How to Survive and Prosper in the C++ Laboratory

Part I

A Lab Manual
for
Beginning C++ Programmers

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

Latest Revised Printing: August 23, 2004

Table of Contents

Preface

An "objects later" approach which can be used as a text supplement or for independent study

How to Survive and Prosper in the C++ Laboratory is designed to be used as a supplement to any C++ text that uses an "objects somewhat later" approach to a study of the language. It can also be used for independent study to provide a quick but thorough introduction to ANSI/ISO[1] Standard C++ by anyone familiar with another programming language, such as Pascal or Java.

The title is an expression of the author's hope (and belief) that if a student exercises due diligence by working through all of the activities this manual offers, then he or she will in fact achieve the title's promise. This is not a full text on C++, however, and it will be helpful to have at hand one of the many standard C++ texts[2], or references, for use as needed. On the other hand, with each sample program we do explicitly point out what is new and exciting about that particular program, and often add extra material to summarize, or place into context, the feature(s) currently being illustrated. Thus a reader with some prior programming language experience in another language would not, in fact, necessarily need an auxiliary text to move quickly through the modules and gain a good understanding of C++.

A platform-independent discussion of the programming environment

Use of the Lab Manual may begin with a discussion of all the things a programmer needs to know about the local programming environment to get started. If so, then students should be directed to one or more of the Appendices F through I. The required programming environment topics are treated there in a generic-question-with-local-answers-to-be-supplied manner that makes the treatment platform independent. This may or may not be a useful feature of our presentation. However, many authors do seem to throw up their hands in the face of programming environment questions, and leave the whole thing to "local documentation". In the aforementioned Appendices we at least go one step further, by making sure the relevant questions are asked, and providing a

[1] ANSI is an acronym for the American National Standards Institute and ISO stands for International Standards Organization. On November 14, 1997 the combined ANSI X3J16/ISO WG21 C++ Committee approved their latest draft and submitted it for official sanction, which came late in 1998.

[2] As a point of reference, the reader may be interested to know that during the development of this Lab Manual the author has used as the main course text, at different times, both *Programming and Problem Solving with C++* by N. Dale, C. Weems and M. Headington (Sudbury, MA: Jones and Bartlett, 1997) and *Problem Solving with C++, Second Edition* by Walter Savitch (Reading, MA: Addison-Wesley, 1999). Both of these texts have more recent editions.

place to record each answer.

Introductory Modules 1 and 2 contain much more detail than most later Modules.

Modules 1 and 2 provide introductions to C++ and to program development, respectively. After completing these two Modules, but again only if desired or necessary, students can return to a discussion of the local programming environment in Appendix J, and look at the question of organizing their files and placing them in appropriately-named subdirectories. In any case, Modules 3 to 5 in the mainstream part of the manual, next deal with simple input and output (I/O), and by then we're off and running.

A menu-driven "shell" program is introduced early and reused frequently as a generic driver.

The student is also introduced very early (Module 6) to a simple menu-driven "shell" program that makes use of both selection and looping, in a very natural way, and can be used as a starting point for the development of many programs in the hands-on exercises that follow, if desired. The hands-on activities involving the shell permit the student to do reasonably interesting things almost immediately. The Manual structure does permit this Module to be postponed, if the instructor does not wish to introduce this much this soon, and all activities in the following Modules that recommend use of the shell are explicitly identified by a margin note. However, postponing that particular Module will take something away from the student's experience in the following Modules. The frequent subsequent opportunities for using the shell program as a generic driver in different situations expose the student, early and in a simple way, to the benefits of code reuse.

Independent Modules permit various orderings.

Subsequent Modules fall into two categories. In the first and larger group, each Module deals more or less with one specific topic—the remaining material on selection or looping, for example, or programmer-written value-returning functions. The second, and smaller, group consists of a number of consolidation Modules, in which the various topics from previous Modules are brought together in more complex programs. Because the Modules on individual topics that lie between the consolidation Modules are essentially independent, an instructor is free to cover the intervening Modules in any desired order.

The Module structure remains consistent throughout the text.

Each Module has essentially the same structure, which is suggested by the following outline:

- List of Module objectives
- List of files associated with the Module
- Module overview
- Discussion of sample programs included with the Module
    - First program listing
        - What you see for the first time in this program
        - Additional notes and discussion on this program
        - Follow-up hands-on activities for this program
    - Second program listing ...

Those Modules from the Appendices designed to be used when dealing with the programming environment may not have any associated files, and also will have the section

- Discussion of sample programs included with the Module

replaced by one entitled

- Questions needing local answers, with follow-up hands-on activities

Furthermore, a programming environment topic will occasionally appear within the sample program discussions of a mainstream Module, if that is the most appropriate location for that topic. In Module 1, for example, which introduces the C++ programming language, there is a short programming environment section that deals with what is needed to get a program up and running on the local system. And in Module 13, Section 13.4.4, we see a program with keyboard input terminated by an end-of-file character, so that section contains a short programming environment subsection dealing with the system-dependent question of how to enter an end-of-file character from the keyboard on the local system.

Among the design goals for the hands-on activities are the following:

Design goals for the hands-on activities

1. Keeping them to a reasonable number and length, so that most students who are willing to apply themselves can be expected to complete *all* of them, and neither the instructor nor the student will have to waste time choosing or wondering which of them to do.

   Just the "right" number

2. Making sure that each sample program, and the sequence of activities associated with it, are all designed to illustrate some feature of C++, or to provide important and relevant hands-on experience; nothing is there just to provide busy work to keep students occupied.

   No "fluff"

3. Making it easy for an instructor to monitor a student's progress by asking, in most of the activities, for the creation of files with specific names, the existence and/or the functionality of which can then be checked by the instructor or an assistant. In many programming environments this process can be automated, especially if it is possible in the local environment to read and run particular files in each student's account directly from a lab instructor's workstation. In addition, throughout the hands-on activities, at strategic points, you will see a line like this:

   Specific file names make it easy to monitor student progress.

   ○ Instructor checkpoint m.n for evaluating prior work

   Milestone markers

   This is really a (numbered) message to the student, which makes the following statement:

> Your instructor may wish to check, at this point, and before you proceed, your progress since the last such checkpoint If so, you should be prepared to show, submit in some way, or possibly even demonstrate, whatever may be necessary to establish that you have, in fact, finished the most recent activities up to that point. This will normally mean that you must have prepared source code and/or executable files having particular names, and that when one of those executable files is run, the program must exhibit particular functionality.

In other words, it's a place where the instructor may (or may not) choose to perform the suggested check. And, depending on the tools available in the local programming environment, and the nature of the check that an instructor wants to make at the time, the check may be automated to a greater or lesser extent.

Activities sometimes extend the material.

4. Using the occasional activity as a vehicle for introducing some new, but relatively minor, aspect of the C++ language that hasn't yet shown up explicitly in any of the sample programs. For example, the escape sequences `\t` and `\n` might show up in a sample program, with the additional escape sequence `\"`, and why it is needed, appearing only in a follow-up hands-on activity.

Programming, like riding a bicycle, is a skill that needs to be developed.

If you are a student, you should be aware that computer programming is a bit like riding a bicycle, playing the piano, or swimming, in at least the following respects:

- First of all, it is a skill, and therefore a few people are "naturals", but most people require some effort (some a great deal more than others), and (let's face it) some people never quite get the hang of it.
- Initial attempts to acquire the skill are probably best supported by acquiring a few basics, observing how other (good) practitioners do it, and then imitating them as best you can.
- Finally, with time, you grow tired of imitating others, and begin to strike out on your own, perhaps developing an independent approach and a new style that reflects your own personality.

Initial bewilderment is much more widespread than you might think.

Beginning students in *computing science*[3] often have the experience of finding themselves suddenly thrust into a strange environment, where they are faced very quickly with a "sink or swim" situation. Much is expected of them, they may not have been told many of the answers they need to know, and they cannot even be expected to know all of the questions they need to ask. Some of the students in this situation may not yet have the necessary survival skills, if only because no one has thought to provide them.

[3] Also referred to as *computer science* by a large number of people, possibly because they have no fear of actual hardware.

Yell for help if you start to sink.

A major goal of this Lab Manual is to make each student's ride over the inevitable rough spots as smooth as possible, while also making it relatively easy for an instructor to monitor the student's progress. However, the most important thing to be remembered by students, and emphasized repeatedly by instructors, is that in all cases where difficulty is encountered, help must be sought, and sought as soon as possible after the discovery that it is needed, since it is always easier than you may think to fall behind.

An expedition, not a tour

However, do not lose sight of the fact that you are on *an expedition, not a tour*[4]. You can't just sit at the back of the bus and watch the world of C++ roll by. You're going to have to get out and trudge through some of the thickets.

Pedagogical approach

The particular pedagogical approach used throughout includes emphasis on each of the following:

"bugging"

1. *Bugging* (by which we shall mean the deliberate insertion of bugs into perfectly good sample programs, just to see the effect on the compiler or the output, and making notes for future reference), *antibugging* (programming in such a way that bugs are avoided, if at all possible), and *debugging* (finding and eliminating the inevitable bugs that will occur, sadly, despite all your best efforts).

   An attempt is made to keep the student's mind alive during the bugging process. To this end, many of the bugging activities are worded in such a way that the student is encouraged to become more familiar with the requisite terminology, in context, rather that simply going through the required motions. As a simple example, in "bugging" a program containing a loop, instead of being asked explicitly to remove the line of code

   ```
   int n = 1;
   ```

   (or referring to its line number) before recompiling and running the program again, the student might be asked to remove "the statement that initializes the loop control variable".

   On the other hand, when there is nothing in particular to be gained by this approach, a direct reference to a source code line number may be used.

Testing, testing, testing

2. Testing sample programs with typical data to see how they behave under "normal" conditions, and also with extreme values to see when and why they "break"; guiding the development of test sets for the student's own programs; modifying programs to produce related but different output, or to produce the same output in a different way.

[4] This phrase, which captures so well the nature of things, is lifted shamelessly from the preface to *Navigating C++ and Object-Oriented Design* by Paul Anderson and Gail Anderson, (Upper Saddle River, NJ: Prentice Hall, 1998). To drive the point home, the author has been known to appear at his first class of the term decked out in pith helmet, short pants, knee socks, hiking boots and walking stick.

Style and substance are equally important.

3. Applying a consistent set of programming style and documentation guidelines throughout, since our view is that it is just as important for a program to be clear and concise as it is to be correct and efficient. As far as we are concerned, "clear and concise" will mean "source code that is easy to read and understand, and a user interface that makes the program easy to use when it runs".

   In an attempt to practice what we preach, virtually every sample program that the student encounters will be at least minimally documented, and will display a self-description when it runs. On those rare occasions when this is *not* the case, there is a reason, and the reason will be given in the text. It always gives the user a warm fuzzy feeling, when a program runs, to be told by the program itself what that program is going to do, and what input it needs, in order to do whatever it does. And this is true, of course, independent of the program's complexity.

Structured programming

4. Initial program development via top-down design with step-wise refinement, pseudocode, structure diagrams, embedded debugging code, and stubs and drivers (in keeping with the "objects later" approach).

Programming with objects comes later.

5. Programming that uses *structured data types*, including *classes* and *objects*, will be covered in Part II of this Lab Manual.

Constructive criticism of any kind is always welcome. In particular, 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 the many instructors who have used this Lab Manual during its development over several years. Special thanks are due to Patrick Lee for his detailed lists of typos and comments, which have thankfully grown shorter at time goes by. Many thanks as well to Pawan Lingras and Stavros Konstantinidis, each of whom supplied a number of corrections and suggestions along the way. Among the students who have been very helpful in pointing out errors in both Part I and Part II of the Manual I would especially like to thank Maurice McNeille and John Wallace.

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

August 23, 2004

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.

  There are some exceptions to this particular convention, with respect to C++ reserved words that are used frequently in a very obvious way. For example, from a certain point we may often use terms like “if-statement”, “while-loop”, “void functions”, and so on, without using the typewriter-like monospaced font to highlight the C++ reserved word.

  See also the **Objectives** section of Module 12 on page 95.

Here are some other conventions that we follow:

- Each sample C++ program is in a file with a `.cpp`[5] extension, and the name of the file 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.

- All supplied textfiles of input data will have an extension of`.in`, (or `in?`, where `?` will be replaced by a digit indicating which of several input files is under consideration). The name of a data file will be the same as the name of the program file for which the given file is an input data file.

- All textfiles, other than those that are also input data files, will have an extension of`.txt`.

- In earlier printings, we adhered 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) in order to retain backward compatibility with older DOS environments. With this latest revision, it seems the time has come when we can use longer names for files, in the interest of providing more meaningful file identifiers, and in the hope that it will not cause any undue hardship to users. When longer names consist of more than one word, the words in the name are separated by an underscore (_). Notable exceptions are `filename` and `textfile` which, for some reason (probably convenience) the author has come to regard as single words in their own right.

[5] Some operating systems are case-sensitive, i.e., they distinguish between capital letters and lowercase letters in filenames, and some are not.

Module 1

A first look at C++: simple programs that only display text

1.1 Objectives

- To understand the overall structure of a simple C++ program.
- To learn how the *edit-compile-link-run-test cycle* works in your programming environment.
- To understand the following basic C++ language features:
  - *comments*
  - *keywords*
  - *string constants*
  - *libraries*, the `#include` *compiler directive*, and *namespaces*
  - The `main` *function* with an `int` *return-type*
  - The `return` *statement*
  - How a simple *output statement* uses the *insertion operator* `<<` and the *manipulator* `endl` to output a string constant
- To understand what is meant by *syntax* and *semantics* in a C++ program.
- To understand what is meant by *programming style*.
- To get some practice using the edit-compile-link-run-test cycle to modify the given sample program, and to create some new programs of your own.

1.2 List of associated files

- `hello.cpp` contains a program to display "Hello, world!" on the screen.

1.3 Overview

Bjarne Stroustrup, inventor of C++

In this Module we begin our examination of the C++ programming language, designed by Bjarne Stroustrup at Bell Laboratories in the 1980's. The language was first standardized in 1998, but it always takes some time for all the different compilers that are available to comply with a new standard. In fact, it often happens that by the time compilance becomes widespread the standard is revised and the cycle begins all over again. In any case, we should never be too surprised to encounter differences between what the C++ standard says should be the case, and our "real world" experience with a particular compiler.

Learn by example, and learn by doing.

Our approach to learning the language, in this Lab Manual, will be to *learn by example* and to *learn by doing*. This means that in each Module that deals explicitly with C++ we will present one or more sample programs and use them to point out new features of the language and new guidelines for programming style. Following each sample program, you will find one or more hands-on activities that require you to work with that program in various ways, and to design and write one or more related programs of your own.

Some important considerations to think about as you begin to program

Learning to program is a complex activity, and a very important part of it involves the reading, and subsequent modification (together with follow-up testing and debugging), of already-written programs. This is how you become familiar with the nuts and bolts of the language. Designing and writing your own programs so that they are correct, easy to read and understand (by others as well as yourself), and present a good *user interface* when they run is a whole other kettle of fish, so to speak, for which good knowledge of the nuts and bolts is necessary but (unfortunately) not sufficient. We will provide what we hope is good advice along the way, but there is no substitute for lots of practice and lots of positive response to constructive criticism of your work, and such criticism should be both external (coming from others) and internal (coming from yourself).

Starting with this Module you must also begin to use your particular programming environment tools, whatever they may be. These will allow you to *edit*, *compile*, *link* and *run* the sample programs provided, as well as those you will write for yourself. As time goes on, you will be learning more about the programming environment and the tools available to you. Becoming proficient in their use will help you increase your *productivity* as a programmer, or (to use the fifty-dollar term) as a *software developer*. See Appendices F to J for more on the programming environment, if necessary.

1.4 Sample Programs

This Module actually contains only one sample program, and a very short program at that, but it is your first, and hence there are quite a number of things in it that you are seeing for the very first time. For that reason, we discuss each new feature in some detail in what follows.

1.4.1 hello.cpp is everyone's first C++ program and simply displays "Hello, world!" on the screen

```
//Filename: hello.cpp
//Purpose:  Displays a "Hello, world!" greeting.
//Author:   P. Scobey
//Date:     2004.08.17

#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, world!" << endl;

    return 0;
}
```

1.4.1.1 What you see for the first time in hello.cpp

- C++ *comments*

C++ *comments* are illustrated by the first four lines of code in the program found in the file `hello.cpp`. Anything to the right of the two forward slashes (`//`) on a line is treated as text for human readers only. Comments do not affect the way the program itself works when it runs.

For the sake of brevity, our sample programs will usually contain only the first two of the four comment lines shown in the above program, which give the name of the file containing the program, and a brief description of what the program is supposed to do. Furthermore, we will even omit the explanatory (but somewhat redundant) labels `Filename:` and `Purpose:`. In longer programs, of course, more comments will be required, both at the beginning of the program and elsewhere, to provide additional information or clarification.

- C++ *keywords*: both *reserved words* and *predefined identifiers*

C++ has certain words that are special to the language. Some are very special, must be used in very special ways, and are called *reserved words* (they are reserved for use in their very special way). The reserved words in `hello.cpp` are: `int`, `using`, `namespace` and `return`.

Reserved words as keywords

A reserved word can *only* be used by a programmer in the way in which C++ requires it to be used. The other kind of special word, which is called a *predefined identifier*, *could* perhaps be used in a way different from the way in which it is normally used in C++ but to do so would only cause confusion and

Predefined identifiers as keywords

doing so should thus be avoided. The predefined identifiers in `hello.cpp` are: `include`, `iostream`, `std`, `main`, `cout`, and `endl`.

C++ is a case-sensitive programming language.

We shall indicate below how each of these reserved words and predefined identifiers is used. It is very important to remember that each reserved word *and* each predefined identifier must be spelled correctly *and* capitalized correctly because C++ is a *case-sensitive* programming language. All reserved words in C++ are spelled using exclusively lowercase letters. Some predefined identifiers use all uppercase (capital) letters, but none of the ones in this program do so.

- The `main` function and the `return` statement

The `main` *function*

Every C++ *program* that you will see for a while will consist of a collection of *functions* that work together to accomplish whatever it is that the program does. Exactly one of those functions must be called `main`, and it is with this function that the program begins execution when it runs.

Function statements

Function body enclosed by braces

Any function, in turn, including `main`, consists of a sequence of *statements* that, when executed, (ideally) perform a single specific task well. Each statement is terminated by a semi-colon (`;`) and the sequence of statements (called the *body* of the function) is enclosed within a pair of *braces* (i.e., "curly brackets") that look like this: `{ }`. This is an oversimplification, of course, but at least in the case of `hello.cpp` it is the task of the `main` function simply to display on the screen the greeting:

```
Hello, world!
```

Function header and function definition

Returning a value of data type `int` *with a* `return` *statement*

In mathematics it is the job of a function to compute, or, as we sometimes say, "return" a single value. Many C++ functions also do this. The kind of value "returned" by a C++ function is indicated by placing the name of a *data type* in front of the name of the function in the *function header* (the first line of the *function definition*). That is the purpose of the `int` in front of `main` in the program of `hello.cpp`: to indicate that `main` will return, in this case, an integer value (think of `int` as short for "integer"). The `main` function in our sample programs will return an `int`[1] value. The second and last statement of the program (`return 0;`) is the one that actually returns the integer value, which is 0 in this case. The value 0, in this context, is generally used to denote "success" (the function has succeeded in performing its task), and if you are wondering to whom or to what this value is actually "returned", it's the operating system, which may or may not be "watching for" a value.

Parentheses required in function header

The parentheses `()` that appear after `main` are important, and they *must* be present.

- The `iostream` C++ *library*, the `#include` *compiler directive*, and *namespaces*

C++ libraries

The C++ programming language makes very extensive use of *libraries*. A C++ *library* often contains a number of related functions. Unlike the functions in a program which are all designed to work together for some higher purpose, the functions in a library are related in a different way: they usually perform

[1] See the last hands-on activity in this Module for an important note on the omission of the `return` statement in `main`.

similar or related tasks within some particular context. A case in point: the `iostream` library, which contains functions for handling input to a program from the keyboard and output from a program to the screen. Like many libraries, the `iostream` library also contains things other than functions. In particular it contains `cout`, which is not a function but a *stream object* (in fact, an *output stream object*). Think of `cout` as representing a "stream of characters" that get sent to the screen, and you insert something into the stream (and hence send it to the screen) with the *insertion operator* `<<`. This is how we send the "Hello, world!" message to the screen in `hello.cpp`, via the statement:

`iostream` *library*

Streams, output streams and `cout`

The insertion operator `<<`

```
cout << "Hello, world!" << endl;
```

For your C++ program to make use of what is in a particular C++ library you must "include" the *header file* corresponding to that particular library. In `hello.cpp` this is accomplished by this line[2] of the program:

```
#include <iostream>
```

This line is an instruction (or *directive*) to the C++ compiler that says: This program is going to use something from the `iostream` library, so before proceeding, go get the descriptions of everything in that library and put them into this program (i.e., "include" the actual text) so that you will recognize whatever is being used when you see it. This is why a line like the one above is called a *compiler directive*, and must not be confused with a *C++ statement*. (Note, in particular, that a compiler directive does *not* end with a terminating semi-colon, unlike a C++ statement.) The compiler always knows where to "go get" the information, i.e., the library header files.

Compiler directives, or to be somewhat more precise, "preprocessor directives"

The line

```
using namespace std;
```

illustrates one of the more recent features to be added to the C++ language before the Standard was approved. We do not discuss this topic—*namespaces*—in detail here. We simply point out that this line will appear in all of our programs, and its purpose is to indicate that the material included by the previous `#include` compiler directives (just one in this case) belongs to the *namespace* called `std` (which is provided in Standard C++).

Namespaces

Programmers (including yourself) may produce other namespaces containing additional features (or replacements for features found in the Standard Library) but this too is a topic for (much) later discussion. Think of it this way: Sometimes many different programmers contribute individual parts of the code for a large program, and sometimes the same names get used for different things. When this potential problem exists (and it always does in a large programming project), use of namespaces can help to keep C++ (and those who program using the language) from getting confused.

[2] Some compilers may (still) require the older (pre-Standard) form <`iostream.h`> instead of <`iostream`> and some may permit either, but this Manual assumes you are working with a compiler that is sufficiently up to date that you do not have to use the `.h` form of *any* header file.

- C++ *string constants*

String constants

In C++ a *string constant* (often just called a *string*) is a sequence of characters enclosed in double quotes, like `"Hello, world!"` in `hello.cpp`. Strings are very important in programming, but for the moment we will use string constants only to permit our programs to display human-readable text on the screen when they run.

- A C++ *output statement* that displays text on the screen

When a C++ program runs, its output is often displayed on the screen by a statement of the form

```
cout << thing_to_be_displayed << endl;
```

Splitting output that extends over more than one line

or sometimes by a single statement that extends over more than one line and has the form

```
cout << first_thing << second_thing
     << third_thing << last_thing    << endl;
```

where, for the moment, the only kind of "thing" we can display is a string constant. Later on, of course, we will be able to display the results of numerical calculations, the contents of memory locations that may contain values of different kinds, and so on.

Even now, though, we may occasionally need to break up a single long string constant that we wish to have displayed on one line into several shorter string constants within a single `cout` statement as shown in the second pattern given above. As you will see in Module 2, you can do this with or without placing the insertion operator `<<` between the successive segments of an extended string constant that extends over more than one line.

Manipulators

`endl` "flushes" output stream and sends a newline character

The term *manipulator* is used to describe `endl`, which you can think of as an abbreviation for "endline". When this manipulator is inserted into the output stream, the output stream is "flushed" (i.e., anything not yet sent to the screen is now sent) and a *newline character* is also entered into the output stream and sent to the screen (which means that any subsequent output will appear on a new line of the screen display).

- A first glimpse at C++ *syntax* and *semantics*

syntax

The term *syntax* refers to the "rules of grammar" that must be followed when constructing a C++ program if that program is to be valid. Thus, getting the syntax right involves such things as spelling all the keywords correctly, putting things in the correct order, and making sure the punctuation is correct.

semantics

The term *semantics* refers to the "meaning" of a *syntactically correct* (i.e., valid) program. With good luck, and good management, our programs will eventually be correct and meaningful (i.e., they will have the meaning we intended them to have, and do what we expect of them). Of course it is possible to have a syntactically correct program that is meaningless, and along the way we may see more of those than we had hoped for.

Compilers are very good at telling us when we have *syntax errors*, which is why such errors are also called *compile-time errors*. Unfortunately, *semantic errors* are often more subtle and often do not show up until we notice errors in a program's output at run-time. Hence semantic errors are also sometimes called *logic errors* or *run-time errors*, and we will have to find those ourselves.

Syntax errors (compile-time errors) and semantic errors (logic or run-time errors)

In the program `hello.cpp` it would be a *syntax error* if we placed the first insertion operator `<<` before `cout` instead of after it in the statement

```
cout << "Hello, world!" << endl;
```

which sends the greeting to the screen. However, the given output statement is *syntactically correct*, and the *semantics* of that statement mean that the string constant is to be sent to the screen and then a newline character is sent, moving the cursor to the next line in the output.

- A first glimpse at C++ *programming style*

Because this is such a short program there is little opportunity for it to show very much in the way of programming style. However, again because it is our first program, it is all new and there are in fact a few things we should take note of.[3]

Programming style

First of all, note the extension on the name of the file containing our program. We shall use `.cpp` to indicate that a file contains a C++ program. This is a very common convention, but it is by no means universal. Thus the choice of extension to denote a C++ program file may be thought of as a matter of style, to some extent, although your programming environment may try its best to force a particular extension upon you.

File extension used for C++ programs will be .cpp

Other very important aspects of programming style include the appropriate use of *vertical spacing* and *horizontal spacing*, as well as *indentation* and *alignment*. The `hello.cpp` program illustrates a number of style conventions which we shall continue to follow, except perhaps for certain situations where we may violate the conventions for reasons which we will explain at the time. Here are those conventions:

1. Put each statement on a separate line.

   One statement per line

2. Use one or more blank lines (this is what we mean by *vertical spacing*) to separate logically distinct parts of a program. In our example these parts consist of the initial comments, the `#include` directive, and the `main` function. Within `main` the output statement and the return-statement are also separated by a blank line, though this is hardly necessary.

   Vertical spacing

3. Place a blank space on each side of an operator such as `<<` (an example of what we mean by *horizontal spacing*).

   Horizontal spacing

4. Always indent a function body with respect to the corresponding function header, and our *indentation level* will be *four* spaces.

   Indentation level

[3] Or, "of which we should take note". A brave editor once complained to Winston Churchill about his occasional ending of a sentence with a preposition. Winston's reply: "This is the type of arrant pedantry up with which I will not put!"

alignment

5. Align (i.e., "line up") the beginning of each statement in a function body. Also, place the braces enclosing a function body on separate lines and align them with the beginning of the function header.

- The overall structure of a C++ program

It is important to get a feeling for what a C++ program "looks like", i.e., its overall structure. Again, because this is such a small program there is much that we don't see, but the order of things that we *do* see will be typical for a while at least. This order is:

1. initial comments
2. one or more compiler directives
3. the "`using namespace std;`" line
4. the `main` function.

1.4.1.2 Additional notes and discussion on hello.cpp

There is a long-standing tradition, which we have followed, that whenever a programmer begins to study a new programming language, the first program that should be examined is one that simply displays "Hello, world!" (or some such greeting or statement) on the screen. Once a beginning programmer has managed to get such a program to execute in the local programming environment, he or she may be considered to be *up and running*!

Commenting as an art form

This program was so short, and its purpose so "obvious", that one could argue that it needed no comments at all. In fact, commenting computer programs in any language is a bit of an art form, and one can always debate how much commenting is required in any given program. Too much commenting can make a program almost as hard to read and understand as too little. The goal should *always* be *program understandabilty*, so whatever level of commenting contributes most to the achievement of that goal is the right one.

Always choose good names.

We will try, as much as possible, to make our programs *self-documenting*. This term usually refers to the practice of choosing good names for the entities in your program and formatting your program well, thus reducing the need for extensive comments.

1.4.1.3 Getting a program to run and getting comfortable with the edit-compile-link-run-test cycle on your system

Once you know how to use your editor to enter a C++ program into a file on your computer, the next thing you need to know is how to get that program to run. This process differs from one system to another, so it is another of the "local" pieces of information that you have to discover and record. Though the details differ, and on many systems two or more steps are often combined into a single command, the conceptual steps are generally the same and consist of the following:

Edit the program, i.e., enter the *source code* of your program into a file with the appropriate name.

Compile the program, i.e., submit the program to your compiler, which will examine it for syntax errors and report any that it finds; if the program is OK, the compiler will produce an intermediate file called an *object file*.

Link the object[4] file, i.e., submit the object file to the *linker*, a program which takes the instructions in your own program and "links" them with whatever else is needed (code for stuff from the C++ libraries that you have "included" in your program, for example) to produce an *executable file*.

Run the executable file, i.e., instruct the operating system to execute your program, at which point your program takes over and does its thing.

Test the program, by entering all test data that you have previously (and carefully) prepared, in those cases where the program requires input (usually the case, but not so for `hello.cpp`), and in any case checking the output from the program to see that it is not only correct but formatted and positioned properly.

Because very few programs compile, link, run and test correctly on the first pass, for all but the simplest programs (such as `hello.cpp`, for example) the above steps will have to be repeated a number of times before a program works correctly. Hence we often refer to the edit-compile-link-run-test *cycle*. The cycle begins again when you go back to the editor to remove one or more bugs that the compilation phase or the testing phase has revealed.

Answer

On your system, how do you get the program in a source code file to compile, link and run?

[4] By the way, it probably wouldn't hurt to mention that the use of the term *object* in this context has nothing at all to do with *object-oriented programming*, whatever that may be!

Are there special things you should know about what to do when you have errors?

Answer

1.4.1.4 Follow-up hands-on activities for hello.cpp

This is your first set of hands-on activities dealing with a sample C++ program, and they are reasonably typical. You begin by making your own copy of the file containing the sample program from wherever the files for this Manual are stored. Then you compile, link and run your copy to see how it performs.

Program "bugging"

After this, you introduce changes (which may or may not be errors) into the program, and then try to re-compile, re-link and re-run, to see what kind of error(s) (if any) you encounter. This is the process of *bugging* a program, and provides experience that will be invaluable later when you see the same kinds of errors "for real" during your program development.

Program modification and program writing

Next, you make one or more copies of the sample program and modify it in various ways. Finally, you design, write, and test one or more new programs of your own, based on what you have learned from dealing with the given sample program. You may choose to check the box at the left as you complete each activity.

□ Activity 1 Copy `hello.cpp` to your workspace and give it the same name. Study the source code in `hello.cpp`. Then compile, link and run the program, and observe the output. Did that work out OK? If so, great! You're on your way!

How to properly "bug" your programs

□ Activity 2 Make another copy of the file `hello.cpp` and call it `hello1.cpp`. Edit `hello1.cpp` and make each of the changes to the program shown below, *one at a time*. After each change, try to compile, link and run the program again. In the blank space provided, record either that the program ran successfully with the same behavior as before, or not, as the case may be, and if not, then describe briefly the behavioral change. Or, record (in your own words) the error message provided by the compiler if the program fails to compile after the given change has been introduced. Don't forget to *undo* the previous change each time before making a new one, since the idea here (and in future similar situations) is first of all to experience the effect of each change independently of any effect caused by other changes.

a. Remove the line containing the compiler directive[5].

____

b. Remove the two forward slashes (`//`) at the beginning of the last comment line.

____

c. Remove the left brace (`{`) before the body of the `main` function.

____

d. Remove the right brace (`}`) after the body of the `main` function.

____

e. Remove the reserved word `int` from the function header.

____

f. Remove the parentheses at the end of the function header.

____

g. Remove the semi-colon (`;`) after `endl`.

____

h. Replace the value 0 by the value 5 in the `return` statement.

____

i. Remove the double quote (`"`) at the beginning of `"Hello, world!"`.

____

j. Remove the double quote (`"`) at the end of `"Hello, world!"`.

____

k. Remove the double quotes at the beginning *and* at the end of the string `"Hello, world!"`.

____

[5] Note that, as in this example, in many activities we do *not* always tell you to remove a specific line by showing you exactly what line to remove, or by referring to a line number. Instead, we often “describe” the line to be removed, in a perhaps feeble attempt to get you to think more about what you are doing.

l. Remove the line that says `namespace std` will be used.

____

m. Again remove the line that says `namespace std` will be used, but this time also change the `include` compiler directive to appear as follows:
`#include <iostream.h>`
and, since this is the last change, leave the contents of the file `hello1.cpp` in this state after you have tried to compile, link and run it.

____

□ Activity 3 Make another copy of the file `hello.cpp` and call it `hello2.cpp`. Then edit `hello2.cpp` and change the program so that it also prints out a second line of text immediately after printing out the line

```
Hello, world!
```

and that second line of text must be this one:

```
Welcome to the world of C++!
```

Also, both lines of text in the output must be indented four spaces from the left margin. Go through the edit-compile-link-run-test cycle until your program is working correctly.

○ Instructor checkpoint 1.1 for evaluating prior work

□ Activity 4 Design and write a C++ program that displays on the screen the following table, by displaying four lines of text. Make sure everything is positioned exactly as shown, with the word Fahrenheit beginning at the left margin. Put your program in a file called `temperature_table.cpp`.

```
Fahrenheit | Celsius
-----------|--------
   32      |    0
  212      |   100
```

○ Instructor checkpoint 1.2 for evaluating prior work

□ Activity 5 This activity is a follow-up to the footnote on page 4. Begin by making another copy of `hello.cpp` and calling it `hello3.cpp`. Then change the `main` function by removing the `return` statement. Compile, link and run the program in `hello3.cpp` to see if it works as before. If it does, your compiler is up to date, and supplies an implicit `return 0;` statement at the end of each `main` function. If you get a warning or error, your compiler is not up to date, and you will need to supply this `return` statement at the end of each program that you write to avoid the warning message.

○ Instructor checkpoint 1.3 for evaluating prior work

Module 2

A first look at program development: top-down design with step-wise refinement

2.1 Objectives

- To appreciate the importance of the *program development process*.
- To learn what is meant by *top-down design with step-wise refinement*.
- To learn what is meant by *pseudocode*.
- To learn some C++ *escape sequences* and how to use them.
- To learn more about output formatting and programming style.

2.2 List of associated files

- `name_address.cpp` displays two names and addresses.

2.3 Overview

Divide and conquer

In this Module we begin our examination of the process of *program design*. There are many *program design methodologies*, but we shall concentrate for now on one called *top-down design with step-wise refinement*. This is quite a mouthful, but in reality it is conceptually quite simple and represents the programming counterpart of a classic approach to problem solving called *divide* *and conquer.* Applying the method well to our program development tasks is, however, a non-trivial task, and one that we need to begin thinking about early in the game and to continue thinking about during the remainder of our programming careers. Hence this Module.

2.4 Sample Programs

2.4.1 name_address.cpp displays two names and addresses and shows how to position output

```
//name_address.cpp
//Displays two names and addresses.

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

int main()
{
    cout << "\nThis program displays two names and addresses, "
        "each having the same format.\nHowever, the format of "
        "each name/address pair is achieved in a different way.\n\n";

    //Display a first name and address
    cout << "  Andrew Williams\n"
        "\t123 Main Avenue\n"
        "\tHalifax, NS\n"
        "\tB3H 3C3\n\n";

    //Display a second name and address
    cout << setw(18) << "Angela Williams\n";
    cout << setw(8)  << "" << "123 Main Avenue\n";
    cout << setw(8)  << "" << "Halifax, NS\n";
    cout << setw(8)  << "" << "B3H 3C3\n";
    cout << endl;
}
```

2.4.1.1 What you see for the first time in name_address.cpp

This second sample program is designed to illustrate additional aspects of C++ formatting and style. You should observe carefully the position of the text on each line of output when the program runs, and then study the source code equally carefully to see how this positioning is achieved. Note in particular that the same effect in the output is not always achieved in the same way in the source code. The new features shown in the program are:

- Use of *automatic string concatenation* in a `cout` statement

Automatic string concatenation

The use of *automatic string concatenation* in the first two `cout` statements avoids the repeated use of an insertion operator `<<` in a long output string that must be broken over two or more lines of code.

- Use of C++ *escape sequences*, in particular `\t` and `\n`

C++ "escape sequences"

The output from the program in `name_address.cpp` when it runs will not necessarily be what you might expect just from examining the source code. In particular, neither of the *two* characters in the expression `\t` nor the two in `\n` will be printed on the screen, even though they appear in the source code as part of a string constant. The reason for this is that each of these two-character sequences forms what is called an *escape sequence* and each has a special meaning in C++.

An escape sequence represents one *character.*

The first thing to realize about these escape sequences is that even though we write each one using *two* characters, conceptually each one represents a *single* character.

The tab character `\t` *and tab stops*

The escape sequence `\t` represents a *tab character*, and when it is inserted into the output stream it is interpreted as a message to "move the cursor to the next *tab stop*". The default *tab stops* on most output screens are set every eight columns. If this is your current setting, then pressing the tab key repeatedly in an editor should cause the cursor to move to columns, 9, 17, 25 and so on, and sending a sequence of `\t` escape sequences from a program to the screen will have the same effect[1] in the output (assuming in each case that the cursor is initially positioned at the beginning of the line).

The newline character `\n`

The escape sequence `\n` represents a *newline character*, and when it is inserted into the output stream it is interpreted as a message to "move the cursor to the beginning of the next line". Thus the use of the newline character in this way often has the same effect as the manipulator `endl`.

Avoid this misconception.

Be very careful, however, to avoid the following misconception: that sending either `\n` or `endl` to `cout` will produce a blank line in the output. This will only be true if the cursor is already at the beginning of a line; otherwise, the effect is just to terminate the current line of output and move the cursor to the beginning of the next line.

- Including library `iomanip`, and using manipulator `setw()` from that library

`setw()` *manipulator from the* `iomanip` *library*

We have already discussed the `endl` manipulator, which is available from the `iostream` library. The `iomanip` library is another of the many standard C++ libraries. It contains many additional manipulators, including `setw()`, which is used to specify the number of spaces in the output to be used for the *next* output item (and *only* for the *very next* output item, which we express by saying that the effect of `setw()` is not *persistent*). Also, by default, the item displayed is right-justified within the given spaces, which means that the displayed item (i.e. the output text, in our case, so far) is placed as far to the right as possible within the spaces allotted for it in the output. The term used for the number of spaces is the *fieldwidth* and the corresponding area where the text is written is called a *field.*

For example, when the program displays the second name on the screen, 18 spaces are allotted. The string to be printed contains 16 characters (15

[1] Don't get confused: A tab character really *does* cause the cursor to *move to the next tab stop*; it does *not* cause the cursor to "move 8 spaces", though that may well be the net effect. The number of spaces actually moved depends on where the next tab stop is located.

characters for the first and last names and the blank space between them, plus one character space for the escape character `\n`). So, the name starts printing in column 3 and finishes in column 18, because of the right-justification.

- The use of the output statement `cout << endl;` by itself

Pay special attention to the distinction described here.

A line like this in a program often produces a blank line in the output. Note that we say "often", not "always". If the cursor is already positioned at the beginning of a line, then execution of this statement will produce a blank line in the output. Otherwise, the statement will simply have the effect of terminating output to the current line and ensuring that any subsequent output starts on the following line. It is a good idea to output the `endl` manipulator at the end of each program, since doing so will ensure that all remaining output (if any) is sent to the screen before the program terminates.

- A useful technique for indenting output lines

How to indent output lines

Note first of all that the *null string*, i.e., the *empty string* (the unique string constant containing no characters) is written using two adjacent double quotes (`""`). And note how the program uses the writing of this null string in 8 spaces to, in effect, indent an output line 8 spaces from the left margin. If 8 spaces is the required indent then perhaps `\t` is a more compact and convenient choice to achieve the same effect. But if an indent of some other "level" (i.e., some other number of spaces) is required, this method is more versatile. Note also, for example, that `setw()` could have been used to indent the second name in the output by two spaces, and doing so would have avoided the need for counting the number of characters in the name. There are a number of subtleties in this paragraph that you would do well to think about, and come back to later when you are doing the activities.

- More programming style features

Program readability and user-friendliness

User interface

There are two very important, but very different, aspects to good programming style: *program readability* and *program user-friendliness.* The first refers to the degree of understandability your source code has when someone is reading your code and trying to figure out what it does and how it does what it does. The second refers to how pleasant a *user interface* the program has when it runs, in other words how easy it is for users to understand and use your program.

Note that the first thing that this sample program does when it runs is this: *It displays a description of itself.* As a rule, this is a good thing for a program to do, and, with rare exceptions, all of our sample programs from now on will do this. In other words, having a program describe its purpose for the user of the program is critical to having a good user interface.

The other side of the coin (the behind-the-scenes side, if you like) is the source code of the program, which in the "real world" is seldom seen by the program user. But those programmers (who may or may not include the original author(s)) who must read and perhaps modify the source code (possibly long after the program was written), are *very* interested in having source code that is easy to understand.

In this particular sample program, note again how we have used vertical spacing (i.e., blank lines) to separate the logically distinct sections of the `main` function, as well as horizontal spacing and vertical alignment to achieve a consistent formatting style.

Our particular choices represent just one programmer's decisions, rather than some universally followed rule, but they are good (and typical) choices to emulate if you want your programs to be readable. Your instructor will provide guidance on the particular aspects of style to be followed in your case.

2.4.1.2 Additional notes and discussion on name_address.cpp

Let's look at the program in `name_address.cpp` from a somewhat different perspective, and take this opportunity to talk for the first time about another very important concept in programming: the notion of *pseudocode* for a program or part of a program, and the fact that a computer program is an *implementation* of an *algorithm* for solving a problem. These terms represent relatively simple ideas, but understanding them thoroughly is critical to success in programming.

algorithm

An *algorithm* is a finite sequence of unambiguous steps which, when they are performed in a prescribed order, will accomplish a particular task. Many problems have algorithms as their solutions, but not all.[2]

pseudocode

The term *pseudocode*, on the other hand, means *short English phrases used to describe the steps that comprise an algorithm, and formatted in such a way as to make the instructions clearer than they would be without such formatting.*

With this in mind, suppose we have the following problem: Display the name and address of both Andrew Williams and Angela Williams on the screen.

Then we could write pseudocode for an algorithm to solve this problem as follows:

```
Display the name and address of Andrew Williams
Display the name and address of Angela Williams
```

A program is an implementation of the algorithmic solution to a problem.

Now we immediately recognize the *program* in `name_address.cpp` as a *solution* to this *problem*, or, as we also say, an *implementation* of the solution expressed by the pseudocode for the algorithm which solves the problem. Notice that we say "an" implementation, not "the" implementation, since there are many other possible implementations. We could have written a program to do exactly the same thing using the Pascal programming language, for example, or FORTRAN, or BASIC. Or, for that matter, we could have taken a marker and scrawled the names and addresses on the screen, since the problem did not specify in detail how the information was to look on the screen, or how it was to get there.

specifications

This last remark brings up the notion of *specifications*, by which we mean *a written description of what a problem is, and what form its solution must take.* If the solution to the problem is to be provided by a computer program, it is often convenient to express the specifications by describing both the nature of the input to the program and what output the program must produce for this given input.

[2] For example, no one yet seems to have discovered an algorithm for achieving world peace.

If some part of the problem specifies exactly how something is to be done in the solution, then those specifications *must* be followed by the *implementor* of the solution, i.e., the writer of the program. But sometimes the implementor must make assumptions about how something is to be done. Even then, of course, the description of *how* it is to be done must be recorded, and then this description becomes a part of the specifications.

Top-level (or level 0) pseudocode

Note that the solution expressed in the pseudocode given above is devoid of low-level details. This is characteristic of *top-level pseudocode*. Rare indeed is the case where there are no other "levels" of pseudocode. This would only be the case if the problem to be solved were simple enough that this "level 0" pseudocode[3] could be translated directly into C++ code. Even in the very simple case of our sample program we can "refine" our pseudocode as follows:

```
Display the name "Andrew Williams" starting at the left margin
Display each line of the address of Andrew Williams,
    and indent each line of the address by 8 spaces
Display the name "Angela Williams" starting at the left margin
Display each line of the address of Angela Williams,
    and indent each line of the address by 8 spaces
```

Top-down design with step-wise refinement

In every case, this process of *step-wise refinement* must continue until we have reached the point where the steps that we are describing in pseudocode can be translated directly into the programming language we are using to write our programs. In our case that programming language will always be C++, of course. We always begin at the "top" by designing the "high-level" steps of the process that will ultimately lead to a solution. We cannot expect to have the solution immediately, and those high-level steps will need to be refined to more detailed steps at the next level, and so on. This approach to programming is called *top-down design with step-wise refinement*, and works very well for programs of small to moderate size. Just what "small to moderate" means is a matter of subjective judgment, but all of our programs for the foreseeable future will fall into this category.

