively fewer in number, but each one that *is* present is there to make a point of some kind and, as always, you are encouraged to create your own additional bugging activities for each sample program to explore any questions that you may have.

Our understanding is that by the time you come to a given Module in this Part II of the Lab Manual, you will have seen in class, and read about in your text, the topics and terminology referred to in the list of objectives for that Module. Thus you will not generally find a full explanation of all of those terms and concepts repeated here, and the list of objectives is best viewed as a summary of things that you should at least know about before you begin the Module, and which you should have a thorough grasp of when you finish it, because of the activities you have undertaken in working through the Module. In certain cases, though, there may be even more detail than you would normally find in a standard text, either because we are discussing a topic that tends to get short shrift in most texts (command-line parameters in Module 16) or because the example(s) under discussion require a more elaborate treatment (the author-supplied `Menu` and `TextItems` classes in Module 6).

Module 1 introduces *structured data types* and provides a brief introduction to our first example: the `struct` data type. Three sample programs show four different structs, including one *hierarchical struct*. You are also referred to Appendix F, which suggests a convenient way to think about each new data type as you encounter it, and you might as well begin with the `struct`.

Module 2 gives a similarly brief introduction to the `class` data type, which will be used much more that the `struct` in what follows. The sample programs show the class analogs of two of the structs considered in the previous Module. In this Module we deal only with single-file programs, so both the class specification and its implementation will be in the same file as the driver program that uses the class.

Among other examples in Module 1 and Module 2 we show the time-honored and much-used `Time struct` and the equally time-honored `Time class`, both for simplicity and for the sake of comparison. Many C++ textbook authors have used this class in one form or another and it does make an excellent illustrative class. The one you see here is based on the one that appears in Dale *et al.*[1]

In Module 3 we introduce the idea of a multi-file program, i.e., a program in which the source code for a single program has been placed in two or more files. This requires that we discuss *file inclusion* and *separate compilation*, as well as the linking together of two or more files produced by the programmer (together with the usual C++ library files) to form a single executable. For simplicity, this Module does not involve either structs or classes, so it could in fact be covered much earlier. We point out that C++ (and C) permit even a single function to be placed in a separate source code file and separately compiled. Many of the details of the file inclusion and separate compilation process are system-dependent, so you will once again find answer boxes for recording the local answers to relevant questions whose answers differ from one environment to another.

In Module 4 we show how a class is typically separated into a specification file (usually with extension .h) and an implementation file (with an extension .cpp, or whatever extension is used for C++ source files on your system). This again gives us a multi-file program, and requires

- the inclusion of a header file for the class in the (driver program) file where the class is being used
- the separate compilation of the driver file and the class implementation file
- the linking of the two resulting object files to form the required executable

Module 5 introduces the notion of a *constructor* for a class, explains why constructors are needed, and illustrates both the *default constructor* and other constructors.

In Module 6 we introduce very briefly the notion of an *abstract data type* and show how a C++ class can be used to implement an *ADT* (as an abstract data type is often called). This Module also introduces two author-supplied classes—`Menu` and `TextItems`—which are very useful for implementing the user-interface part of a typical *console program* (i.e., non-windows oriented program).

[1] *Programming and Problem Solving with C++* by N. Dale, C. Weems and M. Headington (Sudbury, MA: Jones and Bartlett, 1997)

Module 7 introduces *friend functions* and compares and contrasts them with *member functions* and *non-member functions*. *Inline functions* are not covered in this Lab Manual, but if they were, this would be the place to discuss them as well and you may wish to check another source for details if you are interested.

Module 8 introduces *overloaded operators*, and shows examples of overloaded arithmetic, relational, input and output operators. The Module also contains discussions of overloaded operators as member functions and as friend functions.

Module 9 introduces one-dimensional arrays in which the component type is restricted to being one of the simple data types. Emphasis here is on C++ array basics (declaration and initialization syntax, the fact that indices always start at 0, the for-loop idiom for processing all elements of an array, and so on) and array applications are ignored until Module 11.

Module 10 introduces *pointers*. However, pointers in this Module have nothing to do with the *free store* and *dynamic data storage*, and are used only to point at things that already exist (*static data*). It is never a bad idea to bring in a topic as potentially troublesome as pointers in a reasonably non-threatening situation (which the lack of dynamic data storage provides, in this case) so that the student can get a sense of what pointers are all about from a syntactic and semantic point of view before having to use them in the heat of battle, so to speak. In addition, introducing them immediately after arrays also gives us a chance to point out the intimate relationship between pointers and arrays (though, again, only static arrays), and to do some *pointer arithmetic*. The Module also looks briefly at pointer parameters, and introduces the (implicit) `this` pointer, which is always present in a class object.

Module 11 presents one simple sorting algorithm (*selection sort*) from among the many sorting algorithms we might have chosen, and two of the standard search algorithms: *linear search* and *binary search*. This Module should not be omitted, even on the first pass. The algorithms it contains are so standard and so useful that every student should become aware of them (and get comfortable using them) as soon as possible. They help to convince the student of the usefulness of arrays, and we assume familiarity with them from time to time in the hands-on activities of subsequent Modules.

Module 12 introduces multi-dimensional arrays, with emphasis on the two-dimensional case.

Module 13 introduces classic C-strings and the usual functions needed to manipulate them. No explicit dynamic data storage is used in this Module, but pointer access to a C-string and the individual characters in such a string is illustrated.

Module 14 introduces the C++ string class and compares and contrasts objects of this class with the C-strings studied in the previous Module.

Note that Appendix G also contains a summary of useful information on both kinds of strings, and some additional features not found in either of these two Modules.

Module 15 considers problems and programs that involve some combination of two or more of the following: structs, classes, arrays, strings and files. Each of these topics has previously been considered in one or more Modules devoted

exclusively to that topic, but in this Module you have a chance to combine these different data structures to deal with more complex situations. Pointers are not prominent here in any way, since they have only been given a brief treatment, but pointer access to arrays may of course be used. As a kind of "major project", the reader is left to implement a `Fraction` class in this Module. This will require the ability to find the greatest common divisor within the class. If the reader is unfamiliar with this function, it may be "borrowed" from the GCD_ITER.CPP sample program in Module 22.

Module 16 introduces the notion of a *command-line parameter* and shows how a C++ program can read in (C-string) input values from the same *command line* on which the program is executed. Two sample *utility programs* (or *filters*) involving text indentation and encryption/decryption help to show the general usefulness of command-line parameters.

Module 17 returns to the pointer data type and introduces the *free store* and *dynamic data storage*. Even though this is our second Module on pointers, there are still no linked structures. Just simple *dynamic variables* and *dynamic arrays* are the focus of our attention here.

Module 18 looks at some of the issues that arise when a class contains data members that are stored dynamically. We discuss the need for, and introduce, the "big three" class requirements in this situation: a *class destructor*, a *class copy constructor*, and an *overloaded assignment operator* for the class.

Module 19 introduces the notion of *inheritance* and considers some of the issues that arise when one class *inherits from* (or *is derived from*) another class, including *function redefinition*, the *slicing problem*, *static vs. dynamic binding* and *polymorphism* with *virtual functions*.

Module 20 looks at the notion of *class composition*, which refers to the situation that occurs when a class has one or more data members which are themselves objects of other classes.

Module 21 returns once again to the notion of pointers and dynamic data, this time to deal with "sequences of linked nodes"[2] and the kinds of linked structures that can be built using them. The self-referential nodes in these structures are structs rather than classes. This simplifies the discussion and permits the student to concentrate on the pointer manipulations necessary when dealing with linked structures without having to worry about the complications that would come with having classes involved as well.

Module 22 includes a number of programs illustrating miscellaneous topics, including

- random number generation
- making system calls from a C++ program
- additional string-handling routines

[2]This author prefers the term "sequence of linked nodes" to refer to the thing for which many authors use the term "linked list", and therefore (just in case you were wondering) I reserve the term "linked list" to refer to a "linked" implementation of the ADT List (not something you need to worry about for the moment).

- usage of string streams
- finding the greatest common divisor of two positive integers

Constructive criticism of any kind is always welcome. In particular, any specific suggestions for clarification in the wording, changes in the order of topics, inclusion or exclusion of particular items, or anything that would improve the "generic" aspect of the author's approach to the programming environment, would be especially appreciated.

For the constructive criticism received thus far I would like to thank first of all the many lab instructors who have used Part I of this Lab Manual in its various editions, and especially Patrick Lee for his detailed list of typos and comments. Dr. Pawan Lingras also supplied a number of corrections and suggestions for Part I.

For the second and subsequent printings of Part II, Patrick Lee has once again outdone himself in tracking down typos, obscure passages, and outright errors. I am eternally grateful for his diligence and attention to detail, and students using this manual should be as well. Among those students, I would like to thank as well Maurice McNeille and John Wallace for pointing out a number of problems.

Porter Scobey
Department of Mathematics and Computing Science
Saint Mary's University
Halifax, Nova Scotia
Canada B3H 3C3

Voice: (902) 420-5790
Fax: (902) 420-5035
E-mail: porter.scobey@stmarys.ca

Typographic and Other Conventions

Here are the typographic conventions used in this Lab Manual:

- A *slant font* will be used for *technical terms* (and possibly variations thereof) that are appearing either for the first time, or later in a different context, or perhaps to call your attention to the term again for some reason.
- An *italic font* will be used to emphasize *short* words, phrases, and occasionally sentences of a *non-technical* nature, as well as for *margin notes.* *This is a margin note.*

  However, regular text will be used for a question placed in the margin when an answer box needs to be filled in:

Answer	OK so far?

- **A bold font like this will be used from time to time for a longer passage that is of particular importance, either in context, or for the longer term.**
- A `typewriter-like monospaced font like this` will be used for all C++ code, including C++ reserved words and other identifiers within regular text, for showing input and output to and from programs, and for showing contents of textfiles.

Here are some other conventions that we follow:

- Each sample C++ program or class implementation is in a file with a .CPP (or .cpp)[3] extension,

  and the name of the file (in capital letters) will always appear in a comment in the first line of the file, followed by a comment on one or more lines indicating briefly the purpose of that particular program.

[3]Some operating systems are case-sensitive, i.e., they distinguish between capital letters and lower-case letters (in file names, for example) and some are not.

- Each header file for a class (or collection of functions) is in a file with a .H (or .h) extension,
- All supplied input data files will have an extension of .DAT (or .dat).
- All textfiles, other than those that are also input data files, will have an extension of .TXT (or .txt).
- Although it's a bit anachronistic, we adhere to an 8.3 file naming convention (names of no more than 8 characters, a period, and an extension of no more than 3 characters) for those who may still be working in a DOS environment or wish to transfer sample programs or other files to a DOS environment with a minimum of hassle. Another advantage of this convention (at least in the author's experience) is this: If you stick to short file names without blank spaces, there are a surprising number of problems you *don't* have.
- All discussions about object files and executable files and all associated file names refer to files with .OBJ and .EXE extensions (respectively), as if this were the way the world is. Some systems do not follow this naming convention, so you may have to make name adjustments to reflect your local system, but if this is the case for you it should be possible to decide in advance on a global transformation of some kind that will apply to any particular file or files that you encounter.
- All header files included from the Standard Library appear in lower case, while those that correspond to files appearing in the text will be given in upper case to help set them apart.
- The reader cannot help but notice that all filenames in this manual are capitalized. The reason for this is historical, and stems from the origin of these files on a VMS system, where filenames are case-insensitive, but capitalized by default. This causes no problem under Windows, which is also case-insensitive, but will probably be changed in a future edition.
- Finally, in this manual we often place a line like

  ```
  //*****************************************************************
  ```

  immediately before a function header, particularly in class implementation files, to help the reader identify more readily the beginning of the function.

Module 1

The struct (our first structured data type)

1.1 Objectives

- To understand what is meant by *structured data* and why it is often useful to view data in this way.
- To understand what is meant by a *structured data type* (also called a *data structure*).
- To understand the difference between *procedural encapsulation* (which is implemented by functions) and *data encapsulation* (which is implemented by data structures).
- To understand what a `struct` is and learn how to define and use both *simple structs* and *hierarchical structs*.

New C++ reserved word
`struct`

- To understand the use of the `const` keyword with a reference parameter.
- To understand what a *constant reference parameter* is, and learn how and when to use one.

1.2 List of associated files

- TSTRUCT.CPP illustrates a simple `struct` data type for representing a time in "military" style within a 24-hour day.
- ISTRUCT.CPP illustrates a simple `struct` data type for representing inventory items.
- BSTRUCT.CPP illustrates a hierarchical `struct` data type for representing a bank account.

1.3 Overview

Simple data types

In previous Modules (those of Part I of this Lab Manual) every data value that appeared in any of our programs was a value of some *simple data type*[1] (`int`, `double`, `char`, `bool`, or `enum`). Such a value is *simple* in the sense that conceptually it represents a single value of some kind in the programmer's (or the user's) mind. Also, and again conceptually, in the computer's memory it occupies one "chunk" of memory, even though physically it may take up one or more actual memory locations, depending on the computer's memory architecture and the particular data type of the value in question.

For example, a data value of type `int` may represent a real-world value such as the number of a certain item in inventory or the number of unsuccessful attempts the user has made in guessing the correct answer in a game of some kind. In either case, the value is a single quantity or piece of information, both in the mind of the human thinking about it and in the computer's memory. The actual number of memory locations used to store this quantity may vary from one machine to another, but this fact is irrelevant to our discussion in this context, i.e., irrelevant to our current "high-level" view of the information.

Structured data types

In this Module we begin our discussion of *structured data types*. In a structured data type each value may be viewed in two distinct ways. First, it may be viewed as a single entity in the same way as a value of a simple data type. Second, it may be viewed as a collection—in fact it really *is* a collection—of *component values* ("sub-values", if you like). Just how the collection of component values is actually bound together to form the larger or "higher-level" value—i.e., how the collection is *structured*—may vary considerably, giving rise to a wide variety of different *data structures*.

Data structures

For example, a "list" of bowling scores or a "record" of all the information a university wishes to keep on each student cannot be represented by any of the simple data types that we have been using. Hence we need new data types capable of representing such data to give ourselves more power and flexibility when modeling real-world situations. In this Module we begin the study of such data types.

It is useful to have a certain conceptual pattern in mind as you encounter each new structured data type. In particular, you will find it helpful to ask (and, of course, answer) the sequence of questions listed in Appendix F, so now would be a good time to study that list. The sample programs that illustrate use of each data type will of course help you to answer some or all of those questions, as will the comments we make about those sample programs.

Separation of data structures and functions that act on them

One thing to take away from your study of the sample programs appearing in this Module is the fact that a `struct` data structure and the functions that manipulate it (when it is passed as a parameter) are separate and distinct entities. You will see a major shift in this view of data and the functions that act on it when we begin the study of classes in Module 2.

[1] Except for a brief mention and use of a variable of data type `string` to read in the name of a file.

1.4 Sample Programs

1.4.1 TSTRUCT.CPP illustrates a simple struct data type for representing a time in "military" style within a 24-hour day

```
// Filenme: TSTRUCT.CPP
// Purpose: Illustrates a simple struct for representing time.

#include <iostream>
using namespace std;

struct Time
{
    int hours;
    int minutes;
    int seconds;
}; // <-- Note the (necessary) semi-colon!

void SetTime(/* out */ Time& t,
             /* in */  int newHours,
             /* in */  int newMinutes,
             /* in */  int newSeconds);
void IncrementTime(/* inout */ Time& t);
void DisplayTime(/* in */ const Time& t);


int main()
{
    cout << endl
         << "This program uses a struct to "
         << "represent time in military form. "   << endl
         << "You should study the source code "
         << "at the same time as the output. "    << endl
         << endl;

    Time t1 = {11, 59, 50};  // t1 initialized at time of declaration

    Time t2;  // t2 has its fields assigned after declaration
    t2.hours = 10;
    t2.minutes = 40;
    t2.seconds = 45;

    Time t3;  // t3 is initialized by a call to SetTime
    SetTime(t3, 12, 0, 0);

    DisplayTime(t1);  cout << endl;
    DisplayTime(t2);  cout << endl;
    DisplayTime(t3);  cout << endl;
    cout << endl;

    for (int i = 1; i <= 11; i++)
    {
        DisplayTime(t1);  cout << endl;
        IncrementTime(t1);
    }

    cout << endl;

    return 0;
}
```

```
//*****************************************************************
void SetTime(/* out */ Time& t,
             /* in */  int newHours,
             /* in */  int newMinutes,
             /* in */  int newSeconds)
// Pre:  0 <= newHours   <= 23
//       0 <= newMinutes <= 59
//       0 <= newSeconds <= 59
// Post: Time t is set according to the incoming parameters.
// Note that this function must be called for t prior to a
// call to any other function with t as parameter.
{
    t.hours   = newHours;
    t.minutes = newMinutes;
    t.seconds = newSeconds;
}


//*****************************************************************
void IncrementTime(/* inout */ Time& t)
// Pre:  t has been initialized.
// Post: t has been advanced by one second, with
//       23:59:59 wrapping around to 00:00:00.
{
    t.seconds++;
    if (t.seconds > 59)
    {
        t.seconds = 0;
        t.minutes++;
        if (t.minutes > 59)
        {
            t.minutes = 0;
            t.hours++;
            if (t.hours > 23)
                t.hours = 0;
        }
    }
}



//*****************************************************************
void DisplayTime(/* in */ const Time& t)
// Pre:  t has been initialized.
// Post: t has been output in the form HH:MM:SS.
{
    if (t.hours < 10)
        cout << '0';
    cout << t.hours << ':';
    if (t.minutes < 10)
        cout << '0';
    cout << t.minutes << ':';
    if (t.seconds < 10)
        cout << '0';
    cout << t.seconds;
}
```

1.4.1.1 What you see for the first time in TSTRUCT.CPP

- The definition of a simple `struct` structured data type (namely, `Time`) in which each *field* has the same simple data type (namely, `int`)
- The capitalization convention of user-defined type names: Start with a capital, and use a capital at the beginning of each new word if the name contains more than a single word (not the case with `Time`, of course), but otherwise use lower-case.
- The declaration of three variables (`t1`, `t2`, and `t3`) of type `Time`
- Three different ways in which a variable of type `Time` can be given a value: initialization at the time of declaration; assigning a value to each individual field directly, after declaration; and passing the variable to a function which then sets its value
- The syntax used to refer to the value of a field in a struct variable (`variableName.fieldName`)
- Use of the keyword `const` to modify a reference parameter

1.4.1.2 Additional notes and discussion on TSTRUCT.CPP

Note that in this sample program we *always* pass the struct variable as a *reference* parameter, even when it is conceptually an in-parameter. This is not consistent with our usual practice, and requires some explanation. This is actually the first time you see evidence of a new convention that we will be following:

Always pass a structured variable as a reference parameter.

Why should we do this? Well, a *structured variable* (by which we simply mean a variable whose type is a structured data type) is potentially "large", i.e., it may occupy a lot of storage in memory, and if we pass it as a value parameter a copy is made and that takes time. Passing it by reference does not involve copying the value and thus is more "efficient" (takes less time *and* less space).

But, you complain, what if we want the advantage of "protecting" the original value that "pass by value" gave us in the first place? Well, that's what the `const` modifier of the reference parameter does for us: it *prevents us from modifying, inside the function, the parameter thus passed.* Trying to make such a modification will cause a compile-time error.

One other thing to note in this sample program is the placement of the statement `cout << endl;` on the same line as the call to the function `DisplayTime`. This is another instance where we put two statements on the same line to emphasize their close connection. We include this `cout` statement whenever we want to terminate the line after printing the time. Note that we would not want this statement to be included in the `DisplayTime` function itself since a caller of that function might *not* want to end the line after printing a time value.

1.4.1.3 Follow-up hands-on activities for TSTRUCT.CPP

☐ Copy, study and test the program in TSTRUCT.CPP.

☐ Copy the file TSTRUCT.CPP to TSTRUCT1.CPP and bug it as follows:

Forgetting this semi-colon is a very common error.

1. Remove the semi-colon at the end of the definition of `Time`.

Trying to modify a constant reference parameter is a compile-time error.

2. Add the line

   ```
   t.hours = 0;
   ```

   as the last statement in the body of function `DisplayTime` and re-compile to show that you do in fact get a compile-time error.

---

*Pass-by-value **would** work for struct parameters but we don't use it.*

3. Remove the `const` and the `&` from both the prototype and the function header of `DisplayTime`. Then re-compile, re-link and run again to show that the program behaves exactly as before.

☐ Make another copy of TSTRUCT.CPP called TSTRUCT2.CPP. Add to it the three functions indicated below, for each of which you are given the prototype and a brief comment describing what the function is to do. In each case replace the comment with appropriate pre- and post-conditions for the function. Note from the first requested function that a value-returning function can have a struct value as its return-type.

```
Time TimeFromSeconds(/* in */ int numberOfSeconds);
// Returns the Time value which is numberOfSeconds beyond
// the time 00:00:00.

int TotalSeconds(/* in */ const Time& t);
// Returns the total number of seconds from 00:00:00 to time t.

void DecrementTime(/* inout */ Time& t);
// Decrements t by 1 second, with 00:00:00
// "wrapping around" to 11:59:59.
```

Add appropriate code to the main function to call each of these functions at least twice and display the necessary values to demonstrate that each function is working properly.

◯ Instructor checkpoint 1.1

1.4.2 ISTRUCT.CPP illustrates a simple struct data type for representing inventory items

```
// Filenme: ISTRUCT.CPP
// Purpose: Illustrates a simple struct for representing an
//          item in inventory.

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

struct ItemType
{
    int numberInStock;
    bool warrantyAvailable;
    double price;
};

void InitItem(/* out */ ItemType& item,
              /* in */  int numberInStock,
              /* in */  bool warrantyAvailable,
              /* in */  double price);
void UpdateItemCount(/* inout */ ItemType& item,
                     /* in */    int numberInOrOut);
void DisplayItemInfo(/* in */    const ItemType& item);
void Pause(/* in */ int indentLevel);


int main()
{
    cout << endl
         << "This program uses a struct to "
         << "represent an item in inventory. "   << endl
         << "You should study the source code "
         << "at the same time as the output. "   << endl << endl;

    ItemType item = {15, true, 29.95};
    DisplayItemInfo(item);  cout << endl;
    UpdateItemCount(item, 7);
    DisplayItemInfo(item);  cout << endl;
    Pause(0);

    item.numberInStock = 10;
    item.warrantyAvailable = false;
    item.price = 45.50;
    DisplayItemInfo(item);  cout << endl;
    UpdateItemCount(item, 5);
    DisplayItemInfo(item);  cout << endl;
    Pause(0);

    InitItem(item, 35, true, 100.00);
    DisplayItemInfo(item);  cout << endl;
    UpdateItemCount(item, -15);
    DisplayItemInfo(item);  cout << endl;
    Pause(0);

    ItemType otherItem = item;
    DisplayItemInfo(otherItem);  cout << endl;

    return 0;
}
```

```
//****************************************************************
void InitItem(/* out */ ItemType& item,
              /* in */  int numberInStock,
              /* in */  bool warrantyAvailable,
              /* in */  double price)
// Pre:  All in-parameters have been initialized.
// Post: The information in all in-parameters has been
//       stored in item.
{
    item.numberInStock = numberInStock;
    item.warrantyAvailable = warrantyAvailable;
    item.price = price;
}


//****************************************************************
void UpdateItemCount(/* inout */ ItemType& item,
                     /* in */    int numberInOrOut)
// Pre:  item and numberInOrOut have been initialized.
// Post: The number available of item has been updated by the
//       amount numberInOrOut, which may be positive or negative.
{
    item.numberInStock += numberInOrOut;
}



//****************************************************************
void DisplayItemInfo(/* in */ const ItemType& item)
// Pre:  item has been initialized.
// Post: The information in all fields of item has been displayed.
{
    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);
    cout << setprecision(2);
    cout << "Number in stock:   "   << item.numberInStock << endl;
    cout << "Current price:     $"  << item.price         << endl;
    cout << "Warranty Avaiable: "
         << (item.warrantyAvailable ? "Yes" : "No")       << endl;
}


//****************************************************************
void Pause(/* in */ int indentLevel)
// Pre:  indentLevel has been initialized.
//       The input stream cin is empty.
// Post: The program has paused and displayed a one-line message
//       indented indentLevel spaces, after which the user has
//       pressed the ENTER key.
{
    cout << setw(indentLevel) << ""
         << "Press return to continue ... ";
    char returnChar;
    cin.get(returnChar);
    cout << endl;
}
```

1.4.2.1 What you see for the first time in ISTRUCT.CPP

- The definition of another simple `struct` data type (`ItemType`) in which each field has a *different* simple data type
- The use of the word `Type` as a suffix on the type name, a convention followed by some programmers to clearly identify the names of types in code

  We may do this occasionally, but do not follow the convention strictly. But note that the name `ItemType` confirms our capitalization convention for user-defined type names that was mentioned in our comments on the previous sample program.
- This time we declare a *single* struct variable called `item`, to which we give a value in the same three ways we did in the previous program, displaying the value each time (before it is overwritten).
- The initialization of one struct variable (`otherItem`) with the value of another struct variable (`item`) that already has a value

  The fact that we can do this suggests that we can assign one struct variable to another as well, and this is in fact the case. That is, we could also have written:

```
ItemType otherItem;
otherItem = item;
```

1.4.2.2 Additional notes and discussion on ISTRUCT.CPP

As this sample program and the previous one show, a struct may have fields which are all of the same type, or all different, or anything in between. Also, the two struct definitions you have seen thus far each had three fields, but you may have as many fields as you need, and the number of fields actually required will of course depend on the application.

1.4.2.3 Follow-up hands-on activities for ISTRUCT.CPP

☐ Copy, study and test the program in ISTRUCT.CPP.

☐ Make a copy of ISTRUCT.CPP called ISTRUCT1.CPP. Add to it the function whose prototype and a brief comment describing what the function is to do are given below. Replace the comment with proper pre- and post-conditions.

```
void GetItemInfoFromUser(/* out */ ItemType& item);
// Prompts the user for the information for each field of the
// item, reads in the information and stores it in item.
```

Make sure you display (in the `main` function) the information entered by the user and retrieved by this function.

◯ Instructor checkpoint 1.2

1.4.3 BSTRUCT.CPP illustrates a hierarchical struct data type for representing a bank account

```
// Filenme: BSTRUCT.CPP
// Purpose: Illustrates a simple struct for representing a bank account.

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


struct Person           // Must be defined first so that
{                       // it can be used in BankAccount
    char firstInitial;
    char lastInitial;
    int  idNumber;
};

struct BankAccount
{
    Person owner;       // This field is itself a struct.
    double balance;
};


void InitAccount(/* out */ BankAccount& account,
                 /* in */  int idNum,
                 /* in */  char first,
                 /* in */  char last,
                 /* in */  double balance);
void UpdateBalance(/* inout */ BankAccount& account,
                   /* in */    double amount);
void DisplayAccountInfo(/* in */ const BankAccount& account);
void Pause(/* in */ int indentLevel);


int main()
{
    cout << endl
         << "This program uses a struct "
         << "to represent a bank account. "      << endl
         << "You should study the source code "
         << "at the same time as the output. "   << endl
         << endl;
    Pause(0);

    BankAccount account = { {'P', 'S', 123}, 148.37};
    DisplayAccountInfo(account);  cout << endl;
    UpdateBalance(account, 12.59);
    DisplayAccountInfo(account);  cout << endl;
    Pause(0);

    account.owner.firstInitial = 'W';
    account.owner.lastInitial  = 'C';
    account.owner.idNumber     = 321;
    account.balance            = -175.50;
    DisplayAccountInfo(account);  cout << endl;
    UpdateBalance(account, -345.20);
    DisplayAccountInfo(account);  cout << endl;
    Pause(0);
```

```
    InitAccount(account, 246, 'B', 'C', 100.00);
    DisplayAccountInfo(account);  cout << endl;
    UpdateBalance(account, 150);
    DisplayAccountInfo(account);  cout << endl;
    Pause(0);

    return 0;
}


//****************************************************************
void InitAccount(/* out */ BankAccount& account,
                 /* in */  int idNum,
                 /* in */  char first,
                 /* in */  char last,
                 /* in */  double balance)
// Pre:  All in-parameters have been initialized.
// Post: The information in each in-parameter has been
//       stored in account.
{
    account.owner.idNumber     = idNum;
    account.owner.firstInitial = first;
    account.owner.lastInitial  = last;
    account.balance            = balance;
}


//****************************************************************
void UpdateBalance(/* inout */ BankAccount& account,
                   /* in */    double amount)
// Pre:  account and amount have been initialized.
// Post: The balance in account has been updated by the
//       quantity amount, which may be positive or negative.
{
    account.balance += amount;
}


//****************************************************************
void DisplayAccountInfo(/* in */ const BankAccount& account)
// Pre:  "account" has been initialized.
// Post: The information in "account" has been displayed.
{
    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);
    cout << setprecision(2);
    cout << "Owner:           "  << account.owner.firstInitial
                                 << account.owner.lastInitial << endl;
    cout << "Owner's ID#:     "  << account.owner.idNumber    << endl;
    cout << "Current balance: $"  << account.balance          << endl;
}



//****************************************************************
void Pause(/* in */ int indentLevel)
// Pre:  indentLevel has been initialized.
//       The input stream cin is empty.
// Post: The program has paused and displayed a one-line message
//       indented indentLevel spaces, after which the user has
//       pressed the ENTER key.
```

```
{
    cout << setw(indentLevel) << ""
         << "Press return to continue ... ";
    char returnChar;
    cin.get(returnChar);
    cout << endl;
}
```

1.4.3.1 What you see for the first time in BSTRUCT.CPP

- The definition of a *hierarchical struct* structured data type (`BankAccount`) in which one of the fields is *itself* a `struct` (of data type `Person`)

  It is this "struct within a struct" feature is what makes this data structure *hierarchical*.

- The syntax for the initialization of a hierarchical struct variable at the time of its declaration

- The syntax used to refer to the value of the "innermost" fields in a hierarchical struct variable, which in general is:

  `outerStructVariableName.innerStructvariableName.fieldName`

1.4.3.2 Additional notes and discussion on BSTRUCT.CPP

In this program you begin to get a glimpse of the complexity that structured data types allow us to model, since we can of course "nest" one struct within another to even greater depths.

1.4.3.3 Follow-up hands-on activities for BSTRUCT.CPP

□ Copy, study and test the program in BSTRUCT.CPP.

□ Copy the file BSTRUCT.CPP to BSTRUCT1.CPP and bug it as follows:

1. Interchange the order of the definitions of the two struct data types.

   ---

2. Leave out the inner braces in `{{'P', 'S', 123}, 148.37}`.

□ Make another copy of BSTRUCT.CPP called BSTRUCT2.CPP. Add to it the function whose prototype and a brief comment describing what the function is to do are given below. Replace the comment with pre- and post-conditions.

```
void GetBankAccountInfoFromUser(/* out */ BankAccount& account);
// Prompts the user for all necessary information for a bank
// account, reads in the information and stores it in account.
```

Make sure you display (in the `main` function) the information entered by the user and retrieved by this function.

◯ Instructor checkpoint 1.3

Module 2

The class (our second structured data type)

2.1 Objectives

- To understand what is meant by a new form of *encapsulation* that includes both procedures *and* data, and to appreciate the difference between this view and the prior one, which held that procedures encapsulated tasks to be performed, data encapsulated those things on which the procedures acted, but the two were quite separate and distinct.
- To understand what a `class` is and learn how to define simple classes.
- To understand the *access specifiers* `public` and `private`.
- To understand what an *object* is, and learn how to declare and use simple objects.
- To understand the analogy between the pair (*data type*, *variable*) and the pair (*class*, *object*).
- To understand and be able to define *data members* and *member functions* for a class.[1]
- To understand the use of `const` as a member function modifier.
- To understand the use of the `::` operator (the *scope resolution operator*).
- To understand that although the client of a class (i.e., a programmer who uses objects of the class) does not have direct access to any of the private

New C++ reserved words

```
class
public
private
```

[1] For some reason most authors seem to prefer this minor inconsistency in naming, rather than saying either "member data and member functions" or "data members and function members". For *data members* you will also see the terms *member variables* and *attributes*. For *member functions* you will see *methods* used as well.

data members, every member function of the class *does* have direct access to *any* private data member of the class. This is why the private data members can appear in the body of any member function without being declared in that function or appearing in its parameter list.

2.2 List of associated files

- TCLASS.CPP illustrates a simple class data type for representing military time.
- ICLASS.CPP illustrates a simple class data type for representing an item in inventory.

  You may well have expected to see a file here called "BCLASS.CPP" containing the class analog of the sample program in BSTRUCT.CPP from Module 1. It's no accident or oversight that you don't, however. It's just that the analog would involve having a class in which one of the private data members is itself a class, something we do not discuss until Module 20.

2.3 Overview

In this Module we take our first look at C++ classes. Let's begin by saying this: in C++ the `struct` and the `class` are very similar, except for the way in which they are used. In fact, structs and classes are virtually interchangeable except for one fundamental difference between the two which we will not mention now since such a discussion would not be meaningful at this point. We shall wait until we have looked at our first class in TCLASS.CPP and try to make the point in the hands-on activities.

The `struct` is actually a carry-over from the C programming language, and is used almost exclusively for representing data that is to be manipulated by functions that take struct variables as parameters, in the way that we illustrated in Module 1. Such use is consistent with the classic *procedural approach* to program development, which regards data and the procedures that operate on that data to be separate entities that are somewhat loosely connected by the fact that the data may appear in the interface (parameter list) of the functions.

A more recent methodology, the *object-oriented approach* to program development, regards both the data and the functions that operate on that data as a single entity. It is the C++ `class` that allows us to represent a real world object of some kind by "encapsulating" in a *single* structure both the data that are relevant to the object's representation *and* the functions that we need to work with that data.

As you study the sample programs involving classes which follow, refer back to their analogs involving structs in Module 1 to highlight the differences in the two approaches.

2.4 Sample Programs

2.4.1 TCLASS.CPP illustrates a simple class data type for representing military time

```
// Filenme: TCLASS.CPP
// Purpose: Illustrates a simple class for representing time.

#include <iostream>
using namespace std;


class Time
{
public:
    void Set(/* in */ int hoursNew,
             /* in */ int minutesNew,
             /* in */ int secondsNew);
    // Pre:  0 <= hoursNew   <= 23
    //       0 <= minutesNew <= 59
    //       0 <= secondsNew <= 59
    // Post: Time is set according to the incoming parameters.
    // Note that this function must be called prior to a call
    // to any other member function.

    void Increment();
    // Pre:  The Set function has been called at least once.
    // Post: Time has been advanced by one second, with
    //       23:59:59 wrapping around to 0:0:0.

    void Display() const;
    // Pre:  The Set function has been called at least once.
    // Post: Time has been output in the form HH:MM:SS.

private:
    int hours;
    int minutes;
    int seconds;
};



int main()
{
    cout << endl
         << "This program illustrates the use of "
         << "a simple class for representing time. "  << endl
         << "You should study the source code "
         << "at the same time as the output. "        << endl
         << endl;

    Time t1;
    Time t2;

    t1.Set(11, 59, 50);
    t2.Set(12, 0, 0);

    t1.Display();  cout << endl;
    t2.Display();  cout << endl;

    cout << endl;
```

```
    for (int i = 1; i <= 11; i++)
    {
        t1.Display();  cout << endl;
        t1.Increment();
    }

    cout << endl;

    return 0;
}


//*****************************************************************
void Time::Set(/* in */ int hoursNew,
               /* in */ int minutesNew,
               /* in */ int secondsNew)
// Pre:  0 <= hoursNew <= 23
//       0 <= minutesNew <= 59
//       0 <= secondsNew <= 59
// Post: Time is set according to the incoming parameters.
// Note that this function MUST have been called prior to
// any call to *any* of the other member functions.
{
    hours = hoursNew;
    minutes = minutesNew;
    seconds = secondsNew;
}



//*****************************************************************
void Time::Increment()
// Pre:  The Set function has been called at least once.
// Post: Time has been advanced by one second, with
//       23:59:59 wrapping around to 0:0:0.
{
    seconds++;
    if (seconds > 59)
    {
        seconds = 0;
        minutes++;
        if (minutes > 59)
        {
            minutes = 0;
            hours++;
            if (hours > 23)
                hours = 0;
        }
    }
}



//*****************************************************************
void Time::Display() const
// Pre:  The Set function has been called at least once.
// Post: Time has been output in the form HH:MM:SS.
{
    if (hours < 10)
        cout << '0';
    cout << hours << ':';
```

```
    if (minutes < 10)
        cout << '0';
    cout << minutes << ':';
    if (seconds < 10)
        cout << '0';
    cout << seconds;
}
```

2.4.1.1 What you see for the first time in TCLASS.CPP

On this occasion we opt to take a more narrative approach and add a little more discussion to the mix, rather than simply listing, in the usual bulleted form, the new things you see in this program.

There are many new features in this program, and you must be prepared to spend some time thinking about them. Begin by taking a good look at the overall structure of the program. Notice that when we define a `class` we are in fact just defining a new data type, so that definition (the class `Time`, in this case) appears in the usual place for programmer-defined types. In keeping with our usual overall program structure (see Appendix D, Section D.5 for a reminder), the complete definitions for those functions that deal with the `Time` data follow the `main` function, but it is certainly a new feature to have the prototypes of those function located *inside* the class definition. But that's how it is with classes.

Note that the enclosing syntactical elements for a class are analogous to those for the struct:

```
struct Time          class Time
{                    {
    ...                  ...
};                   };
```

The `class` differs from the `struct`, however, in that the functions that operate on the data (or at least their prototypes) are located inside the class definition as well, and those functions are preceded by the *access specifier* "`public:`". In addition, the data fields (or *data members* as they are called in the case of classes) are preceded by the access specifier "`private:`".

Thus, although there are other variations, and we will see some in later Modules, for simplicity and for the moment our convention will be to write each class definition in the following general form:

```
class NameOfClass
{
public:
    member function prototypes (with pre- and post-conditions)

private:
    data member declarations
};
```

The access specifiers determine whether or not a client of the class can have direct access to a member (data or function) of the class or not. Not surprisingly, a client is not permitted to have direct access to the "private parts"[2] of a class object.[3]

To make this more concrete, note that a statement like

```
t.hours = 6;
```

in `main` is *not* permitted, because all of the data members (including `hours`, therefore) are private and hence may not be accessed in this way. The *only* access to the private data members of a `Time` object is provided by the three functions in the *public interface* to the class, namely `Set`, `Increment` and `Display`. In other words, the client of a `Time` object can `Set` it (which requires setting the `hours`, `minutes` *and* `seconds` via the parameter list of the `Set` function), can increment the value by one second (*after* the value has been set, by calling `Increment`), and can display its current value (if the value has been set) in a certain format by calling `Display`. No other access to a `Time` object or its individual data members is possible.

An important thing to note here is that inaccessibility does not imply invisibility. The client (user) can certainly "see" the private data members of the class, but cannot access them directly when using objects of the class.

If you look at each of the function definitions following `main`, you will note a strange new phenomenon. In each case the variables `hours`, `minutes` and `seconds` appear in the function body without being declared there and without appearing in the parameter list of the function. How is this possible? Well, note that the expression `Time::` appears immediately before the name of each of these functions in the function header. This means that each of these function definitions corresponds to one of the prototypes appearing in the class definition for `Time`, i.e., each function "belongs to" the class (or, more technically, is a *member function* of the class) and hence has *direct access* to the private data members of the class. The name of the `::` operator, by the way, is the *scope resolution operator*, since it indicates the "scope" of the following function is the class `Time`.

In the `Display` function you see yet another use of the `const` keyword. This time it appears immediately following the function header, and this time it has the effect of preventing the `Display` function (or any other function thus modified by `const`) from altering the value of any of the private data members of the class of which the function is a member.

---

[2] There's no telling what kind of comment you would find in the footnotes of less serious authors at this point. But as you know, we try to keep things on a higher plane.

[3] An *object* `t` of a class type `Time` is analogous to an "ordinary" variable `i` of type `int`, for example. An *object* is also frequently referred to as *instance* of the class, in much the same way a given house is an "instance" of the blueprint used to design that particular house. One way of thinking of a class definition is this: it's the "blueprint" used to construct each object of that class type.

2.4.1.2 Additional notes and discussion on TCLASS.CPP

This program also illustrates our conventions for formatting a class definition. We won't bore you by repeating the details here, but take a good look[4] and note when you see other class definitions later how these conventions are followed. Our conventions are not completely standard, but you will see them used quite frequently. We do sometimes vary the amount of vertical spacing used in class definitions and elsewhere to obtain better page breaks, and trust that you will forgive us this minor inconsistency, once again in the name of a higher goal: readability.

Note that the pre/post-conditions of each function appear in two places: with the function prototype in the class definition and also in the function definition itself. There is good reason for this: in Module 4 we will be placing the class definition and the function definitions in separate files, and we want the pre- and post-conditions to be available in both.

2.4.1.3 Follow-up hands-on activities for TCLASS.CPP

□ Copy, study and test the program in TCLASS.CPP.

□ Copy the file TCLASS.CPP to TCLASS1.CPP and bug it as follows:

1. Remove the semi-colon at the end of the definition of `Time`.

   ---

2. Add the line

   ```
   hours = 0;
   ```

   as the last statement in the body of function `Display` and re-compile to show that you do in fact get a compile-time error.

   ---

3. This "bugging exercise" is a key one, and designed, among other things, to highlight the major difference between structs and classes.

   First, to demonstrate the very close parallel between structs and classes, replace the word `class` with `struct` and note that you can compile, link and run as before without any other changes. This might lead you to believe that in fact there is no difference between a `struct` and a `class`. But wait ...

   Next, leave the previous change in place while commenting out the line containing "`public:`". Once again you can compile, link and run as before.

[4] Yogi Berra, the great New York Yankee catcher, and possibly greater philosopher, had the definitive relevant comment here: "You can see a lot just by looking."

Finally, replace struct with class, so that you are back to the original program except that the line containing "`public:`" is still commented out. But now you get compile-time errors.

So, what have you demonstrated? You have illustrated the only major difference between structs and classes: namely, that in a struct, everything is *public by default*, while in a class everything is *private by default*. The fact remains, however that the two tend to be used as we have shown them being used, i.e., in very different ways.

□ This activity attempts to answer the following question: What if the designer of a class decides that the client should have access (or at least read access) to the private data members of the class, for whatever reason? The designer may then decide to provide what are called *accessor functions* in the public interface of the class. That is, an accessor function is (often) a value-returning function that returns the value of one of the private data members.

For example, if we were to provide accessor functions for *all* of the private data members of the Time class, the appropriate prototypes would look like this:

```
int Hours() const;
int Minutes() const;
int Seconds() const;
```

Make a copy of TCLASS.CPP called TCLASS2.CPP and add these three functions to the class interface. This will involve placing the prototypes, complete with pre- and post-conditions, in the class definition, and adding the function definitions after `main`. Be sure to follow the style guideline that requires the order of the function definitions to be the same as the order of the function prototypes, and be sure to add to `main` test code for the functions.

◯ Instructor checkpoint 2.1

□ In this activity you will implement the analogs of the following functions that you implemented earlier when you extended the program in TSTRUCT.CPP:

```
Time TimeFromSeconds(/* in */ int numberOfSeconds);
int TotalSeconds(/* in */ const Time& t);
void DecrementTime(/* inout */ Time& t);
```

What changes are needed in these function prototypes if we wish to do this? For the first two functions in the list, none at all. Of course the function *implementations* will be different, since `Time` is no longer a `struct` but a `class`. But both functions are "external" to the class. Note that in implementing the

`TimeFromSeconds` function you will find the previously implemented accessor functions very handy. What if you didn't have them?

The last of the three functions requires some re-thinking, however. In the case of the `Time` class, you *could* leave this function as is, implementing it as a non-member function, and you may wish to try this (or at least think about how you would do it). But the action of "decrementing" time by one second is analogous to "incrementing" time by one second, and because `Increment` is one of the functions in the public interface of the class it seems more reasonable and consistent to implement the `Decrement` function as a member function in the class interface as well, i.e., as a member function with the following prototype (note the removal of the class name from the function name when the function becomes a member function, another style detail):

```
void Decrement();
```

Make a copy of your TCLASS2.CPP from the previous activity and call it TCLASS3.CPP. Add to TCLASS3.CPP the above three functions, and be sure to add as well test code for these functions in `main`.

◯ Instructor checkpoint 2.2

□ There are some things that you should do at least once, just for the experience, and this activity falls into that category. First, look again at the structure of the program in TCLASS.CPP and note how the class definition was separated from the definitions of the member functions. We did this in preparation for an even more drastic separation, which we will see in Module 4 when we place the class definition and the function definitions in two separate files (called the *class specification file*, or *class header file*, and the *class implementation file*, respectively).

However, it *is* possible to have all of the code for a class together in the same place, even though the standard way of "packaging" a class so that it can be used in many different places most easily is to have the specification and the implementation in separate files. But, just this once, you should prove that you *can* have all the code in one place if you want, as follows.

Make another copy of TCLASS.CPP and call it TCLASS4.CPP. Replace, in the class definition, the prototype of each member function and its associated comment with the entire (corresponding) function definition. When all three replacements have been completed, also remove "`Time::`" from before the name of each function (in the function header). Then show that the revised version of the program works exactly the same as before.

◯ Instructor checkpoint 2.3

2.4.2 ICLASS.CPP illustrates a simple class data type for representing an item in inventory

```
// Filenme: ICLASS.CPP
// Purpose: Illustrates a simple class for representing an
//          item in inventory.

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


class ItemType
{
public:
    void Init(/* in */  int numberInStockNew,
              /* in */  bool warrantyAvailableNew,
              /* in */  double priceNew);
    // Pre:  All in-parameters have been initialized.
    // Post: The information in all in-parameters has been
    //       transferred to the private data members of self.

    void UpdateCount(/* in */ int numberInOrOut);
    // Pre:  self and numberInOrOut have been initialized.
    // Post: The number available of self has been updated by the
    //       amount numberInOrOut, which may be positive or negative.

    void DisplayInfo() const;
    // Pre:  self has been initialized.
    // Post: The information in all fields of item has been displayed.

private:
    int numberInStock;
    bool warrantyAvailable;
    double price;
};

void Pause(/* in */ int indentLevel);


int main()
{
    cout << endl
         << "This program uses a struct to "
         << "represent an item in inventory. "   << endl
         << "You should study the source code "
         << "at the same time as the output. "   << endl << endl;

    ItemType item;

    item.Init(15, true, 29.95);
    item.DisplayInfo();  cout << endl;
    item.UpdateCount(7);
    item.DisplayInfo();  cout << endl;
    Pause(0);

    item.Init(10, false, 45.50);
    item.DisplayInfo();  cout << endl;
    item.UpdateCount(5);
    item.DisplayInfo();  cout << endl;
    Pause(0);

    item.Init(35, true, 100.00);
    item.DisplayInfo();  cout << endl;
```

```
    item.UpdateCount(-15);
    item.DisplayInfo();  cout << endl;
    Pause(0);

    return 0;
}


//*****************************************************************
void ItemType::Init(/* in */  int numberInStockNew,
                    /* in */  bool warrantyAvailableNew,
                    /* in */  double priceNew)
// Pre:  All in-parameters have been initialized.
// Post: The information in all in-parameters has been
//       transferred to the private data members of self.
{
    numberInStock = numberInStockNew;
    warrantyAvailable = warrantyAvailableNew;
    price = priceNew;
}


//*****************************************************************
void ItemType::UpdateCount(/* in */ int numberInOrOut)
// Pre:  self and numberInOrOut have been initialized.
// Post: The number available of self has been updated by the
//       amount numberInOrOut, which may be positive or negative.
{
    numberInStock += numberInOrOut;
}


//*****************************************************************
void ItemType::DisplayInfo() const
// Pre:  self has been initialized.
// Post: The information in all fields of item has been displayed.
{
    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);
    cout << setprecision(2);
    cout << "Number in stock:   "   << numberInStock << endl;
    cout << "Current price:     $"  << price         << endl;
    cout << "Warranty Avaiable: "
         << (warrantyAvailable ? "Yes" : "No")       << endl;
}


//*****************************************************************
void Pause(/* in */ int indentLevel)
// Pre:  indentLevel has been initialized.
//       The input stream cin is empty.
// Post: The program has paused and displayed a one-line message
//       indented indentLevel spaces, after which the user has
//       pressed the ENTER key.
{
    cout << setw(indentLevel) << ""
         << "Press return to continue ... ";
    char returnChar;
    cin.get(returnChar);
    cout << endl;
}
```

2.4.2.1 What you see for the first time in ICLASS.CPP

There is not a great deal new in this program. Its main purpose is to provide another useful class example for you to work with. However, you do see:

- An example of a class in which all private data members have *different* data types.
- Use of the "self" terminology in the pre- and post-conditions of the member functions.

  This is just a handy (and very concise) way of referring to the current instance of the class (the current class object, if you like) when you want to refer to this object as a whole.

2.4.2.2 Additional notes and discussion on ICLASS.CPP

It's probably obvious, but let's point it out anyway: The `Pause` function is *not* a member function. What is it? It's just another "ordinary" function, in fact a "utility" function that turns out to be useful here, and elsewhere, to make a program pause when it's running so that the user can contemplate part of the output before proceeding. It's useful to have a special place to keep such functions and to know how to make use of them with a minimum of effort, and we shall return to this question in Module 3.

2.4.2.3 Follow-up hands-on activities for ICLASS.CPP

□ Copy, study and test the program in ICLASS.CPP.

□ Make a copy of ICLASS.CPP called ICLASS1.CPP. Add to it the function whose prototype and purpose are the same as the analogous function requested in the activities for ISTRUCT.CPP in Module 1. Here is that prototype, in which `ItemType` is now a `class` rather than a `struct`:

```
void GetItemInfoFromUser(/* out */ ItemType& item);
```

Make sure you display (in the `main` function) the information entered by the user and retrieved by this function.

◯ Instructor checkpoint 2.4

Module 3

Multi-file programs and separate compilation

3.1 Objectives

- To understand why you might want to perform *file inclusion*, i.e., to include a separate file of your own code into a program at compile-time, and to learn how to do this on your system.
- To understand what is meant by *separate compilation* and learn how to perform separate compilation on your system.
- To learn how to link several separately compiled files together to form a single executable file on your system.
- To create two files PAUSE.H and PAUSE.OBJ, which together will provide ready and easy access to a `Pause` utility function, and place these files in specific permanent locations from which they can be accessed by any program that needs them.

3.2 List of associated files

- DISPFILE.CPP displays the contents of a file.
- PAUSE.CPP contains just a `Pause` function.

3.3 Overview

As your programs get larger and larger, and because we are soon going to place each of our class definitions into two separate files, we need to look at the whole question of how C++ (and in particular how C++ on *your* system) handles a program when the pieces that make up the program are spread across multiple files, thus giving a *multi-file program*.

3.4 Sample Programs and Other Files

3.4.1 DISPFILE.CPP displays the contents of a file

```
// Filename: DISPFILE.CPP
// Purpose:  Displays the contents of a textfile on the standard output.

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

void DescribeProgram();
void DisplayFile(ifstream& inFile);


int main()
{
    DescribeProgram();

    ifstream inFile;
    inFile.open("in_data");
    cout << "========== Contents of file appear "
         << "below this line ==========" << endl;
    DisplayFile(inFile);
    inFile.close();

    return 0;
}


void DescribeProgram()
// Pre:  none
// Post: The program description has been displayed,
//       preceded and followed by a blank line.
{
    cout << endl;
    cout << "This program displays the contents "
         << "of a textfile called \"in_data\"."         << endl;
    cout << endl;
    cout << "If no output (or bizarre output) appears "
         << "below, the file is not yet available "     << endl;
    cout << "on your system.  Check to be sure the "
         << "file is present, and then try again. "     << endl;
    cout << endl;
}


void DisplayFile(/* in */ ifstream& inFile)
// Pre:  The file denoted by "inFile" exists, and has been opened.
// Post: The data in "inFile" has been displayed.
//       The file is still open.
{
    char ch;

    inFile.get(ch);
    while (!inFile.eof())
    {
        cout.put(ch);  // Another way to ouput a character
        inFile.get(ch);
    }
}
```

3.4.1.1 What you see for the first time in DISPFILE.CPP

The only new C++ feature in this program is the use of `cin.put(ch)` in the body of the `DisplayFile` function to output a single character to the screen (the single character in the `char` variable `ch`). But that's just something to note on the way by, since our main use for this program is going to be to show you how separate compilation works, and we don't want too much that's new distracting you while this is happening.

3.4.1.2 Additional notes and discussion on DISPFILE.CPP

Take a look at this program and note that its structure is typical of the program structure we have been using since we introduced functions and function prototypes. In particular the prototypes come *before* `main` and the full function definitions come *after*. An additional restriction, of course, is that if one function calls another, then the definition of the function doing the calling must *follow* that of the function being called.

But now, think about this ... When the compiler examines the `main` function, all it needs to make sense of the function calls in `main` is its knowledge of the prototypes, which it has already seen. It does *not* need the full function definitions. Put another way, what if the function definitions weren't even there? What if they were removed and placed in a completely separate file, for example? Would this be a problem? For the record, the answer is "no". That's the beauty of prototypes.

However, before the program *runs* it is obviously going to need the information that is contained *only* in the function *definitions*. But as long as that information shows up at *link time* (when the executable is being put together) everything will be OK. To make this happen, we need to *separately compile* the file containing the two function definitions (very fortunately C++ permits us to do this) to produce a separate .OBJ file that we can then *link* with the .OBJ file produced by compiling separately the source file containing `main`.

Now, unfortunately, the details of how you actually perform the separate compilation are highly system-dependent. Even on the same system the details can vary, and depend on the kind of programming environment you happen to be using on that system. For example, a command-line environment may simply require application of the compile command to each file in turn, or simultaneously, while an IDE (Integrated Development Environment) may require adding each file to the current "project" before "building" the executable.

Answer

How do you perform separate compilation and linking on your system?

3.4.1.3 Follow-up hands-on activities for DISPFILE.CPP

□ Copy, study and test the program in DISPFILE.CPP. Choose a short textfile to display (short just so that you can see all of it on a single screen) and make a copy of this file called "`in_data`" so the program will find it.

□ Make a copy of DISPFILE.CPP called DF_MAIN.CPP and delete from it the two functions following `main`. Make another copy of DISPFILE.CPP called DF_FUNS.CPP and delete the prototypes and the main function, leaving the comments and includes at the beginning and the two function definitions at the end. Now adjust the comments at the beginning of each of these two files to accurately reflect their current revised contents. (If you were thinking of *not* bothering to revise these comments, or perhaps not even thinking about it at all, then maybe now is the time to wonder whether your attention to detail is up to par.)

Note that the "includes" that were in the original file have been left in *both* of the new files. Why? Well, in any particular case this may or may not be necessary, depending on what is in each file. But remember that the compiler has to make sense of whatever *is* in each file since each file is to be *compiled separately*. Thus, whatever header files need to be present to provide the compiler with that necessary information *will* have to be included.

So, do whatever is required on your system to compile the files separately and link them to form a single executable. Run this executable to confirm that it works as before.

◯ Instructor checkpoint 3.1

3.4.2 PAUSE.CPP contains just a Pause function

3.4.2.1 What you see for the first time in PAUSE.CPP

This file does not contain a complete program, just a single function. The file also contains the usual comments, as well as the necessary includes so that the file will compile separately. None of this is new. However, what you see that *is* new are the three lines

```
#ifndef PAUSE_CPP
#define PAUSE_CPP
   ...
#endif
```

which are referred to, for brevity if somewhat facetiously, as the three "magic" lines in the activities, and we shall explain their purpose in the course of the discussion.

Note first of all that this particular function is one of those general "utility" functions that we might want to keep around for use in many different programs. One way of doing this, of course, is simply to place both the full function definition and the function prototype in any program where we want to use it. This is what we did, for example, in the program of ICLASS.CPP from Module 2, section 2.4.2. But doing this on all occasions when we want to use the `Pause` function requires a lot of code repetition. There must be a better way, we say, and of course there is.

One approach is to place a line containing `#include "PAUSE.CPP"` on the line after[1] "`using namespace std;`". This will make the `Pause` function available throughout any single-file program that contains this include directive and it can then be called wherever it is needed in that program.

A serious problem may arise with this approach, however. It has to do with the *location* of the file PAUSE.CPP. If PAUSE.CPP is in the same directory as the file into which it is being included, there should be no problem. But will PAUSE.CPP be found if it is located in some other directory? If not, then we will have to make sure a copy of the file PAUSE.CPP is in every directory where we are compiling a program that uses the `Pause` function, once again a severe code duplication problem. Thus we encounter another question that needs a local answer. If you wish to place the file PAUSE.CPP, as well as other similar "utility files" in a specific location, most systems have a way of indicating to C++ where to look for such things at compile-time, so what is it?

[1] Placing it *after* the `using` statement may serve as a reminder that our `Pause` function is *not* part of `namespace std`.

How does your system deal with the problem of locating "include files"?

Answer

Now let's explain what those mysterious three lines do for us. Their purpose is to help us avoid the *multiple inclusion problem*, and you will see three similar lines very frequently if you examine C++ header files. In a nutshell, here's what happens: When the C++ preprocessor encounters the two lines

```
#ifndef PAUSE_CPP
#define PAUSE_CPP
```

it checks to see if the identifier `PAUSE_CPP` has already been defined. If not, the identifier is "defined" at that point. It's pointless to ask anything like, "What *value* is it defined to have?" The question is simply one of being "defined" or not; any particular value would be irrelevant in any case, since the value is not used for anything.

Now here's the neat part. If `PAUSE_CPP` was already defined, then all the code between "`#define PAUSE_CPP`" and "`#endif`" is simply ignored. That is, this code is *not* included with the rest of the code that is passed from the preprocessor to the actual compilation stage.

On the other hand, if `PAUSE_CPP` was *not* already defined, and is therefore defined for the first time at this point, then this code *is* included. This algorithm has the effect of ensuring that the given code is only included *once* for compilation, even though it may be physically included in many different files, several or all of which we may wish to include in a particular program.

By including a similar set of three lines in all such files that we might wish to include in our program, we neatly avoid the nightmare that would otherwise ensue of having multiply-defined functions and/or other entities in our code, and the consequent multiple-definition errors at compile time.

Note the rather unusual form of the identifier `PAUSE_CPP`, which we have formed by taking the name of the file being included, making all letters uppercase, replacing the period with an underscore, and adding an underscore to the beginning and the end. This is the convention that we will follow for identifiers used in this context. The goal is simply to have an identifier which is both meaningful in the context and unlikely to conflict with any other name in the local environment.

We are not yet finished. Many programmers don't like to have their actual source code made available to the whole wide world, for lots of different reasons, even though they may not mind (and might even desire) to have the whole wide world *using* their code. The *principle of information hiding*, which we will encounter again in Module 6, even tells us that being selfish in this regard is not such a bad idea from a software development point of view.

So, the following question now arises: How can we make our `Pause` function available to whoever wants to use it without divulging the source code? Once again, C++ provides a neat solution. What we do is this: We compile our PAUSE.CPP to produce PAUSE.OBJ, we then create a *header file* called PAUSE.H corresponding to PAUSE.OBJ and containing (along with appropriate comments, of course) just the prototype of the `Pause` function.

To prospective users of the `Pause` function we then supply only PAUSE.H and PAUSE.OBJ, while keeping PAUSE.CPP to ourselves. Our users can thus quite happily make use of our `Pause` function by including our PAUSE.H in any file where they wish to call `Pause` and then linking PAUSE.OBJ with the rest of the .OBJ files being linked to form the executable. But none of those users can change or "improve" our `Pause` function because they do not have access to the source code. This illustrates the principle of information hiding, since we have "hidden" the code of the `Pause` function from the users of the function, who have no need to see it.

There is now just one final item to tidy up. We have previously dealt with the question of where to put the PAUSE.H file (which is *included* at *compile-time*) so that it will be found when we attempt to compile a file that includes it. But can a similar arrangement be made for .OBJ files (which are *linked* at *link-time*)? In other words, if we have some .OBJ files that we use quite frequently and we want to keep them in a certain place so that they will always be found when we attempt to link them with other files at link-time, can we arrange for this to happen? Happily the answer is normally yes, but unhappily the answer is again not the same on all systems. So we have yet another question requiring a local answer.

Answer

How does your system deal with the problem of locating .OBJ files for linking?

3.4.2.2 Additional notes and discussion on PAUSE.CPP

The previous discussion may have seemed a little abstract, so it is very important that you make the details concrete in your own mind, and that is what the following hands-on activities are designed to do. They will essentially lead you through that discussion again by giving you explicit instructions to perform the required steps to illustrate the various points being made. You should be prepared to refer back to the above discussion when necessary as you go.

When you have finished the activities, it will be worthwhile to step back a bit and consider the larger picture. What we will have illustrated using PAUSE.CPP and DISPFILE.CPP is a pattern that you will see over and over again when we come back to classes in Module 4, since it is exactly this process

that we will use to separate the code for a class into its *specification file* (*header file*) and its *implementation file*.

3.4.2.3 Follow-up hands-on activities for PAUSE.CPP

□ These activities use both PAUSE.CPP and the program DISPFILE.CPP from the previous section, so you should begin by making a copy of PAUSE.CPP and another copy of DISPFILE.CPP called DFTEST1.CPP. Then re-familiarize yourself with their contents.