2.4.1.3 Follow-up hands-on activities for name_address.cpp

This is your second set of hands-on activities dealing with a sample C++ program, and, as we predicted they would be, they are similar to those you did when you worked with `hello.cpp` in the previous Module. However, you are not just doing exactly the same things over again. There is, of course, some repetition, but there are many are new things to be learned and practiced as well.

Look in particular at the first activity below. It contains a couple of steps that you will be asked to perform many times as you proceed, though they will henceforth be stated somewhat more succinctly. The point is this: Whenever we ask you, in the future, to make a copy of a file and then compile, link and run it, you should take as given the fact that before compiling, linking and running

[3] The top-level pseudocode is also called "level-0 pseudocode", since programmers quite often start counting at 0.

the program you must study the source code to decide how the program will behave when it runs, choose some test data of your own if none is given, and only *then* compile, link and run the program.

In fact, we will often condense these steps to something like this:

Copy, study, test and then write pseudocode for xxx.cpp.

We will frequently ask you to write out the pseudocode, even in cases where a program is not doing very much, so that you will become comfortable with the relationship between pseudocode and "real" code, in preparation for the time when using pseudocode properly will become a critical part of your programming skills. In this particular case we have, of course, already provided the pseudocode.

The second activity below is also typical, and we will generally condense the instructions in future similar activities to something like this:

Copy xxx.cpp to xxx1.cpp and bug it as follows:

Keep in mind that you should not feel limited in any way by the given activities. As you do them, other activities are sure to suggest themselves and you must take every opportunity to pursue additional efforts to help you clarify the concepts under discussion.

□ Activity 1 Copy `name_address.cpp` to your workspace and give it the same name. Study the source code, and predict what the output will look like when the program runs. Then compile, link and run the program to check your prediction. If the output was not what you predicted, go back and study the source code again until you have reconciled the discrepancy.

□ Activity 2 Make another copy of `name_address.cpp` and call it `name_address1.cpp`. Edit `name_address1.cpp` and make each of the changes to the program shown below, *one at a time*. After each change, try to compile, link and run the program again. In the blank space provided, record either that the program ran successfully with the same behavior as before, or not, as the case may be, and if not, then describe briefly the behavioral change, or describe in your own words the error(s) discovered by the compiler if the altered program fails to compile. And again, don't forget to *undo* the previous change each time before making a new one.

a. Remove the compiler directive which causes the `iomanip` library header file to be included.

   ____

b. Remove the insertion operator `<<` before `setw(18)`.

   ____

c. Remove the insertion operator `<<` after `setw(18)`.

____

d. Remove all instances of the escape sequence `\t`.

____

e. Remove all instances of the escape sequence `\n` in the first `cout` statement.

____

f. Remove all instances of the backslash character `\` in the first `cout` statement.

____

g. Change all instances of the backslash character `\` to the forward slash character `/` in the first `cout` statement.

____

h. Change each of the three fieldwidth values from 8 to 5.

____

i. Change each null string (`""`) to a string constant containing one blank space (`"␣"`) (Note that the symbol ␣ is sometimes used to emphasize the existence of a blank space, usually within a string constant, as shown here.).

____

j. Change each null string to a string constant containing three blank spaces.

____

k. Change each null string to a string constant containing eight blank spaces.

____

l. Change each null string to a string constant containing twelve blank spaces, and since this is the last change leave `name_address1.cpp` in this state when you have finished.

____

Note that in the last "bugging" activity immediately above we again (as we did in the previous Module) explicitly ask you to leave the

program in the state it was in after this particular activity. From now on, however, we won't *explicitly* ask you to do so, but you are to continue as you have been doing at the end of the bugging activities, i.e., leaving the program in its current state after you have inserted and tested the last change or "bug". The reason for this is so that your instructor will have some evidence that you have completed the bugging activities, should he or she wish to make this check.

□ Activity 3 Make another copy of `name_address.cpp` and call it `name_address2.cpp`. Edit `name_address2.cpp` so that in the output display the address lines for each person are indented 5 spaces relative to the left margin (i.e., relative to the beginning of the name).

○ Instructor checkpoint 2.1 for evaluating prior work

Code segment

□ Activity 4 A *code segment* is a few lines of code that may be shown outside the context of any particular program, and is generally used for illustrative purposes. Study the following code segment:

```
cout << "How much\n\twood could\na woodchuck ";
cout << "chuck" << endl << "if a woodchuck\ncould\tchuck wood?";
```

Now predict what the output of this code will look like when it is executed, by entering your predicted output on the lines shown below (assuming standard default tab settings on the output screen, i.e., every 8 spaces starting in column 9). The space between each two vertical bars is to hold one character only and the leftmost such space represents the first character position on the line.

Next, design and write a suitable program to contain this code segment, and then test the program to see if your output prediction was correct. Call the program file `woodchuck.cpp` and make sure your program exhibits all the features of good programming style that we have discussed.

○ Instructor checkpoint 2.2 for evaluating prior work

Special use of `setw()`

□ Activity 5 Repeat the previous activity for the code segment shown below. This time call the program file `cities.cpp`, and again be sure your program exhibits good programming style. Before doing this exercise, it will be useful to know the following additional information about `setw()`: If the specified fieldwidth is not large enough, the width *automatically expands to the size needed* to hold the number of characters being output. Among other things, this means that `setw(0)` or `setw(1)` can be used whenever we want a fieldwidth of exactly the right size.

```
cout << setw(12) << "Halifax"    << setw(6) << "NS" << endl;
cout << setw(12) << "Saint John" << setw(6) << "NB" << endl;
cout << setw(3)  << "Halifax"    << setw(3) << "NS" << endl;
cout << setw(0)  << "Halifax"    << setw(0) << "NS" << endl;
```


○ INSTRUCTOR CHECKPOINT 2.3 FOR EVALUATING PRIOR WORK

□ Activity 6 In order to complete this activity, you need to know a new fact about the backslash character \ which, as you already know, is the first character in the two-character escape sequences `\t` and `\n`.

First, note that if you wish to print out a double quote (") as part of a string constant, then the double quote must be "escaped", i.e., it must be preceded by a \ in the string constant. For example, to print out a line like

```
He said, "How are you?"
```

using a single string constant, that string constant would have to appear in the program like this:

```
"He said, \"How are you?\""
```

The reason that the double quotes enclosing the question need to be "escaped", of course, is so that C++ does not get confused as to which double quotes actually enclose the string constant. Also, the backslash character itself (\) must be "escaped" if you wish to display one in the output. That is, to have \ appear in the output, \\ must appear in the string constant at the position where you want the \ to appear.

More escape sequences: \\ *and* \"

So, design and write a C++ program that produces the six lines of output shown below. The purpose of the first line of output, here as well as in the actual output from the program, is just so that you can see where everything is located. Call your program file `escape_sequences.cpp`.

```
123456789012345678901234567890123456789012345678901234567890
    So far we have seen these four C++ "escape sequences":
             \t         tab character
             \n         newline character
             \"         double quote character
             \\         backslash character
```

○ INSTRUCTOR CHECKPOINT 2.4 FOR EVALUATING PRIOR WORK

Module 3

Displaying both text and numerical output

3.1 Objectives

- To undertand how C++ handles *integer* and *floating point* (*real number*) values, i.e., to learn about the C++ `int` and `double` data types.
- To learn what a *variable* is, how to *declare* numerical variables (variables of data types `int` and `double`), how to *assign* a value to such a variable, how to *initialize* such a variable at the time of its declaration, and how to display the value such a variable contains.
- To learn how to format numerical output, and how to combine it with text output.

3.2 List of associated files

- `integers_text.cpp` displays output containing both integers and text.
- `reals_text.cpp` displays output containing both real numbers and text.
- `errors.cpp` contains both syntax and output formatting errors.

3.3 Overview

In this Module we look at programs that combine text and numerical output. The C++ programming language provides different kinds of numbers—*integers* and *real numbers*, for example—so we also begin our study of the available numerical *data types*.

3.4 Sample Programs

3.4.1 integers_text.cpp displays output containing both integers and text

```
//integers_text.cpp
//Illustrates text and integer output combined, as well as
//declaration of, and assignment to, an integer variable
//(a variable of data type "int").

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

int main()
{
    cout << "\nThis program displays some combinations "
        "of integer values with text.\n\n";

    cout << "There are " << 7  << " days in a week."   << endl;
    cout << "There are " << 12 << " months in a year." << endl;

    int numberOfDays;   //Reserves a memory location
    numberOfDays = 365; //Assigns 365 to that location
    cout << "There are (usually) " << numberOfDays
         << " days in a year.\n\n";

    cout << setw(10) << ""
        "This line starts in column " << 11 << ".\n";

    cout << "\nHere is a list of " << 4 << " items:\n"
         << setw(12) << "computers" << setw(6) << 80  << endl
         << setw(12) << "monitors"  << setw(6) << 80  << endl
         << setw(12) << "mice"      << setw(6) << 120 << endl
         << setw(12) << "printers"  << setw(6) << 6   << endl << endl;

    cout.setf(ios::left, ios::adjustfield);
    cout << "Here is the same list of " << 4 << " items:\n"
         << setw(12) << "computers" << setw(6) << 80  << "<<\n"
         << setw(12) << "monitors"  << setw(6) << 80  << "<<\n"
         << setw(12) << "mice"      << setw(6) << 120 << "<<\n"
         << setw(12) << "printers"  << setw(6) << 6   << "<<\n";

    cout << endl;
}
```

3.4.1.1 What you see for the first time in integers_text.cpp

- The output to the screen, via `cout`, of integer *literal values* (i.e., actual values of integer constants, such as 7, 12, 11, and so on) in the *default output format* for integer values, which is simply to display the given integer with no spaces automatically inserted either before or after it

  Integer literal values

- The *declaration* of an *integer variable* (a variable of data type `int`), as seen in the following line (and note the capitalization style):

  Declaration of an integer variable

  ```
  int numberOfDays;
  ```

- An *assignment* to an integer variable, and the first use of the *assignment operator* (`=`), as seen in the line:

  Assignment (=)

  ```
  numberOfDays = 365;
  ```

- The output of a value in an integer variable (namely, the value in the `int` variable `numberOfDays`) using the *default output format*, as seen in the executable statement:

  Displaying the value of a variable

  ```
  cout << "There are (usually) " << numberOfDays
       << " days in a year.\n\n";
  ```

- Display of mixed integer and text output (as in the output statement above), including "tabular" output (i.e., output in the form of a table), which you will see when you run the program

  Output of text and integer values

- The use of the manipulator `setw()` with integer output to override the default output format for integer values; as with string constants, `setw()` when used with integers causes an output value to be right-justified in the output field

  Programmer-formatted output using `setw()`

- The use of the statement `cout.setf(ios::left, ios::adjustfield)` to alter the *default behavior* of `setw()` (which is right-justification of the output values) to left-justification of the output values[1]

3.4.1.2 Additional notes and discussion on integers_text.cpp

Note that even in a short program like this one, which is used only to show how certain kinds of output look when displayed on the screen, we provide a program description as part of the output. This is a very good habit to develop. It is also useful, of course, to think about the corresponding pseudocode for this program, and you are asked do so in the hands-on activities. For practice, you must continue to think about, and write down, the pseudocode of sample programs you encounter, and there will be lots of opportunity to do so. In the sample programs we will continue to be careful to indicate the logical chunks of code in the program by proper formatting, in particular the use of vertical spacing to separate them.

[1] Note that this is only one way to accomplish this task. You will see different approaches to solving the same problem in different textbooks.

3.4.1.3 Follow-up hands-on activities for integers_text.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `integers_text.cpp`.

□ Activity 2 Copy `integers_text.cpp` to `integers_text1.cpp` and bug it as follows:

a. Remove the line that declares the integer variable `numberOfDays`.

____

b. Remove the line that assigns a value to `numberOfDays`.

____

c. Change each instance of `setw(6)` to `setw(1)`.

____

□ Activity 3 Copy `integers_text.cpp` to `integers_text2.cpp` and modify the copy so that only the first of the two tables of computer equipment items is displayed, and in the output the item names are lined up left-justified in column 4 and the numbers are lined up left-justified in column 16. All other program output remains the same.

□ Activity 4 First, predict the output of the code shown below by entering what you believe to be the exact output in the spaces provided following the code. Then write a suitable program that includes this code and test your prediction. Put the program in file called `fruit.cpp`.

```
cout << setw(10) << "apples" << setw(4) << 6;
cout << "\n   oranges" << setw(2) << "" << 12 << endl;
cout << "\nTotal items " << 18;
```

○ Instructor checkpoint 3.1 for evaluating prior work

3.4.2 reals_text.cpp displays output containing both real numbers and text

```
//reals_text.cpp
//Illustrates real number (floating point) and text output
//combined, as well as declaration of, and assignment to, a
//real number variable (a variable of data type "double").

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

int main()
{
    cout << "\nThis program displays some floating point values, with "
        "and without text.\nStudy the output and the source code to see "
        "how such values are displayed.\n\n";

    cout << 2.345     << endl
         << 2.3456    << endl
         << 2.34567   << endl
         << 2.345678  << endl
         << 11111.1   << endl
         << 111111.1  << endl
         << 1111111.1 << endl << endl;

    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);

    cout << setw(8) << setprecision(1) << 2.3456 << endl;
    cout << setw(1) << setprecision(2) << 2.3456 << endl;
    cout << setw(1) << setprecision(8) << 2.3456 << endl;
    cout << setw(1) << setprecision(8) << 1.23456789 << endl;
    cout << setw(1) << setprecision(8) << 123456789.123456789 << endl;
    cout << endl;

    double price1;  //<-- This is a "declaration".
    price1 = 2.95;  //<-- This is an "assignment".
    double price2 = 1099.50;  //<-- This is an "initialization".
    cout << "The first price is $"
         << setw(1) << setprecision(2) << price1 << ".\n"
         << "The second price is $"
         << setw(1) << setprecision(2) << price2 << ".\n";
    cout << endl;
}
```

3.4.2.1 What you see for the first time in reals_text.cpp

- The output, via `cout`, of floating point (real number) *literal values* (2.345, ..., 11111.1, and so on) using the *default output format* for real numbers

  Real number (floating point) literal values

- The use of `cout.setf` to set the `fixed` and `showpoint` *flags* so that subsequent floating point output will be displayed in *fixed point format*, and so that the decimal point is always shown, even if the value is an integer

  Forcing fixed point notation for real values

  The two C++ statements using `cout.setf` that you see in this program have the effect of causing all subsequent floating point values to be displayed using *fixed point notation* (see discussion in the following subsection). Unless you are prepared to undertake a more detailed study of C++ formatting flags (not necessary at this point, certainly), it's probably best just to think of these statements as "magic lines" which you insert in your program at some point (perhaps the beginning) before you wish to start displaying real values using fixed point notation. After these lines have executed, the `setprecision()` manipulator determines the number of digits to appear after the decimal in the numbers displayed.

The `setprecision()` *manipulator*

- The use of the `setw()` manipulator (whose purpose is the same as before) and the `setprecision()` manipulator with floating point output to indicate the number of digits to be placed after the decimal point in the output

  Persistence of `setprecision()`

  We should make the point here that, unlike the effect of `setw()`, which only applies to the *immediately following* item in the output stream, the effect of `setprecision()` is *persistent*. That is, any given instance of `setprecision()` applies to all subsequently displayed real numbers, until another `setprecision()` is inserted into the output stream.

Variables for real numbers

- The declaration of, and assignment of a value to, a *floating point* (or *real*) variable (a variable of data type `double`)

Variable initialization vs. variable assignment

- The *initialization* of a variable, i.e., giving a variable a value at the same time that it is declared (Note that *initialization* should not be confused with *assignment*; initialization has the same net effect as a declaration *followed by* an assignment, but is conceptually distinct.)

Mixed output of text and real (floating point) numbers

- The output of mixed floating point numbers and text, including numbers that represent dollars and cents, and numbers in *scientific notation* (see below)

3.4.2.2 Additional notes and discussion on reals_text.cpp

There are many other manipulators and many other ways of using `setf()` to set flags that indicate alternate ways of formatting output. Some of these you may encounter later on, but for the moment the ones you see here are all that you need.

You should be familiar with *scientific notation* from your previous experience with mathematics, but we shall give you a quick reminder here, mainly to point out the distinction between the two terms *fixed point notation* and *scientific notation* for real (floating point) numbers.

Our "usual" way of writing numbers has the more formal name of *fixed point notation*. So, for example we might write a number like `123.45` in the "usual" way (the fixed point notation way), while the same number written in the version of scientific notation used by C++ would appear as `1.2345e02`.

Scientific notation always has one digit to the left of the decimal point and (usually) a default maximum of five digits to the right of the decimal point in C++. The two digits following the `e` are the power of 10 (the exponent of 10) which must be multiplied by the value preceding the `e` to give the actual number. Check your local documentation for any additional information you need.

3.4.2.3 Follow-up hands-on activities for reals_text.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `reals_text.cpp`.

□ Activity 2 Copy `reals_text.cpp` to `reals_text1.cpp` and bug it as follows:

a. Insert a semi-colon at the end of line 16.

____

b. In the second `cout` statement, remove all instances of `<< endl` except the last.

Remember that you want to undo this before moving on, so try to find a way that's easily undone.

____

c. Remove the first `cout.setf` statement.

____

d. Remove the second `cout.setf` statement.

____

e. Remove both `cout.setf` statements.

____

□ Activity 3 Copy `reals_text.cpp` to `reals_text2.cpp` and modify the copy so that in the output the first seven numbers displayed are all on one line, with two spaces separating each two numbers, and the next five numbers are also on one line, with the first number starting at the left margin, two spaces between each two numbers, and each number having five places after the decimal.

□ Activity 4 Design and write a C++ program that produces the output shown below, and in which every numerical value is output as a *literal value*, *not* as part of a *string constant*. Both lines must be indented *four spaces* from the left margin in the output, and *each line must be indented using a different method.* Put your program in a file called `costs.cpp`.

```
If 1 item costs $3.95, then 12 items will cost $47.40.
I need six (6) programmers for one-and-a-half (1.5) months.
```

○ INSTRUCTOR CHECKPOINT 3.2 FOR EVALUATING PRIOR WORK

3.4.3 errors.cpp contains both syntax errors and output formatting errors for you to fix

```
//errors.cpp
//Illustrates some C++ syntax and output formatting errors.

#include <iostream>

int main()
{
    cout << "This is "
    cout << "still the first "
    cout << "line of output.\n";

    cout << "And this "
         << "is the second "
         << "line of output.\n";

    cout << "First value\n";
         << "is " << 6
    cout << "Second value "
         << "is " << 2.35;

    cout << endl;
}
```

3.4.3.1 What you see for the first time in errors.cpp

- A C++ "program" that does not compile, because it contains deliberately-inserted syntax errors (so you could say that it really isn't a program, yet)
- C++ code containing deliberately-inserted formatting errors that you must discover and correct (along with the syntax errors alluded to above)

3.4.3.2 Additional notes and discussion on errors.cpp

You are by now familiar with the process of "bugging" the sample programs, but this is your first encounter with a sample "program" that is not actually working to begin with (a "pre-bugged" program, you might say).

Now is not a bad time to remind you that the better you become at spotting errors in source code on the page or on the screen, the more time you will save during program development. We must all strive, of course, to produce code without errors in the first place, but knowing the impossibility of achieving this goal in the real world means that we also need to continually sharpen our bug-detecting skills as well.

3.4.3.3 Follow-up hands-on activities for errors.cpp

□ Activity 1 Copy `errors.cpp` to `no_errors.cpp` and study the code. Try to find and correct all of the syntax and formatting errors before attempting to compile the program. An important skill to develop is the ability to read C++ code and spot errors of various kinds. Being able to do so can save you enormous amounts of time in the development cycle, when you are doing this for your own code. The output from `no_errors.cpp`, when you are finished, is supposed to look like this:

```
This is still the first line of output.
And this is the second line of output.

First value is 6.
Second value is 2.35.
```

So, compile, link, run and test the program until its performance is consistent with its name.

○ Instructor checkpoint 3.3 for evaluating prior work

Module 4

Reading numbers and characters from the keyboard

4.1 Objectives

- To learn about the `char` data type, which is also considered to be one of the C++ *integral data types*.
- To understand how `cin` from the `iostream` library is used to read both numerical values and character values from the keyboard, and store them in variables of the appropriate data type.

Numerical values can be integer or real (floating point).

- To understand what *whitespace* is.
- To understand how and why whitespace is (usually) used to separate data values in an input stream.
- To understand how whitespace itself can be read in, if necessary, using `cin.get()`.
- To understand how you can arrange for a program to ignore certain parts of the data in an input stream by using `cin.ignore()`.
- To understand the importance of *prompts* to inform the user of what input is expected when a program is reading data from the keyboard.
- To understand how to incorporate user prompts into your programs.
- To understand why you should, and how you can, make your program pause and wait for a user to press the Enter key before continuing.

- To become aware of some of the many pitfalls that lie in wait for the programmer (as well as the user[1]) when dealing with program input.

4.2 List of associated files

- `simple_io.cpp` illustrates simple I/O (as input/output is often abbreviated) of integer, real and character values, and includes a prompt to the user for the input of each value.
- `pausing.cpp` illustrates how to create a pause in a program to permit the user to read descriptive material, instructions or displayed output.
- `whitespace_ignore.cpp` illustrates how whitespace characters in the input stream can be read, and how "unwanted" characters in the input stream can be ignored.
- `test_io.cpp` provides further practice with keyboard input.

4.3 Overview

Every program seen prior to this Module has displayed output on the screen, but none has required input, either from the keyboard or from any other source. In this Module you will deal with programs that read input from the keyboard. Each input value will be either a single number (an integer or a real) or a single character, since we do not know enough at this stage to have our programs read in string[2] input.

4.4 Sample Programs

The sample programs of this Module will show you how a C++ program reads input values from the keyboard into a program and stores them in variables. This is a very straightforward process in the simplest cases, particularly if only one value is being read at a time.

On the other hand, as soon as the input consists of multiple values, possibly of different data types, separated by whitespace (or not), the situation becomes more complicated and various subtleties start to intrude. This is particularly true if the whitespace itself must be read or accounted for explicitly.

[1] We shall generally use the term *programmer* to refer, not surprisingly, to the person who writes a program, while the term *user* will refer to the person who runs the finshed program and "uses" it to do whatever it does. Of course, the programmer must from time to time take on the role of user as well, at least during program development and testing.

[2] Remember that a *string* is a sequence of characters considered as a single entity. Output of *strings*, or at least *string constants*, is easy (we have been doing it all along), but input to *string variables* (variables having the `string` data type) is another story, to come later.

Experience shows that input/output is one of the more troubling topics for beginning C++ programmers to master. Take advantage of the opportunities provided by this Module to start your journey down the road to mastery.

4.4.1 simple_io.cpp illustrates how to prompt a user for input and how to input and output simple values

```
//simple_io.cpp
//Illustrates output of user prompts, with corresponding input
//and output of integer, real and character values.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program illustrates the input and output of "
        "integer (int), real (double),\nand character (char) values. "
        "In each case, note the cursor position when the\nprogram is "
        "waiting for input.\n";

    //Input, and then output, a single integer value.
    //Value is entered immediately following the prompt, with a single
    //space between the end of the prompt and the start of the value.
    int someInteger;
    cout << "\nEnter an integer value here: ";
    cin >> someInteger;
    cout << "The integer you entered was " << someInteger << ".\n";

    //Input, and then output, a single floating point (real) value.
    //Value is entered at the start of the line following the prompt.
    double someReal;
    cout << "\nEnter a real number value here:\n";
    cin >> someReal;
    cout << "The real number you entered was " << someReal << ".\n";

    //Input, and then output, two character values. Tab characters
    //are used to line up the entry positions of the two input values.
    char firstChar, secondChar;
    cout << "\nEnter a character value here:\t\t";
    cin >> firstChar;
    cout << "Enter another character value here:\t";
    cin >> secondChar;
    cout << "The first character entered was " << firstChar << ".\n";
    cout << "The second character entered was " << secondChar << ".\n";
    cout << endl;
}
```

4.4.1.1 What you see for the first time in simple_io.cpp

Reading values from the keyboard

- A second method of giving a value to a variable, namely "reading" a value *from* the keyboard *into* the variable in memory (in addition to the method of "assigning" a value, which we saw earlier)

Declaration of a character variable

- The declaration of a variable of `char` data type

`cin` *for input*

- The use of `cin` from the `iostream` library to "input" (i.e., to "read") values of different data types (`int`, `double`, `char`) from the keyboard, via the *extraction operator* `>>`, into variables of the appropriate data type.

User prompts

- The use of a *prompt* to tell the user what kind of data values the program is expecting for input
- The declaration of more than one variable in a single declaration statement (`firstChar` and `secondChar`)

Positioning the cursor

- Techniques for positioning the cursor on different lines, or at varying positions on the input line, before data entry

Variables declared close to location where used

- The declaration of variables at different places inside the `main` function (with the rule of thumb being to declare each variable as close as possible to where it is first used)

4.4.1.2 Additional notes and discussion on simple_io.cpp

How `cin` *works*

It is very important to understand the default behavior of `cin` whenever your program is attempting to read either a number (integer or real), *or* a character. First, *leading whitespace* is skipped. (The term *whitespace* means *blank spaces*, *tabs*, or *newline characters*.) Then characters are read as long as those characters continue to make sense as part of a single value of the type that it's looking for. When reading stops, the next character in the input stream does not make sense as part of that value, and that character will be the first one read the next time any value is read from the input stream. Often a whitespace character will be the one that stops the reading, but not always.

If you understood the previous description, then congratulations and you know how `cin` works. If you didn't, then you may or may not know how it works, but careful attention to detail when you study each of the sample programs in this Module will help to fill any gaps in your understanding.

Wrong input does not necessarily cause a program to "crash".

It is also very important to know that wrong input does *not* usually cause a C++ program to *crash*[3]. It simply causes the input stream to "shut down" and refuse to accept any more input (perhaps only for the time being, as you will see later). The remainder of the program will continue to execute, but perhaps without critical data that hasn't been read in. So, clearly, bad input is a potentially serious problem.

[3] A program is usually said to *crash* when it emits one or more error messages to the screen display, stops working, and returns control to the operating system

Note that when this program runs, the cursor is at a different positions on the input line for different values input. Study the source code to see how this is done. The question of cursor positioning for input is a matter of individual taste and programming style.

4.4.1.3 Follow-up hands-on activities for simple_io.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `simple_io.cpp`.

□ Activity 2 Copy `simple_io.cpp` to `simple_io1.cpp` and bug it as follows:

a. Replace the extraction operator in the statement `cin >> someInteger;` with an insertion operator.

____

b. Remove the statement that reads in a real number.

____

c. Initialize `firstChar` to the value `'a'` and `secondChar` to the value `'b'`.

____

□ Activity 3 Copy `simple_io.cpp` to `simple_io2.cpp` and modify the copy so that when the program runs, every input value is entered on the line *below* the line containing the prompt for that value.

□ Activity 4 Copy `simple_io.cpp` to `simple_io3.cpp` and modify the copy so that when the program runs, every input value is entered on the *same* line as the line containing the prompt for that value, with one space between the end of the prompt and the beginning of the entered value.

□ Activity 5 Copy `simple_io.cpp` to `simple_io4.cpp` and modify the copy so that when the program runs, every input value is entered on the same line as the line containing the prompt for that value, and all values are input beginning in column 41. (Assume that your screen is set up for "normal" tab spacing, with a tab stop every 8 columns beginning in column 9.)

□ Activity 6 Copy `simple_io.cpp` to `simple_io5.cpp` and remove from the copied program all code that displays the program description and the prompts when the program runs. Now compile, link and run the program. With luck, you will need no further convincing of the need, in the output, for both a program description and appropriate prompts in every program you write.

○ Instructor checkpoint 4.1 for evaluating prior work

4.4.2 pausing.cpp illustrates how to make a program pause and wait for the user to press the Enter key before continuing

```
//pausing.cpp
//Illustrates how to make a program pause and wait for the user to press
//Enter before continuing, an often useful "user interface" feature.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program illustrates how to create a \"pause\" in "
        "the output of a program,\nso that a user may read the preceding "
        "output before continuing. The user then\ncontinues by pressing "
        "the Enter key.\n";

    //The following code causes the program to pause and wait
    //for the user to press the Enter key before continuing.
    //BUT ONLY IF INPUT STREAM cin IS EMPTY WHEN THE CODE EXECUTES
    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’); //1

    //The following three code sections are identical to those in
    //simple_io.cpp, except for the addition of three instances of
    //cin.ignore(80, ’\n’);
    int someInteger;
    cout << "\nEnter an integer value here: ";
    cin >> someInteger;   cin.ignore(80, ’\n’); //2
    //Study carefully the above useful C++ "idiom" for reading
    //an integer and ignoring whatever else in on the same line.
    cout << "The integer you entered was " << someInteger << ".\n";

    double someReal;
    cout << "\nEnter a real number value here:\n";
    cin >> someReal;  cin.ignore(80, ’\n’); //3
    cout << "The real number you entered was " << someReal << ".\n";

    char firstChar, secondChar;
    cout << "\nEnter a character value here: \t\t";
    cin >> firstChar;  cin.ignore(80, ’\n’); //4
    cout << "Enter another character value here: \t";
    cin >> secondChar; cin.ignore(80, ’\n’); //5
    cout << "The first character entered was " << firstChar << ".\n";
    cout << "The second character entered was " << secondChar << ".\n";
    cout << endl;

    //A pause at the end of the program
    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’); //6
}
```

4.4.2.1 What you see for the first time in pausing.cpp

- A technique for causing a program to pause and wait for the user to press the Enter key before continuing (to let the user read something on the screen, for example) *Pausing a program*
- The use of the statement `cin.ignore(80, '\n');` to ignore, essentially, "anything left on the current line of input", or, equivalently, to "clear" the input line *Clearing the input line*

4.4.2.2 Additional notes and discussion on pausing.cpp

There are many differernt potential uses for a call to `cin.ignore()`, with different values of the parameters, and we will come back to this question when we look at the sample program of the following section. In this sample program, however, we make very specific use of a call with very specific parameter values: `cin.ignore(80, '\n')`. The first parameter is `80`, which is the maximum number of characters from the input stream that might be ignored by this call; the second parameter is `'\n'`, which will be the *last* character ignored if it appears among the next 80 characters. Because whenever the user presses the Enter key, a newline character (`'\n'`) is entered into the input stream `cin`, the net effect of this combination of parameters is this: Whenever `cin.ignore(80, '\n')` is called, everything upto *and including* the next `'\n'` will be ignored; or, the next 80 characters will be ignored, whichever comes first. Since users do not normally enter 80 characters or more before pressing the Enter key, what normally happens is the first option, and this statement causes the input stream to be "cleared" of all remaining characters, *including the newline.* This is extremely important in the context of causing a program to "pause", since the code which this program uses to do the pausing relies upon the input stream being empty when that code executes.

You may wish to place the call to `cin.iginore(80, '\n')` on the same line as the input statement with which it is associated, to emphasize that association. This, of course, is a mild departure from the usual style rule of having only one executable statement per line of code. When programmers use a particular construct like this consistently in a given context, the particular construct is sometimes referred to as an *idiom*[4].

`cin.ignore()` *"idiom"*

Also, it is not a bad idea to "officially" terminate any prompt which asks for a value to be input on the same line as the prompt itself. We would do this by placing a `cout << endl;` statement after the `cin` that reads the value. Doing this for the first input value in `pausing.cpp` would mean that our "idiom" in that case would be revised to appear as follows:

```
cin >> someInteger;  cin.ignore(80, '\n');  cout << endl; //2
```

This is often not necessary, but perhaps conceptually we should be happier using the idiom in this way, since doing so means we have dealt completely with both input and output *in the current (local) context.* That is,

[4] Not to be confused with *idiot*, which is something else altogether, though some would say the two are not entirely unrelated.

- We have finished reading the data value.
- We have cleared the rest of the input line (officially ignored any remaining data on the input line containing the value that we just read).
- We have terminated the preceding output line containing the prompt.

If several values are to be input, involving several prompts, we may prefer to wait till all values have been read before performing the "ignore action" and sending the terminating `endl`.

Problem due to extra newline character in the input stream

Most of the time, the only problem you have to watch out for is that of having an extra newline character left over in the input stream when, at some later time, you begin to read a character. This is a subtle problem, and many a novice programmer has come to grief as a result of missing this point. Using the "complete" form of the above idiom consistently will avoid the problem, but will also make your code somewhat more cluttered than it needs to be. Study the sample programs as you proceed. We do not always use the idiom, and sometimes you will see the idiom where it isn't needed, so it's a useful exercise to stop and ask yourself whenever you see it whether it is really necessary. A certain amount of experimentation will prove quite helpful.

4.4.2.3 Follow-up hands-on activities for pausing.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `pausing.cpp`.

□ Activity 2 Copy `pausing.cpp` to `pausing1.cpp` and bug it as follows:

a. Run the program as is, and each time the program pauses and asks you to press Enter to continue, enter these characters before pressing Enter: `abcdef`.

____

b. Repeat the previous exercise, but before doing so, replace each instance of 80 in the source code with the value 4, and produce a revised executable.

____

□ Activity 3 Copy `pausing.cpp` to `pausing2.cpp`. Modify the program so that it reads in and displays a second integer (two integers in all) and a second real number (two real numbers in all). It must display both integers in the place where it now displays one, *and then pause.* And it must exhibit analogous display and pausing behavior for the two real numbers as well.

○ Instructor checkpoint 4.2 for evaluating prior work

4.4.3 whitespace_ignore.cpp illustrates how to read whitespace and how to ignore unwanted input

```
//whitespace_ignore.cpp
//Illustrates more C++ input, this time including cin.get() to read
//whitespace and cin.ignore() to skip over "unwanted" input.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program is a further lesson on input, illustrating "
        "both how to read\nwhitespace, and how to skip over (ignore) "
        "unwanted values in the input\nstream. Read the instructions "
        "for input carefully, enter values carefully\naccording to those "
        "instructions, and then study both the resulting output\nand the "
        "source code equally carefully to reconcile them with the given "
        "input.\n\nHere are the input instructions:\n"
        "1. Enter any amount of whitespace, followed by an integer.\n"
        "2. Enter anything you like on the rest of the line, "
        "then press Enter.\n"
        "3. Enter any amount of whitespace, followed by an integer.\n"
        "4. Now enter up to 20 characters. If fewer than 20 are\n"
        "   entered, make the last one an exclamation mark (!).\n"
        "5. Enter any amount of whitespace, followed by an integer.\n"
        "6. Finally, enter at least three more characters, "
        "then press Enter.\n\n"
        "Start entering data on the following line:\n";

    int i1, i2, i3;
    char c1, c2, c3;

    cin >> i1;  cin.ignore(80, '\n');
    cin >> i2;  cin.ignore(20, '!');
    cin >> i3;
    cin >> c1;
    cin.get(c2); //This statement will read a whitespace character.
    cin.get(c3); //And so will this one.

    cout << "\nHere are the 3 integers and 3 characters read in:\n"
         << "i1: " << i1 << "\ni2: " << i2 << "\ni3: " << i3 << "\n"
         << "c1: " << c1 << "<<\n"
         << "c2: " << c2 << "<<\n"
         << "c3: " << c3 << "<<\n";
    cout << endl;
}
```

4.4.3.1 What you see for the first time in whitespace_ignore.cpp

cin.get()

- The use of `cin.get()` to read in a single whitespace character (blank space, tab, or newline character)

cin.ignore()

- The use of `cin.ignore()` to ignore certain characters in the input stream

  For example, the statement `cin.ignore(12, 'a');` causes *either* the next 12 characters in the input stream to be ignored, *or* all characters up to and *including* the *first* occurrence of the character 'a' if it appears *before* 12 characters have been counted.

4.4.3.2 Additional notes and discussion on whitespace_ignore.cpp

Be prepared to spend a little extra time studying and experimenting with this program, even beyond the hands-on activities given below. If you have difficulty predicting the output from the given input data sets, continue experimenting by making up your own data sets and predicting the output from them.

You might well ask, "When would I actually use a call to `cin.ignore()` like the one above?" In fact, probably not very often when entering data from the keyboard. But in the real world, most program data comes from files, and it may well be the case that only some of the input in a file is actually of interest. In that situation, because the `ignore()` function works with files of text as well, its use may be very helpful, since it is better to simply ignore input, rather than having to go to the trouble of reading it in and then ignoring it.

So, we can become familiar with the `ignore()` function in the context of keyboard input, and be ready to use it in a precisely analogous way, if required, when reading data from files of text. And, as you can see from studying the sample program `pausing.cpp` in Module 3, a particular form of this function (`cin.ignore(80, '\n')`) is very useful when we want to have a program pause during its display of output.

4.4.3.3 Follow-up hands-on activities for whitespace_ignore.cpp

□ Activity 1 Copy, study, test and write pseudocode for `whitespace_ignore.cpp`. Use at least the data sets shown below when testing the program, and enter your predicted output from each data set in the spaces following the data *before* running the program. These data sets are meant to be entered as shown, with the Enter key pressed at the end of each line of input.

a. First data set:

```
123
456!789a b
```

i1 = _____ i2 = _____ i3 = _____ c1 = __ c2 = __ c3 = __

b. Second data set:

```
12Buckle my shoe!
34Shut the door!56And lock it!
```

`i1 = ____  i2 = ____  i3 = ____  c1 = __  c2 = __  c3 = __`

c. Third data set:

```
6.358 This of course is a real number, is it not?
4. Most of the time, 26. is too.
```

`i1 = ____  i2 = ____  i3 = ____  c1 = __  c2 = __  c3 = __`

d. Fourth data set:

```
1
2!3!
And this is the third line of data!
```

`i1 = ____  i2 = ____  i3 = ____  c1 = __  c2 = __  c3 = __`

□ Activity 2 Copy `whitespace_ignore.cpp` to `whitespace_ignore1.cpp` and bug it as follows:

a. Replace the statement `cin.get(c2);` with `cin.get(i2);`.

____

b. Replace `(80, '\n')` with `(80, '/n')` in line 31.

____

c. Replace `(20, '!')` with `(20, 6)` in line 32.

____

○ Instructor checkpoint 4.3 for evaluating prior work

4.4.4 test_io.cpp provides further practice with user input from the keyboard

```
//test_io.cpp
//Illustrates more integer, character and real input,
//including input of whitespace characters.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program is designed to help you experiment with "
        "C++ input. When entering\ndata, use varying amounts and kinds "
        "of whitespace to separate values, just to\nsee what happens.\n\n"
        "Here are the input instructions:\n"
        "1. Enter three integers, followed by three characters, "
        "then three real numbers.\n"
        "2. Now press the Enter key.\n"
        "3. Next, once again, enter another three integers, followed "
        "by another three\n   characters, and finally another three "
        "real numbers.\n"
        "4. Finally, press the Enter key once more.\n\n"
        "Start entering input on the following line:\n";

    int i1, i2, i3, i4, i5, i6;
    char c1, c2, c3, c4, c5, c6;
    double r1, r2, r3, r4, r5, r6;

    cin >> i1 >> i2 >> i3;
    cin >> c1 >> c2 >> c3;
    cin >> r1 >> r2 >> r3;
    cin >> i4 >> i5 >> i6;
    cin.get(c4);
    cin.get(c5);
    cin.get(c6);
    cin >> r4 >> r5 >> r6;

    cout << "\nHere are the eighteen values read in, with a "
        "\"<<\" marker\nat the end of each output line:\n";
    cout << "i1: " << i1
         << "\t\t\ti2: " << i2
         << "\t\t\ti3: " << i3 << "<<";
    cout << "\nc1: " << c1 << "<<"
         << "\nc2: " << c2 << "<<"
         << "\nc3: " << c3 << "<<\n";
    cout << "r1: " << r1 << "\t\tr2: "
         << r2 << "\t\tr3: "
         << r3 << "<<\n";
    cout << "i4: " << i4
         << "\t\t\ti5: " << i5
         << "\t\t\ti6: " << i6 << "<<";
    cout << "\nc4: " << c4 << "<<"
         << "\nc5: " << c5 << "<<"
         << "\nc6: " << c6 << "<<\n";
    cout << "r4: " << r4
         << "\t\t\tr5: " << r5
         << "\t\t\tr6: " << r6 << "<<\n";
    cout << endl;
}
```

4.4.4.1 What you see for the first time in test_io.cpp

Actually, there is not much new in this program. You've seen it all before in one program or another, except for the reading in of several values with a single `cin` statement, so note carefully the syntax for doing this.

4.4.4.2 Additional notes and discussion on test_io.cpp

This program is supplied solely for the purpose of giving you lots of additional practice with keyboard input, and you should take full advantage of it to get that practice. In particular, feel free to add to, delete, or change the order of, the various input statements, and make up your own additional test data sets.

4.4.4.3 Follow-up hands-on activities for test_io.cpp

□ Activity 1 Copy, study, test and write pseudocode for `test_io.cpp`. Use at least the data sets shown below as input data when testing the program, but make up some of your own as well. For each input data set, enter your predicted output values in the spaces following the given input data set *before* running the program on that data set.

a. First data set:

```
1 2 3
a b c
4.5 6.7 8.9
-1-2-3-4-5-6-7
```

```
i1 = ________    i2 = ________    i3 = ________

c1 = ________    c2 = ________    c3 = ________

r1 = ________    r2 = ________    r3 = ________

i4 = ________    i5 = ________    i6 = ________

c4 = ________    c5 = ________    c6 = ________

r4 = ________    r5 = ________    r6 = ________
```

b. Second data set:

```
6-2+3.6*5.2e-1+4.02-3.6E+2-9-8-7
/\.2-3+0
```

```
i1 = ________    i2 = ________    i3 = ________

c1 = ________    c2 = ________    c3 = ________

r1 = ________    r2 = ________    r3 = ________

i4 = ________    i5 = ________    i6 = ________

c4 = ________    c5 = ________    c6 = ________

r4 = ________    r5 = ________    r6 = ________
```

c. Third data set:

```
2
1-3.
+6.54E1-.1.7
+6 54 1-.1.7.6+5
```

```
i1 = ________    i2 = ________    i3 = ________

c1 = ________    c2 = ________    c3 = ________

r1 = ________    r2 = ________    r3 = ________

i4 = ________    i5 = ________    i6 = ________

c4 = ________    c5 = ________    c6 = ________

r4 = ________    r5 = ________    r6 = ________
```

□ Activity 2 Make a copy of `test_io.cpp` called `test_io1.cpp` and modify the copy so that it outputs the eighteen values in two groups of nine each, with a pause in between and a request that the user press Enter to see the other nine. Think carefully about this, and try to get it right on the first try, *with the minimal number of changes to the original program.* If you do, it should increase your confidence in your understanding of the relevant details.

○ Instructor checkpoint 4.4 for evaluating prior work

Module 5

Simple file input and output

5.1 Objectives

- To understand how to use the `fstream` library when you need to perform file input and/or output.
- To appreciate the strong parallels between *textfile I/O* (input from, and output to, a file of human-readable characters) and *standard I/O* (which is also called *console I/O*, and refers to input from the keyboard and output to a screen which scrolls up a line at a time until the top line "disappears" off the top of the screen).

5.2 List of associated files

- `textfile_io.cpp` inputs data from a textfile and outputs data to a textfile.
- `textfile_io.in1`, `textfile_io.in2` and `textfile_io.in3` are sample input data files for the program in `textfile_io.cpp`.

5.3 Overview

Every program seen prior to this Module sent output to the screen, and read input, if at all, from the keyboard. In this Module we briefly introduce the idea of a program writing output to a textfile and/or reading input from a textfile. This is clearly an essential requirement if there are large amounts of data.

5.4 Sample Programs

The one sample program in this Module should be compared closely with the sample program `test_io.cpp` from the previous Module, after which you should agree with the following statement: One of the nice things about C++ is the ease with which one can go from standard input/output (keyboard and screen) to textfile input/output.

5.4.1 textfile_io.cpp illustrates how to read input from a textfile and write output to a textfile

```
//textfile_io.cpp
//Illustrates C++ input from a file and output to a file.

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

int main()
{
    cout << "\nThis program reads data from a textfile called "
        "\"in_data\", and writes it out\nto another textfile called "
        "\"out_data\". The file in_data must be present.\n";
    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’);

    ifstream inFile;          //Declare a file variable for input
    ofstream outFile;         //Declare a file variable  for output
    inFile.open("in_data");   //Connect "inFile" to a physical file
    outFile.open("out_data"); //Connect "outFile" to a physical file

    int i1, i2, i3, i4, i5, i6;
    char c1, c2, c3, c4, c5, c6;
    double r1, r2, r3, r4, r5, r6;

    inFile >> i1 >> i2 >> i3;
    inFile >> c1 >> c2 >> c3;
    inFile >> r1 >> r2 >> r3;
    inFile >> i4 >> i5 >> i6;
    inFile.get(c4);
    inFile.get(c5);
    inFile.get(c6);
    inFile >> r4 >> r5 >> r6;
    inFile.close(); //Close the input file

    outFile << "\nHere are the eighteen values read in, with a "
        "\"<<\" marker\nat the end of each output line:\n";
    outFile << "i1: " << i1
            << "\t\t\ti2: " << i2
            << "\t\t\ti3: " << i3 << "<<";
    outFile << "\nc1: " << c1 << "<<"
            << "\nc2: " << c2 << "<<"
            << "\nc3: " << c3 << "<<\n";
    outFile << "r1: " << r1 << "\t\tr2: "
            << r2 << "\t\tr3: "
            << r3 << "<<\n";
    outFile << "i4: " << i4
            << "\t\t\ti5: " << i5
            << "\t\t\ti6: " << i6 << "<<";
    outFile << "\nc4: " << c4 << "<<"
            << "\nc5: " << c5 << "<<"
            << "\nc6: " << c6 << "<<\n";
    outFile << "r4: " << r4
            << "\t\t\tr5: " << r5
            << "\t\t\tr6: " << r6 << "<<\n";
    outFile << endl;
    outFile.close(); //Close the output file

    cout << "\nIf all went well, the input file has been read, and "
        "the output file has been\nwritten. However, you should, of "
        "course, check to be sure.\n";
    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’);
}
```

5.4.1.1 What you see for the first time in textfile_io.cpp

- The reading of data values from an input textfile and the writing of values to an output textfile

  Reading from, and writing to, textfiles

- The inclusion of the `fstream` library, for access to the input filetype `ifstream`, and to the output filetype `ofstream`

  `fstream` *library*

- The declaration of an input file variable (`inFile`) of data type `ifstream` and an output file variable (`outFile`) of data type `ofstream`

  Declaration of file types for input and output

  Technically, `inFile` is a *class object* of `class` type `ifstream` and similarly `outFile` is a class object of `class` type `ofstream`, but thinking of objects as variables and classes as data types for the moment will do no harm, and possibly much good.

- The use of `program_filename.open("operating_system_filename")` to associate the name used by a program for a data file with the name used by the operating system for the same file

  Connecting a program filename with the physical filename

- The use of program filenames like `inFile` and `outFile` in a manner completely analogous to the way in which `cin` and `cout` are used to read input and display output, respectively

  Strong parallel with use of `cin` *and* `cout`

- The use of `program_filename.close()` to "close" a file when the program is no longer using it

  Closing a file

5.4.1.2 Additional notes and discussion on textfile_io.cpp

You should be impressed, and thankful to Mr. Stroustrup[1], for the way in which C++ input from textfiles and output to textfiles parallels the way C++ inputs data from the keyboard and outputs it to the screen. If you look for a moment at the program in `textfile_io.cpp` and concentrate just on the statements that begin with `inFile` and `outFile`, you will note that these are precisely analogous to the way the corresponding `cin` and `cout` statements would look if we were getting input from the keyboard and sending output to the screen.

Setting up a file for input or output

Thus, the only *additional* work we have to do to use textfiles for input or output consists of the following four steps:

- Include the `fstream` header file.
- Declare the necessary file variable(s).
- Connect each file variable to a physical file.
- "Close" each file when we are finished with it.

  A best practice

[1] Bjarne Stroustrup, who designed the C++ programming language at Bell Labs in the early-to-mid-1980's, and has continued to play a leading role in its development ever since.

Note that the last three steps are always performed automatically for `cin` and `cout` whenever the `iostream` header file is included in a program.

When we read input from a file, there is of course no need for prompts. However, this need is replaced by another: namely, the need to know exactly what the format of the input data in the file is, so that the program knows what to expect when it attempts to read data from the file.

Screen image and keyboard image

It is quite useful to think of the input from a textfile as a *keyboard image* and the output to a textfile as a *screen image*. What we mean by these terms is this: Thinking of the input from a file as a "keyboard image" means that if the input data in an input data file appears in exactly the same format as it would have if you typed it in from the keyboard, then it will be read in exactly the same way by the program as if you were entering it from the keyboard and the program were reading it with `cin`. Similarly, thinking of the output data sent to an output data file as a "screen image" means that if the output data file is displayed on the screen as a textfile the display will look exactly as if the program had itself sent the data directly to the screen with `cout`.

5.4.1.3 Connecting program filenames and actual filenames on your system

If a C++ program is to read from a file, or write to a file, then it needs a name for the file in order to refer to the file. For example, in `textfile_io.cpp` the program uses the name `inFile` (a programmer-chosen identifier) to refer to the file from which it reads input. Note that this is *not* (or at least doesn't *have* to be) the actual name of the file as far as the operating system is concerned (i.e., you might not see anything called `inFile` if you looked at a list of the files in your working directory).

Program name for a file may be different from the actual name of the physical file on disk

So, what is the actual name of the data file? It's important to note that you can't deduce the answer to this question by looking at the program in `textfile_io.cpp`. It *may* be `in_data`, the same name that appears within the double quotes in the `inFile.open` statement, but it may just as well *not* be. What *is* true, conceptually, however, is that `in_data` is the name passed by the C++ program to the operating system. It is up to the operating system to give back to the program the name of a file which is also set up for the action (input or output of data, for example) which the program wants to perform on that file (and which the program will henceforth think of as `inFile`).

The default behavior may be for the operating system to look for a file having the same name as the program passes to it, but it may also be the case that the operating system has been told via some other means to think of the name `in_data` as some particular file, say `myOldData` or `FEBRUARY.DAT`. Analogous statements hold for the file `outFile`.

Answer

How does your operating system deal with the name that appears within the double quotes in a C++ executable statement like `file.open("some_name");`? Or, what do you do if `some_name` is *not* the actual name of the file?

5.4.1.4 Follow-up hands-on activities for textfile_io.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `textfile_io.cpp`. For testing, use the sample input data files whose contents are shown below. For the output files, use the same filenames as the input filenames, but supply extensions `out1`, `out2` and `out3`, to correspond with the input file extensions `in1`, `in2` and `in3`. Find a way to complete this activity *without* making any changes to `textfile_io.cpp`. Fill in your predictions for the output values in the given spaces following each input data set. Then compare the contents of the corresponding output file with your predictions.

The first sample input data set is in a file called `textfile_io.in1`, and is shown between the heavy lines below:

```
3 2-1
ab c
-5.4-7.6+9.8
+7+6+5+4+3+2+1
```

`i1 =` ________ `i2 =` ________ `i3 =` ________

`c1 =` ________ `c2 =` ________ `c3 =` ________

`r1 =` ________ `r2 =` ________ `r3 =` ________

`i4 =` ________ `i5 =` ________ `i6 =` ________

`c4 =` ________ `c5 =` ________ `c6 =` ________

`r4 =` ________ `r5 =` ________ `r6 =` ________

The second sample input data set is in a file called `textfile_io.in2`, and is shown between the heavy lines below:

```
5-3+2.5*6.3e-2+3.03-4.5E+1-7-6-5
+=.3-2+0
```

i1 = ________ i2 = ________ i3 = ________

c1 = ________ c2 = ________ c3 = ________

r1 = ________ r2 = ________ r3 = ________

i4 = ________ i5 = ________ i6 = ________

c4 = ________ c5 = ________ c6 = ________

r4 = ________ r5 = ________ r6 = ________

The third sample input data set is in a file called `textfile_io.in3`, and is shown between the heavy lines below:

```
4
3-2.
+7.23E2-.2.5
+7 23 2-.2.5.9+3
```

i1 = ________ i2 = ________ i3 = ________

c1 = ________ c2 = ________ c3 = ________

r1 = ________ r2 = ________ r3 = ________

i4 = ________ i5 = ________ i6 = ________

c4 = ________ c5 = ________ c6 = ________

r4 = ________ r5 = ________ r6 = ________

□ Activity 2 Copy `textfile_io.cpp` to `textfile_io1.cpp` and bug it as follows:

a. Omit the line that includes the `fstream` header file.

____

b. Omit the declaration of `inFile`.

____

c. Omit the declaration of `outFile`.

____

d. Omit the statement which opens the input file.

____

e. Omit the statement which opens the output file.

____

f. Combine the first three input statements into a single input statement.

____

g. Change the executable statement `inFile.get(c4);` to `inFile >> c4;`, and make analogous changes to the two statements that read values into `c5` and `c6`.

____

h. Omit the statement which closes the input file.

____

i. Omit the statement which closes the output file.

____

○ Instructor checkpoint 5.1 for evaluating prior work

Module 6

A simple “shell” starter program with I/O, a menu, selection, and looping

6.1 Objectives

- To learn about the following C++ reserved words: `switch`, `case`, `break`, `do`, and `while`.
- To understand what is meant by a *menu-driven program*.
- To understand the basic notion of *selection*, which allows a program to choose what to do next from two or more alternative actions.
- To understand the basic notion of *looping*, which allows a program to repeat one or more actions.
- To understand how the C++ switch-statement works to permit selection in the context of a simple menu-driven program, by choosing the appropriate one of several “cases”, each of which is an action to be performed.
- To understand how the C++ break-statement works in the context of a switch-statement.
- To understand how the C++ do...while-statement works to permit looping in the context of a simple menu-driven program.
- To understand how formatting is used (in particular, indentation and alignment) to make a program with selection and looping more readable.
- To understand the pseudocode structure for any simple menu-driven program.

- To gain further appreciation for the way pseudocode can be used to "hide the details" in a program and allow the programmer to describe clearly the actions to be performed at a "higher level" and postpone having to deal with the "lower level" details.

6.2 List of associated files

- `shell.cpp` contains a general-purpose "shell" program.

6.3 Overview

Moving beyond simple sequential logic

Up until now, all of our programs have been entirely sequential. That is, each program started by executing the first executable statement (in the `main` function), then the second, and so on in sequence until all of the executable statements in `main` had been executed, at which point the program ended and returned a value of 0 to the operating system to indicate "success". Programs structured in this way are of course quite limited in the kinds of things they can do, since they have no decision-making ability and no ability to repeat any actions unless all of the code to repeat the action is itself duplicated in the program.

Selection and looping

In this Module we introduce in a simple way two concepts that are vital to computer programming: the notion of *selection*, and the notion of *looping*. The first allows a program to decide which of various alternatives to perform and the second allows actions to be repeated.

Both of these ideas will be covered in much more detail in later Modules, but we introduce them here in the context of a simple *menu-driven program* which serves two purposes: it illustrates selection and looping in a context in which they are actually often used, and the program itself can be used as a model and starting point for many programs that you will write from now on.

6.4 Sample Programs

This is another Module with just one sample program, but a very important one. It introduces several new ideas you will be using continually from now on. Be sure you get a good grip on the two central concepts ("big ideas") that it illustrates, i.e., *selection* (choosing one from several alternative actions) and *looping* (repeating one or more actions). You will have lots of opportunity to deal with the small details of the actual constructs used here, as well as other related constructs, as time goes on.

6.4.1 shell.cpp provides a general-purpose menu-driven "shell" starter program

```
//shell.cpp
//Acts as a general-purpose "shell" program.

#include <iostream>
using namespace std;

int main()
{
    int menuChoice;
    do
    {//Body of do...while-statement must be enclosed in braces
        cout << "\n\n\t\t" << "Main Menu"
                "\n\n\t\t" << "1. Quit"
                "\n\t\t"   << "2. Get information"
                "\n\t\t"   << "3. Perform some action"
                "\n\t\t"   << "4. Perform some other action\n\n";

        cout << "Enter the number of your menu choice here "
            "and then press the Enter key: ";
        cin >> menuChoice;  cin.ignore(80, '\n');
        cout << endl << endl;

        //A break-statement within a switch-statement causes a "jump"
        //to the first statement following the switch-statement.
        switch (menuChoice)
        {//Body of switch-statement must be enclosed in braces
            case 1: //Do this if menuChoice is 1.
                cout << "You have chosen to quit. "
                    "Program is now terminating.\n";
                break;

            case 2: //Do this if menuChoice is 2.
                cout << "This program can be used as a starting point "
                    "for many C++ programs.\nA general description of "
                    "whatever your program does should go here.\n";
                break;

            case 3: //Do this if menuChoice is 3.
                cout << "Now performing some action in response "
                    "to user choosing menu option 3 ...\n";
                break;

            case 4: //Do this if menuChoice is 4.
                cout << "Now performing some action in response "
                    "to user choosing menu option 4 ...\n";
                break;
        }
        cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');

    } while (menuChoice != 1); //While user does not choose quit option
    cout << endl << endl;
}
```

6.4.1.1 What you see for the first time in shell.cpp

More C++ keywords

- The following new C++ reserved words: `switch`, `case`, `break`, `do`, and `while`.

Menu-driven programs

- A *menu-driven program*, which is a program that offers the user a *menu* of possible actions, prompts the user to enter a *menu choice*, and then responds appropriately to the user's choice

More on output formatting

- Some variants on the `\n\t` character combination used for positioning output, which can often (as you see here) lead to somewhat more compact code than using `endl` and blank spaces to achieve a similar effect

Selection with `switch`

- The use of a switch-statement to make a selection (in this case, based on the user's menu choice)
- The use of a *case-selector* and a break-statement within a switch-statement

Looping with `do...while`

- The use of a do...while-statement to perform a loop

6.4.1.2 Additional notes and discussion on shell.cpp

Flow of control

First, you will already have observed that this program doesn't really do very much, so it is the overall structure and *flow of control* in the program that's important here.

Choosing appropriate control structures

There are alternate C++ *control structures* that provide other ways to perform both selection and looping, and we will meet them in later Modules. However, it can be argued that the do...while-statement and the switch-statement are the "natural" choices for this program, if not the only ones. Thus, we could have used a while-statement and a nested-if construct or sequential-if construct (both to be discussed in later Modules), but for the task at hand we argue that the most "readable" choices are the ones we've made, for the reasons we now discuss.

`do...while` *is "best" here*

The do...while-loop is a "natural" choice here since it is an example of what is called a *post-test loop* (the test for whether the statements in the *body* of the loop should be executed comes *after* the statements, i.e., at the *end* of the loop) and in this program that's what we need since we *always* want the menu displayed and a choice made *at least once.*

`switch` *is "best" here*

The switch-statement is a natural choice here for the decision-making because it provides a mechanism for making a *multi-way decision*, based on which of several constant values is matched by the value of a variable (or expression). That is, there are more than two choices and the choice of the action to be performed can be determined by matching the user's menu choice entry with one of the *labels* or "cases" in the switch-statement.

Take particular note of the formatting of both the switch-statement and the do...while-statement. Note how the "body" of each is indented relative to the braces which enclose it. Note how those enclosing braces are aligned with one another and with the key words of the construct. What you see here is not the way everyone does it, but a fairly common convention and the one we will follow.

It is worth pointing out that although this particular menu-driven program is of the simplest and somewhat "old-fashioned" kind, the principles and structure would remain the same, no matter how complex the details. In other words, if we were dealing with a *graphical user interface* and a *drop-down menu* in color to display the choices to the user, and then a *mouse click* by the user to make the choice, we would still need code to produce the display and code to get the user's response. The code would be much more complicated and the interface "fancier", but the underlying principles are exactly the same.

The principles of a menu-driven program remain independent of the actual user interface.

This program is a very useful starting point for many console programs. It is easily *modifiable* and easily *extensible* just by adding more menu options and more corresponding choices in the switch-statement, and by including code that actually *does* something instead of code that just *says* something.

The shell program is easily modified and extended.

Note that the pseudocode for a simple menu-driven program like the one in `shell.cpp` takes the following form, in which we take advantage of the fact that our language of choice is C++ by incorporating some C++ keywords into the pseudocode.

Pseudocode for a menu-driven program

```
do
    Display a menu
    Get a menu choice from the user
    Perform the action corresponding to the user's choice
while the user has not chosen to quit
```

This is fair game, and a practice commonly followed by programmers. That is, C++ programmers will often write pseudocode that "looks like" C++, Pascal programmers might well write Pascal-like pseudocode, and so on.

For example, without going into details about Pascal, let's say that it's entirely possible that a Pascal programmer might write the above pseudocode like this:

```
repeat
    Display a menu
    Get a menu choice from the user
    Perform the action corresponding to the user's choice
until user chooses to quit
```

And, though those familiar with the Pascal programming language would recognize this as Pascal-like pseudocode, it should be no harder for a C++ programmer with equivalent experience to understand this version than it was for that same programmer to understand the first version. This, of course, is the whole point of pseudocode.

One could even take a stab at writing the above pseudocode in a "language independent" fashion, as in

```
loop
    Display a menu
    Get a menu choice from the user
    Perform the action corresponding to the user's choice
endloop if choice is ``quit''
```

but there are, thankfully, no universally accepted conventions for writing pseudocode which we all have to follow.

6.4.1.3 Follow-up hands-on activities for shell.cpp

□ Activity 1 Copy, study and test the program in `shell.cpp`. On this occasion, we have already provided the program's pseudocode.

□ Activity 2 Copy `shell.cpp` to `shell1.cpp` and bug it as follows:

a. Change the type of `menuChoice` from `int` to `char` in its declaration.

   ____

b. Remove the line of code that causes the program to pause after a menu choice has been executed. Do you like the "user interface" when the program runs better before this change, or afterwards?

   ____

There are many other changes we could ask you to try in this program, and you should feel free to introduce some of your own. However, since we have introduced the do...while-statement, switch-statement and break-statement here just for the convenience they provide to our "shell", and not to be studied in detail at this point, we postpone the "bugging" of these constructs till later when we consider them in depth.

○ Instructor checkpoint 6.1 for evaluating prior work

□ Activity 3 Make a copy of `shell.cpp` called `shell_hello.cpp` and modify the copy so that it is essentially a menu-driven version of the "Hello, world!" program of `hello.cpp` from Module 1. That is, the program must be menu-driven, must quit if the user chooses option 1, must display a brief description of itself if option 2 is chosen, and must display "Hello, world!" if option 3 is chosen. There need not be an option 4 in this case. Pay some attention to the formatting of your output so that it is pleasing to the eye.

○ Instructor checkpoint 6.2 for evaluating prior work

□ Activity 4 Make a copy of `shell.cpp` called `shell_myinfo.cpp` and modify the copy so that choosing option 1 quits, choosing option 2 provides a program description, choosing option 3 displays your name and address while at university, and choosing option 4 displays a list of the courses you are taking during the current term. Pay some attention to the formatting of your output so that it is pleasing to the eye.

○ Instructor checkpoint 6.3 for evaluating prior work

Module 7

Evaluating arithmetic expressions

7.1 Objectives

- To learn how to form and evaluate *arithmetic expressions* involving the *arithmetic operators* for addition (`+`), subtraction (`-`), multiplication (`*`), and division (both `/` and `%`).
- To understand the *order of precedence* of these operators when evaluating expressions.
- To learn how the following special C++ operators work: `++`, `--`, `+=`, `-=`, `*=`, `/=`, `%=`

7.2 List of associated files

- `eval_expressions.cpp` illustrates the evaluation of some simple arithmetic expressions.
- `more_operators.cpp` illustrates some special C++ operators.

7.3 Overview

Arithmetic is much like you'd expect, except for division.

The evaluation of arithmetic and other numerical expressions forms a large part of what computer programs do, so it will be important for you to know how arithmetic works in C++. For the most part, things are much like you would expect from whatever prior experience you may have had, but as always there are some things to watch out for. Also, C++ is a language rich in arithmetic operators so there are some new ones to learn, as well as a few "shortcut" assignment operators, each of which combines the usual assignment operator with an arithmetic operator.

7.4 Sample Programs

7.4.1 eval_expressions.cpp evaluates arithmetic expressions and displays those values

```
//eval_expressions.cpp
//Illustrates arithmetic expression evaluation, and
//arithmetic operator precedence rules.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program displays the results of "
        "evaluating several arithmetic expressions.\n\n";

    cout << "This first group involves only integer values:\n"
         << "1 + 2 * 5 - 3 * 5     --> " << 1 + 2 * 5 - 3 * 5     << "\t"
         << "(1 + 2) * 5 - (3 * 5) --> " << (1 + 2) * 5 - (3 * 5) << "\n"
         << "13 / 2 * 3            --> " << 13 / 2 * 3            << "\t"
         << "12 + 3 / 4 * 2        --> " << 12 + 3 / 4 * 2        << "\n"
         << "238 % 10 + 3 % 7      --> " << 238 % 10 + 3 % 7      << "\t"
         << "5 * 2 / 4 * 2         --> " << 5 * 2 / 4 * 2         << "\n"
         << "5 * 2 / (4 * 2)       --> " << 5 * 2 / (4 * 2)       << "\t"
         << "5 + 2 / (4 * 2)       --> " << 5 + 2 / (4 * 2)       << "\n"
         << "10 % 3 - 4 / 2        --> " << 10 % 3 - 4 / 2        << "\n\n";

    cout << "This second group involves only floating point values:"
         << "\n2.61 + 13.4 - 6.2 / 0.2 --> " << 2.61 + 13.4 - 6.2 / 0.2
         << "\n(-5.3 + 1.1) * 5.0      --> " << (-5.3 + 1.1) * 5.0
         << "\n\n";

    cout << "This third group involves both "
         << "integer and floating point values:\n"
         << "3.1 + 2         --> " << 3.1 + 2         << "\t\t"
         << "5 - 3.6       --> "   << 5 - 3.6         << "\n"
         << "3.14 * 4        --> " << 3.14 * 4        << "\t"
         << "0.123 / 3     --> "   << 0.123 / 3       << "\n"
         << "3.6 + 6 / 10    --> " << 3.6 + 6 / 10    << "\t\t"
         << "-1.7 + 10 / 4 --> "   << -1.7 + 10 / 4   << "\n"
         << "-1.7 + 10 / 4.0 --> " << -1.7 + 10 / 4.0 << "\t\t"
         << "1/3 + 2/3     --> "   << 1/3 + 2/3       << "\n"
         << "1.0/3 + 2/3     --> " << 1.0/3 + 2/3     << "\t"
         << "1.0/3 + 2/3.0 --> "   << 1.0/3 + 2/3.0   << "\n"
         << endl;
}
```

7.4.1.1 What you see for the first time in eval_expressions.cpp

- The use of all four of the "usual" arithmetic operators—the `+`, `-`, `*` and `/` operators—in arithmetic expressions involving integer values, real number values, and a mixture of both

The usual arithmetic operators: `+`, `-`, `*`, `/`

- Examples showing that the division operator `/` may be used to perform both *integer division* (which occurs when both numerator and denominator, i.e., both *operands*, are integers, and the result of which is just the *quotient* of the division) and *real division* (which occurs when at least one of the operands is a *floating point value*, which is the usual division you would probably expect, and the result of which is the one you would normally get if you divided the same two numbers using a calculator)

But be very careful how you use the `/` *operator.*

- The use of the modulus operator `%` to find the remainder after dividing one integer by another (i.e., after *integer division*)

Know what the `%` *operator does for you.*

- Examples in which a knowledge of *operator precedence* is necessary for proper evaluation

Operator precedence

  An *operator precedence table* is the most convenient way to view arithmetic (or other) operators and their precedence. Here is the first of several such tables that you will see, and note that the assignment operator (=) is also included:

```
highest     *  /  %
            +  -
lowest      =
```

  The fact that the assignment operator has the lowest precedence in the table is the reason why, in an assignment statement like

  `n = 2 * 3;`

  the multiplication takes place *first*, and *then* the assignment.

- The use of parentheses to clarify, as well as to change, the order in which operations will be performed

Parentheses used for change or clarification

- The behavior of `cout` when asked to output an arithmetic expression: It first evaluates the expression, then outputs the result; to output the actual expression you have to make the expression a string constant by enclosing it in double quotes.

How `cout` *"outputs an arithmetic expression"*

7.4.1.2 Additional notes and discussion on eval_expressions.cpp

Compare carefully the source code of this program with the output it produces, and make sure you understand how the formatting of the output is achieved. Not that this particular program is a model to be admired and emulated. It's just that, as always, it is useful to study code and see how it does what it does. This program will require some attention to detail to do this, but will therefore provide some good practice in program reading.

7.4.1.3 Follow-up hands-on activities for eval_expressions.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `eval_expressions.cpp`.

Of course it is *always* important, and we take it to be *implicit* most of the time, but here it's *really* important so we mention it *explicitly*: Don't forget to *predict* the output when you run the sample program *before* you actually run it.

Remember, in this case as in all similar situations, that just running the program and staring at the output is a *complete waste of time.*

□ Activity 2 Copy `eval_expressions.cpp` to `eval_expressions1.cpp` and bug it as follows:

a. In both instances of (1 + 2) in line 15 remove the parentheses.

b. Replace `0.123 / 3` by `0.123 % 3` in line 34.

□ Activity 3 First, predict the output of the code shown below by entering what you believe to be the output in the spaces provided following the code. Then write a suitable program that includes this code and test your prediction. Put the program in file called `eval_expressions2.cpp`.

```
cout << 8 - 5 % 7 + 16 / 3 * 2 << "  "
     << -1.5*5 - 3 + 1/2              << endl;
cout << 8.2 - 4 / 3 + 6 / 0.5 << "  "
     << 7 - 4 / 3 + 9 % 4 - 1         << endl;
```

□ Activity 4 Make another copy of the file `eval_expressions.cpp` and call the copy `eval_expressions3.cpp`. Modify the copy so that each literal numerical value in the program is 2 more than its value in the original program. This gives you a whole new set of computed values to predict. So, predict what values will be output when this revised program is now compiled, linked and run. Run the program and compare your predictions with the output.

Based on shell.cpp

□ Activity 5 Make a copy of the file `shell.cpp` from Module 6 and call the copy `shell_calculator.cpp`. Modify the copy to produce a simple "calculator program" which allows the user to add, subtract, multiply or divide two integers, and to do this as many times as the user desires before quitting.

○ INSTRUCTOR CHECKPOINT 7.1 FOR EVALUATING PRIOR WORK

7.4.2 more_operators.cpp illustrates the increment, decrement, and some special assignment operators

```
//more_operators.cpp
//Illustrates the use of the increment and decrement
//operators and some additional assignment operators.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program illustrates the use of the increment and "
        "decrement operators\n(++ and --) as well as the following "
        "assignment operators: +=, -=, *=, /=, %=\n\n";

    int i = 6;
    int j = -3;
    double x = 4.23;
    double y = -0.72;

    cout << "On the line below each arrow points at "
         << "the value of the variable on its left:\n"
         << "  i --> "      << i
         << "       j --> " << j
         << "     x --> "   << x
         << "   y --> "     << y << endl << endl;

    i++; //Increments i by 1 (equivalent to: ++i; or i = i + 1;)
    --j; //Decrements j by 1 (equivalent to: j--; or j = j - 1;)
    ++x; //Increments x by 1 (equivalent to: x++; or x = x + 1;)
    y--; //Decrements y by 1 (equivalent to: --y; or y = y - 1;)

    cout << "On the line below each arrow points at "
         << "at the value of the same variable\n"
         << "after the statement on its left has executed:\n"
         << "  i++; --> " << i  << "  --j; --> " << j
         << "  ++x; --> " << x  << "  y--; --> " << y << "\n\n";

    cout << "Now we execute the following statements:\n"
         << "    x += 6.32;\n"
         << "    y -= -1.73;\n"
         << "    i *= 6 + 3;\n"
         << "    j /= 3;\n"
         << "    i %= 11;\n\n";
    x += 6.32;
    y -= -1.73;
    i *= 6 + 3; //What about operator precedence here?
    j /= 3;
    i %= 11;

    cout << "And finally, once more ...\nOn the line below each arrow "
        "points at the value of the variable on its left:\n"
        << "  i --> "      << i
        << "       j --> " << j
        << "     x --> "   << x
        << "   y --> "     << y << endl << endl;
}
```

7.4.2.1 What you see for the first time in more_operators.cpp

Special operators for incrementing/decrementing numerical variables.

- The use of the increment and decrement arithmetic operators—the `++` and `--` operators—to increment or decrement by 1 the value in a variable containing an integer or a variable containing a real number

Special assignment operators

- The use of these special assignment operators: `+=`, `-=`, `*=`, `/=`, and `%=`

7.4.2.2 Additional notes and discussion on more_operators.cpp

At least for now, use the increment and decrement operators only in stand-alone statements.

At least for the moment you should follow this advice: Use the increment and decrement operators *only* in the way shown in this program, i.e., use them *only* to increment or decrement a single variable in a stand-alone statement.[1] Do *not* use these operators in arithmetic expressions. For that reason we need not consider their precedence. The precedence of the new assignment operators is the same as that of the original assignment operator.

7.4.2.3 Follow-up hands-on activities for more_operators.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `more_operators.cpp`.

□ Activity 2 Copy `more_operators.cpp` to `more_operators1.cpp` and then bug it as follows:

a. In the statement `x += 6.32;` of line 43, change `+=` to `=+`.

b. Change the statement `i %= 11;` of line 47 to `x %= 11`.

□ Activity 3 First, predict the output of the code shown below by entering what you believe to be the output in the spaces provided following the code. Then write a suitable program that includes this code and test your prediction. Put the program in file called `more_operators2.cpp`.

```
int i = -12;
double x = 4.3;
i--;
++x;
i *= -3;
i %= 8;
x -= i;
cout << i << "  " << x << endl;
```

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

○ INSTRUCTOR CHECKPOINT 7.2 FOR EVALUATING PRIOR WORK

[1] But for the curious and venturesome, see `increment_decrement.cpp` in Module 19 for some of the things that can go wrong if you do not follow the advice of the above paragraph.

Module 8

Using library functions

8.1 Objectives

- To learn about some of the mathematical functions in the C++ standard `cmath` and `cstdlib` libraries.
- To learn the names of some of the other C++ standard libraries, the names of some of the functions contained in those libraries, and what those functions can do for you.

8.2 List of associated files

- `library_functions.cpp` performs calculations using some of the common mathematical functions found in the Standard C++ libraries `cmath` and `cstdlib`.

8.3 Overview

The C++ libraries contain many useful functions.

Many calculations and other procedures are best performed with the help of "built-in" functions available from one or more of the standard C++ libraries. We are already familiar with a few of the C++ libraries—the `iostream` and `iomanip` libraries, for example—from which we get the things we usually need to perform keyboard input and screen output.

Always try to avoid reinventing the wheel. Check your C++ libraries!

In this Module you explore additional C++ libraries and functions contained in them. Becoming familiar with what is available from the C++ libraries is an important part of becoming a good C++ programmer. In fact, whenever you need to perform a certain calculation or task, the first question to cross your mind should be: Is there a function in one of the C++ libraries that will do the job for me? This approach will help to keep you from "reinventing the wheel" on many occasions.

8.4 Sample Programs

8.4.1 library_functions.cpp performs calculations using functions from the standard cmath library

```
//library_functions.cpp
//Illustrates computations involving functions from the C++ cmath library.

#include <iostream>
#include <cmath>
//#include <cstdlib>
using namespace std;

int main()
{
    cout << "\nThis program displays the results of evaluating "
        "expressions\ninvolving functions from the C++ cmath "
        "library.\n\n";

    cout << "The square root of 25.0 is "  << sqrt(25.0)  << ".\n"
         << "The square root of 12.34 is " << sqrt(12.34) << ".\n\n";

    cout << "The absolute value of 2 is "       << abs(2)      << ".\n"
         << "The absolute value of -2 is "      << abs(-2)     << ".\n"
         << "The absolute value of 13.456 is "  << abs(13.456) << ".\n"
         << "The absolute value of -13.456 is " << abs(-13.456)
         << ".\n\n";

    cout << "2.0 raised to the power 3   is " << pow(2.0, 3)   << ".\n"
         << "2.4 raised to the power 3.1 is " << pow(2.4, 3.1) << ".\n"
         << ".04 raised to the power -.5 is " << pow(.04, -.5) << ".\n\n";

    double r;
    r = sqrt(1 + pow(4.0, 3));
    cout << "The square root of 1 more than "
         << "the cube of 4 is " << r << ".\n";
    cout << endl;
}
```

8.4.1.1 What you see for the first time in library_functions.cpp

Most commonly-used mathematical functions are readily available.

- The inclusion of the `cmath` and `stdlib` header files[1] for access to many of the common mathematical functions that the C++ Standard Library provides
- Use of the `sqrt`, `abs`, and `pow` mathematical functions
- Conversion of the verbal description of an expression into an actual expression for evaluation (a kind of "word problem")

[1] The fact that the names of these two header files both begin with a `'c'` indicates that they are two of the "legacy" libraries carried over to C++ from the C programming language. The new C++ Standard naming convention for the old C libraries is to add this extra `'c'` to the beginning of the old name. Your compiler may permit (or still require) either <`math.h`> or <`stdlib.h`>, or both, instead.

8.4.1.2 Additional notes and discussion on library_functions.cpp

There are many more functions in the C++ `math` library than we have used in `library_functions.cpp`, including exponential, logarithmic, trigonometric and other functions. On the other hand, not all mathematical functions available in C++ are found in the math library either. Many of the libraries available to C++ programmers are carried over from the C programming language, and so for historical reasons the distribution of functions throughout libraries is not quite what it might have been if the libraries had all been designed from scratch.

Check your local documentation to see what's available.

8.4.1.3 Follow-up hands-on activities for library_functions.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `library_functions.cpp`.

□ Activity 2 Copy `library_functions.cpp` to `library_functions1.cpp` and bug it as requested below. Except for the last change, which should cause an error on every system, any of these "bugs" may or may not cause a problem, depending on your particular compiler.

a. Omit the compiler directive that includes the `cmath` header file.

____

b. Omit the compiler directive that includes the `cstdlib` header file.

____

c. Replace `sqrt(25.0)` in line 15 with `sqrt(25)`.

____

d. Replace `pow(2.0, 3)` in line 24 with `pow(2, 3)`.

____

e. Replace `pow(2.0, 3)` in line 24 with `pow(2.0)`.

____

□ Activity 3 The ability to read carefully is always a useful skill. This activity is designed to test your ability to read carefully the description of an expression to be evaluated, and then turn that description into a corresponding equivalent mathematical expression for purposes of evaluation in a program.

Each of the statements shown below is meant to be such a description, with the three question marks `???` representing the result that would be obtained by actually computing the value described. Your task is to first determine what each value would be, then design and write a suitable program that produces the exact output shown, except (of course) that each `???` is replaced in the output by the actual value as computed by the program. These exercises are reminiscent of the "word problems" that most stduents have encountered in high school.

If you find that one or more of the statements is ambiguous (i.e., it admits to more than one interpretation) then your program should output a statement (and include a corresponding formula) for *each* possible interpretation.

The values computed by your program must, naturally, agree with what you deduced they would be before writing the program. Place the program in a file called `word_problems.cpp`.

```
The absolute value of 4 less than the square root of 9 is ???.
The square root of twice the value of 3 to the power 4 is ???.
The value of 4 to the power of 3 less than the absolute value
    of -2.5 is ???.
The square root of the sum of 3 squared and 5 squared is ???.
```

□ Activity 4 This activity requires you to search through whatever information you have at hand on the C++ libraries. In each case, find a function that performs the given task or calculation and in the space provided list the name of the function and the library in which it is found (i.e., the header file that would have to be included in your program if your program were to make use of that function).

a. A function that will convert a lower case letter to the corresponding upper case letter ('b' to 'B', for example).

____

b. A function that will compute the sine of an angle.

____

c. A function that will compute the logarithm to the base 10 of a positive real number.

____

Based on shell.cpp

□ Activity 5 Make a copy of the file `shell.cpp` from Module 6 and call the copy `shell_squares_roots.cpp`. Modify the copy to produce a program which allows the user to find either the square or the square root of any positive number (integer or real) as many times as desired before quitting.

○ INSTRUCTOR CHECKPOINT 8.1 FOR EVALUATING PRIOR WORK

Module 9

Converting data values from one data type to another

9.1 Objectives

- To understand what it means to convert a data value from one data type to another.
- To understand the similarities and the differences between *type coercion* (*implicit type conversion*) and *type casting* (*explicit type conversion*).
- To learn how to perform a *type cast* (the old way *and* the new way).
- To become familiar with the *ASCII table*, and to learn how to convert values between the `char` and `int` data types by type casting.

9.2 List of associated files

- `number_conversion.cpp` converts values of one numerical data type to values of another numerical data type.
- `char_int_conversion.cpp` converts between `char` and `int` data values.

9.3 Overview

C++ permits values to be converted from one data type to another.

In this Module we consider the question of *type conversion*. In strongly-typed programming languages (like C++), most or all data values should never change their types. However, it is sometimes necessary (or at least convenient) for this to happen, and the rules may be relaxed somewhat to permit it. In any case, C++ *does* permit it, and you should be aware of the details.

9.4 Sample Programs

9.4.1 number_conversion.cpp converts values from one numerical data type to another

```
//number_conversion.cpp
//Illustrates the conversion of some values from one
//numerical data type to another, by coercion and by casting.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program shows instances in which a value of one "
        "data type is converted to\na value of another data type. "
        "Study both source code and output very carefully.\n\n";
    int i;
    double r;

    //Type coercion (implicit type conversion) in assignment statements:
    i = 3.24;  cout << i << endl; //Each of these two lines may generate
    i = 3.97;  cout << i << endl; //a conversion warning. Why?
    r = 5;     cout << r << endl << endl;
    //What style rule have we violated here (for the sake of readability)?

    //Type coercion in arithmetic expressions:
    cout << 2 * 3.51 + 4 / 1.2 << endl << endl;

    //Type casting (explicit type conversion) in assignment statements:
    i = static_cast<int>(3.24);  cout << i << endl;
    i = static_cast<int>(3.97);  cout << i << endl;
    r = static_cast<double>(5);  cout << r << endl << endl;


    //Type casting is sometimes necessary in arithmetic expressions
    //to obtain a correct answer, as is illustrated by:
    int numberOfHits = 68;
    int numberOfAtBats = 172;
    double battingAverage;

    battingAverage = 68 / 172;
    cout << battingAverage << endl; //Gives the wrong value. Why?

    battingAverage = static_cast<double>(numberOfHits / numberOfAtBats);
    cout << battingAverage << endl; //Still wrong. Why?

    battingAverage = static_cast<double>(numberOfHits) / numberOfAtBats;
    cout << battingAverage << endl; //OK, but ...
    battingAverage = static_cast<double>(numberOfHits) /
                     static_cast<double>(numberOfAtBats);
    cout << battingAverage << endl; //... even better (more specific)
    cout << endl;


    //Type casting is also useful for rounding numbers:
    cout << static_cast<int>(4.37 + 0.5) << endl;
    cout << static_cast<int>(4.61 + 0.5) << endl;
    cout << static_cast<int>(4.37 * 10 + 0.5) / 10.0 << endl;
    cout << static_cast<int>(4.61 * 10 + 0.5) / 10.0 << endl << endl;
}
```

9.4.1.1 What you see for the first time in number_conversion.cpp

Two kinds of type conversion

- Two forms of numerical *type conversion*
  - *Implicit* type conversion via *type coercion*
  - *Explicit* type conversion via *type casting*

Sometimes type casting is necessary.

- The need, in certain situations, to perform type casting in order to obtain the correct result in an arithmetic calculation

Casting used for rounding

- The use of type casting to perform rounding of floating point values

9.4.1.2 Additional notes and discussion on number_conversion.cpp

Type-safe programming languages

A perfectly *type-safe* programming language would not permit a floating point value like 3.24 to be assigned to an integer variable. However, such an assignment *is* permitted in C++, and in many other programming languages as well (a case of conceptual integrity giving way to operational convenience). C++ has, in fact, quite a complex set of rules for type conversion, only a few of which are illustrated here.

C-style casting still works in C++.

The mechanism for type casting used here is the `static_cast` operator. This and three other casting operators that we have no need for at the moment have been introduded into C++ relatively recently. There are other, simpler, but "old-fashioned" ways of casting carried over from the C language that also continue to work in C++. We will look at these alternative casting methods from C in the hands-on activities, and use them in later programs.

9.4.1.3 Follow-up hands-on activities for number_conversion.cpp

□ Activity 1 Copy, study and test the program in `number_conversion.cpp`, and then write out its pseudocode.

□ Activity 2 Copy `number_conversion.cpp` to `number_conversion1.cpp` and bug it as follows:

a. Replace `static_cast<int>(3.97)` by `(int)3.97` in line 27.

This is one way to do a cast in C.

____

b. Replace `static_cast<int>(3.97)` by `int(3.97)` in line 27.

This is another way to do a cast in C.

____

c. Following the two lines (lines 54 and 55)

```
cout << int(4.37 * 10 + 0.5) / 10.0 << endl;
cout << int(4.61 * 10 + 0.5) / 10.0 << endl << endl;
```

add the two lines shown below, and indicate what these two new lines do:

```
cout << int(4.372 * 100 + 0.5) / 100.0 << endl;
cout << int(4.618 * 100 + 0.5) / 100.0 << endl << endl;
```

□ Activity 3 Evaluate the expression `double(15/10)` and also the expression `(double)15/10`. Show the two values, and say what the results tell you about the precedence of the C-style cast, considered as an operator.

□ Activity 4 First, predict the output of the code shown below by entering what you believe to be the output in the spaces provided following the code. Then write a suitable program that includes this code and test your prediction. Put the program in file called `number_conversion2.cpp`.

```
cout << 12/36 + int(6.58)*2.1 << "  "
     << -1.5 * double(-3 + 1/2)            << endl;
cout << 8.2 - 4 / 3 + 6 / 0.5 << "  "
     << int(3.8 / 2 + 1.6 * 3)             << endl;
cout << double(4 + 5 * 2) / double(-4 + 5 * 2) << endl;
```

○ Instructor checkpoint 9.1 for evaluating prior work

9.4.2 char_int_conversion.cpp illustrates how to convert between char and int data values by type casting

```
//char_int_conversion.cpp
//Illustrates some local character codes on your system
//and conversion between "int" and "char" data values.

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

int main()
{
    cout << "\nThis program shows the int and char values for some "
        "of the characters\nin your local character set, and how to "
        "convert between them.\n";
    cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');

    char charValue;
    cout << "\nEnter any upper or lower case letter, digit, "
        "or punctuation character: ";
    cin >> charValue;  cin.ignore(80, '\n');
    cout << "The internal integer value corresponding to " << charValue
         << " is " << static_cast<int>(charValue) << ".\n";
    cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');

    int intValue;
    cout << "\nEnter an integer value from the range "
         << static_cast<int>('A') << ".." << static_cast<int>('Z')
         << ": ";
    cin >> intValue;  cin.ignore(80, '\n');
    cout << "The capital letter corresponding to " << intValue
         << " is " << static_cast<char>(intValue) << ".\n";
    cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');

    cout << "\nEnter an integer value from the range "
         << static_cast<int>('a') << ".." << static_cast<int>('z')
         << ": ";
    cin >> intValue;  cin.ignore(80, '\n');
    cout << "The lowercase letter corresponding to " << intValue
         << " is " << static_cast<char>(intValue) << ".\n";
    cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');

    cout << "\nEnter an integer value from the range "
         << static_cast<int>('0') << ".." << static_cast<int>('9')
         << ": ";
    cin >> intValue;  cin.ignore(80, '\n');
    cout << "The digit character corresponding to " << intValue
         << " is " << static_cast<char>(intValue) << ".\n";
    cout << endl;

    cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');
}
```

9.4.2.1 What you see for the first time in char_int_conversion.cpp

Casting between character and integer values

- Casting a `char` value to an `int` value

  When this type of cast is done, the resulting integer value will be the internal integer representation of that character in the character code set of your computing system.