□ In DFTEST1.CPP, after the line containing "`using namespace std;`", add the line

```
#include "PAUSE.CPP"
```

and after the line containing "`DescribeProgram();`", add the line

```
Pause(10);
```

and then compile, link and run to test the included `Pause` function.

□ Repeat the previous activity, but with *two* copies of the line
`#include "PAUSE.CPP"`
in DFTEST1.CPP to show that in fact multiple inclusions of the same file do *not* cause a problem when the code is "protected" by those three "magic" lines.

□ Repeat the previous activity, but just before you do, comment out (temporarily) the three "magic" lines in PAUSE.CPP. This time you will have multiple-definition errors, proving that those three "magic" lines do indeed "protect" the code in the way that we indicated.

Before leaving this activity, restore PAUSE.CPP by "un-commenting" the three lines you made into comments and also remove the extra line containing the second

```
#include "PAUSE.CPP"
```

from DFTEST1.CPP.

□ In this activity you will produce the two files PAUSE.H and PAUSE.OBJ that you can continue to use whenever you need a `Pause` function. Begin by compiling separately the file PAUSE.CPP to produce PAUSE.OBJ. Do not try to link it to produce an executable since this will be impossible. It's only a function, not a complete program.

Now make a copy of PAUSE.CPP called PAUSE.H. Delete from PAUSE.H, the body of the function, the include directives, and the `using` statement. Add a semi-colon to the function header to make it a prototype, adjust the comments to accurately describe the contents of PAUSE.H, and (finally) modify the identifier `PAUSE_CPP` to conform to the naming convention mentioned earlier.

Make another copy of DISPFILE.CPP called DFTEST2.CPP and in the new DFTEST2.CPP, immediately following the `using` statement, insert the line

```
#include "PAUSE.H"
```

and also add the line

```
Pause(0);
```

as before.

Now compile DFTEST2.CPP separately, and then link DFTEST2.OBJ with the previously-compiled PAUSE.OBJ. Finally, run the resulting DFTEST2.EXE to show that it behaves as before.

◯ Instructor checkpoint 3.2

□ For this final activity, begin by removing PAUSE.H and PAUSE.OBJ from your current directory and placing them in whatever permanent location(s) you have decided to use for these files.

Delete DFTEST2.OBJ from the previous activity and re-compile DFTEST2.CPP. If you have made whatever arrangement is necessary for your system to know where to find PAUSE.H, then DFTEST2.CPP should compile fine this time as well; otherwise you will get an "include file not found" error and you should make that arrangement now.

When you have succeeded in compiling DFTEST2.CPP under these new conditions, try linking DFTEST2.OBJ and PAUSE.OBJ. If you have arranged for your system to know where to find PAUSE.OBJ then the linking should take place without difficulty; otherwise you will have a link error and you should make the necessary arrangement now.

Some of the "arrangements" that we speak of here and that were the subjects dealt with in the question-and-answer boxes earlier in this Module may have been taken care of automatically by your system administrator and/or your instructor. How much has been done for you will of course determine how many of the details you have to arrange on your own. Dealing with these sorts of questions can cause some frustrating moments, but is always a good learning experience.

In any case, once these arrangements have been completed and are working properly, you will have PAUSE.H and PAUSE.OBJ ensconced in locations from which they can be easily accessed by any of your future programs. You can also make available, in the same way, other similar utility functions or even classes of your own creation. These can be made available from the same location, or other locations set up in an analogous way.

◯ Instructor checkpoint 3.3

Note that in the Hands-On Activities of subsequent Modules the use of the Pause function may be implicitly required, or useful, but may not be explicitly listed as one of the "associated files" for that module or that activity. Such is the way with "utility functions", and you will need to be on the alert for such situations.

Module 4

The class specification and class implementation files

4.1 Objectives

- To learn how to separate the definition of a class into a *specification file* and an *implementation file*.
- To learn how to use a class definition that appears in this form.

4.2 List of associated files

- TESTIME1.CPP is a test driver for the `Time` class defined in TIME1.H and TIME1.CPP.
- TIME1.H is the *specification file* (*header file*) corresponding to TIME1.CPP.
- TIME1.CPP is the *implementation file* corresponding to TIME1.H.
- ICLASS.CPP is the file of the same name from Module 2.

4.3 Overview

In this Module we learn how a C++ class is usually made available to its users: via a *specification file* (also called a *header file*) which contains just the interface to the class and which can be included into the source code where objects of that class are needed, and an *implementation file* which is usually made available as an object code (often with an extension of .OBJ) file, to which any program file that uses the class can be linked.

We use what we learned in Module 3 about separate compilation, but since the classes in this Module are not of general utility we don't bother to put them in special locations for general use.

4.4 Sample Programs and Other Files

4.4.1 TESTIME1.CPP is a driver for the Time class in TIME1.H and TIME1.CPP

```
// Filename: TESTIME1.CPP
// Purpose:  Driver for testing the Time class in the files
//           TIME1.H and TIME1.CPP, with the Time class being made
//           available by including TIME1.H and linking with TIME1.OBJ.

#include <iostream>
using namespace std;

#include "TIME1.H"


int main()
{
    cout << endl
         << "This program accesses a simple Time "
         << "class via separate specification and "   << endl
         << "implementation files. You should study "
         << "the source code at the same time "       << endl
         << "as the output. "                         << endl << endl;

    Time t1;
    Time t2;

    t1.Set(11, 59, 50);
    t2.Set(12, 0, 0);

    t1.Display();  cout << endl;
    t2.Display();  cout << endl << endl;

    for (int i = 1; i <= 11; i++)
    {
        t1.Display();  cout << endl;
        t1.Increment();
    }
    cout << endl;

    return 0;
}
```

4.4.1.1 What you see for the first time in TESTIME1.CPP

It is useful to look at this test driver program for the `Time` class *before* looking at the specification and implementation files which define the class, so that you get a good feeling for how a program obtains access to a class and uses the class, without worrying about the details of the class itself. This, after all, is one of the goals of class development: permitting clients to use them without having to know all the gory details.

Note that the single line

```
#include "TIME1.H"
```

provides all the information about the `Time` class that the program needs to know to compile, since TIME1.H is the specification file for `Time` and hence

contains the prototypes of all member functions in the public interface to the class, and that is all the client of the class is permitted to use.

Including the header file (specification file) for a class in this way is something that you will see often, and often use yourself, from now on.

4.4.1.2 Additional notes and discussion on TESTIME1.CPP

This program is of course not complete by itself, so we shall wait till we have looked at TIME1.H and TIME1.CPP (our next two files) before we attempt to run the program.

But perhaps now is a good time to say a few words about the difference between the two kinds of `#include` directives you have now seen, i.e., the difference between these:

```
#include <...>
#include "..."
```

The difference between the angle brackets (`<...>`) and the double quotes (`"..."`) is that your system uses these to determine where to look for whatever is inside the brackets or quotes. The angle brackets say to the system, "OK, you already know where to look for this, so go ahead." (This assumes your C++ system has been installed properly, of course!) On the other hand, the double quotes say to the system, "Look first for this in the current directory. If you don't find it there, well, maybe this user has set up some environment variables to tell you where else to look so start looking in all those places." Recall that the details of all this vary from system to system and that was a large part of the discussion in Module 3.

4.4.1.3 Follow-up hands-on activities for TESTIME1.CPP

□ Copy and study the program in TESTIME1.CPP.

□ Compile separately TESTIME1.CPP to produce TESTIME1.OBJ that you will use later. Do not try to link and run this file now.

4.4.2 TIME1.H and TIME1.CPP contain the same class definition of Time found earlier in TCLASS.CPP but in separate files

```
// Filename: TIME1.H
// Purpose:  Specification file for a simple Time class.

#ifndef TIME1_H
#define TIME1_H

class Time
{
public:
    void Set(/* in */ int hoursNew,
             /* in */ int minutesNew,
             /* in */ int secondsNew);
    // Pre:  0 <= hoursNew   <= 23
    //       0 <= minutesNew <= 59
    //       0 <= secondsNew <= 59
    // Post: Time is set according to the incoming parameters.
    // Note that this function must be called prior to a call
    // to any other member function.

    void Increment();
    // Pre:  The Set function has been called at least once.
    // Post: Time has been advanced by one second, with
    //       23:59:59 wrapping around to 00:00:00.

    void Display() const;
    // Pre:  The Set function has been called at least once.
    // Post: Time has been output in the form HH:MM:SS.

private:
    int hours;
    int minutes;
    int seconds;
};

#endif
```

Implementation file TIME1.CPP starts below heavy line:

```
// Filename: TIME1.CPP
// Purpose:  Implementation file correponding to TIME1.H.

#include <iostream>
using namespace std;


#include "TIME1.H"


// Private data members of the Time class:
// int hours;
// int minutes;
// int seconds;
```

```
//*****************************************************************
void Time::Set(/* in */ int hoursNew,
               /* in */ int minutesNew,
               /* in */ int secondsNew)
// Pre:  0 <= hoursNew   <= 23
//       0 <= minutesNew <= 59
//       0 <= secondsNew <= 59
// Post: Time is set according to the incoming parameters.
// Note that this function MUST have been called prior to
// any call to *any* of the other member functions.
{
    hours = hoursNew;
    minutes = minutesNew;
    seconds = secondsNew;
}



//*****************************************************************
void Time::Increment()
// Pre:  The Set function has been called at least once.
// Post: Time has been advanced by one second, with
//       23:59:59 wrapping around to 00:00:00.
{
    seconds++;
    if (seconds > 59)
    {
        seconds = 0;
        minutes++;
        if (minutes > 59)
        {
            minutes = 0;
            hours++;
            if (hours > 23)
                hours = 0;
        }
    }
}



//*****************************************************************
void Time::Display() const
// Pre:  The Set function has been called at least once.
// Post: Time has been output in the form HH:MM:SS.
{
    if (hours < 10)
        cout << '0';
    cout << hours << ':';
    if (minutes < 10)
        cout << '0';
    cout << minutes << ':';
    if (seconds < 10)
        cout << '0';
    cout << seconds;
}
```

4.4.2.1 What you see for the first time in TIME1.H and TIME1.CPP

- The separation of a class definition into two separate files which are called the *specification file* (or *header file*) and the *implementation file*

  In the case of the class `Time` the specification file is TIME1.H and the implementation file is TIME1.CPP.

- The protection of the class specification against *multiple inclusion errors* (via `TIME1_H` in TIME1.H) (See Module 3 for details.)

- The inclusion, in the implementation file (TIME1.CPP) of both a C++ Standard Library header file (`iostream`) and our own class specification file (TIME1.H) so that the class implementation file (TIME1.CPP) can be compiled separately

- The appearance, in the implementation file, of the private data members as a comment, just to aid any human reader of the file

4.4.2.2 Additional notes and discussion on TIME1.H and TIME1.CPP

In order to *use* a class in a program, the client must of course be able to *see* the specification file in order to know what is available for use in the public interface of that class. In looking at the specification file, the client also sees the private data members, which, as you know from Module 2 are nevertheless inaccessible unless the class designer has provided accessor functions. But even the fact that they *are* visible is actually a violation of the *principle of information* hiding, which means that C++ is not as good as we might like it to be at enforcing this principle.

4.4.2.3 Follow-up hands-on activities for TIME1.H and TIME1.CPP

□ Copy and study the files TIME1.H and TIME1.CPP.

□ Compile separately TIME1.CPP to get TIME1.OBJ.

□ Now link your previously obtained TESTIME1.OBJ with TIME1.OBJ to form the executable TESTIME1.EXE and run the executable to show that it works properly.

□ This activity assumes you have available PAUSE.H and PAUSE.OBJ, either in your current directory or in their "permanent" locations as discussed previously in Module 3.

Modify TESTIME1.CPP by adding the include directive

```
#include "PAUSE.H"
```

immediately after the `using` statement, and also add two calls to the `Pause` function that look like this:

```
Pause(0);
```

Add one of these after the two calls to the `Display` member function, and the other after the for-loop. Then re-compile TESTIME1.CPP and link the resulting TESTIME1.OBJ with both TIME1.OBJ and PAUSE.OBJ. Run the new TESTIME1.EXE to show that pauses do appear at the appropriate places.

◯ Instructor checkpoint 4.1

□ Now add a brand new function to the file TESTIME1.CPP, one with the following header and pre/post-conditions:

```
void DisplayFileTimes();
// Pre:  A textfile called MYTIMES.DAT exists, and contains at
//       at least one time of the (typical) form 11:22:33.
//       Each time appears on a separate line.
// Post: The file has been opened, all times have been read
//       and displayed on the screen, one per line, and the
//       file has then been closed.
```

Add a call to the function `DisplayFileTimes` just before the return statement in the `main` function of TESTIME1.CPP to test this function. Then re-compile TESTIME1.CPP and re-link TESTIME1.OBJ once again.

Next, create a file called MYTIMES1.DAT containing the following data

```
11:22:33
00:23:54
12:56:01
03:12:00
```

Now copy MYTIMES1.DAT to MYTIMES.DAT and run TESTTIME1.EXE to show that these four times are displayed.

Finally, make a copy of MYTIMES1.DAT called MYTIMES2.DAT and add to the end two more times so that this new file contains the following data:

```
11:22:33
00:23:54
12:56:01
03:12:00
01:33:10
22:13:24
```

Copy MYTIMES2.DAT to MYTIMES.DAT and run your program again to show that all six `Time` values are now displayed. If they are *not* displayed, either because you did not finish reading this problem before you went ahead and did what you thought was required, or because you "hard-wired" the printing out of four values into your code (or both), this is a sign you need to stop and re-think some things. If the six values were printed out properly with no problem, wonderful! Congratulations! Carry on!

◯ Instructor checkpoint 4.2

4.4.3 ICLASS.CPP is the file with the same name from Module 2

You will now revise this program from a previous Module in a way that parallels the transformation of TCLASS.CPP, also from Module 2, to TESTIME1.CPP, TIME1.H and TIME1.CPP.

4.4.3.1 Notes and discussion on ICLASS.CPP

It is probably worth pointing out that what you are doing here and the way you are doing it are for pedagogical purposes so that you get a good first look at the details of the way the pieces of a class definition are arranged in the specification and implementation files. *We certainly do not mean for you to conclude that you should begin by putting it all in one file and then later chopping the file up into the necessary pieces.* When we are designing classes "for real", we want to keep the required parts of each class in the corresponding specification and implementation files right from the start.

4.4.3.2 Follow-up hands-on activities for ICLASS.CPP

□ Reacquaint yourself with the program in ICLASS.CPP. Then make three copies of ICLASS.CPP called TESTITEM.CPP, ITEM.H and ITEM.CPP. Begin by deleting from ITEM.H everything except what is necessary for the specification file for the `ItemType` class. Next, delete from ITEM.CPP everything except what is necessary for the implementation file corresponding to ITEM.H. Then delete from TESTITEM.CPP everything except what is necessary for the remaining code to be the test driver for the `ItemType` class.

Revise the comments in each of the three files so they are consistent with the revised file contents, "protect" the class specification in ITEM.H with `ITEM_H`, and replace the prototype and definition of the `Pause` function with the appropriate include directive.

Finally, remove the prototype as well as the definition of the `Pause` function from TESTITEM.CPP. In their place insert, into the `#include` section of the program, the line

```
#include "pause.h"
```

because this time around you want to make use of a separately-compiled file containing this function.

When all is ready, first compile TESTITEM.CPP to get TESTITEM.OBJ. Next, compile ITEM.CPP to get ITEM.OBJ and PAUSE.CPP to get PAUSE.OBJ. Then link TESTITEM.OBJ with ITEM.OBJ and PAUSE.OBJ to get the executable TESTITEM.EXE. Run this executable to show that it behaves like the original executable ICLASS.EXE obtained from ICLASS.CPP.

◯ Instructor checkpoint 4.3

Module 5

Class constructors

5.1 Objectives

- To understand what a *constructor* for a class object is, and why we need constructors for objects.
- To understand how to define and invoke both the *default constructor* and other constructors for a class object.

5.2 List of associated files

- TESTIME2.CPP is a test driver for the `Time` class defined in TIME2.H and TIME2.CPP.
- TIME2.H is the specification file for the second version of the `Time` class and contains constructors.
- TIME2.CPP is the implementation file corresponding to TIME2.H.
- TESTITEM.CPP, ITEM.H and ITEM.CPP are the files of the same names *you* created in Module 4, section 4.4.3.2.

5.3 Overview

In this Module we introduce the notion of a *class constructor*. You will recall from previous versions of the class `Time` (for example) that the client (user) of the class had to remember to call the `Set` member function for any object of the class before any other member function could be applied to that object. This is a reasonable request, and the requirement was spelled out explicitly in the class interface, as all such requirements should be. But what if the client forgets? Well, there could be dire consequences if an object is used before it has

a value (if some event were to take place at a time that hadn't been properly established, for example).

The purpose of having class constructors is to avoid this problem by ensuring that *any* object of the class *always* has a value. So, we can think of a constructor as a function that makes sure all the data members of a class object have initial values at the time the object comes into existence. But, *what* values? The answer to this question is: it depends. The thing is, we can have many different constructors for a class and therefore it is possible to have an object initialized in many different ways. However, as long as we have a *default constructor* we are guaranteed that each object will be initialized even if we "forget" to explicitly initialize it ourselves.

For example, if we make a declaration like

```
Time t;
```

then in our previous versions of the `Time` class the object `t` has no value until we apply the `Set` member function to it.

However, as you will see, with the version of `Time` we present in this Module t *does* have a value the moment it is declared because the default constructor executes "behind the scenes" and gives it the value we have decided it should have "by default", i.e., if we forget, or don't bother, to give it any other value.

Notice what we've said here: the default constructor executes "behind the scenes", which is just another way of saying we don't have to explicitly invoke it. If we had to explicitly invoke it, what would be the point? We could still forget to do it! Also, even though the default constructor is automatically invoked in this situation, the value it gives to `t` has been pre-determined by the designer of the class when the default constructor itself was implemented.

As we mentioned above, we can have many other (non-default, if you like) constructors. (Obviously there can be only *one* default constructor, right?) In this Module we give only one non-default constructor, but in the activities we ask you to provide a second.

Suppose, at the time of declaring an object `t` of class `Time`, we also want that particular `t` object to have the value 15:25:00. (Maybe this is an alarm time so we can catch our favorite soap opera at 3:30pm.) Then we could of course use the `Set` function on `t` after declaring `t`, but we can also have a constructor for the `Time` class that does the job for us at the same time that `t` is declared. Assuming for the moment that this constructor has been defined appropriately, the syntax for making this happen would be this:

```
Time t(15, 30, 0);
```

Notice that this is quite different from calling the `Set` function, and note too that this does not at all mean that we no longer need the `Set` function, since we may still want to change, i.e., "set", the value of a `Time` object *after* it has been declared.

So, to invoke a non-default constructor, the general syntax is

```
ObjectType objectName(value1, value2, ..., valueN);
```

where `value1`, `value2`, ..., `valueN` provide some or all of the values to be given to the data members of `objectName`.

5.4 Sample Programs and Other Files

5.4.1 TESTIME2.CPP is a driver for the Time class in TIME2.H and TIME2.CPP

```
// Filenme: TESTIME2.CPP
// Purpose: Illustrates a simple time class, with the class accessed from a
//          separate header (.H) file and separately compiled object (.OBJ)
//          file.  This driver tests the version of the Time class containing
//          two constructors, and two other (non-member) functions.

#include <iostream>
using namespace std;


#include "TIME2.H"
#include "PAUSE.H"


void GetTimeFromUser(Time&);
Time IncrementedTime(const Time&, int);


int main()
{
    cout << endl
         << "This program illustrates the Time class "
         << "with constructors, plus two non-member "   << endl
         << "functions.  You should study the source "
         << "code at the same time as the output. "     << endl << endl;

    Time t1(11, 14, 57);
    Time t2;
    t1.Display();  cout << endl;
    t2.Display();  cout << endl;
    Pause(0);

    Time t3 = t1;
    Time t4;
    for (int i = 1; i <= 3; i++)
        t1.Increment();
    t4 = t1;
    t3.Display();  cout << endl;
    t4.Display();  cout << endl;
    Pause(0);

    Time myTime;
    GetTimeFromUser(myTime);
    myTime.Display();  cout << endl;
    Pause(0);

    int numberOfSeconds;
    cout << "Enter a number of seconds to increment the time: ";
    cin >> numberOfSeconds;  cin.ignore(80, '\n');  cout << endl;
    Time yourTime = IncrementedTime(myTime, numberOfSeconds);
    yourTime.Display();  cout << endl;
    Pause(0);

    IncrementedTime(yourTime, 5).Display();  cout << endl;

    return 0;
}
```

```
void GetTimeFromUser(/* out */ Time& t)
// Pre:  none
// Post: "t" contains a time value entered by the user from the keyboard.
{
    int hours;
    int minutes;
    int seconds;

    cout << endl;
    cout << "Enter the hours, minutes and "
         << "seconds for a time. "               << endl
         << "Enter the values in that order, "
         << "separated by a blank space: ";
    cin >> hours >> minutes >> seconds; cin.ignore(80, '\n'); cout << endl;

    t.Set(hours, minutes, seconds);
}


Time IncrementedTime(/* in */ const Time& t,
                     /* in */ int numberOfSeconds)
// Pre:  "t" contains a valid time value.
// Post: The return-value of the function is the time value obtained
//       by incrementing "t" by numberOfSeconds seconds.  The value
//       of "t" itself is unchanged.
{
    Time tempTime = t;
    for (int i = 1; i <= numberOfSeconds; i++)
        tempTime.Increment();
    return tempTime;
}
```

5.4.1.1 What you see for the first time in TESTIME2.CPP

We'll try not to belabor the point too much in subsequent Modules, but let's say again that we think it useful to look at driver programs before we look at what they're driving (a kind of top-down approach to understanding what is going on), and though we may not always do it this will be our general approach.

In this Module, although our main focus is on constructors, in this sample program you will see that there are six different `Time` objects, and each one receives its value in a different way:

- `t1` is initialized at the time of declaration by the non-default constructor, which takes three parameters.
- `t2` is initialized at the time of declaration by the default constructor.

  Note that the details of both of these constructors are not available here. You will see them shortly in section 5.4.2.
- `t3` is initialized at the time of declaration with the value of another `Time` object (namely, t1) that already has a value.
- `t4` is first initialized by the default constructor at the time of declaration and then is *assigned* the value of another `Time` object (namely, `t1`) whose value has been changed since it was used to give `t3` a value.

- `myTime` is also first initialized by the default constructor at the time of declaration and is then passed to the function `GetTimeFromUser` as a reference parameter and it is this function that gets the hours, minutes and seconds from the user and uses `Set` to give `myTime` a value which is then returned by the reference parameter.
- `yourTime` is initialized at the time of declaration by the value of a value-returning function whose return-type is the `Time` class.

One other new thing to note in this program is what the line

```
IncrementedTime(yourTime, 5).Display();  cout << endl;
```

Note this syntax.

near the end of the program shows you. This may seem strange syntax indeed, till you think about it. What we have done, in effect, is treat the function call as a `Time` object and apply the member function `Display` to it. But of course this is a value-returning function and its return-type is `Time`, so that's what makes this work.

5.4.1.2 Additional notes and discussion on TESTIME2.CPP

Study the two function definitions `GetTimeFromUser` and `IncrementedTime` that you find in this program and note again that there may be many things you want to do with a `Time` object that you cannot do using the supplied public interface alone. It may be that you can write your own function to do some or all of those things, like we did here with the two functions shown. Not everything you want to do may be possible, and preventing you from doing certain things is the whole point of having private data members only accessible via a public interface in the first place.

5.4.1.3 Follow-up hands-on activities for TESTTIME2.CPP

□ Copy and study the program in TESTIME2.CPP. Without necessarily looking ahead to TIME2.H and TIME2.CPP, compile TESTIME2.CPP and TIME2.CPP separately and then link the .OBJ files with PAUSE.OBJ to get the TESTIME2.EXE executable. Before running it study the program, decide what input you will supply, and predict all output. Then run the program to confirm your predictions.

□ Now make a copy of TESTIME2.CPP called TT2A.CPP. Revise the program so that the `Time` values output when this revised driver runs will be the ones shown below, which should appear in the given order but on separate lines in the output. Arrange to have the given output by *only* changing *literal numerical values*) and choosing appropriate input values,

```
13:21:55  00:00:00  13:21:55  13:22:12  11:55:44  12:00:50  12:01:01
```

◯ Instructor checkpoint 5.1

5.4.2 TIME2.H and TIME2.CPP extend the Time class by adding two constructors

```
// Filename: TIME2.H
// Purpose:  Extends the class in TIME1.H by adding two constructors.

#ifndef TIME2_H
#define TIME2_H

class Time
{
public:

    Time();
    // Default constructor
    // Pre:  none
    // Post: Class object is constructed and the time is 00:00:00.
    //       That is, hours, minutes and seconds all have value 0.

    Time(/* in */ int hoursInitial,
         /* in */ int minutesInitial,
         /* in */ int secondsInitial);
    // Constructor
    // Pre:      0 <= hoursInitial   <= 23
    //           0 <= minutesInitial <= 59
    //       and 0 <= secondsInitial <= 59
    // Post: Class object is constructed.
    //       The time is set according to the incoming parameters.


    void Set(/* in */ int hoursNew,
             /* in */ int minutesNew,
             /* in */ int secondsNew);
    // Pre:  0 <= hoursNew   <= 23
    //       0 <= minutesNew <= 59
    //       0 <= secondsNew <= 59
    // Post: Time is set according to the incoming parameters.

    void Increment();
    // Pre:  none
    // Post: Time has been advanced by one second, with
    //       23:59:59 wrapping around to 00:00:00.

    void Display() const;
    // Pre:  none
    // Post: Time has been output in the form HH:MM:SS.


private:

    int hours;
    int minutes;
    int seconds;
};

#endif
```

```
// Filename: TIME2.CPP
// Purpose:  Implementation file corresponding to TIME1.H.
//           This version adds two constructors to the version in TIME1.CPP.

#include <iostream>
using namespace std;

#include "TIME2.H"


// Private data members of the Time class:
//     int hours;
//     int minutes;
//     int seconds;


//*****************************************************************
Time::Time()
// Default constructor
// Pre:  none
// Post: Class object is constructed and the time is 00:00:00.
//       That is, hours, minutes and seconds all have value 0.
{
    hours = 0;
    minutes = 0;
    seconds = 0;
}


//*****************************************************************
Time::Time(/* in */ int hoursInitial,
           /* in */ int minutesInitial,
           /* in */ int secondsInitial)
// Constructor
// Pre:      0 <= hoursInitial   <= 23
//           0 <= minutesInitial <= 59
//       and 0 <= secondsInitial <= 59
// Post: Class object is constructed.
//       The time is set according to the incoming parameters.
{
    hours = hoursInitial;
    minutes = minutesInitial;
    seconds = secondsInitial;
}


//*****************************************************************
void Time::Set(/* in */ int hoursNew,
               /* in */ int minutesNew,
               /* in */ int secondsNew)
// Pre:  0 <= hoursNew <= 23
//       0 <= minutesNew <= 59
//       0 <= secondsNew <= 59
// Post: Time is set according to the incoming parameters.
{
    hours = hoursNew;
    minutes = minutesNew;
    seconds = secondsNew;
}
```

```
//*************************************************************
void Time::Increment()
// Pre:  none
// Post: Time has been advanced by one second, with
//       23:59:59 wrapping around to 00:00:00.
{
    seconds++;
    if (seconds > 59)
    {
        seconds = 0;
        minutes++;
        if (minutes > 59)
        {
            minutes = 0;
            hours++;
            if (hours > 23)
                hours = 0;
        }
    }
}


//*************************************************************
void Time::Display() const
// Pre:  none
// Post: Time has been output in the form HH:MM:SS.
{
    if (hours < 10)
        cout << '0';
    cout << hours << ':';
    if (minutes < 10)
        cout << '0';
    cout << minutes << ':';
    if (seconds < 10)
        cout << '0';
    cout << seconds;
}
```

5.4.2.1 What you see for the first time in TIME2.H and TIME2.CPP

The main thing to note in these two files, of course, is the prototypes and the definitions of the two class constructors, and we need to make the following observations:

- Each constructor is a function whose prototype appears in the specification file and whose complete definition appears in the implementation file, just like any member function of the class.
- Both constructors have the *same* name, which in fact is also *the name of the class itself* (`Time`, in this case).

  So, how are constructors distinguished? As usual, by the number and types of their parameters, which must be different for each constructor. Thus, it is the fact that C++ allows *function overloading* that permits us to have many constructors for a class.
- One constructor has *no* parameters. This is the default constructor, and

in fact it is always the case that the (unique) default constructor for *any* class has no parameters.

- The second constructor (and the only non-default constructor in this case) has three parameters, but it could have any number (except 0).
- Observe how the pre/post-conditions of the member functions `Increment` and `Display` have changed, in that we no longer require the comment indicating that `Set` must called before invoking these functions.

5.4.2.2 Additional notes and discussion on TIME2.H and TIME2.CPP

Note again that in many ways constructors are functions like any other member functions of a class, but they are *not* invoked in the same way, and in fact it would be an error to write something like `t.Time()` or `t.Time(11,22,33)`.

5.4.2.3 Follow-up hands-on activities for TIME2.H and TIME2.CPP

□ Copy and study the files TIME2.H and TIME2.CPP. Then make the following (temporary) changes:

- Change the default constructor so that the default `Time` value is noon.
- Add a second non-default constructor that has a single in-parameter which is a number of seconds in the range 0..86399 (which will not seem strange once you have calculated the number of seconds in a day) that sets a `Time` value equivalent to this number of seconds from 00:00:00.
- To prove that each constructor really is being called "behind the scenes", insert at the end of the body of each constructor two additional statements: the first should be a cout statement saying which constructor you're in; the second should be a call to the `Pause` function.

Generate a new TESTIME2.EXE with this revised TIME2.CPP and run it to show that you get the expected output. Don't forget to remove the changes before proceeding, or simply make a fresh copies of TIME2.H and TIME2.CPP.

□ This activity is designed to illustrate a very important point, to which we shall return in Module 6: the distinction between the public interface to a class and the (private) implementation of that class.

To make the idea concrete, note that a client of the `Time` class need not know, and should not care, how the `Time` value of a `Time` object is actually stored. So, whether the class stores hours, minutes and seconds as three separate data members or whether the class simply stores the total number of seconds does not matter to the client, so long as the interface (the public member functions) does not change.

Make copies of the original TIME2.H and TIME2.CPP called, respectively, TIME2S.H and TIME2S.CPP (S for Seconds, if you like), and make the change

suggested above. That is, replace the three private data members `hours`, `minutes`, and `seconds`, with a single private data member called `totalSeconds`, of type `int`. Keep in mind that you *must not change the class interface* (i.e., the function prototypes or headers) but of course you *will* have to change some or all of the member function implementations, i.e., the function bodies. This, in fact, is the point we wish to make.

When you have made the necessary changes, make one small (and temporary) change in TESTIME2.CPP to include TIME2S.H rather than TIME2.H. Then compile TESTIME2.CPP and TIME2S.CPP separately and link the .OBJ files with PAUSE.OBJ to prove that the driver program works exactly as before with this new version of the class.

◯ Instructor checkpoint 5.2

5.4.3 TESTITEM.CPP, ITEM.H and ITEM.CPP

These files are the ones of the same names that *you* produced in Module 4, section 4.4.3.2.

5.4.3.1 Notes and discussion on TESTITEM.CPP, ITEM.H and ITEM.CPP

The class definition and test driver contained in these files provide you with an excellent opportunity to gain additional experience with constructors.

5.4.3.2 Follow-up hands-on activities for TESTITEM.CPP, ITEM.H and ITEM.CPP

□ The instructions in this activity read as though you should create all the code first and then test it, simply because it's much easier to read when it's described that way. As you know, this is not necessarily a good idea, even with the small amount of code required here. So, feel free, when you do what is asked, to do it one constructor at a time. In fact, that's the recommended approach.

To the class definition in ITEM.H and ITEM.CPP add the following two constructors:

- A default constructor that sets `numberInStock` to `0`, `warrantyAvailable` to `false`, and `price` to `0.0`.
- A constructor with three in-parameters that accepts a value for each of the private data members and initializes the `ItemType` class object with those values

Also add code to the driver in TESTITEM.CPP to test the two constructors that you have added to the class.

Then compile the revised TESTITEM.CPP and ITEM.CPP separately and link the resulting .OBJ files with PAUSE.OBJ to produce a new TESTITEM.EXE. Run this executable and make sure you get the output you expected.

◯ Instructor checkpoint 5.3

Module 6

Abstract data types and classes

6.1 Objectives

- To understand the notion of an *abstract data type* (often referred to simply as an *ADT*, which is pronounced by just saying the letters A–D–T).
- To understand how a C++ class can be used to implement an ADT.
- To understand the *principle of information hiding*, and in particular to understand that a knowledge of the *interface* to a class (i.e., the prototypes of the public member functions) is all that is needed to *use* the class.

  In other words, the client of a class does *not* need to know how a class has been implemented in order to declare and use objects of that class. On the other hand, neither does the designer or the implementor of a class need to know where or when or how or for what purpose a class will be actually used by the clients of the class. The public interface of the class is the only knowledge the client and the implementor have in common.
- To become familiar with the `Menu` class and `TextItems` class and know how to use each one in your programs, *without knowing the details of either class implementation.* (This, in some sense, is "what it's all about".)

6.2 List of associated files

- SHELL.CPP is a new shell starter program that incorporates both a `Menu` class that can be used to display menu options and get user choices, and a `TextItems` class that can be used, among other things, for on-line help.
- SHELL.DAT contains the text item data file for running the program in SHELL.CPP.

- MENU.H is the specification file for the `Menu` class.
- MENU.OBJ contains the implementation of the `Menu` class corresponding to MENU.H. (MENU.CPP is for your instructor's eyes only.)
- MENUDEMO.CPP is a demo program that illustrates in greater detail the features of the `Menu` class.
- TXITEMS.H is the specification file for the `TextItems` class.
- TXITEMS.OBJ contains the implementation of the `TextItems` class corresponding to TXITEMS.H. (Like MENU.CPP, TXITEMS.CPP is for your instructor's eyes only.)
- TXITDEMO.CPP is a demo program that illustrates in greater detail the features of the `TextItems` class
- TXITDEMO.DAT contains a sample data file needed for running the demo program in TXITDEMO.CPP.

6.3 Overview

First of all, a couple of disclaimers. There is a great deal more to learn about *abstract data types* than you will find here, but what we *do* show here will give you a good start. Also, many students start to get nervous when any topic has the word "abstract" in its title, so we should say at the outset that abstract data types are, for the most part, quite concrete.

In fact, you've already seen a few, since any C++ class may be regarded as an abstract data type. But let's back up a bit.

Suppose we are writing a program that needs for some reason to deal with the notion of "time". We may decide that a reasonable conceptual view of a particular time involves hours, minutes and seconds. The next question that arises is this: What would we like to be able to do with our time values? Well, we'd certainly like to be able to set the value to be whatever we want, display a value to see what it is, and perhaps we'd also like to be able to change the value, possibly by incrementing it one second at a time.

There may be many other things that we would like to do with our time values, but let's stop there for the sake of argument. If we do, and then ask ourselves how we are going to represent such time values in our programming language, the first thing that we will realize (probably, at least with most programming languages) is that there is no "time" data type built into the language. So, we must find a way to implement such a thing ourselves. Fortunately, as you already know, at least with this example, C++ provides a very convenient mechanism for doing just that, namely the `class`. And, in fact, we have already implemented this particular `Time` class.

And so it is with just about any other kind of "real-world entity" that we wish to represent in one of our programs. We need to begin by thinking about

what kind of data are needed to represent the various parts of that entity and follow this up with questions about what we want to do with the entity, or, equivalently, what are the operations that we wish to perform on the entity. It's a short stretch from here to realize that the same general notion, like time, might mean different things to different people or even different things to the same people at different times and/or in different situations. That, of course, is one of the beauties of abstract data types: Just what you mean by one depends on the application in which you intend to use it, and you can use the class mechanism to define it to be just what you want. Of course, one of the goals of software development is to design *reusable software*, so many software developers spend countless long and sleepless nights trying to develop and implement abstract data types that the rest of the world will find useful and spend big bucks to obtain.

Your humble author, for example, has produced two ADTs that untold legions of students before you have found immensely useful in their programs and we shall present them to you now without further ado.

The first is a `Menu` ADT, which has been implemented as a C++ class and is available for your use. Let's think about what a useful menu consists of, at least in a simple *console program* of the kind we have been discussing. Surely every menu should have a title, and a sequence of numbered options. And what would we like to do with such a thing? Not much, really: just display it, and then get a menu option from the user. There are details, of course, but from the perspective of a client (user) of the `Menu` class, that's about it.

The second is a `TextItems` ADT, which has also been implemented as a C++ class and is also available for your use. What exactly is this all about? Well, you know that almost any program that's even slightly beyond trivial usually involves the display of quite a bit of text for one reason or another. We can always accomplish this by placing enough `cout` statements in our code, but this is tedious and time consuming at best, and can be very frustrating when we have to change those statements in any way.

A much better alternative is to simply put all our "text items" (conceptually distinct pieces of text that we want displayed at various places throughout our program code) in one file, load the contents of that file into memory at the start of our program, and then display whichever text item we need at any time with a single call to a display function to which we pass an item identifier. Wow!

So what does this `TextItems` class consist of? Once again there are details, but from the client's perspective the "data" is just the file of the client's text items, and all the client ever needs to do is load those text items in from the file and then display whichever item is needed whenever it is needed.

The rest of this Module is taken up with showing you how these ADTs (classes) work. You will first look at a shell program that shows you the essential features of both, from a purely operational point of view. You will be able to use this shell as a starting point for many of your own subsequent programs. The rest of the Module just looks in more detail at the features of these two classes, always from the client perspective though, since you do not get to see the implementation of either class. (You may want to implement one or both later.)

By the way, using a class like `Menu` or `TextItems` without any access to the implementations is completely analogous to your use of what's in the C++ `iostream` library (for example), which you have been using for a long time, and (probably) doing so without even looking at the contents of that header file, let alone at what's in the corresponding implementation file.

6.4 Sample Programs and Other Files

6.4.1 SHELL.CPP and SHELL.DAT provide a shell program that uses both a Menu class and a TextItems class

```
// Filename: SHELL.CPP
// Purpose:  Provides a shell starter program utilizing both
//           a "menu" and on-line help or other "text items".

#include <iostream>
using namespace std;

#include "MENU.H"     // Header file for the Menu class
#include "TXITEMS.H"  // Header file for the TextItems class

int main()
{
    Menu m("Main Menu"); // Declare a Menu object "m" with title "Main Menu"
    m.AddOption("Quit"); // Then add four options to the menu "m" ...
    m.AddOption("Get information");
    m.AddOption("Do something");
    m.AddOption("Do something else");

    TextItems itemList("SHELL.DAT"); // Load text items into memory
    int menuChoice;

    do
    {
        m.Display();  // Display the menu
        menuChoice = m.Choice();  // Get the user's choice

        if (menuChoice == -1) // If user choice not OK ...
        {
            cout << "\nUser failed to make valid menu choice."
                 << "\nProgram now terminating.";
            return 1;  // Exit program, indicating unsuccessful termination.
        }

        switch (menuChoice) // Display text item depending on user's choice
        {
            case 1: /* Do nothing */;                              break;
            case 2: itemList.DisplayItem("Program Info");         break;
            case 3: itemList.DisplayItem("Doing Something");      break;
            case 4: itemList.DisplayItem("Doing Something Else"); break;
        }

    } while (menuChoice != 1); // Quit when user chooses first menu option

    return 0;
}
```

Contents of the file SHELL.DAT are shown between the heavy lines:

```
Program Info

This program provides a "shell starter program" that you can use as the
starting point for almost any "console" program.

By looking at the source code you can get at least a basic idea of how the
Menu class works.

By looking at the contents of the file SHELL.DAT and then looking at the
source code while the program runs, you can also get a very good idea of
how to use the TextItems class.

There are several important things to notice about the structure of the
file of "text items", and there is no room for error on these points when
you are constructing such a file:

1. The first line of each text item contains the "name of the item", i.e.,
   the exact string that you must supply as the argument to the DisplayItem
   function when you wish to display that item. This line is not displayed.

2. A line of 80 dashes (no more, no fewer) terminates a text item.

3. A line of 80 exclamation marks (no more, no fewer) determines a "pause
   point" where the program stops and waits for the user to press RETURN.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--------------------------------------------------------------------------------
Doing Something

This is just a text item saying that the program is doing something.

In an actual program the program would actually be doing something here
rather than just displaying this inane message ...

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--------------------------------------------------------------------------------
Doing Something Else

This is the second of two text items saying that the program is doing
something when it really isn't.

We have to beg your indulgence to let us get away with this ...

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--------------------------------------------------------------------------------
```

6.4.1.1 What you see for the first time in SHELL.CPP and SHELL.DAT

This program illustrates typical use of both the `Menu` class and the `TextItems` class. These are not part of the C++ Standard Library, but are supplied by the author.

Since the program is sprinkled liberally with informative comments about what is going on, you should read through them while studying the adjacent code and note that you see:

- How to declare a menu (the object `m` in the program) and provide a menu for the title at the same time

- How to add options to a menu once it has been declared
- How to set up a list of "text items" in memory (the `itemList` object in the program) so that each text item is readily available for display
- How to display any one of the text items by calling the `Display` member function and supplying the title of the item as in-parameter

You also need to know the precise format of a "file of text items", so you should also study SHELL.DAT and note, first, that it contains three text items, and second, that the first of those text items describes the required format, so we won't repeat it here.

6.4.1.2 Additional notes and discussion on SHELL.CPP and SHELL.DAT

This program and the two associated classes (`Menu` and `TextItems`) can be used as a starting point for virtually any console program that you wish to write, since any such program should have a menu of options and will have to display various "text items". To adapt it to your needs you have only to choose your menu title and options to place in the program in the obvious places and prepare your file of text items, whose name must replace SHELL.DAT in your copy of SHELL.CPP. And of course your switch statement will contain calls to functions that do whatever your program does, rather than just displaying text items as is done in SHELL.CPP.

Note as well that the two classes (`Menu` and `TextItems`) may be regarded as *utilities* in much the same way as the `Pause` function, so it might be a good idea to place MENU.H and TXITEMS.H in the same location you are storing PAUSE.H and to place MENU.OBJ and TXITEMS.OBJ in the same location you are storing PAUSE.OBJ.

6.4.1.3 Follow-up hands-on activities for SHELL.CPP and SHELL.DAT

□ Copy the files SHELL.CPP and SHELL.DAT, then study and test the program in SHELL.CPP. You will have to link (in the usual way for your system) the code of SHELL.OBJ with the code of MENU.OBJ and TXITEMS.OBJ (and of course SHELL.CPP needs to include MENU.H and TXITEMS.H when it compiles).

□ This activity is designed to convince you that it is possible to develop programs much more rapidly when much of the "busy work" can be handled with minimal effort by the menuing and text-display "infrastructure" provided by the `Menu` and `TextItems` classes.

Your goal is to design and write a "Simple Calculator" program that allows a user to either add or multiply a line of integers. The program is to have a menu with the following title and options (don't worry about the spacing and placement of the title and options, just accept where the `Display` function of the `Menu` class places them on the screen, since you have no control over these factors in any case):

```
Simple Calculator Menu

  1. Quit
  2. Get Information
  3. Add
  4. Multiply
```

When option 3 (or 4) is chosen, the user must be prompted to enter some integers on a line, which will then be added (or multiplied) and the result displayed, after which the user will be returned to the menu.

What you should notice when developing this program is that you can spend your time concentrating on the "meat and potatoes" part of the development, i.e., the details of options 3 and 4, because the overall structure of the program is already in place, provided you start with SHELL.CPP.

So, make a copy of SHELL.CPP and call it MYCALC.CPP. Then make the appropriate revisions to this copy so that the above menu will be displayed. Also prepare a suitable and correctly formatted file of text items called MYCALC.DAT and replace the line

```
TextItems itemList("SHELL.DAT"); // Load text items into memory
```

with the line

```
TextItems itemList("MYCALC.DAT"); // Load text items into memory
```

in MYCALC.CPP. This file should contain at least three text items:

- One describing the program

  This item should be displayed in response to the user choosing menu option 2.

- A brief reminder to the user of what to do that when the user chooses menu option 3.

  This item should be displayed in response to the user choosing menu option 3.

- A brief reminder to the user of what to do that when the user chooses menu option 4.

  This item should be displayed in response to the user choosing menu option 4.

Each of the last two items in the above list can also include the user prompt to enter values when one of those options has been chosen.

Be sure to test your program until it is working properly. You will have to combine the code in MYCALC.CPP with the code for the `Menu` and `TextItems` classes in the same way that you did for SHELL.CPP in the previous activity.

◯ Instructor checkpoint 6.1

6.4.2 MENU.H and MENU.OBJ provide a Menu class

```
// Filename: MENU.H
// Purpose:  Specification file for a Menu class.

/************************************************************************
This Menu class provides a reasonably useful menuing facility for simple
text-based I/O.  Menus have a title and up to nine (numbered) options
separated from the title, when displayed, by a blank line.  The only
reason the number of options is limited to nine is to avoid the minor
irritant of dealing with two-digit option numbers as opposed to single-
digit option numbers.  Each menu knows what its range of valid options
is, and can get from the user a valid choice in the appropriate range or
return an (error) value of -1 if the user fails to enter a valid option
number during three attempts.
************************************************************************/

#ifndef MENU_H
#define MENU_H

class Menu
{
public:

    Menu();
    // Default constructor, which initializes a "blank menu" which,
    // when displayed, simply says that it is a blank menu.
    // Pre:  none
    // Post: numberOfOptions == 0, and first row of "text" contains
    //       "This is currently a blank menu ... "


    Menu(const char title[]);
    // Constructor, which initializes a menu with the title passed in
    // but no options.
    // Pre:  "title" has been initialized.
    // Post: numberOfOptions == 0 and the contents of "title" have
    //       been copied into the first row of "text".


    void AddTitle(const char title[]);
    // Adds a title to the menu (or changes the current title).
    // Pre:  "title" has been initialized.
    // Post: The contents of "title" have been copied into the
    //       first row of "text".


    void AddOption(const char option[]);
    // Adds a new option to the menu and gives it the next available
    // number, unless the menu is "full" (i.e., already contains nine
    // options), in which case an error message has been displayed,
    // followed by the pause message.
    // Pre:  numberOfOptions <= 9, and "option" has been initialized.
    // Post: If numberOfOptions < 9 its value has been incremented by 1.
    //       The resulting value of numberOfOptions, a period, and
    //       and a space have been added to the front of the "option"
    //       string, and this final string has been copied into
    //       text[numberOfOptions].  If numberOfOptions is 9, then
    //       the following two-line message is displayed:
    //       Maximum number of menu options exceeded.
    //       Option: <<text_of_option_goes_here>> not added.
```

```
    void Display() const;
    // Displays the menu on the screen.
    // Pre:  The menu has been initialized.
    // Post: The menu has been displayed on the screen, more or less
    //       centered left-to-right and top-to-bottom.  More specifically,
    //       there are two more blank lines at the bottom of the screen
    //       than at the top, and there are two more blank columns on
    //       the right than on the left, giving a slight top-left bias.
    //       A blank menu, or a menu with just a title, however, is
    //       displayed left-justified with only the bias toward the top.


    int Choice() const;
    // Returns a menu choice entered by the user, or a value of -1
    // if the user does not enter a valid menu choice for the current
    // menu during one of three permitted attempts, or if the user
    // attempts to get a choice from a blank menu.
    // Pre:  A menu has been initialized and displayed.
    // Post: Either a valid menu choice for the menu has been returned
    //       or -1 has been returned and both an error message and the
    //       pause message have been displayed.  If three unsuccessful
    //       attempts have been made the displayed message is:
    //       Sorry, but you only get three chances ...
    //       If a choice was sought from a blank menu, the message is:
    //       Menu has no options from which to choose ...


private:

    int numberOfOptions;
    // Number of options currently on the menu.

    char text[1+9][81];  // Includes 1 title row and 9 option rows.
    // Array of strings to hold the menu text, the title in the first
    // row, followed by up to nine options in the remaining nine rows.
};

#endif
```

6.4.2.1 What you see for the first time in MENU.H

This specification file shows you both the public interface member functions and the private data members of the `Menu` class. The fact that the private data members are publicly visible (if not publicly accessible) is a feature of how things are done in C++. It is possible to "hide" more of this information more effectively than we have done here, but it is usually not worth the trouble to do so.

So, even though you can "see" the private data members of the `Menu` class, your "attitude" should be that you have no interest in them, and if they were in fact hidden it would make no difference to you, since your only interest (as a *client* of the `Menu` class) is in *what* the class can do for you, *not* in *how* it does this.

The member functions of the public interface, on the other hand, are of great interest to you as a client/user of the class. There are two constructors and four other member functions in the class interface. In SHELL.CPP you have

already seen the non-default constructor, as well as the `AddOption`, `Display`, and `Choice` functions at work, and in fact these are really all you need use the class in most situations.

There is, of course, a default constructor, which sets up a blank menu (no options *and* no title), which requires a way to add a title so we also have an `AddTitle` member function.

You should read carefully the comments and pre/post-conditions for each member function, just to have a more complete understanding of how each works.

Finally, you should note the parameter list in the non-default constructor and the `AddTitle` and `AddOption` member functions. In each case it has the form

```
const char parameterName[]
```

You will see what all this means in Module 13, but for now all you need to know (as, once again, you have already seen in SHELL.CPP) is that this kind of formal parameter allows you to supply as an actual parameter any quoted string, such as `"Main Menu"` or `"Get Information"` or `"Go Fly a Kite!"`.

6.4.2.2 Additional notes and discussion on MENU.H

Since the combination of MENU.H and MENU.OBJ provide a menuing utility class that can be used in any console program, these two files should be placed in locations where they are easily accessible to any of your programs that use them (the same location(s) as PAUSE.H and PAUSE.OBJ, for example).

6.4.2.3 Follow-up hands-on activities for MENU.H

□ Copy MENU.H and study the `Menu` class specification.

□ Place the files MENU.H and MENU.OBJ in readily accessible locations, if you have not already done so.

◯ Instructor checkpoint 6.2

6.4.3 MENUDEMO.CPP is a demo program that shows more features of the Menu class

```
// Filename: MENUDEMO.CPP
// Purpose:  A test driver for the Menu class.

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

#include "MENU.H"
#include "PAUSE.H"


int main()
{
    // Declare a first Menu object.  The default constructor will
    // be invoked to create a "blank" menu.
    Menu m1;

    // Show what a menu with no title and no options
    // (i.e. a "default menu) looks like when it is displayed:
    m1.Display();
    Pause(0);

    // Add a title to the blank menu and then re-display the menu:
    m1.AddTitle("New Menu");
    m1.Display();
    Pause(0);

    // Notice what happens when we attempt to get a menu choice from
    // a blank or "title-only" (which is what we have here) menu.
    m1.Display();
    int menuChoice = m1.Choice();
    cout << "The menu choice obtained was " << menuChoice << "." << endl;
    Pause(0);



    // Declare a second menu object and initialize it with a title, then
    // add a number of options, and display the menu after each step.
    // Notice how the menu remains "centered".
    Menu m2("Main Menu");
    m2.Display();
    Pause(0);
    m2.AddOption("Quit");
    m2.Display();
    Pause(0);
    m2.AddOption("Get information");
    m2.Display();
    Pause(0);
    m2.AddOption("Do something");
    m2.Display();
    Pause(0);
    m2.AddOption("Do something really, really long and involved");
    m2.Display();


    // Loop while the user does not choose the Quit option, i.e.
    // while the user does not choose option 1 from the menu.
    bool finished = false;
```

```
    while (!finished)
    {
        m2.Display();
        menuChoice = m2.Choice();
        finished = (menuChoice == 1  ||  menuChoice == -1);
        cout << endl
             << "The menu choice returned was " << menuChoice << "."
             << endl;
        Pause(0);
    }



    // Create a third menu with nine options (the maximum number)
    Menu m3("Test Menu");
    m3.AddOption("Option 1");
    m3.AddOption("Option 2");
    m3.AddOption("Option 3");
    m3.AddOption("Option 4");
    m3.AddOption("Option 5");
    m3.AddOption("Option 6");
    m3.AddOption("Option 7");
    m3.AddOption("Option 8");
    m3.AddOption("Option 9");
    m3.Display();
    Pause(0);

    // Now note what happens when you attempt to add another option.
    m3.AddOption("And yet another option ... ");

    cout << endl;

    return 0;
}
```

6.4.3.1 Notes and discussion on MENUDEMO.CPP

This program is designed to give you more insight into the workings of the `Menu` class and you should study the source code carefully while running the program.

6.4.3.2 Follow-up hands-on activities for MENUDEMO.CPP

□ Copy, study and test the program in MENUDEMO.CPP. You will have to combine, in the usual way for your system, the code in this program with the code for both the `Menu` class and the `Pause` function.

6.4.4 TXITEMS.H and TXITEMS.OBJ provide a TextItems ADT

```
// Filename: TXITEMS.H
// Purpose:  Specification file for the TextItems class.
//
// This class provides a structure to hold in memory any number of
// "items of text", and display any one of them on the screen as needed.
// One obvious use would be for on-line help, but *any* text item of any
// size is a candidate for storage in the "text items" file and retrieval
// from that file for display.
//
// The items must be read in from a textfile and that textfile must be
// formatted in the following way:
//
// - The first line of a "text item" must be its "title", i.e., the exact
//   string that will be used to access that particular text item.
//
// - Every text item is terminated by a line of 80 dashes (hyphens)
//   which is *not* regarded as part of the item itself.
//
// - Any line of 80 exclamation marks (!) within a text item
//   indicates a point where the display must pause and ask the
//   user to press ENTER to continue.


#ifndef TXTITEMS_H
#define TXTITEMS_H


struct ItemNode;
typedef ItemNode* ItemPointer;
typedef char MyString80[81];


class TextItems
{
public:

    TextItems();
    // Default constructor
    // Pre:  none
    // Post: The list of text items has been intialized and is empty.


    TextItems(/* in */ const MyString80 fileName);
    // Constructor
    // Pre:  "fileName" contains the name (or the full pathname) of
    //       a file of properly formatted text items.
    // Post: All text items in "fileName" have been read into memory.
    //       "itemList" contains the list of text items.
    //       If the file supplied as the parameter in the declaration
    //       of the object is empty, the program displays this message:
    //       Error: Input file of text items is empty,
    //              Program is now terminating.
    //       If the file doesn't exist, the program displays this message:
    //       Error: Input file of text items does not exist.
    //              Program is now terminating.
    //       In either case, as the messages suggest, the program ends.
```

```
    void DisplayItem(/* in */ const MyString80 title) const;
    // Displays a text item on the screen.
    // Pre:  A string value has been assigned to "title".
    // Post: The text item identified by "title" has been displayed.
    //       If the item designated by "title" cannot be found, the
    //       following message is displayed: <<title>> not found.
    //       If the user attempts to display an item and there are
    //       no items (because the TextItems object was declared
    //       without the file parameter) the program displays the
    //       following message and then ends:
    //       Error: No text items available for display.
    //       A filename must be supplied as a constructor
    //       parameter when declaring a TextItems object.
    //       Program is now terminating.

private:

    ItemPointer itemList;
};

#endif
```

6.4.4.1 What you see for the first time in TXITEMS.H

As was the case for MENU.H, in TXITEMS.H you are interested only in the interface, and for such a useful class the interface is remarkably simple,[1] consisting of just the two constructors and a `DisplayItem` function. You have already seen the non-default constructor and the `DisplayItem` function in action in SHELL.CPP. Each one has a single in-parameter, namely a quoted string such as `"MYITEMS.DAT"` or `"My First Item"`. There is also a default constructor, in case the client forgets to supply the name of a file containing text items.

6.4.4.2 Additional notes and discussion on TXITEMS.H

The files TXITEMS.H and TXITEMS.OBJ should be placed in the same location(s) as the files MENU.H and MENU.OBJ, and for the same reasons.

6.4.4.3 Follow-up hands-on activities for TXITEMS.H

☐ Copy TXITEMS.H and study the `TextItems` class specification.

☐ Place the files TXITEMS.H and TXITEMS.OBJ in readily accessible locations, if you have not already done so.

[1] Actually, there is something missing from the interface to this class which really ought to be there, since this class makes use of *dynamic data storage*. The missing item is a *class destructor*. But more on all of this later, in Module 18.

6.4.5 TXITDEMO.CPP and TXITDEMO.DAT provide a driver and sample data file to show more features of the TextItems class

```
// Filename: TXITDEMO.CPP
// Purpose:  A test driver for the TextItems class.

#include <iostream>
using namespace std;

#include "TXITEMS.H"

int main()
{
    cout << "\nThis program demonstrates use of the "
         << "TextItems class to display \"text items\". "
         << "\nBe sure that you know what a file of "
         << "such items looks like, and that you have "
         << "\nat least one available, before you attempt "
         << "to use this program. ";

    MyString80 fileName, title;

    cout << "\n\nEnter full name of file containing text "
         << "items, or end-of-file to quit program: \n";
    cin.getline(fileName, 82);
    while (cin)
    {
        TextItems itemList(fileName);

        cout << "Enter title of text item to display, "
             << "or end-of-file to quit displaying items: \n";
        cin.getline(title, 82);
        while (cin)
        {
            itemList.DisplayItem(title);
            cout << "Enter title of text item to display, "
                 << "or end-of-file to quit displaying items: \n";
            cin.getline(title, 82);
        }

        cin.clear();
        cout << "Enter full name of file containing text "
             << "items, or end-of-file to quit program: \n";
        cin.getline(fileName, 82);
        cout << endl;
    }

    return 0;
}
```

Contents of the file TXITDEMO.DAT are shown between the heavy lines:

```
First Item
This is the first text item in the test file called TXITDEMO.DAT.
The previous line was the first line of the item.
Here are three blank lines.



And here is a pause in the middle of the item ...
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Now there is another blank line and then we pause again at the end.

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--------------------------------------------------------------------------------
Second Item
1 One ...
2 Two ...
3 Three ...
4 Four ...
5 Five ...
This is the second text item.
There are five numbered lines at the beginning of the item.

          These two lines follow a blank line
          and are indented 10 spaces.

This item does not contain a pause.
This is its last line.
--------------------------------------------------------------------------------
Third Item
This third text item contains just this line and a pause.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--------------------------------------------------------------------------------
Fourth Item
This is the 4th and last item, which contains one full screen of text.
Line 2
Line 3
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
Line 13
Line 14
Line 15
Line 16
Line 17
Line 18
Line 19
Line 20
Line 21
Line 22
And, of course, a pause ...
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--------------------------------------------------------------------------------
```

6.4.5.1 Notes and discussion on TXITDEMO.CPP and TXITDEMO.DAT

Like MENUDEMO.CPP, this program is designed to give you more insight into the workings of the `TextItems` class. It permits you to read in any file of properly formatted text items and display them in any order as many times as you like, and to do this for as many files as you like. As usual, you should study the source code while running the program, as well as the file of text items TXITDEMO.DAT itself, so that you understand the format required for that file.

The use of `cin.getline(..., 82)` to read strings into variables of type `MyString80` should simply be taken on faith at this point. How this works and why the "magic number" 82 is necessary should become clear when Module 13 is covered.

6.4.5.2 Follow-up hands-on activities for TXITDEMO.CPP and TXITDEMO.DAT

□ Copy, study and test the program in TXITDEMO.CPP, which uses the file of text items TXITDEMO.DAT. You will have to link, in the usual way for your system, the code in this driver program with the code for the `TextItems` class in TXITEMS.OBJ and TXITDEMO.CPP will have to include TXITEMS.H when it compiles.

◯ Instructor checkpoint 6.3

Module 7

Member functions, non-member functions and friend functions

7.1 Objectives

- To understand what a *friend function* is and how to define and use one.
- To get some appreciation for the situations in which you need to make an appropriate choice from the following list when you are implementing a class and trying to decide how a particular task should be performed:
  - member function
  - non-member function
  - *friend function*
- To understand how *right associativity* of the assignment operator can help to write more concise code when many variables are assigned the same value.

New C++ reserved word
`friend`

7.2 List of associated files

- TESTIME3.CPP is a test driver for the `Time` class defined in TIME3.H and TIME3.CPP.
- TIME3.H is the specification file for the third version of the `Time` class and contains *accessor functions* (member functions that allow a client of the class to have read access to one or more private data members) and the prototype of a friend function.
- TIME3.CPP is the implementation file corresponding to TIME3.H.

7.3 Overview

This Module deals with the "high-level" questions you need to ask and answer when you are designing a class (i.e., an ADT), have decided what your data and operations are going to be, and have reached the point where you are implementing those operations: How, exactly, am I going to implement each operation?

Even in the few simple classes we have encountered so far (`Time`, for example), we have seen both member functions (`Increment`, say) and non-member functions (`IncrementedTime`) for doing various things with objects of the class.

So, the obvious question arises: How is the decision made as to which operations get implemented as member functions and which do not? Unfortunately, there are no rules that give us all the answers we would like, but there are some helpful guidelines and it is those guidelines that we discuss here.

First, let's note that it *is* the class designer who has control over these choices. Clearly, at one extreme, the class designer could place in the class interface an operation for every conceivable action the client might want to take with the class. Or, the designer could take a very minimalist approach and limit class clients to just a few basic operations in the interface. Practically, most class designers will try to take some middle ground, guided by personal experience and the experience of others.