- Casting an integer value (i.e., an `int` value) to a `char` value

  This, of course, is just a cast "in the other direction".

9.4.2.2 Additional notes and discussion on char_int_conversion.cpp

Different computers may use different character sets.

On every computing system, communication between the human using it and the system itself is based on some particular *character set*. All systems do not use the same character set, of course. The most widely used character set (at least in North America) is ASCII[1], but there are others, like EBCDIC[2], which is used in the *mainframe* world of IBM[3].

The characters themselves are what the human user sees when information is displayed by the computer, on a screen or on paper from a printer, for example. But, internally, each of the characters is represented by a small positive integer.

There is no universal agreement on which integer should represent which character, and, for that matter, no agreement on exactly what the character set should contain. Hence the existence of several character sets.

However, you *would* like to be able to count on the upper case letters of the alphabet forming a contiguous sequence, and the lower case letters and the numerical digit characters doing the same, and this is true in ASCII at least, and in most other commonly-used character sets as well.

In any case, the program in `char_int_conversion.cpp` should exhibit the same behavior on all systems.

Robust programs

The program is *not robust*, which means that it does *not* check the input entered by the user to see if the user complied with instructions, so you may want to experiment a little just to see what happens when you enter values other than those asked for by the program.

By the way, this lack of robustness with respect to input is typical of most of the programs that we write. What this means, in general, is that a user of one of our programs is expected to respond appropriately to the prompts for input provided by the program. If the user does not do this, the program may or may not do something sensible, but the program is not responsible for what happens.

Making programs respond sensibly to whatever a user might do when the program is running requires a great deal of additional effort on the part of the program developer. In fact, in many commercial programs it may well be the case that most of the code is devoted to just this problem of dealing with the

[1] ASCII = American Standard Code for Information Interchange

[2] EBCDIC = Extended Binary-Coded Decimal Interchange Format

[3] Either you know what IBM stands for, or you don't need to know, or you don't care.

user, rather than doing whatever it is that the program is designed to accomplish for the user.

9.4.2.3 Character codes on your system

In the question-and-answer boxes which follow, you are asked to fill in some information about the character codes used on your system[4].

It’s important to remember that a single character, like the capital letter A, or the escape sequence `\n`, must be indicated to C++ by enclosing the character in single quotes, as in `’A’` or `’\n’`.

Answer

What character code scheme is used on your computing system?

Answer

What integer codes correspond to the upper case letters `’A’, ’B’, ... ’Z’`?

Answer

What integer codes correspond to the lower case letters `’a’, ’b’, ... ’z’`?

Answer

What integer codes correspond to the digit characters `’0’, ’1’, ... ’9’`?

Answer

What integer codes correspond to the “control characters”?

Answer

In particular, what integer code corresponds to the *tab character* (i.e., `\t`)?

[4] See Appendix B for details if your system uses ASCII, the most commonly-used character scheme on North American computers.

9.4.2.4 Follow-up hands-on activities for char_int_conversion.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `char_int_conversion.cpp`.

□ Activity 2 Copy `char_int_conversion.cpp` to `char_int_conversion1.cpp` and bug it as follows:

a. Replace `cin >> charValue` with `cin.get(charValue)` in line 19.

____

b. When the program is running, and asks for input of a single character value, enter several character values.

____

c. Replace all casts using `static_cast` with C-style casts. In simple situations like we have here, C-style casts are still used much of the time.

____

○ Instructor checkpoint 9.2 for evaluating prior work

□ Activity 3 Design and write a program that does the following things. First, it asks the user to enter a capital letter. Then it asks the user to enter a "shift value", which may be a positive integer, or a negative integer. The program then "shifts" the capital letter entered a number of spaces to the right or left within the alphabet (depending on whether the shift value entered was positive or negative), and displays the character at the new position. The number of positions moved is, of course, the absolute value of the shift value. Put your program in a file called `shift_letter.cpp`. *Note that there are a number of things to think about in this problem. For example, what will your program do if the shift value entered by the user causes a shift to a position that is "off the end" of the alphabet?*

○ Instructor checkpoint 9.3 for evaluating prior work

Based on shell.cpp

□ Activity 4 Design, write and test a program that will allow the user to convert an integer code to the corresponding character or vice versa (i.e., a character to the corresponding integer code), or a real number to the corresponding (rounded) integer. Assume that only integers that represent valid character codes will be entered by the user when a corresponding character is desired. The program must continue to do these conversions until the user quits. Put your program in a file called `shell_conversion.cpp`.

○ Instructor checkpoint 9.4 for evaluating prior work

Module 10

A second look at program development: structure diagrams and tracing

10.1 Objectives

- To understand what is meant by a *design tree diagram* for a program.
- To understand what is meant by a *trace* of a program, and to learn how to perform a trace of a simple program.
- To understand what is meant by a *named constant*, when named constants should be used, and how to define one.

10.2 List of associated files

- `shopping_list.cpp` contains a simple "application program" which prints a "shopping list".

10.3 Overview

In this Module we come back to the process of program development and look at the notion of *structure diagrams*, in particular the *design tree diagram*, and how it relates to the idea of top-down design with step-wise refinement in the context of a "mini-application". We shall also examine what it means to *trace* a program, and learn how tracing can help to discover logic (run-time) errors.

10.4 Sample Programs

10.4.1 shopping_list.cpp contains a simple "application program" that prints a shopping list

```
//shopping_list.cpp
//Illustrates a very small "application program"

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

int main()
{
    const double TAX_RATE = 0.15;

    cout << "\nThis program computes costs for a number of computer "
        "related items.\nThe user must enter the number of each item "
        "required, and the price per item.\nThe tax is fixed at "
        << TAX_RATE * 100 << "%, and the total cost includes tax.\n\n";

    int numberOfComputers, numberOfPrinters;
    double pricePerComputer, pricePerPrinter;

    cout << "Enter the number of computers to buy: ";
    cin >> numberOfComputers;
    cout << "Enter the price per computer: ";
    cin >> pricePerComputer;
    cout << "Enter the number of printers to buy: ";
    cin >> numberOfPrinters;
    cout << "Enter the price per printer: ";
    cin >> pricePerPrinter;
    cin.ignore(80, '\n');

    double costOfComputers = numberOfComputers * pricePerComputer;
    double costOfPrinters = numberOfPrinters * pricePerPrinter;
    double tax =  (costOfComputers + costOfPrinters) * TAX_RATE;
    double grandTotalCost = (costOfComputers + costOfPrinters) * (1 + TAX_RATE);

    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);
    cout << setprecision(2);

    cout << "\n\n"
        "Item           Number       Price        Total\n"
        "Name          of Units     Per Unit      Cost\n"
        "----------------------------------------------\n"
        "computers" << setw(10) << numberOfComputers
                    << setw(16) << pricePerComputer
                    << setw(11) << costOfComputers
        << "\nprinters" << setw(11) << numberOfPrinters
                        << setw(16) << pricePerPrinter
                        << setw(11) << costOfPrinters
        << "\ntax" << setw(43) << tax
        << "\n----------------------------------------------\n"
        << "Grand total cost of all items ... "
        << setw(12) << grandTotalCost
        << endl << endl;
}
```

10.4.1.1 What you see for the first time in shopping_list.cpp

- A "mini-application" which, though very small, contains many of the components of a "real-world" application program:

  Features found in any "application program"

  - A description of itself that is displayed when it runs
  - Both constant and variable quantities
  - The need to input data from outside the program itself
  - Computations involving this input data
  - Output of both the input data and the results computed by the program from that data

  It is *always* a good idea for a program to *echo* its input data. This is the term frequently used to mean "to display the input somewhere in the output". Doing this gives the user an opportunity to check whether what the user thought was entered was in fact received by the program, and is particularly important in a "real-world" application.

  Echoing input in the output

- The reserved word `const`

  `const`

- Use of a programmer-defined *named constant* (namely, `TAX_RATE`)

  Named constant

  Note the *capitalization convention* for named constants: all upper case letters, with an *underscore character* (_) separating words in the name). A named constant should be used for any quantity that must not change when a program runs, and for quantities that may change over time but very infrequently (such as a tax rate, as in this program). There are two major advantages in using a named constant: first, preventing the value from being inadvertently altered somewhere in the program; and second, if it must be changed at some point the change needs to be made in one place only (the one place where it is defined, not every place it is used).

  Capitalization convention for named constants

  Advantages of using named constants

10.4.1.2 Additional notes and discussion on shopping_list.cpp

Recall that all sample programs that we have shown up to this point have had as their main focus one or more new features of C++, or were designed as a vehicle for practice with input and/or output. Our ultimate goal, however, is to write useful programs that help us solve real-world problems. To write such programs, you of course need to understand the syntax and semantics of various C++ language constructs, but no less critical are the need to understand how to put these "low-level" language constructs together in such a way that they can accomplish something useful, and the need to keep track of what you are doing during the process. Top-down design, which we have already discussed, is one such tool. The following subsections discuss some additional tools for helping you with these organizational tasks during the program development process.

Putting it all together is the hard part, not learning the C++ details.

10.4.1.3 More on pseudocode

We introduced the notion of *pseudocode* for describing what a program does in Module 2. That was shortly after we introduced the notion of a program itself, and we have been asking you ever since, in the hands-on activities, to practice writing pseudocode for each new sample program that you have encountered.

You've been putting the cart before the horse!

By now you should be comfortable with the idea of pseudocode and how it corresponds to actual code, and it's time to make an important observation: Thus far your use of pseudocode has been the reverse of the way it should normally be used!

That is, in actual practice we do *not* write actual code first and *then* write the corresponding pseudocode. Instead, we write pseudocode *before* writing actual code, as one of the aids to help us design and write that code properly and thus produce correct programs. In fact, if we cannot (or will not, or just do not) write pseudocode to describe what we want a program to do, there is little hope that we shall be able to write actual code to perform the task.

Now we expect you to put the horse before the cart!

From now on, you will be expected to write pseudocode for your programs as part of the *design process*. In other words, the instructions, "Design and write a program to ... " *implicitly* include the expectation that you *will* write the necessary pseudocode for each task to be performed by your program, *before* you write the actual code.

But don't forget what got you here!

However, it is also important to continue the practice of writing pseudocode for pre-existing code (the sample programs, for example), since this will provide ongoing and very valuable experience and practice.

10.4.1.4 Design tree diagrams and top-down design

Structure diagrams, the design tree diagram in particular

Another valuable tool to help us picture the structure of a program is the *design tree diagram*, also called a *structure diagram*, though this last term is somewhat more generic and we prefer the former.

A design tree diagram for the program in `shopping_list.cpp` is shown in Figure 10.1. Such a diagram reflects the top-down design of a "structured" program. This particular diagram has only two "levels": `main` at the top (level 0), and each of the "logical chunks" of code in `main` represented by a descriptive phrase at the next level down (level 1).

In more complicated situations, which we will see later, there will of course be more levels, and the descriptive phrases representing the "logical chunks" of code will be replaced by the names of *programmed-defined functions*, when our programs come to contain more programmer-written functions than `main` alone.

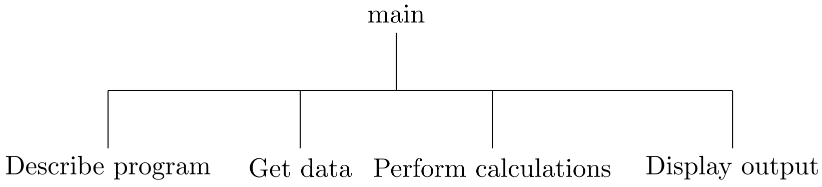

Figure 10.1: Design Tree Diagram for `shopping_list.cpp`

10.4.1.5 Tracing a simple program "by hand"

Tracing is a very useful skill.

Being able to *trace* a program is a skill that will be useful as long as you continue to write programs. Since the basic principle remains the same, irrespective of program complexity, it is a good idea to make sure that you learn what is involved in performing a program trace within the context of a simple situation like that given here.

Reason for tracing

First, note that the main reason for doing a trace is this: Your program compiles, links and runs, but either it crashes or does not produce the correct output, and you have been unable to discover the problem by studying the source code. This means that you have to examine in detail the effect of each line of code. In other words, you have to perform a *trace* of the program.

In a simple "hand trace", you do this by recording, on paper, and using a template like that shown in Figure 10.2, the value of each variable and/or expression of interest (which may well be *all* variables and expressions, at least in simple programs) after the execution of each line of code.

Executable	Value of each variable and expression of interest	Output
Statement	(after the statement finishes executing)	(if any)
	`v1 v2 v3 v4 ... e1 e2 ...`	
statement1	values of the above after execution of statement1	
statement2	values of the above after execution of statement2	
statement3	values of the above after execution of statement3	output
...	...	

Figure 10.2: Typical Style of a Generic Tracing Form

These values thus determined are computed by mental or hand calculation, *not* by the computer. The point is, to compare what *you* think the values should be with what the *computer* knows they are.

Compare your trace results with actual program output.

Once you have completed the template, of course, you run the program and compare your predictions with the program output and try to resolve any discrepancies. You will often want to add extra output statements to your program to display the values of variables or expressions that the program would not normally display in its output. (We discuss this at more length later when we talk about *embedded debugging code* in Module 14.)

10.4.1.6 Tracing a simple program with the help of your system's "debugger" (optional)

What we discussed in the previous subsection was a *hand-trace* of a program, sometimes also called a *pencil-and-paper trace*, or *manual trace*. Most systems also have a *software debugger*, i.e. a program that effectively performs a trace for you. We do not discuss such programs in detail here because they are highly system-dependent. However, you may wish to explore the capabilities of your local debugger, if you have one readily available.

If there is a debugger available on your system, how do you use it to trace a C++ program and help you find run-time or logic errors? Or, where can you get information on how to use your system's debugger?

Answer

10.4.1.7 Follow-up hands-on activities for shopping_list.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `shopping_list.cpp`.

□ Activity 2 Copy `shopping_list.cpp` to `shopping_list1.cpp` and bug it as follows:

a. Omit the keyword `const` in the definition of the constant `TAX_RATE`.

____

b. Omit the keyword `double` in the definition of the constant `TAX_RATE`.

____

□ Activity 3 Copy `shopping_list.cpp` to `shopping_list2.cpp` and make all necessary modifications to the copied program to add two items to the shopping list: modems and sound cards.

○ Instructor checkpoint 10.1

□ Activity 4 In this activity you are to complete—perhaps with the help of your instructor, and using a full-page template having the same design as the one shown above in Figure 10.2—the trace template for the code computing the weighted average calculation shown below.

Begin by entering the code into a suitable enclosing program in a file called `bad_average.cpp`, and choose some suitable test data. Do not look too closely at the code for now. Compile, link and run the program. The idea, of course, is that the code is supposed to compute an average mark out of 10, given two other "raw" marks as input, and you are (or should be) suspicious of this code because of the results it is giving you.

Now complete the trace, and compare the program output with your trace results. Can you identify the problem(s) with the code? In this case the problems may have been obvious before you started. In other cases you may not be so lucky!

```
const int WEIGHT = 0.10;
float rawMark1, rawMark2, average, weightedAverage;
cout << "This program computes the weighted "
     << "average of two marks.\n";
cout << "Enter two marks out of 100: ";
cin >> rawMark1 >> rawMark2;  cin.ignore(80, '\n');
average = rawMark1 + rawMark2 / 2;
weightedAverage = WEIGHT * average;
cout << "Mark1  Mark2  WeightedAverage\n";
cout << setw(5)  << rawMark1 << setw(7) << rawMark2
     << setw(13) << weightedAverage << endl;
```

○ Instructor checkpoint 10.2

□ Activity 5 Copy `shopping_list.cpp` to `shopping_list3.cpp`, change `double` to `int` in the definition of `TAX_RATE`, and repeat the steps of the previous activity.

○ Instructor checkpoint 10.3

□ Activity 6 Design, write and test a program that will perform like the one you wrote for `shopping_list2.cpp` in Activity 3 above, except that it is based on `shell.cpp` and therefore allows the user to produce as many computer "configuration lists" as desired before quitting, with different quantities and prices for each list of (the same) items. Put the program in a file called `shell_shopping.cpp`.

Based on shell.cpp

○ Instructor checkpoint 10.4

Module 11

The bool data type and conditional expressions

11.1 Objectives

- To understand the `bool` data type, and its two *boolean constant* data values `true` and `false`.

  `bool`, `true` and `false` *are reserved words.*

- To learn how to declare and use a *boolean variable* (a variable of data type `bool`).

- To learn how to use the six *relational operators*, which are also sometimes called the *comparison operators*, since each one is used to *compare* two quantities: `<`, `<=`, `>`, `>=`, `==` and `!=`

  Six new (relational) operators

- To understand and learn how to use the three *boolean operators*, which are also sometimes called the *logical operators*, since they help us to construct *logical expressions*: `!`, `&&`, and `||`

  Three more new (boolean) operators

- To understand the precedence of the relational and boolean operators, and how their precedence relates to the precedence of the previously-studied arithmetic and assignment operators. The precedence of the five arithmetic, six relational, and three boolean operators is shown below in a revised table of precedence which you should study carefully.

  Having more operators means we need a revised precedence table.

```
highest     !
            *    /    %
            +    -
            <    <=   >   >=
            ==   !=
            &&
            ||
lowest      all assignment operators
```

Understanding terminology helps to avoid confusion.

- To understand what is meant by each of the following terms: *conditional expression*, *relational expression* and *boolean expression*

- To understand the difference between a *simple* conditional (boolean) expression and a *compound* conditional (boolean) expression.

Try to get a handle on this.

- To understand the relationship, in C++, between the boolean values `true` and `false` and the `int` values `1` and `0`.

`boolalpha` *manipulator*

- To understand what happens (by default) when you attempt to display a boolean value, and to learn how to alter that behavior by using the `boolalpha` manipulator.

This can come in handy.

- To understand what is meant by *short-circuit evaluation* of a (compound) boolean expression.

`typedef` *will be more useful later.*

- To learn the C++ reserved word `typedef` and understand how it may be used to define your own "boolean data type" if your C++ implementation does not yet[1] include the data type `bool`.

11.2 List of associated files

- `bool_data.cpp` illustrates the `bool` data type, a boolean variable, and the evaluation of expressions which have boolean values

There is only one sample program in this Module. This should not suggest that the topics of the `bool` data type, boolean values and variables, and expressions whose values are boolean deserves little attention. On the contrary, they are central players in the power structure of any programming language. It's just that it's rather difficult and artificial to illustrate them in isolation. They really need to be appreciated in the context in which they are most useful, i.e., in decision-making (Module 12 and Module 14) and looping (Module 13 and Module 14). Nevertheless, it should be helpful to have at least this much exposure to these topics, in isolation, so that you don't have all those other things to think about during your first encounter with using them "under fire".

[1] Standard C++ *does* include the `bool` data type, and at the time of writing so would most C++ implementations.

11.3 Overview

In subsequent Modules many of the sample programs you will see and the programs that you will write will have to make decisions of one kind or another. Those decisions will be based, in the simplest cases, on whether a single *simple condition* is true or false. In more complex situations, the decision may be based on the truth or falsity of several conditions in some combination. Such conditions often take the form of an expression to be evaluated, in particular a *relational expression* or *boolean expression* formed by using the *relational operators* and/or the *boolean operators*. This Module is designed to introduce you to these concepts and to provide some exposure to them before you actually begin to use them in context.

Let's take a moment and try to clarify some of the terms introduced at the beginning of this Module, since there is quite a bit of terminology associated with this topic, and there is plenty of opportunity for confusion.

First, any one of the terms *conditional expression*, *boolean expression* or *logical expression* may be used in a very generic way to refer to *any* kind of *condition*, i.e., to any expression which has a *boolean value* of either `true` or `false` when evaluated. That is, the terms *boolean expression*, *conditional expression* and *logical expression* are often used interchangeably. And any one of these terms may be used to mean *any* expression that contains one or more operators and operands, and in which each operand is itself a constant, a variable or an expression whose value is boolean.

A *relational expression* refers to a *simple* conditional expression, i.e., one containing just *one* relational operator and *two* operands (such as `(x <= 2.5)` or `(ch > 'A')`).

The term *compound boolean expression* refers to an expression of virtually arbitrary complexity, but which contains at least two or more "smaller" expressions, each of which would by itself produce a boolean value when evaluated. The smaller expressions are connected by the `&&`, `||` and `!` boolean operators to form the larger, or "compound" expression. The complete expression will also, of course, have a boolean value. A simple example, containing only two "smaller" expressions, is this one, taken from the sample program:

```
7 == 3  &&  -5.3 <= 3.5
```

Beginning programmers are often surprised when they attempt to display a boolean value and do not see the expected `true` or `false`. The reason is that, by default, the boolean values are converted to their integer equivalent values of `1` and `0`, respectively, before being displayed. To avoid this, the manipulator `boolalpha` can be inserted into the output stream before the boolean values are displayed, after which the expected (boolean) values will be output.

The effect of inserting the manipulator `boolalpha` into the output stream is *persistent*, which means that from that point on boolean values will be output as `true` and `false`. To "turn off" this effect, and go back to `1` and `0`, you can insert the corresponding manipulator `noboolalpha`. The use of both manipulators is illustrated in `bool_data.cpp`.

11.4 Sample Programs

11.4.1 bool_data.cpp illustrates the bool data type, a boolean variable, and the evaluation of expressions which have boolean values

```
//bool_data.cpp
//Illustrates the "bool" data type, boolean variables,
//and evaluation of boolean expressions.

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

int main()
{
    cout << "\nThis program illustrates the boolean data type, use of "
        "boolean variables,\nand the evaluation of boolean expressions. "
        "Be sure to study the source code.\n\n";


    cout << "Some relational expressions involving "
        "just constants, and their values:\n"
        "Expression: 7 == 3   'A' != 'a'   4.6 > 9.3\n"
        "Value: " << setw(8)  << (7 == 3)
                  << setw(11) << ('A' != 'a')
                  << setw(13) << (4.6 > 9.3) << endl << endl;


    cout << "Some compound boolean expressions involving "
        "just constants, and their values:\n"
        "Expression: 7 == 3  &&  -5.3 <= -3.5      "
        "2 == 6  ||  'A' != 'a'    \n"
        "Value: " << setw(14) << (7 == 3  &&  -5.3 <= 3.5)
                  << setw(30) << (2 == 6  ||  'A' != 'a') << endl
        << boolalpha <<
        "Value: " << setw(14) << (7 == 3  &&  -5.3 <= 3.5)
                  << setw(30) << (2 == 6  ||  'A' != 'a') << endl
        << noboolalpha <<
        "Value: " << setw(14) << (7 == 3  &&  -5.3 <= 3.5)
                  << setw(30) << (2 == 6  ||  'A' != 'a')
        << endl << endl;


    int menuChoice;
    bool choiceIsValid; //Declaration of a "boolean variable"

    cout << "Enter a \"menu choice\" value from 1 to 5: ";
    cin >> menuChoice;  cin.ignore(80, '\n');  cout << endl;

    choiceIsValid = (menuChoice >= 1  &&  menuChoice <= 5);
    cout << "Your menu choice was valid (1) or invalid (0), "
        "depending on this value: " << choiceIsValid;
    cout << endl << endl;
}
```

11.4.1.1 What you see for the first time in bool_data.cpp

- Evaluation of *relational expressions* containing *relational operators*
- Evaluation of *boolean expressions* containing *boolean operators*, including *compound boolean expressions*
- A *compound boolean expression* that is evaluated by the technique of *short-circuit evaluation* (terminating the value of the expression as soon as the answer is known, which may be before all parts of the expression have been evaluated)
- A *boolean variable*, and assignment to a *boolean variable*

11.4.1.2 Additional notes and discussion on bool_data.cpp

The `bool` data type is relatively new in C++.

There are some background facts about the `bool` data type that you should know. First, the C++ language standard requires this data type to be included as part of the language definition. As we have mentioned before, compilers always take some time to comply with the latest standard, though at the time of writing most compilers would contain this data type. The next subsection tells you what to do if your compiler does not have a `bool` data type, and you want to "create" your own version. It's unlikely that you will have to do this, but the discussion is interesting in its own right, since it's a nice simple way to become acquainted with the `typedef` keyword.

Be aware of this issue of "backward compatibility".

Since the `bool` data type is a relatively late addition to the C++ language, clearly it is not "necessary". The reason for this is that C++ was designed to be backward compatible with the C language, in which the value 0 is interpreted as `false`, and the value 1 (or, in fact, *any* non-zero value) is interpreted as `true`. This should help to explain the way we show you how to define "your own" boolean data type (if necessary) in the following subsection.

A major reason for having an explicit `bool` data type in any programming language is the same reason that we have some other language features: it is convenient, and using it can help to make our programs both more readable and "safer". But the C approach—1 is true and 0 is false—continues to work in C++, as the program in `bool_data.cpp` also demonstrates.

Programming style: operators and whitespace

A style note: One of the things you will observe in the sample program is the two blank spaces that appear on either side of the boolean operators `&&` and `||`. Usually we place only a single space on either side of an operator (and sometimes, to emphasize that the operands go with the operator, we may even *leave out* the surrounding spaces). So why have two spaces on either side in this instance? The reason is to emphasize what the left and right operands of the boolean operator are. We could, of course, do this with parentheses as well, and an argument can be made that this is how we *should* do it. However, using the double surrounding spaces is meant to have the same effect, and one could argue that, being less cluttered, the code is more readable than the corresponding equivalent with the extra parentheses. It's a personal style choice.

Note that omission of the inner parentheses here is only possible because the relational operators have a *higher* precedence than the boolean operators, which is *not* the case in all programming languages. And that would be the best reason for including parentheses in all such expressions, without regard to the language in use at the time!

11.4.1.3 How to define your own bool data type in case your C++ implementation does not have the real thing (optional)

`typedef` *creates a synonym for a pre-existing data type.*

This subsection may be of interest even if your C++ compiler *does* have the `bool` data type. For one thing, you may possibly encounter an older C++ compiler that does not yet know about the `bool` type. For another, you also get to see how to use the C++ keyword `typedef` to define a new type in terms of an already existing type, a useful piece of information in its own right.

If your compiler does *not* contain the `bool` data type, then including the following three lines in your program will provide a definition for a bool data type that you can use in the same way you would use the built-in `bool` type if it were present.

```
typedef int bool; //Establishes "bool" as a synonym for "int".
const bool true = 1;  //Defining "true" and "false" like this
const bool false = 0; //maintains backward compatibility with C.
```

Since this definition effectively simulates the definition required by the C++ standard, if at any later time your compiler is updated then simply removing the three-line definition and recompiling should permit your programs to work properly, without making any other changes.

On the other hand, if your C++ compiler already includes a definition of the `bool` data type, then including your own definition like that shown above will cause a *name-conflict error* because of the duplication.

`typedef` *is also a reserved word.*

Note that the name `bool` is really just a synonym for `int` and in particular for the two special integer values `1` and `0` (which themselves are given the special names `true` and `false`, respectively). That is, the keyword `typedef` has the effect of defining a synonym for an already existing data type, when used in this way.

Definition placement for a programmer-defined data type

The definition of `bool` should be placed *outside* and *prior to* the `main` function. In a simple program like that of `bool_data.cpp` it could in fact be placed inside `main`, and things would work in the same way. However, it is usual to place it outside and prior to `main` so that it becomes “globally” available to other functions besides `main`, if there are any, and though there are none now there certainly will be, later on. When the definition is placed inside `main` it becomes “hidden”, and available only to `main`. But much more on all of that later.

11.4.1.4 Follow-up hands-on activities for bool_data.cpp

□ Activity 1 Copy, study and test the program in `bool_data.cpp`. Don't bother with the pseudocode on this occasion. Take particular note of those values actually displayed at the places where the program outputs the values of the boolean expressions it has evaluated. How do you explain what you see?

____

____

____

____

□ Activity 2 Copy `bool_data.cpp` to `bool_data1.cpp` and bug it as follows:

a. Remove the parentheses from `(7 == 3)` in line 19.

   ____

b. Remove the parentheses from `(2 == 6  ||  'A' != 'a')` in line 29.

   ____

c. Remove the parentheses from the statement that assigns a value to the variable `choiceIsValid`.

   ____

□ Activity 3 Which expressions in this program, if any, are evaluated by short-circuit evaluation?

____

○ Instructor checkpoint 11.1 for evaluating prior work

Module 12

Making decisions with selection control structures (no looping)

12.1 Objectives

- To learn about the following C++ reserved words: `if`, `else`, `switch`, `case` and `break`.

  You will have seen `switch`, `case` and `break` already, in one particular context, if you have covered Module 6 and its associated “shell” program.

- To understand when and how to use the C++ `if` statement, which we refer to in normal text in hyphenated form as an “if-statement”.

- To understand when and how to use the C++ `if...else` statement, which we refer to in normal text as an “if...else-statement”.

- To understand when and how to use a C++ `switch` statement (switch-statement) including the associated use of the `case` *selector* and `case` *labels* (case-selector and case-labels), and the `break` statement (break-statement).

  You will have already seen one typical use of the switch-statement, with a case-selector, case labels, and break-statements if you have covered the material in Module 6.

- To understand the relationship between a *nested if-statement* and a *switch-statement.*

- To understand *sequential decision-making.*

- To understand *nested decision-making.*

- To understand the importance of analyzing carefully the nature of any decision to be made by a program before choosing a particular selection statement to implement the decision-making process.

12.2 List of associated files

- `if.cpp` illustrates the if-statement, the if...else-statement, sequential if-statements, and a nested if-statement.
- `switch.cpp` illustrates the switch-statement.
- `wages.cpp` illustrates conditional computation of wages.

12.3 Overview

Programs must be able to decide what to do next.

In this Module you deal with programs[1] that are able to make various kinds of decisions, i.e., programs that at various times during their execution are able to decide which of several possible tasks to perform next. There are many occasions when a program must be able to decide whether or not to execute a given statement, and other situations when a choice must be made from two or more alternatives. This notion of decision-making is also called *selection* or *branching*, and the programming language constructs that permit these decisions to be made by a program are often referred to as *selection control structures*, since they help to "control" the "flow of the action" as a program executes. We will also examine situations in which *sequential decision-making* (one decision after another) and *nested decision-making* (one decision within another) are necessary to express the logical flow of a situation.

12.4 Sample Programs

The first two sample programs in this Module (`if.cpp` and `switch.cpp`) contain quite a number of illustrations of decision-making, one of each of the various kinds outlined above, and you should study them very carefully.

The third and final sample program (`wages.cpp`) performs some decision-making in a "practical" context, though a very simple one.

[1] Any program that appeared in a Module prior to this one consisted entirely of sequentially executed statements, except for the "shell" program of Module 6 and any others based on it that you may have written for yourself, there or later.

12.4.1 if.cpp illustrates typical uses of the if-statement, the if...else-statement, sequential if-statements, and a nested if-statement

```
//if.cpp
//Illustrates an if-statement, an if...else-statement,
//a sequence of if-statements, and a nested if-statement.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program analyses a lecture section and a mark.\n\n";

    int mark;
    char lectureSection;

    cout << "Enter A, B or C, followed by a mark in the range 0..100: ";
    cin >> lectureSection >> mark;  cin.ignore(80, ’\n’);
    cout << endl << endl;

     //An if...else-statement (control structure)
    if (lectureSection==’A’ || lectureSection==’B’ || lectureSection==’C’)
        cout << "OK, your lecture section is " << lectureSection << ".\n";
    else
        cout << "That’s not a valid lecture section.\n";

    if (mark >= 50) //An if-statement (control structure)
    {//if-statement body with more than one statement requires braces
        cout << "\nYou have received a passing mark.\n";
        cout << "And that was not an easy course!\n";
    }

    //Here is a sequential-if construct (sequence of if-statements):
    if (mark >= 90) cout << "That is in fact a top-notch mark!\n";
    if (mark >= 80) cout << "Congratulations on doing so well!\n";
    if (mark >= 65) cout << "You may proceed to the next course.\n";
    if (mark >= 50  &&  mark < 65)
        cout << "You should probably not proceed.\n";
    if (mark < 50) cout << "You may not proceed.\n\n";

    //Here is a nested-if construct (nested if-statement):
    if (mark >= 80)
        cout << "Your letter grade is an A.\n";
    else if (mark >= 70)
        cout << "Your letter grade is a B.\n";
    else if (mark >= 60)
        cout << "Your letter grade is a C.\n";
    else if (mark >= 50)
        cout << "Your letter grade is a D.\n";
    else
        cout << "Your letter grade is an F.\n";
    cout << endl;
}
```

12.4.1.1 What you see for the first time in if.cpp

- The use of the truth or falsity of both *relational expressions* and *boolean expressions* to help make decisions

`if` *and* `else`

- The C++ reserved words `if` and `else`

Decision-making takes various forms ...

- The use of a simple if-statement, which can be used to decide whether or not to perform a particular task
- The use of an if...else-statement, which can be used to decide which *one* of *two* alternative tasks to perform
- The use of a sequential-if construct, which can be used to perform *zero or more* of several tasks
- The use of a nested-if construct, which can be used to perform either *none*, or *exactly one*, of several tasks

Formatting an if-statement

- Typical formatting of the if, if...else, sequential-if and nested-if control structures

12.4.1.2 Additional notes and discussion on if.cpp

You should study this program carefully, and note the nature of each decision being made. Note in particular that although this program evaluates several boolean expressions, it nevertheless does *not* explicitly use the `bool` data type.

Match the code segments from the sample program with the conceptual possibilities listed above.

You should be able to reconcile what the code in the program does with the description given in the preceding subsection of the kinds of decisions that each new decision-making construct is capable of making.

In the descriptions of the various constructs, a “task” may consist of a single C++ statement, or more than one statement, in which case the statements comprising the task to be performed need to be enclosed in braces to form a *block* (a group of statements enclosed in braces and treated as a single entity for some purpose).

12.4.1.3 Follow-up hands-on activities for if.cpp

□ Activity 1 Copy, study, test and then write the pseudocode for `if.cpp`. When testing, you should try to find enough different combinations of input values to produce all possible outputs from the program. Otherwise, how can you be sure the program works properly. Think of this as "putting the program through its paces".

□ Activity 2 Copy `if.cpp` to `if1.cpp` and bug it as follows:

a. Remove the parentheses from `(mark >= 50)` in line 25.

   ____

b. Remove the braces from around the body of the onlyl lif-statement that has more than one statement in its body.

   ____

c. Replace each double equal sign (`==`) with a single equal sign (`=`).[2]

   ____

□ Activity 3 Make a copy of `if.cpp` called `if2.cpp`, and modify the copy by removing the last `else` and the last `cout` statement from the nested-if construct. Note, and verify when you test it, that this changes this particular nested-if construct from one that chooses exactly one of the possible alternatives to a nested-if construct that may not choose any of the possible alternatives.

□ Activity 4 Make a copy of `if.cpp` called `if3.cpp`, and modify the copy in such a way that if the mark entered does not lie in the range 0..100, the program does not output any comment on the mark except for the message

```
A mark of ??? is not valid.
```

in which the `???` is to be replaced by the actual mark entered.

○ Instructor checkpoint 12.1 for evaluating prior work

[2] Further discussion of this very common programming error may be found at the end of the hands-on activities on page 129.

12.4.2 switch.cpp illustrates how to use the switch-statement

```
//switch.cpp
//Illustrates the switch-statement.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program gets a user’s opinion of Microsoft "
        "(1 = very low, 5 = very high).\n\n";

    int opinion;

    cout << "Enter a value from 1 to 5: ";
    cin >> opinion;

    switch (opinion)
    {
        case 1:
            cout << "That’s OK. I won’t tell Bill!\n";
            break;

        case 2:
            cout << "Not so high, eh? You and many other users!\n";
            break;

        case 3:
            cout << "Hmmm! A fence sitter on the subject, eh?\n";
            break;

        case 4:
            cout << "Oh yeah? What have they done for you lately?\n";
            break;

        case 5:
            cout << "The cheque from Bill is in the mail!\n";
            break;

        default:
        {
            if (opinion > 5)
                cout << "Your opinion is off-scale!!\n";
            else
                cout << "Now, now, now ... !\n";
        }
    }
    cout << endl;
}
```

12.4.2.1 What you see for the first time in switch.cpp

Once again, note that you will have seen `switch`, `case`, and `break` already if you have covered Module 6.

- The C++ reserved words `switch`, `case`, `break`, and `default`

`switch`, `case`, `break` *and* `default`

- The use of a switch-statement to decide which one of several alternative tasks to perform, including the use of a case-selector and case-labels to assist in making the choice

A switch-statement can be used for some multi-way decisions.

- The use of a break-statement to "break out of" a switch-statement
- The use of the (optional) *default option* in a switch-statement

  The default option is the one executed if the value of the case-selector does not match *any* of the case-labels. If there is no default option and the value of the case-selector does not match any of the case-labels, then the entire switch-statement has no effect.

`default` *option in a switch-statement*

- Another form of nested decision-making, produced by having an if...else-statement as one of the choices inside a switch-statement
- Typical formatting for a switch-statement

Formatting a switch-statement

12.4.2.2 Additional notes and discussion on switch.cpp

The switch-statement is a multi-way decision-making mechanism, used to decide which of several tasks to perform. It works by matching the value of the case-selector (the variable or expression in parentheses after the keyword `switch`) to one of a list of constant values (the case-labels). Each one of the constant values acts as a "label" for one of the possible tasks to be executed.

A break-statement must follow each of the tasks so that the program "breaks out of the switch-statement" and carries on with the first statement following the switch-statement. Otherwise, in addition to the task corresponding to the matching case-label, all tasks *subsequent to* the one containing the matching label of the switch-statement will be performed, at least until another break-statement is encountered. This so-called "fall-through" behavior *may* be what is required on occasion, but only rarely does it seem to crop up.

It's almost always an error to leave out the `break` *after any of the cases in a* `switch`.

It should be clear that there will be many situations in which relatively complex decisions (decisions within decisions, for example) need to be made. That is, there are many possible variations on the "nested decision-making" you see in `switch.cpp`, which only has one if...else-statement within a switch-statement.

Note that this program, like the previous one, does not explicitly use the `bool` data type.

12.4.2.3 Follow-up hands-on activities for switch.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `switch.cpp`.

□ Activity 2 Copy `switch.cpp` to `switch1.cpp` and bug it as follows:

a. Remove the parentheses in `switch (opinion)`.

____

b. Remove the default option, i.e., remove everything from (and including) the keyword `default` to the next closing brace `}`.

____

c. Remove *all* of the break-statements.

____

d. Replace `(opinion > 5)` by `(opinion > 7)` and also replace the line

```
case 5:
```

Note this syntax for multiple cases requiring the same action.

with the following three lines:

```
case 5:
case 6:
case 7:
```

and then re-compile, re-link and re-run with the test values 5, 6, 7 and 8. This illustrates the necessary syntax when you need to have the same task executed for two or more different case-labels.

____

□ Activity 3 Make a copy of `switch.cpp` called `switch2.cpp` and modify the copy so that the switch-statement is replaced with a nested-if construct.

□ Activity 4 Make a copy of `switch.cpp` called `switch3.cpp` and modify the copy so that the switch-statement is replaced with a sequential-if construct.

□ Activity 5 Which of the two do you think is likely to be the more "efficient", on average, the sequential-if construct or the nested-if construct, and why?

____

This exercise illustrates an important point. What is it?

□ Activity 6 Can you replace the nested-if construct in the sample program `if.cpp` you encountered previously in this Module with a sequential-if construct? If you can, make a copy of `if.cpp` called `if_switch.cpp` and modify the copy accordingly. If not, explain why not in the space given below.

____

____

____

____

○ Instructor checkpoint 12.2 for evaluating prior work

12.4.3 wages.cpp performs conditional computation of wages

```
//wages.cpp
//Illustrates the use of an if...else-statement in a "practical" context.

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

int main()
{
    const int    STANDARD_HOURS  = 8;   /* number of hours per day */
    const double WAGE_RATE       = 9.5; /* $ per hour              */
    const double OT_RATE         = 1.5; /* "time-and-a-half"       */

    cout.setf(ios::fixed, ios::floatfield);
    cout.setf(ios::showpoint);
    cout << setprecision(2);

    cout << "\nThis program computes a worker's wages for a single day.\n"
        "The wage rate is fixed at $" << WAGE_RATE << " per hour for a "
        "regular work day.\nTime and a half is paid for overtime, which "
        "is anything beyond " << STANDARD_HOURS << " hours.\nA worker "
        "gets paid in full for the last hour, even if the hour is not "
        "complete.\n\n";

    double numHoursWorked;
    double wages;
    bool overtimeWorked;

    cout << "Enter the number of hours worked: ";
    cin >> numHoursWorked;  cin.ignore(80, '\n');
    cout << endl;

    if (numHoursWorked > (int)numHoursWorked)
        numHoursWorked = (int)numHoursWorked + 1;

    overtimeWorked = (numHoursWorked > STANDARD_HOURS);
    if (overtimeWorked)
        wages = (STANDARD_HOURS * WAGE_RATE) +
                (numHoursWorked - STANDARD_HOURS) * WAGE_RATE * OT_RATE;
    else
        wages = WAGE_RATE * numHoursWorked;

    cout << "Total wages for " << numHoursWorked
         << " hours at this rate: $"  << wages << endl;
    cout << endl;
}
```

12.4.3.1 What you see for the first time in wages.cpp

- Use of C-style comments of the form `/* some_comment */`

  Because a C-style comment consists of all text that lies between the two characters `/*` and the next occurrence of the two characters `*/`, it follows that a C-style comment, unlike a C++ comment, can extend over multiple lines. The C-style comments you see in this sample program could be replaced by C++ comments. Later (in Module 15) you will begin to see situations in which C-style comments are necessary because of the placement of those comments.

12.4.3.2 Additional notes and discussion on wages.cpp

Follow-up use of concepts introduced briefly in Module 11

Although we mentioned them briefly before (in Module 11), here you see the *explicit* use of the `bool` data type and a *boolean variable* in a typical context for the first time. The program is also another instance of a "mini-application" similar to the shopping list program of Module 10, but this one, of course, involves some decision-making and shows you a typical use for an if...else-statement and a boolean variable.

One other feature shown by this program is the placing of the code for establishing real number formats (`cout.setf()` and `setprecision()`) and the definitions of the named constants before the program description code, since these items are actually used in the code that displays the program description.

Recall the advantages of using named constants.

Note again the use of named constants and how they are capitalized. And remember that if one or more of these values has to be changed then it needs to be changed in only *one* location—namely, in its definition—no matter how many places it is used.

12.4.3.3 Follow-up hands-on activities for wages.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `wages.cpp`.

□ Activity 2 Copy `wages.cpp` to `wages1.cpp` and bug it as follows:

a. Remove the first three lines in the `main` function.

   ____

b. Remove the first set of parentheses in the if...else-statement.

   ____

c. Remove the second set of parentheses in the if...else-statement.

   ____

d. Remove the third set of parentheses in the if...else-statement.

   ____

□ Activity 3 Make a copy of `wages.cpp` called `wages2.cpp` and modify it in such a way that the boolean variable is eliminated from the program, but the program continues to work in the same way. Comment on the advantage/disadvantage of having/not having a boolean variable in a situation like the one you see in this program.

____

____

____

____

○ Instructor checkpoint 12.3 for evaluating prior work

□ Activity 4 Design, write the pseudocode for, and then write the actual code for, a program that reads in from a user at the keyboard three letter grades—any one of which may be A, B, C, D or F—and then outputs a table showing all three grades input and the grade point average. The points assigned to each grade are: A = 4, B = 3, C = 2, D = 1, F = 0. Place your program in a file called `shell_gpa.cpp`. Compile, link, run and test it until it is working to your satisfaction.

This program could be based on shell.cpp, but doesn't have to be, if it only processes one set of grades.

Technically, the title of this Module promised "no looping", but what we really meant was that we would not introduce any new looping constructs while we were concentrating on selection. Since the next two activities do involve our old friend `shell.cpp`, which does contain a loop, we have not strictly kept our promise, but we hope you will agree we have kept it in spirit.

○ Instructor checkpoint 12.4 for evaluating prior work

□ Activity 5 Design, write and test a program that will perform like `wages.cpp`, except that it is based on `shell.cpp` and therefore allows the user to compute as many daily wages as required (one at a time) before quitting. Put the program in a file called `shell_wages.cpp`.

Based on shell.cpp

○ Instructor checkpoint 12.5 for evaluating prior work

Module 13

Repeating one or more actions with looping control structures (no selection)

13.1 Objectives

- To learn about the following C++ keywords: `while`, `do`, and `for`.
- To understand when and how to use a C++ *while-statement* (also called a *while-loop*).
- To understand when and how to use a C++ *do...while-statement* (also called a *do...while-loop*).
- To understand when and how to use a C++ *for-statement* (also called a *for-loop*).
- To understand *sequential looping* and learn how to implement it.
- To understand *nested looping* and learn how to implement it.
- To understand what is meant by the following terms and concepts:
  - *definite iteration*
  - *indefinite iteration* (or *conditional iteration*)
  - *loop condition*
  - *loop body*
  - *loop iteration*
  - *loop control variable*
  - *loop control variable initialization*

  - *loop control variable modification*
  - *pre-test loop*
  - *post-test loop*
  - *infinite loop*
  - *counter* and *counter-controlled loop*
  - *accumulator* and *accumulator-controlled loop*
  - *sentinel* and *sentinel-controlled loop*
  - *flag* and *flag-controlled loop*
  - *end-of-file character* and *end-of-file-controlled loop*

- To understand the conceptual differences between all of the various looping constructs, and to learn some guidelines for choosing the "best" loop for a given situation.

13.2 List of associated files

- `square_integers.cpp` calculates squares of integers using a while-loop.
- `sum_integers.cpp` computes sums of integers using a do...while-loop.
- `display_sequences.cpp` displays character and numerical sequences using for-loops, and illustrates sequential loops.
- `rounded_average.cpp` computes the rounded integer average of all integer values on each input line, and illustrates nested loops.
- `draw_box.cpp` displays an empty box with user chosen size, border and position.

13.3 Overview

Programs must be able to repeat one or more actions.

In the previous Module we discussed one major departure from the exclusively sequential execution we had been used to seeing in our programs by introducing the notion of decision-making. In this Module we extend the power of our programs in another direction by learning how to make our programs repeat one or more executable statements when the need arises. This whole idea of *repetition* or *looping* in a program is one of the central ideas in computer programming and, though conceptually straightforward, it has more than its share of traps and pitfalls that you must learn to avoid.

In this Module will also examine situations in which *sequential loops* (one loop after another) and *nested loops* (one loop within another) are necessary to express the logical flow of a situation.

13.4 Sample Programs

13.4.1 square_integers.cpp calculates squares of integers using a while-loop

```
//square_integers.cpp
//Displays a table of integer squares.
//Illustrates a count-controlled while-loop.

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

int main()
{
    cout << "\nThis program displays a table of integer squares from 1\n"
        "up to a maximum value chosen by the user.\n\n";

    int numberOfSquares;

    cout << "Enter the maximum value to be included in the table: ";
    cin >> numberOfSquares;  cin.ignore(80, '\n');

    cout << "\n\n   n     n * n" << endl;

    int n = 1;
    while (n <= numberOfSquares)
    {
        cout << setw(4) << n << setw(10) << n * n << endl;
        ++n;
    }
    cout << endl;
}
```

13.4.1.1 What you see for the first time in square_integers.cpp

- A *while-loop*, which contains:

  while-loop

  - The C++ reserved word `while`

    `while`

  - A *conditional expression*

    This is the expression in the (required) parentheses following the keyword `while`. It controls whether the loop begins and, if it begins, when it ends.

    Condition tested before body of loop executed

  - A *loop body*

    These are the statements to be executed on each "pass" through the loop, i.e., during each *loop iteration*. If there is more than one statement in the body of the loop, those statements must be enclosed in braces. The loop body must contain at least one statement that modifies the loop condition in the appropriate "direction" (toward a "termination value") if the loop is to terminate.

    Loop body must modify loop control variable

Formatting a while-loop

- The typical formatting for a while-loop

  Note in particular the level of indentation of the statements in the loop body, which is one level deeper than the level of the while-statement itself.

Loop control variable

- A *loop control variable* (`n` in this case) whose value actually determines whether the loop ever begins executing at all, as well as when the loop stops executing, and, like all loop control variables, it must be

  – *initialized* before the loop begins

    That is, it must be given a starting value.

  – *modified* within the body of the loop

    That is, it must have its value changed appropriately within the body of the loop so that the loop condition eventually becomes false and the loop terminates. Otherwise you will have an *infinite loop*, i.e., a loop that never terminates.

Infinite loop

13.4.1.2 Additional notes and discussion on square_integers.cpp

Note the features of a pre-test loop.

Note that every while-loop, like the one you see in `square_integers.cpp`, is an example of a *pre-test loop*, which means a loop in which the first test is done (the first loop condition is evaluated) *before* the loop is executed. Also, the condition is re-evaluated before each subsequent loop iteration (execution of the loop body). Since the first test is done *before* the loop is entered, this means that the loop *may never be executed at all.* This is one of the things you must remember about a *pre-test loop* like the while-loop.

Count-controlled loop

All loops are controlled by a (possibly compound) boolean expression, such as the (simple, relational) expression `(n <= numberOfSquares)` in this program. Because the variable `n` is “counting” the number of iterations of the loop body and the loop will stop when the required number of iterations have been performed (provided the loop is correctly coded), this is also called a *count-controlled loop.*

13.4.1.3 Follow-up hands-on activities for square_integers.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `square_integers.cpp`.

□ Activity 2 Copy `square_integers.cpp` to `square_integers1.cpp` and bug it in the ways indicated below. And note once again, by the way, the manner in which these “bugging instructions” are given. Rather than mention specific variable names, for example, they are expressed in a way that is designed to help you become familiar with the terminology.

a. Leave out the parentheses enclosing the loop condition.

____

b. Leave out the braces enclosing the loop body.

____

c. Leave out the statement that initializes the loop control variable.

____

d. Leave out the statement that modifies the loop control variable within the loop body.

____

e. Replace the relational operator `<=` in the loop condition with the relational operator `<`.

____

□ Activity 3 Make a copy of `square_integers.cpp` called `square_integers2.cpp` and modify the copy so that the values in the table are displayed in decreasing order rather than increasing order.

□ Activity 4 Make a copy of `square_integers.cpp` called `square_integers3.cpp` and modify the copy so that the table contains only values corresponding to the even values from 2 up to a maximum value input by the user (which may or may not itself be an even value). (Be sure, as always, that all variables in your program have appropriate names.)

□ Activity 5 Make a copy of `square_integers.cpp` called `square_integers4.cpp` and modify the copy so that it performs exactly like the original, but uses a boolean variable called `finished` as the loop condition. Don't forget that you will have to decide exactly what `finished` means, you will have to initialize it correctly, and you will have to modify it correctly in the body of the loop. By the way, a boolean variable used in this way is often called a *flag*, and the resulting loop is referred to as a *flag-controlled loop*. Thus we see that terms like *count-controlled* and *flag-controlled* are not necessarily mutually exclusive when applied to loops.

Note the use of the boolean variable.

Flag-controlled loop

○ Instructor checkpoint 13.1 for evaluating prior work

13.4.2 sum_integers.cpp computes sums of integers using a do...while-loop

```
//sum_integers.cpp
//Computes the sum of all integers between and including two integers
//input by the user. Illustrates a count-controlled do...while-loop.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program computes the sum of all integers in a "
        "range of\nintegers entered by the user.\n\n";

    int small, large;
    cout << "Enter the smaller integer, then the larger: ";
    cin >> small >> large;  cin.ignore(80, ’\n’);  cout << endl;

    int numberToAdd = small;
    int max = large;
    int sum = 0;

    do
    {
        sum = sum + numberToAdd;
        numberToAdd++;
    } while (numberToAdd <= max);

    cout << "The sum of all integers from " << small << " to "
         << large << " (inclusive) is " << sum << ".\n";
    cout << endl;
}
```

13.4.2.1 What you see for the first time in sum_integers.cpp

do...while-loop

- A *do...while-loop*, which contains:

`do` *and* `while`

  - The C++ reserved word `do` used in conjunction with the reserved word `while`
  - A *conditional expression*

Condition tested after body of loop executed

    Once again this is the expression found within the (required) parentheses following the keyword `while`. This time it controls the *termination* of the loop, but *not* its beginning.

  - A *loop body*

Loop body must modify loop control variable

    These are the statements to be executed on each pass through the loop. The statements must be enclosed in braces if there is more than a single statement in the loop body. And once again the loop body must contain at least one statement that modifies the loop condition appropriately if the loop is to terminate.

- The typical formatting for a do...while-loop

  Note again the level of indentation of the statements in the loop body.

Formatting a do...while-loop

13.4.2.2 Additional notes and discussion on sum_integers.cpp

In the case of `sum_integers.cpp`, the loop control variable is `numberToAdd`, and of course it must be initialized and subsequently modified, like any other loop control variable. Note that the do...while-loop, unlike the while-loop, is a *post-test loop*. This means that the loop condition is tested at the *end* of the loop, rather than at the beginning, and it also means that the first loop iteration (i.e. the first execution of the loop body) takes place *before* the condition is tested for the first time. Or, equivalently, you are always guaranteed *at least one execution* of the loop body in a do...while-loop.

Note the features of a post-test loop.

The particular do...while-loop in `sum_integers.cpp` may also be regarded as a count-controlled loop, since the loop is essentially "counting" the values between two limits to determine which values to add.

13.4.2.3 Follow-up hands-on activities for sum_integers.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `sum_integers.cpp`.

□ Activity 2 Copy `sum_integers.cpp` to `sum_integers1.cpp` and bug it as follows:

a. Leave out the braces enclosing the loop body.

____

b. Leave out the statement that initializes the loop control variable.

____

c. Leave out the statement that initializes `max`.

____

d. Leave out the statement that initializes `sum`.

____

e. Leave out the statement that modifies the loop control variable within the loop body.

____

f. Replace `<=` in the loop condition with `<`.

____

g. Replace the `<=` in the loop condition with `!=`.

____

h. Rewrite the assignment to `sum` so that it uses the `+=` assignment operator, rather than the "ordinary" assignment operator `=`.

____

□ Activity 3 Make a copy of `sum_integers.cpp` called `sum_integers2.cpp` and modify the copy so that the user enters the number of positive even integers (starting with 2) that he or she wishes to sum, and the program computes and displays the sum of that many positive even integers. (Hint: As a check, the sum of the first 500 positive even integers is 250500, for example.)

○ Instructor checkpoint 13.2 for evaluating prior work

□ Activity 4 Make a copy of `sum_integers.cpp` called `sum_integers3.cpp` and modify the copy so that it performs as follows. First, the two values the user enters are a *start value*, and a *maximum sum.* Then the program begins summing integers, starting with *start value*, and stopping as soon as the sum reaches or exceeds *maximum sum.* The program must then display the *starting value*, the *maximum sum*, the *actual sum*, the *last value added*, and the *number of values summed.*

By the way, the loop control variable you will need for this program will be the variable that is being used to sum or "accumulate" the values being added, and the loop terminates when this value exceeds the allowed maximum. A loop like this is sometimes referred to as an *accumulator-controlled loop*, and the loop control variable in this case might be called an *accumulator.*

Accumulator, and accumulator-controlled loop

Typical output from your program might look like this:

```
Starting value ......... 6
Maximum sum ............ 47
Actual sum ............. 51
Last value added ....... 11
Number of values added .. 6
```

To get a feeling for what the program is supposed to do, fill in the table below. Some values have been supplied.

Starting value	6	10	−3	17	1
Maximum sum	47	60	4	139	5050
Actual sum	51				
Last value added	11				
Number of values added	6				

○ Instructor checkpoint 13.3 for evaluating prior work

13.4.3 display_sequences.cpp displays character and numerical sequences using for-loops, and also illustrates sequential loops

```
//display_sequences.cpp
//Displays sequences of both characters and integers in ranges chosen
//by the user. Illustrates a count-controlled for-loop, as well as
//sequential loops and some C-style casts.

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

int main()
{
    cout << "\nThis program displays both character and integer "
        "sequences in user-chosen\nranges. For characters, it "
        "assumes the user knows (from prior experience)\nthe "
        "internal integer codes for the first and last character "
        "of the required\nsequence.\n\n";

    int first, last; //Range boundaries
    int i;           //Loop control variable

    cout << "First, characters in \"increasing\" order.\nEnter the "
        "internal integer codes for the first and last characters "
        "you want:\n";
    cin >> first >> last;  cin.ignore(80, ’\n’);
    for (i = first; i <= last; i++)
        cout << char(i);

    cout << "\n\nNow, integers in decreasing order.\n"
        "Enter the first and last integers you want (larger first):\n";
    cin >> first >> last;  cin.ignore(80, ’\n’);
    for (i = first; i >= last; i--)
        cout << i << " ";

    cout << "\n\nFinally, characters marching down the display to the "
        "right in decreasing order.\nEnter the internal integer codes "
        "for the first and last characters you want:\n";
    cin >> first >> last;  cin.ignore(80, ’\n’);
    int indentLevel = 0;
    for (i = first; i >= last; i--)
    {
        cout << setw(indentLevel) << "" << char(i) << endl;
        indentLevel++;
    }
    cout << endl;
}
```

13.4.3.1 What you see for the first time in display_sequences.cpp

for-loop

- A *for-loop*, several in fact, each of which contains:

`for`

  - The C++ reserved word `for`
  - A set of (required) parentheses, which follows the keyword `for` and contains three key items:

The initialization

    - A first expression which provides any necessary *initialization* for the loop (including, but not limited to, the *loop control variable*)

The test

    - A second expression which is the *conditional expression* representing the *loop condition* and which will be tested to determine if the *loop body* should be executed (before each iteration, including the first)

The "modification"

    - A third expression which usually performs the necessary loop control variable modification
      Although there are many variations of the C++ for-loop, it is most convenient, and safer, to think of a for-loop as containing these three items, separated by semi-colons, and to use a for-loop only for those situations where you need a simple count-controlled loop.

Do not modify the loop control variable in the body of a for-loop.

  - A loop body, enclosed in braces when there is more than a single executable statement, and containing the statements that are executed during each loop iteration (Note that in the case of a for-loop the loop body does not in general contain a statement modifying the loop control variable, since this is handled auotmatically in a for-loop.)

Formatting a for-loop

- The typical formatting for a for-loop

  Note once again the indentation level of the statements in the loop body.

Sequential loops

- Sequential loops (nothing very mysterious, just one loop after another)

13.4.3.2 Additional notes and discussion on display_sequences.cpp

Consider which loop is most appropriate for your situation.

The C++ for-loop, as we hinted above, is actually quite versatile and can be used, for example, in place of any kind of while-loop. However, although any of the three types of loops we have seen can be used in virtually any situation, each kind of loop is better suited for use in certain situations. Typically it goes like this:

- for-loops for simple count-controlled loops (*definite iteration*)
- while-loops when you may not want the loop to execute at all (i.e., zero or more iterations) (*conditional iteration* or *indefinite iteration*)
- do...while-loops when you want at least one pass through the loop (i.e., one or more iterations) (also conditional iteration)

These are, of course, general guidelines only, but they cover a multitude of cases.

The while-loop is often best for reading input.

It's not perhaps completely obvious from the preceding discussion, but it turns out, for example, that a while-loop is often the best choice for reading input since it is the one that easily permits the loop not to execute at all, which is exactly what you want to happen if there is in fact no input to be read. This sort of knowledge is the kind of information you should salt away in whatever part of your brain you keep such things.

When to use sequential loops

The question of when to use "sequential loops" is an easy one to answer: whenever you have several tasks that must be performed one after the other, and two or more of them each require a loop, you will need "sequential loops". The sequential loops shown in `display_sequences.cpp` are all for-loops, but this does not have to be the case, of course.

13.4.3.3 Follow-up hands-on activities for display_sequences.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `display_sequences.cpp`.

□ Activity 2 Copy `display_sequences.cpp` to `display_sequences1.cpp` and bug it as follows:

a. In the first for-loop, remove `i = first`. (Leave the semi-colon (`;`) after `first`.)

   ____

b. In the first for-loop, remove `i = first;`. (Removethe semi-colon as well this time.)

   ____

c. In the first for-loop, remove `i <= last`.

   ____

d. In the first for-loop, remove `i++`.

   ____

□ Activity 3 Make a copy of `display_sequences.cpp` called `display_sequences2.cpp` and modify the copy so that the last for-loop displays the characters not "marching down to the right", but "marching down to the left" instead, with the first character having the largest indentation and the last character (the one on the last line) at the left margin.

○ Instructor checkpoint 13.4 for evaluating prior work

13.4.4 rounded_average.cpp computes the rounded integer average of all integer values on each input line, and also illustrates nested loops

```
//rounded_average.cpp
//Computes the (rounded) average of all integers on each of several
//input lines. Illustrates nested loops, and C-style casts.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program computes the rounded average of all integer "
        "values on each\nof a number of input lines. Each line of "
        "integers to be averaged must be\nterminated by the value "
        "-9999.\n\n";
    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’);

    int numberOfLines;
    int value;
    int sum;
    int count;
    int average;

    cout << "\nEnter number of input lines, or your end-of-file "
        "character to quit: ";
    cin >> numberOfLines;  cin.ignore(80, ’\n’);

    while (cin)
    {
        for (int lineNumber=1; lineNumber<=numberOfLines; lineNumber++)
        {
            cout << "\nEnter integer values for line number "
                 << lineNumber << " below (end with -9999):\n";
            sum = 0;
            count = 0;
            cin >> value;
            while (value != -9999)
            {
                sum = sum + value;
                count++;
                cin >> value;
            }
            average = int(double(sum) / count + 0.5);
            cout << "The rounded average of the values "
                 << "on line number " << lineNumber
                 << " is " << average << ".\n";
        }
        cout << "\n\nEnter number of input lines, or end-of-file "
            "character to quit: ";
        cin >> numberOfLines;  cin.ignore(80, ’\n’);
    }
    cout << endl;
}
```

13.4.4.1 What you see for the first time in rounded_average.cpp

- The use of *nested loops* (a *nested loop* is a loop that appears within the body of another loop)

  Nested loops

- The declaration of a loop control variable of the for-loop *within* the for-loop at the same time the variable is initialized (this may be done for *any* for-loop, if appropriate, i.e., there is no connection with loop nesting)

  Declaration of the loop control variable inside a for-loop

- A *sentinel-controlled loop* (the innermost while-loop is terminated by a *sentinel value* of –1)

  Sentinel-controlled loops

- An *end-of-file-controlled loop* (the outermost while-loop is terminated by an *end-of-file character*)

  End-of-file-controlled loops

  We need to say a bit more about this. This is actually a sneak preview of a wider topic that we will come back to more than once. The point is that the name of the input stream (`cin`) is being used as a boolean variable which is `true` (i.e., the stream is OK) till we enter the end-of-file character, at which point it becomes `false` (not OK). One of the nice things about C++ is its ability to treat input streams in this way.

  Input stream as a boolean variable

- A bit of a *portability problem* (see discussion below)

  Portability problem

13.4.4.2 Additional notes and discussion on rounded_average.cpp

As this program shows, loops may be nested within other loops "several levels deep". Note that here, for example, we have a for-loop inside an outer while-loop and the for-loop itself has a second while-loop as part of its body.

The general structure for nested loops looks like this:

Structure of nested loops

```
start of first loop
    part of first loop body
    start of second loop
        part of second loop body
        start of third loop
            body of third loop
            ... possibly other loops in here
        end of third loop
        more of second loop body
    end of second loop
    more of first loop body
end of first loop
```

Note that the loops do not "overlap", and note how the indentation and alignment make it easy to identify the various parts of each loop. Your C++ nested loops must always be formatted so that the structure is equally clear.

As for when to use a nested-loop construct, the need arises in a fairly natural way: Whenever you have a loop in which one or more of the statements themselves need to be repeated, you are dealing with a nested-loop situation.

When to use nested loops

Input stream `true` *if OK,* `false` *if not*

The outermost while-loop in `rounded_average.cpp` requires some additional comment. First note that the loop condition is simply `cin`, the name of the input stream. The name of an input stream can be treated as a boolean variable which has the value `true` unless something has gone wrong with the stream. What can go wrong with an input stream? Well, if an attempt is made to read a value and an end-of-file character (also denoted by the acronym EOF) is encountered, the input stream will "shut down" and not provide any more input, at which point its "boolean value" will be false, and this value can be used to terminate the loop.

End-of-file is system-dependent.

A *portability problem* arises here. Clearly it would be ideal if the program could tell its user exactly what character to enter to denote the *end-of-file*. The problem is, this character is not the same on all systems, and you need to know what it is on your system in order to run this program properly.

13.4.4.3 The end-of-file character on your system

How do you enter an end-of-file character from the keyboard on your system?

Answer

13.4.4.4 Follow-up hands-on activities for rounded_average.cpp

□ Activity 4 Copy, study, test and then write pseudocode for `rounded_average.cpp`. For simplicity when testing, use input with at least one data value per line.

□ Activity 5 Copy `rounded_average.cpp` to `rounded_average1.cpp` and bug it as follows:

a. Declare all variables in a single declaration statement.

____

b. Remove the braces enclosing the body of the outermost while-loop.

____

c. Remove the braces enclosing the body of the for-loop.

____

□ Activity 6 Make a copy of `rounded_average.cpp` called `second_last.cpp` and modify the copy so that the output is *not* the average of the values on each line, but the *second last* value on each line. Test the program with data that always provides at least *two* values per line, and write the program assuming such input.

○ Instructor checkpoint 13.5 for evaluating prior work

13.4.5 draw_box.cpp displays an empty box with user chosen size, border and position

```
//draw_box.cpp
//Displays an empty box. User chooses the size, the border character,
//and the indentation. Illustrates sequential loops.

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

int main()
{
    cout << "\nThis program draws an \"empty\" box on the display. "
        "Box \"height\" refers to\nthe number of lines and \"width\" "
        "refers to the number of character widths.\n\n";

    int width, height;
    char ch;
    int indentLevel;

    cout << "Enter width, then height, of box: ";
    cin >> width >> height;  cin.ignore(80, ’\n’);
    cout << "Enter character to be used for the box border: ";
    cin >> ch;  cin.ignore(80, ’\n’);
    cout << "Enter the number of spaces to indent the box: ";
    cin >> indentLevel;  cin.ignore(80, ’\n’);  cout << endl;

    int charCount;
    cout << setw(indentLevel) << "";
    for (charCount = 1; charCount <= width; charCount++)
        cout << ch;
    cout << endl;
    for (int line = 2; line <= height-1; line++)
    {
        cout << setw(indentLevel) << "";
        cout << ch << setw(width-2) << "" << ch << endl;
    }
    cout << setw(indentLevel) << "";
    for (charCount = 1; charCount <= width; charCount++)
        cout << ch;
    cout << endl << endl;
}
```

13.4.5.1 What you see for the first time in draw_box.cpp

This program does not show any new C++ language features. It does show how loops (in particular, for-loops) can be used to display patterns of various kinds. Here you see three sequential for-loops being used to draw a simple “empty” box. Of course, both sequential and nested loops may be required in a given program, depending on what the display requirements are.

13.4.5.2 Additional notes and discussion on draw_box.cpp

We have seen this sort of thing many times before, but you should again take a close look at how the output from this program is positioned on the screen. As always, though, keep in mind that this is not *the* way to do it, just *a* way.

13.4.5.3 Follow-up hands-on activities for draw_box.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `draw_box.cpp`.

□ Activity 2 Copy `draw_box.cpp` to `draw_box1.cpp` and bug it as follows:

a. First replace `cin >> ch` by `cin.get(ch)`. Then remove all instances of the statement `cin.ignore(80, '\n');` and reinsert them one at a time. Indicate which, if any, are critical to the operation of the revised program and which are not.

____

____

____

b. Replace each instance of `""` with `"␣"` (i.e., replace each null string with a string constant containing a single blank) and study the position of all output from the program for different input values. Describe what difference, if any, this change would make in the display. (Note, by the way, that the symbol ␣ is sometimes used to emphasize the existence of a blank space.)

____

____

____

□ Activity 3 Make another copy of `draw_box.cpp` called `draw_box2.cpp` and modify the copy so that a second character is obtained from the user and the displayed box is no longer "empty", but instead its interior is filled with this second character.

○ INSTRUCTOR CHECKPOINT 13.6 FOR EVALUATING PRIOR WORK

Module 14

Consolidation of I/O (including file I/O), selection and looping

14.1 Objectives

- To understand how selection constructs and looping constructs can work together to build programs of greater power and flexibility (by performing tests of one kind or another before, during, and/or after a loop, for example).
- To understand why we often need to perform one or more checks *after* a conditional loop has terminated to determine the reason *why* it ended, if that loop is controlled by a *compound boolean condition*.
- To understand when and how to test an input stream to see if it is still functioning properly.
- To understand when and how to use *embedded debugging code*.
- To learn how to make a program read in, at run-time, the name of file to be used for input or output, and how to then open and use the file for input or output.

14.2 List of associated files

- `positive_average.cpp` computes the average of just the positive integers entered from the keyboard.
- `extreme_values.cpp` finds the largest negative and the smallest positive integers entered from the keyboard.

- odd_squares.cpp computes the squares of just the odd positive integers entered from the keyboard.
- two_flags.cpp tries to have the user enter an integer from 1 to 3.
- count_characters.cpp counts both the capital letters and the small letters in a textfile.
- count_characters.in contains sample input data for count_characters.cpp.
- sum_odd_positives.cpp computes the sum of all odd positive integers up to a limit entered by the user.
- test_input_stream.cpp illustrates testing of an input stream.
- four_characters.in is a sample input data file for Activity 3 following test_input_stream.cpp.
- filename.cpp shows how to read in, at run-time, the name of a file, and then open that file for input.

14.3 Overview

In the beginning, all of our programs were entirely sequential. That is, each program started by executing the first executable statement (in the main function), then the second, and so on in sequence until all of the executable statements in main had been executed, at which point the program ended and returned a value to the operating system, usually 0 to indicate success.

In the previous two Modules we saw, first, how a program could alter this rather pedestrian way of doing things by making decisions from time to time about which statement, or group of statements, to execute next, and then, second, how a program could repeat one or more statements if called upon to do so.

The three things a computer can do

These three things—*sequential execution*, *selection*, and *looping*—constitute the full repertoire of what a program can do, in a certain sense. But, of course, those three actions can be combined and duplicated in a myriad of ways, and this is what gives a computer program its power. In this Module we look at some such combinations. The ability to produce your own such combinations that correctly perform the required tasks in your own programs is just one of the many skills that you need to develop on your way to becoming a good programmer.

14.4 Sample Programs

14.4.1 positive_average.cpp computes the average of just the positive integers entered from the keyboard

```
//positive_average.cpp
//Finds the average of all positive integer values input by
//the user. Illustrates an EOF-controlled loop, and a test
//after the loop to make sure the input data set was not empty.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program computes the average of any number of "
        "positive integer values.\nInput values may be positive, "
        "negative, or zero, but non-positive values are\nsimply "
        "ignored. Enter as many values as you like, then press the "
        "Enter key.\nTo compute and display the avarage enter an "
        "end-of-file and press Enter again.\n";

    int i;
    int numberOfPositiveIntegers = 0;
    int sum = 0;
    double average;

    cout << "\nStart entering values on the line below:\n";
    cin >> i;
    while (cin)
    {
        if (i > 0)
        {
            sum = sum + i;
            numberOfPositiveIntegers++;
        }
        cin >> i;
    }

    if (numberOfPositiveIntegers == 0)
        cout << "No positive integers were input.\n";
    else
    {
        average = double(sum) / double(numberOfPositiveIntegers);
        cout << "The average of the "         << numberOfPositiveIntegers
             << " positive integer values is " << average << ".\n";
    }
    cout << endl;
}
```

14.4.1.1 What you see for the first time in positive_average.cpp

This program illustrates for the first time (except for `shell.cpp` and any programs based on it) the use of both selection and looping constructs in the same program.

14.4.1.2 Additional notes and discussion on positive_average.cpp

Decision-making and looping working together

This program is our first to exhibit both decision-making and looping working together. It uses a while-loop to read input (remember that while-loops tend to be better for this job than either of the other two kinds of loops), and it uses a simple if-statement in the body of the while-loop to "filter out" the values that aren't wanted, i.e., the non-positive values, since these are not to be included in the average.

Division-by-zero error

Following the loop there is another test which makes sure the value of the variable `numberOfPositiveIntegers` is strictly greater than zero before attempting to compute the average, since if it isn't this would mean that no positive data values had been input, and we would then have a division-by-zero error when computing the average. It is always better to write programs that are as *robust* as possible, i.e., that deal in a graceful manner, as much as possible, with things that can go wrong. Good programmers are able to anticipate many of the things that can go wrong during program execution and write code to deal with them if they do.

14.4.1.3 Follow-up hands-on activities for positive_average.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `positive_average.cpp`.

□ Activity 2 Copy `positive_average.cpp` to `positive_average1.cpp` and bug it as follows:

a. Remove the statement that initializes `count`.

____

b. Remove the statement that initializes `sum`.

____

c. Remove the first instance of `cin >> i;` (i.e., remove what is called the *priming read* for the while-loop).

____

d. Remove the second instance of `cin >> i;`.

____

e. Change the statement that gives the variable `average` its value to the statement `average = sum / count;`.

____

f. Remove the braces enclosing the body of the if-statement within the while-loop.

____

g. Remove the braces following the `else` in the if...else-statement.

____

□ Activity 3 Make a copy of `positive_average.cpp` called `even_average.cpp` and modify the copy so that it averages only the positive even integers entered.

○ INSTRUCTOR CHECKPOINT 14.1 FOR EVALUATING PRIOR WORK

14.4.2 extreme_values.cpp finds the largest negative and smallest positive integers entered from the keyboard

```
//extreme_values.cpp
//Finds the largest negative integer and smallest integer among the
//input values. Illustrates a sentinel-controlled loop containing one
//test in its body, and followed by two sequential tests.

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

int main()
{
    cout << "\nThis program determines the largest negative integer "
        "and the smallest\npositive integer entered by the user. "
        "Terminate input by entering 0.\n\n";

    int largestNeg = INT_MIN;
    int smallestPos = INT_MAX;
    int newValue;

    cout << "Enter your integer sequence starting on the line below:\n";
    cin >> newValue;
    while (newValue != 0)
    {
        if (newValue < 0  &&  newValue > largestNeg)
            largestNeg = newValue;
        else if (newValue > 0  &&  newValue < smallestPos)
            smallestPos = newValue;
        cin >> newValue;
    }

    if (largestNeg == INT_MIN)
        cout << "Either no negative integers were input or the largest "
            "was the value\n" << "of INT_MIN, i.e. " << INT_MIN << ".\n";
    else
        cout << "The largest negative integer was "
             << largestNeg << ".\n";

    if (smallestPos == INT_MAX)
        cout << "Either no positive integers were input or the smallest "
            "was the value\nof INT_MAX, i.e. " << INT_MAX << ".\n";
    else
        cout << "The smallest positive integer was "
             << smallestPos << ".\n";
    cout << endl;
}
```

14.4.2.1 What you see for the first time in extreme_values.cpp

There are no new C++ language features in this program, but we do see the use of two (predefined) named constants (`INT_MIN` and `INT_MAX`) from one of the C++ standard libraries (the `climits` library in this case) to initialize variables.

14.4.2.2 Additional notes and discussion on extreme_values.cpp

This program contains a while-loop with a nested if-statement in its body. Note that this nested-if construct may choose to perform *neither* of the two explicit possibilities it contains. The loop is followed by two sequential tests to determine what the loop has actually accomplished.

14.4.2.3 Follow-up hands-on activities for extreme_values.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `extreme_values.cpp`.

□ Activity 2 Copy `extreme_values.cpp` to `extreme_values1.cpp` and bug it as follows:

a. Leave out the compiler directive which includes the `limits` library header file.

   ____

b. Change the operator `==` in line 31 to `=`.

   ____

   This change illustrates a very common C++ programming bug, namely, using a single equal sign (`=`) when a double one (`==`) is required.

   Remember that the single equal sign is an assignment operator, while the double one is a relational operator. Smart compilers will warn you if you make such a "mistake". It is conceivable, but unlikely, that when doing this you have *not* made a "mistake", in the technical sense, but you would certainly have made a programming-style error.

   A best practice

   Note that when you have a variable on one side of the `==` and a constant on the other, a "best practice" might be to make sure the constant is on the left. Then, if you inadvertently use `=` when you meant to use `==`, any compiler will report an error since an assignment cannot be made to a constant.

□ Activity 3 As we hinted above, the nested if-statement in the while-loop body of the program in `extreme_values.cpp` is such that *either* or *neither* of its two options may actually be chosen. Suppose, in the interests of readability, we decide that the code should be made more explicit by explicitly including the "missing" third option, i.e., that *neither* of the two options given is executed. Show below the code you would add to do this, and indicate as well where you would place that code. If you want to test the code in a program, put your revised program in a file called `extreme_values2.cpp`.

____

____

____

____

____

____

○ Instructor checkpoint 14.2 for evaluating prior work

□ Activity 4 Design and write a program that reads in any number of real numbers (as opposed to integers) and then prints out both the largest and the smallest values seen among all input values. An additional requirement is that no variable in the program can be given a value other than one of the values read in. (This is just another way of saying that you are *not* permitted to initialize any variable with a built-in value from one of the libraries. So, you need an algorithm that does not require such an initialization.) Put your program in a file called `large_small.cpp`.

○ Instructor checkpoint 14.3 for evaluating prior work

14.4.3 odd_squares.cpp computes the squares of just the odd positive integers entered from the keyboard

```
//odd_squares.cpp
//Outputs squares of odd integers input by the user. Also counts and
//reports number of odd values seen. Illustrates both a count-controlled
//loop and a second counter that counts something else (odd integers).

#include <iostream>
using namespace std;

const bool DEBUGGING_ON = false;

int main()
{
    cout << "\nThis program outputs the square of any odd\ninteger "
        "entered, but ignores all even values.\nIt also indicates how "
        "many odd values were seen.\nAll input values must be *positive* "
        "integers.\n\n";

    int numberOfValues;
    cout << "Enter the number of data values to be entered: ";
    cin >> numberOfValues;  cin.ignore(80, '\n');  cout << endl;

    int i;
    bool iIsOdd;
    int oddCount = 0; //Initialize count of odd input values properly

    for (int loopCount = 1; loopCount <= numberOfValues; loopCount++)
    {
        cout << "\nEnter a data value here: ";
        cin >> i;  cin.ignore(80, '\n');
        iIsOdd = (i%2 == 1); //Check to see if value read is odd
        if (iIsOdd)
        {
            cout << i << " squared is " << i*i << ".\n";
            ++oddCount;
        }
        else
            cout << "That one wasn't odd." << endl;
        if (DEBUGGING_ON)
        {
            cout << "loopCount: " << loopCount << "   "
                 << "i: "         << i         << "   "
                 << "iIsOdd: "    << iIsOdd    << "   "
                 << "oddCount: "  << oddCount  << endl << endl;
        }
    }
    cout << "\nThe total number of odd integers was "
         << oddCount << ".\n";
    cout << endl;
}
```

14.4.3.1 What you see for the first time in odd_squares.cpp

Embedded debugging code

This program shows the use of *embedded debugging code*[1] to "turn on" or "turn off" a program's ability to display various quantities of interest during execution.

14.4.3.2 Additional notes and discussion on odd_squares.cpp

So, just what do we mean by *embedded debugging code*? What we mean is code that we intentionally put in our programs, as we design and build them, for that sad day when we realize our program has a bug and we need to take a very close look at what we have done if we expect to find out what's wrong. We could, of course, wait until that time before inserting such code, but if we do we may well be in a panic state, do a hurried and quite possibly poor job of entering this code "on the fly", and waste more time as a result before finding the error, if in fact we do find what is wrong.

It is therefore much better to expend, as part of the design process, some effort toward including embedded debugging code that is for one purpose only: displaying quantities of interest at certain points in the program's execution so that those values can be examined to determine whether they are correct.

There must be a way to "turn on" and "turn off" the output of this code, since the output is not something we want to see when the program is finished and working properly. We could insert the code when we want it, and remove it when we don't want it, but that would generally be more work than we would like to do.

Use a global constant as an "on-off switch" for debugging.

So, we use a *global constant* of data type `bool` (called `DEBUGGING_ON` here, but this is just a programmer chosen name) which can be set to `true` or `false`, according to whether we want the embedded debugging code to output its diagnostic values, or not. This constant is then used in appropriate selection structures which control the output of the debugging code, i.e., the code that outputs the values of interest.

In this program, note that although `oddCount` is initialized to 0, as it should be, and in fact needs to be, neither `i` nor `iIsOdd` is given an initial value. Why not? Some programmers will argue that *every* variable should be initialized when declared, to make sure that *no uninitialized variable is used.* We do not subscribe to that theory, since it seems to make about as much sense to use a variable which has been given some arbitrary or "unnatural" value as it does to use it uninitialized, and doing so is potentially just as confusing. So, if it does not make conceptual sense to initialize a variable at the time of declaration, we don't do it. Besides, compilers are smart enough to warn you when you try to use the value in an uninitialized variable, so letting them do so may be "safer"..

[1] Such code can be used to help us perform a *program trace*.

14.4.3.3 Follow-up hands-on activities for odd_squares.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `odd_squares.cpp`.

□ Activity 2 Make another copy of `odd_squares.cpp` called `odd_squares1.cpp`. Then change `false` to `true` in the definition of `DEBUGGING_ON`. Finally, re-test the program as in the previous activity.

□ Activity 3 Copy `odd_squares.cpp` to `odd_squares2.cpp` and bug it as follows:

a. Replace `i % 2 == 1` by `i % 2 == 0`.

____

b. Remove the initialization of `oddCount` (but leave the declaration).

____

c. Move the declaration of `iIsOdd` from its current location to the line in which it is used in the body of the outer for-loop.

____

From the previous "bugging" activities you will have observed that "slight" errors (whether caused by lapses in logic or stemming from some other source) can cause rather serious problems with your programs, problems that may not be easy to find just by staring at your code. That's when you need to take more serious measures, by performing a trace, for example, or "turning on" your embedded debugging code, if you have been thoughtful enough to include it. The following two activities are designed to familiarize you with this process.

□ Activity 4 Make a copy of `odd_squares.cpp` called `odd_squares3.cpp` and first modify the copy by reintroducing the first of the "bugging" changes you tried above (replacing `i % 2 == 1` by `i % 2 == 0`). Test again to make sure you know how the altered program behaves. Now change the value of the named constant `DEBUGGING_ON` to `true`, and test once more to see the information that the embedded debugging code provides you. Analyze the information thus obtained to help you zero in on the problem with the program. It's easy in this case, of course, because you know in advance what the problem is, but you should be able to see how the same technique would be used in situations where that was not the case.

□ Activity 5 Repeat the previous activity with another copy of `odd_squares.cpp` called `odd_squares4.cpp` and the second bugging change above (removal of the initialization of `oddCount`).

○ Instructor checkpoint 14.4 for evaluating prior work

14.4.4 two_flags.cpp tries to have the user enter an integer from 1 to 3

```
//two_flags.cpp
//Tests whether either of two conditions is satisfied. Illustrates a
//loop controlled by two flags (two conditions). Note that when such
//a loop ends it may be because either (or perhaps both) of the two
//conditions failed.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program tries to get the user to enter an integer "
        "from 1 to 3.\nHowever, only three tries are permitted before "
        "the program gives up and quits.\n\n";

    bool validEntry = false; //Since there are no entries yet
    bool outOfTries = false; //Since no tries have been made yet

    int i;
    int tryCount = 0;

    while (!validEntry && !outOfTries)
    {
        cout << "Enter an integer in the range 1..3: ";
        cin >> i;
        cout << endl;
        tryCount++;
        validEntry = (i >= 1  &&  i <= 3);
        outOfTries = (tryCount == 3);
        if (!validEntry && !outOfTries)
            cout << "Invalid entry. Try again.\n";
    }

    if (validEntry)
        cout << "The valid entry "      << i
             << " was entered on try #" << tryCount << ".\n";
    else if (outOfTries)
        cout << "You ran out of tries!\n";
    cout << endl;
}
```

14.4.4.1 What you see for the first time in two_flags.cpp

- A *loop condition* which is a *compound boolean condition*
- An example of a program which deals *robustly* with user input, though only *range robustness* (values input must lie within a certain subrange of a given data type), not *type robustness* (values input must be of the correct data type), is actually illustrated in the program

Program robustness

14.4.4.2 Additional notes and discussion on two_flags.cpp

One thing to note about this program is the loop condition of the while-loop, which is a *compound boolean expression*. The form of this particular compound expression means that there are *three* distinct situations which will cause the loop to terminate:

Multiple conditions for loop termination

- `validEntry` becomes `true` (but `outOfTries` is still `false`).
- `outOfTries` becomes `true` (but `validEntry` is still `false`).
- Both `validEntry` and `validEntry` become true simultaneously.

Because we have different situations that may lead to loop termination, we need to check, following the loop, to see exactly why the loop terminated and then take appropriate action. Note also that only two of the three situations mentioned are actually dealt with explicitly in the program.

A second thing to observe about the program is that both of the boolean variables, `validEntry` and `outOfTries`, as well as the counter `tryCount`, need to be initialized when declared, unlike `i`, which is just used to read values entered by the user. If the boolean variables are not initialized properly, the while-loop will never be entered. And if `tryCount` is not initialized to zero, the count of the user's attempts to enter a valid value will not be correct.

Proper initialization of boolean variables ...

and other variables!

14.4.4.3 Follow-up hands-on activities for two_flags.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `two_flags.cpp`. Among other tests you may conduct, make sure you enter data that allows you to answer the following question: What message is output by the program if *both* of the boolean variables are true when the loop terminates, and why? Be sure to enter *both* parts of your answer to this question below.

____

____

____

____

□ Activity 2 Copy `two_flags.cpp` to `two_flags1.cpp` and bug it as follows:

a. Remove the initialization of `tryCount` (but leave the declaration).

____

b. Remove the initialization of `validEntry` (but leave the declaration).

____

c. Remove the initialization of `outOfTries` (but leave the declaration).

____

d. Remove the update of `tryCount` in the body of the while-loop.

____

e. Remove the update of `validEntry` in the body of the while-loop.

____

f. Remove the update of `outOfTries` in the body of the while-loop.

____

□ Activity 3 Make a copy of `two_flags.cpp` called `two_flags2.cpp` and modify the copy so that if *both* of the boolean variables are true when the loop terminates the message

```
Wow! You entered a valid value of ? on your very last attempt!
```

is printed out (in which the `?` would be replaced by the actual value entered), while if only *one* of the boolean variables is true when the loop terminates, the same message is printed out that the original program would print out, and in this case *only* that message is displayed.

○ INSTRUCTOR CHECKPOINT 14.5 FOR EVALUATING PRIOR WORK

14.4.5 count_characters.cpp counts capital letters and small letters in a textfile

```
//count_characters.cpp
//Reads a file of text and reports the number of upper case (capital)
//letters and the number of lower case (small) letters on each line of
//the file. The file may contain blank lines, and may also be empty,
//but in the latter case the program says nothing.

#include <iostream>
#include <fstream>
#include <cctype>  //for access to the "isupper" and "islower" functions
using namespace std;

int main()
{
    cout << "\nThis program counts the number of capitals and small "
        "letters on each line of\na textfile. If no output is shown "
        "below, make sure the input file is available\nand is non-empty, "
        "since no message reporting either its absence or the fact\n"
        "that it is empty is displayed.\n\n";

    const char NEW_LINE = '\n';

    ifstream inFile;
    int lineCount = 0;
    int upperCount;
    int lowerCount;
    char ch;

    inFile.open("in_data"); //Assume file is present; no error
                            //message is displayed if it isn't
    while (inFile) //Loop terminates when end-of-file reached
    {
        inFile.get(ch); //Read a character (unless there are none)
        if (inFile)     //Process character (if one actually read)
        {
            ++lineCount;
            upperCount = 0;
            lowerCount = 0;
            while (ch != NEW_LINE) //Loop terminates at end-of-line
            {
                //How could the if...else-statement below be rewritten to
                //improve "efficiency", assuming more lowercase letters
                //than uppercase letters in the input data?
                if (isupper(ch))
                    ++upperCount;
                else if (islower(ch))
                    ++lowerCount;
                inFile.get(ch);
            }
            cout << "Line " << lineCount << " contains "
                 << upperCount << " capitals and "
                 << lowerCount << " small letters.\n";
        }
    }
    cout << endl;
}
```

14.4.5.1 What you see for the first time in count_characters.cpp

Determining the case of a character

- The inclusion of the `cctype` library header file, for access to the functions `isupper()` and `islower()`, which can be used to determine if a letter is upper case or lower case (respectively)
- The definition of the *end-of-line character* `\n` as a *named constant* (i.e., a *programmer-defined constant*)
- Nested loops combined with tests to read lines of input as characters, one at a time, and perform some analysis on the characters as they are read

14.4.5.2 Additional notes and discussion on count_characters.cpp

Reminder: Continue to increase your knowledge of what's available from your C++ libraries.