The kinds of questions the class designer has to think about include:

- What will *every* user of the class need to be able to do with *any* single object of the class?

  The usual guideline here is that such operations should be implemented as member functions of the class.

- Are there some operations that a significant number of users (but not all) might want to perform on one or more objects and that would require such users to know some or all values of the private data members?

  The guideline here is that, instead of trying to second-guess these users and provide all such operations in the class interface, the class designer might generally be better advised to simply provide the necessary *accessor functions* (member functions that return values of the private data members) in the class interface and let each group of users write their own (non-member) functions for their own purposes on an as-needed basis, with the accessor functions and other member functions providing (it is hoped) the user with whatever access to class objects that the user needs. This keeps the class interface smaller (and therefore simpler).

  Especially if the (non-absolutely-essential) action to be performed involves two or more objects of the class, a non-member function is the better alternative. That non-member function can be written by the client using accessor and other member functions. Another possibility is that such a function can be implemented by the class designer as a *friend function*, which we deal with next.

- A *friend function* is technically a non-member function, but it is granted special privileges by a class (because it is a "friend" of the class, if you like). The special privileges that a friend function of a class has, and that an ordinary non-member function does *not* have, is direct access to the private data members[1] of the class. For example, an ordinary non-member function might have only read access to the `hours` private member of the `Time` class via a call like `t.Hours()` (where `Hours()` is a call to an `Hours` accessor function provided in the class interface), while a friend function has "full direct access" in the sense that the use of `t.hours` within the body of the friend function would not cause a compile-time error like it would in the body of an ordinary non-member function.

  The guidelines for when a non-member function should be a friend of a class are not easy to state concisely, but let's make a few relevant comments. Some authors will say to avoid[2] friend functions where possible. Sometimes, however, functions do need to be implemented as friends, as we will see in Module 8. Also, although efficiency is not something we have paid much attention to, sometimes it may be better to have a function as a friend to avoid the extra overhead of accessor function calls.

Another brief aside: As long and we are classifying and naming things, let's mention that a member function which "observes" something about its object (such as EqualTo1 in TIME3.H/TIME3.CPP, which observes whether itself is equal to some other Time value) is sometimes called (not surprisingly) an *observer function*.

[1] All comments about friends having access to your private parts will be studiously avoided.

[2] These authors will often give as their reason that, like some human friends, some friend functions are more trouble than they're worth.

7.4 Sample Programs and Other Files

7.4.1 TESTIME3.CPP is a test driver for the Time class in TIME3.H and TIME3.CPP

```
// Filenme: TESTIME3.CPP
// Purpose: Uses the Time class to compare member, non-member and
//          friend functions.


#include <iostream>
using namespace std;


#include "TIME3.H"
#include "PAUSE.H"


bool EqualTo2(/* in */ const Time& t1,
              /* in */ const Time& t2);


int main()
{
    cout << endl
         << "This program uses the Time class to "
         << "compare member, non-member and friend "   << endl
         << "functions.  You should study the source "
         << "code at the same time as the output. "    << endl << endl;

    Time t1(11, 59, 50);
    Time t2(12, 0, 0);

    t1.Display();  cout << endl;
    t2.Display();  cout << endl;
    Pause(0);

    if (t1.EqualTo1(t2))
        cout << "The two times are equal."     << endl;
    else
        cout << "The two times are not equal." << endl;
    Pause(0);

    while (!EqualTo2(t1, t2))
    {
        t1.Display();  cout << endl;
        t1.Increment();
    }
    Pause(0);

    if (EqualTo3(t1, t2))
        cout << "The two times are equal."     << endl;
    else
        cout << "The two times are not equal." << endl;

    return 0;
}
```

```
// Note that the following function is just an "ordinary"
// function (not a member function and not a friend function)
// and hence it must use the "accessor" functions to obtain the
// values of the private data members.
bool EqualTo2(/* in */ const Time& t1,
              /* in */ const Time& t2)
// Pre:  t1 and t2 have been initialized.
// Post: Returns true if t1 and t2 are the same Time value
//       and false otherwise.
{
    return (t1.Hours()   == t2.Hours()    &&
            t1.Minutes() == t2.Minutes()  &&
            t1.Seconds() == t2.Seconds());
}
```

7.4.1.1 What you see for the first time in TESTIME3.CPP

Direct your attention to the three different versions of the "`EqualTo`" function you see in this program. Each of these functions is a simple boolean function that tests whether two `Time` values are the same, but each is implemented in a different way to illustrate the three different possibilities that we are highlighting in this Module. Here they are:

- `EqualTo1` is implemented as a public member function of the `Time` class.

  A call to this function therefore looks like `t1.EqualTo1(t2)`, in which the member function `EqualTo1` is being applied to the `Time` object `t1` and is passed the second `Time` value `t2` as an in-parameter.

- `EqualTo2` is an "ordinary" non-member function whose prototype *and* full definition both appear in the driver file along with `main`.

  A call to this function looks like `EqualTo2(t1, t2)`, with *both* `Time` values `t1` and `t2` being passed as in-parameters.

- `EqualTo3` is also a non-member function, but one with special privileges relative to the `Time` class (direct access to its private members) which make it a *friend* of that class.

  A call to this function also looks like `EqualTo3(t1, t2)`, again with *both* `Time` values `t1` and `t2` being passed as parameters, so the difference between this function and `EqualTo2` will be seen in their implementations. However, do note that neither the prototype nor the function definition for `EqualTo3` is present in the driver file.

7.4.1.2 Additional notes and discussion on TESTIME3.CPP

The main thing that this program is designed to get you thinking about is the distinction between *member functions*, *non-member functions*, and the new notion of *friend functions*. The difficulty you have to grapple with as a class designer is that it is often possible to implement a particular operation in any one of these three ways.

In the discussion of the **Overview** section of this Module we have talked about some of guidelines for choosing between the three possible implementations for a given operation, but in this program you should just try to get a grip on how each one appears from a syntactical and a physical placement point of view, the "look and feel" of each one as it were. Study how each variation is used in the program.

The particular function we have chosen to work with here is a boolean function, i.e., a value-returning function whose return-value is of type `bool`, but there's nothing special about this. Analogous remarks hold for any other kind of value-returning function, and also for void functions.

7.4.1.3 Follow-up hands-on activities for TESTIME3.CPP

□ Copy and study the program in TESTIME3.CPP. You may run and test the program now if you wish, or wait until you have studied the revised version of the `Time` class in TIME3.H and TIME3.CPP.

How's your comfort level with multi-file programs?

As time goes on, we will be giving fewer details in the hands-on activities concerning the precise steps that you need to perform in order to run a program when you are dealing with a multi-file program. The reason we are going to do this, of course, is that by now you should be getting comfortable with those details on your system.

But, in this activity for example, if someone just handed you TESTIME3.CPP and said, "Run this program!", what would you have to do? Well, first of all you would have to make sure that the two "programmer include files" TIME3.H and PAUSE.H were available when you compiled TESTIME3.CPP, and that the corresponding .OBJ files TIME3.OBJ and PAUSE.OBJ were also available at link time to be linked with your TESTIME3.OBJ. Keep in mind that these file names might not be quite right on your system, and/or the steps "compiling" and "linking" may or may not be distinct and other details may have to be taken care of, but the point is that by now you know in general how all of this goes on your system, and making sure that it goes properly in any given instance then becomes a matter of detail. It is this "comfort level" that we assume you are fast approaching, if you have not already reached it.

7.4.2 TIME3.H and TIME3.CPP extend the Time class by adding accessor, observer and friend functions

```
// Filename: TIME3.H
// Purpose:  Specification file corresponding to TIME3.CPP. This version
//           adds three accessor functions and a friend function.

#ifndef TIME3_H
#define TIME3_H

class Time
{
    // The following function is a "friend" function of the Time
    // class and has direct access to its private data members.
    friend bool EqualTo3(const Time& t1, const Time& t2);
    // Pre:  t1 and t2 have been initialized.
    // Post: Returns true if t1 and t2 are the same time
    //       and false otherwise.

public:

    // The following five "interface functions" are exactly the
    // same as those in TIME2.H.  For brevity we have omitted the
    // comments and pre/post-conditions, which are also the same.
    Time();
    Time(int hoursInitial, int minutesInitial, int secondsInitial);
    void Set(int hoursNew, int minutesNew, int secondsNew);
    void Increment();
    void Display() const;


    // The following functions are called "accessor functions"
    // and allow a client/user to get the value of the data members.
    // But note that the client/user still has no direct access
    // to the data members, only the access provided.

    int Hours() const;
    // Pre:  none
    // Post: Value of data member "hours" is returned.

    int Minutes() const;
    // Pre:  none
    // Post: Value of data member "minutes" is returned.

    int Seconds() const;
    // Pre:  none
    // Post: Value of data member "seconds" is returned.


    // The following function is another member function.
    bool EqualTo1(const Time& otherTime) const;
    // Pre:  self and otherTime have been initialized.
    // Post: Returns true if self and otherTime are the same time
    //       and false otherwise.


private:
    int hours;
    int minutes;
    int seconds;
};

#endif
```

```
// Filename: TIME3.CPP
// Purpose:  Implementation file corresponding to TIME3.H. This version
//           adds three accessor functions and a friend function.

#include <iostream>
using namespace std;

#include "TIME3.H"

// Private data members of the Time class:
//     int hours;
//     int minutes;
//     int seconds;


// The following five "interface functions" are exactly the same
// as those in TIME2.CPP.  For brevity we have omitted the pre-
// and post-conditions, which are also the same. These functions
// are also implemented in exactly the same way as in TIME2.CPP,
// except for the default constructor, which should be compared
// with that in TIME2.CPP.

//*****************************************************************
Time::Time() { hours = minutes = seconds = 0; }


//*****************************************************************
Time::Time(int hoursInitial, int minutesInitial, int secondsInitial)
{
    hours = hoursInitial;
    minutes = minutesInitial;
    seconds = secondsInitial;
}


//*****************************************************************
void Time::Set(int hoursNew, int minutesNew, int secondsNew)
{
    hours = hoursNew;
    minutes = minutesNew;
    seconds = secondsNew;
}


//*****************************************************************
void Time::Increment()
{
    seconds++;
    if (seconds > 59)
    {
        seconds = 0;
        minutes++;
        if (minutes > 59)
        {
            minutes = 0;
            hours++;
            if (hours > 23)
                hours = 0;
        }
    }
}
```

```
//*****************************************************************
void Time::Display() const
{
    if (hours < 10)
        cout << '0';
    cout << hours << ':';
    if (minutes < 10)
        cout << '0';
    cout << minutes << ':';
    if (seconds < 10)
        cout << '0';
    cout << seconds;
}


// The following three functions are "accessor functions" and provide
// read access to the private data members of the class.

//*****************************************************************
int Time::Hours() const
// Pre:  none
// Post: Value of data member "hours" is returned.
{
    return hours;
}


//*****************************************************************
int Time::Minutes() const
// Pre:  none
// Post: Value of data member "minutes" is returned.
{
    return minutes;
}


//*****************************************************************
int Time::Seconds() const
// Pre:  none
// Post: Value of data member "seconds" is returned.
{
    return seconds;
}



// The following function is a member function. Note the Time::
// preceding the function name and note that as a member function
// it has direct access to the private data members.
bool Time::EqualTo1(const Time& otherTime) const
// Pre:  self and otherTime have been initialized.
// Post: Returns true if self and otherTime are the same time
//       and false otherwise.
{
    return (hours   == otherTime.hours    &&
            minutes == otherTime.minutes  &&
            seconds == otherTime.seconds);
}
```

```
// The following function is *not* a member function (note the
// *absence* of Time:: preceding the function name). But note
// that it nevertheless has direct access to the private data
// members of the class because it is a "friend" of the class.
bool EqualTo3(/* in */ const Time& t1,
              /* in */ const Time& t2)
// Pre:  "t1" and "t2" have been initialized.
// Post: Returns true if t1 and t2 are the same time
//       and false otherwise.
{
    return (t1.hours   == t2.hours    &&
            t1.minutes == t2.minutes  &&
            t1.seconds == t2.seconds);
}
```

7.4.2.1 What you see for the first time in TIME3.H and TIME3.CPP

The first thing to note about these two files is that both of them have been condensed from what their "normal" size would be by reducing the prototypes and function definitions we have already looked at to their bare minimum, omitting the associated comments and pre/post-conditions.

Another small space saving is achieved by replacing the three assignment statements on three separate lines in the original version of the default constructor by the single statement

```
hours = minutes = seconds = 0;
```

which makes use of what is called the *right associativity* of the assignment operator. This means that the assignment operator "associates" from right to left, or, equivalently, that this statement is equivalent to

```
(hours = (minutes = (seconds = 0)));
```

which, in turn, means that the statement is evaluated from right to left as follows: First, the value 0 is assigned to `seconds`, and the value returned from this operation is 0 (the value of the assignment) which is then assigned to `minutes`. The value returned by *that* operation is again 0, which is, finally, assigned to `hours`.

The above item is really an aside at this point, but a useful technique whenever several variables all have to be assigned or initialized to have the same value. It will come up again somewhat later when we overload the assignment operator in Module 19.

As for what's new and exciting in the context of the subject matter of this Module, we have

- Three *accessor functions* (also called *reader functions*): `Hours`, `Minutes` and `Seconds` (Note that the names of the functions are capitalized, to distinguish them from the names of the corresponding private data members, whose values they return and whose names are *not* capitalized.)

  Actually, we asked *you* to implement these functions yourself as long ago as Module 2, but now we formally add them to the `Time` class interface ourselves. A client/user of the class can (in fact *must*) use these member functions to obtain any value(s) of the private data members, if the client

needs them for any reason, since direct access to private data members by clients is prohibited.

- The prototype and definition of the member function `EqualTo1`

  Note, in the body of this function, that the "unqualified" `hours`, `minutes` and `seconds` refer to "self", i.e., to the private data members of the object to which the `EqualTo1` function is being applied, while the same data members preceded by "`otherTime.`" refer to the private data members of the `Time` object that has been passed to `EqualTo1` as an in-parameter. By the way, this is an illustration of the fact that *any member function of a class* has direct access to the private data members of *any object* of that class (i.e., not only to the data members of the object to which the function is being applied).

  A function like `EqualTo1` is sometimes called an observer function, since it "observes" and reports on the status of something, in this case whether or not the value of its own `Time` value ("self") is equal to some other `Time` value.

- The prototype and definition of the *friend function* `EqualTo3` (Remember that `EqualTo2`, unlike `EqualTo3`, was completely contained in the driver file, TESTIME3.CPP.)

  Note that a function cannot decide to be a friend of some class of its own accord, for obvious reasons. That is, "friendship" is something that must be *granted by the class* to a non-member function. As you can see from TIME3.H, this is done by placing the prototype of the function that is to be a friend (`EqualTo3`, in this case) in the specification file and preceding it with the qualifying keyword `friend`, which is another of the reserved words in C++.

  There is no universally followed convention for the placement of friend function prototypes within the specification file, so you will see them in different places if you look at different C++ texts. Our convention, shown in TIME3.H, is to place them (just one in this case) *before* both the public *and* private sections of the specification file to emphasize that friend functions are *neither* public *nor* private[3] (they are, recall, *non-member*).

  A typical location for a friend function definition is in the implementation file for the class of which the function is a friend, but there is no requirement that this be the case.

  Note, in the function definition for `EqualTo3`, in TIME3.CPP, that the friend qualifier that appeared before the function prototype in the specification file is *not* repeated before the function header in the implementation file.

[3] Even putting a friend function in the `private` section of a class definition will not make it private.

7.4.2.2 Additional notes and discussion on TIME3.H and TIME3.CPP

After considering the three options for implementing a function like the `EqualTo` function we have seen in this version of the `Time` class, we are still left with the question: Which one is "best"? And, like many questions in the real world, the answer is not clear cut, in that not all programmers will agree on which one to use.

However, our general guidelines would indicate that making the `EqualTo` function a member function is not the best choice, since the function really deals with *two* `Time` values, and in fact deals with them in a "symmetric" way (i.e., two `Time` values are the same, or not, irrespective of the order in which you look at them).

So, if a member function is not the way to go, how to choose between just an ordinary non-member function and a friend function? First of all, letting the class user write the non-member function (as in `EqualTo2`) is one possibility, but this is only possible if there are accessor functions (as there are in this case) and only feasible if the extra overhead of the function calls does not slow down the program to the point where the users complain (not likely to be a problem in this case). This might therefore be the preferred one of the three options in this case. If it turns out that efficiency does become a problem, the next "maintenance round" might require implementing the friend option which gives direct (and hence faster) access to the private data members.

But, as we shall see in Module 8, there is an even better solution in the cases (like this one) where a function (the various versions of `EqualTo` in the current context) is really taking the place of an operator (the equality-testing operator `==` in this case, which is used for testing equality of two values in C++). This "better solution" is to "overload" the operator `==` so that this operator can be used to test the equality of two `Time` values in the same way that it can be used to test the equality of two `int` values (for example).

7.4.2.3 Follow-up hands-on activities for TESTIME3.CPP, TIME3.H and TIME3.CPP

□ Copy TIME3.H and TIME3.CPP and study this version of the `Time` class. Then compile this class code separately, link with TESTIME3.OBJ and PAUSE.OBJ, and test the resulting driver program in TESTIME3.EXE, if you have not already done so.

□ Make copies of TESTIME3.CPP, TIME3.H and TIME3.CPP for testing purposes and call them, respectively, TT3.CPP, T3.H and T3.CPP.

□ In TT3.CPP, in the body of the function `EqualTo2`, change each "`Hours()`" to "`hours`" and then try to compile TT3.CPP. You will get an access error that proves `EqualTo2` cannot access the private data members of `Time` directly. Remove this change before proceeding.

□ Design and write three new functions—`EarlierThan1`, `EarlierThan2` and `EarlierThan3`—which are analogous to `EqualTo1`, `Equalto2`, and `EqualTo3`, that are also boolean functions, and which test whether `t1` is an earlier `Time` value than `t2`. Test the `EarlierThan` functions by replacing, in TT3.CPP, the while-loop condition with the appropriate form of, first, `EarlierThan1`, then `EarlierThan2`, and finally, `EarlierThan3`. Be sure to re-compile, re-link and re-run for each test.

◯ Instructor checkpoint 7.1

Module 8

Operator overloading: arithmetic, relational and I/O operators

8.1 Objectives

- To understand what is meant by an *overloaded operator*.
- To understand how to define an overloaded arithmetic operator.
- To understand how to define an overloaded relational operator.
- To understand how to define an overloaded input or output operator and to understand why these overloaded operators have to be defined as friend functions.
- To understand what is meant by a *default parameter* in a function and how default parameters can be used to define, in effect, several functions at once (or, in particular, in our case, several constructors at once).

New C++ reserved word
`operator`

8.2 List of associated files

- TESTIME4.CPP is a test driver for the `Time` class defined in TIME4.H and TIME4.CPP.
- TIME4.H is the specification file for the fourth version of the `Time` class and contains overloaded operator definitions.
- TIME4.CPP is the implementation file corresponding to TIME4.H.

8.3 Overview

This Module deals with *operator overloading*, so let's begin our discussion with some motivation.

As you know, you can use the C++ operator `==` to test whether two values of type `int` are the same, and we regularly do so in such expressions as "`if (m == n) ...`".

It would be much more "natural" to be able to do a similar thing with objects of the classes we design and use as well. In other words, it would be nice if we could just use an expression like "`if (t1 == t2) ...`" to test whether two `Time` values are the same, rather than a more cumbersome thing like "`if (EqualTo(t1, t2)) ...`".

Fortunately, C++ allows us to do this. But it doesn't just happen. The `==` operator is not "aware" of `Time` values, so it has no idea how to tell whether two such values are the same or not. Thus we have to make the `==` operator aware of `Time` values and, in particular, tell it how to determine whether two `Time` values are the same. When we do this, we are said to *overload* the `==` operator (i.e., give it more work to do, one supposes).

Most (but not all) C++ operators can be overloaded, and *should* be overloaded whenever it makes sense to do so. For example, if the notion of addition makes sense for objects of a class, then we can always write an `Add` function and call `Add(v1, v2)` to add the two values `v1` and `v2`. However, we can also overload the `+` operator and then simply write `v1 + v2` instead, which is a much more "natural" thing to do if we are "adding" two quantities of any kind.

Similarly, we know that a statement like

```
cout << i;
```

will output the integer value in `i` if `i` is an `int` variable, but we had to write a `Display` function in order to output a `Time` value `t`, and use the syntax

```
t.Display();
```

which is also quite awkward. But, if we could instead simply use

```
cout << t;
```

to output a `Time` value, would this not be much more natural? Once again, we can. But this time we have to overload the `<<` operator. The input operator `>>` can be similarly overloaded so that a statement like

```
cin >> t;
```

for reading a `Time` value can be used.

In general, we like our own classes to behave as much as possible like the built-in data types. Programmer-defined data types that do so are sometimes called *first-class types*. The advantage of operator overloading is that it allows the programmer to use the same sort of concise expressions for programmer-defined types that C++, with its extensive collection of operators, provides for built-in types.

8.4 Sample Programs and Other Files

8.4.1 TESTIME4.CPP is a driver for the Time class in TIME4.H and TIME4.CPP

```
// Filenme: TESTIME4.CPP
// Purpose: Tests the Time class in TIME4.H and TIME4.CPP, which
//          has a four-in-one constructor, accessor functions,
//          friend functions, and overloaded operators.

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

#include "TIME4.H"
#include "PAUSE.H"


int main()
{
    cout << endl
         << "This program tests the Time class with "
         << "a four-in-one constructor, "             << endl
         << "accessor functions, friend functions, "
         << "and overloaded operators. "              << endl
         << "You should study the source code "
         << "at the same time as the output. "        << endl << endl;

    Time t0;      // Default constructor (no parameters) invoked
    Time t1(12);  // One-parameter constructor invoked
    Time t2(11, 45);      // Two-parameter constructor invoked
    Time t3(11, 44, 55);  // Three-parameter constructor invoked
    cout << t0;  cout << endl;  // Overloaded << operator used with cout
    cout << t1;  cout << endl;
    cout << t2 << setw(12) << t3;  cout << endl;  // How does setw work?
    Pause(0);

    while (!(t3 == t2))  // Overloaded == operator
    {
        cout << t3;  cout << endl;
        t3.Increment();
    }
    if (t3 == t2)
        cout << "\nt2: " << t2 << "\nt3: " << t3 << endl;
    Pause(0);

    Time t4 = t1 + t2;  // Overloaded + operator used in initialization
    Time t5;
    t5 = t1 + t3;       // Overloaded + operator used in assignment
    cout << t4;  cout << endl;
    cout << t5;  cout << endl;
    Pause(0);

    ofstream outFile("times.tmp"); // Initialize outFile object with specific file
    outFile << t1 << endl // Overloaded << operator used with outFile
            << t2 << endl // Cascading of output values, i.e.,
            << t3 << endl // ... << value1 << value2 << value3 << ...
            << t4 << endl
            << t5 << endl;
    outFile.close();
```

```
    ifstream inFile("times.tmp");
    Time t;

    inFile >> t;  // Overloaded >> operator used with inFile
    while (inFile)
    {
        cout << t << endl;
        inFile >> t;
    }
    inFile.close();
    Pause(0);

    inFile.open("times.tmp");
    // Cascading of input values (... >> value1 >> value2 >> value3 ...):
    inFile >> t1 >> t2 >> t3 >> t4 >> t5;
    inFile.close();
    cout << t1 << "  "
         << t2 << "  "
         << t3 << "  "
         << t4 << "  "
         << t5 << endl;

    return 0;
}
```

8.4.1.1 What you see for the first time in TESTIME4.CPP

- `Time` objects declared with zero, one, two and three in-parameters

  If you look at the declarations of `t0`, `t1`, `t2` and `t3`, they would suggest that the `Time` class has at least four different constructors, including the default constructor, and in one sense it does. However, as you will see in the class specification, the use of *default parameters* allows us to place all four constructors in a single definition. Can you make an educated guess now as to what `Time` values might be set for each of `t0`, `t1`, `t2`, and `t3` by the various constructors?

- Use of the overloaded `+` operator to add two `Time` values

  We compute `t1 + t2` and use the result to initialize `t4` at the time of its declaration, and also compute `t1 + t3` and assign the result to `t5`, which has previously been declared.

- Use of the overloaded `==` operator to test whether two `Time` values are the same

  On two occasions we use the boolean expression `(t1 == t2)`, which employs the overloaded `==` operator.

- Use of the overloaded `<<` stream operator to output `Time` values (with "cascading")

  Once we have overloaded the `<<` operator for `Time` values we can use it in a manner precisely analogous to the way we have used it to output values of the built-in types like `int` and `double`, including the "cascading" of `Time`

values with and without other values. By "cascading" we simply mean the use of more than one instance of the `<<` operator and a value with a single instance of the output stream, as in

```
cout << timeValue1 << otherValue << timeValue2 << ...
```

But if we try to use a manipulator (say `setw`, for example) in an expression like

```
cout << setw(12) << someTimeValue
```

will it work, and if so, how? Think about this, and note that such an expression actually appears in this program.

- Use of the overloaded `>>` stream operator to input `Time` values (also with cascading)

  As with the `<<` operator, once we have overloaded the `>>` operator for `Time` values we can use it to read in `Time` values from an input stream into `Time` variables, and "cascade" those variables as well if we choose.

- Use of both standard streams and file streams with the overloaded input and output operators

  This program illustrates use of the standard output (`cout`) and a file output stream (`outFile`) with `<<` and a file input stream (`inFile`) with `>>`. The activities ask you to use `cin` with `>>`.

- Use of `ofstream outFile("times.tmp");` to initialize an `ofstream` object (`outFile`) with the name of a specific file (`times.tmp`).

8.4.1.2 Additional notes and discussion on TESTIME4.CPP

This program should give you a sense of how much more "natural" the code looks when you have overloaded operators available for the "usual" operations associated with your class objects, such as addition, equality testing, input and output.

Note that in the condition for the while-loop we used `!(t1 == t2)` because we did not have available either `!=` or `<` for `Time` values.

Two other questions may have occurred to you, so let's deal with them now.

First, you may have been wondering why we left the `Increment` function and did not replace it by an overloaded `++` operator. We could have done this, but there are more issues involved in this case, so we omit the overloaded increment operator for the time being.

Second, you may have wondered about the assignment operator and why we have not had to overload it. That is, we blithely went ahead and assigned a `Time` value to a `Time` variable, assuming this would work fine. And in fact it does, *in this case*. The assignment operator works for objects of *any* class by default, but its default behavior is to perform a *memberwise assignment* of an object's data. For the next while this will be fine, but eventually we will encounter cases involving *dynamic data* where this is no longer adequate, and we will then have to overload the assignment operator for our classes.

8.4.1.3 Follow-up hands-on activities for TESTIME4.CPP

□ Copy and study the program in TESTIME4.CPP. You may run and test the program now if you wish, or wait until you have studied the revised version of the Time class in TIME4.H and TIME4.CPP.

□ This activity is designed to get you thinking about the question of *automatic type conversion* when dealing with classes. It's a topic we could not mention and you might not suffer from the omission, but you should know something about it to avoid some problems that are always waiting in the weeds for you to come by.

Here we just want you to start thinking about it. We'll get you to do some experimentation after you've had a look at TIME4.H and TIME4.CPP.

You can see from TESTIME4.CPP that the overloaded `+` operator allows you to add two `Time` values in an expression like `t1 + t2`. But what if you tried to "compute" an expression like `t + 1` or `1 + t`, where `t` is also a `Time` value? Does such an expression make sense? It may make sense to you, but what, if anything, does it mean to C++? For that matter, what does it really mean to you—in `t + 1`, for example, do you mean to add 1 second, 1 minute, 1 hour, or what, to `t`?

Note first that when we overload an operator like + and write expressions like `t1 + t2`, such an expression is equivalent to writing `t1.operator+(t2)`, in which operator+ is the name of the member function which overloads the `+` operator.

Now, here's how C++ thinks when asked to evaluate an expression like `t + 1`, with t of type `Time`: First, it looks to see if there an overloaded version of the `+` operator with `Time` as the type of its first parameter and `int` as the type of its second. In TIME4.H and TIME4.CPP we have `operator+` as a member function, so it looks to see if there's a version of `operator+` that it can use in the form `t.operator+(1)`. Of course there isn't, so what to do? The next thing it does is to look to see if there is a constructor which it can use to convert the '`1`' to a `Time` value. And, luckily, there is. The one-parameter constructor is used to convert the '`1`' to the `Time` value `01:00:00`, which is then added to `t` to compute the result. So, `t + 1` really means, to C++, `t + (1 hour)`.

But now look at what happens when you try to compute `1 + t`. The first attempt is to match `1.operator+(t)` with something that makes sense. This expression itself does not make sense, since `1` is not even an object, let alone an object with `operator+` defined for it, so we have a syntax error.

Do what you can to verify (and understand) the above remarks.

8.4.2 TIME4.H and TIME4.CPP extend the Time class with default parameters and overloaded operators

```
// Filename: TIME4.H
// Purpose:  Specification file corresponding to TIME4.CPP.
//           This version adds three overloaded operators and default
//           parameters in order to combine constructor definitions.


#ifndef TIME4_H
#define TIME4_H


class Time
{
    // The following two functions are "overloaded operators",
    // implemented as friend functions.

    friend istream& operator>>(/* inout */ istream& inputStream,
                               /* out */    Time& t);
    // Pre:  The output stream outputStream is open.
    // Post: Time t has been output in the form HH:MM:SS.
    //       outputStream is still open.

    friend ostream& operator<<(/* inout */ ostream& outputStream,
                               /* in */    const Time& t);
    // Pre:  The output stream outputStream is open.
    // Post: Time t has been output in the form HH:MM:SS.
    //       outputStream is still open.

public:

    // A new "four-in-one" constructor with "default parameter values".
    Time(/* in */ int hoursInitial   = 0,
         /* in */ int minutesInitial = 0,
         /* in */ int secondsInitial = 0);

    // The following five "interface functions" are exactly the
    // same as those in TIME3.H.  For brevity we have omitted the
    // comments and pre/post-conditions, which are also the same.
    // Note that the constructors and Display are missing.
    // The constructors have been replaced by the new one above.
    // Display will now be implemented as an overloaded operator,
    // as will the (unique) equality operation and an add operation.
    void Set(int hoursNew, int minutesNew, int secondsNew);
    void Increment();
    int Hours() const;
    int Minutes() const;
    int Seconds() const;


    // The following two functions are also "overloaded operators",
    // but are member functions.

    Time operator+(/* in */ const Time& otherTime) const;
    // Pre:  self and otherTime have been initialized.
    // Post: Value returned is equivalent to self advanced by otherTime.

    bool operator==(/* in */ const Time& otherTime) const;
    // Pre:  none
    // Post: Returns true if self equals otherTime and false otherwise.
```

```
private:

    int hours;
    int minutes;
    int seconds;
};

#endif
```

Implementation file TIME1.CPP starts below heavy line:

```
// Filename: TIME4.CPP
// Purpose:  Implementation file corresponding to TIME4.H.
//           This version adds three overloaded operators and default
//           parameters in order to combine constructor definitions.

#include <iostream>
using namespace std;

#include "TIME4.H"


// Private data members of the Time class:
//     int hours;
//     int minutes;
//     int seconds;


// This is a "four-in-one" constructor, which may take zero, one,
// two, or three parameters.
//*********************************************************************
Time::Time(/* in */ int hoursInitial,
           /* in */ int minutesInitial,
           /* in */ int secondsInitial)
// Constructor
// Pre:  0 <= hoursInitial   <= 23  and
///      0 <= minutesInitial <= 59  and
//       0 <= secondsInitial <= 59
// Post: Class object is constructed
//       and the time is set according to the incoming parameters.
{
    hours = hoursInitial;
    minutes = minutesInitial;
    seconds = secondsInitial;
}


// The following five "interface functions" are exactly the same as
// the corresponding ones in TIME3.CPP.  For brevity we have omitted
// comments and pre/post-conditions, which are also the same, as are
// the function implementations.


//*********************************************************************
void Time::Set(int hoursNew, int minutesNew, int secondsNew)
{
    hours = hoursNew;
    minutes = minutesNew;
    seconds = secondsNew;
}
```

```
//*************************************************************
void Time::Increment()
{
    seconds++;
    if (seconds > 59)
    {
        seconds = 0;
        minutes++;
        if (minutes > 59)
        {
            minutes = 0;
            hours++;
            if (hours > 23)
                hours = 0;
        }
    }
}

//*************************************************************
int Time::Hours()   const { return hours; }
int Time::Minutes() const { return minutes; }
int Time::Seconds() const { return seconds; }



// The following functions are the definitions of the four overloaded
//  operators supplied by this implementation of the Time class.

//*************************************************************
Time Time::operator+(/* in */ const Time& otherTime) const
// Pre:  self and otherTime have been initialized.
// Post: Value returned is equivalent to self advanced by otherTime.
{
    int hrs;
    int mins;
    int secs;

    Time t;

    secs = seconds + otherTime.seconds;
    t.seconds = secs % 60;
    mins = minutes + otherTime.minutes + secs / 60;
    t.minutes = mins % 60;
    hrs = hours + otherTime.hours + mins / 60;
    t.hours = hrs % 24;

    return t;
}



//*************************************************************
bool Time::operator==(/* in */ const Time& otherTime) const
// Pre:  none
// Post: Returns true if self equals otherTime and false otherwise.
{
    return (hours == otherTime.hours      &&
            minutes == otherTime.minutes  &&
            seconds == otherTime.seconds);
}
```

```
//*************************************************************
istream& operator>>(/* inout */ istream& inputStream,
                    /* out */   Time& t)
// Pre:  The input stream inputStream is open.
//       Time values in the input stream have the form HH:MM:SS.
// Post: Time t has been read in.
//       inputStream is still open.
{
    char ch;

    inputStream >> t.hours   >> ch
                >> t.minutes >> ch
                >> t.seconds;

    return inputStream;
}


//*************************************************************
ostream& operator<<(/* inout */ ostream& outputStream,
                    /* in */    const Time& t)
// Pre:  The output stream outputStream is open.
// Post: Time t has been output in the form HH:MM:SS.
//       outputStream is still open.
{
    if (t.hours < 10)
        outputStream << '0';
    outputStream << t.hours << ':';
    if (t.minutes < 10)
        outputStream << '0';
    outputStream << t.minutes << ':';
    if (t.seconds < 10)
        outputStream << '0';
    outputStream << t.seconds;

    return outputStream;
}
```

8.4.2.1 What you see for the first time in TIME4.H and TIME4.CPP

- A four-in-one constructor that uses *default parameter values*

  Default parameter values can be used in *any* function, not just in class constructors as shown here. The idea is this: If we omit a parameter when we call a function (or invoke a constructor), then if that parameter has a *default value*, that value will be used automatically for that parameter in that function call (or constructor invocation).

  There are some restrictions. All of the parameters (as in our case here), or only some, can have default values. But if not all parameters do have default values, those that *do* must be the *rightmost* (i.e., the trailing) parameters. This is so that, for example, if a function with five formal parameters is called with only three actual parameters, the compiler can assume (*must* assume, in fact) that the default values are to be used for the *last two* parameters.

Note that a default parameter is given its value in the function prototype in the specification file by following the associated parameter with an equal sign and the default value, and that this is *not* repeated in the function header of the function definition in the implementation file.

- The arithmetic operator `+` and the relational operator `==` implemented as member functions

  These two overloaded operators could be implemented as friend functions and the activities ask you to do this. Though similar flexibility is possible for other arithmetic and relational operators, you should not assume that overloading of all arithmetic or relational operators is always this straightforward or that there are no other issues to be considered.

- The input stream operator `>>` and the output stream operator `<<` implemented as friend functions

  An overloaded operator must always have at least one parameter that is a class object (or a reference to a class object) of the operator's class and an overloaded operator can only be implemented as a member function if the first (or only) parameter is such a parameter. This means that neither `>>` nor `<<` can be implemented as a member function since in each case the first parameter must be a stream object, not a `Time` object. As a result, these two overloaded operators are generally implemented as friend functions, though they could also be implemented simply as non-member functions, provided the necessary access to the private data members of the class is avaiable in some other way.

- Return-values that are a *stream references* (`istream&` and `ostream&`)

8.4.2.2 Additional notes and discussion on TIME4.H and TIME4.CPP

Whether an overloaded operator (operator function) is implemented as a member function or a friend function, it is used in the same way. That is, we write `t1 + t2` to indicate the "sum" of two `Time` objects in either case.

Note that although we write "`operator+`" as the name of the operator function, you will also see "`operator +`" used, in which there is a space between the reserved word "`operator`" and "`+`". Either is acceptable.

8.4.2.3 Follow-up hands-on activities for TESTIME4.CPP, TIME4.H and TIME4.CPP

□ Copy TIME4.H and TIME4.CPP and study this version of the `Time` class. Then combine the class code with TESTIME4.CPP and test the driver program if you have not already done so. Be sure to explain the behavior of the manipulator `setw(12)` when used with a `Time` value.

□ Make copies of TESTIME4.CPP, TIME4.H and TIME4.CPP called, respectively, TT4.CPP, T4.H and T4.CPP for testing purposes.

□ In T4.H and T4.CPP change the implementations of the overloaded `+` and `==` operators from member functions to friend functions. Test with TT4.CPP to make sure they are working properly.

□ Design and write two new overloaded operator functions for the less than operator (`<`) and the inequality operator (`!=`). Implement them both as member functions. Test both operators by replacing the condition `!(t1 == t2)` of the while-loop in TT4.CPP first with an equivalent expression involving the overloaded `<` operator and then with an equivalent expression involving the `!=` operator. Finally, repeat the previous steps, implementing both overloaded operators as friend functions.

Based on SHELL.CPP

□ Design and write a program based on SHELL.CPP (and using, therefore, the `Menu` and `TextItems` classes) which has the following menu:

```
Computing with Time

 1. Quit
 2. Get Information
 3. Add Two Times
 4. Subtract Two Times
```

When the user chooses option 3, the program should prompt for the entry of two `Time` values from the keyboard (here's where you use `cin` to read `Time` values), add the two values, and display the result. For option 4 the program performs analogously, except that the two `Time` values are subtracted. You will have to implement an overloaded subtraction operator −, which behaves like this: `t1 - t2` means the `Time` value obtained by starting with `t1` and moving "backward in time" an amount `t2`, and wrapping around if you pass `00:00:00`.

Give your copy of SHELL.CPP the name COMPTIME.CPP and revise it accordingly. You will also need a program description file which should be called TIMECOMP.DAT, and the program should display this file when the user chooses option 2 from the menu.

□ This activity is where you experiment with the notion of *automatic type conversion* that we started you thinking about earlier, in the activities associated with TESTIME4.CPP. Make another copy of TESTIME4.CPP called TT4A.CPP, and replace the body of the `main` function with

```
Time t(9, 10, 11);
cout << t + 1 << endl;
cout << 1 + t << endl;
```

and try to compile. One of these output statements will cause a compile-time error, if you `#include` the original TIME4.H header file in TT4A.CPP, and if you remember our earlier discussion in the activities for TESTIME4.CPP you will know which one.

If you then `#include` the revised T4.H, which has the overloaded `+` operator implemented as a friend function, TT4A.CPP should compile and run fine, if you link with TT4.OBJ.

If you now re-read that earlier discussion, you should be able to figure out why this is so, and hence deduce one reason why the friend function implementation of the overloaded `+` operator is sometimes better than the member function version.

◯ Instructor checkpoint 8.1

Module 9

One-dimensional arrays with a simple component data type (our third structured data type)

9.1 Objectives

- To appreciate the need for the *array data type* and to be able to recognize problem situations in which this data type will be useful.
- To understand that a *one-dimensional array* is a *structured data type* and that the nature of its particular structure may be described by the terms *linear* and *homogeneous*.
- To understand that an *array variable* can hold several values of a given, fixed data type, each one in an *element* or *component* of the array variable.
- To learn how to declare an array variable
  - directly (or anonymously), without benefit of a prior type definition
  - by first defining an array data type, giving it a name with `typedef`, and then using that definition in the variable declaration

    This second option is the recommended one when you have an array data type of fixed size that you need to use in many different places.
- To learn what an *array index* is, and how to use the index of an array variable to access an element (component) of that variable.

- To learn how to process some or all of the elements of an array, including the usual *for-loop idiom* for processing all the elements of an array from first to last.
- To understand how to use an array as a function parameter.

9.2 List of associated files

- MEANDIF1.CPP motivates the need for the array data type, but does not itself contain any arrays.
- MEANDIF2.CPP shows how the program of MEANDIF1.CPP would be written using an array and thereby introduces the fundamental array notions.
- MEANDIF3.CPP extends MEANDIF2.CPP by introducing an enumeration type as the array index and defining an array data type with `typedef`.
- ARRPROC.CPP illustrates some array processing, as well as arrays as function parameters.

9.3 Overview

This Module introduces the *array data type*, which is the simplest data structure to use whenever you have a situation where you would otherwise need a lot of simple variables, all of which will be used in the same way.

The first sample program (MEANDIF1.CPP) gives you a feel for the kind of situation in which it would become very tedious to deal with a large amount of data using only our current knowledge.

The second sample program (MEANDIF2.CPP) re-solves the problem of MEANDIF1.CPP, using an array data structure. You will see that in this implementation it becomes almost trivial to make the changes necessary for dealing with as much data as you would like. In this program you also see how to declare an array variable directly, read values into an array, use the array component values in a computation, and display the values in the array.

The third sample program (MEANDIF3.CPP) extends MEANDIF2.CPP by showing how to use `typedef` to give a name to an array data type, and then how to use that definition in the declaration of an array variable. This program also uses an enumerated type for the array index, which is sometimes expressed by saying that we have an index with *semantic content*. This is just a fancy way of saying that we have an index with values that mean something, instead of just numbers for which we have to make a "mental translation" every time we use one. (As in, "If the index is 3, this must be Wednesday ...")

The last sample program (ARRPROC.CPP) shows some array initializations, some array processing, and the passing of arrays as parameters.

9.4 Sample Programs

9.4.1 MEANDIF1.CPP motivates the array data type

```
// Filename: MEANDIF1.CPP
// Purpose:  This program reads in seven temperatures, computes and
//           prints out their average (i.e. their "mean"), and then
//           finally prints out the difference of each individual
//           temperature from the average.

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

int main()
{
    int t1, t2, t3, t4, t5, t6, t7;
    double averageTemp;

    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);
    cout << setprecision(1);

    cout << endl;
    cout << "This program asks for 7 daily "
         << "temperatures, then prints out their average " << endl;
    cout << "and their differences from that average. "    << endl;
    cout << endl;

    cout << "Enter the 7 daily temperatures "
         << "as integers, then press ENTER: "   << endl;
    cin >> t1 >> t2 >> t3 >> t4 >> t5 >> t6 >> t7;

    averageTemp = double(t1 + t2 + t3 + t4 + t5 + t6 + t7) / 7;

    cout << endl;
    cout << "The average temperature was "
         << averageTemp << "."                     << endl;
    cout << endl;

    cout << "The daily temperatures and their "
         << "differences from the average are: "   << endl;
    cout << endl;
    cout << setw(4) << t1 << setw(7) << t1-averageTemp << endl;
    cout << setw(4) << t2 << setw(7) << t2-averageTemp << endl;
    cout << setw(4) << t3 << setw(7) << t3-averageTemp << endl;
    cout << setw(4) << t4 << setw(7) << t4-averageTemp << endl;
    cout << setw(4) << t5 << setw(7) << t5-averageTemp << endl;
    cout << setw(4) << t6 << setw(7) << t6-averageTemp << endl;
    cout << setw(4) << t7 << setw(7) << t7-averageTemp << endl;
    cout << endl;

    return 0;
}
```

9.4.1.1 Notes and discussion on MEANDIF1.CPP

This program is meant only to motivate our introduction of the array data type, though the program itself does not contain any arrays. The idea is to show you the solution to a problem (finding the average temperature for a week) for which it is very difficult (or at least very tedious) to extend the solution to a larger problem, even though the larger problem is precisely analogous to the smaller one.

9.4.1.2 Follow-up hands-on activities for MEANDIF1.CPP

□ Copy, study and test the program in MEANDIF1.CPP.

□ Revise MEANDIF1.CPP so that it performs in the same way, except that it computes the average temperature for ten days rather than seven.

◯ Instructor checkpoint 9.1

9.4.2 MEANDIF2.CPP performs just as MEANDIF1.CPP but uses a one-dimensional array

```
// Filename: MEANDIF2.CPP
// Purpose:  This program reads in seven temperatures, computes and
//           prints out their average (i.e. their "mean"), and then
//           finally prints out the difference of each individual
//           temperature from the average.

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

int main()
{
    const int NUMBER_OF_DAYS = 7;

    int i;
    int temp[NUMBER_OF_DAYS];
    float averageTemp;
    float sum;

    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);
    cout << setprecision(1);

    cout << endl;
    cout << "This program asks for "
         << NUMBER_OF_DAYS << " daily "
         << "temperatures, then prints out their average " << endl;
    cout << "and their differences from that average. "    << endl;
    cout << endl;

    cout << "Enter the " << NUMBER_OF_DAYS
         << " daily temperatures "
         << "as integers, then press RETURN: "           << endl;
    for (i = 0; i < NUMBER_OF_DAYS; i++)
        cin >> temp[i];

    sum = 0;
    for (i = 0; i < NUMBER_OF_DAYS; i++)
        sum = sum + temp[i];
    averageTemp = sum/NUMBER_OF_DAYS;
    cout << endl;
    cout << "The average temperature was "
         << averageTemp << "."                       << endl;
    cout << endl;

    cout << "The daily temperatures and their "
         << "differences from the average are: "    << endl;
    cout << endl;
    for (i = 0; i < NUMBER_OF_DAYS; i++)
        cout << setw(4) << temp[i]
             << setw(7) << temp[i]-averageTemp      << endl;
    cout << endl;

    return 0;
}
```

9.4.2.1 What you see for the first time in MEANDIF2.CPP

- A one-dimensional array of component-type `int` and number of components 7, declared by the statement

  `int temp[NUMBER_OF_DAYS];`

  in which `NUMBER_OF_DAYS` is a previously defined named constant with a value of 7.

- The syntax for referring to a single array element

  Note that `temp[i]` refers to the element which has index `i`, but that this is actually the element in *position* `i+1`, because in C++ array indices always start at 0.

- The standard *for-loop idiom* for accessing all elements of an array from first (index `0`) to last (index `n-1`, if there are `n` elements)

  In general the idiom would look like this:

  ```
  for (int i = 0; i < n; i++)
      ...
  ```

  in which the loop control variable declaration may actually *not* appear, if it has been given earlier.[1]

9.4.2.2 Additional notes and discussion on MEANDIF2.CPP

Note that the entire array in this program is processed three different times—once for reading in values, once for summing the values, and once for displaying the values—and the same for-loop idiom is used in each case.

9.4.2.3 Follow-up hands-on activities for MEANDIF2.CPP

□ Copy, study and test the program in MEANDIF2.CPP.

□ Revise MEANDIF2.CPP so that it performs in the same way, except that it computes the average temperature for ten days rather than seven. Compare the necessary modification(s) in this program with those required to modify the program in MEANDIF1.CPP to perform the same task.

□ Replace the first line of any one of the for-loops with a line like
`for (i = 0; i <= n-1; i++)`
and show that it works equally well. In fact, you might be excused for thinking that writing it this way makes more sense. However, the idiom we described above is the much more commonly seen form.

◯ Instructor checkpoint 9.2

[1]The new C++ Standard requires that the loop-control variable in a for-loop be local to the loop if it is declared in the loop, but as of this writing not all C++ compilers comply with this part of the Standard. For example, Visual C++ 6 does not.

9.4.3 MEANDIF3.CPP extends MEANDIF2.CPP with an enumerated type as array index and use of typedef to define an array data type

```
// Filename: MEANDIF3.CPP
// Purpose:  This program reads in temperatures for each day of
//           the week, computes and prints out their average
//           (i.e. their "mean"), and then finally prints out
//           the difference of each individual temperature from
//           the average.

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

int main()
{
    const int NUMBER_OF_DAYS = 7;
    enum DayType {SUN, MON, TUE, WED, THU, FRI, SAT}; // For array indices
    typedef int TemperatureType[NUMBER_OF_DAYS]; // Define an array data type.

    TemperatureType temp; // Use type definition to declare array variable.
    double averageTemp;
    double sum;
    DayType day;

    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);
    cout << setprecision(1);

    cout << endl;
    cout << "This program asks for "
         << NUMBER_OF_DAYS << " daily "
         << "temperatures, then prints out their average " << endl;
    cout << "and their differences from that average. "    << endl;
    cout << endl;

    cout << "Enter the " << NUMBER_OF_DAYS
         << " daily temperatures "
         << "as integers, then press ENTER: "           << endl;
    for (day = SUN; day <= SAT; day = DayType(day + 1)) // Use enumerated values
        cin >> temp[day];                               // as array indices.

    sum = 0;
    for (day = SUN; day <= SAT; day = DayType(day + 1))
        sum = sum + temp[day];
    averageTemp = sum/NUMBER_OF_DAYS;
    cout << endl;
    cout << "The average temperature was "
         << averageTemp << "."                    << endl;
    cout << endl;

    cout << "The daily temperatures and their "
         << "differences from the average are: "  << endl;
    cout << endl;
    for (day = SUN; day <= SAT; day = DayType(day + 1))
        cout << setw(4) << temp[day]
             << setw(7) << temp[day]-averageTemp  << endl;
    cout << endl;

    return 0;
}
```

9.4.3.1 What you see for the first time in MEANDIF3.CPP

- The use of `typedef` to define an array data type, i.e., to give a name (`TemperatureType`) to a certain kind of array of a certain size, so that the name can be used in a type declaration

  Note that because we only use that name once here, there is no real advantage. The usefulness of doing this comes when you have to declare several arrays of the same kind at various points throughout your program, and can then define the name once but use it many times. One reason for doing this is the same reason you define named constants: if you have to change the definition you only have to make the change in one place.

- The use of an enumerated type for an array index

  Using an enumerated type for an array index makes the indices more meaningful and works because the internal integer values that represent the enumerated values start at 0 and increase to n-1, if there are n enumerated values, just as the index values of an array start at 0 and increase to n-1 if there are n elements in the array. We sometimes say that an array index value that is an enumerated type has "semantic content", a fancy way of saying that the indices have "meaning".

9.4.3.2 Additional notes and discussion on MEANDIF3.CPP

Note the use of a variable of enumerated data type `DayType` as a loop control variable in the for-loop as well as for the index of the array variable. Observe in particular the use of the expression `DayType(day + 1)` to make sure that day is assigned a value of the proper type (`DayType`). The reason for this is that adding `1` to `day` means that the result (`day + 1`) is converted to an integer and must therefore be cast back to a `DayType` value.

Note as well that this particular enumerated type really only applies if we have just seven days worth of temperatures, at least in the current context.

9.4.3.3 Follow-up hands-on activities for MEANDIF3.CPP

□ Copy, study and test the program in MEANDIF3.CPP. Verify that its behavior is the same as that of the previous two versions of this program.

◯ Instructor checkpoint 9.3

9.4.4 ARRPROC.CPP illustrates array initialization, more array processing and passing array parameters

```
// Filename: ARRPROC.CPP
// Purpose:  Illustrates array initialization, some additional
//           array processing, and passing array parameters.

#include <iostream>
using namespace std;


int MaxArrayValue(/* in */ const int a[],
                  /* in */ int first,
                  /* in */ int last);

void GetArrayValuesFromUser(/* out */ int a[],
                            /* in */  int howMany);

void Swap(/* inout */ int& i1,
          /* inout */ int& i2);


int main()
{
    cout << endl;
    cout << "\nThis program illustrates some features "
         << "of arrays, as well as some array "
         << "\nmanipulation and the use of arrays as "
         << "function parameters.  Study the code "
         << "\nwhile running the program. ";
    cout << endl << endl;

    // Inititalizing a 10-element array with 10 values:
    int a1[10] = {5, -6, 12, 43, 17, -3, 29, 14, 35, 4};

    // Initializing a 10-element array with only 2 values,
    // so the other 8 values are 0 by default.
    int a2[10] = {-5, -6};

    // Initializing a 6-element array with 6 values.
    // Size of array is determined by number of values in braces.
    int a3[] = {1, 3, 5, 7, 9, 11};  // Values increasing

    // Initializing a 7-element array with 7 values.
    // Size of array is determined by number of values in braces.
    int a4[] = {26, 24, 22, 20, 16, 12, 0};  // Values decreasing

    // Initializing a 9-element array with 9 values.
    // Size of array is determined by number of values in braces.
    int a5[] = {31, 24, -6, 0, -2, 13, 8, 11, 16};  // Values "at random"

    cout << endl;

    cout << "The maximum value in the first array is "
         << MaxArrayValue(a1, 0, 9) << "." << endl << endl;
    cout << "The maximum value in the second array is "
         << MaxArrayValue(a2, 0, 9) << "."  << endl << endl;
    cout << "The maximum value in the third array is "
         << MaxArrayValue(a3, 0, 5) << "."  << endl << endl;
    cout << "The maximum value in the fourth array is "
         << MaxArrayValue(a4, 0, 6) << "."  << endl << endl;
```

```
    cout << "The maximum value in the fifth array is "
         << MaxArrayValue(a5, 0, 8) << "."  << endl << endl;

    cout << "The amount of storage used by the first array is "
         << sizeof(a1) << " bytes." << endl;

    cout << "\nWe now enter some values into an array, "
         << "then choose two indices and report the "
         << "\nlargest value between those two indices (inclusive). ";

    bool finished;
    char response;

    int b[15];

    int numberOfValues;
    int first, last;

    do
    {
        cout << "\nHow many values would you "
             << "like to put into your array? ";
        cin >> numberOfValues;  cin.ignore(80, '\n');  cout << endl;
        GetArrayValuesFromUser(b, numberOfValues);

        cout << "Now we will find the largest value "
             << "in a range of your choosing. ";
        cout << "\nEnter the first and last array positions "
             << "you wish to examine: ";
        cin >> first >> last;  cin.ignore(80, '\n');  cout << endl;

        cout << "The maximum value in the array between "
             << "position " << first << " and position " << last
             << " is " << MaxArrayValue(b, first-1, last-1) << ".\n";

        cout << "\nWould you like to process another array? (Y/N) ";
        cin >> response;  cin.ignore(80, '\n');  cout << endl;

        finished = (response != 'Y'  &&  response != 'y');

    } while (!finished);

    cout << endl;
    cout << "The last set of array values entered was: " << endl;
    for (int i = 0; i < numberOfValues; i++)
        cout << b[i] << "  ";
    cout << endl;

    cout << "Now enter the positions of two values to swap: ";
    cin >> first >> last;  cin.ignore(80, '\n');  cout << endl;

    // Note the array componets used as actual parameters:
    Swap(b[first-1], b[last-1]);

    cout << "The revised array values, with "
         << "the two values swapped, is: "    << endl;
    for (/* int */ i = 0; i < numberOfValues; i++)
        cout << b[i] << "  ";
    cout << endl;

    return 0;
}
```

```
int MaxArrayValue(/* in */ const int a[],
                  /* in */ int first,
                  /* in */ int last)
// Pre:  The array "a" has been initialized, and
//       0 <= first <= last <= number of elements in "a" - 1.
// Post: Function value is the maximum value in the array "a"
//       between the indices "first" and "last", inclusive.
{
    int i, maxVal;

    maxVal = a[first];
    for (i = first+1; i <= last; i++)
    {
        if (a[i] > maxVal)
            maxVal = a[i];
    }
    return maxVal;
}


void GetArrayValuesFromUser(/* out */ int a[],
                            /* in */  int howMany)
// Pre:  howMany contains the number of values to be entered into "a".
// Post: "a" contains howMany integer values entered by the user.
{
    cout << "Enter " << howMany << " integer values on the "
         << "following line and press ENTER: \n";
    for (int i = 0; i < howMany; i++)
        cin >> a[i];
    cin.ignore(80, '\n');  cout << endl;
}


void Swap(/* inout */ int& i1,
          /* inout */ int& i2)
// Pre:  i1 and i2 have been initialized.
// Post: The values of i1 and i2 have been swapped.
{
    int temp = i1;
    i1 = i2;
    i2 = temp;
}
```

9.4.4.1 What you see for the first time in ARRPROC.CPP

- The initialization of a one-dimensional array in three different ways:
  - by indicating the size of the array in the declaration and supplying exactly that many values to initialize the components, as in
    ```
    int a1[10] = {5, -6, 12, 43, 17, -3, 29, 14, 35, 4};
    ```
  - by indicating the size of the array in the declaration, but supplying fewer values than needed to initialize all components so that all components after those specified will have a default value of 0 (for an array having a numeric component type), as in
    ```
    int a2[10] = {-5, -6};
    ```
  - by supplying only a set of values for initializing the components and letting the compiler count the values and thus determine the array's size, as in
    ```
    int a3[] = {1, 3, 5, 7, 9, 11};
    ```
- Passing an array as a parameter

  Note that we use `const` with an array parameter that is a conceptual in-parameter. This is consistent with our convention of passing *any* structured data type which is a conceptual in-parameter as a constant reference parameter. But, you ask, where's the ampersand (`&`) that indicates a reference parameter? The answer is that in C++ an array is *always* passed as a reference parameter (we have no other choice), so no ampersand is necessary.

  We should point out that there is no value within the square brackets of the formal array parameter, and since an array of any size may be passed, any value inserted will simply be ignored. And, finally, an *actual* array parameter is simply the name of the passed array.
- The use of `sizeof` to determine the amount of storage used by an array (which, of course, is *not* the same as the number of elements in the array)

9.4.4.2 Additional notes and discussion on ARRPROC.CPP

Note that in the single call to `Swap`, in `main`, each actual parameter is just an array element, *not* an entire array. The point is, sometimes we pass an entire array as a parameter, and sometimes just an array element, depending on what we are doing. Don't confuse the two.

Note also that in general you have to pass the *size* of the array (if the entire array is to be processed) or the *indices* of the *starting* and *ending* elements (if you are only processing part of the array), since otherwise the receiving function has no information on the size of the array or which portion of it you wish to process.

Position vs. index value in "locating" array elements

It is important to point out that from a program user's point of view, when asking that user what to do with the array, it makes more sense to refer to the *positions* of elements in the array (1, 2, 3, ... or first, second, third, ...) rather than *indices* (0, 1, 2, ...), although internally, i.e., from the programmer's point of view, it is the indices that are important. It's therefore up to the programmer to put on a "user hat" when testing or running this program.

9.4.4.3 Follow-up hands-on activities for ARRPROC.CPP

□ Copy, study and test the program in ARRPROC.CPP. Along with whatever other bugging you will be doing, try inserting different values within the square brackets of the array parameter in the prototype and header of the function `MaxArrayValue` to show that such array values are not necessary and are in fact ignored.

□ Design and write a program that will prompt the user to enter up to 20 capital letters, in no particular order, into an array of component type `char`. The program will then print out the first and last (in the alphabetical sense) from among all characters entered. For determining the first and last characters you must use a single function called `FindFirstAndLast`. Part of the problem is to determine the appropriate parameter list for this function. The basic ideas and much of the code can be taken from ARRPROC.CPP. Put your program in a file called F_AND_L.CPP.

◯ Instructor checkpoint 9.4

Module 10

A new simple data type: pointers to static data storage, including static arrays

10.1 Objectives

- To understand what is meant by a *pointer data type*.
- To learn how to declare and initialize, or assign a value to, a pointer variable.
- To understand what it means to *dereference* a pointer value.
- To understand the (intimate) relationship between pointers and arrays.
- To understand *pointer arithmetic* and know how to perform it.
- To understand the use of the `*`, `&`, and `->` operators in the context of pointers.
- To understand the `this` pointer in the context of class objects.
- To understand the basic mechanics of pointer manipulation in C++.

New C++ reserved word
`this`

10.2 List of associated files

- PTR_EX1.CPP illustrates simple integer pointers (pointers to `int`).
- PTR_EX2.CPP illustrates pointers to `double`, `char`, `bool`, `enum` and `struct` values.

- PTR_EX3.CPP illustrates the relationship between pointers and arrays.
- PTR_EX4.CPP illustrates pointers used in the context of arrays of structs.
- PTR_EX5.CPP illustrates pointer parameters.
- PTR_SWAP.CPP compares pointer and reference parameters for swapping two values.
- PTR_THIS.CPP illustrates the `this` pointer which is implicitly present in every class object.

10.3 Overview

This Module provides several sample programs that illustrate *pointer variables* in their simplest context: pointing at something that already exists. In other words we do not deal here with *dynamic storage* (creating new storage space for a pointer to point to) and all that that entails.

There are pros and cons to an initial simplistic approach like this. On the up side, you will have a chance to see what a pointer is, and get used to manipulating pointers, in a relatively non-threatening situation, i.e., without having to deal with the more involved aspects of the subject. On the down side, you will not get to see, in this Module, the really interesting ways in which pointers can be used. In fact, you may wonder what the fuss is all about, and the question most likely to occur to you here is, "Why bother?" So, we need to ask you to bear with us until we come back to pointers later for the more exciting stuff.