Note the convenience of having built-in functions like `isupper()` and `islower()` to use in situations like that shown this program. If you are going to be a "real-world" C++ programmer, it is important that you become familiar with the C++ standard libraries, so that if you need a function that is readily available from one of those libraries you don't go to the trouble of writing your own version of that function.

The initializations, as always, are important, and here we have a somewhat subtle, and critically important, observation to make. Note that `lineCount` is initialized in its declaration, but `upperCount` and `lowerCount` are *not* . Why not? Well, they could both have been initialized to 0 in their declarations, and no harm would have been done, but that might have lulled us to sleep and caused us to forget that each of these variables must also be initialized to 0 *before processing each line*, and hence *before each execution of the inner while-loop.*

14.4.5.3 Follow-up hands-on activities for count_characters.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `count_characters.cpp`. Shown below are the contents of a supplied data file for this program called `count_characters.in`, which you can copy and use in your testing, but you should also create some additional data files of your own to try.

Contents of the file `count_characters.in` are shown between the heavy lines:

```
This is line One of THE filE.
Here IS line two.
        It doesn't matter wHeRe we StarT a line.
or finish                                           one, FORTHATMATTER...


Note that blank lines are OK too.
```

□ Activity 2 Copy `count_characters.cpp` to `count_characters1.cpp` and bug it as follows:

a. Comment out the current initializations of `upperCount` and `lowerCount` temporarily, and initialize each one to 0 *only* in its declaration.

   ____

b. Replace each instance of the input statement `inFile.get(ch);` with the input statement `inFile >> ch;`.

   ____

□ Activity 3 Make a copy of `count_characters.cpp` called `count_characters2.cpp` and modify the copy so that the program performs exactly as before but does not use `isupper()` or `islower()` (or any other library function) to do what each of these functions does. That is, make believe you don't know these functions exist.

□ Activity 4 What is your answer to the question asked in the comments opposite the inner while-loop in `count_characters.cpp`? Show your rewritten code below. If you wish to try your code in a program, put this revised version of the program in a file called `count_characters3.cpp`.

____

____

____

____

____

○ Instructor checkpoint 14.6 for evaluating prior work

□ Activity 5 Make another copy of the file `count_characters.cpp` and call it `count_characters4.cpp`. Modify the copy so that in addition to the output that the program already produces, it also counts and displays similar information on the number of blank spaces, digit characters and punctuation characters in the file. Find and use suitable functions for this from the C++ standard library.

○ Instructor checkpoint 14.7 for evaluating prior work

14.4.6 sum_odd_positives.cpp computes the sum of all odd positive integers up to a limit entered by the user

```
//sum_odd_positives.cpp
//Sums the odd integers from 1 to a value input by the user.
//Illustrates a variable local to a block.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program finds the value of the sum of all odd "
        "positive integers from 1\nto an integer input by the user "
        "(inclusive, if the integer input is odd).\nTerminate by "
        "entering a non-positive integer.\n\n";

    int sum = 100 + 20 + 3;  //This is the first "sum".
    cout << "Outer (fixed) sum: " << sum << "\n\n";

    int j;
    cout << "Enter an integer: ";
    cin >> j;  cin.ignore(80, '\n');
    while (j > 0)
    {
        int sum = 0;  //This is the second "sum".
        int i = j;
        while (i > 0)
        {
            if (i % 2  ==  1)
                sum = sum + i;  //Which "sum" is this, and why?
            --i;
        }
        cout << "Inner (varying) sum of odd integers, "
            "from 1 to " << j << " is " << sum << ".\n";
        cout << "\nEnter another integer: ";
        cin >> j;  cin.ignore(80, '\n');
    }

    cout << "\nOuter (fixed) sum: " << sum << endl;
    cout << endl;
}
```

14.4.6.1 What you see for the first time in sum_odd_positives.cpp

But just because it can *be done does not mean that it* should *be done.*

This program shows that a variable (namely, `sum` here) may be declared *twice* (or more times, for that matter) inside a `main` function, under appropriate circumstances.

14.4.6.2 Additional notes and discussion on sum_odd_positives.cpp

If you study the output of this program, the evidence is quite strong that we do indeed have two *different* variables named `sum` in the program, since a first `sum` is initialized to 123 (i.e., 100 + 20 + 3), then a second `sum` is declared, initialized to 0, and used to compute the sums of odd integers, and yet when we output the value of `sum` one final time, the original value of 123 shows up again.

So, what's going on? When `sum` is *re-declared* inside the outer while-loop (in line 23) with the statement `int sum = 0;`) we get an entirely *new and distinct* variable called `sum` which is *local* to the *block* within which it is declared (the body of the outer while-loop), and is not *accessible* (or *visible*, or *known*) *outside* that block.

Local vs. global variables

The whole question of *local variables* vs. *global variables* comes up again in the context of functions, which we deal with in subsequent Modules.

14.4.6.3 Follow-up hands-on activities for sum_odd_positives.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `sum_odd_positives.cpp`.

□ Activity 2 Test the program with enough different input data values so that you can make a pretty good guess at what must replace the ??? in the following statement:

If n is an odd positive integer, then 1 + 3 + 5 + ... + n = ???.

Now write a sentence that does not involve any formulas but that describes the relationship that you have deduced above.

____

____

□ Activity 3 Copy `sum_odd_positives.cpp` to `sum_odd_positives1.cpp` and bug it as follows:

a. Remove the first declaration and initialization of `sum`.

____

b. Remove the second declaration and initialization of `sum`.

____

□ Activity 4 What is the answer to the question asked in the comments to the right of the inner while loop?

____

____

○ Instructor checkpoint 14.8 for evaluating prior work

14.4.7 test_input_stream.cpp illustrates how to test an input stream to determine if it is still functioning properly

```
//test_input_stream.cpp
//Illustrates how to test the input stream.

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

int main()
{
    cout << "\nThis program tests a file input stream in certain ways.\n"
        "It will either open and read a file, reporting the values it "
        "found.\nOr, it will report an error of some kind.\n";

    int i1, i2;

    ifstream inFile;
    inFile.open("in_data");
    if (!inFile)
    {
        cout << "\nError: Input data file not found ..."
            "\nCreate data file and run program again.\n";
        cout << endl; //Why have this statement?
        return 1;
    }

    inFile >> i1 >> i2;
    if (!inFile)
    {
        if (inFile.eof())
        {
            cout << "\nError: End of file reached before "
                "all data read.\n";
            cout << endl; //Why have this statement?
            inFile.close();
            return 2;
        }
        else
        {
            cout << "\nError: Input data improperly formatted.\n";
            cout << endl; //Why have this statement?
            inFile.close();
            return 3;
        }
    }
    inFile.close();

    cout << "\nThe file was actually read. The values in it were:\n";
    cout << "   i1: " << i1 << "   i2: " << i2 << endl << endl;
    return 0;
}
```

14.4.7.1 What you see for the first time in test_input_stream.cpp

Testing the state of of an input file stream

- The testing of the "state" of an input file stream for various potential error conditions
- The testing of an input file stream to see if the end of the file has been reached (i.e., to see if the *end-of-file character* has been encountered while attempting a read operation)
- The use of the statement `return ?;`, where `?` is some small positive integer whose value will be returned to the caller of this program to indicate the nature of an error, if one occurred

  Note that a different return-value can be used for each kind of problem that might occur. Now you can see why it makes sense to choose `0` to indicate "success" (the absence of *all* errors), if we are going to use different positive integer values to differentiate the potential error conditions.

14.4.7.2 Additional notes and discussion on test_input_stream.cpp

Input and output of data from a program is an area of much potential difficulty, and hence an area that you should give special attention when you are designing and writing your programs. You should not think for a moment that the testing you see in this program covers all of the things that might go wrong when you are using textfiles for input and output. Again, the best advice is simply this: Be vigilant!

14.4.7.3 Follow-up hands-on activities for test_input_stream.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `test_input_stream.cpp`. There are four distinct situations in which you want to make sure the program behaves properly:

- The input file simply doesn't exist.
- The input file exists, but does not contain enough data.
- The input file exists, and contains "enough" data, but some of the data is bad.
- The input file is as it should be, i.e., it exists and contains the right amount of the right kind of data.

Make sure you provide suitable versions of the input file to exercise all four options during your testing.

□ Activity 2 Copy `test_input_stream.cpp` to `test_input_stream1.cpp` and bug it as per the items listed below. These changes may cause your program not to compile or to behave improperly when it runs. If a change does not cause a compile-time error and lets the program run, then begin your testing with a properly formatted file and move one by one through the variations mentioned in the previous activity.

a. Remove the outer parentheses from `(inFile.eof())` in line 29.

____

b. Remove the set of braces from lines 28 and 44.

____

c. Remove the set of braces from lines 30 and 36.

____

d. Remove the set of braces from lines 38 and 43.

____

□ Activity 3 What is the answer to the question asked is the comments in this program? (It's the same answer in all three cases.)

____

□ Activity 4 Design and write a program that will read from an input textfile the first four characters on each of the first two lines of the file, and display those characters on one line of the screen. In the output there must be a space between the first four characters and the second four characters, and the line containing the output on the screen must be preceded by, and followed by, a blank line.

Each line of the input file may or may not contain characters other than the ones your program is to read. Any such additional characters should simply be ignored.

Place your program in a file called `four_characters.cpp`; then compile, link, run and test it, in the first instance, with the sample file `four_characters.in`, which contains the two lines of data shown below, in which both `SEND` and `CASH` begin in the first column on the line (and *all* of the text on each of the two lines is actually *in* the data file).

Contents of the file `four_characters.in` are shown between the heavy lines:

```
SEND  <-- First line (contains a verb in this case)
CASH  <-- Second line (contains the object of the verb)
```

When this is the input file, your program should display the following line of text (preceded by, and followed by, a blank line):

```
SEND CASH
```

Your program must also respond appropriately if the input file is missing or does not contain enough data, and you must test it for these possibilities as well.

○ Instructor checkpoint 14.9 for evaluating prior work

14.4.8 input_filename.cpp shows how to read in the name of an input file at run-time

```
//input_filename.cpp
//Reads the name of an input file from the user. Only an input
//file is used, but the same approach applies to output files.

#include <iostream>
#include <fstream>
#include <string>  //--Include the "string" header file
using namespace std;

int main()
{
    cout << "\nThis program asks the user for the name of a file, then "
        "displays the contents\nof that file on the screen. It continues "
        "to ask for the names of files and to\ndisplay their contents "
        "until the user terminates the program by entering an\n"
        "end-of-file character. Be sure to enter both the file name and "
        "the extension.\nActually, you can also enter a full pathname to "
        "a file, or use a logical name\nname as part of the file "
        "designation, if your system permits such a thing.\n\n";

    char ch;
    string fileName;  //--Declare a variable of type "string"
    ifstream inFile;

    cout << "Enter name of first file, or "
        "end-of-file to quit entering filenames:\n";
    cin >> fileName;  //--Read in the name of the file
    while (!cin.eof())
    {
        inFile.open(fileName.c_str()); //--Convert C++ string to C-string
        if (inFile) cout << "\nThe contents of " << fileName << " are:\n";
        inFile.get(ch);
        while (!inFile.eof())
        {
            cout << ch;
            inFile.get(ch);
        }
        inFile.close();
        inFile.clear();
        cout << endl << endl;
        cout << "Enter name of next file, or "
            "end-of-file to quit entering filenames:\n";
        cin >> fileName;
    }
    //When we come out of the above loop, cin has been "shut down" by
    //the entry of the end-of-file. If more input from cin is required,
    //cin must be "cleared", or reset, as follows:
    cin.clear();  //Check program behavior with this line removed!
    int i;
    cout << "\nNow enter an integer: ";
    cin >> i;  cin.ignore(80, '\n');
    cout << "The integer entered was " << i << ".\n";

    cout << endl;
}
```

14.4.8.1 What you see for the first time in input_filename.cpp

Before reading the following, take another look at the the program, especially the lines marked with comments introduced by `//--`, and then look for:

- The inclusion of the `string` header file and the declaration of a variable of data type `string`, which is provided by that header file

  `string` *header file and declaration of a string variable*

  Such a variable is capable of holding a "string value". We have known for some time what a string value, or *string literal*, is—"Hello, world!" is one example—but till now we have only had variables that could hold a single character at a time (i.e., `char` variables).

- Reading a string value from the keyboard into a string variable in memory

  Reading a string value into a string variable

  Actually, when we use `cin` to read a string value, we must be sure there is no whitespace in the string we wish to read. This is OK here, as long as the name of the file we are reading does not have any whitespace in it. This is frequently the case, and on some systems it may be forbidden to have whitespace in the name of a file.

- Converting the string representing the name of the file to an alternate (C-string) form so that it can be used by the `open()` function

  Watch out for this!

  This is perhaps the most mysterious part of the process and you might wonder why any "conversion" has to take place. Ideally, we should be able to use

  ```
  inFile.open(fileName);
  ```

  directly. Perhaps, in fact, this will work on your system, but the form we have given should work on all systems. The form to which `fileName` is converted is called a *C-string*, a topic to which we will return when we examine strings in more detail, but we leave it at that for now.

- Using `cin.clear()` and `inFile.clear()` to "clear" the two input streams for re-use after they have been "shut down", or "closed"

  Clearing a file variable for "reuse"

14.4.8.2 Additional notes and discussion on input_filename.cpp

This entire program is really a sneak preview of the `string` data type, since everything else in the program, with the exception of `clear()`, has been seen elsewhere. We do not discuss string variables in any detail here, but introduce the notion just because it is so convenient to be able to have a program read in the name of a file at run-time and then open and work with that file. Otherwise we are forced to "hard wire" the actual name of the file into the source code of the program or use some other (and probably system-dependent) technique.

So, the main thing to take away from this program is the usage pattern, i.e., the steps involved in setting up a variable to receive a filename from the user at run-time and then use that file for either input or output. The program shows only the case of an input file, but the analogous changes needed for the case of an output file should be obvious.

A best practice

The other idea to take away is the fact that once an input stream—the keyboard in this program, but the same applies to an input file stream as well—has encountered an end-of-file it will no longer accept input. This is fine as long as the program is finished reading input at that point. But, if the program must read additional input via that input stream, then the stream must be "cleared". Hence the use of `cin.clear()` in this program, for example.

14.4.8.3 Follow-up hands-on activities for input_filename.cpp

□ Activity 1 Copy, study, test and then write pseudocode for `input_filename.cpp`. Keep the following in mind when you are testing this program. First you should use it only to display textfiles. Second, it will display a textfile of any size but there's not much point in using it to display a file with more lines than will fit comfortably on your screen, since you won't be able to see anything but the last part of the file without scrolling backward.

□ Activity 2 Copy `input_filename.cpp` to `input_filename1.cpp` and then bug it as follows:

a. Remove the compiler directive which includes the `string` header file.

____

b. Remove the declaration of `fileName`.

____

c. Replace `fileName.c_str()` with `fileName`.

____

d. Remove the statement `inFile.close();`.

____

e. Remove the statement `cin.clear();`.

____

Based on shell.cpp

□ Activity 3 Make a copy of `input_filename.cpp` and call it `file_copy.cpp`. Then modify the copy so that it reads in the names of two files. The first is to be the *source*, and the second the *destination*, and the program must copy the contents of the source file to the destination file. The user must be able to do this for as many source/destination file pairs as desired before quitting.

○ Instructor checkpoint 14.10 for evaluating prior work

Module 15

Programmer-defined value-returning functions

15.1 Objectives

- To understand each of the following terms in the context of a programmer-defined value-returning function other than `main`:
  - function *definition*, function *declaration*, and function *prototype*
  - function *header* and function *body*
  - function *return-type* and function *return-value*
  - function *call* (i.e., function *invocation*)
  - *declaration order*, *call order* (i.e., *invocation order*) and *execution order* of a function
  - function *parameter* and function *parameter list*
  - function *interface*
  - *formal parameter* and *actual parameter*

    Sometimes an *actual parameter* is called an *argument*, in which case a *formal parameter* might be called simply a *parameter*.
  - function *pre-conditions* and *post-conditions*
- To understand the analogy between a programmer-defined function that returns a single value and the usual mathematical functions available from the `cmath` library.
- To understand how the principle of *declare before you use* applies to the declaration and use of functions.
- To understand the difference between the *declaration order*, the *call order* (i.e., the *invocation order*), and the *execution order* of a group of functions.

- To understand why the caller of a function may need to pass information to the function at the time of the call, and why the caller may want the called function to "return" information to the caller when it finishes.
- To understand why parameters are needed and how they are used.
- To understand what is meant by a *value parameter* and how it relates to the conceptual notion of an *in-parameter*.
- To understand what is meant by the *scope* of a variable.
- To understand what is meant by the *lifetime* of a variable.
- To understand what is meant by a *local constant* and a *local variable* in a function.
- To learn some style conventions for coding and documenting functions.

15.2 List of associated files

- `ftemp_to_ctemp1.cpp` converts a Fahrenheit temperature to Celsius via a value-returning function (first version).
- `ftemp_to_ctemp2.cpp` converts Fahrenheit temperatures to Celsius via a value-returning function (second version).

15.3 Overview

So far our programs have been quite "simple".

All programs in previous Modules contained just a single *programmer-defined function*, namely the `main` function. All of them were also very small programs, generally under a page in length. Such short programs have the advantage of simplicity, but the disadvantage of not being able to do very much. They can illustrate very well the new features of the language that we wish to discuss, or show us some particular aspect of input, output, or programming style, but beyond that their functionality is limited. To do anything really useful we are going to need larger—and therefore more complex—programs.

But they're getting more complex, and we humans need help when managing complexity.

Because humans are not very good at managing complexity, we need to use whatever tools we can find to help us keep our programs as simple as possible. One of the ways of reducing the complexity inherent in large programs is to divide the code into "logical chunks", each of which is self-contained in the sense that, taken as a whole, it performs a single task, and we need only look at the "logical chunk" of code itself to see what it does and how it does what it does. That is, we don't have to go searching through the rest of the program to see the complete story on the chunk of code that currently has our attention.

Actually, we have been doing this since the beginning, if only in a very simple way, by separating the "logical chunks" of the main function by vertical whitespace (one or more blank lines). We could continue to place all of the code for our bigger and bigger programs in the `main` function, and continue to use whitespace to separate the logically distinct parts of our code. But, if we did so, it would not be too long before things would get out of control and we would find it difficult to understand exactly what this one huge function was doing much of the time. The problem would be, you see, that the one function (`main`) would be doing *too many* things.

Thus we come to the notion that it might be a good idea to add more programmer-defined functions to our programs (over and above `main`), each of which will be designed to perform one simple task well (i.e., each one will *encapsulate* the *procedure* required to perform that task). The idea is to be able to collect the code that performs a particular task into a separate entity (a *programmer-defined function*, in fact), give it a name, and then call upon this entity to perform its task whenever we want to, simply by *using* its name (i.e., by *calling*, or *invoking*, the function). Illustrating just how we do all this, and the terminology, details, and pitfalls of so doing, form a major part of the material in this Module, as Modules 16 and 17.

A function should perform a single task well.

As we will see when we do this, various possibilities arise. A function may be able to perform its task with no additional information from outside itself. If so, it will be a function with no *parameters*. On the other hand, it may be that the function will need one or more additional pieces of information to be supplied to it when we invoke the function, in order to complete its task. If so, then the function will have one or more *parameters* in its *parameter list*. Similarly, when a function finishes its task, it may be required to pass one or more pieces of information back to its caller. Often, if there is just one piece of information to be returned by the function, this will be sent back as the *return-value* of the function. Other times, and almost always when there is more than one value to be returned, values are sent back via *reference parameters*, which is discussed in Module 16.

The rest of a program communicates with a function via (through) its parameter list.

This is quite a bit of terminology, and you should re-read it several times as you look over the various sample programs and the discussions that follow each of them, in this and in the other Modules dealing with functions. It is absolutely critical to the study of programming to understand the passing of information to and from functions via the parameters in their parameter lists.

Two other important concepts you need to be thinking about as you work through these Modules are the *scope* of a variable (the region of a program where the name of the variable is meaningful) and the *lifetime* of a variable (the portion of program execution time during which memory has been allocated for the variable). Both of these notions are actually somewhat more general, in that they apply to C++ entities other than variables, but it is with variables that we shall be most concerned.

Variable scope and variable lifetime

We begin, in this Module, by looking at functions defined by the programmer for calculating and returning a single value. Such functions are analogous to the mathematical functions that we looked at earlier and which we use by including the necessary C++ libraries. Not every function we need is available from a library, however, which is why we often have to write our own.

15.4 Sample Programs

15.4.1 ftemp_to_ctemp1.cpp converts a Fahrenheit temperature to Celsius with a value-returning function

```
//ftemp_to_ctemp1.cpp
//Converts a Fahrenheit temperature to Celsius.

#include <iostream>
using namespace std;


int CelsiusTemp(double fTemp)
{
    const double C_DEGREE_PER_F_DEGREE = double(5) / double(9);

    double cTemp = (fTemp - 32) * C_DEGREE_PER_F_DEGREE;

    if (cTemp >= 0)
        return int(cTemp + 0.5);
    else
        return int(cTemp - 0.5);
}


int main()
{
    cout << "\nThis program converts a Fahrenheit temperature to "
        "an equivalent Celsius value.\nBoth values are reported in "
        "rounded form.\n\n";

    double fTemp;
    cout << "Enter your Fahrenheit value here: ";
    cin >> fTemp;  cin.ignore(80, ’\n’);

    //Round the Fahrenheit temperature to the nearest integer
    if (fTemp >= 0)
        fTemp = int(fTemp + 0.5);
    else
        fTemp = int(fTemp - 0.5);

    cout << "\nFahrenheit Temperature:   " << fTemp
         << "\nEquivalent Celsius Value: " << CelsiusTemp(fTemp);
    cout << endl << endl;
}
```

15.4.1.1 What you see for the first time in ftemp_to_ctemp1.cpp

- A *programmer-defined value-returning function* other than `main`

  Value-returning functions

  This is a function written by the programmer which contains a `return` statement and which "returns" a value to the *caller* of the function. In this case, the function `main` *calls* the function `CelsiusTemp`.

  This method by which a function returns a value to its caller is different from, and must not be confused with, the use of a *reference parameter*, which we discuss in Module 16. Both have their uses and their place in the grand scheme of things.

  Analogy with mathematical functions

  The notion of a *value-returning function* in C++ is consistent with the classical mathematical notion of a function returning a single value when it is *called*, or "used", and you will not go wrong if you think of this analogy and use a value-returning function whenever you need a function to return a single value. The implicit part of this rule of thumb says that whenever you need a function to do something else, do *not* use a value-returning function. In fact, you should then use a *void function*, another concept we deal with in Module 16. Also implicit in this is that if you need a function to return *two* values, you should not be trying to use a value-returning function either, since a value-returning function only returns *one* value.

  `main` *is also a value-returning function.*

  By the way, our `main` function is, of course, also a value-returning function which returns an integer to its caller (i.e., to the operating system).

  Parts of a function

  A programmer-defined *value-returning function*, like any other function, including `main`, is established by a *function definition*, which in turn consists of a *function header*, followed by the corresponding *function body* (a block of code enclosed in braces and containing the executable statements to be performed when the function is *called*, i.e., *invoked*).

  Kinds of parameters

  Each *function header* begins with the *return-type* of the function, i.e., the data type of the *return-value* of the function (`int` in the case of `CelsiusTemp` in this program). This is followed by the programmer-chosen name of the function (`CelsiusTemp`), which in turn is followed by a set of parentheses that are either empty, or enclose the *parameter list* of the function (a list of data-type/variable-name pairs, separated by commas if there are two or more pairs). In this function there is only one parameter (`fTemp`) of type `double`. Each variable-name in one of these pairs is a *parameter* of the function, and is used either to pass information *into* the function when the function is called (an *in-parameter*), to send information *back* from the function when the function has finished executing (an *out-parameter*), or both (an *inout-parameter*).

In-parameters will normally be value parameters.

- A *value parameter*, which is a conceptual *in-parameter*

  Information may be passed *into* a value-returning function when it is called so that the information can be used *inside* the function *by* the function to compute the value it will return. A parameter used in this way is, conceptually, an *in-parameter* since it sends information *into* the function, and will normally be implemented by using a *value parameter*.

  When a function uses a *value parameter*, it means that the function will make a *copy* of the value that is passed to the function when the function is used. The original value is therefore not affected by anything that happens to the passed value "inside" the function. By default, all parameters are value parameters.

  In this program, the value contained in `fTemp` in `main` is passed into the function `CelsiusTemp` via a value parameter when `main` calls `CelsiusTemp`.

Formal and actual parameters

- Instances of both a *formal parameter* and an *actual parameter*

  The distinction between these two terms is relatively easy to make. A parameter that appears in a function *definition* is a *formal parameter*, while a parameter that appears in a function *call* is an *actual parameter*.

  The *formal parameters* in the parameter list of the function definition indicate what information must be passed when a call is made to that function. The corresponding *actual parameters* (which may, in the case of a value parameter, be literal values, variables or even expressions) supply the *actual* information to be used when a call is *actually* made.

  In this program the formal parameter and the actual parameter of the function `CelsiusTemp` have the same name (namely, `fTemp`), but this does not have to be the case.

- The definition of a named constant *within* a programmer-defined function

  Named constants within functions

  Scope of a named constant

  Defining a constant inside a function makes that constant *local* to the function, and *not accessible* outside that function. This is the way it should be since the constant is only needed and used within the function. This situation is also expressed by saying that the *scope* of the constant is restricted to the function.

  In this program the named constant is `C_DEGREE_PER_F_DEGREE`.

Local variables within functions

- The declaration and initialization of a *local variable* within a programmer-defined function

  The same comments made above about the named constant apply again here. Note as well that both the named constant and the variable defined locally within the function only exist during the call to the function. We express this by saying that the *lifetime* of the local variable `cTemp`, for example, is the duration of the function call. That variable did not exist before the function call, and does not exist after the function call has terminated.

In this program the name of the *local variable* in the `CelsiusTemp` function is `cTemp`.

- A *function call* to a programmer-defined value-returning function

  The function `CelsiusTemp` that is *defined* in this program is also *called* (i.e., *invoked*) once in the program, by the `main` function. In general, of course, any function (except `main`) can be called any number of times, and a given function may be called by any other function (not just by `main`) if the situation requires it.

  Using a function means "calling" or "invoking" the function.

  If the program is structured like this one (i.e., with the function definition preceding `main`), then the requirement of *define before using* is clearly satisfied. As we will see in the next sample program of this Module, the use of function prototypes alters this requirement somewhat, to *declare before using*, and is useful for other reasons, so in fact we normally do not structure our programs in the way we see here. Stay tuned for the update.

  Understand the principle of having to "describe" something before using it, by definition or declaration.

15.4.1.2 Additional notes and discussion on ftemp_to_ctemp1.cpp

Some remarks are in order on how (or where) a value-returning function is used (or called). If you look at where the call to the value-returning function in `ftemp_to_ctemp1.cpp` appears, you will see that it appears in the middle of a `cout` statement as one of the items to be output. Why is this? It's because the call to the function simply *returns a value* (it is, after all, a value-returning function) and it is *this value* that the `cout` statement outputs.

Placement of the call to a value-returning function

Which is *not* to say that value-returning functions *only* appear in `cout` statements. However, what *is* true is that value-returning functions should *only* be called (i.e., should only appear) in those locations where it would make sense for any value of the data type that the function will return to appear.

The name of a value-returning function, and the purpose for which the value returned is likely to be used, are analogous to how a variable or constant is named and how a variable or constant is used. Thus the naming convention applied to value-returning functions is similar to that for naming variables—all lower case letters except for capitals at the start of the second and any subsequent words in the variable name—but a function name also *starts* with a capital letter, according to our particular naming conventions.

Naming and capitalization conventions for a value-returning function

15.4.1.3 Follow-up hands-on activities for ftemp_to_ctemp1.cpp

□ Activity 1 Copy, study, test and then write pseudocode and draw a design tree diagram for `ftemp_to_ctemp1.cpp`. When testing, try at least the following Fahrenheit values as input: 212, 32, 0, 98.6, and –40

□ Activity 2 Copy `ftemp_to_ctemp1.cpp` to `ftemp_to_ctemp1a.cpp` and bug it as follows:

a. Replace `int` by `char` as the return-type of the function in the function definition.

   ____

b. Replace `double(5) / double(9)` by `5/9` in line 10.

   ____

c. Interchange the order of the two functions.

   ____

d. Remove the definition of the function `CelsiusTemp`.

   ____

e. Replace each occurrence of the identifier `fTemp` in the definition of the function `CelsiusTemp` with `fahrTemp`.

   ____

□ Activity 3 Make a copy of `ftemp_to_ctemp1.cpp` called `ctemp_to_ftemp1.cpp` and revise the copy so that it works in an analogous way but performs conversions in the other direction, i.e., from Celsius to Fahrenheit. Be sure to make *all* appropriate changes, including identifier names and comments.

○ INSTRUCTOR CHECKPOINT 15.1 FOR EVALUATING PRIOR WORK

15.4.2 ftemp_to_ctemp2.cpp contains a function prototype and function documentation but performs just like ftemp_to_ctemp1.cpp

```
//ftemp_to_ctemp2.cpp
//Converts a Fahrenheit temperatures to Celsius.

#include <iostream>
using namespace std;

int CelsiusTemp(double fahrenheit);


int main()
{
    cout << "\nThis program converts a Fahrenheit temperature to "
        "an equivalent Celsius value.\nBoth values are reported in "
        "rounded form.\n\n";

    double fTemp;
    cout << "Enter your Fahrenheit value here: ";
    cin >> fTemp;  cin.ignore(80, '\n');

    //Round the Fahrenheit temperature to the nearest integer
    if (fTemp >= 0)
        fTemp = int(fTemp + 0.5);
    else
        fTemp = int(fTemp - 0.5);

    cout << "\nFahrenheit Temperature:   " << fTemp
         << "\nEquivalent Celsius Value: " << CelsiusTemp(fTemp);
    cout << endl << endl;
}


int CelsiusTemp(/* in */ double fahrTemp)
//Pre:  "fahrTemp" contains a valid Fahrenheit temperature,
//      which may be an integer or real quantity
//Post: Return-value of the function is the equivalent Celsius value,
//      rounded to the nearest integer
{
    const double C_DEGREE_PER_F_DEGREE = double(5) / double(9);

    double cTemp = (fahrTemp - 32) * C_DEGREE_PER_F_DEGREE;

    if (cTemp >= 0)
        return int(cTemp + 0.5);
    else
        return int(cTemp - 0.5);
}
```

15.4.2.1 What you see for the first time in ftemp_to_ctemp2.cpp

Function prototypes

- A *function prototype*, which consists of the function header followed by a semi-colon

Placement of function prototypes

Function prototypes are generally placed near the beginning of the file (before `main`) and effectively take care of the "declare before use" requirement without having the full function definition in that location. The function header (without the body) is all the compiler needs to have seen in order to tell whether any later call to the function is properly made. Likewise, the prototype of a function will be informative to a human reader as well.

Function documentation

- Function *documentation*, including *pre-conditions* and *post-conditions* in comments following the function header, as well as a comment in the parameter list indicating the direction of flow of the information contained in the parameter

Pre-conditions and post-conditions

The *pre-conditions* and *post-conditions* of a function are a critical part of that function's documentation. The pre-conditions tell the user of the function what must be true before the function is called. The post-conditions tell the user of the function what will be true after the function has finished executing.

Function interface

The *interface* to a function consists of its name, its return-type, its parameter list, and its pre-conditions and post-conditions.

C-style comments are required in this situation.

- Use of a C-style comment in a location where a C++ comment (with `//`) could not be used

15.4.2.2 Additional notes and discussion on ftemp_to_ctemp2.cpp

Many different programs may have exactly the same user interface.

This program behaves exactly like its predecessor in `ftemp_to_ctemp1.cpp` from the user's point of view. In fact, the *user interface* (what the user sees) is identical. Any differences, then, are behind the scenes, but no less important.

Do note, however, the name of the formal parameter in the `CelsiusTemp` function this time, as well as the parameter name used in the prototype. The name used for the Fahrenheit temperature is different in the three different situations in which it crops up: in `main`, in the definition of the `CelsiusTemp` function, and in the prototype for the `CelsiusTemp` function. The three names used could also be identical.

15.4.2.3 Follow-up hands-on activities for ftemp_to_ctemp2.cpp

□ Activity 1 Test `ftemp_to_ctemp2.cpp` with the same values used for the program in `ftemp_to_ctemp1.cpp` to make sure the behavior is the same. Here are those values: 212, 32, 0, 98.6, and –40

□ Activity 2 Copy `ftemp_to_ctemp2.cpp` to `ftemp_to_ctemp2a.cpp` and bug it as follows:

a. Replace `int` by `double` as the return data type of the function in both the function prototype and the function definition.

____

b. Remove the function prototype.

____

c. Remove the definition of the function `CelsiusTemp`. What is different this time when compared with removal of the function definition in the previous sample program `ftemp_to_ctemp1.cpp`. Can you explain the difference?

____

____

____

____

d. Change `double` in the parameter list of the function prototype to `int`.

____

e. Make the name the same for the Fahrenheit temperature in `main`, in the definition of the `CelsiusTemp` function, and in the prototype for the `CelsiusTemp` function.

____

○ INSTRUCTOR CHECKPOINT 15.2 FOR EVALUATING PRIOR WORK

Module 16

Programmer-defined void functions

16.1 Objectives

- To understand how a programmer-defined function is used to implement *procedural abstraction* by *encapsulating* the code that performs a single task.
- To understand the C++ reserved word `void`.
- To understand that the terms *procedure* and *void function* will mean the same thing in our context.
- To review each of the following terms and concepts—previously encountered in the context of value-returning functions—in the new context of void functions:
  - function *definition*, function *declaration*, and function *prototype*
  - function *call* (i.e., function *invocation*)
  - function *header*, function *return-type*, and function *return-value*
  - function *body*
  - function *interface*
  - *declaration order*, *call order* (i.e., *invocation order*) and *execution order* of a function
  - *parameter*, *parameter list*, *formal parameter*, and *actual parameter*
  - *pre-conditions* and *post-conditions*
  - The concept of *declare before use* as it relates to functions
- To understand the differences and similarities between a *value parameter* and a *reference parameter*, and to learn how to use each.

Direction of information flow in functions

- To understand the concept of the *direction of information flow* to and from a function, as well as the conceptual difference between an *in-parameter*, an *out-parameter* and an *inout-parameter*, and to learn how each one is implemented in C++.

Two ways for a function to return a value

- To understand the distinction between the two different mechanisms by which a function can return a value to its caller: via the *return-value* of the function (in the case of a *value-returning function*), or via a *reference parameter* in its parameter list (in the case of a *void function*).

Placement of the call to a void function

- To understand that when a *void function* is called, it appears as a separate statement, on a line by itself, together with its actual parameters, *unlike* a *value-returning function*, which must appear in a location where a single value (having the same type as the function's return-type) can appear.

Function overloading

- To understand what is meant by *function overloading* and to learn when and how to overload a function.

16.2 List of associated files

- `say_hi1.cpp` doesn't use functions.
- `say_hi2.cpp` shows void functions without parameters.
- `say_hi3.cpp` shows a void function with value parameters.
- `say_hi4.cpp` shows void functions with reference parameters.
- `swap.cpp` illustrates function overloading and the algorithm which exchanges two variable values.

16.3 Overview

Compare and contrast void functions with value-returning functions

In the previous Module we looked at *value-returning functions*, i.e., functions whose sole task is *to compute and return a single value.* Although the kind of value returned by a function can vary widely, as can the method used to compute it, conceptually the task performed by a value-returning function is a very simple and specialized one.

In this Module we look at functions that perform more general tasks, or *procedures.* In C++ these become *void functions*, i.e., functions whose return type is `void` (a C++ reserved word) and which, though they *may* (or may not) return values to their caller, do *not* do so in the same way as value-returning functions. If a void function does return a value, it must do so by using a certain kind of parameter called a *reference parameter*, and there may be as many of these as you like.

16.4 Sample Programs

The first four of the following sample programs form a coordinated sequence that gradually introduces void functions, first without parameters, then with *value parameters*, and finally with *reference parameters*. What each program does is simplicity itself—it displays a greeting, kind of like the "Hello, world!" program—but we *need* a simple situation like this in which to illustrate these new and important concepts for the first time, so that the details of the situation itself do not get in the way. So, we begin by displaying a simple greeting, then display it to different "individuals" and in different positions on the output line, and use these variations to illustrate the new concepts.

A gentle introduction to void functions

16.4.1 say_hi1.cpp doesn't use functions

```
//say_hi1.cpp
//Displays a greeting.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program displays a greeting.\n";

    cout << "\nHi there!\n";
    cout << endl;
}
```

16.4.1.1 What you see for the first time in say_hi1.cpp

This program contains no new C++ features, and even though it contains no void functions itself, it serves as a convenient starting point for our introduction and discussion of void functions.

Just a starting point, with no functions at all

16.4.1.2 Additional notes and discussion on say_hi1.cpp

The program may be thought of as performing two tasks: first, it describes what it is going to do; second, it then does what it said it was going to do. Each of these two "tasks" is going to be performed by a task-performing *void function* in our next version of the program. We shall see that this reduces the "complexity" of the `main` function by "hiding" the details of each task in its corresponding function definition and replacing those details in `main` by simple function calls.

16.4.1.3 Follow-up hands-on activities for say_hi1.cpp

□ Activity 1 Copy, study, test and then write pseudocode and draw a design tree diagram for `say_hi1.cpp`.

○ Instructor checkpoint 16.1 for evaluating prior work

16.4.2 say_hi2.cpp shows void functions without parameters

```
//sayhi2.cpp
//Displays a greeting.

#include <iostream>
using namespace std;


void DescribeProgram();
void DisplayGreeting();


int main()
{
    DescribeProgram();
    DisplayGreeting();
    cout << endl;
}


void DescribeProgram()
//Pre:  The cursor is at the left margin.
//Post: The program description has been displayed,
//      preceded and followed by at least one blank line.
{
    cout << "\nThis program displays a greeting.\n\n";
}


void DisplayGreeting()
//Pre:  The cursor is at the left margin.
//Post: A one-line greeting has been displayed,
//      preceded and followed by at least one blank line.
{
    cout << "\nHi there!\n\n";
}
```

16.4.2.1 What you see for the first time in say_hi2.cpp

`void` *return-type for a function*

- The use of the C++ reserved word `void`

  The reserved word `void` is used as the return-type of a function to indicate specifically that the function does *not* return any value (in the sense that a value-returning function returns a value), and instead "performs a task". The terminology is potentially confusing, however, since we still refer to a function as having a "return-type of `void`".

Void function with no parameters

- The definition of a *void function* with *no* parameters, i.e., a function whose return-type is `void` and whose parameter list is empty (actually, two of them: `DescribeProgram` and `DisplayGreeting`)

- The prototype of a void function with no parameters (two of these as well, one for each of the functions `DescribeProgram` and `DisplayGreeting`)

  Prototypes

- The listing, in comments following the function header, of pre-conditions and post-conditions for void functions having no parameters

  Pre-conditions and post-conditions

- The call to, or invocation of, or use of, a void function with no parameters

  Note that to use, or call, a void function without parameters, you simply place the name (followed by parentheses and a semi-colon) on a line by itself as a separate executable statement, and it is at that point in the "action flow" of the program that the particular task "encapsulated" by that function is performed.

  How to call a void function with no parameters.

- A *naming convention* which comes to you very highly recommended: The name of every void function *must* begin with a verb (as in `Describe...`, `Get...`, `Draw...`)

  A very important programming style rule

  This is a very important programming style requirement, and there is a very good reason for it. Just think of it this way: It is the job of a void function to *do* something, the name of a function should tell you what the function *does*, and without a *verb* in the name the most the name can tell you is what the function is *about*, *not* what it *does*. You will find that starting the names of void functions with a verb will help to reduce the amount of commenting needed to explain your code. A final reminder: The name of a value-returning function will generally *not* start with a verb. Note, however, that the capitalization convention for function names is the same for void functions as for value-returning functions.

16.4.2.2 Additional notes and discussion on say_hi2.cpp

One of the first things you should note about this program is this: Although the source code of the program is quite different from that of `say_hi1.cpp`, the *user interface* (i.e., what the user sees when the program runs and the way in which the user interacts with the program when it is running) is *exactly* the same. This suggests that there may be many different programs which appear to the user to be the *same* program, and this is in fact the case. We sometimes describe such programs by saying that they *have the same user interface.*

Different programs may have the same user interface.

Thus, one view of program development might be this: First, create the "best" user-interface design that you can manage. Then make sure that your program is the "best" of all the possible programs that would provide this user interface. The devil is, as always, in the details, and this is an oversimplification, of course, but the general approach is sound.

Note the overall structure of the program: The prototypes of the two functions whose definitions *follow* `main` are placed *before* `main`. This is the same kind of program structure we saw when using a value-returning function and its prototype in Module 15, and it is used here for the same reasons. First, when the prototype of a function appears before `main`, the compiler can verify

Note the structure of the program.

that a call to that function in main has been made correctly. Second, a human can get a good idea of what the program does without having to wade through the details of each function called by main. Critical to this second reason, of course, is the proper naming of all void functions with a starting verb so that the function calls "tell the story" of what the program does.

Note too the style of this particular main function. Here we have not followed our usual practice of separating the "logical chunks" of code by vertical white space (blank lines). There is a simple reason for this: In this case each "logical chunk" of code is a single line, so there is simply nothing to be gained by vertical separation; in fact, main is probably somewhat more readable without it.

16.4.2.3 Follow-up hands-on activities for say_hi2.cpp

□ Activity 1 Copy, study, test and then write pseudocode and draw a design tree diagram for say_hi2.cpp. In drawing the design tree diagram, use the names of the void functions to indicate the subtasks.

□ Activity 2 Copy say_hi2.cpp to say_hi2a.cpp and bug it as follows:

a. Omit the prototype of the function DisplayGreeting.

____

b. Omit the definition of the function DisplayGreeting.

____

c. Interchange the order of the two function prototypes.

____

d. Interchange the order of the two function definitions (other than main).

____

○ INSTRUCTOR CHECKPOINT 16.2 FOR EVALUATING PRIOR WORK

16.4.3 say_hi3.cpp shows a void function with value parameters

```
//sayhi3.cpp
//Displays a greeting to three particular individuals, using their
//initials, and positioned at various places on the output line.

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


void DescribeProgram();
void DisplayGreeting(int column, char firstChar, char lastChar);


int main()
{
    DescribeProgram();

    DisplayGreeting(12, ’J’, ’C’);
    DisplayGreeting(1, ’P’, ’S’);
    DisplayGreeting(68, ’A’, ’F’);
    cout << endl;
}


void DescribeProgram()
//Pre:  The cursor is at the left margin.
//Post: The program description has been displayed,
//      preceded and followed by at least one blank line.
{
    cout << "\nThis program displays a greeting to three individuals, "
        "using their\ninitials, and positioned at various locations on "
        "the output line.\n\n";
}


void DisplayGreeting(/* in */ int column,
                    /* in */ char firstChar,
                    /* in */ char lastChar)
//Pre:  "column", "firstChar" and "lastChar" have been initialized, with
//      1<=column<=68, ’A’<=firstChar<=’Z’, and ’A’<=lastChar<=’Z’.
//Post: A greeting has been displayed, using the initials in
//      "firstChar" and "lastChar", starting in position "column",
//      and preceded and followed by at least one blank line.
{
    cout << endl;
    cout << setw(column-1) << ""
         << "Hi there, " << firstChar << lastChar << "!\n";
}
```

16.4.3.1 What you see for the first time in say_hi3.cpp

Value parameters ...

... are in-parameters.