In the meantime, though, you need to know what a pointer is, at least. A *pointer* is a variable which contains the address in memory of another variable (the one "pointed to", so to speak). The *ampersand operator* (`&`) is used here again to indicate the "address of" a variable, in the following way (for example): If `p` is a pointer which can point to a variable of type int, then to make `p` point to an actual variable `n` of type `int` we could write

```
p = &n; // Read this as, "p gets assigned the address of n".
```

To declare p a variable of this type we would write

```
int* p;
```

which we read as "`p` is a pointer to `int`", or "`p` is an `int` pointer". Thus we now have two ways of referring to the integer value stored in `n`: using `n` itself as always, or referring to it "indirectly" by *dereferencing* the pointer `p`. This means writing `*p`, an expression which we can use in the same way we would use `n` itself and which would give us the same value.

These ideas are illustrated throughout the sample programs. We also introduce the `->` operator for accessing the fields of a struct or the members of a class object (when we have a pointer to the object and permission to access the members in question). Finally, we give a brief example illustrating the `this` pointer, which may be used within any class object to refer to itself.

10.4 Sample Programs

10.4.1 PTR_EX1.CPP illustrates simple integer pointers

```
// Filename: PTR_EX1.CPP
// Purpose:  Illustrates some simple integer pointers (pointers to int).

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

int main()
{
    cout << "\nThis program illustrates integer pointers."
         << "\nStudy the source code and the output simultaneously.\n\n";

    // Note the possible variations in the position of *.
    // The first variation is the one we use.
    int* iPtr1;
    int *iPtr2;
    int * iPtr3;

    // Here are some "ordinary" initialized variables of type "int".
    int i1 = 7;
    int i2 = 10;
    int i3 = -3;

    // We can assign the address of an "ordinary" variable to a
    // pointer variable, using the "address of" operator &, as in:
    iPtr1 = &i1;
    iPtr2 = &i2;
    iPtr3 = &i3;

    // Next we output some "values pointed to", i.e. the "referents"
    // of some pointers, which we get by "dereferencing" the pointer
    // variables, and we also output the values of the pointer variables
    // themselves, just out of curiosity, and because we can.
    cout << endl;
    cout << setw(4) << i1 << setw(4) << *iPtr1 << "  " << iPtr1 << endl;
    cout << setw(4) << i2 << setw(4) << *iPtr2 << "  " << iPtr2 << endl;
    cout << setw(4) << i3 << setw(4) << *iPtr3 << "  " << iPtr3 << endl;
    cout << endl;

    // Pointers can also be initialized at the time of declaration:
    int i = 15;
    int* iPtr = &i;
    cout << setw(4) << i << setw(4) << *iPtr << endl;
    cout << endl;

    // Don't use declarations like this:
    int* jPtr1, jPtr2;
    // This suggests you have declared two "pointer to int" variables,
    // when in fact you have only declared one (i.e. jPtr1). The second
    // variable (jPtr2) is in fact just an ordinary int variable:
    jPtr2 = 123;
    jPtr1 = &jPtr2;
    cout << setw(4) << jPtr2 << setw(4) << *jPtr1 << endl;
    cout << endl;

    return 0;
}
```

10.4.1.1 What you see for the first time in PTR_EX1.CPP

Study the code of this sample program carefully and be sure you identify each of the following:

- Several pointer variables of type "pointer to `int`" in three different syntactic variations, including the one we will normally use
- The assignment of a value to a pointer variable after the pointer variable has been declared
- The initialization of a pointer variable at the time of declaration
- The dereferencing of a pointer variable to access the value to which it points (in this case, to print out that value)
- The displaying of the actual value (i.e., the memory location) contained in the pointer variable itself (as opposed to displaying the value pointed to)

10.4.1.2 Additional notes and discussion on PTR_EX1.CPP

Note carefully the potential pitfall mentioned in the comments near the end of the program. This is (possibly) an argument against the syntactic convention we have chosen for pointer declarations, but it causes no problem at all if we adhere strictly to our longstanding convention of one declaration per line, which is itself not a bad convention to follow.

You can do it, but why bother?

Although this program demonstrates how to display the actual memory location contained in a pointer variable (just because it's possible, at least in C++), there is normally no call for the average programmer (except for the intrepidly curious) to know or care about this information.

10.4.1.3 Follow-up hands-on activities for PTR_EX1.CPP

□ Copy, study and test the program in PTR_EX1.CPP.

□ Make another copy of PTR_EX1.CPP called PTR_EX1A.CPP and revise the copy so that instead of initializing the variables `i1`, `i2` and `i3` in their declarations the program reads values for these variables that are entered at the keyboard by the user. The program must also display the values of the sum of `i1` and `i2` and the product of `i2` and `i3`. But here's the kicker: you *must not* use any of `i1`, `i2` or `i3` in any input (i.e., `cin`) statement or in any output (i.e., `cout`) statement. That is, you *must* use pointers to deal with the necessary values in all cases.

◯ Instructor checkpoint 10.1

10.4.2 PTR_EX2.CPP illustrates pointers to double, char, bool, enum and struct values

```
// Filename: PTR_EX2.CPP
// Purpose:  Illustrates pointers to double, char, bool, enum
//           and struct types.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program illustrates pointers to various types."
         << "\nStudy the source code and the output simultaneously.\n\n";

    double* dPtr;  // Pointer to double
    char*  cPtr;   // Pointer to char
    bool*  bPtr;   // Pointer to bool

    enum DayType { SUN, MON, TUE, WED, THU, FRI, SAT };
    DayType* dayPtr;  // Pointer to enum

    struct BankAccount
    {
        int idNum;
        char kind;
        double balance;
    };
    BankAccount* baPtr;  // Pointer to struct

    double cost = 19.95; // These are just ordinary variables,
    char grade = 'B';    // each one of a different type, and
    bool valid = true;   // each one initialized at the time
    DayType day = WED;   // of declaration.
    BankAccount myStash = {123, 'C', 199.99};

    dPtr = &cost;        // Here we make the pointers point
    cPtr = &grade;       // at the "ordinary" variables.
    bPtr = &valid;
    dayPtr = &day;
    baPtr = &myStash;

    // Now display the values:
    cout << *dPtr << endl;
    cout << *cPtr << endl;
    cout << ((*bPtr == true) ? "true" : "false") << endl << endl;
    if (*dayPtr == WED)
    {
              // Because this notation    ... C++ supplies this
              // is so cumbersome ...     alternate notation
              // vvvvvvvvvvv                vvvvvvvvvvv
        cout << (*baPtr).idNum   << "  " << baPtr->idNum   << endl;
        cout << (*baPtr).kind    << "  " << baPtr->kind    << endl;
        cout << (*baPtr).balance << "  " << baPtr->balance << endl;
    }
    cout << endl;

    return 0;
}
```

10.4.2.1 What you see for the first time in PTR_EX2.CPP

- Pointers of the following types:
  - pointer to `double`
  - pointer to `char`
  - pointer to `bool`
  - pointer to an enumerated type (`DayType`)
  - pointer to `struct`
- Two different ways of accessing a field of a struct variable by using a pointer to the struct:

  `(*p).fieldName` (cumbersome to use)

  or

  `p->fieldname` (the recommended way)

10.4.2.2 Additional notes and discussion on PTR_EX2.CPP

Although we don't show it here, we can also (of course) have pointers to class objects, a subject which we postpone till later. When we do have a pointer to a class object, we will see that the syntax for accessing the data (or function) member of the object is analogous to that used for accessing the fields of a struct.

10.4.2.3 Follow-up hands-on activities for PTR_EX2.CPP

□ Copy, study and test the program in PTR_EX2.CPP.

□ Make another copy of PTR_EX2.CPP called PTR_EX2A.CPP and bug the program as follows:

1. Remove the `&` in the statement "`dPtr = &cost;`".

   ---

2. Remove the `*` from `(*dayPtr == WED)`.

   ---

3. Remove the parentheses from `(*baPtr).idNum`.

◯ Instructor checkpoint 10.2

10.4.3 PTR_EX3.CPP illustrates the relationship between pointers and arrays

```
// Filename: PTR_EX3.CPP
// Purpose:  Illustrates the connection between pointers and arrays.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program illustrates pointers with arrays."
         << "\nStudy the source code and the output simultaneously.\n\n";

    int count[5] = { 2, 5, 9, 12, 20 };
    double real[4] = { -1.2, 3.5, 6.3, 9.1 };


    int* iPtr;
    iPtr = count;  // Equivalent to:  iPtr = &count[0];
    cout << endl;
    for (int i = 0; i < 5; i++)
    {
        cout << *iPtr << " ";
        iPtr++;
    }
    cout << endl;


    // Initialize dPtr to point at last array element:
    double* dPtr = &real[3];

    // Print out array values in reverse order:
    for (int j = 3; j >= 0; j--)
    {
        cout << *dPtr << " ";
        dPtr--;
    }
    cout << endl;

    // Illustrate some additional "pointer arithmetic":
    iPtr = &count[2];
    cout << *(iPtr + 2) << endl;
    cout << *(iPtr - 1) << endl;
    iPtr -= 2;
    cout << *iPtr       << endl;

    return 0;
}
```

10.4.3.1 What you see for the first time in PTR_EX3.CPP

- The initialization of an array pointer with the name of an array

  The net effect of doing this is to make the pointer point to the first element of the array, since the name of an array *is* the address of the first element of that array. Hence, in the following, with the given declaration, the two assignment statements are equivalent:

```
SomeType* p;

p = someArray;  // someArray has components of type SomeType.
p = &someArray[0];
```

- The initialization of an array pointer with the address of a particular array element (which makes the pointer point at that element)

- The use of *pointer arithmetic*

  This refers to the incrementing of pointer values, decrementing of pointer values, and combining pointer values with integers via addition or subtraction, all in the context of pointers to arrays. In each case, the pointer arithmetic is "smart", which is to say the amount by which the pointer value is changed by an increment, decrement, or the addition or subtraction of a given integer, depends on the component type of the array in question.

10.4.3.2 Additional notes and discussion on PTR_EX3.CPP

Let's say a bit more about the "smartness" of pointer arithmetic. What we mean to say is this: When we add 1 (for example) to a pointer that is pointing at an array element, the new pointer value (memory address) computed is not the next address in memory, but instead the address of the next array component, which may be several memory locations away. Just how "far away" it is (i.e., how many memory locations away) will depend on how many memory locations are occupied by a value of the component type of the given array. For example, if we make the assignment `p = &array[6]`, then `*(p+3)` refers to the 10^{th} element of `array`, irrespective of the component type of `array`.

10.4.3.3 Follow-up hands-on activities for PTR_EX3.CPP

□ Copy, study and test the program in PTR_EX3.CPP.

□ Make a copy of PTR_EX3.CPP and call it PTR_EX3A.CPP. Modify the copy so that it prints out the values of the integer array backwards and the values of the `double` array forwards.

◯ Instructor checkpoint 10.3

10.4.4 PTR_EX4.CPP illustrates pointers used in the context of arrays of structs

```
// Filename: PTR_EX4.CPP
// Purpose:  Illustrates pointers in the context of arrays of structs.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program illustrates pointers with arrays of structs."
         << "\nStudy the source code and the output simultaneously.\n\n";

    // Define a BankAccount struct:
    struct BankAccount
    {
        int idNum;
        char kind;
        double balance;
    };

    // Declare and initialize an array of three bank accounts:
    BankAccount account[3] = { {123, 'C', 199.99},
                               {456, 'S', 998.98},
                               {789, 'C', 321.45} };

    // Display the information in the three bank accounts via
    // the "usual" array component access using the array index:
    cout << endl;
    for (int j = 0; j < 3; j++)
    {
        cout << account[j].idNum   << " ";
        cout << account[j].kind    << " ";
        cout << account[j].balance << endl;
    }

    // Declare a BankAccount pointer which points to
    // the array of bank accounts:
    BankAccount* baPtr = account;

    // Display the information in the three bank accounts via
    // a BankAccount pointer which is incremented to point to
    // the next bank account after the current one has been displayed:
    cout << endl;
    for (int i = 0; i < 3; i++)
    {
        cout << baPtr->idNum   << " ";
        cout << baPtr->kind    << " ";
        cout << baPtr->balance << endl;
        baPtr++;
    }

    cout << endl;

    return 0;
}
```

10.4.4.1 What you see for the first time in PTR_EX4.CPP

- A pointer to an array of structs
- The use of pointer arithmetic in the context of an array of structs

10.4.4.2 Additional notes and discussion on PTR_EX4.CPP

Review the discussion about pointer arithmetic in the context of the previous sample program, and make sure that you understand how what was said there applies to the pointer arithmetic used in the current sample program.

10.4.4.3 Follow-up hands-on activities for PTR_EX4.CPP

□ Copy, study and test the program in PTR_EX4.CPP.

□ Make another copy of PTR_EX4.CPP and call it PTR_EX4A.CPP. Modify the copy so that it prints out the information in the array of structs backwards. Your code must make use of appropriate pointer arithmetic.

◯ Instructor checkpoint 10.4

10.4.5 PTR_EX5.CPP illustrates pointer parameters

```
// Filename: PTR_EX5.CPP
// Purpose:  Illustrates array/pointer parameters

#include <iostream>
using namespace std;

#include "PAUSE.H"

void MakeZero1(/* inout */ double a[], /* in */ int size)
// Pre:  "size" has been initialized.
// Post: All "size" values in a have been initialized to 0.0.
{
    for (int i = 0; i < size; i++) a[i] = 0.0;
}

void MakeZero2(/* inout */ double* a, /* in */ int size)
// Pre:  "size" has been initialized.
// Post: All "size" values in a have been initialized to 0.0.
{
    for (int i = 0; i < size; i++) a[i] = 0.0;
}

void PrintArray(/* in */ double* a, /* in */ int size)
// Pre:  "a" and "size" have both been initialized.
// Post: The values of "a" have been displayed, separated by one space.
{
    for (int i = 0; i < size; i++) cout << a[i] << " ";
}


int main()
{
    cout << "\nThis program illustrates pointers as function parameters."
         << "\nStudy the source code and the output simultaneously.\n\n";

    int i;
    double values[10];

    for (i = 0; i < 10; i++)
        values[i] = i*i;
    cout << endl;
    PrintArray(values, 10);  cout << endl;
    Pause(0);

    MakeZero1(values, 10);
    PrintArray(values, 10);  cout << endl;
    Pause(0);

    for (i = 0; i < 10; i++)
        values[i] = 2.5*i;
    cout << endl;
    PrintArray(values, 10);  cout << endl;
    Pause(0);

    MakeZero2(values, 10);
    PrintArray(values, 10);  cout << endl;
    Pause(0);

    return 0;
}
```

10.4.5.1 What you see for the first time in PTR_EX5.CPP

This program shows a function with a pointer parameter. The formal parameter has the syntax of a pointer variable declaration (as you would expect), while the corresponding actual parameter must be the address of a variable of the appropriate type.

10.4.5.2 Additional notes and discussion on PTR_EX5.CPP

Note that we have two versions of the function that initializes an array's values to zero, but that the body of each of those functions is *exactly* the same. How can this be? Well, it's just proof positive, if more was needed, that the name of an array really *is* a pointer to the beginning of that array or, to flip the coin, a pointer to the beginning element of an array can be used as a name for the array (which in turn can then have square brackets and indices used with *it*).

10.4.5.3 Follow-up hands-on activities for PTR_EX5.CPP

□ Copy, study and test the program in PTR_EX5.CPP. During your testing, change the `2` to a `1` in all instances of `MakeZero2` and try to compile. You will get a compile-time error (What is the error?), providing yet more proof that C++ really does consider the name of an array to be a pointer to the array's first element.

□ Make another copy of PTR_EX5.CPP and call it PTR_EX5A.CPP. Modify the copy to include two additional functions.

The first function is to be called `PrintArrayRange` and takes three in-parameters: the array, a first position, and a last position. It then prints out the values between (and including) the two positions, separated by a single blank space. And *do* make the parameters user-viewpoint *positions* (1, 2, 3, ...) rather than programmer-viewpoint *indices* (0, 1, 2, ...). Also, be sure to include test code in `main` that calls each function and demonstrates that the function is working properly.

The second function is to be called `PrintArrayReversed`, has the same parameter list as `PrintArray`, and prints out the values of the array in reverse order, again separated by a single blank space.

◯ Instructor checkpoint 10.5

10.4.6 PTR_SWAP.CPP compares pointer and reference parameters for swapping two values

```
// Filename: PTR_SWAP.CPP
// Purpose:  Illustrates the passing of pointer parameters.

#include <iostream>
using namespace std;

#include "PAUSE.H"


void Swap(/* inout */ int& a, /* inout */ int& b)
// Pre:  "a" and "b" have been initialized.
// Post: The values of "a" and "b" have been swapped.
{
    int temp = a;
    a = b;
    b = temp;
}


void Swap(/* inout */ int* ap, /* inout */ int* bp)
// Pre:  "ap" and "bp" contain the addresses of integer values.
// Post: The integer values in the locations pointed to by ap
//       have been swapped.
{
    int temp = *ap;
    *ap = *bp;
    *bp = temp;
}


int main()
{
    cout << "\nThis program show the connection between pointer parameters"
         << "\nand reference parameters."
         << "\nStudy the source code and the output simultaneously.\n\n";

    int m = 7;
    int n = -3;

    cout << endl;
    cout << "Original values of m and n:  " << m << " " << n << endl;
    Pause(29);

    // This call uses the Swap with reference parameters:
    Swap(m, n);
    cout << "Swapped values of m and n:   " << m << " " << n << endl;
    Pause(29);

    // This call uses the Swap with pointer parameters:
    Swap(&m, &n);
    cout << "And then swapped back again: " << m << " " << n << endl;
    Pause(29);

    return 0;
}
```

10.4.6.1 What you see for the first time in PTR_SWAP.CPP

This program shows an overloaded `Swap` function that does exactly the same job of switching the values of two integer variables in two different ways: once using reference parameters and once using pointer parameters.

10.4.6.2 Additional notes and discussion on PTR_SWAP.CPP

There are a couple of reasons you might find it interesting to look at this program and compare the two versions of the `Swap` function.

One reason is to reinforce again the close connection between the idea of a reference parameter (in which it is the address of the actual variable that is passed to the function when it is called) and the idea of a pointer parameter (where an address is also passed since a pointer is really just an address).

A second reason is this. As you know, the C++ language "descended" from the C language, which did not have reference parameters, and so when it was required to pass a parameter "by reference" it was *necessary* to use pointers. Since pointers are generally regarded as a tricky subject, you should be thankful you are studying C++ and not C, so that your encounters with pointers have been postponed until this late date, when you are so much more experienced.

10.4.6.3 Follow-up hands-on activities for PTR_SWAP.CPP

□ Copy, study and test the program in PTR_SWAP.CPP.

□ Make another copy of PTR_SWAP.CPP and call it BA_SWAP.CPP. Modify the copy so that the two values being swapped are no longer integers but values of the type `BankAccount` seen earlier. Revise the test code accordingly.

◯ Instructor checkpoint 10.6

10.4.7 PTR_THIS.CPP illustrates the this pointer of class objects

```
// Filenme: PTR_THIS.CPP
// Purpose: Illustrates the implicit "this" pointer in every class.

#include <iostream>
using namespace std;

class Time
{
public:
    void Set(/* in */ int hours,
             /* in */ int minutes,
             /* in */ int seconds);

    void Display() const;

private:
    int hours;
    int minutes;
    int seconds;
};


int main()
{
    cout << "\nThis program illustrates the \"this\" pointer."
         << "\nThe output is irrelevant, but you should study "
         << "\nthe source code carefully to see how \"this\" is used.\n";

    Time t;
    t.Set(11, 59, 50);
    t.Display();  cout << endl;

    return 0;
}

//*****************************************************************
void Time::Set(/* in */ int hours,
               /* in */ int minutes,
               /* in */ int seconds)
{
    this->hours = hours;     // Note the use of the "this" pointer.
    this->minutes = minutes;
    this->seconds = seconds;
}

//*****************************************************************
void Time::Display() const
{
    if (hours < 10)
        cout << '0';
    cout << hours << ':';
    if (minutes < 10)
        cout << '0';
    cout << minutes << ':';
    if (seconds < 10)
        cout << '0';
    cout << seconds;
}
```

10.4.7.1 What you see for the first time in PTR_THIS.CPP

This program uses a new version, and very stripped-down one, of the `Time` class to illustrate the `this` pointer. The keyword `this` is actually a new C++ reserved word.

10.4.7.2 Additional notes and discussion on PTR_THIS.CPP

The `this` pointer is a rather strange beast that lives inside every class object. Normally we don't need or want to use it, but sometimes it comes in very handy. The pointer `this` is implicitly available for any class object and is a pointer that points to that class object (a "pointer to self", if you like).

The way we have used it in this case allows us to use the same names for the input parameters of the `Set` member function as we used for the member variables for the class itself. This is not a big deal, and certainly not necessary (look back to Module 2 at how we handled the situation when we did not have the this pointer), but it does illustrate the way in which `this` is often used to refer to the members of an object.

10.4.7.3 Follow-up hands-on activities for PTR_THIS.CPP

□ Copy, study and test the program in PTR_THIS.CPP.

□ Make another copy of PTR_THIS.CPP called P_THIS.CPP and bug it as follows:

1. Remove `this->` from `this->hours`.

---

2. In the `Display` function add `this->` to every use of a member variable to show that, although not necessary in this case, it can be done.

◯ Instructor checkpoint 10.7

Module 11

Sorting and searching with one-dimensional arrays

11.1 Objectives

- To understand the *linear search algorithm*, and to understand why it must be used in preference to the *binary search algorithm* when the data is *not* sorted, or, more generally, when you know nothing at all about the data.
- To understand the *binary search algorithm*, and in particular to understand why it can only be used with *sorted* data.
- To understand the *selection sort algorithm*.
- To appreciate why the array data structure is convenient for holding data when you wish to apply the above algorithms to the data.
- To understand, from an operational point of view, how to use the library functions `rand`, `srand` and `clock` to help you generate *random values* in a given range.

11.2 List of associated files

- SSDEMO.CPP is a demo program illustrating the linear and binary search algorithms and the selection sort algorithm for data stored in a one-dimensional array.
- SSDEMO.DAT is the program information file to accompany SSDEMO.CPP, which is in turn based on SHELL.CPP.

11.3 Overview

In this Module we put arrays to work, holding data that we wish to process in various ways. In particular, we will wish to search through the data, looking for a particular value, and also to sort the data.

You will have an opportunity to examine in detail two search algorithms (*linear search* and *binary search*) and one sorting algorithm (*selection sort*). When you have finished this Module, you should have a very good high-level (pseudocode) view of each of these algorithms firmly fixed in your mind, so that you could implement any one of them in a different context at the drop of a hat, if necessary.

You should also be fully aware of when the *binary search algorithm* can be used (only with *sorted* data) and when you must fall back on the *linear search algorithm* (when nothing is known about the data, and in particular when the data is *not* sorted).

You should be aware that there are many other algorithms for both searching and sorting, that these algorithms often depend on how the data is stored (not all data is stored in arrays, as is the case in this Module) and often the software designer must choose carefully between using the various alternatives available for storing data and deciding which algorithms to use for manipulating the data, based on criteria that we will not discuss in detail here.

Another feature that we introduce here, because of its convenience, is that of randomly generating values to use for illustrating these algorithms. In other words, we use a random number generator to give us values to store in the array and to use for testing our algorithms. This is preferable to reading in the values from the keyboard or a file, since we really don't much care what the actual values are.

The library functions that we use to do this are also very useful in many other contexts, and you should know about them. You see them here from a purely operational point of view (they are simply shown doing what they do), but they are covered in more detail in Module 22. The three functions are:

`rand` which generates a random integer in the range `0...MAX_RAND`, where `MAX_RAND` is a built-in named constant

`srand` which determines the value used to begin the *random number generation* process by the `rand` function

`clock` which returns the integer giving the number of "clock ticks" since your process started running

Of course, once a random integer has been generated by a call to `rand`, it will usually need to be adjusted in some way to provide a number suitable for the job at hand, and this kind of transformation is also illustrated in the sample program of this Module.

11.4 Sample Programs and Other Files

11.4.1 SSDEMO.CPP and SSDEMO.DAT illustrate searching and sorting

```
// Filename: SSDEMO.CPP
// Purpose:  To demonstrate searching and sorting in arrays, and also
//           to demonstrate random generation of array elements.

#include <iostream>
#include <cstdlib>  // For access to the rand and srand functions
#include <ctime>    // For access to the clock function
using namespace std;

#include "MENU.H"
#include "TXITEMS.H"
#include "PAUSE.H"


const int MAX_SIZE = 20;
typedef int ComponentType;


void GenerateRandomValues(/* out */ ComponentType a[],
                          /* out */ int& currentSize);

void Display(/* in */ const ComponentType a[],
             /* in */ int currentSize);

void SortViaSelection(/* inout */ ComponentType a[],
                      /* in */ int currentSize);

void Search(/* in */ const ComponentType a[],
            /* in */ int currentSize);

void SearchLinear(/* in */  const ComponentType a[],
                  /* in */  ComponentType target,
                  /* in */  int first,
                  /* in */  int last,
                  /* out */ int& index,
                  /* out */ bool& found);

void SearchBinary(/* in */  const ComponentType a[],
                  /* in */  ComponentType target,
                  /* in */  int first,
                  /* in */  int last,
                  /* out */ int& index,
                  /* out */ bool& found);


int main()
{
    Menu m("Main Menu");
    m.AddOption("Quit");
    m.AddOption("Get information");
    m.AddOption("Generate a new array");
    m.AddOption("Display the array");
    m.AddOption("Sort the array in ascending order");
    m.AddOption("Search the array for a target value");

    TextItems ssDemo("ssdemo.dat");
```

```
    ComponentType a[MAX_SIZE];
    int currentSize = 0;
    int menuChoice;

    do
    {
        m.Display();
        menuChoice = m.Choice();
        switch (menuChoice)
        {
            case -1:
            case  1:
                break;

            case 2:
                ssDemo.DisplayItem("Program Description");
                break;

            case 3:
                GenerateRandomValues(a, currentSize);
                break;

            case 4:
                Display(a, currentSize);
                break;

            case 5:
                SortViaSelection(a, currentSize);
                break;

            case 6:
                Search(a, currentSize);
                break;
        }

    } while (menuChoice != 1  &&  menuChoice != -1);

    return 0;
}


void GenerateRandomValues(/* out */ ComponentType a[],
                          /* in */ int& currentSize)
// Pre:  none
// Post: currentSize has been entered by the user, and
//       "a" contains currentSize randomly chosen values.
{
    srand(clock()); // Establish a random "seed" for calling rand

    cout << "\nHow many array elements (no more than "
         << MAX_SIZE << ") would you like? ";
    cin >> currentSize;  cin.ignore(80, '\n');  cout << endl;

    cout << "What range would you like your values to lie in? " << endl;
    cout << "Enter first and last values for the range: ";
    ComponentType first, last;
    cin >> first >> last;  cin.ignore(80, '\n');  cout << endl;

    for (int i = 0; i < currentSize; i++)  // Get currentSize random values
        a[i] = first + rand() % (last - first + 1); // in range first..last.
}
```

```
void Display(/* in */ const ComponentType a[],
             /* in */ int currentSize)
// Pre:  "a" has been initialized.
// Post: The values of "a" have been displayed, with two blank
//       spaces separating each two values.
{
    cout << endl;
    for (int i = 0; i < currentSize; i++)
        cout << a[i] << "  ";
    cout << endl << endl;
    Pause(0);
}


void SortViaSelection(/* inout */ ComponentType a[],
                      /* in */ int currentSize)
// Pre:  "a" has been initialized.
// Post: The values in "a" have been sorted in ascending order.
{
    ComponentType temp;
    int minIndex;

    for (int pass = 0; pass < currentSize-1; pass++)
    {
        minIndex = pass;
        for (int position = pass+1; position < currentSize; position++)
            if (a[position] < a[minIndex]) minIndex = position;

        temp = a[minIndex];
        a[minIndex] = a[pass];
        a[pass] = temp;
    }
}


void Search(/* in */ const ComponentType a[],
            /* in */ int currentSize)

// Pre:  "a" has been initialized.
// Post: A search for a "target" value in "a" has been performed.
{
    cout << endl;
    cout << "What value would you like to search for? ";
    ComponentType target;
    cin >> target;  cin.ignore(80, '\n');  cout << endl;

    cout << "Between what two positions would you like to search? ";
    int first, last;
    cin >> first >> last;  cin.ignore(80, '\n');  cout << endl;

    cout << endl;
    cout << "You have a choice of linear or binary search. " << endl;
    cout << "But, you can only use binary "
         << "search on a sorted list. "     << endl
         << "Which would you like to use? (L/B) ";
    char response;
    cin >> response;  cin.ignore(80, '\n');  cout << endl;

    bool found;
    int index;
```

```
    switch (response)
    {
        case 'L':
        case 'l':
            SearchLinear(a, target, first, last, index, found);
            break;

        case 'B':
        case 'b':
            SearchBinary(a, target, first, last, index, found);
            break;
    }

    cout << endl << target;

    if (found)
        cout << " located at position " << index+1 << "."  << endl;
    else
        cout << " not found." << endl;
    cout << endl;

    Pause(0);
}



void SearchLinear(/* in */  const ComponentType a[],
                  /* in */  ComponentType target,
                  /* in */  int first,
                  /* in */  int last,
                  /* out */ int& index,
                  /* out */ bool& found)
// Pre:  "a", "target", "first" and "last" have been initialized.
//       1 <= first <= last <= MAX_SIZE.
// Post: "found" == true and a[index] == "target", or
//       "found" == false and "index" == "last".
{
    index = first-1;
    while (index <= last-1  &&  target != a[index])
        ++index;
    found = (index < last);
}



void SearchBinary(/* in */  const ComponentType a[],
                  /* in */  ComponentType target,
                  /* in */  int first,
                  /* in */  int last,
                  /* out */ int& index,
                  /* out */ bool& found)
// Pre:  "a", "target", "first" and "last" have been initialized.
//       1 <= first <= last <= MAX_SIZE.
//       The values in "a" must be sorted.
// Post: "found" == true and a[index] == "target", or
//       "found" == false and "index" is undefined.
{
    int low = first;
    int high = last;
    int middle;

    found = false;
```

```
    while (high >= low  &&  !found)
    {
        middle = (low + high)/2;
        if (target < a[middle])
            high = middle - 1;
        else if (target > a[middle])
            low = middle + 1;
        else
            found = true;
    }

    index = middle;
}
```

Contents of the file SSDEMO.DAT are shown between the heavy lines:

```
Program Description
This program allows the user to generate an array of data values
(currently int values) but that can be changed by changing the
definition of "ComponentType" in the typedef statement and making
any other required changes to deal with the resulting data type.
Of course the program will then need to be re-compiled and re-linked.

The maximum number of data values is currently fixed at 20, but
that too could be altered, just by changing the value of MAX_SIZE.
Once again, re-compilation and re-linking would be necessary.

The data values are generated randomly within a range of values
determined by the user.

After the generated data values have been stored in the array they can be

a) displayed

b) sorted

c) searched (for a particular target value), using either
   - a linear search, or
   - a binary search (but only if the values have been previously sorted)

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
-------------------------------------------------------------------------------
```

11.4.1.1 What you see for the first time in SSDEMO.CPP

- An implementation of the linear search algorithm
- An implementation of the binary search algorithm
- An implementation of the selection sort algorithm
- The use of `rand`, `srand` and `clock` to generate random values in a given range of values for use as test data

11.4.1.2 Additional notes and discussion on SSDEMO.CPP

Note that we have said that you see in this program *an* implementation of each of these algorithms, not *the* implementation of the algorithms. This wording is deliberate, and meant to suggest that the algorithm is independent of any particular implementation of it. That's why it's so important to understand each such algorithm "at the pseudocode level" so that you can recognize and/or implement the algorithm when necessary in other situations.

This program also presents another example of those situations in which the user will not necessarily have the same view of something as the programmer. For example, the programmer knows that the elements of an array are "numbered" (i.e., indexed) starting from 0, but the average user is going to think of the array values as being numbered starting from 1, so the prompts to the user must be worded from the user's viewpoint, not the programmer's.

This program is another good example of how easy it is to avoid a lot of tedious effort if you simply make appropriate use of the `Menu` and `TextItems` classes.

Finally, note that there is a "higher-level" `Search` routine that interacts with the user and lets the (intelligent) user decide which of the two possible "lower-level" search routines to use, based on the user's knowledge of the current data.

11.4.1.3 Follow-up hands-on activities for SSDEMO.CPP

□ Copy, study and test the program in SSDEMO.CPP. Be sure to generate several different arrays of values of different sizes and with values from different ranges, and be sure to exercise all of the options the program provides.

□ Make a copy of SSDEMO.CPP called SSCHAR.CPP and revise the copy as follows:

1. Change the size limit of the array of data values from 20 to 30.
2. Make the data values capital letters instead of integers.
3. Permit the user to sort in descending as well as ascending order.
4. Be sure to make all the necessary associated changes in the prompts, the new .DAT file (call it SSCHAR.DAT), and so on.

Test your revised program thoroughly to make sure it is working properly.

◯ Instructor checkpoint 11.1

Module 12

Multi-dimensional arrays with a simple component data type (our fourth structured data type)

12.1 Objectives

- To compare and contrast *multi-dimensional arrays* (and in particular, *two-dimensional arrays*) with one-dimensional arrays.
- To understand how to define a two-dimensional array data type, and then declare and use a variable of this type.
- To learn how to access some or all of the elements in a two-dimensional array in various ways, and in particular how to access:
  - Any particular single element
  - All the elements in the array, in either *row-wise* order or *column-wise* order
  - All the elements in a particular row or column
  - All the elements on either of the diagonals, if the array is square
  - All the elements in a particular (rectangular or square) sub-array
- To understand the issues involved in using a two-dimensional array as a function parameter.

12.2 List of associated files

- TWODIMA1.CPP illustrates declaration, initialization and processing of a two-dimensional array.
- TWODIMA2.CPP illustrates the passing of a two-dimensional array as a function parameter.

12.3 Overview

In this Module we look at two-dimensional arrays in some detail and mention briefly higher-dimensional arrays, all of which may be collectively referred to as multi-dimensional arrays.

In Module 9 we introduced the concept of an array, but dealt only with one-dimensional arrays, which meant that we were dealing with data for which a single index value was sufficient to locate or identify the data value.

Sometimes data needs to be indexed by two (or more) index values. For example if our data represents temperatures recorded over some geographical region over some period of time, then we need both a place index and a time index to locate a particular temperature for that place and time. In fact, we often encounter "tables" of information, i.e., information arranged into (horizontal) rows and (vertical) columns, and we refer to any particular value in the table by its row number (row index) and column number (column index). Representing data like this is the job of a two-dimensional array, and such arrays are the topic of interest in the sample programs of this Module.

Of course, there are many situations where three or more indices are necessary or useful as well, and once we see how two-dimensional arrays are defined and manipulated, it is a relatively short step to arrays of higher dimension. However, we limit ourselves to the two-dimensional case in this Module. As a typical example of where a three-dimensional array would be useful we might mention the problem of keeping track of an inventory of (say) women's slacks, which we might index by size, color and material.

12.4 Sample Programs

12.4.1 TWODIMA1.CPP illustrates declaration, initialization and processing of a two-dimensional array

```
// Filename: TWODIMA1.CPP
// Purpose:  Illustrates declaration, initialization and
//           processing of a two-dimensional array.

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

int main()
{
    cout << "\nThis program illustrates a two-dimensional array."
         << "\nStudy the source code and the output simultaneously.\n\n";

    // Declare *and* initialize a two-dimensional array
    const NUM_ROWS = 3;
    const NUM_COLS = 5;
    int a[NUM_ROWS][NUM_COLS] =
        { {  2,  4,  6,  8, 10 },
          { -1, -2 }, // Remaining values in 2nd row will be 0.
          {  1,  3,  5,  7,  9 }
        };

    int row, col;  // For use as row and column indices

    // Display all values in the array:
    cout << endl;
    for (row = 0; row < NUM_ROWS; row++)
    {
        for (col = 0; col < NUM_COLS; col++)
            cout << setw(4) << a[row][col];
        cout << endl;
    }

    const int SIZE = 3;
    int b[SIZE][SIZE];

    // Initialize a two-dimensional array with nested for loops:
    for (row = 0; row < SIZE; row++)
        for (col = 0; col < SIZE; col++)
            b[row][col] = row + col;  // Just some value

    // Display all values in the array again:
    cout << endl;
    for (row = 0; row < SIZE; row++)
    {
        for (col = 0; col < SIZE; col++)
            cout << setw(4) << b[row][col];
        cout << endl;
    }

    // Display values in 2nd row, followed by 3rd columns:
    cout << endl;
    for (col = 0; col < SIZE; col++)
        cout << setw(4) << b[1][col];
    cout << endl;
    for (row = 0; row < SIZE; row++)
        cout << setw(4) << b[row][2];
    cout << endl;
```

```
    // Display values on main diagonal:
    for (row = 0; row < SIZE; row++)
        cout << setw(4) << b[row][row];
    cout << endl;

    // Display values on minor diagonal:
    for (row = SIZE-1; row >= 0; row--)
        cout << setw(4) << b[row][SIZE-1 - row];
    cout << endl << endl;

    return 0;
}
```

12.4.1.1 What you see for the first time in TWODIMA1.CPP

- The declaration and initialization of a two-dimensional array

  Note, in the declaration, the *two* pairs of square brackets, with the number of *rows* of the array within the *first* set and the number of *columns* of the array within the *second* set. Note too that the values used for the initialization of each row are contained within an inner set of braces, with "missing" values being replaced by 0 (as for one-dimensional arrays).

- The use of for-loops or nested for-loops to process some or all of the values in the array

  For-loops tend to be used when processing arrays since it is usually the case that we are processing either all the elements in the array, or, in the case of a two-dimensional array, all the elements in a row or column or on a diagonal. This kind of processing lends itself readily to the use of a for-loop since in situations like this we know exactly how many elements we are dealing with.

12.4.1.2 Additional notes and discussion on TWODIMA1.CPP

This program illustrates the processing of various portions of the given array, but keep in mind that when processing two- or multi-dimensional arrays there is much more variety in the ways one can process the elements than there is in the case of one-dimensional arrays. Each case needs to be considered carefully on its own to make sure the required processing is being performed correctly.

12.4.1.3 Follow-up hands-on activities for TWODIMA1.CPP

□ Copy, study and test the program in TWODIMA1.CPP.

□ Make a copy of TWODIMA1.CPP called TWODA1A.CPP and add code to display the array `a` in such a way that its rows appear as columns and its columns appear as rows.[1]

◯ Instructor checkpoint 12.1

[1] In mathematical terms, you are displaying the *transpose* of the array `a`.

12.4.2 TWODIMA2.CPP illustrates the passing of a two-dimensional array as a function parameter

```
// Filename: TWODIMA2.CPP
// Purpose:  Illustrates passing a two-dimensional array as a parameter.

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

void FindMaxRowVals(/* in */  const int a[][5],
                    /* out */ int maxRowVals[],
                    /* in */  int numRows);

int main()
{
    cout << "\nThis program illustrates passing a "
         << "two-dimensional array as a function "
         << "\nparameter.  Study the source code "
         << "and the output simultaneously.\n\n";

    int a[3][5] = { {  2,  4,  6,  8, 10 },
                    { -1, -2 },
                    {  7,  9,  5,  1,  3 }
                  };

    int maxValues[3];

    FindMaxRowVals(a, maxValues, 3);

    int row;
    cout << endl;
    for (row = 0; row < 3; row++)
    {
        cout << "The maximum value in row "
             << row << " is " << maxValues[row] << "." << endl;
    }
    cout << endl;

    return 0;
}


void FindMaxRowVals(/* in */  const int a[][5],
                    /* out */ int maxRowVals[],
                    /* in */  int numRows)
// Pre:  "a" and "numRows" have been initialized.
// Post: "maxRowVals" contains the maximum values in the rows of "a",
//       with the maximum value in row 0 of the (two-dimensional) array
//       "a" in component 0 of the (one-dimensional) array "maxRowVals",
//       and so on for the other rows of "a" and "maxRowVals".
{
    int row, col, max;
    for (row = 0; row < numRows; row++)
    {
        max = a[row][0];
        for (col = 1; col < 5; col++)
        {
            if (a[row][col] > max)
                max = a[row][col];
        }
        maxRowVals[row] = max;
    }
}
```

12.4.2.1 What you see for the first time in TWODIMA2.CPP

This program shows how to use a two-dimensional array as a function parameter, and you should make the following observations:

- In the formal parameter list there is *no value* within the *first* set of square brackets of the two-dimensional array parameter, but there *is* a value within the *second* set of square brackets.

  This generalizes to multi-dimensional arrays of dimension three or higher as follows: The first set of square brackets contains no value, but each subsequent set of square brackets must contain a value.

- The actual two-dimensional array parameter in the function call is, once again, just as it was for one-dimensional arrays, the name of the array being passed.

- As with one-dimensional arrays, the function must either "know" how big the array is that has been passed, or information giving the size of the array must also be passed along with the array.

- As with any structured data type, if a two-dimensional array parameter is a conceptual in-parameter, it has a `const` modifier.

12.4.2.2 Additional notes and discussion on TWODIMA2.CPP

The reason for the missing value within the square brackets is analogous to the reason for the missing value in the case of the single set of square brackets with one-dimensional array parameters. In all cases, with array parameters, we pass the name of the array, which is just a pointer to the first element. Two-dimensional arrays are stored in memory in a row-wise fashion, i.e., first the first row, then the second row, and so on. There can be as many rows as you like (hence no value in the first set of brackets) but there must be information as to how many elements in a row so the compiler can tell how the information is laid out in memory. The reason that there can be as many rows as you like is that in C++ it is the *programmer's responsibility* to know this, *not* the compiler's!

The compiler will not *check the number of elements in your array.*

12.4.2.3 Follow-up hands-on activities for TWODIMA2.CPP

□ Copy, study and test the program in TWODIMA2.CPP.

□ Make a copy of TWODIMA2.CPP and call it TWODA2A.CPP. Add to this copy a value-returning function called `NumberOfOddVals` that has two in-parameters (the same as the first and third parameters of `FindMaxRowVals`) and returns the number of odd values found in its array in-parameter, *excluding all values on the border of that array.* In the body of this function use a pointer rather than an array index to access the array components.

◯ Instructor checkpoint 12.2

Module 13

C-style strings (our fifth structured data type)

13.1 Objectives

- To understand that a *C-style string variable* is an array of type `char`, and that
  - The array has a fixed size and it's the *programmer's responsibility* not to exceed that limit by inserting more characters than the array will hold.
  - The last character of the actual string must always be followed by a *null character* (`'\0'`). This character must be included in the count when determining the number of characters the array will hold, but it is not regarded as one of the characters in the string. The null character is not displayed in any way when the C-string is output; it is simply used as a marker for the end of the C-string so that functions that deal with the C-string know where the actual string ends.
- To learn how `cin` behaves when reading a string value into a C-string variable.
- To learn how to read a string value into a C-string variable with both `cin.get` and `cin.getline`.
- To learn how to write out a C-string with `cout`.
- To understand that the behavior of file streams is analogous to that of `cin` and `cout` when C-strings are being read from an input file stream or written to an output file stream.

- To understand that a C-string variable may be declared and also *initialized* at the time of declaration, but that it may *not* have a value assigned to it *after* it has been declared. Such a variable may, however, have a value *copied into* it, which accomplishes the same effect.
- To understand how to
  - Copy a constant string value into a C-string variable, or the value of one such variable into another using the `strcpy` function
  - Concatenate (join) two C-string values using the `strcat` function
  - Find the length of a C-string using the `strlen` function
  - Compare (in the alphabetic sense) two C-strings using the `strcmp` function

13.2 List of associated files

- CSTR1.CPP illustrates declaration, initialization, copying and concatenation of C-style strings, as well as computing their lengths and writing them out with `cout`.
- CSTR2.CPP illustrates C-string comparisons and more input/output.
- CSTR3.CPP illustrates use of `cin.get` and `cin.getline`.
- CSTR4.CPP illustrates the use of an explicit delimiter character with `cin.getline`.
- CSTR5.CPP illustrates reading lines from a textfile, and points out some potential pitfalls.
- STRING1.DAT is a data file for use with CSTR5.CPP.
- STRING2.DAT is a second data file for use with CSTR5.CPP.
- CSTRPTR.CPP illustrates the use of a "pointer to `char`" with C-strings.

13.3 Overview

In this Module we finally deal in some detail with strings in ways that we have, for the most part, been avoiding up to now. That is, we look at reading in strings, modifying strings, computing their lengths, and printing them out again.

There are quite a number of sample programs[1] to illustrate these various operations and you will do well to spend some time experimenting with them.

[1] See also Appendix G and sections 22.4.3 and 22.4.4 of Module 22 for more on strings.

13.4 Sample Programs and Other Files

13.4.1 CSTR1.CPP illustrates basic C-string features

```
// Filename: CSTR1.CPP
// Purpose:  Illustrates C-style strings: declaration, initialization,
//           copying, concatenation, computing length, output with cout.

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

#include "PAUSE.H"

int main()
{
    cout << endl;
    cout << "This program illustrates declaration, "
         << "initialization, copying, concatenation, "   << endl
         << "and output with cout, of C-strings. "
         << "Study the source code while running it. "   << endl;
    cout << endl;
    Pause(0);

    // You can initialize a C-string variable in its declaration:
    char s1[8] = "Hello, "; // OK
    char s2[] = "world!";   // OK, and also avoids character counting!

    // But you can't assign to a C-string variable after declaring it:
    // char s3[21];
    // s3 = "Hello, world!";  // Not OK; gives a compile-time error

    // So, you have to "copy", *not* assign, one string to another:
    char s3[21];
    strcpy(s3, s1);  // Note that the copying is *from* s1 *to* s3.

    // This is how we append string s2 to the end of string s3:
    strcat(s3, s2);

    // Now we print out the three strings and their lengths:
    cout << "Length of \"" << s1 << "\" is " << strlen(s1) << "." << endl;
    cout << "Length of \"" << s2 << "\" is " << strlen(s2) << "." << endl;
    cout << "Length of \"" << s3 << "\" is " << strlen(s3) << "." << endl;
    Pause(0);

    // We can also use "typedef" to define a "C-string type", as in
    typedef char String20[21];  // The "extra" location is for the '\0'.

    // Note that strcpy may also be treated as a value-returning
    // function that actually returns the string result of the copy.
    String20 s4, s5;
    cout << strcpy(s4, "How are you?") << "<<" << endl; // Here we are
    cout << strcpy(s5, "Fine!")        << "<<" << endl; // displaying the
    Pause(0);                          // return-value of function strcpy.

    // Note that copying a shorter string to a longer string
    // "does the right thing" and puts the '\0' in the right place.
    cout << s4 << "<<" << endl;
    strcpy(s4, s5);
    cout << s4 << "<<" << endl;
    Pause(0);

    return 0;
}
```

13.4.1.1 What you see for the first time in CSTR1.CPP

- The initialization of a C-string variable at the time of declaration with the value of a quoted string
- The use of the following functions from the `ctring` library:

  `strcpy` copies its *second* argument `to` its *first* argument. Note the "direction" in which the copying takes place.

  `strcat` adds its *second* argument to the end of its *first* argument (i.e., it *concatenates* the two arguments and the first argument is actually replaced by the result of the concatenation)

  `strlen` computes the length (number of characters) of its argument (the `'\0'` character is not included in the character count)

- The display of C-strings with `cout`

  The way `cout` works with C-strings is this: All characters in the string are simply displayed in the order in which they occur in the string, with no whitespace inserted either before or after.

- The use of the naming convention illustrated by the definition of `String20` using `typedef`

  That is, String20 (for example) denotes a `char` array of size 21 (*not* 20, note) that can hold up to 20 characters *plus* the terminating null character.

13.4.1.2 Additional notes and discussion on CSTR1.CPP

Note that the functions `strcpy` and `strcat`, in addition to performing the copy or concatenation, also return the address of the result of the copy or the concatenation, as the case may be. This may or may not be useful, but it is illustrated in the sample program where we display the "function value" of `strcpy` (as in `cout << strcpy(s5, "Fine!")`, for example).

13.4.1.3 Follow-up hands-on activities for CSTR1.CPP

□ Copy, study and test the program in CSTR1.CPP. Among the other tests you may wish to perform, replace the statement that initializes `s1` with the following statement to show that it is equivalent:

```
char s1[8] = {'H', 'e', 'l', 'l', 'o', ',', ' ', '\0'};
```

This is precisely analogous to the way we have previously initialized arrays, but it also shows why we are grateful that C++ allows us to initialize string variables using string constants (quoted strings) as shown in the sample program.

◯ Instructor checkpoint 13.1

13.4.2 CSTR2.CPP illustrates C-string comparisons and input with cin

```
// Filename: CSTR2.CPP
// Purpose:  Illustrates C-style strings: input with cin, comparison,
//           and output with cout.

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

#include "PAUSE.H"

int main()
{
    cout << endl;
    cout << "This program illustrates input with cin, "
         << "comparison, and more output with "         << endl
         << "cout, of C-strings.  Study the source "
         << "code while running it.  In response to "   << endl
         << "each prompt, enter either two strings, "
         << "each containing 10 or fewer characters, "  << endl
         << "or an end-of-line to quit. "               << endl;
    cout << endl;
    Pause(0);  cout << endl;

    typedef char String10[11];
    String10 s1, s2;

    cout << "Enter the two strings below: " << endl;
    cin  >> s1 >> s2;  cin.ignore(80, '\n');

    while (cin)
    {
        cout << "The two strings entered and read "
             << "are on the next two lines:"        << endl;
        cout << s1 << "<<" << endl;
        cout << s2 << "<<" << endl;
        Pause(0);  cout << endl;

        // The following three if-statements, written separately here to
        // show each of the three possibilities in full, would normally
        // be written as a nested if with an else at the end.
        if (strcmp(s1, s2) == 0)
            cout << "Those two strings were the same.";
        if (strcmp(s1, s2) < 0)
            cout << "The 1st string precedes the "
                 << "2nd string, alphabetically.";
        if (strcmp(s1, s2) > 0)
            cout << "The 2nd string precedes the "
                    "1st string, alphabetically.";
        cout << endl << endl;
        Pause(0);  cout << endl;

        cout << "Enter the two strings below: " << endl;
        cin  >> s1 >> s2;  cin.ignore(80, '\n');
    }

    cout << endl;

    return 0;
}
```

13.4.2.1 What you see for the first time in CSTR2.CPP

- The reading of a string value into a C-string variable using `cin`

  The thing to remember here is that in this context `cin` behaves "as usual". That is, it ignores leading whitespace, the reading is stopped by the first whitespace character it encounters after the reading of characters starts, and that whitespace character is left in the input stream. This implies that you *cannot* read a string containing whitespace using `cin`, and this is in fact the case.

- The use of the `strcmp` function from the `cstring` library to determine whether one string is the same as another, or if it comes before or after the other (in the alphabetical sense)

  Note that `s1` is the same as `s2`, precedes `s2`, or follows `s2`, according as the value of `strcmp(s1,s2)` is `== 0`, `< 0`, or `> 0`.

13.4.2.2 Additional notes and discussion on CSTR2.CPP

Note that the `strcmp` function is necessary for C-strings, precisely because the `==`, `<` and `>` operators are not overloaded for C-strings.

13.4.2.3 Follow-up hands-on activities for CSTR2.CPP

□ Copy, study and test the program in CSTR2.CPP.

□ This activity is designed to help you confirm that when reading strings from an input stream, the behavior is the same whether the reading is taking place from the keyboard with `cin` or from a file via a file input stream, say `inFile`. So, make a copy of CSTR2.CPP called CSTR2A.CPP and revise the copy so that the two string variables s1 and s2 have values read into them from a file called CSTR2A.DAT. Create your own string data for entry into CSTR2A.DAT. Experiment by changing the values in this file and re-running the program.

◯ Instructor checkpoint 13.2

13.4.3 CSTR3.CPP illustrates get and getline with C-strings

```
// Filename: CSTR3.CPP
// Purpose:  Illustrates input of C-style strings with cin.get and
//           with cin.getline.

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

#include "PAUSE.H"

int main()
{
    cout << endl;
    cout << "This program illustrates input of C-strings "
         << "with cin.get, as well as input "               << endl
         << "of C-string \"lines\" with cin.getline.  "
         << "Study the source code carefully while "        << endl
         << "running it.  Note that you get to choose "
         << "the string size, or the line size, "           << endl
         << "on each pass.  When in one of the string-"
         << "reading loops, at each pause you "             << endl
         << "may either press ENTER to continue or "
         << "enter an end-of-file to quit. "                << endl;
    cout << endl;
    Pause(0);  cout << endl;

    char ch;

    const int MAX_STR_SIZE = 10;
    typedef char String10[MAX_STR_SIZE + 1];
    String10 s1, s2;

    int numChars;

    do
    {
        cout << "Now reading with cin.get ..." << endl;
        cout << "Enter the string size (<= " << MAX_STR_SIZE
             << ") for this pass: ";
        cin >> numChars;  cin.ignore(80, '\n');

        cout << "Enter the strings below, on separate lines:" << endl;
        cin.get(s1, numChars + 1);  cin.get(ch);
        cin.get(s2, numChars + 1);  cin.get(ch);

        cout << "The two strings are on the next two lines:" << endl;
        cout << s1 << "<<" << endl;
        cout << s2 << "<<" << endl;
        Pause(0);  cout << endl;

    } while (cin);

    cout << endl;

    cin.clear();


    do
    {
        cout << "Now reading with cin.getline ..." << endl;
        cout << "Enter the string size (<= " << MAX_STR_SIZE
             << ") for this pass: ";
        cin >> numChars;  cin.ignore(80, '\n');
```

```
        cout << "Enter the two lines below: "   << endl;
        cin.getline(s1, numChars + 1);
        cin.getline(s2, numChars + 1);

        cout << "Here are the two lines:" << endl;
        cout << s1 << "<<" << endl;
        cout << s2 << "<<" << endl;

        Pause(0);  cout << endl;

    } while (cin);

    cout << endl;

    return 0;
}
```

13.4.3.1 What you see for the first time in CSTR3.CPP

- The use of `cin.get` to read a string into a C-string variable
- The use of `cin.getline` to read a string into a C-string variable

  Both of these functions work with input streams other than `cin`, and work as follows: Reading starts with the next character in the input stream (whether or not that character is a whitespace character). The input stream is the standard input stream `cin` in this program, but analogous behavior applies if we are reading from a file input stream as well. Reading stops when `numChars` characters have been read, or when a newline character is encountered, whichever comes first. (Check the program to see how `numChars` is used.). The reason for the "`+ 1`" following `numChars` is to allow for the null character `'\0'`, which terminates every C-string.

  So, what's the difference between `get` and `getline`? Just this: If the read is terminated by a newline character, then `get` leaves that newline character in the input stream, while `getline` removes the newline character from the input stream. In neither case is the newline character placed in the string that is being read.

13.4.3.2 Additional notes and discussion on CSTR3.CPP

Finding out how `get` and `getline` *really* work is not something you find out by reading the above discussion. That's just the start. Then you experiment, re-read, experiment again, ...

13.4.3.3 Follow-up hands-on activities for CSTR3.CPP

□ Copy, study and test the program in CSTR3.CPP with input data of your own choosing, but be sure to try the same input data with different values of `numChars` and `MAX_STR_SIZE`. Changing `MAX_STR_SIZE` will of course require both changing `String10` and re-compiling.

◯ Instructor checkpoint 13.3

13.4.4 CSTR4.CPP illustrates use of an explicit delimiter character with getline

```
// Filename: CSTR4.CPP
// Purpose:  Illustrates explicit delimiter with
//           cin.getline and C-strings.

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

#include "PAUSE.H"

int main()
{
    cout << endl;
    cout << "This program allows the user to choose "
         << "both the maximum number of characters "         << endl
         << "to read per string for each of two strings, "
         << "as well as the delimiter character "            << endl
         << "which terminates the reading of each string. "  << endl;
    cout << endl;

    const int MAX_STR_SIZE = 10;
    typedef char String10[MAX_STR_SIZE + 1];
    String10 s1, s2;


    int numChars;
    cout << "Enter the number (<= " << MAX_STR_SIZE
         << ") of characters to read: ";
    cin >> numChars;
    cout << endl;  cin.ignore(80, '\n');

    char delimiter;
    cout << "Enter the delimiter character: ";
    cin.get(delimiter);
    if (delimiter != '\n')
        cin.ignore(80, '\n');
    cout << endl;

    cout << "Enter some text: " << endl;
    cin.getline(s1, numChars+1, delimiter);
    cin.getline(s2, numChars+1, delimiter);

    cout << endl;
    cout << "Here are the two strings as read "
         << "(\"<<\" points to the end of each):"   << endl;
    cout << s1 << "<<" << endl;
    cout << s2 << "<<" << endl;

    cout << endl;

    return 0;
}
```

13.4.4.1 What you see for the first time in CSTR4.CPP

This program also illustrates the use of `getline`, but this time with an explicit delimiter character entered by the user. In the previous sample program we were using the *default delimiter* for `getline`, namely the newline character. What we see here is that in fact you can use whatever character you like for the delimiter, which means, among other things, that if you use some character other than the default newline character for the delimiter, then the newline character is just another character, and hence can itself be read as one of the characters in a string.

So, how exactly does the read work in this case. Well, once again no whitespace is skipped, and the reading starts with the very next character in the input stream, whitespace or not. The reading stops either when the number of characters read is `numChars` (as in the previous program) or when the delimiter character is read, whichever comes first. As before, `getline` removes the delimiter character from the input stream, but does *not* store it in the string variable.

13.4.4.2 Additional notes and discussion on CSTR4.CPP

A subtle point to note: If exactly `numChars` are read by `cin.getline` and the very next character is the delimiter character, that delimiter character is *not* removed from the input stream.

Also, although this program only shows the user entering his or her own delimiter for `getline`, the same could be done for `get`, which (as before) does not remove the delimiter character from the input stream. And as you probably suspect by now, if `inFile` is an input file stream, then `inFile.get` and `inFile.getline` would behave analogously to `cin.get` and `cin.getline`.

13.4.4.3 Follow-up hands-on activities for CSTR4.CPP

□ Copy, study and test the program in CSTR4.CPP. Among other tests you make, try the following input:

```
10
*
This *is it.
This is also it.
```

If you understand exactly why the output is

```
This <<
is it.
Thi<<
```

you may be well on your way to understanding the point of this sample program, but don't let that stop you from additional testing.