- A void function having *value parameters*
- Conceptual *in-parameters*, with documentation, in the parameter list of a void function
- A call to a void function having value parameters
- The use of pre-conditions and post-conditions in a void function with parameters
- The typical formatting of a function with several parameters

16.4.3.2 Additional notes and discussion on say_hi3.cpp

The formal parameters in a function definition are replaced by actual parameters in any call to that function.

This program shows how information may be passed *into* a void function when it is called so that the information can be used *inside* the function *by* the function as it performs its assigned task. The *formal parameters* in the parameter list of the function prototype and the function definition indicate what information must be passed when a call is made to that function. The corresponding *actual parameters* (integer and character *literal values* in this case) supply the *actual* information to be used when a call is *actually* made. In this particular program there are three different calls to the function `DisplayGreeting`, and a different set of actual parameters is supplied for each call.

Contrast the style of `main` is this program with the style of `main` in the previous sample program. Note that vertical whitespace is re-introduced here for separation, and it should be clear that it doesn't take much in the way of additional code "complexity" before vertical separation begins to enhance readability.

16.4.3.3 Follow-up hands-on activities for say_hi3.cpp

□ Activity 1 Copy, study, test and then write pseudocode and draw a design tree diagram for `say_hi3.cpp`. In drawing the design tree diagram, use the names of the void functions to indicate the subtasks.

□ Activity 2 Copy `say_hi3.cpp` to `say_hi3a.cpp` and bug it as follows:

a. Interchange the order of the two function prototypes.

   ____

b. Interchange the order of the two function definitions (other than `main`).

   ____

c. Remove the second formal parameter from the prototype of the function `DisplayGreeting`.

____

d. Remove the second formal parameter from the definition of the function `DisplayGreeting`.

____

e. Remove the first actual parameter from the first call to `DisplayGreeting`.

____

f. Remove the second or third actual parameter from the second call to `DisplayGreeting`.

____

g. Change the data type of `first` from `char` to `int` in the prototype of `DisplayGreeting`.

____

h. Change the data type of `column` from `int` to `char` in the definition of `DisplayGreeting`.

____

i. This change is meant to emphasize that when values are supplied as actual parameters, variables and expressions can also be used, as well as literal values as shown in the sample program listing.

An actual value parameter can be a literal value, a variable, or even an expression.

This last point is a key one to note and remember.

So, replace the line

`DisplayGreeting(12, ’J’, ’C’);`

with the *two* lines

```
char testChar = ’A’;
DisplayGreeting(2*8-4, char(’A’+3), testChar);
```

____

○ Instructor checkpoint 16.3 for evaluating prior work

16.4.4 say_hi4.cpp shows void functions with reference parameters

```
//say_hi4.cpp
//Displays a greeting to an individual, using his or her
//initials, and (possibly) indented from the left margin.

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


void DescribeProgram();
void GetPosition(int& column);
void GetInitials(char& firstChar, char& lastChar);
void DisplayGreeting(int column, char firstChar, char lastChar);


int main()
{
    int column;
    char firstChar;
    char lastChar;

    DescribeProgram();

    GetPosition(column);
    GetInitials(firstChar, lastChar);

    DisplayGreeting(column, firstChar, lastChar);

    cout << endl;
}


void DescribeProgram()
//Pre:  The cursor is at the left margin.
//Post: The program description has been displayed,
//      preceded and followed by at least one blank line.
{
    cout << "\nThis program displays a greeting to some individual, "
        "using his or her initials\nand positioned at a location on "
        "the output line chosen by the user.\n\n";
}


void GetPosition(/* out */ int& column)
//Pre:  none
//Post: "column" contains an integer from 1 to 68 (inclusive)
//      entered by the user, and the input line has been cleared.
{
    cout << "Enter column (1 to 68) where greeting is to start: ";
    cin >> column;  cin.ignore(80, '\n');  cout << endl;
}



```

```
void GetInitials(/* out */ char& firstChar, /* out */ char& lastChar)
//Pre:  none
//Post: "firstChar" and "lastChar" each contain a capital letter
//      entered by the user, and the input line has been cleared.
{
    cout << "Enter the (capital) initials of the person to greet: ";
    cin >> firstChar >> lastChar;  cin.ignore(80, ’\n’);  cout << endl;
}


void DisplayGreeting(/* in */ int column,
                     /* in */ char firstChar,
                     /* in */ char lastChar)
//Pre:  "column", "firstChar" and "lastChar" have been initialized, with
//      1<=column<=68, ’A’<=firstChar<=’Z’, and ’A’<=lastChar<=’Z’.
//Post: A greeting has been displayed, using the initials in
//      "firstChar" and "lastChar", starting in position "column",
//      and preceded and followed by at least one blank line.
{
    cout << endl;
    cout << setw(column-1) << ""
         << "Hi there, " << firstChar << lastChar << "!\n\n";
}
```

16.4.4.1 What you see for the first time in say_hi4.cpp

- A void function having *reference parameters*

  A reference parameter is indicated by placing an *ampersand character* (&) after the type name for that parameter. The best way to think of a reference parameter is this: it is an *alias* (i.e., another name) for whatever actual parameter is passed (in the corresponding parameter position, of course) when the function is called. In this particular program, the “alias” is actually the same as the original name in each case, but this need not be true, and often will not be true.

  Reference parameters ...

- A conceptual out-parameter,with documentation, in the parameter list of a void function

  Because a reference parameter is just another name for an already existing location in memory, any change made in that location by the function will be reflected in the state (i.e., the value) of the variable that was passed as the actual parameter when the function is finished executing.

  ... are often, but not always, conceptual out-parameters.

- A call to a void function having reference parameters

  The *actual parameter* passed in the case of a reference parameter *must* be a *variable*. That is, an actual reference parameter *cannot* be a literal value or an expression, unlike the situation in the case of a value parameter.

  Actual reference parameters must be variables.

- A choice of styles in parameter-list formatting, either of which is OK (though a long parameter list may force one style upon you)

16.4.4.2 Additional notes and discussion on say_hi4.cpp

Information may be returned by a function to its caller via a reference parameter but not by a value parameter.

This program shows how information may be *returned from* or *passed back from* or *passed out of* a void function to the caller of that void function, via a *reference parameter* in the function parameter list.

Once again, the data types of the *formal parameters* in the parameter list of the function indicate what kind of information (values) will be passed back when a call to that function is finished executing.

After the call, the variables which are supplied as the *actual* parameters by the caller of the function will contain the information (values) supplied by the function. Those values can then be used by the caller, or simply passed along to another function, as is the case in this sample program.

Compare once again the style (in particular, the vertical spacing used) in `main` with that of the previous two sample programs. Compare also the formatting used for the parameter list in the definitions of the functions `GetInitials` and `DisplayGreeting`. The style used for `GetInitials` is OK for one or two parameters, but as soon as there are three or more it is better (i.e., more *readable*) to use the style shown in `DisplayGreeting`.

16.4.4.3 Follow-up hands-on activities for say_hi4.cpp

□ Activity 1 Copy, study, test and then write pseudocode and draw a design tree diagram for `say_hi4.cpp`. In drawing the design tree diagram, use the names of the void functions to indicate the subtasks.

□ Activity 2 Copy `say_hi4.cpp` to `say_hi4a.cpp` and bug it as follows:

a. Replace `int&` by `int` in the prototype for the function `GetPosition`.

____

b. Replace `int&` by `int` in the definition of the function `GetPosition`.

____

c. Replace `int&` by `int` in the prototype *and* in the definition of the function `GetPosition`.

____

d. Replace `char first` by `char& first` in the prototype for the function `DisplayGreeting`.

____

e. Replace `char first` by `char& first` in the definition of the function `DisplayGreeting`.

____

f. Replace `char first` by `char& first` in both the prototype and in the definition of the function `DisplayGreeting`.

____

g. In `DisplayGreeting` replace each occurrence of `column` by `position`, each occurrence of `first` by `ch1`, and each occurrence of `last` by `ch2`.

____

h. This program contains four function definitions other than `main` and four corresponding prototypes. The four prototypes can be arranged in $4! = 24$ different orders, and so can the function definitions, which gives a total of $24 * 24 = 576$ different overall orderings for the definitions and prototypes combined. Try at least two of these to see if they still work. That is, rearrange both the prototype and function definitions into some other order, then try to compile, link, run and test the program again. (Be sure to keep all of the function prototypes in a group and located *before* `main`, and all of the function definitions in a group and located *after* `main`.)

Record your findings below, as well as your answers the following questions: Did one of your alternate orderings not work? If so, why not? If not, can you find some other ordering which does not work? If so, what is it? If not, why not?

____

____

____

____

____

____

□ Do we really need two functions to get input? Make a copy of the file `say_hi4.cpp` called `say_hi4b.cpp` and modify the copy so that the two input functions are replaced with a single input function that accomplishes the same task as the two in the original program. Think about what name you will use for the single function, and also think about whether the use of a single function in this situation is "better" or "worse" than the use of two functions, and what arguments you might use to support either choice.

○ Instructor checkpoint 16.4 for evaluating prior work

16.4.5 swap.cpp illustrates function overloading and the algorithm which exchanges two variable values

```
//swap.cpp
//Illustrates function overloading and the standard
//algorithm for exchanging the values of two variables.

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


void DescribeProgram();
void Swap(int& i1, int& i2);
void Swap(char& c1, char& c2);
void Pause();
void Pause(int indentLevel);


int main()
{
    DescribeProgram();

    //The (deliberate) naming of the following variables here in "main"
    //and the naming of the formal function parameters in the function
    //definitions below emphasizes that formal function parameters may
    //(or may not) have the same names as the corresponding actual
    //parameters in subsequent parmeter calls.
    int i1, i2;
    char first, second;

    cout << "Enter two integers: ";
    cin >> i1 >> i2;  cin.ignore(80, '\n');  cout << endl;
    cout << "In the order entered, the integers are: ";
    cout << i1 << ", " << i2 << endl;
    Pause(7);
    Swap(i1, i2); //Actual/formal parameters have same names
    cout << "And here are the two integers reversed: ";
    cout << i1 << ", " << i2 << endl;
    Pause(7);

    cout << endl;

    cout << "Enter two characters: ";
    cin >> first >> second;  cin.ignore(80, '\n');  cout << endl;
    cout << "In the order entered, the characters are: ";
    cout << first << ", " << second << endl;
    Pause();
    cout << endl;
    Swap(first, second); //Actual/formal parameters have different names
    cout << "And here are the two integers reversed: ";
    cout << first << ", " << second << endl;
    Pause();

    cout << endl;
}

```

```

void DescribeProgram()
//Pre:  The cursor is at the left margin.
//Post: The program description has been displayed,
//      preceded and followed by at least one blank line.
{
    cout << "\nThis program illustrates the standard \"two-value "
        "exchange algorithm\" and\nfunction overloading.  It is "
        "necessary to study both the source code and\nthe output "
        "simultaneously, but in the meantime, just follow "
        "instructions.\n\n";
}


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;
}


void Swap(/* inout */ char& c1, /* inout */ char& c2)
//Pre:  "c1" and "c2" have been initialized.
//Post: The values of "c1" and "c2" have been swapped.
{
    char temp = c1;
    c1 = c2;
    c2 = temp;
}


void Pause()
//Pre:  The input stream cin is empty.
//Post: The program has displayed a one-line message beginning in
//      column 1 and then paused, and the user has pressed Enter.
{
    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’);
}


void Pause(/* in */ int indentLevel)
//Pre:  The input stream cin is empty, and 0 <= indentLevel <= 50.
//Post: The program has displayed a one-line message beginning in
//      column indentLevel+1 and then paused. The user has pressed
//      Enter, and a blank line has been inserted in the output stream.
{
    cout << setw(indentLevel) << "" << "Press Enter to continue ... ";
    cin.ignore(80, ’\n’);
    cout << endl;
}
```

16.4.5.1 What you see for the first time in swap.cpp

- Two void functions with the same name but different parameter lists (In fact, the program contains two instances of this: two versions of `Swap` and two of `Pause`.)

  Overloaded functions

  Whenever we have this situation, we describe it by saying that the function name is *overloaded*, which simply means that the same name is being used for two (or more) functions. This is no problem for the C++ compiler as long as the parameter lists of each function are distinct, since the compiler will then use the parameter lists to distinguish the functions and decide which function is to be used in any particular function call.

- A conceptual *inout-parameter*

  Inout-parameters must be reference parameters.

  Each of the two parameters in each of the two `Swap` functions is an *inout-parameter*, since each parameter takes one value into the function and sends another value back out. Note that a *reference parameter* must be used to do this since information is being returned *from* the function, as in the case of an *out-parameter*.

Remember this!

- The use of the algorithm for exchanging the values of two variables

Encapsulation

- The *encapsulation* into a procedure (i.e., into a void function) of the "task" of causing a program to pause and wait for the user to press Enter to continue

16.4.5.2 Additional notes and discussion on swap.cpp

When to use overloaded functions

Function overloading, as illustrated in this program, should be used sparingly and carefully. The kind of situation it should be used in is illustrated here by the `Swap` function, i.e., a situation where the *same* operations are being performed on *different* kinds of data. Note that we really didn't need to overload the `Pause` function, since a call to `Pause(0)` gives the same effect as a call to `Pause()`, except for the blank line inserted by `Pause(0)`.

16.4.5.3 Follow-up hands-on activities for swap.cpp

□ Activity 1 Copy, study, test and then write pseudocode and draw a design tree diagram for `swap.cpp`. In drawing the design tree diagram use the names of the void functions to indicate the subtasks.

□ Activity 2 What problem with drawing the design tree diagram crops up here for the first time, and how would you suggest dealing with it?

____

____

____

□ Activity 3 Copy `swap.cpp` to `swap1.cpp` and bug it as follows:

a. Replace the `Pause(7)` in line 33 by `Pause(60)`.

____

b. Replace `int&` by `int` everywhere in both the prototype and the definition of the first version of the function `Swap`. Explain below why you get what you get when you run the program.

____

____

c. Replace the first `int&` (i.e., the one for the first parameter) by `int` in both the prototype and the definition of the first version of the function `Swap`. Explain below why you get what you get when you run the program.

____

____

d. Replace the second `int&` (i.e., the one for the second parameter) by `int` in both the prototype and the definition of the first version of the function `Swap`. Explain below why you get what you get when you run the program.

____

____

e. Remove the call to `cin.ignore(80, '\n')` in line 30.

____

____

f. Remove the call to `cin.ignore(80, '\n')` in line 42.

____

____

○ INSTRUCTOR CHECKPOINT 16.5 FOR EVALUATING PRIOR WORK

Module 17

A third look at program development: program modularity, program structure, information flow, stubs and drivers

17.1 Objectives

- To understand the *structured approach* (also called the *procedural approach*) to the *program development process*, and its relationship to *top-down design with step-wise refinement*.
- To understand how functions fit into the top-down design with step-wise refinement approach to procedural program development.
- To revisit the notion of a *design-tree diagram*, in the context of functions, with and without *data-flow information*.
- To learn what is meant by a *stub* and a *driver*, and to understand how they are used in program development.
- To understand what is meant by the term *interface*, and to be able to distinguish between the *interface to a program* (i.e., the *user interface*) and the *interface to a function*.
- To learn some additional style conventions for structuring programs.

- To work through in detail the steps involved in developing a program consisting of several functions by applying the structured programming approach via top-down design with step-wise refinement.

17.2 List of associated files

- `stub_driver.cpp` contains a generic shell program with stubs and a driver.

17.3 Overview

In Modules 15 and 16 we examine both value-returning functions and void functions. We point out thee that as soon as a program becomes non-trivial (which, as a rough guide, we may think of as "longer than a page or so") it becomes advantageous to break the program up into functions, each of which encapsulates the necessary code to perform some task—calculating one value, in the case of a value-returning function, and some more general task, usually, in the case of a void function—and then calling these functions as often as needed and in the proper order to get the overall task done.

Program modularity

We saw a number of examples, all of them quite small, which contained one or more programmer-defined functions in addition to `main`, and were able to appreciate, to some extent at least, how increasing the *modularity* of a program by breaking it up into *modules* (or, to be more specific in our case, into functions) enhanced the readability of the program by isolating the detailed code for each task in a *function definition* and replacing the detailed code by a *function call.*

Program structure

The *program structure* in each case was the same: included header files first, followed by the function prototypes, then `main`, and finally the definitions of the functions other than `main`. We shall retain that general structure as our programs grow to include more and more functions, with the added proviso that the function definitions other than `main` that *follow* `main` will appear *in the same order* as the prototypes of those functions that *precede* `main`, which is yet another attempt to avoid confusion and enhance readability.

Information flow via parameters

We also saw how parameters could be used to move information into and out of functions as required. Pre-conditions and post-conditions told us exactly what was involved in each case, and we also added comments in all parameter lists to tell us whether each parameter was an in-parameter, an out-parameter, or an inout-parameter.

A difficulty with that presentation, however, is this: Like the rest of what we have seen, it showed us certain things that we need to know and which will be very useful as we proceed, but it showed us just *the end result, and not how that result was arrived at.* That is, we began by looking at a "finished" program, and asked what it told us, or what it showed us that we hadn't seen before. What we have not had, thus far, is a serious look at the details of the process by which such a program, i.e., a program consisting of several functions, comes into existence.

In this Module we come to grips, finally, with what it takes to develop a program consisting of several functions "from scratch". Unfortunately, it is practically impossible for you to see what *really* happens during this process unless you are "lucky" enough to be able to look over someone's shoulder during the entire development of a program and that person not only tells you what he or she is doing at every step, but also "thinks out loud" in your presence along the way.

An iterative process (and often a "messy" one)

In fact the process is not a very "clean" one, and it can sometimes, in fact, be quite "messy". Nor is it nicely "linear", proceeding from problem statement to a working program which provides the solution in some well-defined sequence of steps from beginning to end. Rather, it is an "iterative" process, in which one often has to return to some prior point and come forward again, after discovering that one has gone down a blind alley or made some minor or major incorrect decision at an earlier stage.

Programming methodologies

There is not even general agreement on what is the "best" *programming methodology* to use, and in fact it is probably fruitless to search for such a thing. Different approaches work better in different situations for different problems, and that is about all one can safely say.

Structured programming (procedural programming)

But don't be discouraged. The particular approach we shall examine in this Module is called *structured programming* (or *procedural programming*), in which emphasis is placed on identification of the various "procedures" or "tasks" to be performed, with the data to which the tasks must be applied having a somewhat secondary role, and the data and tasks are viewed as, treated as, and appear in the code as, separate entities. This approach actually works quite well, when properly applied, for programs of small to moderate size, and all of our programs will fall within this range.

Object-oriented methodology

The *structured programming methodology* is quite distinct from the *object-oriented methodology*, in which the data or *objects* are front and center and procedures or tasks are viewed as simply "behavioral" aspects of the various objects. We will come back to these object-oriented ideas later.[1]

Steps in the procedural (structured) approach to program development

The procedural approach to program development may be viewed as consisting of the following major steps or phases:

a. Problem Analysis

   If you don't know what *you are trying to do, there's no point worrying about* how *you're going to do it.*

   In the problem analysis phase of program development you must make absolutely sure that you thoroughly understand the problem to be solved. This may require reformulating all or part of the problem. It is your job to remove all ambiguities and make everything explicit, which may require checking with the ultimate end-user for clarification, or, failing that, making some necessary assumptions about how things will be handled. At this stage you should also attempt to identify all entities (things) involved in the problem and, at least in a general way, what needs to be done with them.

[1] But not till Part II of this Lab Manual.

b. Problem Specification (includes design of test data)

Specify what both the input and *the output are going to look like.*

Since it is a given that our ultimate goal is a *program* to solve the problem, we can perform the problem specification phase of the development by specifying exactly what the input and output to that program must be in order that the problem be solved. This might involve specifying things like the following: input source (keyboard or file, for example); format of both the input and output data; and how the program is to respond to "bad" data. This of course requires knowing what form the input data is going to be in, and you need to design appropriate test data sets to use for testing both your algorithm design in the next phase and the program when it is finished. Such data should include not only some easy and typical values, but also some extreme values, special values (if any), and perhaps even illegal values, depending on how *robust* your program is supposed to be.

c. Algorithm Development (includes design *and* testing)

This is where you apply top-down design with step-wise refinement and write some pseudocode.

The algorithm development phase is where you will apply top-down design with step-wise refinement and write pseudocode to describe the various tasks that need to be performed to solve the problem. When you apply the procedural approach to program development you begin by breaking the overall task down into a sequence of sub-tasks, each of which can be more easily managed than the original "big" task. You need to determine exactly what each sub-task is, as well as what entities are involved, what actions need to be performed on those entities, and the order in which those actions need to be performed in order to complete that task. Some actions might be very simple (those that could be implemented as one or two C++ statements when we get to the coding phase) and some will of course be much more complicated, requiring a separate function. In such cases, don't forget that the first line of attack is to ask: Is there a function anywhere in one of the C++ libraries that will do what needs to be done?

Once you have the algorithm that you believe solves your problem, you must test it using all (or as much as is reasonable) of the test data that you developed in the previous phase. Note that this is pencil-and-paper testing, *not* the testing of a computer program.

Note

None of the above three phases involved using the computer.

Now is a good time to point out the *first rule of programming*:

THE SOONER YOU START CODING, THE LONGER IT'S GOING TO TAKE.

d. Coding (and testing)

Now translate your pseudocode into actual C++ code.

Now comes the coding phase, where one's vast knowledge of C++, as well as the by-now-familiar edit-compile-link-run-test-debug phase of program development, come into play. It is here that one must "translate" pseudocode from the previous phase into actual C++ code. Note that up to this stage not only has the computer not been used, one may not even have decided what programming language is going to be used to write the program. Of coure, *we* know that it's going to be C++, but the point is that in general a programmer can postpone even *that* decision until this point in the development process.

The testing involved in this phase is "just" to make sure the program compiles, links and runs correctly in a few typical cases, in preparation for the more thorough testing that comes in the next phase.

e. Program Testing

The testing we are talking about here is more thorough, involves the test data sets that we prepared earlier in the problem specification phase, and must result in a program that we are confident will work correctly in all possible situations. Note that we can *never* be *absolutely sure*[2] of this, but we should not rest until our confidence is at a very high level.

This testing must exercise the program thoroughly using the test data sets developed earlier in the specification and algorithm development phases.

f. Documentation

Although we list a documentation phase as a specific phase (and the last phase) of our program development stages, documentation must actually be a continual activity throughout the process. You are simply asking for trouble to wait till the end to do all of your documentation. A much better plan is to make sure that each part of the process is completely documented as you complete it, and use this final documentation phase in the above list simply as a final opportunity to make sure that all the i's are dotted and the t's crossed.

Documentation must be an ongoing affair, not a one-night stand.

As for the specifics of the documentation, as always there are two things to keep in mind:

Two aspects of program documentation

- Make the source code *readable*.
- Provide a good *user interface* when the program runs. That is, make the program "user friendly".

You are by now familiar with many of the things that make source code readable:

Some style reminders for producing readable source code

- The use of a consistent overall style
- Meaningful identifier names properly and consistently capitalized
- Good spacing, indentation, and alignment
- Informative but not excessive commenting
- The use of programmer-defined constants where appropriate

[2] Which brings to mind a remark world-renowned computer scientist Donald Knuth made about one of his programs, that went something like this: "I have merely *proved* this program to be correct; I have not actually *tested* it."

Some style reminders for producing a good user interface

As for the user interface, make sure that your program

- Describes itself when it runs (and also identifies you as the programmer, if required, as in a programming project submitted for evaluation, for example)
- Provides good user prompts for keyboard input
- Echoes all input data somewhere in the output
- Formats all output in a way that is "pleasing to the eye"

The program development process

The first three of the six steps listed above are sometimes called the *problem-solving phases* and the last three the *solution-implementation phases* of the *program development process*. With any modern sophisticated piece of software there will be several follow-up and maintenance phases as well, involving such things as the delivery of the software to its end-users, the training of those end-users in its use, and the fixing of bugs and making requested changes when requested or required, as time goes on and the users gain experience with the program. We will not discuss these aspects of software development any further, not because they are not important but because they are better dealt with in a more advanced course on *software engineering*.

We do, however, want to discuss a couple of new concepts and some terminology which has a natural home in the current context, but the main goal here is just to get a good sense of how the program development process goes.

Stubs and drivers

One important new concept is that of *stubs* and *drivers*, and how they fit into the program development process. We introduce these in the context of the sample program `stub_driver.cpp` in the next section.

We also want to extend slightly the notion of a design tree diagram, now that we have functions and parameters at our disposal, so let's do that next. In fact, we should take this opportunity to remind you that the ideas embodied in the notions of top-down design with step-wise refinement, pseudocode, and design tree diagrams become even more important and useful as our programs increase in complexity and involve many more functions.

The idea that we wish to introduce here is that of adding to *any* design tree diagram for a program a pictorial indication of the information flow that takes place in that program. The idea is quite simple, and a short example should convey the necessary essentials.

Suppose the (incomplete) body of the main function of a simple program is this:

```
{
       ...
    DescribeProgram();
    GetData(data);
    UseData(data);
       ...
}
```

Showing function names in a design tree diagram

We would draw the corresponding design tree diagram as shown in Figure 17.1, in which the only difference from previous design tree diagrams is that sub-tasks are now indicated by the names of the functions that perform them:

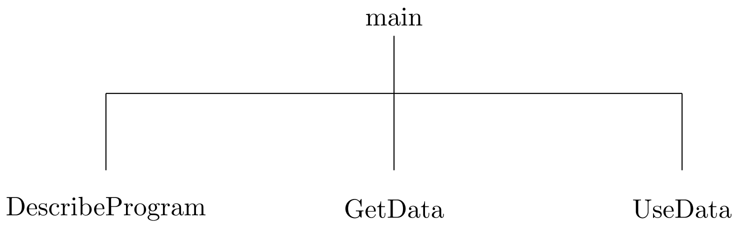

Figure 17.1: Design Tree Diagram for Program with Functions

Showing information flow in a design tree diagram

However, if we know that `data` is an out-parameter of `GetData` and an in-parameter of `UseData` we can indicate that information as in the revised design tree diagram of Figure 17.2.

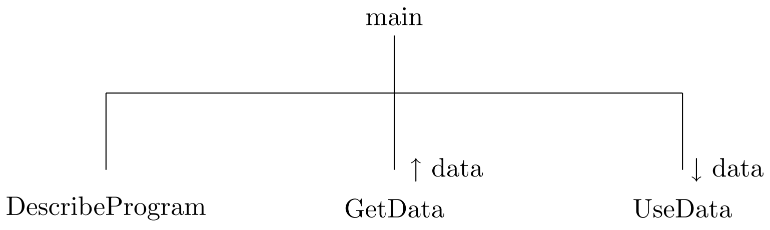

Figure 17.2: Design Tree Diagram Showing Direction of Information Flow

In such a diagram we would place the name of each out-parameter above the name of its corresponding function with an associated up-arrow and each in-parameter above its corresponding function with an associated down-arrow. Inout-parameters (of which none are shown here) would have two associated arrows, one up and one down (or a single double-headed arrow). Such an enhanced design tree diagram conveys more information to the viewer than the corresponding diagram without the information-flow data, and may therefore be quite helpful in providing an overview of how a program behaves.

17.4 Sample Programs

17.4.1 stub_driver.cpp contains a generic shell program with stubs and a driver

```
//stub_driver.cpp
//A new kind of generic shell program containing "stubs" and
//a main "driver" which can be used as a starting point for
//many programs containing several functions.

#include <iostream>
using namespace std;


void DescribeProgram();
void GetInputData();
void ComputeSomeValues();
void DisplayAllValues();


int main()
{
    DescribeProgram();
    GetInputData();
    ComputeSomeValues();
    DisplayAllValues();
    cout << endl;
}


void DescribeProgram()
//Pre/Post conditions
{
    cout << "\nThis program is wonderful ...\n";
}


void GetInputData()
//Pre/Post conditions
{
    cout << "\nNow inside GetInputData ...\n";
}


void ComputeSomeValues()
//Pre/Post conditions
{
    cout << "\nNow inside ComputeSomeValues ...\n";
}


void DisplayAllValues()
//Pre/Post conditions
{
    cout << "\nNow inside DisplayAllValues ...\n";
}
```

17.4.1.1 What you see for the first time in stub_driver.cpp

- A program containing several *stubs* (also called *stub functions*)
- A *driver* (the main function in this case) that *calls* each of the stubs

17.4.1.2 Additional notes and discussion on stub_driver.cpp

Another kind of shell program

This program illustrates what is essentially another form of *shell program*, which differs substantially from the one we saw earlier in the `shell.cpp` program of Module 6, though the two could easily be combined.

The `main` function in this context is called a *driver* for the other functions which are called *stub functions*, or *stubs*, because at the moment they are incomplete, which is what the word "stub" suggests.

Complete the program one stub at a time.

Even though each of the stubs is incomplete, the program nevertheless compiles, links and runs. And even though it doesn't "do" anything, it gives us a working "shell" that we can complete, *one stub at a time.* Of course, in a real situation, such as the "farmer problem" we wish to solve in the hands-on activities of this Module, the names of the stubs might be better chosen to reflect the nature of the problem but the principles remain the same.

Implement top-down as well as design top-down.

The idea is to complete the program by completing each of the stubs in turn (independently if possible, but occasionally it may be more convenient to work on more than one stub at a time, in parallel). This approach to the actual coding parallels our top-down design approach since we will, of course, be using our pseudocode from that design to complete the stubs. It's important to note, though, that the "completion" of a stub might well involve the introduction of one or more other stubs at a "lower level" which will not themselves be completed till later in the development. This in turn is, of course, just step-wise refinement in action.

Once we know what the names of the functions at any particular level are going to be, the next step is to decide on the parameter list for each function, which, together with the name, is sometimes (as you know) referred to as the *interface* to the function. Doing this requires you to decide what information, if any, is to flow into or out of the function, and to choose suitable names for any data values that must flow in or out. When each interface has been completed, it can be tested by supplying "dummy values" to the parameters (via assignment, for example) before the code that produces the actual values has been written. At any point in this process you should be able to compile, link and test a "working" program.

Next, the body of each function must be completed. This involves filling in the code that permits each function to actually do its particular task, according to the specifications for that function, and will be guided by the pseudocode for that function produced during the design phase. Each individual function must be tested independently as thoroughly as possible when it is finished, and if you are implementing the design in top-down fashion using stubs and drivers, starting with your main function as the top-level driver, you will *always* have a built-in driver for each completed function.

Integration testing is the "acid test".

Finally, when all functions have been completed, you must perform what is called *integration testing*, i.e., putting the program through its paces as a whole and making sure that it solves the original problem (i.e., that it *complies with its original specifications*).

17.4.1.3 Follow-up hands-on activities for stub_driver.cpp

□ Activity 1 Copy, study and test the program in `stub_driver.cpp`.

□ Activity 2 Use the ideas discussed in this Module and the kind of shell program with stubs and drivers shown in `stub_driver.cpp` to develop, through all of the various phases, a complete solution to the following "farmer's problem":

Design and write a program that a farmer can use to determine the cost of both fencing and fertilizing a rectangular field.

Though all of the necessary ideas have been presented in this Module, and the concepts are reasonably straightforward, this will nevertheless be a tall order for most beginning programmers. What seems simple in theory turns out not to be so simple in actual practice, and you should not feel intimidated if it seems like a daunting task. In all likelihood, you will need some guidance in class and/or lab as you make your way through the details of this program development. Once you have the basic ideas involved well in hand, however, you are on your way and the sky's the limit.

○ Instructor checkpoint 17.1 for evaluating prior work

Module 18

Consolidating I/O, files, selection, looping, functions and program development

18.1 Objectives

- To get some experience in putting together a number of different lower-level constructs to create useful higher-level constructs to perform some particular tasks.
- To appreciate the importance of having a program terminate "gracefully", and to begin looking at some of the many ways this *graceful termination* can be implemented.
- To understand the use of a call to the `exit()` function from the `cstdlib` library to terminate a program.
- To understand how to make use of files as function parameters.
- To learn how to declare a formal function parameter when the actual parameter can be either `cin` or an input file stream.
- To learn how to declare a formal function parameter when the actual parameter can be either `cout` or an output file stream.

18.2 List of associated files

- `reverse_digits.cpp` displays positive integers entered from the keyboard with their digits in reverse order.
- `display_file_data.cpp` displays on the screen the contents of a textfile of specific data.

- `display_file_data.in` is a sample input file for `display_file_data.cpp`.
- `read_write.cpp` copies data from file to file, and keyboard to screen, using the same functions for both transfers.
- `report_wages.cpp` contains a wage-reporting shell program with stubs and a main driver.
- `draw_boxes.cpp` draws empty boxes using only punctuation characters for the border character.

18.3 Overview

Now's the time to work with some more complex programs and consolidate your knowledge.

In earlier Modules we began to see how much more power and versatility programs could achieve when they were given the ability to make decisions, and the ability to repeat one or more actions. Now that we have programmer-defined functions for encapsulating task performance available as well, we have at our disposal not only a great deal of potential power, but the means of keeping the complexity of our programs under control as we invoke that additional power.

Although this Module contains a few new C++ features, its main purpose is to provide some (small) examples of programs that involve all of the major concepts we have introduced in the last few Modules: decision-making, looping and programmer-defined functions, all working together to accomplish some higher goal, and put together using the principles of structured program development discussed in Module 17.

It is really only now, as our programs continue to grow ever more complex, and we make them perform ever more complicated tasks, that the real need for the method of top-down design with step-wise refinement is felt, and the real usefulness of this tool in program development can be appreciated. The programs that appear here, though still tiny, nevertheless required an organized approach for their development, and you need to apply that same approach to *all* of your programs of this size or larger, in order to become comfortable with top-down design.

However, you must also remember this: The top-down approach is only one part of any *design methodology* that you might choose to use. And even supposing you have the best design in the world, it is still a non-trivial task to actually implement[1] that design and produce a working program that performs as required. To help you carry out this phase of program development—getting from your wonderful on-paper design to a working program—you will need to make use of everything at your disposal and follow as well as you can the steps given in Module 17 for program development via the procedural approach.

[1] As in "to boldly go ...", and, hoping and trusting that Winston would also approve the split infinitive, grammar purists once again be damned. (See page 7 for Winston's opinion on another matter.)

18.4 Sample Programs

18.4.1 reverse_digits.cpp displays positive integers input from the keyboard with their digits in reverse order

```
//reverse_digits.cpp
//Displays each positive integer input with the digits reversed.


#include <iostream>
using namespace std;


void DescribeProgram();
void GetPositiveIntegerFromUser(int& i);
int ReversedDigitsOf(int i);


int main()
{
    DescribeProgram();

    int i;
    GetPositiveIntegerFromUser(i);
    while (i != 0)
    {
        if (i > 0)
            cout << "The reverse of the integer " << i
                 << " is " << ReversedDigitsOf(i) << ".\n\n";
        else
            cout << "Not a positive integer. Try again.\n";
        GetPositiveIntegerFromUser(i);
    }
    cout << endl;
}



void DescribeProgram()
//Pre:  The cursor is at the left margin.
//Post: The program description has been displayed,
//      preceded and followed by at least one blank line.
{
    cout << "\nThis program gets a positive integer from the user, then "
        "computes and outputs\na new integer which is the old one with "
        "its digits reversed.\n\n";
}



void GetPositiveIntegerFromUser(/* out */ int& i)
//Pre:  none
//Post: "i" contains a positive integer entered by the user.
//      The input stream "cin" is empty.
{
    cout << "Enter a positive integer here, or 0 to quit: ";
    cin >> i;  cin.ignore(80, '\n');  cout << endl;
}
```

```
int ReversedDigitsOf(/* in */ int i)
//Pre:  "i" has been initialized and contains a positive integer.
//Post: Function value returned is the integer containing the
//      digits of "i" in reverse order.
{
    int reverse = 0; //Contains no digits from "i"
    while (i != 0)
    {
        reverse = 10*reverse + i%10; //Add last digit of "i" to "reverse"
        i = i/10;                    //Remove last digit from "i"
    }
    return reverse;  //Contains value of "i" with digits reversed
}
```

18.4.1.1 What you see for the first time in reverse_digits.cpp

- Both void functions and value-returning functions that have been defined by the programmer and are working together with decision-making constructs and looping constructs, all in the same program
- Some mildly unorthodox naming and formatting:
  - First, the name of the function that finds the reversed digits is somewhat unorthodox and you might want to comment on its readability (the name was chosen in the hope that it *would* be readable).
  - Second, also in the same function, we have removed the spaces around some (but not all) of the arithmetic operators (again, somewhat of a departure from normal practice, but done to enhance readability by showing more clearly the operands associated with a particular operator).

18.4.1.2 Additional notes and discussion on reverse_digits.cpp

Reverse-engineering a solution program

Note that even though this is a very short program, it nevertheless involves decision-making, looping, and both void and value-returning functions. Clearly the presence of all these features makes the program somewhat more complex in some respects than any we have seen thus far, and there is probably no better exercise that you could do right now than to run the program a few times to see how it behaves, and then try to *reverse-engineer* the program, i.e., try to produce a design that, if implemented, would give you a program that behaves just like the one in `reverse_digits.cpp`. Compare your design with the program in `reverse_digits.cpp`, note any differences, and decide whether the two are in fact equivalent (and, if so, which you like better).

18.4.1.3 Follow-up hands-on activities for reverse_digits.cpp

□ Activity 1 Copy, study and test the program in `reverse_digits.cpp`.

□ Activity 2 Perform the design exercise suggested in the paragraph above.

□ Activity 3 Copy `reverse_digits.cpp` to `reverse_digits1.cpp` and then bug it as follows:

a. Replace `(i != 0)` with `(i > 0)` in the `main` function.

____

b. Replace `(i != 0)` with `(i > 0)` in the `ReversedDigitsOf` function.

____

□ Activity 4 Make a copy of `reverse_digits.cpp` called `reverse_digits2.cpp` and modify the copy so that it essentially works as before, except that it also displays negative integers with their digits reversed instead of outputting the error message that it currently displays when the user enters a negative integer. For example, if the user enters –1234, the revised program should display –4321.

○ Instructor checkpoint 18.1 for evaluating prior work

□ Activity 5 Make a copy of your completed `reverse_digits2.cpp` from the previous activity and call it `reverse_odd_even.cpp`. Modify the copy so that if the user enters a positive integer, the program displays another integer consisting of all odd digits in the input integer in reverse order. For example, if the user enters 1234, the revised program should display 31. Also, if the user enters a negative integer, the program displays another (negative) integer consisting of all even digits in the input integer in reverse order. For example, if the user enters –1234, the revised program should display –42. In either case, if there are no digits of the required type, nothing is displayed.

○ Instructor checkpoint 18.2 for evaluating prior work

18.4.2 display_file_data.cpp displays on the screen the contents of a textfile of specific data

```
//display_file_data.cpp
//Displays the data from a file, provided
//the data in the file has a certain format.

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

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

int main()
{
    DescribeProgram();

    ifstream inFile;
    inFile.open("in_data");
    cout << "Here is the data from the file: " << endl;
    ReadAndDisplayFileData(inFile);
    inFile.close();
}

void DescribeProgram()
//Pre:  The cursor is at the left margin.
//Post: The program description has been displayed,
//      preceded and followed by at least one blank line.
{
    cout << "\nThis program displays all the data from a file called "
        "\"in_data\".\nIf no output (or bizarre output) appears below, "
        "the file is not yet\navailable on your system. Study the "
        "source code, create a file with\nthe appropriate format, and "
        "then try again.\n\n";
}

void ReadAndDisplayFileData(/* in */ ifstream& inFile)
//Pre:  The file denoted by "inFile" exists, contains data in
//      the proper format, and has been opened.
//Post: The data in "inFile" has been displayed.
{
    char firstInitial, lastInitial;
    int mark1, mark2, mark3;

    inFile >> firstInitial >> lastInitial;
    inFile.ignore(80, ’\n’);
    inFile >> mark1 >> mark2 >> mark3;
    inFile.ignore(80, ’\n’);

    cout << "\nStudent’s Initials:   "
         << firstInitial << lastInitial
         << "\nStudent’s Test Marks: "
         << mark1 << "  " << mark2 << "  " << mark3;
    cout << endl << endl;
}
```

18.4.2.1 What you see for the first time in display_file_data.cpp

- A parameter which has an `ifstream` data type (i.e., a parameter which is the name of a file) and which is a conceptual in-parameter even though it is a reference parameter

File parameters must always be reference parameters.

- The use of `inFile.ignore(80, '\n')` for file input in a manner analogous to the way in which `cin.ignore(80, '\n')` was used for keyboard input

18.4.2.2 Additional notes and discussion on display_file_data.cpp

The most important thing to note about this program is that the file variable `inFile` is implemented as a *reference parameter*, even though conceptually it is an in-parameter (it is passing the data in the file *to* the function). This runs counter to our usual guidelines, which would have it that an in-parameter should be implemented as a value parameter.

However, there is a very good reason for requiring a file parameter to be a reference parameter. A function always makes its own copy of the data in a value parameter, and it would be wasteful at best, and impossible at worst, for a function to copy all the data in a file each time a file was passed to it, since the file might be very large indeed. Thus the rule to remember is this: A parameter which refers to the name of a file, used for either input or output (i.e., which has a data type of `ifstream` or `ofstream`) *must* be passed as a reference parameter. It is a compile-time error if you fail to do this.

Note that we again use `inFile.close()` to "close" the file when we are finished reading data from it, as we did in earlier programs that dealt with files. In fact it is technically unnecessary here as it was earlier, since the operating system will ensure that the file is "closed" when the program finishes executing.

However, as we have pointed out before, it is always good programming practice to close any file you have "opened" and used when you are finished with it, rather than leaving such "housekeeping chores" to the operating system. And, it is absolutely necessary to do so if, later in the same program, you wish to use the same file variable `inFile`, say, to refer to a different physical file. We do not do this here, but you will want to do it in other situations.

18.4.2.3 Follow-up hands-on activities for display_file_data.cpp

□ Activity 1 Copy, study, test and then write pseudocode and draw a design tree diagram for `display_file_data.cpp`.

You will want to create input data files of your own, but you can begin with the sample data file shown below.

Contents of the file `display_file_data.in` are shown between the heavy lines:

```
JS         <-- These are the student's initials.
67 59 82   <-- These are the student's three test marks.
```

Keep in mind that the program thinks of its input file as `in_data` when "talking to" the operating system.

□ Activity 2 Copy `display_file_data.cpp` to `display_file_data1.cpp` and bug it as follows:

a. Remove the `&` in the function prototype.

____

b. Remove the `&` in the function definition.

____

c. Remove the `&` in both the function prototype and the function definition.

____

□ Activity 3 Make a copy of `display_file_data.cpp` called `display_file_data2.cpp` and modify the file so that in addition to continuing to do what it already does, it also writes to an output file called `display_file_data2.out` the initials of the student and the average mark that the student received on the three tests whose marks are in the input data file. The average mark must be rounded to one place after the decimal. The output format in the output file must be like this:

```
Student: JS
Average: 69.3
```

For this program also do the following: Make `ReadAndDisplayFileData` into *two separate* functions: `ReadFileData` and `DisplayFileData`.