◯ Instructor checkpoint 13.4

13.4.5 CSTR5.CPP, STRING1.DAT and STRING2.DAT illustrate reading lines from a textfile

```
// Filename: CSTR5.CPP
// Purpose:  Reads and displays the lines from a textfile.
//           Illustrates run-time input of filenames.
//           Try the values 81 and then 80 for "numChars" and
//           explain the difference in behavior when the input
//           file contains lines of exactly 80 characters.

#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;

#include "PAUSE.H"

int main()
{
    cout << endl;
    cout << "This program asks the user for the name "
         << "of a textfile and then displays the "     << endl
         << "lines from that file.  It then repeats "
         << "the entire process one more time. "       << endl;
    cout << endl;

    ifstream inFile;
    typedef char String80[81];
    String80 line, fileName;

    int numChars;
    cout << "Enter the maximum number of characters per line: ";
    cin >> numChars;  cin.ignore(80, '\n');  cout << endl;

    cout << "Enter name of first file to read: ";
    cin >> fileName;  cin.ignore(80, '\n');  cout << endl;
    inFile.open(fileName);
    cout << "Here are the lines in the file " << fileName << ":\n";
    inFile.getline(line, numChars + 1);
    while (inFile)
    {
        cout << line << endl;
        inFile.getline(line, numChars + 1);
    }
    inFile.close();  Pause(0);  cout << endl;

    cout << "Enter name of second file to read: ";
    cin >> fileName;  cin.ignore(80, '\n');  cout << endl;
    inFile.clear();
    inFile.open(fileName);
    cout << "Here are the lines in the file " << fileName << ":\n";
    inFile.getline(line, numChars + 1);
    while (inFile)
    {
        cout << line << endl;
        inFile.getline(line, numChars + 1);
    }
    inFile.close();  Pause(0);  cout << endl;

    return 0;
}
```

Contents of the file STRING1.DAT are shown between the heavy lines:

```
This is the first line of the file.
This is the second.
And now the third and last.
```

Contents of the file STRING2.DAT are shown between the heavy lines:

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

And here is line 3 following a blank line.
1       0
Finally, the 5th and last line.
```

13.4.5.1 What you see for the first time in CSTR5.CPP

This program does not contain anything new. It simply illustrates the reading and displaying of the lines of a textfile using the `getline` function. Unfortunately this function does not appear to behave consistently and/or correctly on all systems[2].

It seems to help if you attempt to read, for every line, *more than* the maximum number of characters that might be on any line, but this is not how it's supposed to be. For example, if you are reading lines from a textfile, and the lines may be up to 80 characters in length, then you should (maybe?) try to read 80 characters each time you try to read a line.

13.4.5.2 Additional notes and discussion on CSTR5.CPP

Experimentation is the only way to find out how things work on your system. A better approach (nowadays) is (or should be, but again you cannot yet be sure things will work as they should) to go with C++ string objects as presented in Module 14.

13.4.5.3 Follow-up hands-on activities for CSTR5.CPP

□ Copy, study and test the program in CSTR5.CPP. Try both STRING1.DAT and STRING2.DAT as input files, and try at least the values 81, 80 and 70 for the maximum number of characters you want to read. If you should be so lucky (or unlucky) to be able to try all three systems mentioned in the footnote below, these two files and three integer values are sufficient to illustrate both the correct and incorrect behavior mentioned.

◯ Instructor checkpoint 13.5

[2]The author has tried gnu C++ under Linux, Visual C++ under Windows NT, and DEC C++ under VMS and at the time of writing the only one that worked like the Standard said it should was DEC C++ under VMS. Go figure ...

13.4.6 CSTRPTR.CPP illustrates use of a char pointer with C-strings

```
// Filename: CSTRPTR.CPP
// Purpose:  Illustrates the connection between pointers and C-strings.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program illustrates pointers with C-strings."
         << "\nStudy the source code and the output simultaneously.\n\n";

    char str[] = "Springtime";


    char* cPtr = str;
    while (*cPtr != '\0')
    {
        cout << *cPtr;  // Output the ***character*** pointed to
        cPtr++;
    }
    cout << endl << endl;


    cPtr = str;
    while (*cPtr != '\0')
    {
        cout << cPtr << endl;  // Output the ***string*** pointed to
        cPtr++;
    }
    cout << endl << endl;

    return 0;
}
```

13.4.6.1 What you see for the first time in CSTRPTR.CPP

The sole purpose of this sample program is to emphasize the distinction between outputting the single character that a character pointer points to in a string, and outputting all the characters from the character pointed to up to the end of the string.

13.4.6.2 Additional notes and discussion on CSTRPTR.CPP

Understanding the distinction being drawn in this program should help you to get a better grasp of pointers as well as strings, and the use of pointer arithmetic in a C-string context..

13.4.6.3 Follow-up hands-on activities for CSTRPTR.CPP

□ Copy, study and test the program in CSTRPTR.CPP. Look very carefully at the commented `cout` statement in the body of each while loop and check the output from each of these statements very carefully when you run the program.

□ Make a copy of CSTRPTR.CPP and call it CSTRPTR1.CPP. Revise the copy so that the output appears as follows:

```
emitgnirpS

e
me
ime
time
gtime
ngtime
ingtime
ringtime
pringtime
Springtime
```

You must use the same pointers (that is, don't use array indices to access any of the characters in the string), but you may want to use a different kind of loop.

◯ Instructor checkpoint 13.6

Module 14

The C++ string class (our sixth structured data type)

14.1 Objectives

- To understand that a *C++ string variable* is an object of the C++ `string` class, and that
  - The number of characters that a C++ string object can hold is not fixed in size and may be regarded (for all practical purposes) as unlimited in size.
  - There is no notion of a C++ string object being "terminated" by a null character (`'\0'`).
- To learn how `cin` behaves when reading string values into C++ *string objects*.
- To learn how to read string values into C++ string objects with `getline` (which is *not* the same getline we used for C-strings)
- To learn how to write out C++ strings with `cout`.
- To understand that the behavior of file streams is analogous to that of `cin` and `cout` when C++ string objects are being read from an input file stream or written to an output file stream.
- A C++ string object may be declared and *initialized* at the time of declaration, and (unlike the situation with C-strings) may also have a value assigned to it *after* it has been declared.
- To understand how to
  - Copy a constant string value into a C++ string object, or the value of one such object into another

- Concatenate two C++ string objects (join them together)
- Find the length of a C++ string object
- Compare (in the alphabetic sense) two C++ string objects

14.2 List of associated files

- CPPSTR1.CPP illustrates declaration, initialization, copying and concatenation of C++ string objects, as well as computing their lengths and writing them out with `cout`.
- CPPSTR2.CPP illustrates C++ string comparisons and more input/output.
- CPPSTR3.CPP illustrates use of `cin.getline` with C++ string objects.
- CPPSTR4.CPP illustrates the use of an explicit delimiter character with `cin.getline` and C++ string objects.
- CPPSTR5.CPP illustrates reading lines from a textfile into C++ string objects, and points out some potential pitfalls.
- STRING1.DAT and STRING2.DAT are the data files with the same names from Module 13 and are used again here with CPPSTR5.CPP.

14.3 Overview

This Module deals with the `string` class[1] in C++ and (therefore) C++ *string objects*. For the most part, new code that you write that uses strings (as most code does, one way or another) should use the string class rather than C-strings to handle those strings. However, there is a great deal of legacy code out there containing C-string usage and it will be a long time before a knowledge of how C-strings behave will no longer be necessary for those who have to read and maintain legacy code. In addition, there remain situations in which C-strings are still required, such as the name of a file used as the parameter of the `open` function.

In addition, we are still not at the stage where you can assume that the string class and all its functionality will be both present and working properly on you system. Indeed, this seems to be one of the most poorly implemented of the features in the new C++ Standard. The unfortunate bottom line is this: if your strings aren't behaving properly, it's probably your fault, but it *may* not be.

The five sample programs in this Module are analogs of those in Module 13, so that you can see how the same things are done in this new context and using C++ string objects with overloaded operators instead of those "old-fashioned" C-strings. There is no analog to CSTRPTR.CPP since we do not use character pointers with C++ string objects in the same way we do with C-strings.

[1] See also Appendix G and sections 22.4.3 and 22.4.4 of Module 22 for more on strings.

14.4 Sample Programs and Other Files

14.4.1 CPPSTR1.CPP illustrates basic features of C++ string objects

```
// Filename: CPPSTR1.CPP
// Purpose:  Illustrates C++ string objects: declaration, initialization,
//           copying, concatenation, computing length, output with cout.

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

#include "PAUSE.H"

int main()
{
    cout << endl;
    cout << "This program illustrates declaration, "
         << "initialization, copying, concatenation, "   << endl
         << "and output with cout, of C++ strings. "
         << "Study the source code while running it. "   << endl;
    cout << endl;
    Pause(0);

    // You can initialize a C++ string object in its declaration:
    string s1 = "Hello, ";
    string s2("world!");  // Note this method of initialization.

    // You can also assign to a C++ string object after declaring it:
    string s3;
    s3 = "Hello, world!";

    // Now we print out the three strings and their lengths:
    cout << "Length of \"" << s1 << "\" is " << s1.length() << "." << endl;
    cout << "Length of \"" << s2 << "\" is " << s2.length() << "." << endl;
    cout << "Length of \"" << s3 << "\" is " << s3.length() << "." << endl;
    Pause(0);

    string s4 = s1;
    string s5;
    s5 = s2;
    s4 += s5;  // Appends s5 to s4 and is equivalent to s4 = s4 + s5;
    cout << s4 << "<<" << endl;
    Pause(0);

    // Note that an "assignment expression" actually
    // returns the string value of the assignment.
    cout << (s4 = "How are you?") << "<<" << endl;
    cout << (s5 = "Fine!") << "<<" << endl;
    Pause(0);

    // Note that copying a shorter string to a longer string
    // "does the right thing" and puts the '\0' in the right place.
    cout << s4 << "<<" << endl;
    s4 = s5;
    cout << s4 << "<<" << endl;
    Pause(0);

    return 0;
}
```

14.4.1.1 What you see for the first time in CPPSTR1.CPP

- The initialization of a C++ string object at the time of declaration with the value of a quoted string

  Note that in the case of a C++ string object (variable) that this initialization can be performed either by placing an assignment operator between the string variable and the quoted string (as we did for C-string variable initialization), or (since we are dealing here with a class object) by placing the quoted string inside parentheses following the string object.

- Compare the following operations with their counterparts in the world of C-strings:

  Copying one C++ string object to another can be performed by simple assignment.

  Recall that we needed the `strcpy` function for C-strings, since it was not possible to assign the value of one C-string value to another.

  Concatenating two C++ string objects can be performed by the (overloaded) + operator.

  Recall that we needed the `strcat` function to concatenate C-strings.

  Computing the length of a string, i.e., the number of characters in the C++ string object, is now done by the member function `length` in the `string` class interface.

  Recall that we needed a separate function `strlen` to compute lengths of C-strings.

- The display of C-strings with `cout` (all characters in the string are simply displayed in the order in which they occur in the string, with no whitespace inserted either before or after)

14.4.1.2 Additional notes and discussion on CPPSTR1.CPP

Even in this first sample program you should begin to have a feeling that dealing with C++ string objects is a little more "intuitive" than dealing with C-strings. With luck, this feeling will intensify as you look at the remaining sample programs in this Module.

14.4.1.3 Follow-up hands-on activities for CPPSTR1.CPP

□ Copy, study and test the program in CPPSTR1.CPP.

◯ Instructor checkpoint 14.1

14.4.2 CPPSTR2.CPP illustrates C++ string comparisons and input with cin

```
// Filename: CPPSTR2.CPP
// Purpose:  Illustrates C++ string objects: input with cin, comparison,
//           and output with cout

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

#include "PAUSE.H"

int main()
{
    cout << endl;
    cout << "This program illustrates input with cin, "
         << "comparison, and more output with "           << endl
         << "cout, of C++ string objects.  Study the "
         << "source code while running it.  "             << endl
         << "In response to each prompt, enter either "
         << "two strings, or an end-of-line to quit. "    << endl;
    Pause(0);

    string s1, s2;

    cout << "Enter the two strings below: " << endl;
    cin  >> s1 >> s2;  cin.ignore(80, '\n');

    while (cin)
    {
        cout << "The two strings entered and read "
             << "are on the next two lines:"        << endl;
        cout << s1 << "<<" << endl;
        cout << s2 << "<<" << endl;
        Pause(0);

        // The following three if-statements, written separately here
        // to show each of the three possibilities in full, would normally
        // be written as a nested if with an else at the end.
        if (s1 == s2)
            cout << "Those two strings were the same.";
        if (s1 < s2)
            cout << "The 1st string precedes the "
                 << "2nd string, alphabetically.";
        if (s1 > s2)
            cout << "The 2nd string precedes the "
                    "1st string, alphabetically.";
        cout << endl << endl;
        Pause(0);

        cout << "Enter the two strings below: " << endl;
        cin  >> s1 >> s2;  cin.ignore(80, '\n');
    }

    cout << endl;

    return 0;
}
```

14.4.2.1 What you see for the first time in CPPSTR2.CPP

- The reading of a string value into a C++ string object (variable) using `cin`

  Once again `cin` behaves "as usual". That is, it ignores leading whitespace, the reading is stopped by the first whitespace character it encounters after the reading of characters starts, and that whitespace character is left in the input stream.

- The use of the operators `==`, `<` and `>` to determine whether one string is the same as another, or if it comes before or after the other (in the alphabetical sense)

  Recall that we needed the `strcmp` function to compare C-strings, since it was not possible to compare the value of one C-string value to another using the usual relational operators.

14.4.2.2 Additional notes and discussion on CPPSTR2.CPP

The fact that we are able to use the "usual" relational operators (including `<=`, `>=` and `!=`, which are not shown in the program) for comparing C++ string objects is precisely because all of these operators *are* overloaded for the C++ string class. This should enhance your feeling that it is more intuitive to deal with C++ string objects, since these relational operators are more "natural" to use for comparisons than the somewhat clumsy `strcmp` function we needed for C-strings.

14.4.2.3 Follow-up hands-on activities for CPPSTR2.CPP

□ Copy, study and test the program in CPPSTR2.CPP.

□ This activity is designed to help you confirm that when reading string objects from an input stream, the behavior is the same whether the reading is taking place from the keyboard with `cin` or from a file via a file input stream, say `inFile`. So, make a copy of CPPSTR2.CPP called CPPSTR2A.CPP and revise the copy so that the two string variables `s1` and `s2` have values read into them from a file called CPPSTR2A.DAT. Create your own string data for entry into CPPSTR2A.DAT.

◯ Instructor checkpoint 14.2

14.4.3 CPPSTR3.CPP illustrates use of getline with C++ string objects

```
// Filename: CPPSTR3.CPP
// Purpose:  Illustrates input of lines, which may or may not be a
//           full 80 characters, into C++ string objects with getline.

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

#include "PAUSE.H"

int main()
{
    cout << endl;
    cout << "This program illustrates input of C++ string "
         << "\"lines\" with getline.  Study the "            << endl
         << "source code while running it.  Note that "
         << "you get to choose the line length "             << endl
         << "on each pass.  In response to each prompt "
         << "for strings, enter either the two "             << endl
         << "strings requested or an end-of-line to quit. "  << endl;
    cout << endl;
    Pause(0);

    string s1, s2;

    cout << "Enter the two lines below: "   << endl;
    getline(cin, s1);
    getline(cin, s2);

    while (cin)
    {
        cout << "Here are the two lines:" << endl;
        cout << s1 << "<<" << endl;
        cout << s2 << "<<" << endl;
        Pause(0);  cout << endl;

        cout << "Enter the two lines below: "   << endl;
        getline(cin, s1);
        getline(cin, s2);
    }

    cout << endl;

    return 0;
}
```

14.4.3.1 What you see for the first time in CPPSTR3.CPP

This program illustrates the use of `getline` to read a value into a C++ string object.

First of all, note the absence of a `get` function for C++ string objects. Note too that this `getline` is not the `getline` that is a member function in the interface of the stream object `cin` and was used to read values into C-string variables. Instead, *this* `getline` is a non-member function provided by the string class that takes an input stream as its first parameter and the C++ string object to be read as its second.

As for the actual reading of a string, however, this `getline` works (or, more accurately, is *supposed* to work) just like the other one: Reading starts with the next character in the input stream (whether or not that character is a whitespace character). The input stream is the standard input stream `cin` in this program, but once again analogous behavior also applies if we are reading from a file input stream as well.

14.4.3.2 Additional notes and discussion on CPPSTR3.CPP

Note that in this case there is no need for the `numChars` variable we used in the analog program for C-strings (CSTR3.CPP), since our C++ string objects are capable of holding an unlimited (for all practical purposes) number of characters.

14.4.3.3 Follow-up hands-on activities for CPPSTR3.CPP

□ Copy, study and test the program in CPPSTR3.CPP.

◯ Instructor checkpoint 14.3

14.4.4 CPPSTR4.CPP illustrates use of an explicit delimiter character with getline and C++ string objects

```
// Filename: CPPSTR4.CPP
// Purpose:  Illustrates explicit delimiter with
//           cin.getline and C++ string objects.

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

int main()
{
    string s1, s2;

    cout << endl;
    cout << "This program allows the user to choose"
         << "the delimiter character which terminates"     << endl
         << "the reading of each of two strings, but "
         << "not the maximum number of characters"         << endl
         << "read, since in the C++ string objects "
         << "used to read there is essentially no "        << endl
         << "limit (at least no small limit) on the "
         << "number of characters that can be read."       << endl;
    cout << endl;

    char delimiter;
    cout << "Enter the delimiter character: ";
    cin.get(delimiter);
    if (delimiter != '\n')
        cin.ignore(80, '\n');
    cout << endl;

    cout << "Enter some text: " << endl;
    getline(cin, s1, delimiter);
    getline(cin, s2, delimiter);

    cout << endl;
    cout << "Here are the two strings "
         << "as read (one per line):"   << endl;
    cout << s1 << "<<" << endl;
    cout << s2 << "<<" << endl;

    cout << endl;

    return 0;
}
```

14.4.4.1 What you see for the first time in CPPSTR4.CPP

Like its C-string analog (CSTR4.CPP), this program also illustrates the use of (the other) `getline` function, with an explicit delimiter character entered by the user. In the previous sample program (CPPSTR3.CPP), we were again using the default delimiter for `getline`, namely the newline character. So we see here once more that in fact you can use whatever character you like for the delimiter. And, once again, this means that if you use some character other than the (default) newline character for the delimiter, then the newline character is just another character, and hence can itself be read as one of the characters in a string.

So, how exactly does the read work in this case. Well, once again no whitespace is skipped, and the reading starts with the very next character in the input stream, whitespace or not. But this time the reading stops only when the delimiter character is read (so it should *be* there). As before, `getline` removes the delimiter character from the input stream, but does *not* store it in the string object.

14.4.4.2 Additional notes and discussion on CPPSTR4.CPP

If `inFile` is an (open) input file stream, then `getline(inFile, someString)` would behave analogously to `getline(cin, someString)`.

14.4.4.3 Follow-up hands-on activities for CPPSTR4.CPP

□ Copy, study and test the program in CPPSTR4.CPP. Among other tests you make, try the following input:

```
10
*
This *is it.
This is also it.
```

How does the output compare with what you got for the same input to CSTR4.CPP in Module 13?

◯ Instructor checkpoint 14.4

14.4.5 CPPSTR5.CPP, STRING1.DAT and STRING2.DAT illustrate reading lines from a textfile into C++ string objects

```
// Filename: CPPSTR5.CPP
// Purpose:  Reads and displays the lines from a textfile.
//           Illustrates run-time input of filenames.

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#include "PAUSE.H"

int main()
{
    cout << endl;
    cout << "This program asks the user for the name "
         << "of a textfile and then displays the "     << endl
         << "lines from that file.  It then repeats "
         << "the entire process one more time. "       << endl;
    cout << endl;

    ifstream inFile;
    string line, fileName;

    cout << "Enter name of first file to read: ";
    cin >> fileName;  cin.ignore(80, '\n');  cout << endl;
    inFile.open(fileName.c_str());
    cout << "Here are the lines in the file " << fileName << ":\n";
    getline(inFile, line);
    while (inFile)
    {
        cout << line << endl;
        getline(inFile, line);
    }
    inFile.close();  Pause(0);

    cout << "Enter name of second file to read: ";
    cin >> fileName;  cin.ignore(80, '\n');  cout << endl;
    inFile.clear();
    inFile.open(fileName.c_str());
    cout << "Here are the lines in the file " << fileName << ":\n";
    getline(inFile, line);
    while (inFile)
    {
        cout << line << endl;
        getline(inFile, line);
    }
    inFile.close();  Pause(0);

    return 0;
}
```

14.4.5.1 What you see for the first time in CPPSTR5.CPP

Like its C-string analog (CSTR5.CPP), this program does not contain anything new. It illustrates the reading and displaying of the lines of a textfile using (the new) `getline`. Unfortunately, like "the other" getline function, this getline function too may or may not work properly on your system.

Experimentation is once again in order to find out how things work on your system, but whatever your local situation it is (probably) still true that using C++ strings (if they are available at all) is a better choice than using the older C-strings.

14.4.5.2 Additional notes and discussion on CPPSTR5.CPP

By now we hope that you will agree that working with C++ string objects is more intuitive and more flexible than working with C-strings. This of course assumes they are available to you and that they do in fact work, at least for most of what you want to do with them. That this situation is not universally true will presumably be rectified over time as compiler writers continue to work out the bugs.

14.4.5.3 Follow-up hands-on activities for CPPSTR5.CPP

□ Copy, study and test the program in CPPSTR5.CPP. Try both STRING1.DAT and STRING2.DAT as input files.

◯ Instructor checkpoint 14.5

Module 15

Combining arrays, structs, classes, strings and files

15.1 Objectives

- To understand how more complex entities can be modeled by creating complex data structures composed of various simpler data structures.
- To gain some practice using, in various combinations, several of the data structures studied separately up to this point.
- To understand what a *helper function* for a class is and how it can be implemented as a `static` function in the implementation file.

New C++ reserved word
`static`

15.2 List of associated files

- CSTR6.CPP reads lines from a textfile into an array of C-strings and then displays those lines on the screen.
- CPPSTR6.CPP reads lines from a textfile into an array of C++ string objects and then displays those lines on the screen.
- STRING1.DAT and STRING2.DAT are the data files with the same names from Module 13 and are used again here as sample input data files for CSTR6.CPP and CPPSTR6.CPP.
- ARRITEM.CPP illustrates an array of `ItemType` structs.
- ARRTIME.CPP illustrates an array of `Time` objects.
- TIME4.H and TIME4.CPP are the files of the same name from Module 8 and are used here to provide a `Time` class implementation for the test driver in ARRTIME.CPP.

- COMPFRAC.CPP illustrates a `Fraction` class which allows the user to perform arithmetic and comparisons with fractional values.
- COMPFRAC.DAT is program description file for COMPFRAC.CPP.
- FRACTION.H is the header file corresponding to the `Fraction` class whose implementation is in FRACTION.OBJ.
- FRACTION.OBJ contains the implementation (in the form of a .OBJ file, not as source code) of the `Fraction` class whose specification is in FRACTION.H. (FRACTION.CPP is for your instructor's eyes only.)
- TESTFRAC.CPP is a test driver for the `Fraction` class.
- TESTFRAC.DAT is the program description file for TESTFRAC.CPP. specification is in FRACTION.H.

  Other associated files include the usual files necessary for providing the `Menu` class, the `TextItems` class and the `Pause` function.

15.3 Overview

In this Module you see more complex data structures than you have encountered up to now. These include arrays of strings (both C-strings in CSTR6.CPP and C++ strings in CPPSTR6.CPP), an array of structs (in ARRITEM.CPP), an array of class objects (in ARRTIME.CPP), and your most "sophisticated" class to date, the `Fraction` class (in the *FRAC*.* series of files).

All of these more complex structures are of course composed of other structures that you have already studied in some detail. Thus there is very little really new in this Module, which is not to imply that it is trivial or easy to work with the various combinations of "simpler" data structures that you see here.

In fact, for a given problem, choosing the "best" (whatever that means) combination of things that you know about to represent the entities and tasks involved is often the hardest part of the problem. That is not what we are looking at here, though. In this Module we just show you some more complex structures, then ask you to work with them and/or modify them in various ways.

The source code of the implementation of the `Fraction` class is not supplied, so producing your own implementation of this class could be all or part of a major homework assignment. Any such `Fraction` class that *you* produce must of course emulate the behavior of the supplied `Fraction` class as given in the files FRACTION.H and FRACTION.OBJ. The two supplied drivers (COMPFRAC.CPP and TESTFRAC.CPP) provide you with lots of opportunity to experiment with the behavior of the `Fraction` class before starting your own implementation.

15.4 Sample Programs and Other Files

15.4.1 CSTR6.CPP reads lines from a textfile into an array of C-strings and then displays them

```
// Filename: CSTR6.CPP
// Purpose:  Reads and displays the lines from a textfile, pausing
//           whenever a line of 80 dollar signs ($) is encountered.

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

#include "PAUSE.H"

int main()
{
    cout << "\nThis program asks the user for the name "
         << "of a textfile and then displays the "        << endl
         << "lines from that file, pausing whenever "
         << "it sees a line of 80 dollar signs ($).  "    << endl
         << "Note that any such line of $ symbols "
         << "is also actually displayed in the output. "  << endl
         << "Since all lines from the files are read "
         << "in before being displayed, there is a "      << endl
         << "maximum number of lines (20) that the "
         << "file can contain. "                          << endl << endl;

    const int MAX_CHARS = 80;
    const int MAX_LINES = 20;
    typedef char LineType[MAX_CHARS+1];
    typedef LineType FileContentsType[MAX_LINES];

    ifstream inFile;
    LineType fileName, line;
    FileContentsType text;
    int lineIndex = 0;

    cout << "Enter name of file to display: ";
    cin >> fileName;  cin.ignore(80, '\n');  cout << endl;

    inFile.open(fileName);
    inFile.getline(line, MAX_CHARS+2);
    while (inFile)
    {
        strcpy(text[lineIndex], line);
        inFile.getline(line, MAX_CHARS+2); // Note the "+2".
        if (inFile) lineIndex++;
    }
    inFile.close();

    int i;
    char DOLLARS[MAX_CHARS+1] = "";
    for (i = 0; i < MAX_CHARS; i++) strcat(DOLLARS, "$");
    cout << "Here are the lines in the file " << fileName << ":\n";
    for (i = 0; i <= lineIndex; i++)
    {
        cout << text[i] << endl;
        if (strcmp(text[i], DOLLARS) == 0) Pause(0);
    }
    Pause(0);  cout << endl;

    return 0;
}
```

15.4.1.1 What you see for the first time in CSTR6.CPP

This program is similar to the one in CSTR5.CPP, except that in this program all lines from the textfile are *first* read into an array and only *then* are they displayed on the screen. In CSTR5.CPP each line was displayed on the screen immediately after it was read from the file.

15.4.1.2 Additional notes and discussion on CSTR6.CPP

The program in CSTR5.CPP is actually more versatile than the current one since it is not limited in any way by the size of the textfile. The current program cannot display a textfile of more than 20 lines, since that is the size of the array in which we are storing the lines before display. We could, of course, change this limit, or alter this program to read a file of any size "20 lines at a time". In any case, we are using this sample program for pedagogical purposes to illustrate the use of an array of C-strings, not necessarily as a model of how to solve this particular problem!

Note the use of `typedef` to make two different type definitions and note how the first of those definitions is used in the second.

Note also the appearance of `MAX_CHARS+2` as the second parameter of the `geline` function in the body of the while-loop. The reason for this is to ensure that the newline character is always encountered when reading a line of the textfile, assuming no line in the file is longer than 80 characters. Any value less than this would not guarantee that this would happen.

15.4.1.3 Follow-up hands-on activities for CSTR6.CPP

□ Copy, study and test the program in CSTR6.CPP. Use both of the files STRING1.DAT and STRING2.DAT as test input files. Try some additional input files, but make sure that each is no longer than 20 lines, or expect trouble if this is not the case. Try replacing the value `MAX_CHARS+2` first with the value `MAX_CHARS+1` and then with the value `MAX_CHARS` in each of the two instances of the statement

```
inFile.getline(line, MAX_CHARS+2);
```

to see if you can detect any differences in what happen when you display the lines of a textfile.

□ Make a copy of CSTR6.CPP and call it CSTR6A.CPP. Revise the copy so that it can display any textfile containing up to 1000 lines. In addition, during the display the program should pause for the user to press ENTER after each "screenful" of lines, and should report at the end how many lines were displayed in total (i.e., how many lines were in the file).

◯ Instructor checkpoint 15.1

15.4.2 CPPSTR6.CPP reads lines from a textfile into an array of C++ string objects and then displays them

```
// Filename: CPPSTR6.CPP
// Purpose:  Reads and displays the lines from a textfile, pausing
//           whenever a line of 80 dollar signs ($) is encountered.

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#include "PAUSE.H"

int main()
{
    cout << "\nThis program asks the user for the name "
         << "of a textfile and then displays the "        << endl
         << "lines from that file, pausing whenever "
         << "it sees a line of 80 dollar signs ($).  "    << endl
         << "Note that any such line of $ symbols "
         << "is also actually displayed in the output. "  << endl
         << "Since all lines from the files are read "
         << "in before being displayed, there is a "      << endl
         << "maximum number of lines (20) that the "
         << "file can contain. "                          << endl << endl;

    const int MAX_LINES = 20;
    typedef string FileContentsType[MAX_LINES];

    ifstream inFile;
    string fileName, line;
    FileContentsType text;
    int lineIndex = 0;

    cout << "Enter name of file to display: ";
    cin >> fileName;  cin.ignore(80, '\n');  cout << endl;

    inFile.open(fileName.c_str());
    getline(inFile, line);
    while (inFile)
    {
        text[lineIndex] = line;
        getline(inFile, line);
        if (inFile) lineIndex++;
    }
    inFile.close();

    const string DOLLARS(80, '$'); // Creates a string of 80 dollar signs
    cout << "Here are the lines in the file " << fileName << ":\n";
    for (int i = 0; i <= lineIndex; i++)
    {
        cout << text[i] << endl;
        if (text[i] == DOLLARS) Pause(0);
    }
    Pause(0);  cout << endl;

    return 0;
}
```

15.4.2.1 What you see for the first time in CPPSTR6.CPP

This program is similar to the one in CPPSTR5.CPP, except that again in this program (as was the case in CSTR6.CPP) all lines from the textfile are *first* read into an array and only *then* are they displayed on the screen. In CPPSTR5.CPP each line was displayed on the screen immediately after it was read from the file.

Note the use of the `c_str` function in the following statement:

```
inFile.open(fileName.c_str());
```

The reason `c_str` is needed is that the `open` function requires a C-string parameter, and that's what this function provides. It is a function in the interface of the `string` class and extracts the corresponding C-string from a C++ object of type `string`.

Note too the use of the definition

```
const string DOLLARS(80, '$');
```

to establish a constant string consisting of exactly 80 dollar signs. This illustrates a very convenient way to define a string constant or variable consisting of an arbitrary number of copies of a single character.

15.4.2.2 Additional notes and discussion on CPPSTR6.CPP

In this program we only use one `typedef` since we are using the `string` class and hence do not need to define our own C-string type. Contrast this with the situation in the previous sample program CSTR6.CPP.

As was the case with CSTR6.CPP, this program cannot display a textfile of more than 20 lines, since that is again the size of the array in which we are storing the lines before display.

15.4.2.3 Follow-up hands-on activities for CPPSTR6.CPP

□ Copy, study and test the program in CPPSTR6.CPP. Use both of the files STRING1.DAT and STRING2.DAT as test input files. Try some additional input files, but make sure that each is no longer than 20 lines, or expect trouble if this is not the case.

□ Make a copy of CPPSTR6.CPP and call it CPPSTR6A.CPP. Revise the copy so that it can display a textfile of any size, but 20 lines at a time.

◯ Instructor checkpoint 15.2

15.4.3 ARRITEM.CPP illustrates an array of structs

```
// Filenme: ARRITEM.CPP
// Purpose: Illustrates an array of (ItemType) structs.

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

#include "PAUSE.H"


const int MAX_NAME_SIZE = 10;
typedef char NameType[MAX_NAME_SIZE+1];

struct ItemType
{
    int idNumber;
    NameType itemName;
    int numberInStock;
    bool warrantyAvailable;
    double price;
};


void InitItem(/* out */ ItemType& item,
              /* in */  int idNumber,
              /* in */  const NameType itemName,
              /* in */  int numberInStock,
              /* in */  bool warrantyAvailable,
              /* in */  double price);

void UpdateItemCount(/* inout */ ItemType& item,
                     /* in */    int numberInOrOut);

void DisplayItemInfo(/* in */    const ItemType& item);


int main()
{
    cout << "\nThis program uses an array of "
         << "structs to deal with items in inventory. " << endl
         << "You should study the source code "
         << "at the same time as the output. "          << endl << endl;

    const int SIZE = 4;
    ItemType items[SIZE] = {{123, "computer", 15, true,   1999.95},
                            {147, "printer",  11, false,   199.95},
                            {216, "monitor",  13, true,    249.95},
                            {321, "modem",     5, false,    69.95}};

    int i;
    for (i = 0; i < SIZE; i++)
    {
        DisplayItemInfo(items[i]);  cout << endl;
        Pause(0);
    }

    UpdateItemCount(items[0], 7);
    UpdateItemCount(items[1], -3);
    UpdateItemCount(items[2], 2);
    UpdateItemCount(items[3], -1);
```

```
    for (i = 0; i < SIZE; i++)
    {
        DisplayItemInfo(items[i]);  cout << endl;
        Pause(0);
    }

    return 0;
}


//*****************************************************************
void InitItem(/* out */ ItemType& item,
              /* in */  int idNumber,
              /* in */  const NameType itemName,
              /* in */  int numberInStock,
              /* in */  bool warrantyAvailable,
              /* in */  double price)
// Pre:  All in-parameters have been initialized.
// Post: The information in all in-parameters has been
//       stored in "item", which is then returned.
{
    item.idNumber = idNumber;
    strcpy(item.itemName, itemName);
    item.numberInStock = numberInStock;
    item.warrantyAvailable = warrantyAvailable;
    item.price = price;
}



//*****************************************************************
void UpdateItemCount(/* inout */ ItemType& item,
                     /* in */    int numberInOrOut)
// Pre:  item and numberInOrOut have been initialized.
// Post: The number available of item has been updated by the
//       amount numberInOrOut, which may be positive or negative.
{
    item.numberInStock += numberInOrOut;
}



//*****************************************************************
void DisplayItemInfo(/* in */ const ItemType& item)
// Pre:  item has been initialized.
// Post: The information in all fields of item has been displayed.
{
    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);
    cout << setprecision(2);
    cout << "Item Name:          "   << item.itemName      << endl;
    cout << "Item ID Number:     "   << item.idNumber      << endl;
    cout << "Number in stock:    "   << item.numberInStock << endl;
    cout << "Current price:     $"  << item.price         << endl;
    cout << "Warranty Avaiable: "
         << (item.warrantyAvailable ? "Yes" : "No")       << endl;
}
```

15.4.3.1 What you see for the first time in ARRITEM.CPP

- Declaration of an array of structs (`ItemType` structs in this case)
- Initialization of an array of structs at the time of declaration

  Obviously this may not be the best way to initialize an array of structs, particularly if it is a very large array, and the activities ask you change the size of the array and initialize it by reading in the initializing data from a file. However, do note the syntax for initializing an array of structs, which is somewhat analogous to that for a two dimensional array, with the values for each array element contained within a nested set of braces.

15.4.3.2 Additional notes and discussion on ARRITEM.CPP

The struct of this program is an extended version of the `ItemType` struct seen earlier in Module 1 in the sample program ISTRUCT.CPP. The new `ItemType` has two additional fields: another `int` field for the ID number of the item, and C-string field for the name of the item.

15.4.3.3 Follow-up hands-on activities for ARRITEM.CPP

□ Copy, study and test the program in ARRITEM.CPP. Among other tests you may perform, be sure to try this, just to see what you get in the output: Remove the fourth and last struct initialization so that `items[3]` is not initialized.

□ Make a copy of ARRITEM.CPP and call it ARRITEM1.CPP. Then make the following revisions to the copy:

- Change the size of the item array from 4 to 10 and implement a void function called `ReadItemsFromFile` that will read *up to* 10 items from a textfile containing item data and in which each line has the following format (exactly one space separates each value on each line):

  ```
  123 computer 15 T 1999.95
  147 printer 11 F 199.95
  ```

  The function must then return (via function parameters) the array of values read, as well as the number of values that have been read.

- Modify the `UpdateItemCount` function so that it has a single `/* inout */` parameter (the items array) and instead of updating a single struct it has a loop which asks the user which struct to update, the value by which to update, and then performs the update, for as many structs as the user wishes.

Now prepare a file containing exactly 10 data items and test your program first with this file. Then remove some of the data items (or create another file with fewer lines of data) and test your program with this file as well.

◯ Instructor checkpoint 15.3

15.4.4 ARRTIME.CPP illustrates an array of class objects

```
// Filenme: ARRTIME.CPP
// Purpose: Illustrates an array of (Time) objects.

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


#include "TIME4.H"


int main()
{
    cout << "\nThis program illustrates an array "
         << "of Time objects. "                       << endl
         << "You should study the source code "
         << "at the same time as the output. "        << endl << endl;

    Time t0;
    Time t1(12);
    Time t2(11, 45);
    Time t3(11, 44, 55);

    ofstream outFile("times.tmp");

    outFile << t0 << endl
            << t1 << endl
            << t2 << endl
            << t3 << endl;

    outFile.close();


    // Each Time object in the following array declaration is
    // initialized using the default constructor for the Time class.
    Time t[4];

    int i;
    for (i = 0; i < 4; i++)   // Display the values to prove they are the
        cout << t[i] << endl; // ones supplied by the default constructor.
    cout << endl;

    ifstream inFile("times.tmp");

    Time timeRead;
    int index = 0;
    while (inFile >> timeRead)
        t[index++] = timeRead;

    for (i = 0; i < index; i++)
        cout << t[i] << endl;

    inFile.close();

    return 0;
}
```

15.4.4.1 What you see for the first time in ARRTIME.CPP

- Declaration and initialization of an array of class objects (`Time` objects in this case)

  Note that there is no explicit indication that the `Time` objects in the array have been initialized, so what you have to know is this: If there is a default constructor for a class, then when an array of objects of that class is declared, each element of the array is initialized using that default constructor. In fact, you have no other choice, since there is no way to pass parameters to the elements of the array in the array declaration.

- The use of the increment operator in a situation other than a standalone statement for incrementing a variable (see the body of the while-loop that reads the `Time` values from a file)

15.4.4.2 Additional notes and discussion on ARRTIME.CPP

Let's say a little more about the second bulleted item above. As you know, we have always recommended that the increment and decrement operators (`++` and `--`) be used only in standalone statements. But here you see the expression `index++` used as an array index. How come? Well, not everyone in the C++ programming world follows our recommendation, so you will from time to time see these operators used in other situations and the situation you see here is a fairly typical one.

Our recommendation still holds, though. In other words, just because we used it here doesn't mean that we recommend its use here, or anywhere, in this way. Note that because the `++` is placed *after* the variable in this case, a value is *first* read into the array element, *after which* the index value is incremented. But we have to look pretty closely at the loop to see what the value of the index is going to be when the loop finishes, and if we had placed the `++` before the variable `index` the expression would not be the one we wanted. The bottom line: Why give yourself this much trouble, even if others insist on doing it to themselves?

15.4.4.3 Follow-up hands-on activities for ARRTIME.CPP

□ Copy, study and test the program in ARRTIME.CPP. Among other tests you may perform, try this one. Since the values printed after the initialization of the array are all `00:00:00`, you might be suspicious as to whether the array elements really were initialized. After all, might not all of those values have been zero in any case? So, allay your suspicions by making a small (and temporary) change in the class of TIME4.*. In TIME4.H, change the default parameter values for hours, minutes and seconds to `1`, `2` and `3` (respectively). When you run the revised program all the default values should come out as `01:02:03`. Don't forget change the values back to `0` after you've finished.

□ Make a copy of ARRTIME.CPP and call it ARRTIME1.CPP. Then make the following revisions to the copy:

- Change the size of the item array from 4 to 10 and implement a function called `ReadTimesFromFile` that will read up to 10 times from a textfile containing `Time` data and in which each line has the format shown in the following two lines:

  ```
  11:23:12
  02:01:59
  ```

  After reading the item values into the array, the function returns the array as well as how many values have actually been read.

- Write a function called `ComputeTimeSums` that takes as in-parameters two arrays of `Time` objects and an integer indicating how many values are in each array. (Assume that both arrays contain the same number of values.) The function must return a third array containing the "pair wise sums" of the `Time` values in the two input arrays. That is, the first element of this array must contain the sum of the first two `Time` values in the two input arrays, and so on.

- Write a function called `DisplayTimes` that takes as in-parameters an array of `Time` objects and the number of elements in the array and then displays those `Time` objects, one per line.

- Revise the main function so that it declares two arrays of Time objects of size 10, reads Time values into these arrays from two different files (each file is assumed to contain the same number of values), displays the values read in, computes the sums, and displays the sums.

Test the resulting program until it is working properly.

Now prepare two files containing exactly 10 data items each and test your program first with these files. Then remove the same number of data items from each file—to provide a test case with non-full arrays—and test again.

◯ Instructor checkpoint 15.4

15.4.5 COMPFRAC.CPP is a test driver for the Fraction class

```
// Filename: COMPFRAC.CPP
// Purpose:  A program that uses a Fraction class to permit the user
//           to compute the sum, difference, product and/or quotient
//           of two fractional quantities, and also to compare two
//           fractional quantities.

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

#include "FRACTION.H"
#include "MENU.H"
#include "TXITEMS.H"
#include "PAUSE.H"

void ComputeWithFractions();
void CompareFractions();

int main()
{
    Menu m("Main Menu");
    m.AddOption("Quit");
    m.AddOption("Get information");
    m.AddOption("Compute the value of a fractional expression");
    m.AddOption("Compare the values of two fractions");

    TextItems compfracText("compfrac.dat");
    int menuChoice;

    do
    {
        m.Display();
        menuChoice = m.Choice();

        switch (menuChoice)
        {
            case -1:
                cout << "\nProgram now terminating.\n\n";
                break;
            case  1:
                cout << endl;
                break;
            case 2:
                compfracText.DisplayItem("Program Description");
                break;
            case 3:
                ComputeWithFractions();
                break;
            case 4:
                CompareFractions();
                break;
        }

    } while (menuChoice != 1  &&  menuChoice != -1);
    cout << endl;

    return 0;
}
```

```
void ComputeWithFractions()
// Pre:  none
// Post: The user has entered one or more fractional expressions and
//       the value of each expression has been computed and displayed.
//       The user has entered any character other than 'Y' or 'y'.
{
    bool finished;
    Fraction f1, f2, result;
    char op, response;

    do
    {
        cout << "\nEnter fractional expression to be evaluated here: ";
        cin >> f1;  cin >> op;  cin >> f2;
        cin.ignore(80, '\n');  cout << endl;
        cout << "Answer: ";
        cout << f1;  cout << " " << op << " ";  cout << f2;
        cout << " = ";
        switch (op)
        {
            case '+': result = f1 + f2; break;
            case '-': result = f1 - f2; break;
            case '*': result = f1 * f2; break;
            case '/': result = f1 / f2; break;
        }
        cout << result;  cout << endl;

        cout << "Would you like to do another calculation? (Y/N): ";
        cin >> response;  cin.ignore(80, '\n');  cout << endl;
        finished = (response != 'Y'  &&  response != 'y');

    } while (!finished);
}


void CompareFractions()
// Pre:  none
// Post: The user has entered an even number of fractions, the values
//       of each pair have been compared and a message has been
//       displayed indicating whether the two values are equal, and,
//       if not, which is greater and by how much.
{
    bool finished;
    Fraction f1, f2;
    char response;

    do
    {
        cout << "\nEnter two fractions to be compared here: ";
        cin >> f1;  cin >> f2;
        cin.ignore(80, '\n');  cout << endl;

        if (f1 == f2)
            cout << "\nThe two fractions are equal.\n";
        else if (f1 < f2)
            cout << f2 << " is greater than " << f1
                 << " and the difference is " << f2 - f1 << ".\n";
        else // f1 must be > f2 if you make it to this point
            cout << f1 << " is greater than " << f2
                 << " and the difference is " << f1 - f2 << ".\n";
```

```
        cout << "Would you like to do another comparison? (Y/N): ";
        cin >> response;  cin.ignore(80, '\n');  cout << endl;
        finished = (response != 'Y'  &&  response != 'y');

    } while (!finished);
}
```

Contents of the file COMPFRAC.DAT are shown between the heavy lines:

```
Program Description


This program allows the user to compute the sum, difference, product and/or
quotient of fractional expressions, and also to compare two fractions and
report on the difference between them.

By choosing option 3 from the main menu, the user can have the program perform
as many arithmetic calculations (+, -, *, /) as desired before quitting.

Fractional expressions of the kind which this program can evaluate have the
(typical) form shown in the following eight examples (try them!):

1/-2 + 2/3    -5/8 - -15/-4    -3/5 * 2/-7    2/3 / -2/-6
5/6 - 1/-6    5/1 - 6/3        6/8 * 4/6      1/6 / -5/9

Note that each fractional expression consists of two fractions, separated by
one of the four "usual" arithmetic operators (+. -. * or /) with at least one
blank space on either side of the operator (for readability).




!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Note also that the '/' does double duty: first, as the separator between
the numerator and denominator of each fraction; and second, as the operator
indicating that the two fractions in the fractional expression are to be
divided.

Finally, observe that any integer n, when written as a fraction, must be
written in the form n/1 (see the integer 3 in the second example on the
second line of examples above).

Here is a sample of the screen display during such an evaluation request:

Enter fractional expression to be evaluated here: 5/8 + 2/5
Answer: 5/8 + 2/5 = 41/40

If necessary, the answer is "reduced to lowest terms", and in the answer line
each fraction from the input line is displayed in "standard form" (lowest terms
and with negative numerator, positve denominator if the fraction is negative).
All of this is illustrated in the following additional example:

Enter fractional expression to be evaluated here: 26/24 - 7/-6
Answer: 13/12 - -7/6 = 9/4

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

By choosing option 4 from the main menu, the user can have the program perform
```

```
as many fractional comparisons as desired before quitting.


A typical screen display in this case looks like this:

Enter two fractions to be compared here: 3/4 5/8
3/4 is greater than 5/8 and the difference is 1/8.


Here is a second example:

Enter two fractions to be compared here: 3/4 12/16
The two fractions are equal.


And here is a third example:

Enter two fractions to be compared here: -13/8 -12/7
-13/8 is greater than -12/7 and the difference is 5/56.


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
-------------------------------------------------------------------------------
```

15.4.5.1 What you see for the first time in COMPFRAC.CPP

The only new thing in this program is the `Fraction` class, or, to be more precise, client code that uses the `Fraction` class.

15.4.5.2 Additional notes and discussion on COMPFRAC.CPP

If you are a little rusty on fraction arithmetic, you may want to find a mathematics text and review the rules for adding, subtracting, multiplying and dividing fractions.

15.4.5.3 Follow-up hands-on activities for COMPFRAC.CPP

□ Copy and study the program in COMPFRAC.CPP. Don't worry about testing it for now; that will come very shortly. In the meantime you should begin by reading the contents of the file COMPFRAC.DAT, the program description file whose contents can also be displayed when the program runs. This will give you a good feeling for what the program is capable of, in preparation for testing. Then study the code to see how the fractional computations and tests are carried out, in preparation for the testing soon to come.

◯ Instructor checkpoint 15.5

15.4.6 FRACTION.H is the specification file for the Fraction class implemented in FRACTION.OBJ

```
// Filename: FRACTION.H
// Purpose:  Specification file for class Fraction.

// Note that in the following pre/post-conditions, "self" simply refers
// to the fractional object to which the function in question is being
// applied (as opposed to the "other" fraction which is passed in as
// a parameter).  Also, "in reduced form" means "in lowest terms", i.e.,
// the numerator and denominator of the fraction have no common factor.

#ifndef FRACTION_H
#define FRACTION_H

class Fraction
{
    friend istream& operator>>(/* in */  istream& inStream,
                               /* out */ Fraction& f);
    // Pre:  The next item in the standard input stream must have the
    //       form: int/int (with spaces possibly occuring before and/or
    //       after the '/'.  The second int value must be non-zero.
    // Post: The value from the standard input stream having the form
    //       int/int has been read into self as numerator/denominator.
    //       That is "numerator" has been initialized with the int
    //       value preceding the '/' and "denominator" has been
    //       initialized with the int value following the '/'.

    friend ostream& operator<<(/* in */ ostream& outStream,
                               /* in */ const Fraction& f);
    // Pre:  self has been initialized.
    // Post: self is printed in the form numerator/denominator,
    //       unless the fraction is actually an integer, in which
    //       case it is simply printed as that integer.  Also, any
    //       negative fraction is printed with a negative numerator
    //       and a positive denominator.  All fractions are displayed
    //       in reduced form (lowest terms).


public:

    Fraction(/* in */ int numer = 0, /* in */ int denom = 1);
    // Pre:  none
    // Post: "numerator" is set to "numer", "denominator" to "denom",
    //       if both parameters have been supplied (unless denom == 0,
    //       in which case the following message is displayed and the
    //       program is terminated):
    //       Error: Zero denominator not permitted.
    //              Program is now terminating
    //              directly from class constructor.
    //       If only one parameter is supplied, "numerator" is set to
    //       that value and denominator defaults to a value of 1.
    //       If both parameters are omitted, then "numerator" defaults
    //       to a value of 0 and "denominator" to a value of 1.

    void Set(/* in */ int numer = 0, /* in */ int denom = 1);
    // Pre:  none
    // Post: "numerator" is set to "numer", "denominator" to "denom",
    //       if both parameters have been supplied (unless denom == 0,
    //       in which case the following message is displayed and the
    //       program is terminated):
    //       Error: Zero denominator not permitted.
    //              Program is now terminating
    //              directly from function Set.
```

```
    //       If only one parameter is supplied, "numerator" is set to
    //       that value and denominator defaults to a value of 1.
    //       If both parameters are omitted, then "numerator" defaults
    //       to a value of 0 and "denominator" to a value of 1.

    Fraction operator+(/* in */ const Fraction& otherFrac) const;
    // Pre:  Both self and "otherFrac" have been initialized.
    // Post: Value of self + otherFrac is returned, in reduced form.

    Fraction operator-(/* in */ const Fraction& otherFrac) const;
    // Pre:  Both self and "otherFrac" have been initialized.
    // Post: Value of self - otherFrac is returned, in reduced form.

    Fraction operator*(/* in */ const Fraction& otherFrac) const;
    // Pre:  Both self and "otherFrac" have been initialized.
    // Post: Value of self * otherFrac is returned, in reduced form.

    Fraction operator/(/* in */ const Fraction& otherFrac) const;
    // Pre:  Both self and "otherFrac" have been initialized.
    // Post: Value of self / otherFrac is returned, in reduced form.


    bool operator<(/* in */ const Fraction& otherFrac) const;
    // Pre:  Both self and "otherFrac" have been initialized.
    // Post: Returned value is "true" if self is less than otherFrac
    //       and "false" otherwise.

    bool operator==(/* in */ const Fraction& otherFrac) const;
    // Pre:  Both self and "otherFrac" have been initialized.
    // Post: Returned value is "true" if self and otherFrac have the
    //       same value, and "false" otherwise.

private:
    int  numerator;
    int  denominator;
};

#endif
```

15.4.6.1 Notes and discussion on FRACTION.H

This file contains the interface for the `Fraction` class. Everything in it you have seen before, in the context of other classes—default parameters used to define several constructors at once, overloaded arithmetic, relational and I/O operators, and so on—so the only thing new is the class itself, which is designed to allow the user to perform "fraction arithmetic".

15.4.6.2 Follow-up hands-on activities for FRACTION.H

□ Copy and study the `Fraction` class interface in FRACTION.H. You should have available the .OBJ file containing the implementation of this class, so now you may compile (separately) the driver in COMPFRAC.CPP and link with FRACTION.OBJ as well as (of course) MENU.OBJ, TXITEMS.OBJ and PAUSE.OBJ. Test the resulting program thoroughly until you have a good grasp of how the `Fraction` class works, from an operational point of view.

◯ Instructor checkpoint 15.6

15.4.7 TESTFRAC.CPP and TESTFRAC.DAT provide a (partial) test driver for use in developing the Fraction class

```
// Filename: TESTFRAC.CPP
// Purpose:  Tests some features of the Fraction class.

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


#include "FRACTION.H"
#include "MENU.H"
#include "TXITEMS.H"
#include "PAUSE.H"


void DisplayFractionTests(const TextItems&);


int main()
{
    Menu m("Main Menu");
    m.AddOption("Quit");
    m.AddOption("Get information");
    m.AddOption("Display specific fraction tests");

    TextItems testfracText("testfrac.dat");
    int menuChoice;

    do
    {
        m.Display();
        menuChoice = m.Choice();

        switch (menuChoice)
        {
            case -1:
                cout << "\nProgram now terminating.\n\n";
                break;

            case  1:
                cout << endl;
                break;

            case 2:
                testfracText.DisplayItem("Program Description");
                break;

            case 3:
                DisplayFractionTests(testfracText);
                break;
        }

    } while (menuChoice != 1  &&  menuChoice != -1);
    cout << endl;

    return 0;
}
```

```
void DisplayFractionTests(/* in */ const TextItems& testfracText)
// Pre:  "testfracText" has been initialized
// Post: A series of tests has been performed on the Fraction
//       class, one of them with the aid of some input from the
//       user, and the results of the tests have been displayed.
//       Two of the tests are optional and may or may not be
//       performed, at the discretion of the user.  Either test,
//       if successful, will lead to immediate termination of
//       the program, so two separate runs of the program are
//       necessary to complete both of these tests, which are
//       the last two performed.
{
    testfracText.DisplayItem("Test Info: Part 1");

    char response;
    cout << "Would you like to check your constructor? (Y/N): ";
    cin >> response;  cin.ignore(80, '\n');  cout << endl;
    if (response == 'Y'  ||  response == 'y')
    {
        Fraction fTest1(1,0);
        cout << "Fraction value is: " << fTest1 << endl;
        Pause(0);
    }

    cout << "Would you like to check your Set function? (Y/N): ";
    cin >> response;  cin.ignore(80, '\n');  cout << endl;
    if (response == 'Y'  ||  response == 'y')
    {
        Fraction fTest2;
        fTest2.Set(1,0);
        cout << "Fraction value is: " << fTest2 << endl;
        cout << fTest2 << endl;
        Pause(0);
    }


    testfracText.DisplayItem("Test Info: Part 2");
    cout << endl;

    cout << "And here are the corresponding values "
         << "according to *your* Fraction class: "   << endl;

    Fraction f11(3,4), f12(-3,-4), f13(15,20), f14(-9,-12);
    cout << f11 << "  " << f12 << "  " << f13 << "  " << f14 << endl;

    Fraction f21(3,1), f22(3);
    cout << f21 << "  " << f22 << endl;

    Fraction f31(-3,1), f32(-3);
    cout << f31 << "  " << f32 << endl;

    Fraction f4;
    cout << f4 << endl;

    Fraction f51, f52, f53, f54;
    f51.Set(3,4);  f52.Set(-3,-4);  f53.Set(15,20);  f54.Set(-9,-12);
    cout << f51 << "  " << f52 << "  " << f53 << "  " << f54 << endl;

    Fraction f61, f62;
    f61.Set(3,1);  f62.Set(3);
```

```
    cout << f61 << "  " << f62 << endl;

    Fraction f71(-3,1), f72(-3);
    cout << f71 << "  " << f72 << endl;

    Fraction f8;
    f8.Set();  // Note that Set called with no parameters *requires*
               // the parentheses, *unlike* the constructor with no
               // parameters.  Compare the declaration of f4 above.
    cout << f8 << endl;
    Pause(0);


    testfracText.DisplayItem("Test Info: Part 3");
    Fraction g11, g12, g13, g14, g21, g31;
    cin >> g11 >> g12 >> g13 >> g14 >> g21 >> g31;
    cin.ignore(80, '\n');  cout << endl;


    testfracText.DisplayItem("Test Info: Part 4");
    cout << "And here are the actual values as read in "
         << "and then displayed by your program: "    << endl;
    cout << g11 << "  " << g12 << "  " << g13 << "  " << g14 << "  "
         << g21 << "  " << g31 << "  " << endl;
    Pause(0);
}
```

Contents of the file TESTFRAC.DAT are shown between the heavy lines:

```
Program Description


This program provides at least a partial "test suite" for your Fraction class.

The program explicitly tests the following from your Fraction class:

- your constructors

- your Set function

- your overloaded input/output operators

The first part of the test involves declaring some Fractions in such a way as
to invoke each of the constructors that should be available, and also declaring
some fractions and later setting their values with the Set function.

In all cases the resulting values are displayed to make sure they are correct.




!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--------------------------------------------------------------------------------
Test Info: Part 1


So, let the testing begin ...

This first part of the test gives you two chances to terminate the program
"abnormally".  Not to worry, though, because an abnormal termination represents
```

```
a successful test!  So, to get a successful test in both cases you will have to
run the program twice.

The first of these two tests checks to see if your two parameter constructor
terminates the program properly when the second parameter has the value zero.
The second test checks to see if your Set functions does this as well.

If the program does *not* terminate abruptly and you get a message saying that
"Fraction value is: " followed by a value, the test has failed and either your
constructor or your Set function (or both, as the case may be) is/are not
working properly.

If you do not wish to perform either test, simply answer "No" (N) to each of
the following two questions and you will carry on with the rest of the tests.



--------------------------------------------------------------------------------
Test Info: Part 2




Let the testing continue ...

We next show you some code.

Then we show you what you *should* get when you execute that code.

And then we show you what results *your* Fraction class gives when
that same code is executed.

So, to see the code ...




!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Fraction f11(3,4), f12(-3,-4), f13(15,20), f14(-9,-12);
cout << f11 << "  " << f12 << "  " << f13 << "  " << f14 << endl;
Fraction f21(3,1), f22(3);
cout << f21 << "  " << f22 << endl;
Fraction f31(-3,1), f32(-3);
cout << f31 << "  " << f32 << endl;
Fraction f4;
cout << f4 endl;
Fraction f51, f52, f53, f54;
f51.Set(3,4);  f52.Set(-3,-4);  f53.Set(15,20);  f54.Set(-9,-12);
cout << f51 << "  " << f52 << "  " << f53 << "  " << f54 << endl;
Fraction f61, f62;
f61.Set(3,1);  f62.Set(3);
cout << f61 << "  " << f62 << endl;
Fraction f71(-3,1), f72(-3);
cout << f71 << "  " << f72 << endl;
Fraction f8;
f8.Set();  // Note that Set called with no parameters *requires*
           // the parentheses, *unlike* the constructor with no
```

```
          // parameters.  Compare the declaration of f4 above.
cout << f8 << endl;

The output from this code *should* be what you see on the following 8 lines:
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
3/4  3/4  3/4  3/4
3  3
-3  -3
0
3/4  3/4  3/4  3/4
3  3
-3  -3
0
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
--------------------------------------------------------------------------------
Test Info: Part 3
The next part of the test requires you to enter a line of values that looks
exactly like the one below, starting on the next line.  That is, enter a line
of fractional data values that looks *just like* the line of data immediately
above your cursor, and press RETURN when you are finished:
3/4  -3/-4  15/20  -9/-12  3/1  -3/1
--------------------------------------------------------------------------------
Test Info: Part 4
Here's what the output should look like when your program has read in and then
displayed these values:
3/4  3/4  3/4  3/4  3  -3

--------------------------------------------------------------------------------
```

15.4.7.1 Notes and discussion on TESTFRAC.CPP

The purpose of this program is to provide a (partial) test suite for the `Fraction` class during development. It is partial in the sense that it does not test everything you will want to test as you implement your own `Fraction` class, so you will need to add additional code to perform the additional tests.

This program should give you a sense of the kind of driver program you need to have when you are developing a C++ class. You should never turn a class loose on yourself, let alone upon an unsuspecting world, before it has been thoroughly tested, since even then you cannot be sure that everything is OK.

15.4.7.2 Follow-up hands-on activities for TESTFRAC.CPP

□ Copy, study and test the program in TESTFRAC.CPP. You may prepare the executable for this program in the same way you prepared the executable for COMPFRAC.CPP earlier in this Module.

□ Now implement your own `Fraction` class, according to the specification file FRACTION.H. Put the code in a file called FRAC.CPP to begin with, so that the .OBJ file obtained when you compile it will not be confused with the instructor-supplied FRACTION.OBJ. Later, when you have finished your `Fraction` class, you may wish to replace the instructor-supplied version with your own. Use TESTFRAC.CPP as a starting test driver and add the necessary test code as you go.

An interesting question will arise during the implementation of the Fraction class—the question of how to reduce each fraction to its lowest terms. The appropriate algorithm for doing so makes use of the so-called Euclidean algorithm, which finds the greatest common divisor of two positive integers (the numerator and denominator of your fraction, say), and which you then divide into the numerator and denominator. If you are not familiar with this algorithm you will find a function that implements it in Module 22 (see GCD_ITER.CPP) and you may "borrow" that function for use here if you wish.

But that's not the interesting question. The interesting question is this: Just how do you make use of this function? You could make the function a public member function but that would put it in the interface and let all clients of the class have access to it as well. But this function is really a "behind the scenes" function for internal use by your class operations and so should not appear in the public interface of the class.

So, what to do. Well, you could make it a private member function, which would then be available to your public member functions but not your clients. Or, and we recommend this, you could take the following approach. You could "hide" your function completely inside the class implementation by giving it the following prototype:

```
static int gcd(/* in */ int a, /* in */ int b);
```

The purpose of the qualifier `static` is to limit the *visibility* or *scope* of the function to the implementation file, i.e., to the file in which the function has been labeled `static`. This means that even if the client defines and uses a function with the same name (`gcd`), there will be no name conflict. Without the `static` qualifier, this *would* cause a name conflict. This, by the way, is just one of several meanings for the keyword `static` in C++.

Functions used in this way—defined internally by the class, used only by other member functions of the class, and "hidden from the outside world"—are called *helper functions* for the class.

◯ Instructor checkpoint 15.7

□ Implement a second version of your own `Fraction` class, which differs from the first only in that this time all the overloaded operators for the class are friend functions instead of member functions. Place the code in files called FRACFRND.H and FRACFRND.CPP. Test this version as before.

◯ Instructor checkpoint 15.8

Module 16

Command-line parameters

16.1 Objectives

- To understand what is meant by a *command-line parameter*.
- To learn how to declare and use command-line parameters.

16.2 List of associated files

- HELLOCLP.CPP illustrates a command-line parameter by saying "Hello" to the person named on the same line as the command that runs the program.
- LISTCLPS.CPP just displays a list of all command-line parameters.
- SHIFTEXT.CPP is a short "utility program" or "filter program" that copies one textfile to another and simultaneously indents each line by four spaces.
- ROTATE.CPP is another short utility program that can be used to encrypt a textfile in a very simple way, or to decrypt a file previously encrypted with the same program.

16.3 Overview

This Module shows you how to define and use *command-line parameters* with your C++ programs. In addition you will see how using command-line parameters allows you to write (usually quite short) programs that serve as very useful "utility programs" or "filters" that can be applied to manipulate files or perform other helpful tasks in a command-line environment.

The way you actually cause a C++ program to "run" depends on your environment, just as the way you compile and link a program depends on the environment. For example, you may choose the "run" option for your program from a dropdown menu if you are working with an IDE (Integrated Development Environment) with windows of some kind. Or, you may enter the name of your program at a command-line prompt[1] and press ENTER.

A command-line parameter is a string entered on the command line following the name of the file containing the executable C++ program, as in

```
prompt> doit parameter_string
```

where `doit` is the name of the executable C++ program and `parameter_string` is the command-line parameter supplied to `doit`. The idea is that the command-line parameter supplies some information for the program to use when it runs, and this is an easy way to get the information into the program, easier than having the program prompt the user for the information while it is running and having the user enter it at that time. If you know what the information is, why not just give it to the program at the moment you start the program running?

To help put the process into context, consider this analogy. If you have used a command-line environment at all, then you have probably seen a command for making a copy of a file that looks something like the following, although details such as the name of the actual `copy` command may vary:

```
prompt> copy source_file destination_file
```

Here there are two command-line parameters: the name of the file to be copied (`source_file`), and the name of the copy (`destination_file`). If you don't like your system's copy command for some reason, once you know about command-line parameters you can write your own. Of course there are better uses for command-line parameters than reinventing the wheel.

Several other points should be mentioned:

- You can have as many command-line parameters as you like (or you need).
- Each command-line parameter is read in as a C-string. What this means, among other things, is that if you want to input one or more numerical values (26, say) then you must make sure that *your program* converts the string `"26"` to the `int` value `26`, since of course this will not happen automatically.
- The string value entered as a command-line parameter must *not* have any blank spaces in it. However, if you really *do* want to enter a string containing blank spaces as a single command-line parameter, you may enclose the entire string in double quotes.

[1] On some systems (VMS, for example) this may be more difficult than on others (Unix, or even DOS, for example), but it should still be possible with a little effort. In other words, the way command-line parameters are handled is another of those annoying system-dependent features.

- There is always at least one command-line parameter, since the name of the program is itself always considered to be the first command-line parameter. (So, in the `copy` example above there are actually *three* command-line parameters: the name of the program (`copy`) and the names of the two files (`source_file` and `destination_file`)).

16.4 Sample Programs

16.4.1 HELLOCLP.CPP greets the user named on the command line

```
// Filename: HELLOCLP.CPP
// Purpose:  Says "Hello" to the person named in the command line parameter.

#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    if (argc != 2)
    {
        cout << "\nError: Enter a name, please. But just one!\n\n";
        return 1;
    }

    cout << "\nHello, " << argv[1] << ".\n";

    return 0;
}
```

16.4.1.1 What you see for the first time in HELLOCLP.CPP

This program shows, for the first time, at long last, two parameters in the `main` function.[2] What you see, in fact, are the two standard parameters of `main` that you always see whenever a C++ program is going to make use of command-line parameters:

`argc` which contains the *number* of parameters actually entered by the user on the command line whenever the program is run

`argv` which is an array of C-strings and contains the *values* of the command-line parameters entered by the user when the program is run

There is nothing special about the names of these variables (`argc` and `argv`), except they are the names that everyone uses, and if you use some other names (`x` and `y`, say) your program will "look funny" to all the professionals who may someday read it. In addition, the names are intended to be mnemonic, since you can think of `argc` as standing for "`arg`ument `c`ounter" and `argv` as standing for "`arg`ument `v`alues".

[2] You probably thought there was *never* going be anything between those parentheses!

The reason that you can have as many command-line parameters as you want is this: Behind the scenes your program will set aside as much *dynamic storage* (see Module 17) as it needs to store however many command-line parameters you enter. The name of the program[3] (always the *first* command-line parameter) will always be stored in argv[0], the next parameter on the command line will be stored in argv[1], and so on.

16.4.1.2 Additional notes and discussion on HELLOCLP.CPP

Note that both of these parameters of the `main` function are in-parameters but, again by convention, we do not explicitly label them as such.

When it is run, this program simply takes the command-line parameter entered after the program name, assumes it is the name of a person, and displays a message saying "Hello" to that person.

Before doing this, however, the program checks to see how many command-line parameters there are. Note that the count must be *two*, since the name of the program itself is always the first command-line parameter and thus is included in the count. If there are not exactly two, the test fails, so the program displays the error message and returns immediately with a value of 1, which is non-zero and therefore (usually) regarded as a sign of failure. If the test succeeds, the program carries on and prints the greeting.

16.4.1.3 Follow-up hands-on activities for HELLOCLP.CPP

□ Copy, study and test the program in HELLOCLP.CPP. First, run the program with just your first name as a command-line parameter. The output should be a greeting to you, by first name. Next, try entering both your first and last names with a space between them. These are two separate command-line parameters and hence will cause the parameter count to be wrong, so the error message will be output. Finally, enter your first and last name, but this time enclosed in double quotes to make them into a single parameter, and you will again receive the greeting, this time with your full name.

□ Make a copy of HELLOCLP.CPP and call it HELLO2NAM.CPP. Revise the copy so that it allows you to enter your first and last name as two separate command-line parameters *without* enclosing them in double quotes. Also, revise the program so that an appropriate error message is displayed if *both* of these parameters are not entered.

◯ Instructor checkpoint 16.1

[3]Once again, the way this name is stored is system-dependent. On some systems it may be the full pathname of the program file, rather than just the name of the program file.

16.4.2 LISTCLPS.CPP displays a list of all command line parameters

```
// Filename: LISTCLPS.CPP
// Purpose:  Displays a list of all command-line parameters (CLPs).

#include <iostream>
using namespace std;

void DisplayCLPs(int, char* []);

int main(int argc, char* argv[])
{
    cout << "\nThis program illustrates the reading "
         << "of \"command-line parameters\" and the \n"
         << "use of an array of character pointers as "
         << "the second parameter of main for \n"
         << "the repository of those parameters. \n\n";

    DisplayCLPs(argc, argv);

    return 0;
}


void DisplayCLPs(int argc, char* argv[])
{
    cout << "Here is a list of the "      << argc
         << " command line parameters: "  << endl;
    for (int i = 0; i < argc; i++)
        cout << argv[i] << endl;
}
```

16.4.2.1 Notes and discussion on LISTCLPS.CPP

Not much new here. This program simply displays a list of all of its command-line parameters, including its own name, so it gives you a chance to experiment with different numbers and kinks of command-line parameters to see how they are read in. In this case the display is actually performed by a function, which takes as in-parameters the same two parameters we have given to the `main` function to deal with the command-line parameters.

One thing you should note in passing is this. For readability purposes, we usually include the parameter names in a function prototype, even though they are not required there (unlike in function headers, where they are of course necessary). We have left them out here just so you could see how the second parameter type looks (in the prototype of `DisplayCLPs`) when the parameter name (`argv`) is omitted.

16.4.2.2 Follow-up hands-on activities for LISTCLPS.CPP

□ Copy, study and test the program in LISTCLPS.CPP. Try the program first with no parameters on the command line, then with one parameter, two parameters, and so on.

◯ Instructor checkpoint 16.2

16.4.3 SHIFTEXT.CPP indents by 4 spaces each line of a textfile

```
// Filename: SHIFTEXT.CPP
// Purpose:  Indents all the lines in a textfile by 4 spaces.

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main(int argc, char* argv[])
{
    if (argc != 3)
    {
        cout << "\nThis program shifts all the lines in "
             << "a textfile 4 spaces to the right. \n\n"
             << "Usage: \n"
             << "shiftext source_file destination_file\n";

        return 1;
    }
    else
    {
        ifstream inFile(argv[1]);
        ofstream outFile(argv[2]);
        string line;
        getline(inFile, line);
        while (inFile)
        {
            line = "    " + line;
            outFile << line << endl;
            getline(inFile, line);
        }
        inFile.close();
        outFile.close();
    }

    return 0;
}
```

16.4.3.1 Notes and discussion on SHIFTEXT.CPP

This program takes two command-line parameters: the first is the name of an input textfile, and the second is the name of the output textfile. The contents of the input file are copied to the output file, with each line indented by 4 spaces.[4]

16.4.3.2 Follow-up hands-on activities for SHIFTEXT.CPP

□ Copy, study and test the program in SHIFTEXT.CPP. Choose any textfile you like for input—STRING1.DAT and STRING2.DAT from Module 13 make good short test files—and be sure to display the output file in all cases to confirm the indentation.

[4]The author has found such a utility program useful for modifying textfiles before printing so that 3-hole punches don't eliminate printed text near the left margins.

16.4.4 ROTATE.CPP encodes or decodes a textfile

```
// Filename: ROTATE.CPP
// Purpose:  Reads in the text of one file, shifts all characters 47 positions
//           to the right, and outputs the resulting text to a second file.
//           The shift is actually a circular rotation or "wrap-around".

#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main(int argc, char* argv[])
{
    if (argc != 3)
    {
        cout << "\nError: Wrong number of parameters. \n"
             << "       Must be exactly two. \n"
             << "\nCommand Usage: "
             << "\nrotate source_file destination_file \n";

        return 1;
    }

    const int SHIFT_VALUE = 47;
    const int HIGHEST_CODE = 125;
    const int BASE_CODE = 31;

    ifstream inFile(argv[1]);
    ofstream outFile(argv[2]);

    string s;
    int overHang;
    int i;

    while(getline(inFile, s))
    {
        for (i = 0; i < s.length(); i++)
        {
            overHang = int(s[i]) + SHIFT_VALUE - HIGHEST_CODE;
            if (overHang > 0)
                s[i] = char(BASE_CODE + overHang);
            else
                s[i] = char(int(s[i]) + SHIFT_VALUE);
        }
        outFile << s << endl;
    }
    inFile.close();
    outFile.close();

    return 0;
}
```

16.4.4.1 What you see for the first time in ROTATE.CPP

This program requires two command-line parameters: the first is the name of an input textfile, and the second is the name of the output file. This program is a bit unusual in that what happens to the input file depends on what has already happened to it, if anything. From a high-level (user) point of view, this is what happens: if the input textfile is in plaintext (human-readable) form, it will be encrypted in the output file; if the input file is already encrypted, it will be returned to plaintext form in the output file.

16.4.4.2 Additional notes and discussion on ROTATE.CPP

The reason this program works the way it does is this. Each time an input file is processed, each character in that file is "rotated" around the circle of 94 printable ASCII characters by 47 character positions. Hence the first rotation will produce gibberish (the encrypted form) while a second pass will complete that rotation and return each character to its original position.

This encoding scheme is actually a form of transposition cypher (or Caesar cypher).[5]

16.4.4.3 Follow-up hands-on activities for ROTATE.CPP

□ Copy, study and test the program in ROTATE.CPP. Once again choose any textfile you like for input, and once again STRING1.DAT and STRING2.DAT from Module 13 make good short test files. Be sure to display the output file after each pass to confirm the encryption or decryption, as the case may be.

□ Make a copy of ROTATE.CPP and call it ROTATE1.CPP. revise the copy so that it takes a third command-line parameter which is an integer in the range 1..94 and uses this integer as a "shift value" either to encrypt or decrypt the file. Note that for this program the user may choose whatever "shift value" he or she wants (in the range 1..94) to encrypt the text, but then the shift value used to decrypt must be the difference between 94 and the value used to encrypt. Test your program thoroughly to make sure it is working properly. (And—hint, hint—don't forget that the shift-value is read in as a *C-string*!)

◯ Instructor checkpoint 16.3

[5] Named after Julius Caesar, possibly the first to use one.

Module 17

The free store, pointers and dynamic data storage

17.1 Objectives

- To understand what is meant by *dynamic storage* and how it differs from *static storage*.
- To understand what is meant by the *free store* (also called the *heap*).
- To compare and contrast simple "ordinary" variables with simple *pointer variables* and simple *dynamic variables*.
- To compare and contrast *static arrays* and *dynamic arrays*.
- To understand the `new` and `delete` operators and the `NULL` pointer.
- To learn some additional basic mechanics of C++ pointer manipulation.
- To be able to trace a C++ program that uses simple pointer variables and dynamic variables, including drawing the necessary pictures for a *pictorial trace*.

New C++ reserved words

```
new
delete
```

17.2 List of associated files

- PTR_EX6.CPP illustrates both simple dynamic variables and dynamic arrays.
- PTR_EX7.CPP illustrates dynamic data storage with an array of structs.
- PTR_CHAR.CPP compares text display methods, including "dynamic storage with pointer to char".
- PTRPARAM.CPP compares and contrasts "regular" parameters with pointer parameters.

17.3 Overview

In Module 10 we introduced pointers but there the pointers pointed only to things that had already been created by other means in the program. But we used that introduction to see how pointers could be manipulated and used to access both simple data and data components in structured data.

In this Module, by contrast, we will see that we can make a pointer point to *brand new storage* at any time we feel like it (or, more responsibly, any time we *need* it). This storage comes from a part of the computer's memory called the *free store* (also called the *heap*). And once we have that storage, we can either keep it and use it until the program itself finishes, or we can give it back to the free store until we need it later or so that some other user can use it in the meantime. This "other user" may actually *be* another human user if we are running our program on a multi-user system, or it may be just another of our own programs if we are running several different programs on the same computer.

In any case, it is extremely important that any of your C++ programs that makes use of dynamic data storage in this way does in fact return any dynamic storage it no longer needs to the free store. Failure to do so is one of the major causes of program failure, even in commercial software. The problem is, not doing so may not cause any immediate problem, but may simply be one of those insidious bugs that lurks there in your program just waiting for the most inopportune time to jump out and bite you (or your unsuspecting user), so every effort must be made to avoid such *memory leaks*.

There are two generic forms using `new` for allocating dynamic storage that you should be aware of at this point. The first of these is

```
SimpleType* ptrToSimpleType = new SimpleType;

Example:
int* iPtr1 = new int;  // See PTR_EX6.CPP.
```

which creates a pointer pointing to a memory location capable of holding a single value of data type `SimpleType`. To return this dynamic storage to the free store when it is no longer needed, you must use `delete` as follows:

```
delete ptrToSimpleType;
```

The second is

```
SimpleType* ptrToArrayOfSimpleType = new SimpleType[SIZE];

Example:
int* a = new int[6];  // See also PTR_EX6.CPP.
```

which creates a pointer pointing to the first of `SIZE` contiguous memory locations capable of holding `SIZE` values of data type `SimpleType`. To return this dynamic storage to the free store when it is no longer needed, you must use `delete` as follows:

```
delete [] ptrToArrayOfSimpleType;
```

To explicitly indicate that a pointer of *any* type is not pointing at anything, you must use a statement like this:

```
anyPtr = NULL;
```

This works because `NULL` is a "typeless" value that may be assigned to a pointer which is a pointer to any data type at all. `NULL` is not a C++ reserved word, but a pre-defined identifier defined in the `cstddef` library (and other libraries as well).

17.4 Sample Programs

17.4.1 PTR_EX6.CPP illustrates both simple dynamic variables and dynamic arrays

```
// Filename: PTR_EX6.CPP
// Purpose:  Illustrates dynamic storage with both simple and array
//           variables, as well as "new" and "delete", and the NULL pointer.

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

#include "PAUSE.H"

int main()
{
    cout << "\nThis program illustrates dynamic storage "
         << "with new, delete and NULL."
         << "\nStudy the source code and the output simultaneously.\n\n";

    int* iPtr1;        // Declare pointer to int.
    iPtr1 = new int;   // Make pointer point to a (dynamic) int location.
    *iPtr1 = 34;       // Put a value (34) in that (dynamic) location.

    int* iPtr2 = new int; // Declare int pointer/allocate memory at same time.
    *iPtr2 = 12;          // Put a value (12) in that (dynamic) location.

    // Display the two values.
    cout << *iPtr1 << " " << *iPtr2 << endl;
    Pause(0);

    delete iPtr1;  // Return storage pointed to by iPtr1 to free store.
    iPtr1 = iPtr2; // Make iPtr1 point at storage pointed to by iPtr2.
    iPtr2 = NULL;  // Make iPtr2 point at nothing at all.
    // Note that at this point we have "lost" 34.


    // Create a dynamic int array of size 6 pointed to by a.
    int* a = new int[6];

    int i;
    for (i = 0; i < 6; i++)  // Put some values in the array.
        a[i] = i*i*i;
    for (i = 0; i < 6; i++)  // Display the values in the array.
        cout << a[i] << " ";
    cout << endl;
    Pause(0);

    a[5] = *iPtr1;           // Change one of the values in the array.
    for (i = 0; i < 6; i++)  // Display the array values again.
        cout << a[i] << " ";
    cout << endl;
    Pause(0);

    delete [] a;  // Return the array storage to the free store.

    return 0;
}
```

17.4.1.1 What you see for the first time in PTR_EX6.CPP

`new` being used to allocate dynamic memory for storing a simple data value, as in the following statement

```
iPtr1 = new int;
```

`delete` being used to return that single memory location to the free store, as in the following statement

```
delete iPtr1;
```

`new` used to allocate dynamic memory for storing an array of simple data values, as in the following statement, which allocates storage for a dynamic int array of size 6:

```
int* a = new int[6];
```

`delete` used to return that dynamic array storage to the free store, as in the following statement:

```
delete [] a;
```

`NULL` used as a pointer value with no type, so that it can be assigned to any pointer variable, as in the following statement:

```
iPtr2 = NULL;
```

or

```
a = NULL;
```

17.4.1.2 Additional notes and discussion on PTR_EX6.CPP

We should point out that dynamic storage allocated to your program from the free store while the program is running will be reclaimed by the operating system when the program ends, even if the program (i.e., the programmer) does not arrange for this to happen before the program finishes. Or at least it will unless there is a major problem with your operating system itself.[1]

So, there is almost no danger of one of your programs eventually rendering all of your computer memory useless because it "forgot to give the memory back". Even if that were to happen, you could recover by shutting your computer down and "rebooting".

17.4.1.3 Follow-up hands-on activities for PTR_EX6.CPP

□ Copy, study and test the program in PTR_EX6.CPP. Be sure to do a pictorial trace, i.e., draw a pictorial representation of what is actually taking place in memory for each statement that causes a change in the state of the memory associated with the program.

◯ Instructor checkpoint 17.1

[1] Hey, you just never know ...

17.4.2 PTR_EX7.CPP illustrates dynamic data storage with an array of structs

```
// Filename: PTR_EX7.CPP
// Purpose:  Illustrates a dynamic array of (BankAccount) structs.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program illustrates a dynamic array of structs. "
         << "\nStudy the source code and the output simultaneously.\n\n";

    struct BankAccount     // This struct definition would be
    {                      // outside main if it were going to
        int idNum;         // be used by several functions, but
        char kind;         // can also appear inside main since
        double balance;    // it is only going to be used here.
    };

    // Declare and initialize a 3-element array of type BankAccount.
    BankAccount accounts[3] = { { 123, 'C', 199.99 },
                                { 456, 'S', 898.98 },
                                { 789, 'C', 321.45 } };

    // Use a BankAccount pointer to access and display the values
    BankAccount* baPtr = accounts;
    for (int i = 0; i < 3; i++)
    {
        cout << baPtr->idNum   << " ";
        cout << baPtr->kind    << " ";
        cout << baPtr->balance << " " << endl;
        baPtr++;
    }
    cout << endl;


    // Create a dynamic 3-element array of type BankAccount:
    BankAccount* newAccounts = new BankAccount[3];

    // Values for the dynamic array are those of the static array (modified):
    for (int k = 0; k < 3; k++)
    {
        newAccounts[k].idNum   = accounts[k].idNum + 2;
        newAccounts[k].kind    = accounts[k].kind == 'C' ? 'S' : 'C';
        newAccounts[k].balance = accounts[k].balance + 20.00;
    }

    // Display the values in the dynamic array using index notation
    for (k = 0; k < 3; k++)
    {
        cout << newAccounts[k].idNum    << " ";
        cout << newAccounts[k].kind     << " ";
        cout << newAccounts[k].balance  << endl;
    }

    delete [] newAccounts;

    return 0;
}
```

17.4.2.1 What you see for the first time in PTR_EX7.CPP

What's actually "new" here? In one sense, nothing. That's because you've already seen all this "stuff" in one form or another previously. But don't be fooled, because in another sense you are seeing something quite new, and that's the fact that all of this "stuff" is being *combined* in ways that you haven't seen before. So, you need to look closely at how the various constructs are being used in this program. The activities give you at least two alternatives for accomplishing the same thing and you should make sure you look just as closely at those, since you cannot always count on seeing the same thing done in the same way by all programmers.

17.4.2.2 Additional notes and discussion on PTR_EX7.CPP

All we do in this program is this: first, declare and initialize a *static array* of `BankAccount` structs, then display the values in that array; second, create a *dynamic array* of `BankAccount` structs of the same size, copy the values from the first array into the second, modifying them along the way, and then displaying these modified values. So, nothing much, but your task is to make sure you understand how each step is accomplished.

17.4.2.3 Follow-up hands-on activities for PTR_EX7.CPP

□ Copy, study and test the program in PTR_EX7.CPP. Again, be sure to do a pictorial trace.

□ Make a copy of PTR_EX7.CPP and call it PTR_EX7A.CPP. Modify the copy so that the array index is used to access the values for printing out the static array, while two different `BankAccount` pointers are used for accessing the two arrays when the static array values are copied (with modification) into the dynamic array, and a `BankAccount` pointer is also used for access when the values in the dynamic array are displayed.

□ The program in the original PTR_EX7.CPP uses a dynamic array of `BankAccount` structs. There are other alternatives and this activity and the next explore two of those. For example, you could begin with a static array of `BankAccount` pointers, as in:

```
// Create a static array of BankAccount pointers.
BankAccount* newAccounts[3];
```

Then you need to make each of these pointers point at a (dynamic) `BankAccount` struct in the free store, which you can do as follows:

```
// Make each pointer point at a (dynamic) BankAccount in the free store.
for (int j=0; j<3; j++)
    newAccounts[j] = new BankAccount;
```

Then, when you assign the fields of one of these dynamic `BankAccount` structs, you can use the `->` operator, since each array element is a `BankAccount` pointer.

```
for (int k = 0; k < 3; k++)
{
    newAccounts[k]->idNum   = accounts[k].idNum + 2;
    newAccounts[k]->kind    = accounts[k].kind == 'C' ? 'S' : 'C';
    newAccounts[k]->balance = accounts[k].balance + 20.00;
}
```

And finally, when you have finished with the dynamic storage pointed to by the array of `BankAccount` pointers, you need to return it to the free store, which you can do as follows:

```
for (int n = 0; n < 3; n++)
    delete newAccounts[n];
```

So, make a copy of PTR_EX7.CPP and call it PTR_EX7B.CPP. Modify the copy to include the changes suggested above. Test the program to make sure it performs as before.

◯ Instructor checkpoint 17.2

17.4.3 PTR_CHAR.CPP compares text display methods, including dynamic storage with pointer to char

```
// Filename: PTR_CHAR.CPP
// Purpose:  Compares text display methods, including
//           dynamic storage with char*.

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

#include "PAUSE.H"

int main()
{
    cout << "\nThis program displays several lines "   // This is just
         << "of text in various ways and you should "  // "old-fashioned"
         << "\nstudy each of those ways to determine " // output of string
         << "the similarities and differences. \n";    // constants.
    Pause(0);


    string text1[4] = { // text1 is an array of C++ string objects.
        "\nThis program displays several lines ",
        "of text in various ways and you should ",
        "study each of those ways to determine ",
        "the similarities and differences. " };
    for (int i = 0; i < 4; i++)
    {
        cout << text1[i];
        if (i%2 == 1) cout << endl;
    }
    Pause(0);


    typedef char String80[81];
    String80 text2[4] = { // text2 is an array of C-strings.
        "\nThis program displays several lines ",
        "of text in various ways and you should ",
        "study each of those ways to determine ",
        "the similarities and differences. " };
    for (i = 0; i < 4; i++)
    {
        cout << text2[i];
        if (i%2 == 1) cout << endl;
    }
    Pause(0);


    char text3[4][81] = { // text3 is a two-dimensional array of char.
        "\nThis program displays several lines ",
        "of text in various ways and you should ",
        "study each of those ways to determine ",
        "the similarities and differences. " };
    for (i = 0; i < 4; i++)
    {
        cout << text3[i];
        if (i%2 == 1) cout << endl;
    }
    Pause(0);
```

```
    char* text4[4] = { // text4 is an array of "pointer to char".
        "\nThis program displays several lines ",
        "of text in various ways and you should ",
        "study each of those ways to determine ",
        "the similarities and differences. " };
    for (i = 0; i < 4; i++)
    {
        cout << text4[i];
        if (i%2 == 1) cout << endl;
    }
    Pause(0);

    return 0;
}
```

17.4.3.1 What you see for the first time in PTR_CHAR.CPP

This program shows you several different ways of "saying the same thing". The most interesting one, within the context of this Module, is the last one, in which each of the initializing strings within the braces is allocated (automatically, by the compiler) just enough storage to hold it (and its terminating null character `'\0'`, since these are C-strings) and then the corresponding array element (of type `char*`) points to the beginning of that string. Since you have not allocated this storage yourself with `new`, you should not attempt to de-allocate it with `delete`.

17.4.3.2 Additional notes and discussion on PTR_CHAR.CPP

One advantage of last method shown, compared with the previous two for example, is that for `text4` just exactly the amount of storage needed is allocated, whereas the previous two versions (involving `text2` and `text3`) waste storage because enough room to store 80 characters plus the terminating character is set aside for each string used in the initialization.

17.4.3.3 Follow-up hands-on activities for PTR_CHAR.CPP

□ Copy, study and test the program in PTR_CHAR.CPP.

◯ Instructor checkpoint 17.3

17.4.4 PTRPARAM.CPP compares and contrasts "regular" parameters with pointer parameters

```
// Filename: PTRPARAM.CPP
// Purpose:  To test your knowledge of pointers as parameters.

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


void DoIt1(int n1, int& n2)
{
    n1 = n1 * 2;
    n2 = n2 + 3;
    cout << setw(3) << n1 << "  " << setw(3) << n2 << endl;
}


void DoIt2(int* p1, int* p2)
{
    *p1 = *p1 * *p1;
    *p2 =  -2 * *p2;
    cout << setw(3) << *p1 << "  " << setw(3) << *p2 << endl;
}


int main()
{
    cout << "\nThis program illustrates pointer (and other) parameters."
         << "\nStudy the source code and the output simultaneously.\n";

    int n1 = 2;
    int n2 = 3;

    int* p1;
    int* p2;
    p1 = new int;
    p2 = new int;
    *p1 = 4;
    *p2 = 5;

    cout << endl;
    cout << setw(3) << n1 << "  " << setw(3) << n2 << endl;
    DoIt1(n1, n2);
    cout << setw(3) << n1 << "  " << setw(3) << n2 << endl << endl;

    cout << setw(3) << *p1 << "  " << setw(3) << *p2 << endl;
    DoIt2(p1, p2);
    cout << setw(3) << *p1 << "  " << setw(3) << *p2 << endl << endl;

    cout << setw(3) << *p1 << "  " << setw(3) << *p2 << endl;
    DoIt1(*p1, *p2);
    cout << setw(3) << *p1 << "  " << setw(3) << *p2 << endl << endl;

    cout << setw(3) << n1 << "  " << setw(3) << n2 << endl;
    DoIt2(&n1, &n2);
    cout << setw(3) << n1 << "  " << setw(3) << n2 << endl;

    return 0;
}
```

17.4.4.1 Notes and discussion on PTRPARAM.CPP

We had our first brief exposure to pointers used as parameters in the sample program PTR_SWAP.CPP of Module 10.

Here we see more examples, and we also see the usual value and reference parameters thrown into the mix. In short, there is "nothing new" here once again, except that (also once again) you are seeing things you've seen before but those things are put together in new ways. So, once again, take a careful look at what you do see.

17.4.4.2 Follow-up hands-on activities for PTRPARAM.CPP

□ Copy, study and test the program in PTRPARAM.CPP, and draw a pictorial trace of this program. This is another of those programs for which it is particularly important to *predict* the output *before* running the program. If you have any difficulty with your predictions, simply change the literal constant values 2, 3, 4 and 5 to different values, predict again, check your new predictions with the new output, and keep trying this sequence of events until you can consistently get it right.

◯ Instructor checkpoint 17.4

Module 18

Classes and dynamic data storage issues: class destructors, shallow and deep copies, and "the big three"

18.1 Objectives

- To understand that when a class contains one or more data members which require dynamic storage, that storage must be allocated at the time of object declaration by the appropriate constructor, and such classes must have a *class destructor* whose job it is to return that dynamic storage to the free store when an object of the class goes "out of scope" (reaches the point where it will no longer be used by the program).
- To understand the difference between the following two concepts:
  - A *shallow copy* of a class object, which occurs when such an object contains a pointer to dynamic storage and its copy has *only a copy of the pointer* still pointing to the *same* dynamic storage associated with the original object
  - A *deep copy* of a class object, in which a copy is also made of the dynamic storage, and the copy of the pointer points to the copy of the dynamic storage (thus providing a full, or "deep" copy, which is completely independent of the original)

  These two concepts have very nice pictures associated with them, which you may have seen in your course text, but which we do not reproduce

here. Nevertheless, if you understand these concepts well you should be able to draw those pictures and explain them to someone else, and you should persevere until you are able to do this.

- To understand that any class having one or more data members requiring dynamic storage must also implement[1] each of the following (referred to by some C++ authors as the "big three"):
  - A class constructor
  - A class copy constructor (a special kind of constructor that is automatically invoked in certain situations to ensure that a deep copy of an object is made)
  - An overloaded assignment operator (which must be implemented in such a way to ensure that a deep copy of the object is assigned)

18.2 List of associated files

Note the order of appearance of the files in the list below. Our usual practice is to look first at a driver, and only then at the thing it is driving. For the most part we continue that practice here, but because TESTREM3.CPP is a driver for both REM3.H/REM3.CPP *and* REM4.H/REM4.CPP and TESTREM4.CPP uses only REM4.H/REM4.CPP, we have placed the four class files *between* TESTREM3.CPP and TESTREM4.CPP.

Also note that in each new version of the REM?.H/REM?.CPP pair of files, the pre/post conditions for functions appearing in a previous file pair are omitted to save space.

- TESTREM1.CPP is a test driver for the `Reminder` class in REM1.H and REM1.CPP.
- REM1.H is the specification file for a `Reminder` class.
- REM1.CPP is the implementation file corresponding to REM1.H.
- TESTREM2.CPP is a test driver for the `Reminder` class in REM2.H and REM2.CPP.
- REM2.H extends the `Reminder` class in REM1.H by adding a class destructor.
- REM2.CPP is the implementation file corresponding to REM2.H.
- REMINDER.DAT is a simple data file required for running TESTREM2.CPP.

[1] And it's just as important to realize that we did not need to implement any of these "big three" functions ourselves until our classes started to use dynamic data, since the simple default ones created for us by C++ were perfectly adequate prior to that time.

- TESTREM3.CPP is a test driver for the `Reminder` classes in REM3.H and REM3.CPP as well as REM4.H and REM4.CPP.
- REM3.H extends the `Reminder` class in REM2.H by adding a "mutilator" function, for pedagogical purposes only.
- REM3.CPP is the implementation file corresponding to REM3.H.
- REM4.H extends the `Reminder` class in REM3.H by adding a copy constructor and and overloaded assignment operator.
- REM4.CPP is the implementation file corresponding to REM4.H.
- TESTREM4.CPP is a test driver that uses the `Reminder` class in REM4.H and REM4.CPP to show that deep copies of objects are made in each of three specific situations.

18.3 Overview

This Module deals with classes in which some of the data members require dynamic storage, though "without loss of generality" our example will only have one such data member.

We present four versions of the same simple `Reminder` class, which contains only two data members: a date data member (of type `int`) and the (C-string) message reminder for that date (of type "pointer to `char`"). Because this second data member is just a pointer to `char`, the storage for the actual message must be dynamic storage obtained from the free store.

In all versions of the class where one or more of them occur, we have inserted code in the constructors, destructor, and overloaded assignment operator saying that the constructor, destructor or overloaded assignment operator has been called, followed by a pause for the user to press RETURN. This code is initially commented out but you may un-comment it and re-compile if you wish to follow the progress of the constructors and destructor and overloaded assignment operator as they do their work.

Note that we have reverted, for simplicity, to a `Display` function for the class, rather than overloading the `<<` operator, but you may overload the output operator for the `Reminder` class yourself if you wish, for practice.

For a class object containing dynamic data, it is the job of the constructor of that object to make sure that the necessary dynamic storage is allotted to that object, and the job of the object's destructor to return to the free store the storage thus allocated when the object no longer needs it. The copy constructor (a new kind of constructor) must ensure that wherever a copy of an object is made, that copy is a deep copy and the overloaded assignment operator must ensure that the object on the left-hand side of an assignment expression receives a deep copy of the object on the right-hand side whenever an assignment is made.

If all this seems a little overwhelming at this point, don't worry about it yet. On our long C++ expedition we are now perhaps entering the tall grass. But all the beasts lurking therein can be tamed! Just make sure you look carefully at each of the following sample programs and find out what it is trying to tell you. Take heart in the fact that all of this really does make sense.

18.4 Sample Programs and Other Files

18.4.1 TESTREM1.CPP is a test driver for the Reminder class in REM1.H and REM1.CPP

```
// Filename: TESTREM1.CPP
// Purpose:  Tests the Reminder class in REM1.H and REM1.CPP.  Reads
//           reminders from the keyboard and displays them in another form.

#include <iostream>
using namespace std;

#include "REM1.H"
#include "PAUSE.H"


int main()
{
    cout << "\nThis program reads in and re-displays "
         << "\"reminders\" with associated dates. "
         << "\n\nReminders input must have this (typical) form: "
         << "\n20001225First Christmas of the millennium"
         << "\n\nStudy the code and output simultaneously.\n\n";

    cout << "First we display the \"default reminder\": \n";
    Reminder someReminder;
    someReminder.Display();
    Pause(0);

    typedef char String80[81];
    int date;
    String80 message;

    cout << "Now we enter and re-display some further reminders: \n";
    cout << "Enter first reminder (or end-of-file to quit): ";
    cin >> date;
    while (cin)
    {
        cin.getline(message, 80);
        Reminder reminder(date, message);
        reminder.Display();
        cout << endl;
        cout << "Enter next reminder (or end-of-file to quit): ";
        cin >> date;
    }

    return 0;
}
```

18.4.1.1 Notes and discussion on TESTREM1.CPP

It's important that you understand what's going on in this program, even if you can't really "see" it, and just as important that you understand why what is happening "behind the scenes" here must be avoided.

The program begins by declaring a `Reminder` object (namely, `someReminder`) which will be initialized by the default constructor. This "default `Reminder` object" is then displayed, and the object remains in existence until the end of the program, which is fine.

Now let's look at what happens when the loop executes. The while-loop uses a single `Reminder` object (`r`) which is re-initialized and displayed each time a user enters a new date and message from the keyboard. Suppose (and this is actually the case, as you will "see" when you look at the `Reminder` class definition in the REM1.H/REM1.CPP files which follow) that each new message entered has new dynamic storage allocated for it in `r`, but that the dynamic storage allocated to the previous message is *not* returned to the heap.

You will not be able to "actually see" that this is happening, even when you run the program, but if the user continued to enter input from the keyboard, this *memory leak* would eventually cause the program to run out of memory and crash.

18.4.1.2 Follow-up hands-on activities for TESTREM1.CPP

□ Copy and study the program in TESTREM1.CPP, in the light of the above discussion. Compile it for later testing, if you like, but wait till you have looked at the `Reminder` class in the REM1.H/REM1.CPP files before actually proceeding.

□ Draw a pictorial trace of the while-loop for three iterations. Make up your own reminder data as you go.

◯ Instructor checkpoint 18.1

18.4.2 REM1.H is the specification file for a simple Reminder class

```
// Filename: REM1.H
// Purpose:  Specification file for Reminder class implemented in REM1.CPP.

#ifndef REM1_H
#define REM1_H

class Reminder
{
public:

    Reminder(/* in */ int date = 19700101,
             /* in */ const char* messageString = "No message");
    // An alternate, equivalent, header for this constructor is:
    // Reminder(/* in */ int date = 19700101,
    //          /* in */ const char messageString[] = "No message");
    // Constructor
    // Pre:  19700101 <= date <= 29990101, and
    //       messageString has been initialized.
    // Post: A new class object has been constructed,
    //       having the input date and messageString.

    void Display() const;
    // Pre:  self is initialized.
    // Post: Date and message have been output in the (typical) form:
    //       April 20, 1944
    //       This is somebody's birthday.
    //       in which the name of the month is displayed as a string, and
    //       the message string is displayed on the line beneath the date.


private:

    int   date;
    char* messagePtr;
};

#endif
```

18.4.2.1 Notes and discussion on REM1.H

This class contains just a three-in-one constructor with default parameter values and a `Display` member function. The main thing of interest in the current context is the second data member (`messagePtr`, which is a pointer to `char`). Because this is only a pointer, the class has no storage of its own for the message and hence this storage must come from the heap.

18.4.2.2 Follow-up hands-on activities for REM1.H

□ Copy the file REM1.H and study the code.

18.4.3 REM1.CPP is the implementation file for REM1.H

```
// Filename: REM1.CPP
// Purpose:  Implementation file corresponding to REM1.H.

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

#include "REM1.H"
#include "PAUSE.H"

// Private members of class:
//     int   date;
//     char* messagePtr;

//*****************************************************************
Reminder::Reminder(/* in */ int date,
                   /* in */ const char* messageString)
// Constructor
// Pre:  19700101 <= date <= 29990101, and
//       messageString has been initialized.
// Post: A new class object has been constructed,
//       having the input date and messageString.
{
    this->date = date;
    messagePtr = new char[strlen(messageString) + 1];
    strcpy(messagePtr, messageString);

    // Un-comment the following two lines for run-time confirmation
    // that this constructor has been called:
    // cout << "Now in constructor for class Reminder (version 1) ... \n";
    // Pause();  cout << endl;
}


//*****************************************************************
void Reminder::Display() const
// Pre:  self is initialized.
// Post: Date and message have been output in the (typical) form:
//       April 20, 1944
//       This is somebody's birthday.
//       in which the name of the month is displayed as a string, and
//       the message string is displayed on the line beneath the date.
{
    static char* monthString[12] =
    {
        "January",   "February", "March",    "April",
        "May",       "June",     "July",     "August",
        "September", "October",  "November", "December"
    };

    int year  = date/10000;
    int month = date%10000 / 100;
    int day   = date%10000 % 100;
    cout << monthString[month-1] << ' ' << day << ", " << year << endl
         << messagePtr << endl;
}
```

18.4.3.1 What you see for the first time in and REM1.CPP

The major new thing you see here is a constructor allocating dynamic storage on the heap for an incoming message and then storing that message (by copying it, in this case) in that storage area.

As an aside, you also see the use (again) of the `static` qualifier, but this time applied to the `monthString` array in the body of the `Display` function. Used in this way, the keyword `static` means that the array containing the names of the months (which is a local variable and would normally have to be created each time the function is called) is instead created only once and remains in existence between function calls.

18.4.3.2 Additional notes and discussion on REM1.CPP

Note the commented-out code in the body of the constructor. All versions of this class in this Module will have similar code that you may activate/de-activate as required during testing.

18.4.3.3 Follow-up hands-on activities for TESTREM1.CPP, REM1.H and REM1.CPP

□ Make sure you have copies of TESTREM1.CPP, REM1.H and REM1.CPP. Generate TESTREM1.EXE and test the program.

□ Now add the following code to TESTREM1.CPP, after the call to the `Pause` function, to test the one-parameter version of the constructor:

```
Reminder anotherReminder(20000401);
anotherReminder.Display();
Pause(0);
```

Re-generate TESTREM1.EXE and run it to make sure this code gives the output you expect.

□ Un-comment the commented code in the body of the constructor, re-compile the class code, regenerate TESTREM1.EXE and run to confirm that the constructor is called once for each loop iteration. What is *not* confirmed, of course, is that the current message is "cut loose", and its storage "lost", each time a *new* message is entered by the user.

◯ Instructor checkpoint 18.2

18.4.4 TESTREM2.CPP is a test driver for the Reminder class in REM2.H and REM2.CPP

Contents of the file REMINDER.DAT are shown between the heavy lines:

```
20000315Ides of March
20000417Don't forget the final exam! (It happens!)
20000701Canada Day
20001225First Christmas of the new millenium ...
```

18.4.4.1 Notes and discussion on TESTREM2.CPP

This program is much the same as the one in TESTREM1.CPP, except that input comes from a file.

18.4.4.2 Follow-up hands-on activities for TESTREM2.CPP

□ Copy TESTREM2.CPP and study the program. Compile it for later testing, if you like, but wait till you have looked at the `Reminder` class in the REM2.H/REM2.CPP files before actually proceeding.

18.4.5 REM2.H is the specification file for the Reminder class with destructor

```
// Filename: REM2.H
// Purpose:  Specification file for Reminder class implemented in REM2.CPP.

#ifndef REM2_H
#define REM2_H

class Reminder
{
public:

    Reminder(/* in */ int date = 19700101,
             /* in */ const char* messageString = "No message");

    ~Reminder();
    // Destructor
    // Pre:  self is initialized.
    // Post: Message string storage pointed to by messagePtr
    //       has been deallocated.

    void Display() const;

private:

    int   date;
    char* messagePtr;
};

#endif
```

18.4.5.1 What you see for the first time in REM2.H

Here you see for the first time the *prototype* of a *class destructor*. Note that, like a class constructor, the name is the same as the class, except that it is preceded by a *tilde character* (`~`). Also, like any constructor, a destructor has no return-type (not even `void`). However, unlike a constructor, the destructor must *not* have any parameters, and this (along with the name) requires that there be *only one* destructor. In other words, the class destructor is *unique*. It is the job of the class destructor to "clean up" after an object as the object "goes out of scope". For the moment, this just means returning to the heap any dynamic storage that has been allotted to the object.

18.4.5.2 Additional notes and discussion on REM2.H

Note that we did not need any class destructors[2] until this Module, when our objects began to have dynamic storage. Before that storage for our objects came and went just like storage for any other "ordinary" variables.

18.4.5.3 Follow-up hands-on activities for REM2.H

□ Copy REM2.H and study the syntax of the class destructor prototype.

18.4.6 REM2.CPP is the implementation file for REM2.H

```
// Filename: REM2.CPP
// Purpose:  Implementation file corresponding to REM2.H.

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

#include "REM2.H"
#include "PAUSE.H"


// Private members of class:
//     int   date;
//     char* messagePtr;

//*********************************************************************
Reminder::Reminder(/* in */ int date,
                   /* in */ const char* messageString)
{
    this->date = date;
    messagePtr = new char[strlen(messageString) + 1];
    strcpy(messagePtr, messageString);

    // Un-comment the following two lines for run-time confirmation
    // that this constructor has been called:
    // cout << "Now in constructor for class Reminder (version 2) ... \n";
    // Pause(0);  cout << endl;
}


//*********************************************************************
Reminder::~Reminder()
// Destructor
// Pre:  self is initialized.
// Post: Message string storage pointed to by messagePtr
//       has been deallocated.
{
    delete [] messagePtr;

    // Un-comment the following two lines for run-time confirmation
    // that the destructor has been called:
    // cout << "Now in the destructor for class Reminder (version 2) ... \n";
    // Pause(0);  cout << endl;
}
```

[2] With one exception, as we noted earlier. The `Menu` class should have had a destructor, but didn't.

```
//*****************************************************************
void Reminder::Display() const
{
    static char* monthString[12] =
    {
        "January",   "February", "March",    "April",
        "May",       "June",     "July",     "August",
        "September", "October",  "November", "December"
    };

    int year  = date/10000;
    int month = date%10000 / 100;
    int day   = date%10000 % 100;
    cout << monthString[month-1] << ' ' << day << ", " << year << endl
         << messagePtr << endl;
}
```

18.4.6.1 What you see for the first time in REM2.CPP

Here you see for the first time the *implementation* of a *class destructor*. In this case, the body of that destructor contains just a single line of code, but a very important one since it's the line that returns the previously-allotted dynamic storage (for the message string) to the heap.

18.4.6.2 Additional notes and discussion on REM2.CPP

We should bring to your attention again the commented-out code in the body of the destructor, code that you will want to activate during testing.

18.4.6.3 Follow-up hands-on activities for TESTREM2.CPP, REM2.H and REM2.CPP

□ Make sure you have copies of TESTREM2.CPP, REM2.H, REM2.CPP and REMINDER.DAT. Generate and test TESTREM2.EXE with the input data file REMINDER.DAT.

□ Now un-comment the output code in the bodies of both the constructor and the destructor definitions in REM2.CPP. Then re-generate TESTREM2.EXE using this revised version of REM2.CPP and once again run the program using REMAINDER.DAT as the input file. This time watch very carefully the messages that tell when the program has reached the constructor or destructor, and match up those calls with what's going on in the program by studying the source code and the output simultaneously.

18.4.7 TESTREM3.CPP is a test driver for the Reminder classes in REM2.H and REM2.CPP and in REM4.H and REM4.CPP

```
// Filename: TESTREM3.CPP
// Purpose:  When used with REM3.H/REM3.CPP, shows the problems that
//           arise when you have a class with dynamic data and yet you
//           *don't* have an overloaded assignment operator = and/or a
//           copy constructor.  When used with REM4.H/REM4.CPP, shows
//           that those problems go away when you *do* have an overloaded
//           assignment operator and a copy constructor.

#include <iostream>
using namespace std;

#include "REM?.H"  // Note: ? to be replaced first by 3, then by 4
#include "PAUSE.H"

int main()
{
    cout << endl;

    // Illustrates that the kind of copy made by assignment depends
    // on whether the assignment operator has been overloaded.
    Reminder r1_1(20000101, "First message");
    Reminder r1_2;
    r1_2 = r1_1; // Copy is "shallow" with REM3.*, "deep" with REM4.*.
    cout << "r1_1: "; r1_1.Display();  Pause(0);
    cout << "r1_2: "; r1_2.Display();  Pause(0);
    r1_1.MutilateMessageString();
    cout << "r1_1: "; r1_1.Display();  Pause(0);
    cout << "r1_2: "; r1_2.Display();  Pause(0);
    cout << endl;

    // Illustrates that the kind of copy made by initialization
    // depends on whether a copy constructor has been provided.
    Reminder r2_1(20000202, "Second message");
    Reminder r2_2 = r2_1; // Copy is "shallow" with REM3.*, "deep" with REM4.*.
    cout << "r2_1: "; r2_1.Display();  Pause(0);
    cout << "r2_2: "; r2_2.Display();  Pause(0);
    r2_1.MutilateMessageString();
    cout << "r2_1: "; r2_1.Display();  Pause(0);
    cout << "r2_2: "; r2_2.Display();  Pause(0);
    cout << endl;

    // Again illustrates that the kind of copy made by initialization
    // depends on whether a copy constructor has been provided.
    // Note the alternate form of initialization used this time around.
    Reminder r3_1(20000303, "Third message");
    Reminder r3_2(r3_1);  // Copy is "shallow" with REM3.*, "deep" with REM4.*.
    cout << "r3_1: "; r3_1.Display();  Pause(0);
    cout << "r3_2: "; r3_2.Display();  Pause(0);
    r3_1.MutilateMessageString();
    cout << "r3_1: "; r3_1.Display();  Pause(0);
    cout << "r3_2: "; r3_2.Display();  Pause(0);
    cout << endl;

    return 0;
}
```

18.4.7.1 Notes and discussion on TESTREM3.CPP

The first thing to note about this program is that to fully appreciate what the program is designed to show you, you have to run it twice using two quite different versions of the `Reminder` class, and examine the output very carefully each time. The first version of the class is in REM3.H/REM3.CPP and the second is in REM4.H/REM4.CPP.

The idea is this. The program makes one assignment (of one object to another) and two initializations (using one object to initialize another). In each case one would hope/expect that a full copy (i.e., a "deep" copy, in the terminology used is this context) would be made, in the sense that if the original is modified *after* the assignment or initialization is completed, one would not expect the modifications to show up in the assigned or initialized object. But those modifications *do* show up when we use the REM3.* version of the class `Reminder` because this version lacks both a *copy constructor* (needed for the initializations) and an *overloaded assignment operator* (needed for the assignment).

Both of these missing functions *are* present in the REM4.* version of the class `Reminder`, so the problems go away when this version of the class is used with the driver.

18.4.7.2 Follow-up hands-on activities for TESTREM3.CPP

□ Copy TESTREM3.CPP and study the program. Study in particular the *structure* of the program and note how the assignment and initializations are followed by a call to the member function `MutilateMessageString`, with displays of the original and the assigned (or initialized) object value before and after this call. Note too the number and position of the various calls to `Pause`, which will allow you to contemplate each displayed value as you run the program. Then go on and study REM3.H/REM3.CPP and REM4.H/REM4.CPP before proceeding with the next activity. But do make sure you come back and complete that activity when you *have* studied the two class definitions.

□ So, are you ready? This activity assumes you have studied REM3.H and REM3.CPP, as well as REM4.H and REM4.CPP. Proceed sequentially through the following numbered steps. There is no need to un-comment the commented-out code in the constructor and destructor bodies for this activity.

1. Change `REM?.H` in TESTREM3.CPP to `REM3.H`.
2. Generate TESTREM3.EXE using the REM3.H/REM3.CPP version of the class.
3. Run TESTREM3.EXE. Follow through each line of the source code in TESTREM3.CPP as you do, and make sure you predict wheat each call to the `Display` function is going to show you. Note any discrepancies and re-examine REM3.H/REM3.CPP for an explanation if necessary.
4. Repeat the previous three steps with the necessary changes to use the REM4.H/REM4.CPP version of the `Reminder` class.

18.4.8 REM3.H is the specification file for the Reminder class with destructor and "message mutilator"

```
// Filename: REM3.H
// Purpose:  Specification file for Reminder class implemented in REM3.CPP.

#ifndef REM3_H
#define REM3_H

class Reminder
{
public:

    Reminder(/* in */ int date = 19700101,
             /* in */ const char* messageString = "No message");

    ~Reminder();

    void Display() const;

    void MutilateMessageString();
    // Used only for pedagogical reasons to help show that only a
    // "shallow" copy of an object has been made.
    // Pre:  self has been initialized with a message of size >= 2.
    // Post: Alternate characters of the message pointed to by
    //       messagePtr have been changed to 'X'.

private:

    int   date;
    char* messagePtr;
};

#endif
```

18.4.8.1 Notes and discussion on REM3.H

This version of the `Reminder` class is the same as that in REM2.H, except for the addition of the `MutilateMessageString` member function. The sole purpose of this rather bizarre member function is pedagogical—it permits the client (the driver in TESTREM3.CPP) to directly alter the "internal" message string of an object so we can better see what is happening when we are making assignments and/or initializations (i.e., when we are making copies of objects in one way or another).

18.4.8.2 Follow-up hands-on activities for REM3.H

□ Copy REM3.H and study the prototype of the `MutilateMessageString` member function.

18.4.9 REM3.CPP is the implementation file for REM3.H

```
// Filename: REM3.CPP
// Purpose:  Implementation file corresponding to REM3.H.

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

#include "REM3.H"
#include "PAUSE.H"


// Private members of class:
//     int   date;
//     char* messagePtr;

//*****************************************************************
Reminder::Reminder(/* in */ int date,
                   /* in */ const char* messageString)
{
    this->date = date;
    messagePtr = new char[strlen(messageString) + 1];
    strcpy(messagePtr, messageString);

    // Un-comment the following two lines for run-time confirmation
    // that this constructor has been called:
    // cout << "Now in constructor for class Reminder (version 3) ... \n";
    // Pause(0);  cout << endl;
}


//*****************************************************************
Reminder::~Reminder()
{
    delete [] messagePtr;

    // Un-comment the following two lines for run-time confirmation
    // that the destructor has been called:
    // cout << "Now in the destructor for class Reminder (version 3) ... \n";
    // Pause(0);  cout << endl;
}


//*****************************************************************
void Reminder::Display() const
{
    static char* monthString[12] =
    {
        "January",   "February", "March",    "April",
        "May",       "June",     "July",     "August",
        "September", "October",  "November", "December"
    };

    int year  = date/10000;
    int month = date%10000 / 100;
    int day   = date%10000 % 100;
    cout << monthString[month-1] << ' ' << day << ", " << year << endl
         << messagePtr << endl;
}
```

```
//*************************************************************
void Reminder::MutilateMessageString()
// Used only for pedagogical reasons to help show that only a
// "shallow" copy of an object has been made.
// Pre:  self has been initialized with a message of size >= 2.
// Post: Alternate characters of the message pointed to by
//       messagePtr have been changed to 'X'.
{
    for (int i = 0; i < strlen(messagePtr); i++)
        if (i % 2 == 0) messagePtr[i] = 'X';
}
```

18.4.9.1 Notes and discussion on REM3.CPP

Here you need only take a look at the implementation of the member function `MutilateMessageString` to see what it does to the message string so that you will recognize the effect when you see it in the output of TESTREM3.CPP.

18.4.9.2 Follow-up hands-on activities for REM3.CPP

□ Copy REM3.CPP and study the implementation of `MutilateMessageString`.

18.4.10 REM4.H is the specification file for the Reminder class with destructor, copy constructor and overloaded assignment operator

```
// Filename: REM4.H
// Purpose:  Specification file for Reminder class implemented in REM4.CPP.

#ifndef REM4_H
#define REM4_H

class Reminder
{
public:

    Reminder(/* in */ int date = 19700101,
             /* in */ const char* messageString = "No message");

    Reminder(const Reminder& otherReminder);
    // Copy constructor
    // Pre:  otherReminder has been initialized.
    // Post: A new class object is constructed with its date and
    //       message string the same as otherReminder's.
    //       Note:
    //       This constructor is implicitly invoked whenever:
    //       - A Reminder object is passed by value, or
    //       - A Reminder object is returned as a function value, or
    //       - A Reminder object is initialized by another Reminder object
    //         in a declaration

    Reminder& operator=(const Reminder& rightSide);
    // Pre:  rightSide has been initialized.
    // Post: A "deep" copy of rightSide has been made and assigned
    //       to the Reminder object on the left side of the assignment
    //       operator =.

    ~Reminder();

    void Display() const;

    void MutilateMessageString();


private:

    int   date;
    char* messagePtr;
};

#endif
```

18.4.10.1 What you see for the first time in REM4.H

In this file you see, finally, the remaining two of the "big three" member functions needed in each class whose objects require dynamic data storage. The three are:

- A class destructor (already discussed)
- A copy constructor

  You can always recognize a copy constructor when you see one. It has the following generic prototype:

  ```
  ClassName& ClassName(const ClassName& objectName);
  ```

- An overloaded assignment operator

  An overloaded assignment operator will (quite often) have the following generic prototype:

  ```
  ClassName& operator=(const ClassName& objectToBeCopied);
  ```

18.4.10.2 Additional notes and discussion on REM4.H

Note in the comments following the prototype of the copy constructor for the `Reminder` class the listing of those situations in which the copy constructor is automatically invoked. The TESTREM4.CPP test driver is designed especially to demonstrate that the copy constructor really is invoked in each of these situations.

18.4.10.3 Follow-up hands-on activities for REM4.H

□ Copy REM4.H and study the prototype (including, especially, the pre/post conditions).

18.4.11 REM4.CPP is the implementation file for REM4.H

```
// Filename: REM4.CPP
// Purpose:  Implementation file corresponding to REM4.H.

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

#include "REM4.H"
#include "PAUSE.H"

// Private members of class:
//     int   date;
//     char* messagePtr;

//*****************************************************************
Reminder::Reminder(/* in */ int date,
                   /* in */ const char* messageString)
{
    this->date = date;
    messagePtr = new char[strlen(messageString) + 1];
    strcpy(messagePtr, messageString);
    // cout << "Now in constructor for class Reminder (version 4) ... \n";
    // Pause(0);  cout << endl;
}


//*****************************************************************
Reminder::Reminder(const Reminder& otherReminder)
// Copy constructor
// Pre:  otherReminder has been initialized.
// Post: A new class object is constructed with its date and
//       message string the same as otherReminder's.
//       Note:
//       This constructor is implicitly invoked whenever:
//       - A Reminder object is passed by value, or
//       - A Reminder object is returned as a function value, or
//       - A Reminder object is initialized by another Reminder object
//         in a declaration
{
    date = otherReminder.date;
    messagePtr = new char[strlen(otherReminder.messagePtr) + 1];
    strcpy(messagePtr, otherReminder.messagePtr);
    // cout << "Now in copy constructor for Reminder ..." << endl;
    // Pause(0);
}


//*****************************************************************
Reminder& Reminder::operator=(const Reminder& rightSide)
{
    date = rightSide.date;
    int newLength = strlen(rightSide.messagePtr);
    if (newLength > strlen(messagePtr))
    {
        delete [] messagePtr;
        messagePtr = new char[newLength+1];
    }
    strcpy(messagePtr, rightSide.messagePtr);
    // cout << "Now in overloaded = operator for Reminder ..." << endl;
    // Pause(0);
    return *this;
}
```

```
//*************************************************************
Reminder::~Reminder()
{
    delete [] messagePtr;
    // cout << "Now in the destructor for class Reminder (version 4) ... \n";
    // Pause(0);  cout << endl;
}


//*************************************************************
void Reminder::Display() const
{
    static char* monthString[12] =
    {
        "January",   "February", "March",    "April",
        "May",       "June",     "July",     "August",
        "September", "October",  "November", "December"
    };

    int year  = date/10000;
    int month = date%10000 / 100;
    int day   = date%10000 % 100;
    cout << monthString[month-1] << ' ' << day << ", " << year << endl
         << messagePtr << endl;
}


//*************************************************************
void Reminder::MutilateMessageString()
{
    for (int i = 0; i < strlen(messagePtr); i++)
        if (i % 2 == 0) messagePtr[i] = 'X';
}
```

18.4.11.1 What you see for the first time in REM4.CPP

This file shows you the implementation of both the copy constructor and the overloaded assignment operator for the `Reminder` class. Study each verly carefully. Note in particular the use of the `this` pointer. Returning `*this` from the overloaded assignment operator is the preferred way to return the "reference to self" that we need in this situation, since we said in the prototype that we would return a reference to an object of the class type being assigned.

18.4.11.2 Additional notes and discussion on REM4.CPP

This rest of this class implementation is the same as that found in REM3.H and REM3.CPP.

18.4.11.3 Follow-up hands-on activities for REM4.CPP

□ Copy REM4.CPP and study the implementations of the copy constructor and the overloaded assignment operator.

When you have finished studying the class versions in REM4.H and REM4.CPP as well as the previous REM3.H and REM3.CPP, go back and complete the last activity for TESTREM3.CPP. Then carry on with TESTREM4.CPP.

18.4.12 TESTREM4.CPP is a test driver for the copy constructor using the Reminder class in REM4.H and REM4.CPP

```
// Filename: TESTREM4.CPP
// Purpose:  Tests copy constructor in three different situations

#include <iostream>
using namespace std;

#include "REM4.H"
#include "PAUSE.H"

void DisplayReminderWithBanner(Reminder r)
// Pre:  "r" is initialized
// Post: "r" is displayed as before, but with *'s above and ='s below.
{
    cout << "****************************************" << endl;
    r.Display();  cout << endl;
    cout << "========================================" << endl;
}

Reminder Halloween()
// Pre:  none
// Post: A date of October 31, 2000, with message "Boo!" is returned.
{
    Reminder r(20001031, "Boo!");
    return r;
}

int main()
{
    cout << endl;

    Reminder r(20000315, "Ides of March");
    r.Display();  cout << endl;  Pause(0);

    Reminder r1;
    // Overloaded assignment operator = invoked here because
    // object r is being assigned to object r1:
    r1 = r;
    r1.Display();  cout << endl;  Pause(0);

    // Copy constructor invoked here because object r2
    // is being initialized with a copy of r:
    Reminder r2 = r;
    r2.Display();  cout << endl;  Pause(0);

    // Copy constructor invoked here because (once again)
    // an object r3 is being initialized with a copy of r:
    Reminder r3(r);
    r3.Display();  cout << endl;  Pause(0);

    // Copy constructor invoked here because object r3 is being
    // passed by value to the formal parameter r of DisplayReminderWithBanner:
    DisplayReminderWithBanner(r3);  cout << endl;  Pause(0);

    // Copy constructor invoked here because r is a local object
    // in the body of the Halloween function and is being returned:
    Halloween().Display();  cout << endl;  Pause(0);

    return 0;
}
```

18.4.12.1 Notes and discussion on TESTREM4.CPP

This program is designed to confirm that the copy constructor for the `Reminder` class really *is* invoked in each of the following situations:

- A `Reminder` object is passed by value
- A `Reminder` object is returned as a function value
- A `Reminder` object is initialized by another `Reminder` object in a declaration

Note the comments in the program that identify each situation, and note as well that there are *two* cases for initialization.

18.4.12.2 Follow-up hands-on activities for TESTREM4.CPP

□ Make sure you have copies of TESTREM4.CPP, REM4.H and REM4.CPP. Un-comment the code in the body of the copy constructor in REM4.CPP so that you can see when the copy constructor is invoked. The copy constructor is the main focus of this sample program, but you may wish to un-comment the analogous code in the other constructor(s) and/or the destructor as well.

◯ Instructor checkpoint 18.3

Module 19

Class inheritance and related issues: function redefinition, the slicing problem, and virtual functions

19.1 Objectives

- To understand what is meant by *class inheritance*, including the terms *base class* (also *parent class* or *superclass*) and *derived class* (also *child class* or *subclass*).
- To understand what it means to *redefine* a member function from a base class in a class derived from that base class.
- To understand what is meant by the *slicing problem*.
- To understand the difference between *static binding* and *dynamic binding* in the context of member functions of a class object.
- To understand the difficulties arising in the context of inheritance that are caused by the slicing problem and static binding, and to understand how these problems are overcome by using reference parameters and *virtual functions* which permit dynamic binding.

New C++ reserved word
`virtual`

- To understand what is meant by *polymorphism*.
- To understand the order of constructor and destructor invocation when class inheritance is involved.

- To understand what is meant by a *constructor initialization list* and know how to use one.

19.2 List of associated files

- TESTZT.CPP is a test driver for the derived class `ZoneTime` defined in ZONETIME.H and ZONETIME.CPP.
- TIME.H and TIME.CPP provide a very simple version of the `Time` class to use as the base class for illustrating inheritance.
- ZONETIME.H is the specification file for the derived class ZoneTime inherited from the `Time` base class in TIME.H.
- ZONETIME.CPP is the implementation file for ZONETIME.H.
- TIMESLIC.CPP illustrates the "slicing problem".
- TIMEVF.CPP is used to illustrate static binding, dynamic binding, and polymorphism via virtual functions.

19.3 Overview

This Module deals with the concept of *class inheritance*, a very powerful and useful notion that is one of the cornerstones of all object-oriented programming languages, including C++.

Much of the effort in modern software development methodologies has gone toward *code reuse*, the idea that if we write our code in a certain way and are careful about how we do it, then at least some of that code should be useful later in other situations. If so, then this increases our productivity by letting us avoid writing that code again ("re-inventing the wheel", so to speak).

Let's take our own situation. We now know how to define and use classes, the `Time` class for example. But suppose we expand our horizons and start to do business in different time zones. All of a sudden our `Time` class is no longer adequate for keeping track of time among our far-flung business ventures.

What to do? Well, we could start from scratch and re-do the `Time` class. We might even be tempted to "cut and paste" a lot of the code, since surely some of it would be the same. But also some of it would likely be different, and then we would have to start modifying perfectly good and already-tested code that would therefore all have to be tested again, with all the difficulties that would entail. And if this were necessary for all of our other classes that weren't quite right for new situations, the result (as many programmers have discovered, much to their chagrin) would be a nightmare.

Fortunately, *inheritance* is the "white knight" that rides in to the rescue in situations like this. If the required new class is related to the pre-existing one in such a way that it can be formed (*derived*) from the existing one by adding new data members and possibly new, or replaced (*redefined*) operations, then *class inheritance* allows all of the code from the pre-existing (and tested) class to be used for our new class without change. All we have to do is add the new features to get the new class.

The devil is in the details, of course, and (as usual) there are "issues" that must be considered.

In TIME.H/TIME.CPP we define a `Time` class which is actually pretty much the same as our very simple original `Time` class, except that it contains the four-in-one constructor. This will be the *base class* which is inherited by the *derived class* `ZoneTime` defined in ZONETIME.H/ZONETIME.CPP and which will show you the syntax of inheritance. The test driver in TESTZT.CPP gives you an opportunity to see the `ZoneTime` class in action.

The relationship of inheritance between classes is often referred to as the "is-a" relationship. For example, any `ZoneTime` time (a `Time` value with a time zone attached to it) also "is a" `Time` time. Or, put another way, there is nothing a `Time` object has or can do that a `TimeZone` object does not also have, or cannot also do. A `TimeZone` object will generally have more and/or be able to do more, but it has and can do at least as much as a `Time` object.

Because any object of a derived class "is-a" object of its base class, we may be permitted to, and in fact sometimes do, use objects of a derived class in situations where we have said we were going to use objects of the corresponding base class.

For example, we might have written a function that displays a `Time` value within a fancy banner and (of course) we would want the `Time` value to be passed as a parameter to such a function. But what if we now pass a `ZoneTime` value (which, don't forget, "is a" `Time` value) instead? Is this OK? Well, yes and no (or, no and yes), as we shall see. The point is, it *ought* to be OK, and it *will* be if we are careful to make it so. But, it just doesn't "happen".

So, we must be aware of some implications of doing this sort of thing and know how to avoid certain problems that might otherwise arise. This leads us to investigate the so-called "slicing problem" (TIMESLIC.CPP), and to follow this up with a look at *static binding* vs. *dynamic binding* and *polymorphism* via *virtual functions* (TIMEVF.CPP).

The notion of *polymorphism* is central to object-oriented programming (along with *encapsulation*, i.e. putting data and operations on the data together in a class), and *inheritance*. In a nutshell, polymorphism simply means that the same *message* can be sent to different objects (or equivalently, the same function call (or "method") can be applied to different objects), and each object will respond in the correct way that makes sense for that object. In the context of this module, that means that a `Display` function applied to a `Time` object may produce a different output than the identically-named `Display` function applied to a `ZoneTime` object, but each of these two kinds of objects can be relied upon to display itself properly.

19.4 Sample Programs and Other Files

19.4.1 TESTZT.CPP is a test driver for the derived class ZoneTime in ZONETIME.H and ZONETIME.CPP

```
// Filename: TESTZT.CPP
// Purpose:  Test driver the ZoneTime class.

#include <iostream>
using namespace std;

#include "ZONETIME.H"

int main()
{
    ZoneTime t1;
    ZoneTime t2(5);
    ZoneTime t3(5, 30);
    ZoneTime t4(5, 30, 10);
    ZoneTime t5(5, 30, 10, CDT);

    cout << endl;
    cout << "t1: ";  t1.Display();  cout << endl;
    cout << "t2: ";  t2.Display();  cout << endl;
    cout << "t3: ";  t3.Display();  cout << endl;
    cout << "t4: ";  t4.Display();  cout << endl;
    cout << "t5: ";  t5.Display();  cout << endl;
    cout << endl;

    ZoneTime t;
    t.Set(23, 59, 55, PST);
    cout << "t: ";  t.Display();  cout << endl;
    cout << endl;

    cout << "Incrementing t:" << endl;
    for (int count = 1; count <= 10; count++)
    {
        t.Display();
        cout << ' ';
        t.Increment();
        cout << endl;
    }
    cout << endl;

    return 0;
}
```

19.4.1.1 Notes and discussion on TESTZT.CPP

This program uses the derived class `ZoneTime`. The main thing to note is that to the client there is no indication that the class has been derived from some other class. That is, using a derived class is "transparent to the client" in that the client uses a derived class in exactly the same manner as any other class.

When you examine the ZoneTime class you will note that it admits five possible constructors, and all five are illustrated in this test driver.

19.4.1.2 Follow-up hands-on activities for TESTZT.CPP

□ Copy TESTZT.CPP and study the program in light of the preceding comments. Compile the program for later testing if you like, but wait until you have studied the `Time` and `ZoneTime` classes (which follow) before proceeding with the testing. Then come back (and make sure you do) and complete the following activity.

□ OK, here we go (provided you have completed your study of the `Time` and `ZoneTime` classes). Begin by generating the TESTZT.EXE executable. You will have to link TESTZT.OBJ with both ZONETIME.OBJ and TIME.OBJ. Run the executable and note the form of the output.

Do not continue with this activity until you have completed section 19.4.4.3.

Now un-comment the commented code in the implementation function bodies of the `Time` and `ZoneTime` constructors. Re-compile both classes and regenerate TESTZT.EXE. Note when you run the program this time that the output should confirm the following statement about the order of constructor invocation:

> If B is a base class and D is a derived class which inherits from B and we make the declaration
> `D object;`
> then the first constructor to be invoked is that of the base class B, followed by the constructor of the derived class D.

Let's make the following observation in passing as well: In this case, neither the base class Time nor the derived class ZoneTime has a destructor, since neither class needs one. If they did have destructors, however, each destructor would be called in the reverse order of its corresponding constructor. As an exercise, you may want to verify this by writing a "do nothing" destructor for each of the two classes which simply outputs a message to the effect it has been called and run the program once again. This is the sort of exercise that takes a little extra time, but in the long run enough of them prevent a certain part of your brain from turning to mush much sooner than it otherwise would.

◯ Instructor checkpoint 19.1

19.4.2 TIME.H and TIME.CPP provide a very simple version of the Time class to use as the base class for illustrating inheritance

```
// Filename: TIME.H
// Purpose:  Specification file for a very simple Time class
//           to serve as base class for illustrating inheritance.

#ifndef TIME_H
#define TIME_H

class Time
{
public:

    Time(/* in */ int hours = 0,
         /* in */ int minutes = 0,
         /* in */ int seconds = 0);
    void Set(/* in */ int hours = 0,
             /* in */ int minutes = 0,
             /* in */ int seconds = 0);
    void Increment();
    void Display() const;


private:

    int hours;
    int minutes;
    int seconds;
};

#endif
```

Implementation file TIME.CPP starts below heavy line:

```
// Filename: TIME.CPP
// Purpose:  Implementation file corresponding to TIME.H.


#include <iostream>
using namespace std;


#include "TIME.H"
#include "PAUSE.H"


Time::Time(/* in */ int hours,
           /* in */ int minutes,
           /* in */ int seconds)
{
    this->hours = hours;
    this->minutes = minutes;
    this->seconds = seconds;

    // cout << "Now in constructor for base class Time ... \n";
    // Pause(0);  cout << endl;
}
```

```
void Time::Set(/* in */ int hours,
               /* in */ int minutes,
               /* in */ int seconds)
{
    this->hours = hours;
    this->minutes = minutes;
    this->seconds = seconds;
}



void Time::Increment()
{
    seconds++;
    if (seconds > 59)
    {
        seconds = 0;
        minutes++;
        if (minutes > 59)
        {
            minutes = 0;
            hours++;
            if (hours > 23)
                hours = 0;
        }
    }
}



void Time::Display() const
{
    if (hours < 10)
        cout << '0';
    cout << hours << ':';
    if (minutes < 10)
        cout << '0';
    cout << minutes << ':';
    if (seconds < 10)
        cout << '0';
    cout << seconds;
}
```

19.4.2.1 Notes and discussion on TIME.H and TIME.CPP

This is just a simplified variation on our old friend the `Time` class, which we are going to use as a base class to illustrate inheritance. By now you are familiar enough with the pre/post conditions that we have left them out, to keep the lines of code to a minimum.

19.4.2.2 Follow-up hands-on activities for TIME.H and TIME.CPP

□ Copy TIME.H and TIME.CPP and study the code. Note the use of default parameter values with `Set` this time around.

19.4.3 ZONETIME.H is the specification file for the derived class ZoneTime inherited from the Time base class in TIME.H

```
// Filename: ZONETIME.H
// Purpose:  Specification file for the "derived" ZoneTime class.
//           This class is "derived from" the "base class" Time.

#ifndef ZONETIME_H
#define ZONETIME_H

#include "TIME.H"

enum ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT};

class ZoneTime : public Time
{
public:

    ZoneTime(/* in */ int      hours = 0,
             /* in */ int      minutes = 0,
             /* in */ int      seconds = 0,
             /* in */ ZoneType zone = EST);
    // Pre:      0 <= hours   <= 23  and  0 <= minutes <= 59
    //       and 0 <= seconds <= 59  and  zone has been assigned
    // Post: Class object is constructed and value of
    //       ZoneTime is set according to the incoming parameters

    // Note that the Increment function does not appear here.
    // The reason is that, unlike the Set and Display functions,
    // the Increment function is *not* being "overridden" with a
    // new definition in this derived ZoneTime class.  Or, looked
    // at another way, to increment a ZoneTime value means exactly
    // the same thing as incrementing a Time value so there is no
    // need to have a new version of the Increment function.  This
    // is, of course, one of the advantages of inheritance.

    // Note that this is a new Set function, since the time zone
    // needs to be set now as well.
    void Set(/* in */ int      hours = 0,
             /* in */ int      minutes = 0,
             /* in */ int      seconds = 0,
             /* in */ ZoneType zone = EST);
    // Pre:      0 <= hours   <= 23  and  0 <= minutes <= 59
    //       and 0 <= seconds <= 59  and  zone has been assigned
    // Post: ZoneTime is set according to the incoming parameters

    // Note that this is a new Display function, since the time zone
    // needs to be displayed as part of the time value now as well.
    void Display() const;
    // Pre:  self is initialized.
    // Post: ZoneTime has been output in the form HH:MM:SS ZZZ
    //       where ZZZ is the time zone


private:

    ZoneType zone;
};

#endif
```

19.4.3.1 What you see for the first time in ZONETIME.H

Here are the things you should note about the specification of the `ZoneTime` class that are relevant to *inheritance*:

- The first line of the class definition is

  ```
  class ZoneTime : public Time
  ```

  which says that the class `ZoneTime` is going to inherit from the class `Time` and the inheritance is going to be `public`. There are other forms of inheritance, but this is the most common and useful, and essentially it means that everything in the `public` interface to the `Time` class will be automatically available[1] to clients of the `ZoneTime` class, except for those things that are *redefined*, in the `ZoneTime` class. More about what this means immediately below.

- The two functions `Set` and `Display` appear in the specification file for the `ZoneTime` class, but the `Increment` function does not. What's going on here? Well, setting and displaying a `ZoneTime` value are *different* from setting and displaying a `Time` value, because the zone value is involved in each of these operations. On the other hand, incrementing a `ZoneTime` value is the same as incrementing a `Time` value, so we can just use the `Increment` function from the `Time` class, which is readily available to us in the `ZoneTime` class. Thus we do not have to redefine the `Increment` function in the new class, like we did the `Set` and `Display` functions.

- This *is* a new class with a new name, so of course it needs constructors that conform to the usual rules. But building a `ZoneTime` object involves building a `Time` object first (recall that a `ZoneTime` time "is a" `Time` time), so the constructor for `ZoneTime` makes good use of the constructor for `Time`, as you will see when you examine the implementation of the `ZoneTime` class.

19.4.3.2 Additional notes and discussion on ZONETIME.H

Note that TIME.H must be included in ZONETIME.H since the `Time` class is used in the definition of the `ZoneTime` class. Also, the definition of the "auxiliary" enumerated type `ZoneType` that defines the time zone constants is placed outsize and before the class so that it may be used by both the class and its clients.

19.4.3.3 Follow-up hands-on activities for ZONETIME.H

□ Copy ZONETIME.H and study the code.

[1] But remember that inheriting a class does not give access to its private data members. This would defeat the whole purpose of *encapsulation*, if we could get access to a class's private data members just by inheriting that class.

19.4.4 ZONETIME.CPP is the implementation file corresponding to ZONETIME.H

```
// Filename: ZONETIME.CPP
// Purpose:  Implementation file corresponding to ZONETIME.H.

#include <iostream>
using namespace std;

#include "ZONETIME.H"
#include "PAUSE.H"

// Additional private members of class ZoneTime:
//     ZoneType zone;


//*****************************************************************
ZoneTime::ZoneTime(/* in */ int      hours,
                   /* in */ int      minutes,
                   /* in */ int      seconds,
                   /* in */ ZoneType zone)
         :Time(hours, minutes, seconds)
// Pre:      0 <= hours   <= 23  and  0 <= minutes <= 59
//       and 0 <= seconds <= 59  and  zone has been assigned
// Post: Class object is constructed and value of
//       ZoneTime is set according to the incoming parameters
{
    this->zone = zone;
    // cout << "Now in constructor for derived class ZoneTime ... \n";
    // Pause(0);
}


//*****************************************************************
void ZoneTime::Set(/* in */ int      hours,
                   /* in */ int      minutes,
                   /* in */ int      seconds,
                   /* in */ ZoneType timeZone)
// Pre:      0 <= hours   <= 23  and  0 <= minutes <= 59
//       and 0 <= seconds <= 59  and  newZone has been assigned
// Post: ZoneTime is set according to the incoming parameters
{
    Time::Set(hours, minutes, seconds);
    zone = timeZone;
}


//*****************************************************************
void ZoneTime::Display() const
// Pre:  self is initialized.
// Post: Time has been output in the form HH:MM:SS ZZZ
//       where ZZZ is the time zone
{
    static char zoneString[8][4] =
    {
        "EST", "CST", "MST", "PST", "EDT", "CDT", "MDT", "PDT"
    };
    Time::Display();
    cout << ' ' << zoneString[zone];
}
```

19.4.4.1 What you see for the first time in ZONETIME.CPP

Here are the things you should note about the implementation of the `ZoneTime` class that are relevant to *inheritance*:

- The first thing you need to get a handle on is how constructors behave under inheritance. Since building a `ZoneTime` object involves building a `Time` object first and then adding the "extras" to it, this means that a constructor for a `Time` object is called first. But, we hear you asking, "*Which* constructor for `Time` is called?" That's where the *constructor initialization list* comes into play. In our case this list consists of just one item, namely

  ```
  :Time(hours, minutes, seconds)
  ```

  which appears immediately after the header of the `ZoneTime` constructor. Now, and here's the neat part, whatever actual parameters (from `hours`, `minutes` and `seconds`) appear in the `ZoneTime` constructor parameter list determine which `Time` constructor is invoked, and those parameter values are then passed along to *that* `Time` constructor, with default parameter values working just like you'd hope they would. Whew!

  And once the `Time` constructor has done its job, the only thing left for the `ZoneTime` constructor to do is set the zone, so that's why there's only that one statement to do this in the body of the `ZoneTime` constructor.

- The second thing you need to get a handle on is the way in which functions in the public interface of the base class are accessed in the derived class, if necessary. We see this in two cases here: `Time::Set` is used in the body of the `Set` function and `Time::Display` is used in the body of the `Display` function. The prefix `Time::` indicates that in each case we are using the function from the `Time` class to set or display the `Time` part of the `ZoneTime` object, after which all that remains for the `ZoneTime Set` or `Display` is to set or display the zone.

19.4.4.2 Additional notes and discussion on ZONETIME.CPP

Recall that in our discussion of ZONETIME.H we mentioned that inheritance does *not* give the derived class member functions access to the private data members of the base class. This is why, when we "override"[2] the `Time` class definitions of `Set` and `Display`, we are "forced" to use those functions from the `Time` class in the way indicated to help in setting and/or displaying `ZoneTime` values. On the other hand, if we had accessor functions that returned the values of `Time`'s private data members we would have more flexibility in the way we could write `Set` and/or `Display` for `ZoneTime`.

[2]Some C++ authors use the term override where we have used redefine; others use override only in the context of virtual functions, which we discuss shortly.

19.4.4.3 Follow-up hands-on activities for ZONETIME.CPP

Now go back and finish the activity in section 19.4.1.2.

□ Copy ZONETIME.CPP and study the code. When you have finished studying both the `Time` and `ZoneTime` classes (specification and implementation), be sure to go back and complete the last activity for TESTZT.CPP.

◯ Instructor checkpoint 19.2

19.4.5 TIMESLIC.CPP illustrates the "slicing problem"

```
// Filename: TIMESLIC.CPP
// Purpose:  Illustrates the "slicing problem", one of the
//           problems that virtual functions help to solve.

#include <iostream>
using namespace std;

#include "TIME.H"
#include "ZONETIME.H"


void DisplayTimeWithBanner(/* in */ Time someTime)
// Pre:  someTime contains a Time object or an object of a derived class.
// Post: The value in someTime has been displayed in "banner form".
{
    cout << "**************************\n";
    cout << "** The time is ";
    someTime.Display();  cout << endl;
    cout << "**************************\n\n";
}


int main()
{
    Time myTime(8, 30, 0);
    ZoneTime yourTime(10, 45, 0, CST);

    cout << endl;
    DisplayTimeWithBanner(myTime);
    // Output is what you would expect, since myTime
    // is an object from the class Time.

    DisplayTimeWithBanner(yourTime);
    // Output is not what you would hope for.  Since yourTime is
    // an object of class ZoneTime, and is thus also a Time (any
    // ZoneTime "is a" Time, because of inheritance), you would
    // hope that the value of the ZoneTime would be printed out.
    // Unfortunately, the ZoneTime value is "bigger than" the
    // Time value (occupies more storage space in memory because
    // of the additional member variable) but the "extra part" is
    // "sliced off" as it is passed to the DisplayTimeWithBanner
    // function because it is passed by value and a copy is made to
    // a local variable only capable of holding a value of type Time.

    // Question: Would passing by reference solve this problem?
    // Answer:   See TIMEVF.CPP.

    return 0;
}
```

19.4.5.1 What you see for the first time in TIMESLIC.CPP

This program illustrates the kind of problem we can have if we abuse the "is-a" relationship that exists between classes when one of them is derived from the other.

We have described the problem in some detail in the comments of the program itself. So, you should read those comments carefully, and we will not repeat them here.

Before looking ahead you might want to think about what you could do to solve the problem.

19.4.5.2 Additional notes and discussion on TIMESLIC.CPP

Note that *any* `ZoneTime` value has `zone` as a data member. That data member may be initialized by default or with a value that we supply, depending on what version of the `ZoneTime` constructor we use to initialize the `ZoneTime` object. But in any case, a `ZoneTime` object is always "bigger than" the corresponding `Time` object (i.e., occupies more storage in memory), because of this additional data member. So, whenever we try to put a `ZoneTime` object into a storage area reserved for a `Time` object, the ZoneTime object will be too big (it "won't fit", so to speak) and the extra `zone` data member will be "sliced off". This is the essence of the so-called *slicing problem*.

19.4.5.3 Follow-up hands-on activities for TIMESLIC.CPP

□ Copy, study and test the program in TIMESLIC.CPP. Be sure you understand the output in the context of the discussion of this program.

◯ Instructor checkpoint 19.3

19.4.6 TIMEVF.CPP illustrates static binding, dynamic binding, and polymorphism via virtual functions

```
// Filename: TIMEVF.CPP
// iurpose:  Illustrates the "static binding" problem that
//           prevents "passing by reference" from solving
//           the "slicing problem" we had in TIMESLIC.CPP.

#include <iostream>
using namespace std;

#include "TIME.H"
#include "ZONETIME.H"


void DisplayTimeWithBanner(/* in */ Time& someTime)
// Pre:  someTime contains a Time object or an object of a derived class.
// Post: The value in someTime has been displayed in "banner form".
{
    cout << "**************************" << endl;
    cout << "** The time is ";
    someTime.Display();
    cout << endl;
    cout << "**************************" << endl << endl;
}


int main()
{
    Time myTime(8, 30, 0);
    ZoneTime yourTime(10, 45, 0, CST);

    cout << endl;
    DisplayTimeWithBanner(myTime);
    // This call continues to work as before, as expected.

    DisplayTimeWithBanner(yourTime);
    // The hoped-for improvement does not happen, and once again
    // we do not get the time zone part of the ZoneTime object
    // printed out.  The reason is that even though no "slicing"
    // takes place this time (because of the passing by reference)
    // the compiler still has to decide *at compile-time* which
    // Display() function is going to be used.  Because the parameter
    // being passed is of type Time and "static binding" is used,
    // the Display() from the Time class is chosen.

    // Question: What to do?
    // Answer:   Go back to the Time class specification file and
    //           make the Display() function a "virtual function"
    //           by placing the keyword "virtual" in front of its
    //           prototype.  This tells the compiler to insert the
    //           necessary code to allow the decision to be made at
    //           run-time as to which version of Display() will be.
    //           called.  This solves our problem.

    return 0;
}
```

19.4.6.1 What you see for the first time in TIMEVF.CPP

If you examine the `DisplayTimeWithBanner` function in this program and compare it with the one in TIMESLIC.CPP you will note that the only difference is that the `Time` parameter has been made a reference parameter.

You may think (or may have thought, if indeed you did think about it beforehand) that this would solve the *slicing problem*. That's a perfectly reasonable assumption. In fact, it does solve the slicing problem, since if we just pass the address no copy is made and so nothing gets "sliced off".

But, unfortunately, another problem now intrudes, as explained once again in the program's comments: The compiler has to make one decision *at compile-time* as to which `Display` function is going to be used in the `DisplayTimeWithBanner` function, and because we said we were going to pass a `Time` object to this function it quite naturally chooses the `Display` function for `Time`. This is described by saying that the `Display` function is *statically bound*.

However, what we really want to happen is for the program to decide which `Display` function should be used on the basis of the calls to the function and the actual parameter object (`Time` or `ZoneTime`) being passed at the time of the call. That is, we want `Display` to be dynamically bound (or *overridden* with a new—and more appropriate—definition) at run-time.

This is polymorphism in action.

There is no way we can make this happen in the driver. We have to go all the way back to the `Time` class and make the `Display` function in that class "virtual" by placing the reserved word `virtual` in front of its prototype as follows:

```
virtual void Display() const;
```

This relatively small change, remarkably, has exactly the effect that we want. The choice of which `Display` function to use will now be made on the basis

19.4.6.2 Additional notes and discussion on TIMEVF.CPP

Don't forget that once you have made the Display function in the Time class virtual, it will be necessary to re-compile that class as well as the ZoneTime class and the driver program. More generally, when a change in made in a base class, it is always necessary to re-compile (and re-link) all files that depend in some way on the changed file. In some programming environments all this is handled automatically, in others it is up to the programmer to keep track of what is going on and make sure everything is "up to date".

19.4.6.3 Follow-up hands-on activities for TIMEVF.CPP

□ Copy, study and test the program in TIMEVF.CPP. Generate the exectuable TIMEVF.EXE and run it to show that in fact the output is not what you would like for yourTime. Then make the Display function in the Time class virtual, in the way indicated above, re-compile, re-link and re-run to show that the problem goes away and you have the desired output in both cases.

◯ Instructor checkpoint 19.4

Module 20

Class composition: classes with class object data members

20.1 Objectives

- To understand the new issues that arise when one or more data members of a class are themselves objects of another class, a class relationship referred to a *class composition*.
- To understand how to use a *constructor initialization list* for object data members, and to understand how such a constructor initialization list is similar to, and different from, the one used in the context of inheritance.

20.2 List of associated files

- TESTAPP.CPP is a test driver for the class `Appointment` class defined in APPOINT.H and APPOINT.CPP.
- APPOINT.H is the specification file for an `Appointment` class that contains as one of its data members an object of class `Time`.
- APPOINT.CPP is the implementation file for APPOINT.H.
- TIME.H and TIME.CPP are the files of the same name from Module 19.

  These last two files are needed since a `Time` object is one of the data members of the `Appointment` class.

20.3 Overview

When several (or many) classes are involved in a program that solves a particular problem, various relationships between those classes are possible. There may be no relationship at all between two classes, one class may be derived from another (as discussed in Module 19), or a class may have as one or more of its data members objects of other classes. This latter situation is referred to as *class composition*[1] and is the subject of this Module.

20.4 Sample Programs and Other Files

20.4.1 TESTAPP.CPP is a test driver for the Appointment class defined in APPOINT.H and APPOINT.CPP

```
// Filename: TESTAPP.CPP
// Purpose:  Test driver for the Appointment class.

#include <iostream>
using namespace std;

#include "APPOINT.H"
#include "PAUSE.H"

int main()
{
    cout << endl;

    Appointment ap1;
    ap1.Display();  cout << endl;
    Pause(0);  cout << endl;

    Appointment ap2("Bill Gates", 7, 25, 30);
    ap2.Display();  cout << endl;
    Pause(0);

    return 0;
}
```

20.4.1.1 Notes and discussion on TESTAPP.CPP

This program simply shows client code using objects of an `Appointment` class, which has at least two constructors, the default one and one with four parameters. There is nothing about the class to indicate to the client whether or not any of the data members consist of objects of other classes, and indeed such matters should be of no interest to a client in any case.

[1] In the high-tech real world of object-oriented design and development, where they use something called *UML* (*Unified Modeling Language*), there's a more technical meaning of *composition*, but this will do for us here.

20.4.1.2 Follow-up hands-on activities for TESTAPP.CPP

□ Copy TESTAPP.CPP and study the code. Once again compile TESTAPP.CPP in preparation for testing, if you like, but wait till you've looked at APPOINT.H and APPOINT.CPP before proceeding. then come back and complete the following activities (and make sure that you do).

□ If you are continuing with this and the following activities, we assume you have studied the `Appointment` class found in APPOINT.H and APPOINT.CPP.

Don't proceed with these activities until you have finished those in section 20.4.3.3.

Begin by making a copy of the file TESTAPP.CPP. Call it TESTAPP1.CPP. At the moment this file only shows use of the default constructor and the four-parameter constructor, so add to TESTAPP1.CPP the object declarations

```
Appointment ap3("Nklaus Wirth");
Appointment ap4("Steve Jobs", 3);
Appointment ap5("Alan Kay", 12, 45);
```

as well as code to display the data in each of these "`Appointment` objects" and hence test the other constructor options. Then generate TESTAPP1.EXE using the class in APPOINT.H/APPOINT.CPP and run the program to make sure you get the anticipated output.

□ Repeat the previous activity but this time when you generate TESTAPP1.EXE use the revised class in APPOINT1.CPP (the one you created later in this Module before coming back here to finish these activities). This will show that the two-item constructor initialization list actually worked.

□ Finally, un-comment the commented code in the body of the implementation of the `Appointment` constructor in APPOINT.CPP and then once again regenerate TESTAPP.EXE (*not* TESTAPP1.EXE this time; we're back to the original driver) using this newly revised version of the original APPOINT.CPP. Also make sure that the analogous code is un-commented in the TIME.CPP implementation file, since the `Time` class is also required and you will want to see when *its* constructors are being called as well, when you run the program. When you run the program, make sure the output confirms the following statement about the order of constructor invocation:

> If `C` is a class and `memberObject` is an object data member of that class and we make the declaration
> `C object;`
> then the first constructor to be invoked is that of the `memberObject` object , followed by the constructor of the class `C`.

And, once again, if there were destructors, each destructor would be called in the reverse order of its corresponding constructor. As in the case of inheritance, you could again verify this by writing "do nothing" constructors that only reported the fact they were being called.

◯ Instructor checkpoint 20.1

20.4.2 APPOINT.H is the specification file for an Appointment class containing as a data member an object of class Time

```
// Filename: APPOINT.H
// Purpose:  Specification file for an Appointment class.

#ifndef APPOINT_H
#define APPOINT_H

#include <string>
using namespace std;
#include "TIME.H"


class Appointment
{
public:

    Appointment(/* in */ string name = "No name ...",
                /* in */ int  hours = 0,
                /* in */ int  minutes = 0,
                /* in */ int  seconds = 0);
    // Constructor
    // Pre:  "name" is assigned       and  0 <= hours <= 23
    //       and  0 <= minutes <= 59  and  0 <= seconds <= 59
    // Post: name == newName             and  hours == newHours
    //       and  minutes == newMinutes  and  secs == newSeconds

    void Display() const;
    // Pre:  self has been initialized.
    // Post: Appointment information has been output in the form
    //       Meeting with: John Smith
    //       Meeting time: 08:14:25


private:

    string name;
    Time   meetingTime;
};

#endif
```

20.4.2.1 Notes and discussion on APPOINT.H

Not much new here either. But do observe that we have a five-in-one constructor with default parameter values and that in fact *both* data members of this class are objects of other classes: one is an object of class string from the Standard Library, and the other is an object of our very own Time class (which, by the way, is what requires the inclusion of TIME.H).

20.4.2.2 Follow-up hands-on activities for APPOINT.H

□ Copy APPOINT.H and study the code.

20.4.3 APPOINT.CPP is the implementation file for APPOINT.H

```
// Filename: APPOINT.CPP
// Purpose:  Implementation file corresponding to APPOINT.H.

#include <iostream>
using namespace std;

#include "APPOINT.H"
#include "PAUSE.H"


// Private members of class:
//    string name;
//    Time   meetingTime;


//*****************************************************************
Appointment::Appointment(/* in */ string name,
                         /* in */ int hours,
                         /* in */ int minutes,
                         /* in */ int seconds)
           :meetingTime(hours, minutes, seconds)
// Constructor
// Pre:  "name" is assigned       and  0 <= hours <= 23
//       and  0 <= minutes <= 59  and  0 <= seconds <= 59
// Post: name == newName             and  hours == newHours
//       and  minutes == newMinutes  and  secs == newSeconds
{
    this->name = name;

    // cout << "Now in constructor for class Appointment ... \n";
    // Pause(0);  cout << endl;
}


//*****************************************************************
void Appointment::Display() const
// Pre:  self has been initialized.
// Post: Appointment information has been output in the form
//       Meeting with: John Smith
//       Meeting time: 08:14:25
{
    cout << "Metting with: " << name << endl;
    cout << "Meeting time: "; meetingTime.Display(); cout << endl;
}
```

20.4.3.1 What you see for the first time in APPOINT.CPP

The only thing new in this program is another *constructor initialization list*. But this one is different from the one we saw in Module 19, where we were looking at inheritance, and where the constructor initialization list involved use of the name of the base class to invoke a base class constructor.

The constructor initialization list we are dealing with here contains not the name of the `Time` *class* but the name of a `Time` *object*, which is a data member of the `Appointment` class. But note that it too has a parameter list which invokes the appropriate `Time` constructor to initialize that data member, and once again (as was the case with inheritance) the parameters actually supplied to the `Appointment` object will be passed along to the `Time` object and will determine which `Time` constructor is called.

20.4.3.2 Additional notes and discussion on APPOINT.CPP

The general form of a constructor initialization list is

```
:object1(par_list), object2(par_list2), ..., objectN(par_listN)
```

which would suggest that, in the case of `Appointment`, we could also remove the initialization of `name` from the body of the constructor, add it to the initialization list, and have an empty body. We can actually do this,[2] and here's the resulting constructor header:

```
Appointment::Appointment(/* in */ string name,
                         /* in */ int hours,
                         /* in */ int minutes,
                         /* in */ int seconds)
            :meetingTime(hours, minutes, seconds),
             name(name)
```

"Hey, wait a minute!" you say. Doesn't C++ get confused now because we are using the same "name" for the parameter `name` and the private data member `name`? Actually, no. C++ is smart enough to determine from context (whether it's inside the parentheses or outside) which `name` it's dealing with. But there's no doubt that if we're going to do this then for readability we should rename the parameter `name` to be something different like `nameInit`.

20.4.3.3 Follow-up hands-on activities for APPOINT.CPP

☐ Copy APPOINT.CPP and study the code.

When you have finished here, go back and complete the activities in 20.4.1.2.

☐ Make a copy of APPOINT.CPP and call it APPOINT1.CPP. Revise the copy so that it has the two-item constructor initialization list suggested above. Compile to make sure you have done this correctly, and then return to complete the remaining activities for TESTAPP.CPP.

◯ Instructor checkpoint 20.2

[2] And by the way, we could have done it in the constructor initialization list that we used with inheritance in Module 19 as well.

Module 21

Linked structures (our seventh structured data type)

21.1 Objectives

- To understand what is meant by a *linked data structure*.
- To understand what is meant by a *sequence of linked nodes*.
- To get some programming practice with linked nodes.

21.2 List of associated files

- LINKINT.CPP illustrates a sequence of linked nodes containing integer data values.
- LNODEDRV.CPP is a test driver for the functions dealing with linked nodes that are contained in LNODEFUN.H and LNODEFUN.OBJ.
- LNODEFUN.H contains the prototypes of the functions implemented in LNODEFUN.OBJ and called by LNODEDRV.CPP. (The code for some of these appears in LNODEFUN.SHL.). Note that this is *not* a class specification file.
- LNODEFUN.OBJ contains implementations of all of the functions in LNODEFUN.H.
- LNODEFUN.SHL is a shell of source code containing many, but not all, of the functions whose prototypes are in LNODEFUN.H, whose implementations are in LNODEFUN.OBJ, and which are called by LNODEDRV.CPP.

21.3 Overview

In Module 10 we introduced pointers but only used them to point at "regular" variables that were already hanging around. In Module 17 we used them to point at newly allocated storage from the heap (dynamic storage) which could be either a simple variable or an array.

In this Module we go one step further, and this is where pointers really come into their own. We create structures that look like the one in Figure 21.1, which is actually a picture of what you would have if you ran the first sample program (LINKINT.CPP) and entered the values 17, 4 and 23 from the keyboard.

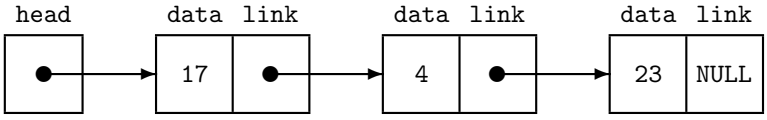

Figure 21.1: A Sequence of Linked Nodes

Each of those box-arrow combinations looking like the one in Figure 21.2 is called a *node*, and the complete structure in Figure 21.1 is itself called a *sequence of linked nodes*.[1]

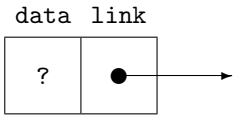

Figure 21.2: A Node

For a sequence of linked nodes the following three features are critical:

- There must be a pointer to the first or (`head`) node.
- Every node except the last must contain a pointer (or `link`) to the next node in the sequence.
- The last node must contain the `NULL` pointer as its `link` value to indicate that this node *is* the last one in the sequence.

The pictures suggest that an appropriate C++ representation for a node might be a `struct` with two fields—one for the data and one for the link—and in fact that is what we do use. In these diagrams the arrow pointing out from the link field is just a pictorial representation of an address in memory—the address of the next node in the sequence.

[1] This is our preferred terminology (see page 2 for the rationale). Many authors call such a thing a *linked list*. Well, OK, I guess.

We could also use a `class` instead of a `struct` to represent a node, but that would make matters somewhat more complex, and for the moment we'd like to keep things as simple as possible while we let you concentrate on the logistics of linked node manipulation.

Note that, for emphasis, we have actually placed the word `NULL` in the link field of the last node of Figure 21.1, but most authors will use simply a diagonal line going from the bottom left-hand corner to the top right-hand corner of the link-field box, which is certainly simpler and we recommend that you use that convention when drawing your pictures.

Our C++ naming conventions for the structure we will use to deal with sequences of linked nodes are illustrated by this extract from the sample program in LINKINT.CPP:

```
typedef int DataType;
struct  Node
{
    DataType data;
    Node* link;
};
typedef Node* NodePointer;
```

This representation is quite general, in that we can easily have a sequence of linked nodes of arbitrary complexity containing whatever we like, just by changing the definition of `DataType` (making it a hierarchical struct, for example).

The above definition is *self-referential*. That is, the link field contains a pointer-to-self, which is permitted (in this context) by C++. An alternative form of the above syntax, which makes use of a *forward declaration* (of the struct data type `Node`, whose name is used before it is actually defined), is given below.

```
typedef int DataType;
struct  Node;
typedef Node* NodePointer;
struct  Node
{
    DataType data;
    NodePointer link;
};
```

The wonderful thing about working with linked structures is that there is *always* a nice picture that shows what you're doing, if you will only take the time to draw it. You should treat these pictures as an integral part of your work with linked structures and plan to add to your ever-expanding toolkit the skill of being able to draw these pictures quickly and accurately.

21.4 Sample Programs and Other Files

21.4.1 LINKINT.CPP illustrates a sequence of linked nodes containing integer data

```
// Filename: LINKINT.CPP
// Purpose:  Illustrate a sequence of linked nodes.

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

typedef int DataType;
struct  Node
{
    DataType data;
    Node* link;
};
typedef Node* NodePointer;


int main()
{
    cout << "\nThis program builds a sequence of "
         << "linked nodes of integers of size 3 with "
         << "\nvalues read from the keyboard and "
         << "then prints out values."
         << "\nEnter three values on the following "
         << "line and then press ENTER: \n";

    NodePointer head, current, next;

    head = new Node;      // Get a first node
    current = head;       // Make current point at the first node
    cin >> current->data; // Read a value into the first node

    next = new Node;      // Get a new node (the second one)
    current->link = next; // Attach the current (first) node to the second
    current = next;       // Make current point at the second node
    cin >> current->data; // Read a value into the second node

    next = new Node;      // Get a new node (the third one)
    current->link = next; // Attach the current (second) node to the third
    current = next;       // Make current point at the third node
    cin >> current->data; // Read a value into the third node

    current->link = NULL; // Make sure sequence is terminated properly

    // Output the data value in each node
    cout << endl;
    current = head; // Make current point at head of sequence
    while (current != NULL)  // while the end of sequence not reached
    {
        cout << setw(5) << current->data; // Output value pointed at by current
        current = current->link;          // Move current pointer to next node
    }
    cout << endl << endl;

    return 0;
}
```

21.4.1.1 What you see for the first time in LINKINT.CPP

This program shows how to start, and then continue, building a sequence of linked nodes: getting a first node and establishing it as the "head" of the sequence, putting some data in that first node, getting another node and attaching it to the end of the sequence-so-far, putting some data in this node, getting another node ... and finally making sure the last node "points to `NULL`".

21.4.1.2 Additional notes and discussion on LINKINT.CPP

This is a very simple program but the steps it illustrates for the building of a sequence of linked nodes are absolutely typical and if you are going to work with linked nodes these steps must become second nature.

21.4.1.3 Follow-up hands-on activities for LINKINT.CPP

□ Copy, study and test the program in LINKINT.CPP. Run the program several times, with a different set of three input values each time.

□ Draw a pictorial trace of LINKINT.CPP. This is where you first get to practice drawing all those nice pictures.

□ Make a copy of LINKINT.CPP and call it LINKINT1.CPP. Collapse the code for building the sequence of linked nodes into a loop that will build a sequence of arbitrary size. Note that the code for displaying the values already works for a sequence of arbitrary size, but you may wish to modify it to display only 10 values per line, say.

◯ Instructor checkpoint 21.1

21.4.2 LNODEDRV.CPP is a test driver for the functions dealing with linked nodes that are contained in LNODEFUN.OBJ and LNODEFUN.SHL

```
// Filename: LNODEDRV.CPP
// Purpose:  Driver for a demonstration program to illustrate
//           operations on a sequence of linked nodes.

#include <iostream>
using namespace std;

#include "LNODEFUN.H"
#include "PAUSE.H"


int main()
{
    NodePointer head;
    int menuChoice;

    DescribeProgram();
    Pause(0);

    InitializeSequence(head);

    do
    {
        DisplayMenu();
        GetMenuChoice(menuChoice);
        switch (menuChoice)
        {
            case  1: /* Quit */                                 break;
            case  2: InitializeSequence(head);                  break;
            case  3: PrintNodeValues(head);
                     Pause(0);                                  break;
            case  4: BuildSequenceFixedSize(head);              break;
            case  5: BuildSequenceAnySize(head);                break;
            case  6: FindDataValues(head);                      break;
            case  7: FindPositions(head);                       break;
            case  8: InsertAfterNodeWithDataValue(head);        break;
            case  9: InsertAfterNodeK(head);                    break;
            case 10: InsertBeforeNodeWithDataValue(head);       break;
            case 11: InsertBeforeNodeK(head);                   break;
            case 12: DeleteNodeWithDataValue(head);             break;
            case 13: DeleteNodeK(head);                         break;
            case 14: ProcessAllNodes(head, Double);             break;
            case 15: ProcessAllNodes(head, Square);             break;
            case 16: ProcessAllNodes(head, ReverseAllDigits); break;
        }

    } while (menuChoice != 1);

    return 0;
}
```

21.4.2.1 What you see for the first time in LNODEDRV.CPP

The major new thing you see in this program (even though it doesn't exactly leap out at you) is embodied in the function call `ProcessAllNodes(head, Double);`, and the other two like it.

At first glance there may not appear to be anything new here, but there is in fact something quite new, different, and occasionally useful, as it is here. What's going on here is that although `head` is just the "usual" kind of actual data parameter that we have been passing to functions ever since we learned how to pass parameters, those other parameters (`Double`, `Square`, and `ReverseAllDigits`) are not that kind of parameter at all. Those three are *function parameters* rather than *data parameters* of the kind we have been passing.

Like any parameter, though, a function parameter supplies some information to the caller to use for the current call. So, when we "pass a function to a function", we are saying to the receiver, "Here, use this function that we've passing you for whatever you're doing this time."

21.4.2.2 Additional notes and discussion on LNODEDRV.CPP

This program is a driver for the collection of functions dealing with sequences of linked nodes that are available to you in LNODEFUN.OBJ. The header file corresponding to LNODEFUN.OBJ is LNODEFUN.H. Note that this is *not* a class header file, just a header file containing the function prototypes of all those functions in LNODEFUN.OBJ. Not all of those functions are actually called by the driver LNODEDRV.CPP itself; some are auxiliary or "helper" functions used by other functions in the collection.

21.4.2.3 Follow-up hands-on activities for LNODEDRV.CPP

□ Copy and study the program in LNODEDRV.CPP. From the menu and the rather long (but informative, we hope) function names you should be able to get a very good idea of what the program is capable of.

□ For this activity you will need copies of LNODEFUN.H and LNODEFUN.OBJ as well as LNODEDRV.CPP. You need not study LNODEFUN.H at this point, though you could cover the next section now before proceeding if you like. In any case, when you are ready, compile LNODEDRV.CPP and link with LNODEFUN.OBJ. Run the program, read the on-line help provided, and then experiment for a while with sequences of linked nodes of various sizes and containing various integer data values.

The idea is to become thoroughly familiar with the program from an "operational" point of view, so that later, when you are examining the code that performs the various operations and trying to understand it (or when you are trying yourself to write some of the code to perform these operations) you have a good sense of what the code is supposed to do.

◯ Instructor checkpoint 21.2

21.4.3 LNODEFUN.H contains the prototypes of the functions in LNODEFUN.OBJ and LNODEFUN.SHL

```
// Filename: LNODEFUN.H
// Purpose:  Header file of definitions and functions to be used
//           with the demonstration program in LNODEDRV.CPP.


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

typedef int DataType;
struct Node
{
    DataType data;
    Node* link;
};
typedef Node* NodePointer;


/*
Part of the documentation of each of the following functions is a
list of what functions are called by that function and/or what functions
that function calls. Unless otherwise indicated, any function missing this
documentation is called only by main and does not call any other functions.
*/



/*
The following three functions form the driver "program infrastructure",
but have nothing to do with sequences of linked nodes.
*/

void DescribeProgram();
// Pre:  none
// Post: The program description has been displayed.

void DisplayMenu();
// Pre:  none
// Post: A menu with 16 options has been displayed.

void GetMenuChoice(/* out */ int& menuChoice);
// Pre:  none
// Post: "menuChoice" contains an integer entered by the user
//       from the keyboard



/*
The following three functions are "utility" functions that initialize
a sequence, test if one is empty, and reclaim the dynamic storage alloted
to a sequence when that sequence is no longer needed.
*/

void InitializeSequence(/* inout */ NodePointer& head);
// Pre:  none
// Post: "head" has been set to NULL.
// Called by: BuildSequenceFixedSize, BuildSequenceAnySize
```

```
bool EmptySequence(/* in */ NodePointer head);
// Pre:  "head" has been initialized.
// Post: The function value is true if the sequence is empty,
//       and false otherwise.
// Called by: BuildSequenceFixedSize, BuildSequenceAnySize

void ReclaimStorage(/* in */ NodePointer head);
// Pre:  "head" has been initialized.
// Post: All storage (if any) in the node sequence pointed to by
//       "head" has been returned to the free store.
// Called by: BuildSequenceFixedSize, BuildSequenceAnySize


/*
The next two functions allow you to build a sequence of either
fixed or arbitray size.
*/

void BuildSequenceFixedSize(/* inout */ NodePointer& head);
// Pre:  none
// Post: The user has entered the number of nodes to be in a
//       newly constructed sequence, as well as the values to
//       be inserted into the sequence, the sequence has been
//       constructed, and "head" points at the first node.
//       All storage in any sequence previously pointed to by
//       "head" has been returned to the free store.


void BuildSequenceAnySize(/* inout */ NodePointer& head);
// Pre:  none
// Post: The user has entered the values to be inserted into
//       a newly constructed sequence (terminating with an
//       end-of-file), the sequence has been constructed, and
//       "head" points at the first node.  All storage in any
//       sequence previously pointed to by "head" has been
//       returned to the free store, and the standard input
//       stream has been readied for further input.
// Calls: InitializeSequence, EmptySequence, ReclainStorage



/*
This function allows you to display the values in any sequence.
*/

void PrintNodeValues(/* inout */ NodePointer& head);
// Pre:  "head" has been initialized.
// Post: If the sequence is empty, a message to that effect has
//       been output.  Otherwise, all values in the sequence
//       pointed to by "head" have been displayed, ten to a line,
//       with each one in a field width of six spaces.
// Calls: InitializeSequence, EmptySequence, ReclainStorage



/*
The following four functions are "utility functions" that are called by
other functions to do part of their work.

The first two find (respectively) the node containing a data value, and
that node plus the previous node.  The previous node is needed if we plan
to insert before a node or delete a node.
```

```
The next two find (respectively) the node having a given position, and
that node plus the previous node.  Once again, the previous node is needed
if we plan to insert before a node or delete a node.
*/

void FindNodeDataValue(/* inout */ NodePointer   head,
                       /* in */    DataType      targetValue,
                       /* out */   bool&         found,
                       /* out */   NodePointer&  targetPointer,
                       /* out */   int&          targetPosition);
// Pre:  "head" and "targetValue" have been initialized.
// Post: If "targetValue" is in the sequence, "found" is true and:
//       - "targetPointer" points to the first node of the sequence
//         containing the "targetValue"
//       - "targetPosition" contains the number of the first node
//         in the sequence containing the "targetValue"
//       Otherwise, "found" is false and both "targetPointer" and
//       "targetPosition" are undefined.
// Called by: FindDataValues, InsertAfterNodeWithDataValue

void FindDataValueAndPrevious(/* inout */ NodePointer&  head,
                              /* in */    DataType      targetValue,
                              /* out */   bool&         found,
                              /* out */   NodePointer&  targetPointer,
                              /* out */   NodePointer&  previousPointer);
// Pre:  "head" and "targetValue" have been initialized.
// Post: If "targetValue" is in the sequence, "found" is true and
//       "targetPointer" points at the first node of the sequence
//       containing "targetValue", with "previousPointer" pointing
//       at the previous node, unless "targetPointer" is pointing at
//       the first node, in which case the value of "previousPointer"
//       is NULL.  Otherwise, "found" is false and the values of both
//       "targetPointer" and "previousPointer" are undefined.
// Called by: InsertBeforeNodeWithDataValue, DeleteNodeWithDataValue


void FindNodeK(/* inout */ NodePointer&  head,
               /* in */    int           k,
               /* out */   bool&         found,
               /* out */   NodePointer&  targetPointer);
// Pre:  "head" and "k" have been initialized.
// Post: If the sequence contains k or more nodes, "found" is true
//       and "targetPointer" points to node k.  Otherwise, "found"
//       is false and "targetPointer" is undefined.
// Called by: FindPositions, InsertAfterNodeK

void FindNodeKAndPrevious(/* inout */ NodePointer&  head,
                          /* in */    int           k,
                          /* out */   bool&         found,
                          /* out */   NodePointer&  targetPointer,
                          /* out */   NodePointer&  previousPointer);
// Pre:  "head" and "k" have been initialized.
// Post: If the sequence has k or more nodes, "found" is true and
//       "targetPointer" points at node k of the sequence, with
//       "previousPointer" pointing at the previous node, unless
//       node k is the first node, in which case the value of
//       "previousPointer" is NULL.  Otherwise, "found" is false
//       and the values of both "targetPointer" and "previousPointer"
//       are undefined.
// Called by: InsertBeforeNodeK, DeleteNodeK
```

```
/*
The next two functions find a particular node, either via the data value
in it, or the node's position.
*/
void FindDataValues(/* inout */ NodePointer& head);
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more integer values and
//       in each case a message has been displayed, indicating
//       either the first position in the sequence where that
//       value was located or that the value was not found in
//       in the current sequence.
// Calls: FindNodeDataValue

void FindPositions(/* inout */ NodePointer& head);
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more integer position
//       numbers and in each case a message has been displayed,
//       indicating either the value found in that position or
//       that position was not found in the current sequence.
// Calls: FindNodeK


/*
The next two functions find a node (via data value or position) and
insert a new node after that node.
*/
void InsertAfterNodeWithDataValue(/* inout */ NodePointer& head);
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more data value pairs, and
//       in each case the second value in the pair has been inserted
//       into the sequence after the first occurrence of the first
//       value in the sequence, or a message has been displayed
//       indicating that the first value of the pair was not found
//       in the current sequence.
// Calls: FindNodeDataValue

void InsertAfterNodeK(/* inout */ NodePointer& head);
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more position/value pairs,
//       and in each case the value in the pair has been inserted
//       into the sequence after the given position, or a message
//       has been displayed indicating that the given position was
//       not found in the current sequence.
// Calls: FindNodeK


/*
The next two functions find a node (via data value or position) and
insert a new node before that node.
*/
void InsertBeforeNodeWithDataValue(/* inout */ NodePointer& head);
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more data value pairs, and
//       in each case the second value in the pair has been inserted
//       into the sequence before the first occurrence of the first
//       value in the sequence, or a message has been displayed
//       indicating that the first value of the pair was not found
//       in the current sequence.
// Calls: FindDataValueAndPrevious
```

```
void InsertBeforeNodeK(/* inout */ NodePointer& head);
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more position/value pairs,
//       and in each case the value in the pair has been inserted
//       into the sequence before the given position, or a message
//       has been displayed indicating that the given position was
//       not found in the current sequence.
// Calls: FindeNodeKAndPrevious



/*
The next two functions find a node (via data value or position) and
then delete that node.
*/
void DeleteNodeWithDataValue(/* inout */ NodePointer& head);
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more data values and in
//       each case the first node of the sequence containing
//       that value has been deleted from the sequence and its
//       storage has been returned to the free store, or a
//       message has been displayed indicating that the value
//       was not found in the current sequence.
// Calls: FindDataValueAndPrevious

void DeleteNodeK(/* inout */ NodePointer& head);
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more node positions and
//       in each case the node of the sequence having the given
//       position has been deleted from the sequence and its
//       storage has been returned to the free store, or a
//       message has been displayed indicating that the given
//       position was not found in the current sequence.
// Calls: FindeNodeKAndPrevious



/*
The next three functions each represent something we might want to do
to the integer data in one node.  To do that particular thing to the
data in all nodes of a sequence we call the last functions and pass the
appropriate one of the other three as a "function parameter".
*/
void Double(/* inout */ NodePointer& p);
// Pre:  "p" has been initialized.
// Post: The value in the node pointed to by "p" has been doubled.

void Square(/* inout */ NodePointer& p);
// Pre:  "p" has been initialized.
// Post: The value in the node pointed to by "p" has been squared.

void ReverseAllDigits(/* inout */ NodePointer& p);
// Pre:  "p" has been initialized.
// Post: The value in the node pointed to by "p" has had its
//       digits reversed.

void ProcessAllNodes(/* inout */    NodePointer& head,
                     /* function */ void (*Process)(NodePointer& p));
// Pre:  "head" and "Process" have been initialized.
// Post: The value in each node in the sequence pointed to by "head"
//       has been "processed" by the function pointed to by "Process".
```

21.4.3.1 What you see for the first time in LNODEFUN.H

This is a large file of function prototypes but the only new thing in it is the very last function prototype, and here it is again:

```
void ProcessAllNodes(/* inout */    NodePointer& head,
                     /* function */ void (*Process)(NodePointer& p));
```

The second parameter in this prototype is the *function parameter*

```
void (*Process)(NodePointer& p)
```

which is unlike anything we've seen before. This is the formal parameter that says we are going to pass as an actual parameter the name of a function. This is a very convenient thing to be able to do if we want a given function to call *different other functions at different times.*

We may not have mentioned this at any time previously since it wasn't relevant till now, but the name of *any* function is really just a pointer to the memory location where the code for that function begins. To make that statement a little more concrete let's observe that if you wanted to output the square root of 6 (for example), you would normally write a statement like

```
cout << sqrt(6) << endl;
```

but our remark above means that you could also write

```
cout << (*sqrt)(6) << endl;
```

in which (by the way) precedence rules make the parentheses in `(*sqrt)` necessary. So now the (*Process) part of the formal function parameter should make sense. In fact, the generic form of a function parameter might be written like this:

```
ReturnType (*FunctionName)(parameter_list)
```

21.4.3.2 Additional notes and discussion on LNODEFUN.H

Note that to see how the function `ProcessAllNodes` is used you need to look back at the driver in LNODEDRV.CPP, where you will see that indeed the actual parameter is, in each case, the name of a function. Not only that, but each function passed as an actual parameter is "the same kind of function" (i.e., they all have the same kind of parameter list, in this case just`NodePointer&`).

21.4.3.3 Follow-up hands-on activities for LNODEFUN.H

□ Copy LNODEFUN.H and study the code, particularly the format of the function parameter discussed above. Compare this header file with the driver in LNODEFUN.CPP and note which of the functions are actually called by the driver and which are not. These latter functions are therefore, by implication, "helper functions" called and used by the remaining functions in the file.

◯ Instructor checkpoint 21.3

21.4.4 LNODEFUN.SHL contains source code for many but not all functions in LNODEFUN.OBJ

```
// Filename: LNODEFUN.SHL
// Purpose:  Shell file of functions to be used with the
//           demonstration program in LNODEDRV.CPP.  Some
//           functions are missing and will need to be
//           supplied by the student.

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

typedef int DataType;
struct Node
{
    DataType data;
    Node* link;
};
typedef Node* NodePointer;


void DescribeProgram()
// Pre:  none
// Post: The program description has been displayed.
{
    cout << endl;
    cout << "This is a demonstration program used "
         << "to illustrate some operations on a "        << endl;
    cout << "sequence of linked nodes containing "
         << "integer data values.  It is best to "       << endl;
    cout << "run it with pencil and paper in hand "
         << "and to draw a pictorial representation "    << endl;
    cout << "of what the sequence of linked nodes "
         << "will look like after you take any "         << endl;
    cout << "particular action offered by the menu. "    << endl;
    cout << endl;
    cout << "You may display the data values in "
         << "the nodes at any time for purposes of "     << endl;
    cout << "comparison.  Each value is printed in six "
         << "spaces, with ten values per line. "         << endl;
    cout << endl;
    cout << "You begin with an empty list. "             << endl;
    cout << endl;
}

void DisplayMenu()
// Pre:  none
// Post: A menu with 16 options has been displayed.
{
    cout << endl;
    cout << endl;
    cout << "        Menu "                                    << endl;
    cout << endl;
    cout << " 1. Quit "                                        << endl;
    cout << " 2. Re-initialize sequence "
         <<      "(create a new empty sequence) "              << endl;
    cout << " 3. Display all node data values "
         <<      "in current sequence "                        << endl;
    cout << " 4. Build a new sequence of fixed size "          << endl;
    cout << " 5. Build a new sequence of any size "            << endl;
```

```
    cout << " 6. Find the location of first node containing "
         <<      "a specified data value "                    << endl;
    cout << " 7. Find data value in kth node "                << endl;
    cout << " 8. Add a new node after first node containing "
         <<      "specified data value "                      << endl;
    cout << " 9. Add a new node after kth node "              << endl;
    cout << "10. Add a new node before first node containing "
         <<      "specified data value "                      << endl;
    cout << "11. Add a new node before kth node "             << endl;
    cout << "12. Delete first node containing a "
         <<      "specified data value "                      << endl;
    cout << "13. Delete kth node "                            << endl;
    cout << endl;
    cout << "... Some things to do with "
         <<      "the node data values ... "                  << endl;
    cout << "14. Double all node data values "                << endl;
    cout << "15. Square all node data values "                << endl;
    cout << "16. Reverse the digits of each node data value " << endl;
    cout << endl;
}

void GetMenuChoice(/* out */ int& menuChoice)
// Pre:  none
// Post: "menuChoice" contains an integer entered by the user
//       from the keyboard
{
    cout << "Enter menu choice here: ";
    cin >> menuChoice;  cin.ignore(80, '\n');
    cout << endl;
}


void InitializeSequence(/* inout */ NodePointer& head)
// Pre:  none
// Post: "head" has been set to NULL.
{
    head = NULL;
}

bool EmptySequence(/* in */ NodePointer head)
// Pre:  "head" has been initialized.
// Post: The function value is true if the sequence is empty,
//       and false otherwise.
{
    return (head == NULL);
}

void ReclaimStorage(/* in */ NodePointer head)
// Pre:  "head" has been initialized.
// Post: All storage (if any) in the node sequence pointed to by
//       "head" has been returned to the free store.
{
    NodePointer current;

    while (head != NULL)
    {
        current = head;
        head = head->link;
        delete current;
    }
}
```

```
void BuildSequenceFixedSize(/* inout */ NodePointer& head)
// Pre:  none
// Post: The user has entered the number of nodes to be in a
//       newly constructed sequence, as well as the values to
//       be inserted into the sequence, the sequence has been
//       constructed, and "head" points at the first node.
//       All storage in any sequence previously pointed to by
//       "head" has been returned to the free store.
{
    if (!EmptySequence(head))
    {
        ReclaimStorage(head);
        InitializeSequence(head);
    }

    int numNodes;
    int nodeCounter;
    NodePointer current;
    NodePointer next;

    cout << "Enter number of nodes in the sequence: ";
    cin  >> numNodes;  cin.ignore(80, '\n');
    cout << endl;
    cout << "Now enter the node values for the sequence "
         << "on the following line: " << endl;

    head = new Node;
    cin >> head->data;
    current = head;

    for (nodeCounter = 2; nodeCounter <= numNodes; nodeCounter++)
    {
        next = new Node;
        cin >> next->data;
        current->link = next;
        current = next;
    }
    cin.ignore(80, '\n');
    current->link = NULL;
}

void BuildSequenceAnySize(/* inout */ NodePointer& head)
// Pre:  none
// Post: The user has entered the values to be inserted into
//       a newly constructed sequence (terminating with an
//       end-of-file), the sequence has been constructed, and
//       "head" points at the first node.  All storage in any
//       sequence previously pointed to by "head" has been
//       returned to the free store, and the standard input
//       stream has been readied for further input.
{
    // Function body goes here
}

void PrintNodeValues(/* inout */ NodePointer& head)
// Pre:  "head" has been initialized.
// Post: If the sequence is empty, a message to that effect has
//       been output.  Otherwise, all values in the sequence
//       pointed to by "head" have been displayed, ten to a line,
//       with each one in a field width of six spaces.
```

```
{
    NodePointer current;
    int nodeCounter;

    if (head == NULL)
        cout << "No values to print.  Current sequence is empty. "
             << endl;
    else
    {
        cout << "Here are the current data values in the sequence: "
             << endl;
        current = head;
        nodeCounter = 0;
        while (current != NULL)
        {
            cout << setw(6) << current->data;
            nodeCounter++;
            if (nodeCounter % 10 == 0) cout << endl;
            current = current->link;
        }
        if (nodeCounter % 10 != 0) cout << endl;
    }
    cout << endl;
}


void FindNodeDataValue(/* inout */ NodePointer   head,
                       /* in */    DataType      targetValue,
                       /* out */   bool&         found,
                       /* out */   NodePointer&  targetPointer,
                       /* out */   int&          targetPosition)
// Pre:  "head" and "targetValue" have been initialized.
// Post: If "targetValue" is in the sequence, "found" is true and:
//       - "targetPointer" points to the first node of the sequence
//         containing the "targetValue"
//       - "targetPosition" contains the number of the first node
//         in the sequence containing the "targetValue"
//       Otherwise, "found" is false and both "targetPointer" and
//       "targetPosition" are undefined.
{
    NodePointer current = head;
    int nodeCounter = 1;

    found = false;

    while (!found  &&  current != NULL)
    {
        if (current->data == targetValue)
            found = true;
        else
        {
            current = current->link;
            nodeCounter++;
        }
    }

    if (found)
    {
        targetPointer  = current;
        targetPosition = nodeCounter;
    }
}
```

```
void FindDataValueAndPrevious(/* inout */ NodePointer&  head,
                              /* in */    DataType      targetValue,
                              /* out */   bool&         found,
                              /* out */   NodePointer&  targetPointer,
                              /* out */   NodePointer&  previousPointer)
// Pre:  "head" and "targetValue" have been initialized.
// Post: If "targetValue" is in the sequence, "found" is true and
//       "targetPointer" points at the first node of the sequence
//       containing "targetValue", with "previousPointer" pointing
//       at the previous node, unless "targetPointer" is pointing at
//       the first node, in which case the value of "previousPointer"
//       is NULL.  Otherwise, "found" is false and the values of both
//       "targetPointer" and "previousPointer" are undefined.
{
    NodePointer current = head;
    NodePointer previous = NULL;

    found = false;

    while (!found  &&  current != NULL)
    {
        if (current->data == targetValue)
            found = true;
        else
        {
            previous = current;
            current = current->link;
        }
    }

    if (found)
    {
        targetPointer   = current;
        previousPointer = previous;
    }
}


void FindNodeK(/* inout */ NodePointer&  head,
               /* in */    int           k,
               /* out */   bool&         found,
               /* out */   NodePointer&  targetPointer)
// Pre:  "head" and "k" have been initialized.
// Post: If the sequence contains k or more nodes, "found" is true
//       and "targetPointer" points to node k.  Otherwise, "found"
//       is false and "targetPointer" is undefined.
{
    // Function body goes here
}


void FindNodeKAndPrevious(/* inout */ NodePointer&  head,
                          /* in */    int           k,
                          /* out */   bool&         found,
                          /* out */   NodePointer&  targetPointer,
                          /* out */   NodePointer&  previousPointer)
// Pre:  "head" and "k" have been initialized.
// Post: If the sequence has k or more nodes, "found" is true and
//       "targetPointer" points at node k of the sequence, with
//       "previousPointer" pointing at the previous node, unless
```

```
//       node k is the first node, in which case the value of
//       "previousPointer" is NULL.  Otherwise, "found" is false
//       and the values of both "targetPointer" and "previousPointer"
//       are undefined.
{
    // Function body goes here
}


void FindDataValues(/* inout */ NodePointer& head)
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more integer values and
//       in each case a message has been displayed, indicating
//       either the first position in the sequence where that
//       value was located or that the value was not found in
//       in the current sequence.
{
    DataType targetValue;
    NodePointer targetPointer;
    int targetPosition;
    bool found;

    cout << "Enter the data value to look for, "
         << "or end-of-file to quit: ";
    cin >> targetValue;  cin.ignore(80, '\n');

    while (cin)
    {
        cout << endl;

        FindNodeDataValue(head, targetValue, found,
                          targetPointer, targetPosition);

        cout << targetValue << " was ";
        if (found)
            cout << "found in node " << targetPosition << ".";
        else
            cout << "not found in the current sequence.";
        cout << endl;

        cout << "Enter another data value to look for, "
             << "or end-of-file to quit: ";
        cin >> targetValue;  cin.ignore(80, '\n');
    }
    cout << endl << endl;

    cin.clear();
}



void FindPositions(/* inout */ NodePointer& head)
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more integer position
//       numbers and in each case a message has been displayed,
//       indicating either the value found in that position or
//       that position was not found in the current sequence.
{
    // Function body goes here
}
```

```
void InsertAfterNodeWithDataValue(/* inout */ NodePointer& head)
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more data value pairs, and
//       in each case the second value in the pair has been inserted
//       into the sequence after the first occurrence of the first
//       value in the sequence, or a message has been displayed
//       indicating that the first value of the pair was not found
//       in the current sequence.
{
    DataType targetValue;
    NodePointer newNode;
    NodePointer targetPointer;
    int targetPosition;
    bool found;

    cout << "\nEnter data value after which to insert, "
         << "or end-of-file to quit: ";
    cin >> targetValue;  cin.ignore(80, '\n');

    while (cin)
    {
        cout << endl;

        FindNodeDataValue(head, targetValue, found,
                          targetPointer, targetPosition);
        if (found)
        {
            newNode = new Node;
            cout << "Enter the data value for the new node: ";
            cin >> newNode->data;  cin.ignore(80, '\n');
            cout << endl;
            newNode->link = targetPointer->link;
            targetPointer->link = newNode;
        }
        else
        {
            cout << endl;
            cout << "There is no node containing the value "
                 << targetValue << "."                      << endl;
        }

        cout << endl;
        cout << "Enter another data value after which to insert, "
             << "or end-of-file to quit: ";
        cin >> targetValue;  cin.ignore(80, '\n');
    }
    cout << endl << endl;

    cin.clear();
}


void InsertAfterNodeK(/* inout */ NodePointer& head)
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more position/value pairs,
//       and in each case the value in the pair has been inserted
//       into the sequence after the given position, or a message
//       has been displayed indicating that the given position was
//       not found in the current sequence.
{
    // Function body goes here
}
```

```
void InsertBeforeNodeWithDataValue(/* inout */ NodePointer& head)
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more data value pairs, and
//       in each case the second value in the pair has been inserted
//       into the sequence before the first occurrence of the first
//       value in the sequence, or a message has been displayed
//       indicating that the first value of the pair was not found
//       in the current sequence.
{
    DataType targetValue;
    NodePointer newNode;
    NodePointer targetPointer;
    NodePointer previousPointer;
    bool found;

    cout << endl;
    cout << "Enter the data value before which to insert, "
         << "or end-of-file to quit: ";
    cin >> targetValue;  cin.ignore(80, '\n');

    while (cin)
    {
        cout << endl;

        FindDataValueAndPrevious(head, targetValue, found,
                                 targetPointer, previousPointer);

        if (found)
        {
            newNode = new Node;
            cout << "Enter the data value for the new node: ";
            cin >> newNode->data;  cin.ignore(80, '\n');
            cout << endl;
            newNode->link = targetPointer;
            if (targetPointer == head)
                head = newNode;
            else
                previousPointer->link = newNode;
        }
        else
        {
            cout << endl;
            cout << "There is no node containing the value "
                 << targetValue << ".";
        }

        cout << endl;
        cout << "Enter another data value, before which to insert, "
             << "or end-of-file to quit: ";
        cin >> targetValue;  cin.ignore(80, '\n');
    }
    cout << endl << endl;

    cin.clear();
}
```

```
void InsertBeforeNodeK(/* inout */ NodePointer& head)
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more position/value pairs,
//       and in each case the value in the pair has been inserted
//       into the sequence before the given position, or a message
//       has been displayed indicating that the given position was
//       not found in the current sequence.
{
    // Function body goes here
}


void DeleteNodeWithDataValue(/* inout */ NodePointer& head)
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more data values and in
//       each case the first node of the sequence containing
//       that value has been deleted from the sequence and its
//       storage has been returned to the free store, or a
//       message has been displayed indicating that the value
//       was not found in the current sequence.
{
    DataType targetValue;
    NodePointer targetPointer;
    NodePointer previousPointer;
    bool found;

    cout << endl;
    cout << "Enter data value to delete, or "
            "end-of-file for no deletions: ";
    cin >> targetValue;  cin.ignore(80, '\n');

    while (cin)
    {
        cout << endl;

        FindDataValueAndPrevious(head, targetValue, found,
                                 targetPointer, previousPointer);

        if (found)
        {
            if (targetPointer == head)
                head = targetPointer->link;
            else
                previousPointer->link = targetPointer->link;
            delete targetPointer;
        }
        else
        {
            cout << endl;
            cout << "There is no node containing the value "
                 << targetValue << ".";
        }

        cout << endl;
        cout << "Enter another data value to delete, "
             << "or end-of-file to end deletions: ";
        cin >> targetValue;  cin.ignore(80, '\n');
    }

    cin.clear();
}
```

```
void DeleteNodeK(/* inout */ NodePointer& head)
// Pre:  "head" has been initialized.
// Post: The user has entered zero or more node positions and
//       in each case the node of the sequence having the given
//       position has been deleted from the sequence and its
//       storage has been returned to the free store, or a
//       message has been displayed indicating that the given
//       position was not found in the current sequence.
{
    // Function body goes here
}


void Double(/* inout */ NodePointer& p)
// Pre:  "p" has been initialized.
// Post: The value in the node pointed to by "p" has been doubled.
{
    p->data = p->data * 2;
}

void Square(/* inout */ NodePointer& p)
// Pre:  "p" has been initialized.
// Post: The value in the node pointed to by "p" has been squared.
{
    p->data = p->data * p->data;
}

void ReverseAllDigits(/* inout */ NodePointer& p)
// Pre:  "p" has been initialized.
// Post: The value in the node pointed to by "p" has had its
//       digits reversed.
{
    int n, nr;
    if (p->data > 9)
    {
        n = p->data;
        nr = 0;
        while (n != 0)
        {
            nr = 10*nr + (n%10);
            n = n/10;
        }
        p->data = nr;
    }
}


void ProcessAllNodes(/* inout */    NodePointer& head,
                     /* function */ void (*Process)(NodePointer& p))
// Pre:  "head" and "Process" have been initialized.
// Post: The value in each node in the sequence pointed to by "head"
//       has been "processed" by the function pointed to by "Process".
{
    NodePointer current;

    if (head == NULL)
    {
        cout << endl;
        cout << "Cannot process nodes in an empty list. ";
        cout << endl;
    }
```

```
    else
    {
        current = head;
        while (current != NULL)
        {
            (*Process)(current);
            current = current->link;
        }
    }
}
```

21.4.4.1 What you see for the first time in LNODEFUN.SHL

In this file you encounter the most extreme example yet of many familiar things being used in new and unfamiliar ways. But for something new and different you need to look again at the function `ProcessAllNodes` to see how the function parameter is used in the body of the function to which it has been passed. Note in particular how `*Process` (the dereferenced pointer that acts like the name of a "placeholder function") is used as the name of the function being called within the body of `ProcessAllNodes`.

21.4.4.2 Additional notes and discussion on LNODEFUN.SHL

Note that this file contains most of the source code corresponding to the object file LNODEFUN.OBJ, but what is missing is the group of functions that use the position of a node in the sequence as opposed to the data value in the node to locate the node for processing. For the most part this code is analogous to the code that *is* present so you should study the functions which are complete for broad hints on how to complete the missing ones.

21.4.4.3 Follow-up hands-on activities for LNODEFUN.SHL

□ Copy LNODEFUN.SHL and study the code. Replace each of the missing function bodies with an output statement that reports the message "Not yet implemented" when the function is called, then compile and link with the driver LNODEDRV.OBJ. Run the program to test all functions, including the ones not yet implemented.

◯ Instructor checkpoint 21.4

□ Implement each of the missing functions, one at a time, re-compiling, re-linking and re-testing each one as you go. Note how the already-present driver does in fact serve as a driver for all that you have to do. Having such a driver in any development situation is always worth the extra time it takes to prepare.

◯ Instructor checkpoint 21.5

Module 22

Miscellaneous topics

22.1 Objectives

- To understand how to generate *random numbers* —both integer and real— and also how to generate random characters and strings.
- To understand how to issue operating system commands from a C++ program. (Such commands are also referred to as *system calls*).
- To learn some additional routines for manipulating strings, both C-strings and C++ string objects.
- To understand what a *string stream* is, and know how to use one.
- To understand how to find the *greatest common divisor* of two positive integers.

22.2 List of associated files

- RAND.CPP illustrates random number generation.
- SYSCALLS.CPP show a how to make system calls from within a C++ program.
- STRMISC.CPP illustrates some miscellaneous string functions for both C-strings and C++ string objects.
- SSTREAM.CPP uses string streams to analyze contents of command-line parameters.
- GCD_ITER.CPP shows how to find the greatest common divisor of two positive integers.

22.3 Overview

This Module provides several sample programs, each of which illustrates some topic of a miscellaneous nature that had no natural home amidst the general flow of our discussion of C++ features, or extends one or more topics covered in an earlier Module.

The program in RAND.CPP shows how to generate "random" (or, more correctly) *pseudorandom* numbers (or other pseudorandom quantities). The kinds of random quantities that can easily be generated using the techniques illustrated by this program include integers, real numbers, single characters and strings.

As you know, when one of your C++ programs starts to run, everything it does is accomplished by statements that you wrote and placed in the program, or by functions your program calls, again either ones you wrote or ones available from various libraries. Sometimes certain tasks might more easily be done by a direct call to the operating system, or sometimes we might need some information that only the operating system can give us, so it would be nice if our C++ programs could communicate directly with the operating system. This can be arranged, and this program shows one simple way to do it, illustrating how to send a command (or *system call*) to the operating system from a C++ program.

STRMISC.CPP shows you a number of additional functions that may be useful for manipulating C-strings and/or C++ string objects. Do no think that even this completes your knowledge of what is available for working with strings. You need a C++ Standard Library reference work for the complete story.

The topic of *string streams* is not an everyday one in C++, but occasionally it can be just what the doctor ordered, and in certain situations life would be much more difficult without this feature, which is illustrated in SSTREAM.CPP. In a nutshell, a *string stream* allows you to treat text stored in a string in memory just like an input or output stream, in the sense that you can read from or write to such a string stream just as though it was a stream (a file, say). Why would you want to do such a thing? Well, the sample program shows how, for example, you can read in command-line parameters (C-strings), convert them to string streams, and then read values of the various data types from the string stream just as if they were being read from the keyboard. Without this facility, extracting the various data types from a C-string command-line parameter would be a real pain.

Finally, GCD_ITER.CPP illustrates a function for computing the *greatest common divisor* of two positive integers, something that comes in handy every once in a while, possibly often enough that you might consider placing a copy of the function in readily accessible form wherever you are keeping your "utilities".

22.4 Sample Programs

22.4.1 RAND.CPP illustrates random number generation

```
// Filename: RAND.CPP
// Purpose:  Illustrates the system random number generator.

#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

#include "PAUSE.H"

int main()
{
    cout << "\nThis program illustrates three system "
         << "\"utility\" functions that are available "  << endl
         << "from the standard libaries: clock(), "
         << "rand() and srand() "                        << endl << endl;
    Pause(0);

    cout << "\nThe clock() function from the <ctime> "
         << "library returns the number of CPU "     << endl
         << "\"clock ticks\" since your process "
         << "started running. "                      << endl;
    cout << "Current value of clock(): " << clock()  << endl;
    Pause(0);

    cout << "\nRAND_MAX is a named constant from <cstdlib>." << endl;
    cout << "It is the largest possible return value "
         << "from a call to the rand() function. "           << endl;
    cout << "Value of RAND_MAX on this system: " << RAND_MAX << endl;
    Pause(0);

    // Run this program three times.  Then un-comment the function call given
    // below and run it three more times.  Be sure you understand that (even
    // if not how) the function "resets" the system "random number generator".
    // srand(clock());

    int i;
    cout << "\nHere are 10 \"random\" integer values "
         << "in the range 0.." << RAND_MAX << ": "    << endl;
    for (i = 1; i <= 10; i++)
        cout << rand() << endl;
    Pause(0);

    int first, second;
    cout << "\nEnter two positive integers, with "
         << "the second larger than the first: ";
    cin >> first >> second;  cin.ignore(80, '\n');  cout << endl;
    cout << "Here are 10 random integer values between "
         << first << " and " << second << ": "           << endl;
    for (i = 1; i <= 10; i++)
        cout << first + rand() % (second - first + 1)    << endl;
    cout << endl;

    return 0;
}
```

22.4.1.1 What you see for the first time in RAND.CPP

- `rand` generates a *pseudorandom* integer in the range `0...MAX_RAND`, where `MAX_RAND` is a built-in named constant
- `srand` supplies a *seed value* to be used as the *random number generation* starting value for number generation by the `rand` function
- `clock` returns the integer giving the number of "clock ticks" since your process started running

22.4.1.2 Additional notes and discussion on RAND.CPP

The "random" numbers generated by a computer are not "truly random", in the same sense that you would get "truly random" numbers from 1 to 6 by throwing a single die and counting the spots on the upturned face. But the numbers generated by a computer programs random number generator are generally "random enough" to satisfy the usual statistical tests for randomness that we might want to apply, which is usually good enough for our purposes, which might be for a game program or a *simulation* (a program representing a real world process involving chance events). In this context, however, we do tend to refer to *random numbers* even though what we really mean is *pseudorandom numbers*.

The rand function generates a random number between `0` and `MAX_RAND`, but often we want either an integer in some other range or even some other type of value such as a real number, a character, or a string. Thus we usually need to apply transformation like one of those shown below to the integer generated by `rand` to get the value we need.

If `r` is in `0..MAX_RAND` then the expression

```
a + r % (b - a + 1)  // gives a number in a..b
r/double(MAX_RAND+1) // gives a real number x such that 0 <= x < 1
char(65 + r % 26)    // gives a single capital letter
```

22.4.1.3 Follow-up hands-on activities for RAND.CPP

☐ Copy, study and test the program in RAND.CPP.

☐ Add code to RAND.CPP that outputs 60 random lower-case letters.

◯ Instructor checkpoint 22.1

☐ Add more code to RAND.CPP that outputs 20 random real number values between -1.5 and 1.5.

☐ Add still more code to RAND.CPP that outputs 15 random string values, each containing capital letters.

◯ Instructor checkpoint 22.2

22.4.2 SYSCALLS.CPP illustrates how to make "system calls" from within a C++ program

```
// Filenme: SYSCALLS.CPP
// Purpose: Illustrates how to make "system calls" from a C++ program.

#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

#include "PAUSE.H"


int main()
{
    cout << endl;
    cout << "This program makes some \"system calls\".\n";
    cout << "That is, it issues several commands to the operating system.\n";
    cout << "Once again you should study the code and the ouput together.\n";
    Pause(0);

    // The program makes a "system call" to see if there are
    // any files in the current directory with a .tmp extension:
    system("dir *.tmp");
    Pause(0);  cout << endl;

    // It then creates a file called "ttt.tmp" and writes out
    // to that file the first three positive integers:
    ofstream outFile("ttt.tmp");
    outFile << 1 << endl << 2 << endl << 3 << endl;
    outFile.close();

    // Then it checks again for files with a .tmp extension:
    system("dir *.tmp");
    Pause(0);  cout << endl;

    // Finally it makes another "system call" to display the file:
    system("type ttt.tmp");
    Pause(0);  cout << endl;

    return 0;
}
```

22.4.2.1 What you see for the first time in SYSCALLS.CPP

This program illustrates *system calls* like

```
system("dir");
```

that use the `system` function from the `cstdlib` library to send commands like the `dir` command that is available on some systems for listing all the files in the current directory. However, any string that represents a valid command for the local operating system could act as the parameter in the call to `system`.

22.4.2.2 Additional notes and discussion on SYSCALLS.CPP

Note that this program is system-dependent, in that the parameters shown will not work on all systems. (They work in DOS and VMS, for example, but not under Unix or Linux, in which the command for listing files is `ls` and the command for displaying a file is, wonder of wonders, `cat`.)

22.4.2.3 Follow-up hands-on activities for SYSCALLS.CPP

□ Copy, study and test the program in SYSCALLS.CPP. If the operating system commands given in the program do not work on your system, replace them with the equivalent commands on your system.

□ Add new code to SYSCALLS.CPP that causes your program to send at least one other system call to your operating system. Test the revised program until the new system call works properly when issued from your program.

◯ Instructor checkpoint 22.3

22.4.3 STRMISC.CPP illustrates additional functions for both C-strings and C++ string objects

```
// Filename: STRMISC.CPP
// Purpose:  To show some miscellaneous string operations.
// Updated:  11-MAR-2000 19:20:05 to add the "find", "insert",
//           and "replace" operations.


#include <iostream>
#include <string>
#include <cstring>
#include <cstdlib>
using namespace std;

#include "pause.h"


int main()
{
    cout << "\nThis program illustrates a number of "
         << "additional functions for working with \n"
         << "C-strings, as well as a number of "
         << "additional operations available in the \n"
         << "C++ string class interface.  Study "
         << "the source code while running the program. \n"
         << "Some of the comments also suggest "
         << "modifications that you should make, and \n"
         << "then try the program again. \n\n";
    Pause(0);



    // First some additional functions for working with C-strings:

    // ****************************************************************
    // There are two functions from stdlib.h that are very helpful when
    // you want to convert a string which "looks like" a number to an
    // actual number.  Here they are:
    // atoi(s) returns the integer form of the string s.
    // atof(s) returns the floating point (double) form of the string s.
    char c_s1[] = "37";
    char c_s2[] = "5.1";
    // cout << c_s1 * c_s2 << endl;
    // The above line would (of course) cause a syntax error (try it!),
    // but the following line works fine:
    cout << atoi(c_s1) * atof(c_s2) << endl << endl;
    Pause(0);


    // ****************************************************************
    // Now some additional properties of C++ string objects and some
    // additional operations in the C++ string class interface:
    string cpp_s, cpp_s1, cpp_s2;


    // ****************************************************************
    // Often we have to be careful to distinguish between a single
    // character and a string containing a single character, and in
    // particular it is worth remembering the following about C++
    // strings:
```

```
cpp_s = 'A';  // This assignment is OK, but neither of the following
              // initializations in a declaration is OK (try them!):
              // string s = 'A';
              // string s('A');
cout << cpp_s << endl << endl;
Pause(0);


// ****************************************************************
// Assigning a C-string to a C++ string object is OK, but (as we
// know) we cannot assign anything to a C-string variable, including
// a C++ string object.
char c_s[] = "This is a C-string.";
cpp_s = c_s;
cout << cpp_s << endl << endl;
Pause(0);


// ****************************************************************
// There are two ways to access individual characters in a C++
// string, one of them "safer" than the other.  We can use square
// brackets just as we do for C-strings, as in
// s[i] which accesses the character at index "i" in s,
// or we can use the "at" operation, as in
// s.at(i) which also accesses the character at index "i" in s.
// The difference is that there is no out-of-bounds error checking
// when using square brackets (to be consistent with the normal use
// of square brackets for array component access), but there *is*
// out-of-bounds error checking when using "at".
cpp_s1 = "man";
cpp_s2 = "man";
cout << cpp_s1 << "  " << cpp_s2 << endl;
cpp_s1[2] = cpp_s2.at(0) = 't';
cout << cpp_s1 << "  " << cpp_s2 << endl;
// The following two lines of code "work" because there is no error
// checking when brackets are used to access a character location
// that does not "belong to " the string.
cpp_s1[3] = 'e';
cout << cpp_s1 << endl << endl;
// But, the following two lines of code do *not* work because there
// *is* error checking when the "at" operation is used.  The error
// occurs at run-time since that is when the attempted access takes
// place.  Try running the program with these two lines uncommented.
// cpp_s2.at(3) = 'g';
// cout << cpp_s2 << endl;
Pause(0);


// ****************************************************************
// We can always "extract" the C-string equivalent from a C++ string
// object with the c_str() operation as follows:
// s.c_str() returns a \0 terminated array of characters containing
//     the characters in the string s.
// Sometimes this is in fact necessary, as for example when we are
// passing a string parameter to the open() function of the fstream
// class (as you know), or to the system() function from stdlib.h.
// Simply displayed on the standard output, however, the two forms
// are (of course) indistinguishable:
cpp_s = "Hello, world!";
cout << cpp_s << "  " << cpp_s.c_str() << endl << endl;
Pause(0);
```

```
// ****************************************************************
// There is a "built-in" swap operation for C++ strings:
cpp_s1 = "Hello!";
cpp_s2 = "Good-bye!";
cout << cpp_s1 << "  " << cpp_s2 << endl;
cpp_s1.swap(cpp_s2);
cout << cpp_s1 << "  " << cpp_s2 << endl << endl;
Pause(0);


// ****************************************************************
// There are a number of operations which provide "stats" on various
// aspects of a string object:
// s.size() returns the number of characters currently in s.
// s.length() returns exactly the same thing as s.size().
// s.max_size() returns the largest possible number of characters
//     that a string object can hold.
// s.empty() returns true or false, depending on whether s currently
//     contains any characters.
// And if we'd like to *make* it empty, we can with this operation:
// s.clear()
cpp_s = "Hello, world!";
cout << cpp_s.size()     << "  "
     << cpp_s.length()   << "  "
     << cpp_s.max_size() << endl;
cout << "Is the string empty? "
     << (cpp_s.empty() ? "Yes!" : "No!") << endl;
//cpp_s.clear();  // Not available in Visual C++
cout << "Is the string empty? "
     << (cpp_s.empty() ? "Yes!" : "No!") << endl;
// The public, static constant "npos" is a member of the string
// class and is often the value returned by various operations on
// C++ string objects, depending on the outcome of the operation.
cout << "Value of \"npos\" is: " << string::npos << endl;
cout << "Value of \"npos\" cast to an int value is: "
     << (int)string::npos << endl << endl;
// So note that the value of npos is *not* an int to begin with.
// This is a potential problem, and a problem with the Standard.
Pause(0);


// ****************************************************************
// There is also an "erase" operation that provides a little more
// flexibility than the "clear" operation mentioned above:
// s.erase() works exactly like s.clear().
// s.erase(i) erases all characters from index "i" onward.
// s.erase(i, num) erases "num" characters starting at index "i".
cpp_s = "Hello, world!";
cpp_s.erase(5);
cout << cpp_s << endl;
cpp_s = "Hello, world!";
cpp_s.erase(5, 7);
cout << cpp_s << endl << endl;
Pause(0);


// ****************************************************************
// Another handy operation is that for extracting a substring from
// a given string, which looks like this:
// s.substr(startIndex, numChars) returns the substring of s
//     starting at "startIndex" and containing "numChars" characters.
```

```
    cpp_s = "Hello, world!";
    cout << cpp_s.substr(7, 5)
         << cpp_s.substr(5, 2)
         << cpp_s.substr(0, 5)
         << cpp_s.substr(12, 1) << endl << endl;
    Pause(0);


    // ****************************************************************
    // There are some "find" operations that allow us to search through
    // a string to find one or more characters:
    // s1.find(s2)
    // s1.rfind(s2)
    // s1.find_first_of(s2)
    // s1.find_last_of(s2)
    // s1.find_first_not_of(s2)
    // s1.find_last_not_of(s2)
    cpp_s = "Now is the time to get serious and get the lead out.";
    cout << cpp_s.find("the")  << endl;
    cout << cpp_s.rfind("the") << endl;
    cout << cpp_s.find_first_of("the") << endl;
    cout << cpp_s.find_last_of("the")  << endl;
    cout << cpp_s.find_first_not_of("Now the out.") << endl;
    cout << cpp_s.find_last_not_of("Now the out.")  << endl << endl;
    Pause(0);


    // ****************************************************************
    // There are some "insert" operations that allow us to insert one or
    // more characters into a string:
    // s1.insert(beforeIndex, s2);
    // s1.insert(beforeIndex, s2, startIndex, numChars);
    cpp_s = "Now is to get serious get the lead out.";
    cpp_s.insert(7, "the time ");
    cout << cpp_s << endl;
    cpp_s.insert(31, "oh my, and woe is me", 7, 4);
    cout << cpp_s << endl << endl;
    Pause(0);


    // ****************************************************************
    // There are some "replace" operations that allow us to replace one
    // or more characters in a string:
    // s1.replace(startIndex, numChars, s2);
    // s1.replace(startIndex, numChars1, s2, startIndex2, numChars2);
    cpp_s = "Now is the time to get serious and get the lead out.";
    cpp_s.replace(7, 3, "that");
    cout << cpp_s << endl;
    cpp_s.replace(24, 7, "smartypants" , 0, 5);
    cout << cpp_s << endl << endl;
    Pause(0);

    return 0;
}
```

22.4.3.1 Notes and discussion on STRMISC.CPP

This program illustrates a number of additional functions for manipulating strings—a few for C-strings and more for C++ string objects. The many comments sprinkled throughout the program give you the necessary information about the new string features, and you should read these comments carefully when you are experimenting with the program.

The functions you saw in Module 13 and Module 14 are sufficient for most of the things you need to do with strings, but occasionally something more esoteric will be required. Before re-inventing the wheel by trying to write your own function to do the job, you should check to see whether you can find one for the task in the Standard Library.

This is a good time to remind you again that Appendix G contains a useful summary of "everyday" string information.

22.4.3.2 Follow-up hands-on activities for STRMISC.CPP

□ Copy, study and test the program in STRMISC.CPP. Predict the output from each part of the program on the basis of the comments in the program and the values of the variables used. If you have a problem, change the values of the relevant string variables, test the program again, and continue this process until you reach a reasonable comfort level with the material.

◯ Instructor checkpoint 22.4

22.4.4 SSTREAM.CPP uses string streams to analyze contents of command-line parameters

```
// Filename: SSTREAM.CPP
// Purpose:  Illustrates the "input string stream" class.

/*
Typical Usage:
prompt> sstream 1/2 Scobey,Porter

Corresponding Output:

The 3 command-line parameters are:
sstream
1/2
Scobey,Porter

The numerator of the fraction is 1 and the denominator is 2.
Hello there, Porter Scobey!

*/

#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main(/* in */ int argc,
         /* in */ char* argv[])
{
    cout << "\nThe " << argc << " command-line parameters are: \n";
    for (int i = 0; i < argc; i++) cout << argv[i] << endl;
    cout << endl;

    string a[3] = { "", argv[1], argv[2] };
    // argv[0] is not used anywhere, so a[0] is set to the null string.

    istringstream iss1(a[1]), iss2(a[2]);

    int numer, denom;
    char ch;
    iss1 >> numer >> ch >> denom;
    cout << "The numerator of the fraction is " << numer
         << " and the denominator is "          << denom << "." << endl;

    string lastName, firstName;
    getline(iss2, lastName, ',');
    iss2 >> firstName;
    cout << "Hello there, " << firstName << " " << lastName << "!" << endl;

    return 0;
}
```

22.4.4.1 What you see for the first time in SSTREAM.CPP

Let's begin by admitting that this program, even though it's very short, is probably one of the hardest that you've seen to figure out quickly just by looking at the code. That's why we added some comments at the beginning to give you an example of some test data to try on your first run. It's also a little easier to explain what's happening in the context of that example, rather than in general.

The idea here is that we are entering command-line parameters that need to be analyzed in some way. In our particular example we enter

```
prompt> sstream 1/2 Scobey,Porter
```

in which the first parameter is a simple fractional expression and the second is a name in the order "lastname", "comma", "firstname".

By putting these parameters into *string stream* variables, we can then first "read" the fraction by reading the integer 1, then the character `'/'`, then the integer 2. Second we can use getline to "read" the lastname (everything up to but not including the comma) and then the firstname (everything beyond the comma).

Thus, putting these command-line parameters into string stream variables, allows us to "get at" the various parts of a single command-line parameter relatively easily.

22.4.4.2 Additional notes and discussion on SSTREAM.CPP

This sample program illustrates "reading from" a string stream variable (variable of data type `istringstream`), but you can also "write to" a string stream variable (provided the variable has a data type of `ostringstream`). Both of these types are defined in the `sstream` library, whose header file must be included if your program is to work with string streams.

22.4.4.3 Follow-up hands-on activities for SSTREAM.CPP

□ Copy, study and test the program in SSTREAM.CPP.

□ Make a copy of SSTREAM.CPP and call it SSTREAM1.CPP. Revise the program so that the last two sentences that are now output directly to the screen are first written out to two separate string stream variables of type ostringstream, and then the contents of those string stream variables are output to the screen. For this you need to know that if `oss` is an `ostringstream` variable, then `oss.str()` returns the string contents in `oss`.

◯ Instructor checkpoint 22.5

22.4.5 GCD_ITER.CPP shows how to find the greatest common divisor of two positive integers using an iterative algorithm

```
// Filename: GCD_ITER.CPP
// Purpose:  Finds the greatest common divisor (GCD) of pairs
//           of positive integers.  The GCD of each pair is
//           output immediately after the input of the pair.
//           The input of pairs is terminated by entering the
//           end-of-file character.

#include <iostream>
using namespace std;

int GCD(/* in */ int a, /* in */ int b)
// Pre:  Both a and b have been initialized with nonnegative integer
//       values, and at least one of a or b is strictly positive.
// Post: The function value is the GCD of a and b.
{
    int temp;
    while (b != 0)
    {
        temp = b;
        b = a % b;
        a = temp;
    }
    return a;
}

int main()
{
    cout << "\nThis program finds the greatest "
         << "common divisor (GCD) of \n"
         << "pairs of positive integers. \n\n";

    int first, second;
    while (cin)
    {
        cout << "Enter two positive integers, "
             << "separated by at least one \n"
             << "space, followed by ENTER, "
             << "or enter end-of-file to quit: \n";

        cin  >> first >> second;
        if (cin)
        {
            cout << "GCD(" << first  << ","
                           << second << ") = "
                 << GCD(first, second)         << "\n\n";
        }
    }

    return 0;
}
```

22.4.5.1 What you see for the first time in GCD_ITER.CPP

Actually, we hope that this is *not* the first time you are seeing the "well-known" *Euclidean algorithm* for finding the *greatest common divisor* (`GCD`) of two positive integers. At least it *used* to be well-known, in the days when it was part of every high school mathematics curriculum. And, in fact to implement the `Fraction` class of Module 15 properly you will have had to make use of an algorithm for finding the greatest common divisor.

A quick review, in case this *is* your first encounter: The Euclidean algorithm is based on the fact that if `a` and `b` are two positive integers, with `a > b`, then we can write

`a = q * b + r, where q == a / b and r == a % b, with r >= 0`

from which it follows that *any* common divisor of both `a` and `b` is a common divisor of both `b` and `r`, and vice versa. Thus the *greatest* common divisor of both `a` and `b` is the greatest common divisor of both `b` and `r`. That is, `GCD(a,b) == GCD(b,r)`. But because `r == a % b` we must have `r < b < a`, and hence finding `G(b,r)` is a "smaller but similar" problem when compared to finding `GCD(a,b)`. Thus, finding GCD(a,b) is an excellent candidate for the recursive approach.[1] Since at each stage, the value of `r` decreases, while remaining `>= 0`, it must eventually reach `0`. At that point the *other* value *is* the `GCD`, since `GCD(n,0) == n` for all `n > 0`. For example,

`GCD(30,12) == GCD(12,6) == GCD(6,0) === 6`

22.4.5.2 Additional notes and discussion on GCD_ITER.CPP

Finding the GCD of two positive integers comes in handy often enough that it is a candidate for your "useful utilities" directory. You may want to place a separately compiled GCD function, together with an appropriate header file, in your utility directory for ready use if needed.

22.4.5.3 Follow-up hands-on activities for GCD_ITER.CPP

□ Copy, study and test the program in GCD_ITER.CPP. The main driver function contains a loop, so you can find the GCD of as many positive integer pairs as you like without re-running the program. For example, try 30 and 12, 80 and 12, 54 and 30, 25 and 18, and 54321 and 12345.

□ When you are entering the two positive integers, does it really matter what order you enter them in (30 and 12, or 12 and 30)? If there is no difference that you can "see", is there any difference at all? If so, what? (Hint try some examples "manually". That is, work through the various calls of the function to perform the necessary calculations.)

◯ Instructor checkpoint 22.6

[1] You may or may not be familiar with *recursion* at the point when you are reading this. If not, don't worry about it.

Appendix A

C++ reserved words and some predefined identifiers

This Appendix contains a list of all the *reserved words* in Standard C++, and a small list of predefined identifiers. Recall the distinction between *reserved words* and *predefined identifiers*, which are collectively referred to (by us, at least) as *keywords*[1].

A.1 C++ Reserved Words

The reserved words of C++ may be conveniently placed into several groups. In the first group we put those that were also present in the C programming language and have been carried over into C++. There are 32 of these, and here they are:

```
auto   const     double  float  int       short   struct   unsigned
break  continue  else    for    long      signed  switch   void
case   default   enum    goto   register  sizeof  typedef  volatile
char   do        extern  if     return    static  union    while
```

There are another 30 reserved words that were not in C, are therefore new to C++, and here they are:

```
asm         dynamic_cast  namespace  reinterpret_cast  try
bool        explicit      new        static_cast       typeid
catch       false         operator   template          typename
class       friend        private    this              using
const_cast  inline        public     throw             virtual
delete      mutable       protected  true              wchar_t
```

[1]But be aware that this terminology is not standard. For example, some authors will use *keyword* in the same sense that we have used *reserved word*.

The following 11 C++ reserved words are not essential when the standard ASCII character set is being used, but they have been added to provide more readable alternatives for some of the C++ operators, and also to facilitate programming with character sets that lack characters needed by C++.

```
and      bitand   compl   not_eq   or_eq   xor_eq
and_eq   bitor    not     or       xor
```

Note that your particular compiler may not be completely up-to-date, which means that some (and possibly many) of the reserved words in the preceding two groups may not yet be implemented.

A.2 Some Predefined Identifiers

Beginning C++ programmers are sometimes confused by the difference between the two terms *reserved word* and *predefined identifier*, and certainly there *is* some potential for confusion.

One of the difficulties is that some keywords that one might "expect" to be reserved words just are not. The keyword `main` is a prime example, and others include things like the `endl` manipulator and other keywords from the vast collection of C++ libraries.

For example, you *could* declare a variable called `main` inside your `main` function, initialize it, and then print out its value (but don't!). On the other hand, you could *not* do this with a variable named `else`. The difference is that `else` is a reserved word, while `main` is "only" a predefined identifier.

Here is a very short list of some of the predefined identifiers you will encounter in this text:

```
cin       endl   include     INT_MIN   iostream   cout   INT_MAX
iomanip   main   MAX_RAND    npos      NULL       std    string
```

Appendix B

The ASCII character set and codes

This Appendix displays a table of the Standard ASCII[1] Character Set and Codes, which contains 128 characters with numerical codes in the range from 0 to 127 (decimal). There are several "extended" ASCII character sets in use which contain 256 characters, including characters for drawing "character graphics" and oddball characters like the "smiley face", but these do not concern us here.

Let's make a few observations about the table, some of which may come in handy when you have to manipulate characters in your programs for one reason or another.

First, note that the table is presented in four "conceptual columns", and that the first one differs from the remaining three since it has two character columns while the other three have only one. The ASCII characters with codes in the range 0 to 31 are called *control codes* and are "invisible", i.e., they do not represent printable characters on the screen. Instead they provide a way to give instructions to peripheral devices such as printers. For example, the character with code `7` (`^G` or `BEL`) will cause the little bell on a terminal to ring when it is sent. Note that each character has "CTRL+key" representation using the `^` character to represent the CONTROL (CTRL) key (as in `^G`) and a two- or three-character mnemonic (`BEL`[2]).

The remaining characters all represent printable characters, except for the last (code `127`, or `DEL`), which is also a control code. The one with code 32 is a special case. This one represents the blank space character which is, of course, invisible by its nature, but is nevertheless regarded as a printing character since it moves the cursor one space to the right, leaving a "blank space" behind.

[1] ASCII is an acronym for "American Standard Code for Information Interchange".

[2] You need not worry about the meaning of these various mnemonics. Only if you need to become much more familiar with the ASCII control codes than is required here will you need to concern yourself with such details.

Here are some other occasionally useful facts about the table:

- The single digits, the capital letters, and the lower-case letters each form a contiguous group of characters.
- The group of capital letters comes before the group of lower-case letters.
- The number of code positions separating any given capital letter from its corresponding lower-case letter is 32.

ASCII Table

Each character appears to the right of its numerical code.

0	^@ NUL	32	space	64	@	96	‘
1	^A SOH	33	!	65	A	97	a
2	^B STX	34	"	66	B	98	b
3	^C ETX	35	#	67	C	99	c
4	^D EOT	36	$	68	D	100	d
5	^E ENQ	37	%	69	E	101	e
6	^F ACK	38	&	70	F	102	f
7	^G BEL	39	’	71	G	103	g
8	^H BS	40	(	72	H	104	h
9	^I HT	41	)	73	I	105	i
10	^J LF	42	*	74	J	106	j
11	^K VT	43	+	75	K	107	k
12	^L FF	44	,	76	L	108	l
13	^M CR	45	-	77	M	109	m
14	^N SO	46	.	78	N	110	n
15	^O SI	47	/	79	O	111	o
16	^P DLE	48	0	80	P	112	p
17	^Q DC1	49	1	81	Q	113	q
18	^R DC2	50	2	82	R	114	r
19	^S DC3	51	3	83	S	115	s
20	^T DC4	52	4	84	T	116	t
21	^U NAK	53	5	85	U	117	u
22	^V SYN	54	6	86	V	118	v
23	^W ETB	55	7	87	W	119	w
24	^X CAN	56	8	88	X	120	x
25	^Y EM	57	9	89	Y	121	y
26	^Z SUB	58	:	90	Z	122	z
27	^[ESC	59	;	91	[	123	{
28	^\ FS	60	<	92	\	124	|
29	^] GS	61	=	93	]	125	}
30	^^ RS	62	>	94	^	126	~
31	^_ US	63	?	95	_	127	DEL

Appendix C

More C++ operators and their precedence

This Appendix contains an abbreviated table of C++ operators and their precedence. C++ is one of the most operator-rich programming languages and the table presented here contains only those operators that you will encounter (and some of which you will use frequently) in this Lab Manual.

The table is arranged from highest to lowest precedence as you go from top to bottom. Operators on the same line have the same precedence. Two groups of operators with the same precedence extend over more than one line and are enclosed between dashed lines.

Precedence of operators is something that we sometimes take for granted, particularly if we are thoroughly familiar and comfortable with the standard precedence rules for the common arithmetic operators. But being too complaisant can put us at some peril, and particularly in a language like C++ which has such a variety of operators it pays to be on our guard.

As a brief example, note from the table that the input/output operators (`>>` and `<<`) have a higher precedence than the relational operators but a lower precedence than the arithmetic operators. This means that a statement like

```
cout << 2 + 7;
```

"does the right thing" and displays a value of 9, while a statement like

```
cout << 2 < 7;
```

which you might expect to output 1 (or `true`), i.e., the value of the relational expression, in fact causes a syntax error, since the precedence of the operators involved means that parentheses are required as follows:

```
cout << (2 < 7);
```

The example is, of course, somewhat artificial (since how often do we really want to output the value of a conditional expression?), but it makes the point.

```
::                  scope resolution operator
-------------------------------------------------------------
.                   member selection via struct or class object
->                  member selection via pointer
[]                  array index
()                  parentheses as function call
-------------------------------------------------------------
++   --             postfix versions of increment/decrement
-------------------------------------------------------------
++   --             prefix  versions of increment/decrement
sizeof              for computing storage size of data
!                   logical not
+ -                 unary, i.e., one-argument, versions
new                 allocates memory to dynamic object
delete              allocates memory to dynamic array
new []              deletes memory allocated to dynamic object
delete []           deletes memory allocated to dynamic array
-------------------------------------------------------------
*    /    %         multiplication, division and modulus
+    -              addition and subtraction
>>   <<             input and output stream operators
<    <=   >   >=    relational
==   !=             more relational
&&                  logical and
||                  logical or
-------------------------------------------------------------
=    *=   /=   %=   assignment
+=   -=             more assignment
-------------------------------------------------------------
?:                  conditional
-------------------------------------------------------------
```

Appendix D

C++ programming style guidelines

This Appendix contains a summary of our recommended programming style guidelines. Your particular guidelines may or may not agree with these, but you should examine closely what is given here and make any notes about the differences. For the sake of brevity, few specific examples are given in this Appendix, but the sample programs throughout the text may be consulted for illustrations of the points stated here.

D.1 The "Big Three" of Programming Style

1. Name things well!
2. Be consistent!
3. Re-format continually! (This will help to *preserve* consistency.)

D.2 Spacing and Alignment

1. Put each program statement on a separate line, unless you can make a very good argument to explain why you didn't.
2. Use vertical spacing to enhance readability. For example, use one or more blank lines to separate logically distinct parts of a program.
3. Use horizontal spacing to enhance readability. For example, place a blank space on each side of an operator such as `<<` or `+`.
4. Use an indentation level of four (4) spaces, and always indent:
    - A function body with respect to the corresponding function header.

- A loop body with respect to the loop construct itself.
- The body of an `if` and (if present) the body of the corresponding `else`. In any `if...else` statement, the body of statement(s) corresponding to the `if` should line up with those corresponding to the `else`.
- Each nested loop or nested if-statement should be indented one level with respect to the enclosing loop or if-statement.

5. Align the beginning of each statement in a function body (or loop body, or `if` body, or `else` body). Also, place the braces enclosing any such function body (or loop body, or `if` body, or `else` body) on separate lines and align them with the beginning of the function header (or loop construct, or `if` construct, or `else` construct).

6. Classes are formatted, in general, like this:

```
class NameOfClass
{
    friend friend_prototype (with pre/post conditions)

public:

    member function prototypes (with pre/post-conditions)


private:

    data member declarations
};
```

D.3 Naming and Capitalization

The importance of choosing a good name for each program entity cannot be overemphasized, and the following two items should be regarded as *rules*:

- Names must be meaningful within their given context whenever possible, which is most of the time.

  One exception to this rule is the use of single-letter loop control variables in those situations where no particular "meaning" attaches to the variable.

- Names must always be capitalized consistently, according to whatever conventions are being used for each category of name.

The two most common program entities for which you will have to choose names are variables and functions. Here are the capitalization conventions we have used in this text:

Variables begin with a lower-case letter, and in fact use all lower-case except that if the name consists of two or more words the first letter of the second and subsequent words is capitalized.

Examples: `cost`, `numberOfGuesses`, `valid`, `timeToQuit`

Note as well, and this relates back more to the *choice* of names, that variables that represent objects are noun-like, while boolean variables tend to be more like adjectives.

Named constants are spelled with all capital letters, and with an underscore separating individual words.

Examples: `MAX_NUMBER_OF_GUESSES`, `PI`

Value-returning functions have names like variables and are capitalized similarly except that their names *begin* with a capital letter.

Example: `CelsiusTemp`

Void functions are capitalized exactly like value-returning functions, but the names of void functions have one important significant feature that distinguishes them from value-returning functions:

The name of every void function begins with a verb.

Examples: `DescribeProgram`, `GetPositiveIntegerFromUser`

User-defined types start with a capital and the first letter of each subsequent word in the name is also capitalized:

Pointer types and variables permit the following three variations when making a declaration of a pointer variable, though we have used (and recommend) the first:

```
SomeType* p;
SomeType *p;
SomeType * p;
```

D.4 Commenting

When commenting your code you should strive to be informative without being excessive. Your goal is to have *readable* code, and too many comments are just as counter-productive as too few. Local rules will govern what absolutely must be included, but we would add the following:

- Always include pre-conditions and post-conditions in a function definition.
- Always place C-style comments in the function header of a function definition to indicate the conceptual nature of the parameters with respect to the direction of information flow.

D.5 Program Structure

When your entire program is contained in a single source code file, that file should have the following structure:

- First, comments indicating the name of the file, purpose of the program, and anything else required by local "authorities".
- Next, the necessary "includes" for the required header files.
- Definitions for any global constants or data types (but *no* global variables, at least not at this stage of your career).
- Prototypes for any required functions, grouped in some intelligent way that will depend on the nature of the program.
- The `main` function.
- Finally, definitions for each of the functions whose prototypes appeared before `main`, and in the *same order* as the corresponding prototypes.

D.6 Random Thoughts

Here are some random thoughts to keep in mind for improving your code, and/or keeping it "safe", and you may wish to add more of your own:

1. Use *named constants* whenever appropriate.
2. Use `typedef` frequently, or, when in doubt as to whether you should, *use* `typedef`.
3. Until you become a more "advanced" C++ programmer, only use the increment and decrement operators in stand-alone statements.
4. If you have written good pseudocode, then portions of that pseudocode should be re-usable as comments in the actual code. Avoid, however, repeating *all* of your pseudocode in your comments. A need to do this would indicate that your code itself is not very well written.

Appendix E

Guidelines for program development

This Appendix contains a summary of (somewhat oversimplified) guidelines for software development. We consider two approaches—structured (or procedural) and object-oriented—but it is worth pointing out that "C++" does not appear in the title above. This is intentional, and a reflection of the fact that these guidelines are essentially language-independent.

You should consider these guidelines (or some other guidelines, if you don't like these) very carefully when you are trying to develop software of *any* size or complexity. "*Anything worth doing is worth doing well*," is an old adage, and nowhere is it more important to plan well and think hard about what you are doing if you are to succeed.

In the two sections that follow, you will find considerable repetition. This too is intentional, the better to let you compare the similarities and differences of these two approaches to software development.

We cannot emphasize too strongly how oversimplified both of these sets of guidelines are, but they give you the necessary flavor in each case, and even following these brief recommendations in a disciplined manner will save you a great deal of frustration or worse.

E.1 For structured (procedural) development

1. Analyze the problem. Analyze it to death if necessary, but do *not* proceed until you *understand* it (or at least until you *think* you understand it, which may or may not be the case). But beware of "analysis paralysis".

*Here you describe **what** but not **how**.*

2. Specify *what* has to be done, but *not* how to do it. This should include specification of what the input and output of the final program will look like, with sample data, and can include user interface features.

Now for the how!

3. Design and develop the algorithm(s) that will accomplish the task at hand.

   This is where you apply top-down design with step-wise refinement, draw design tree diagrams, and write pseudocode. This is also where you test your algorithms with your sample data from the previous step.

   Don't go near the computer before reaching this point, because:

And don't you forget it!

   THE SOONER YOU START TYPING, THE LONGER IT'S GOING TO TAKE.

4. Translate your pseudocode into actual code, build your program, and perform the edit-compile-link-run-test cycle until you have a complete program that will compile, link and run.

   This is where you apply top-down implementation and code-testing that parallels your top-down design from the previous step.

5. Once your entire program compiles, links and runs, thoroughly test the program, putting it through a bank of tests large and detailed enough to convince you that it does indeed satisfy its original specifications.

6. Complete the documentation for your program, which of course has been an ongoing effort throughout the development. There are always two dominant goals to keep in mind when documenting a program:

   - Ensuring that the source code is *readable*.
     For this you only need (ideally) to apply the programming style guidelines of Appendix D.
   - Ensuring that the program has a good *user interface* when it runs.
     This means that the program must:
     - Describe itself (and identify the programmer, if required).
       The program may either do this automatically when it runs, especially in the case of small programs, or it may optionally provide the information if the user chooses to view it.
     - Provide good user prompts for all keyboard input.
     - Echo all input data somewhere in the output.
     - Display all output in a way that is "pleasing to the eye".

E.2 For object-oriented development

1. Analyze the problem. Analyze it to death if necessary, but do *not* proceed until you *understand* it (or at least until you *think* you understand it, which may or may not be the case). But beware of "analysis paralysis".

2. Try to identify appropriate and useful ADTs for the problem. Think about what "objects" are involved in the problem and try to "classify" the objects. These classifications will become the ADTs, and in many programming languages—C++, for instance—the ADTs will become *classes*. A useful rule of thumb to keep in mind is that if you write out a description of your problem, the nouns in the description tend to become class objects, while the verbs tend to become the class methods (member functions). *Find appropriate ADTs.*

3. Decide how the ADTs will interact with one another and what relationships they have to one another. This will include things like ADTs being independent of one another, one ADT "using" another ADT, one ADT being "inherited" by another, one ADT being "composed of" several others, and so on. *Discover relationships and interactions between/among the ADTs.*

4. Decide what you want your user interface to look like, and then write the pseudocode for the top-level driver of your program. *Design the user interface.*

   Don't go near the computer before reaching this point, because:

   THE SOONER YOU START TYPING, THE LONGER IT'S GOING TO TAKE. *And don't you forget it!*

5. Translate your pseudocode into actual code, build your program, and perform the edit-compile-link-run-test cycle until you have a complete program that will compile, link and run. Begin with an implementation of your user interface as a "shell" driver, and use it to drive the rest of the program as you translate the various ADTs to actual code and test the results. Which ADTs are implemented first will of course depend on the nature of the problem being solved.

6. Once your entire program compiles, links and runs, thoroughly test the program, putting it through a bank of tests large and detailed enough to convince you that it does indeed satisfy its original specifications.

7. Complete the documentation for your program, which of course has been an ongoing effort throughout the development. There are always two dominant goals to keep in mind when documenting a program:

   - Ensuring that the source code is *readable*.
     For this you only need (ideally) to apply the programming style guidelines of Appendix D.

- Ensuring that the program has a good *user interface* when it runs.
  This means that the program must:
  - Describe itself (and identify the programmer, if required).
    The program may either do this automatically when it runs, especially in the case of small programs, or it may optionally provide the information if the user chooses to view it.
  - Provide good user prompts for all keyboard input.
  - Echo all input data somewhere in the output.
  - Display all output in a way that is "pleasing to the eye".

Appendix F

How to behave when meeting a new data type

This Appendix contains a summary of what to look for when you encounter a new data type in a programming language. It gives you a sequence of questions that it is useful to ask about that data type, and a consistent framework within which to learn all you need to know about the new type. It is worth pointing out (again, as in Appendix E) that "C++" does not appear in the title. This is again intentional, and again a reflection of the fact that these guidelines are also essentially language-independent.

So, the following questions should help to guide you through your period of getting acquainted with any new data type you might encounter:

- Why would you want to have a data type like this? Or, in what kind of situation would you use this data type?
- What does a "picture" of the data type look like? Or, can you draw a pictorial representation of a value of this data type?
- How do you define the data type? More specifically, what are the syntax, semantics, and terminology involved in such a definition?
- How do you declare and initialize a variable of this type?
- Can you assign to a variable of this type a constant of the same type, or a variable of the same type? Can you perform other *aggregate operations* (such as equality testing, or input/output, for example) on variables of this type?
- How do you obtain access to a component of this data type?
- What issues must be considered if you wish to pass a value of this type to a function, or return a value of this type from a function?

- What issues must be considered if you wish to combine this data type with other data types (or with itself) to define yet another new data type?
- Are there any "idioms" associated with the use of this data type?

Appendix G

Summary of Useful Information on C-strings and C++ String Objects

This Appendix summarizes many of the things you may find it useful to know when working with either C-strings or objects of the C++ `string` class.

The term *string* generally means an ordered sequence of characters, with a first character, a second character, and so on, and in most programming languages such strings are enclosed in either single or double quotes. In C++ the enclosing delimiters are double quotes. In this form the string is referred to as a *string constant* and we often use such string constants in output statements when we wish to display text on the screen for the benefit of our users. For example, the usual first C++ program displays the string constant `"Hello, world!"` on the screen with the following output statement:

```
cout << "Hello, world!" << endl;
```

However, without string variables about all we can do with strings is output string constants to the screen, so we need to expand our ability to handle string data. When we talk about strings in C++, we must be careful because the C language, with which C++ is meant to be backward compatible, had one way of dealing with strings, while C++ has another, and to further complicate matters there are many non-standard implementations of C++ strings. These should gradually disappear as compiler vendors update their products to implement the string component of the C++ Standard Library.

As a programmer, then, you must distinguish between:

- An "ordinary" array of characters, which is just like any other array and has no special properties that other arrays do not have.
- A C-string, which consists of an array of characters terminated by the null character `'\0'`, and which `does` have properties over and above those of

an ordinary array of characters, as well as a whole library of functions for dealing with strings represented in this form. Its header file is `cstring`. In some implementations this library may be automatically included when you include other libraries such as the `iostream` library.

- A C++ string, which is a `class` data type whose actual internal representation you need not know or care about, as long as you know what you can and cant do with variables (and constants) having this data type. Such variables are in fact objects that are instances of the C++ `string` class. There is a library of C++ string functions as well, available by including the `string` header file.

Both the C-string library functions and the C++ `string` library functions are available to C++ programs. But, dont forget that these are two *different* function libraries, and the functions of the first library have a different notion of what a string is from the corresponding notion held by the functions of the second library. There are two further complicating aspects to this situation: first, though a function from one of the libraries may have a counterpart in the other library (i.e., a function in the other library designed to perform the same operation), the functions may not be used in the same way, and may not even have the same name; second, because of backward compatibility many functions from the C++ string library can be expected to work fine and do the expected thing with C-style strings, but not the other way around.

The last statement above might seem to suggest we should use C++ strings and forget about C-strings altogether, and it is certainly true that there is a wider variety of more intuitive operations available for C++ strings. However, C-strings are more primitive, you may therefore find them simpler to deal with (provided you remember a few simple rules, such as the fact that the null character must always terminate such strings), and certainly if you read other, older programs you will see lots of C-strings. So, use whichever you find more convenient, but if you choose C++ strings and occasionally need to mix the two for some reason, be extra careful.

To understand strings, you will have to spend some time studying sample programs. This study must include the usual prediction of how you expect a program to behave for given input, followed by a compile, link and run to test your prediction, as well as subsequent modification and testing to investigate questions that will arise along the way. In addition to experimenting with any supplied sample programs, you should be prepared to make up your own.

In the following examples we attempt to draw the distinction between the two string representations and their associated operations. The list is not complete, but we do indicate how to perform many of the more useful kinds of tasks with each kind of string. The left-hand column contains examples relevant to C-strings (see Module 13) and the right-hand column shows analogous examples in the context of C++ strings (see Module 14). The two types of strings are not mixed in the sample programs of Modules 13 and 14.

```
C-strings  (#include <cstring>)
================================
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Declaring a C-string variable
------------------------------
char str[10];


Initializing a C-string variable
---------------------------------
char str1[11] = "Call home!";
char str2[] = "Send money!";
char str3[] = {'O', 'K', '\0'};
Last line above has same effect as:
char str3[] = "OK";


Assigning to a C-string variable
---------------------------------
Can't do it, i.e., cant do this:
char str[10];
str = "Hello!";


Concatenating two C-strings
----------------------------
strcat(str1, str2);
strcpy(s, strcat(str1, str2));


Copying a C-string variable
----------------------------
char str[20];
strcpy(str, "Hello!");
strcpy(str, otherString);


Accessing a single character
-----------------------------
str[index]


Comparing two C-strings
-----------------------
if (strcmp(str1, str2) < 0)
    cout << "str1 comes 1st.";
if (strcmp(str1, str2) == 0)
    cout << "Equal strings.";
if (strcmp(str1, str2) > 0)
    cout << "str2 comes 1st.";
```

```
C++ strings  (#include <string>)
================================
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Declaring a C++ string object
------------------------------
string str;


Initializing a C++ string object
---------------------------------
string str1("Call home!");
string str2 = "Send money!";
string str3("OK");

string str4(10, 'x');


Assigning to a C++ string object
---------------------------------
string str;
str = "Hello";
str = otherString;


Concatenating two C++ string objects
-------------------------------------
str1 += str2;
str = str1 + str2;


Copying a C++ string object
----------------------------
string str;
str = "Hello";
str = otherString;


Accessing a single character
-----------------------------
str[index]
str.at(index)
str(index, count)


Comparing two C++ string objects
---------------------------------
if (str1 < str2)
    cout << "str1 comes 1st.";
if (str1 == str2)
    cout << "Equal strings.";
if (str1 > str2)
    cout << "str2 comes 1st.";
```

```
Finding the length of a C-string
--------------------------------
strlen(str)
```

```
Finding the length of a C++ string object
-----------------------------------------
str.length()
```

```
Output of a C-string variable
-----------------------------
cout << str;
cout << setw(width) << str;
```

```
Output of a C++ string object
-----------------------------
cout << str;
cout << setw(width) << str;
```

In what follows, keep in mind that `cin` ignores white space when reading a string, while `cin.get`, `cin.getline` and `getline` do not. Remember too that `cin.getline` and `getline` *consume the delimiter* while `cin.get` does not. Finally, `cin` can be replaced with any open input stream, since file input with `inFile`, say, behaves in a manner completely analogous to the corresponding behavior of `cin`. Analogously, in the output examples given immediately above, `cout` could be replaced with any output stream variable, say `outFile`.

```
Input of a C-style string variable
----------------------------------
cin >> s;
cin.get(s, numCh+1);
cin.get(s, numCh+1,'\n');
cin.get(s, numCh+1,'x');
cin.getline(s, numCh+1);
cin.getline(s, numCh+1, '\n');
cin.getline(s, numCh+1, 'x');
```

```
Input of a C++ string object
----------------------------
cin >> s;


getline(cin, s);

getline(cin, s, 'x');
```

A useful naming convention for C-strings is illustrated by examples like

```
typedef char String80[81];
typedef char String20[21];
```

in which the two numbers in each definition differ by 1 to allow for the null character `'\0'` to be stored in the array of characters, but to *not* be considered as part of the string stored there. No analog to this naming convention is necessary for C++ strings, since for all practical purposes, each C++ string variable may contain a string value of virtually unlimited length.

Index