○ Instructor checkpoint 18.3 for evaluating prior work

18.4.3 read_write.cpp copies data from file to file, and keyboard to screen, using the same functions for both transfers

```
//read_write.cpp
//Reads data from a file, and writes that data to another file.
//Then reads data from the keyboard and writes it to the screen.
//So what? So it uses the *same* read and write functions in *each* case.
//Among other things, this means the input data must have the same format
//in both situations: two lines of data, with two characters on the first
//line and three integers on the second.


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


void DescribeProgram();
void ReadData(istream& inFile,
              char& firstInitial, char& lastInitial,
              int& mark1, int& mark2, int& mark3);
void DisplayData(ostream& outFile,
                 char firstInitial, char lastInitial,
                 int mark1, int mark2, int mark3);

int main()
{
    DescribeProgram();

    char firstI, secondI;
    int  m1, m2, m3;


    ifstream inF;
    inF.open("in_data");
    ReadData(inF, firstI, secondI, m1, m2, m3);
    inF.close();

    ofstream outF;
    outF.open("out_data");
    DisplayData(outF, firstI, secondI, m1, m2, m3);
    outF.close();


    cout << "With luck, all data has now been transferred from "
        "\"in_data\" to \"out_data\".\nTo verify this, check the "
        "contents of \"out_data\" when the program finishes.\n";

    cout << "\nNow enter data from the keyboard:\n";
    ReadData(cin, firstI, secondI, m1, m2, m3);
    cout << "\n\nHere is the data read in from the keyboard:\n";
    DisplayData(cout, firstI, secondI, m1, m2, m3);
    cout << endl;
}
```

```
void DescribeProgram()
//Pre:  The cursor is at the left margin.
//Post: The program description has been displayed,
//      preceded and followed by at least one blank line.
{
    cout << "\nThis program reads data from a file called \"in_data\", "
        "and then writes it out\nto a file called \"out_data\". Next it "
        "reads the same kind of data from the\nkeyboard and writes it "
        "out to the screen. In both cases the input data must\nconsist "
        "of two lines with two characters on the first line and three "
        "integers\non the second. And be sure that the input data file "
        "exists!\n\n";
}


void ReadData(/* in */  istream& inFile,
              /* out */ char& firstInitial,
              /* out */ char& lastInitial,
              /* out */ int& mark1,
              /* out */ int& mark2,
              /* out */ int& mark3)
//Pre:  The file denoted by "inFile" exists, contains data in
//      the proper format, and has been opened.
//Post: Data from "inFile" has been read into two char out-parameters
//      and three int out-parameters.  "inFile" is still open.
{
    inFile >> firstInitial >> lastInitial;
    inFile.ignore(80, ’\n’);
    inFile >> mark1 >> mark2 >> mark3;
    inFile.ignore(80, ’\n’);
}


void DisplayData(/* out */ ostream& outFile,
                 /* in */  char firstInitial,
                 /* in */  char lastInitial,
                 /* in */  int mark1,
                 /* in */  int mark2,
                 /* in */  int mark3)
//Pre:  The file denoted by "outFile" exists, and has been opened.
//      All other parameters have been initialized.
//Post: The values in the two char in-parameters and the three
//      int in-parameters have been displayed on "outFile".
//      "outFile" is still open.
{
    outFile << "\nStudent’s Initials:   "
            << firstInitial << lastInitial
            << "\nStudent’s Test Marks: "
            << mark1 << "  " << mark2 << "  " << mark3;
    outFile << endl << endl;
}
```

18.4.3.1 What you see for the first time in read_write.cpp

- A formal parameter of data type `istream` (in `ReadData`)

  This formal parameter appears in the prototype of `ReadData` as well as in the header of the definition of `ReadData`. But note the two calls to `ReadData` in `main`. In the first one the *actual parameter* is the *input file stream* `inF`, while in the second it is the *standard input* `cin` (i.e., the *keyboard*).

An unusual situation

- A formal parameter of data type `ostream` (in `DisplayData`)

  Similarly, this formal parameter appears in the prototype of `DisplayData` as well as in the header of the definition of `DisplayData`. And note the two calls to `DisplayData` in `main`. In the first one the *actual parameter* is the *output file stream* `outF`, while in the second it is the *standard output* `cout` (i.e., the *screen*).

18.4.3.2 Additional notes and discussion on read_write.cpp

What this program illustrates is an example of the object-oriented notion of *inheritance*, which we do not need to discuss in detail at this point. The concept of inheritance shows up here in the following way: Both `cin` and `inF` are in fact *input streams*, though not the same kind of input stream. But since in the formal parameter list of `ReadData` we said we were going to pass an "input stream" of some kind when we called the function, it turns out that we can pass *either* `cin` or `inF` since *both* are input streams. In the language of inheritance, we might say that both `cin` and `inF` have "inherited" the properties that are common to all input streams and that makes either of them eligible to be passed to `ReadData` as an actual parameter.

A first encounter with inheritance, which helps us deal with the unusual situation

Analogous comments also apply to the fact that `cout` and `outF`, though different kinds of output streams, both inherit the general properties of all output streams and either is therefore eligible to be passed to `DisplayData`, which has been told it will receive an output stream as its first actual parameter when it is called.

18.4.3.3 Follow-up hands-on activities for read_write.cpp

□ Activity 1 Copy, study, test and then write pseudocode and draw a design tree diagram for `read_write.cpp`. For testing you will again have to create your own input file(s), but you can use any of the input data files you used for testing the previous sample program `display_file_data.cpp`.

□ Activity 2 Copy `read_write.cpp` to `read_write1.cpp` and bug it as follows:

a. Change the type of the parameter `inFile` from `istream` to `ifstream` in both the prototype and the definition of `ReadData`.

____

b. Change the type of the parameter `outFile` from `ostream` to `ofstream` in both the prototype and the definition of `DisplayData`.

____

c. Remove the line which closes `inF`.

____

d. Remove the line which closes `outF`.

____

Based on shell.cpp

□ Activity 3 The program in `read_write.cpp` first transfers data from one file to another, then from the keyboard to screen. Make a copy of `read_write.cpp` called `read_write2.cpp` and modify the copy so that it permits the user to transfer data from a file either to another file *or* to the screen, and also from the keyboard either to the screen *or* to a file. The program must allow the user to do this as many times as desired before quitting.

○ Instructor checkpoint 18.4 for evaluating prior work

18.4.4 report_wages.cpp contains a shell for a wage reporting program, with stubs and a main driver

```
//report_wages.cpp
//A shell for a wage-reporting program.
//Contains a main "driver" and four "stubs" (or "stub functions")

#include <iostream>
using namespace std;

void DescribeProgram();
void GetWageData();
void ComputeWageInfo();
void DisplayWageInfo();


int main()
{
    DescribeProgram();
    GetWageData();
    ComputeWageInfo();
    DisplayWageInfo();
    cout << endl;
}


void DescribeProgram()
//Pre/Post conditions
{
    cout << "\nThis program is wonderful ...\n";
}


void GetWageData()
//Pre/Post conditions
{
    cout << "\nNow inside GetWageData ...\n";
}


void ComputeWageInfo()
//Pre/Post conditions
{
    cout << "\nNow inside ComputeWageInfo ...\n";
}


void DisplayWageInfo()
//Pre/Post conditions
{
    cout << "\nNow inside DisplayWageInfo ...\n";
}
```

18.4.4.1 What you see for the first time in report_wages.cpp

A chance to put your program development skills to work

This program provides the starting point for another exercise in program development, complete with stubs and a driver for the top level. This one is based on an earlier sample program, `wages.cpp`, from Module 12.

18.4.4.2 Additional notes and discussion on report_wages.cpp

If you look back at the program in `wages.cpp` from Module 12 you should recognize that if we had had functions at our disposal at that time, then the program we might have written for `wages.cpp` could have looked (in outline, at least) quite a bit like the one we have here.

In fact, suppose we had completed our design for the earlier wages program, and in the meantime we had learned about functions. Then we could certainly take the program in `report_wages.cpp` as a starting point for the actual implementation of that program. The main thing to observe is that this program shell contains a `main` function, which already calls all of the necessary other functions to perform the required tasks. Of course, none of those other functions actually performs its task at the moment, but the overall structure of the complete program is in place.

The `main` function in this context is called a *driver* for the other functions which are called *stub functions*, or *stubs*, because at the moment they are incomplete. If we now complete the program by completing each of the stubs in turn (independently if possible, but occasionally it may be more convenient to work on more than one stub at a time, in parallel), this approach to the actual coding parallels our top-down design approach since we will, of course, be using our pseudocode from that design to complete the stubs.

The next step is to decide on the parameter list for each function, which is sometimes (as you know) referred to as the *interface* to the function. Doing this requires you to decide what information, if any, is to flow into or out of the function, and to choose suitable names for any data values that must flow in or out. When each interface has been completed, it can be tested by supplying "dummy values" to the parameters (via assignment, for example) before the code that produces the actual values has been written.

Now, the body of each function must be completed. This involves filling in the code that permits each function to actually do its particular task, according to the specifications for that function, and guided by the pseudocode for that function produced during the design phase. Each individual function must be tested independently as thoroughly as possible when it is finished, and if you are implementing the design in top-down fashion using stubs and drivers, starting with your main function as the top-level driver, you will always have a built-in driver for each completed function.

Finally, when all functions have been completed, you must perform *integration testing*, i.e., putting the program through its paces as a whole and making sure that it solves the original problem (i.e., that it complies with its original specifications).

18.4.4.3 Follow-up hands-on activities for report_wages.cpp

□ Activity 1 Copy, study, test and then write pseudocode and draw a design tree diagram for `report_wages.cpp`.

□ Activity 2 Make a copy of `report_wages.cpp` called `report_wages1.cpp` and complete the program so that it behaves as outlined below. This is actually an enhancement of the earlier `wages.cpp` from Module 12, so you should make another design before proceeding, though the stubs and the driver that you have in `report_wages1.cpp` will serve as the starting point when you begin the actual coding. The completed program is required to handle only one computation per run, and must exhibit the following behavior:

a. It must begin by displaying a description of itself.

b. It must then prompt for and read in both a number of hours worked and a wage rate.

c. Next, it must ompute both gross wages and net wages (see below).

d. Finallly, it must display gross wages, net wages, and deductions, and also show somewhere in the output display all input data read in.

Gross wages are to be calculated on the basis of a regular week of 35 hours, with time-and-a-half for overtime. There are two deductions: a 20% deduction of the gross wages for income tax, and a flat rate of $3.00 per pay for union dues, and "net wages" are computed by making these two deductions from gross wages. One of the many things to think about in this problem is the following question: How should the taxes and union dues be handled? This is a typical question that arises all the time in this kind of context, and it is worth thinking about here.

□ Activity 3 Make a copy of the program you completed in `report_wages1.cpp` from the previous activity and call it `shell_report_wages.cpp`. Modify it so that in a single run it can deal with any number of wage computations, each one like the one computation handled in `report_wages1.cpp`.

This program can be based on shell.cpp.

○ Instructor checkpoint 18.5 for evaluating prior work

18.4.5 draw_boxes.cpp draws empty boxes using only punctuation characters

```
//draw_boxes.cpp
//Prompts the user to enter a punctuation character and
//then uses that character to draw a 4 by 4 "empty" box.
//The box is preceded and followed by a blank line, and centered
//between the left/right margins of a typical 80-column display.
//If the user does not enter a punctuation character, the program
//displays an error message, does not attempt to draw a box,
//and asks the user to try again. The program terminates when
//the user enters the end-of-file character in response to the
//request to enter a character for the box border.


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

void DescribeProgram();
void GetCharFromUser(char& ch, bool& charOK, bool& timeToQuit);
void DrawBox(char borderChar);


int main()
{
    DescribeProgram();

    char borderChar;
    bool borderCharOK;
    bool timeToQuit;

    GetCharFromUser(borderChar, borderCharOK, timeToQuit);
    while (!timeToQuit)
    {
        if (borderCharOK)
            DrawBox(borderChar);
        else
            cout << "Error: Character input was "
                "not punctuation. Try again ...\n";
        GetCharFromUser(borderChar, borderCharOK, timeToQuit);
    }
    cout << endl;
}

void DescribeProgram()
//Pre:  The cursor is at the left margin.
//Post: The program description has been displayed,
//      preceded and followed by at least one blank line.
{
    cout << "\nThis program draws a 4-character by 4-character \"empty\" "
        "box. It uses a\ncharacter input by the user, and centers the "
        "box between the left/right\nmargins on a typical 80-column "
        "display.\n\n";
}

```

```

void GetCharFromUser(/* out */ char& ch,
                     /* out */ bool& chOK,
                     /* out */ bool& timeToQuit)
//Pre:  None
//Post: ch contains the character entered by the user
//      Value of chOK is
//          - true if ch contains a punctuation character
//          - false otherwise
//      Value of timeToQuit is
//          - true if user has entered the end-of-file character
//          - false otherwise
{
    cout << "\nEnter a punctuation character here "
        "(or end-of-file to quit): ";
    cin >> ch;
    timeToQuit = !cin;
    if (!timeToQuit)
    {
        cout << endl;
        chOK = (ispunct(ch) != 0);
    }
}


void DrawBox(/* in */ char borderChar)
//Pre:  borderChar has been initialized.
//Post: A 4 x 4 "empty" box has been displayed,
//      using the character in borderChar.
{
    cout << endl;
    cout << setw(38) << ""
         << borderChar << borderChar << borderChar << borderChar << endl
         << setw(38) << ""
         << borderChar <<             "  "         << borderChar << endl
         << setw(38) << ""
         << borderChar <<             "  "         << borderChar << endl
         << setw(38) << ""
         << borderChar << borderChar << borderChar << borderChar << endl;
    cout << endl;
}
```

Take some time to study the `main` function of this program. Note that without looking at the the function definitions of the functions called by the `main` function you can get an excellent overview of this program and what it does (though *not* how it does it). That is, you can ignore the “low-level” details of the individual functions called by `main`, and still get the “big picture” quite nicely, just by “reading” the body of `main`.

This is an important principle to appreciate and apply: By physically structuring the code in a program properly, the program as a whole can be easily and quickly understood “up front”, and it is only necessary to pursue lower-level details in other parts of the source code if you need those details for some other reason.

18.4.5.1 What you see for the first time in draw_boxes.cpp

- The use of decision-making constructs and looping constructs, as well as several programmer-defined functions (all `void`, in this case), and everything working together in the same program
- A more "sophisticated" program termination mechanism than we have seen thus far, involving a *flag* (i.e., a boolean variable) whose value is returned to `main` from the function that tries to get input from the user (see the use of the boolean variable `timeToQuit` as a reference parameter in the function `GetCharFromUser`)
- Use of the function `ispunct()` to determine if a character is a punctuation character
- Use of `!cin` as an indicator of when it is "time to quit", in the sense that when this expression is true, the user has entered the end-of-file character and the input stream `cin` has shut down

18.4.5.2 Additional notes and discussion on draw_boxes.cpp

This program permits the user to draw as many 4 by 4 "empty" boxes as the user desires in a single run of the program, and also performs a check to see if the character entered by the user for the box border is one of the permitted characters (i.e., a punctuation character in this case). Also, the program permits *graceful termination* by allowing the user to enter the end-of-file character as a signal that no more boxes are to be drawn. You will be asked to enhance this program in various ways in the hands-on activities.

18.4.5.3 Follow-up hands-on activities for draw_boxes.cpp

□ Activity 1 Copy, study, test and then write pseudocode and draw a design tree diagram for `draw_boxes.cpp`.

□ Activity 2 Copy `draw_boxes.cpp` to `draw_boxes1.cpp` and bug it as follows:

a. Replace `!timeToQuit` with `timeToQuit` in the condition of the while-loop of the `main` function.

____

b. Comment out[2] the call to the function `GetCharFromUser`.

____

[2] The term *comment out*, when applied to a code segment, means to turn the lines of the code segment into comments, usually temporarily, while some sort of test is made or an alternate code segment is tried. Your editor may provide an easy way to do this.

□ Activity 3 Make a copy of `draw_boxes.cpp` called `draw_boxes2.cpp` and modify the copy so that instead of drawing an "empty" box, the program draws a "filled" box, using two (generally different but possibly the same) characters, one for the border and one for the interior. The "fill character" for the interior must also be entered by the user, as well as the border character. In addition, the user must be able to choose the size of the box (i.e., its width and height as measured by the number of characters used for each), as well as how many spaces the box will be indented from the left margin.

You may write the program without error checking on the width, height and indentation level. That is, you may assume that the user will always enter values for these quantities that "make sense" and do not run the box over the edge of the screen. (When you have the program finished in this form, however, you may wish to extend it by then adding the necessary robustness to reject user input that is unacceptable or will cause problems. But this is not necessary on the first pass.)

□ Activity 4 So far we have almost always had our main function return a 0 value to indicate its "success", though occasionally (in `test_input_stream.cpp` from Module 14, for example) we have seen situations when it clearly should return some other value to indicate a failure of some sort. But in all cases we have used the return-statement to return the value.

There is another way to terminate a program which is sometimes used, and which can also be used to return a value to the operating system just like the return-statement in a `main` function. This is the `exit()` function, which is available to us if we include the C++ `stdlib` library in our program. A call of `exit(0)` anywhere in our program will cause the program to end immediately and indicate a successful termination by returning the value 0 from `main`, while the same call with another (non-zero) value could be used to indicate some kind of failure.

Using the `exit()` *function from the* `cstdlib` *library*

So, another way to end our `draw_boxes.cpp` program would be to exit from the program in this way directly from the `GetCharFromUser` function, as soon as the user enters an end-of-file character.

Make a copy of `draw_boxes.cpp` called `draw_boxes3.cpp` and modify the program so that it terminates in this way. Indicate below which of the two methods of termination you think is "best" and why.

____

____

____

____

____

○ Instructor checkpoint 18.6 for evaluating prior work

□ Activity 5 Make a copy of your completed program in `draw_boxes3.cpp` from the previous activity and call it `draw_boxes4.cpp`. Modify the copy so that it performs exactly as before, except if the box is square and the size is an odd positive integer greater than or equal to 5. In such a case, instead of drawing a box with a completely filled interior it will draw a box that is empty except for an `X` figure drawn in the interior of the box using the "fill character".

For example, if the border character is `*`, the fill character is `+`, the size is 5, and the indentation level is 0, then the displayed box would look like this:

```
*****
*+ +*
* + *
*+ +*
*****
```

As a second example, if the border character is `+`, the fill character is `*`, the size is 9, and the indentation level is 5, then the displayed box would look like this:

```
     +++++++++
     +*     *+
     + *   * +
     +  * *  +
     +   *   +
     +  * *  +
     + *   * +
     +*     *+
     +++++++++
```

○ Instructor checkpoint 18.7 for evaluating prior work

Based on shell.cpp

□ Activity 6 Make a copy of `draw_boxes.cpp` and call it `shell_draw_boxes.cpp`. Then modify the copy so that it incorporates the "shell-like" behavior of providing a menu with a quit option, a "get information" option, and a "draw box" option. If the user chooses to draw boxes, then the program should allow as many boxes as desired to be be drawn before returning to the menu.

○ Instructor checkpoint 18.8 for evaluating prior work

Module 19

Miscellaneous programs illustrating additional features of C++

19.1 Objectives

- To learn what the value ranges are for the character, integer, and floating point (i.e., real number) data types in your C++ system, as defined by the *system-dependent constants* in the `climits` and `cfloat` libraries.
- To understand how the `sizeof` operator works, for simple data types.
- To understand why the ++ and `--` operators deserve extra care.
- To understand the *dangling else problem* and know how to avoid it.
- To become aware of some of the subtle pitfalls that can befall a programmer if great care is not taken when dealing with conditional expressions.
- To understand how the *conditional operator* `? :` works.
- To understand why *enumerated types* are useful, how they work, and how to define and use them.
- To learn how numbers are represented in different *bases*, in particular base 10 (*decimal*), base 2 (*binary*), base 8 (*octal*), and base 16 (*hexadecimal*).
- To learn some C++ *bitwise operators* and understand how they work.
- To enhance your understanding of the *scope* and *lifetime* of variables.

19.2 List of associated files

- `limits.cpp` displays system-dependent limits for simple data types.
- `type_size.cpp` displays the size in bytes of some C++ simple data types.
- `increment_decrement.cpp` shows why you must be careful when using `++` and `--`.
- `dangling_else.cpp` illustrates the "dangling else" problem.
- `bool_errors.cpp` illustrates some potential pitfalls to avoid when dealing with conditional expressions.
- `conditional_operator.cpp` illustrates the conditional operator `? :`.
- `enumerated_type.cpp` illustrates some properties of enumerated types.
- `number_bases.cpp` displays numbers in *decimal*, *octal*, and *hexadecimal* form.
- `bit_operators.cpp` illustrates some C++ bitwise operators.
- `scope1.cpp` to `scope6.cpp` contain some "pathological" examples to test your understanding of variable scope and lifetime.

19.3 Overview

Much as we would like to know everything at once when we start to learn a new subject, we recognize that this is not possible. And so it is with a new programming language. This Module fills in a few of the gaps that we have left along the way in earlier Modules. Many of the things you see here are not necessary for the vast majority of C++ programs that you will write, but some of them may prove very convenient from time to time, and you should, of course, at least be aware of all of them, if only to be able to recognize and interpret them when you see them in code written by other C++ programmers.

Some of the things you see in any or all of the sample programs in this Module you may or may not be seeing for the first time. These sample programs are meant to be examined and/or discussed in class as the need arises, and if you have finished all of the other Modules before looking at anything in this one, which is entirely possible, then some of the features here will certainly have been seen before. On the other hand, if you've jumped to one of the programs in this Module from somewhere earlier in the text, then some of the features here that are "advertised" as being new really will be.

19.4 Sample Programs

19.4.1 limits.cpp displays system-dependent limits for simple data types

```
//limits.cpp
//Displays constant values associated with various data types.

#include <iostream>
#include <climits>
#include <cfloat>
using namespace std;

int main()
{
    cout << "\nThis program displays maximum and minimum "
         << "values for simple C++ data types.\n\n";

    //The following named constants are defined in <climits>:
    cout << "Number of bits in a byte  " << CHAR_BIT  << endl;
    cout << "Maximum char value ...... " << CHAR_MAX  << "\t\t";
    cout << "Minimum char value ...... " << CHAR_MIN  << endl;
    cout << endl;
    cout << "Maximum short value ..... " << SHRT_MAX  << "\t\t";
    cout << "Minimum short value ..... " << SHRT_MIN  << endl;
    cout << "Maximum int value ....... " << INT_MAX   << '\t';
    cout << "Minimum int value ....... " << INT_MIN   << endl;
    cout << "Maximum long value ...... " << LONG_MAX  << '\t';
    cout << "Minimum long value ...... " << LONG_MIN  << endl;
    cout << endl;
    cout << "Maximum unsigned char value .... " << UCHAR_MAX << endl;
    cout << "Maximum unsigned short value ... " << USHRT_MAX << endl;
    cout << "Maximum unsigned int value ..... " << UINT_MAX  << endl;
    cout << "Maximum unsigned long value .... " << ULONG_MAX << endl;
    cout << endl;

    //The following named constants are defined in <cfloat>:
    cout << "Approx # of sig digits in a float ..... " << FLT_DIG  << endl;
    cout << "Maximum positive float value ......... " << FLT_MAX  << endl;
    cout << "Minimum positive float value ......... " << FLT_MIN  << endl;
    cout << "Approx # of sig digits in a double .... " << DBL_DIG  << endl;
    cout << "Maximum positive double value ........ " << DBL_MAX  << endl;
    cout << "Minimum positive double value ........ " << DBL_MIN  << endl;
    cout << "Approx # of sig digits in a long double " << LDBL_DIG << endl;
    cout << "Maximum positive long double value .... " << LDBL_MAX << endl;
    cout << "Minimum positive long double value .... " << LDBL_MIN << endl;
}
```

19.4.1.1 What you see for the first time in limits.cpp

System-defined constants

- A number of *system-defined constants*[1]

  These are values that differ, or at least potentially differ, from one C++ implementation to another. They determine the largest and smallest values of the simple built-in data types on your system

Header files `climits` *and* `cfloat`

- The inclusion of two new C++ header files—`climits` and `cfloat`—which contain definitions for these constants

- The C++ style convention for capitalization of built-in named constants

  This convention requires all capital letters, with underscore separators between words. This is the same convention that we follow for our own *programmer-defined named constants.*

- Appearance of `\t` as a single character, within single quotes (rather than as a string constant, within double quotes)

  The effect of outputting either `'\t'` or `"\t"` to the screen is the same.

19.4.1.2 Additional notes and discussion on limits.cpp

System dependencies

Every C++ implementation has a number of *system dependencies*, such as the values defined by the constants shown in `limits.cpp`. The implications of this may not be serious in your case, but you must be aware of these differences from one system to another, just in case they do become important.

Just how a system dependency might affect one of your programs, if at all, will depend on what your program is trying to do. But, for example, since the maximum size of an `int` value is one of these system dependencies, a program which worked fine on one system may fail on another, even though the same input was used, because a computed value which was within range on one system was out of range on the other.

Portability questions

Questions such as these often plague programmers in the real world, and bring up the whole problem of *portability*. Writing *portable programs* means writing programs that are unaffected, or affected as little as possible, by system dependencies. Like many other decisions in programming, the amount of portability you want to incorporate into your programs, and the amount of effort you want to expend to achieve it, is a judgment call. The compromise that usually has to be made is something like this: You can perhaps make your program very fancy by taking advantage of all the "bells and whistles" available on a particular system, but the resulting program will *not* be very portable and may, in fact, run *only* on the system on which it was developed.

[1] Note that any *system-defined constant* is also what we have called a *predefined constant* but a system-defined constant refers specifically to one whose value is not only predefined but *may* differ from system to system.

19.4.1.3 Follow-up hands-on activities for limits.cpp

□ Activity 1 Copy, study and test the program in `limits.cpp`.

□ Activity 2 Copy `limits.cpp` to `limits1.cpp` and bug it as follows:

a. Leave out the line that includes the `limits` header file.

____

b. Leave out the line that includes the `float` header file.

____

c. Replace `INT_MAX` with `INT_MAKS` in line 21.

____

d. Replace each instance of `'\t'` with `"\t"`.

____

e. Replace `"\t\t"` with `'\t\t'`.

____

○ Instructor checkpoint 19.1 for evaluating prior work

19.4.2 type_size.cpp displays the size in bytes of some C++ simple data types

```
//type_size.cpp
//Displays the byte-size of common data types.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program displays a table showing the byte-size of "
        "some simple data types,\n(the number of bytes of storage "
        "occupied by a value of each data type).\n";

    cout << "\nData Type    Bytes"
         << "\n---------    -----"
         << "\nchar           "  << sizeof(char)
         << "\nshort          "  << sizeof(short)
         << "\nint            "  << sizeof(int)
         << "\nlong           "  << sizeof(long)
         << "\nfloat          "  << sizeof(float)
         << "\ndouble         "  << sizeof(double)
         << "\nlong double    "  << sizeof(long double);
    cout << endl << endl;
}
```

19.4.2.1 What you see for the first time in type_size.cpp

`sizeof` *computes number of bytes of storage used*

This program uses the C++ `sizeof`[2] operator to obtain the amount of storage space (*bytes*) required by a value of each of the simple built-in C++ data types.

19.4.2.2 Additional notes and discussion on type_size.cpp

As you continue to study and learn about new data types, you should keep the `sizeof` operator in mind and use it to check the storage requirements of values in those new data types.

19.4.2.3 Follow-up hands-on activities for type_size.cpp

□ Activity 1 Copy, study and test the program in `type_size.cpp`.

□ Activity 2 Copy `type_size.cpp` to `type_size1.cpp` and bug it as follows:

a. Remove the parentheses from `(char)` to see if they are really necessary.

b. Try the same thing with `(long double)`.

○ INSTRUCTOR CHECKPOINT 19.2 FOR EVALUATING PRIOR WORK

[2] Note that despite the fact that `sizeof` is often written in such a way that it looks like a function, it is actually an operator.

19.4.3 increment_decrement.cpp shows why you must be very careful when using ++ and --

```
//increment_decrement.cpp
//Illustrates why you shouldn’t use ++ or --, except in a
//standalone statement that increments or decrements a variable.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program shows you some examples of situations in "
        "which the increment\nand decrement operators, ++ and --, are "
        "sometimes used, but shouldn’t be,\nbecause the code is not "
        "very readable, and is fraught with other dangers.\nStudy the "
        "code, predict the output in each case, and then ...\n\n";

    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’);
    cout << endl << endl;

    int i, j;

    i = 4;
    j = i++ + 1;
    cout << i << " " << j << endl;
    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’);

    i = 4;
    j = ++i + 1;
    cout << i << " " << j << endl;
    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’);

    i = 4;
    j = i-- + 1;
    cout << i << " " << j << endl;
    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’);

    i = 4;
    j = --i + 1;
    cout << i << " " << j << endl;
    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’);

    cout << endl;

    i = 25;
    cout << i   << endl;
    cout << ++i << endl;
    cout << i   << endl;
    cout << i++ << endl;
    cout << i   << endl;
    cout << "Press Enter to continue ... ";  cin.ignore(80, ’\n’);
}
```

19.4.3.1 What you see for the first time in increment_decrement.cpp

This program shows situations in which the increment and decrement operators `++` and `--` are *not* used simply to increment or decrement a variable in a standalone statement (which was the way we have recommended they be used).

19.4.3.2 Additional notes and discussion on increment_decrement.cpp

When you read C++ code that other programmers have written, you will often see these operators used in ways that we have recommended that they *not* be used, and this program is simply meant to be further evidence that our recommendation was not a bad one. Even though we recommend you not program like this, it will be helpful if you have some idea what is going on when you read code that uses the operators in this way.

Tricky behavior of operators `++` and `--`

So, if you really *must* use these operators in non-standalone expressions, or if you are trying to read code that does, then here is what you need to remember:

- When `variable++` is used in an expression, the value of `variable` is *first used* in the expression and *then incremented*.
- When `++variable` is used in an expression, the value of `variable` is *first incremented* and *then used* in the expression.

  And similarly ...
- When `variable--` is used in an expression, the value of `variable` is *first used* in the expression and *then decremented*.
- When `--variable` is used in an expression, the value of `variable` is *first decremented* and *then used* in the expression.

Study both the output and the source code of this sample program and make sure you can reconcile what you see with the above descriptions of the behavior of these operators.

19.4.3.3 Follow-up hands-on activities for increment_decrement.cpp

□ Activity 1 Copy, study and test the program in `increment_decrement.cpp`.

○ INSTRUCTOR CHECKPOINT 19.3 FOR EVALUATING PRIOR WORK

19.4.4 dangling_else.cpp illustrates the “dangling else” problem

```
//dangling_else.cpp
//Illustrates the "dangling else" problem.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program allows you to analyze the dangling else "
        "problem.\nStudy the source code to predict the output for "
        "each of these input values:\n9, 5, and 0\n\n";

    int numberOfDays;

    cout << "Enter the number of days it has rained without stopping: ";
    cin >> numberOfDays;  cin.ignore(80, '\n');  cout << endl;

    cout << "\nStart of first construct:  ";
    if (numberOfDays > 1)
        if (numberOfDays <= 7)
            cout << "Get out your umbrellas!\n";
        else
            cout << "Run for higher ground!\n";

    cout << "\nStart of second construct: ";
    if (numberOfDays <= 7)
        if (numberOfDays > 1)
            cout << "Get out your umbrellas!\n";
    else
        cout << "Run for higher ground!\n";
    cout << endl;
}
```

19.4.4.1 What you see for the first time in dangling_else.cpp

Dangling else and nested if

Needless to say, difficulties can arise in a program when that program needs to be told how to make complex decisions. One of those difficulties is the so-called *dangling else problem*, which occurs when a code segment containing a nested-if construct has more occurrences of the keyword `if` than the keyword `else`, as in the (simple) case shown in `dangling_else.cpp`. In such a situation, the following question arises: With which `if` is the `else` (or the final `else`, if there are several) actually associated?

C++ is a "free-format" programming language.

You will note that in the given code the formatting suggests that in the first nested-if the `else` is associated with the second `if`, while in the second nested-if, the formatting suggests that the `else` is associated with the first `if`. You should also recall that because C++ is a *free-format language*, the formatting has *no effect* on *how* C++ interprets these two code segments.

The answer to the above question in this case is that C++ interprets the code in the way that the first formatting suggests, and, in general, the "dangling" `else` will always be associated with the *nearest preceding* `if` that does not already have a corresponding `else`.

If we wish to "force" the `else` to be associated with the first `if`, then we can do so by inserting braces. That is, if we really want what the second formatting suggests that we want, then the second construct should be written as follows:

```
if (numberOfDays <= 7)
{
    if (numberOfDays > 1)
        cout << "Get out your umbrellas!\n";
}
else
    cout << "Run for higher ground!\n";
```

The bottom line is this: The best way to avoid the dangling else problem is to insert whatever braces are necessary to ensure the code is both highly readable by humans and properly interpreted by the computer.

19.4.4.2 Additional notes and discussion on dangling_else.cpp

Good formatting helps to avoid this problem (and many others!).

Once again we are reminded that *good formatting*, though only for the user and not for the compiler (unlike *correct formatting*, which clearly *is* for the compiler), is very important in our programs. The real lesson to be learned is simply this: If you format your code so that it is absolutely clear to a human reader what is to be done, it is quite likely to be absolutely clear to the computer as well.

19.4.4.3 Follow-up hands-on activities for dangling_else.cpp

□ Activity 1 Copy, study and test the program in `dangling_else.cpp`.

□ Activity 2 Copy `dangling_else.cpp` to `dangling_else1.cpp` and bug it as follows:

a. At the end of the if...else in the first construct, add another `else` and a `cout` statement that displays the string “No weather report!”.

____

b. At the end of the if...else in the second construct, add another `else` and a `cout` statement that displays the string “No weather report!”.

____

c. Add braces to force the `else` in the second construct to be associated with the first `if`.

____

○ INSTRUCTOR CHECKPOINT 19.4 FOR EVALUATING PRIOR WORK

19.4.5 bool_errors.cpp illustrates some potential pitfalls to avoid when using conditional expressions

```
//bool_errors.cpp
//The purpose of this program is to highlight some of pitfalls to
//avoid when dealing with boolean data values and testing conditions.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program is designed to freak you out, and "
        "serve as a warning.\n\nHere's what you should do:\n"
        "1. On the first run, enter the values 6, then 3 and 4, "
        "then 1.2, 2.3 and 3.4."
        "\n2. Now check the source code and reconcile "
        "the input and output."
        "\n3. On the second run, enter the values 7, then 8 and 9, "
        "then 3.4, 2.3 and 1.2 "
        "\n4. Now check the source code and reconcile "
        "the input and output once more.\n";

    int i, j;

    cout << "\nEnter an integer: ";
    cin >> i;  cin.ignore(80, '\n');
    if (i = 6)  //First error: What is it?
        cout << "The value entered was 6.\n";


    cout << "\nEnter two more integers: ";
    cin >> i >> j;  cin.ignore(80, '\n');
    if (i == 3 || 4)  //Second error: What is it?
        cout << "The first value entered was either 3 or 4.\n";
    else
        cout << "The first value entered was neither 3 nor 4.\n";

    if (i || j == 4)  //Third error: What is it?
        cout << "One of the values entered was 4.\n";
    else
        cout << "Neither value entered was 4.\n";


    double x, y, z;

    cout << "\nEnter three real numbers: ";
    cin >> x >> y >> z;  cin.ignore(80, '\n');
    if (x < y < z)  //Fourth error: What is it?
        cout << "The values were entered in increasing order.\n";
    else
        cout << "The values were not entered in increasing order.\n";
    cout << endl;
}
```

19.4.5.1 What you see for the first time in bool_errors.cpp

In this program you see some examples of the kinds of errors that all C++ programmers are (unfortunately) prone to make. What makes these types of errors so insidious is that the compiler may or may not complain, the fact that there has been an error may not always show up in the output, and even when you discover that there is a problem, the problem may not be easy to find. This is because in each case the problem code actually does mean something to C++, just not what you intended it to mean. It is to find errors like these that you might have to resort to a trace of your program.

19.4.5.2 Additional notes and discussion on bool_errors.cpp

Wonderful language though it is, C++ is nevertheless a bit of a minefield for the unwary programmer. Part of the reason for this stems from the fact that one of the design goals for C++ was to be *backward compatible* with the C programming language. C is a wonderful language too, but *backward compatibility* with it meant that C++ had to retain some of the more unpleasant and "dangerous" features of the C language. The problem is, as a programmer you can make some very "natural" mistakes in C++ that will go unnoticed by the compiler. So, the best advice is, once again: Be vigilant!

Backward compatibility of C++ with C is a mixed blessing.

19.4.5.3 Follow-up hands-on activities for bool_errors.cpp

□ Activity 1 Copy, study and test the program in `bool_errors.cpp`. (If you get a warning message from your compiler, you should certainly read it and determine what it is trying to tell you, but also, for puposes of this activity, ignore it.)

□ Activity 2 Perform a trace of the program in `bool_errors.cpp` by using a copy of the usual tracing template. You can omit the executable statements that just give the program description, of course.

○ Instructor checkpoint 19.5 for evaluating prior work

19.4.6 conditional_operator.cpp illustrates the conditional operator ? :

```
//conditional_operator.cpp
//Illustrates use of the conditional operator ? :

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program rounds a real number temperature value to "
        "an integer, chooses\nbetween a singular and a plural, and "
        "chooses the maximum of two integers.\nStudy the source code "
        "to see how the conditional operator ? : is used to\naccomplish "
        "these amazing feats.\n";

    double temp;
    cout << "\nEnter a temperature value to the nearest "
        "tenth of a degree: ";
    cin >> temp;  cin.ignore(80, '\n');  cout << endl;
    cout << "Rounded to the nearest integer, the temperature is "
         << (temp >= 0  ?  int(temp+0.5)  :  int(temp-0.5)) << ".\n\n";

    int numberOfTries;
    cout << "Enter the number of tries you took to do something: ";
    cin >> numberOfTries;  cin.ignore(80, '\n');  cout << endl;
    cout << "Wow! You guessed correctly in " << numberOfTries
         << (numberOfTries == 1  ?  " try.\n"  :  " tries.\n");

    int i, j, max;
    cout << "\nEnter two integers: ";
    cin >> i >> j;  cin.ignore(80, '\n');  cout << endl;
    max = (i > j  ?  i  :  j);
    cout << "The larger of the two integers is " << max << ".\n";
    cout << endl;
}
```

19.4.6.1 What you see for the first time in conditional_operator.cpp

The conditional operator ? :

This program shows how to use the C++ *conditional operator* ? : as an alternative to the if...else-statement, though it is best used only in simple cases.

19.4.6.2 Additional notes and discussion on conditional_operator.cpp

The conditional operator is one of those language features that we could do without if we had to, but which is nevertheless very convenient to have available from time to time. Most programming languages have features like this—unnecessary but useful items of one kind or another—and the term *syntactic sugar* is sometimes used to refer to them.

19.4.6.3 Follow-up hands-on activities for conditional_operator.cpp

□ Activity 1 Copy, study and test the program in conditional_operator.cpp.

□ Activity 2 Copy `conditional_operator.cpp` to `conditional_operator1.cpp` and bug it as follows:

a. Remove the outer parentheses from the following conditional expression, which appears in line 20, to see if they are needed:
`(temp >=  0  ?  int(temp+0.5)  :  int(temp-0.5))`

____

b. Remove the outer parentheses from the following conditional expression, which appears in line 26, to see if they are needed:
`(numberOfTries == 1  ?  " try."  :  " tries.")`

____

c. Remove the outer parentheses from the following conditional expression, which appears in line 31, to see if they are needed:
`(i > j  ?  i  :  j)`

____

What is your explanation for any different outcomes observed in making the three above changes in the sample program?

____

____

____

____

□ Activity 3 Make a copy of the file `conditional_operator.cpp` and call the copy `conditional_operator2.cpp`. Then modify the copy so that each instance of the conditional operator `? :` is replaced by an equivalent if...else-statement. Make any other necessary changes as well, of course.

○ Instructor checkpoint 19.6 for evaluating prior work

19.4.7 enumerated_type.cpp illustrates some properties of enumerated types

```
//enumerated_type.cpp
//Illustrates some properties of C++ enumerated types.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program illustrates some properties of enumerated "
        "data types\nand variables. Study the source code for details.\n";

    enum ColorType { RED, WHITE, BLUE }; //Define an enumerated type
    ColorType color, myColor, yourColor; //Declare variables of this type

    myColor = RED;     //Assign enumerated values to
    yourColor = BLUE;  //variables of the enumerated type

    yourColor = myColor; //Assign value of one variable to another

    if (yourColor == RED)
        cout << "\nThe value of \"yourColor\" is now RED.\n";
    else
        cout << "\nI don't know what \"yourColor\" is now.\n";

    color = ColorType (RED + 1); //Note how incrementing is accomplished
    if (color == WHITE)
        cout << "\nThe value of \"color\" is now WHITE.\n";
    else
        cout << "\nI don't know what \"color\" is now.\n";

    int i;
    for (color = RED;                  //Use of a variable of a
         color <= BLUE;                //variable of an enumerated
         color = ColorType (color + 1)) //type as a loop control variable
    {
        cout << "The internal integer representation of ";
        switch (color) // Use of enumerated variable as case-selector
        {
            case RED:   cout << "RED";   break; //Use of enumerated
            case WHITE: cout << "WHITE"; break; //values as case-labels
            case BLUE:  cout << "BLUE";  break; //in a switch-statement
        }
        i = color;
        cout << " is " << i << ".\n";
    }
    cout << endl;

    //The following statement would cause a compile-time error. Why?
    //color = 2;
}
```

19.4.7.1 What you see for the first time in enumerated_type.cpp

`enum` *and enumerated data types*

- The definition of an *enumerated data type* using the C++ reserved word `enum`

  Note that any *enumerated data type* is a *programmer-defined data type*, *not* a *built-in data type.*

- The declaration of variables of an enumerated data type and assignment of values to such variables
- The use of enumerated variables in if-statements and a switch-statement
- The use of an enumerated-type variable as a loop control variable

19.4.7.2 Additional notes and discussion on enumerated_type.cpp

The enumerated data type is another one of those language features that is not strictly needed. However, enumerated types are very useful in permitting programmers to "invent" new data types for many situations, and most modern programming languages include them in some form.

19.4.7.3 Follow-up hands-on activities for enumerated_type.cpp

□ Activity 1 Copy, study and test the program in `enumerated_type.cpp`.

□ Activity 2 Copy `enumerated_type.cpp` to `enumerated_type1.cpp`. Begin by "uncommenting"[3] the line

Be sure you understand what's going on here.

```
//color = 2;
```

to make it an executable statement. Then try to re-compile the program to verify that this statement really does cause a compile-time error, and explain why this happens.

○ Instructor checkpoint 19.7 for evaluating prior work

[3] This rather cumbersome term simply means "to turn the comment(s) back into actual code by removing, in this case at least, the two forward slashes". It's cumbersome, but would "de-comment", say, be any better? Perhaps "activate" would work . . . hmmm . . .

19.4.8 number_bases.cpp displays numbers in decimal, octal, and hexadecimal form

```
//number_bases.cpp
//Displays numbers in decimal, octal and hexadecimal form.

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

int main()
{
    cout << "\nThis program displays numbers in decimal, octal, and "
        "hexadecimal (\"hex\") forms.\nTo make sense of everything, "
        "study both the source code and the output.\n\nIn the first "
        "table below, look at any row.  Each value is written using\n"
        "the same digits (in the source code), but in a different base, "
        "and is\noutput in (default) decimal format in each case.\n"
        "Decimal  Octal  Hex\n"
        "-------------------\n";
    cout << setw(4) <<     6
         << setw(8) <<    06
         << setw(7) <<   0x6 << endl
         << setw(4) <<    14
         << setw(8) <<   014
         << setw(7) <<  0x14 << endl
         << setw(4) <<   123
         << setw(8) <<  0123
         << setw(7) << 0x123 << endl;

    cout << "\nIn this second table, every number actually has the "
        "same value (61 decimal),\nbut each of the three rows shows "
        "the values written using a different base.\n"
        "Dec/Oct/Hex: 61   075   0X3D\n"
        "----------------------------\n";
    cout << "Decimal: "
         << setw(6) <<   61       //Decimal output is the default.
         << setw(6) <<  075
         << setw(6) << 0X3D << endl
         << "Octal:   "
         << setw(6) << oct << 61  //Now we use the "oct" manipulator.
         << setw(6) << oct << 075
         << setw(6) << oct << 0X3D << endl
         << "Hex:     "
         << setw(6) << hex << 61  //And now the "hex" manipulator.
         << setw(6) << hex << 075
         << setw(6) << hex << 0X3D << endl;
    cout << endl;
}
```

19.4.8.1 What you see for the first time in number_bases.cpp

- C++ rules for designating literal constants as base eight (*octal*) or base sixteen (*hexadecimal*) quantities
- The `oct` and `hex` manipulators, which determine the base to be used to represent subsequent output quantities

19.4.8.2 Additional notes and discussion on number_bases.cpp

Any data value is represented internally in a computer by a sequence of zeros and ones. Another way of saying this is that data representation inside a computer is based on the *binary number system* (i.e., the number system with base 2, in which the only two digits are 0 and 1). More correctly, the internal representation is based on some physical phenomenon which has two mutually exclusive and easily identifiable states, such as tiny switches that can be either on or off, or some other electrical device which can retain a voltage that may be high or low.

Binary numbers have base 2.

Although very convenient for computers, binary numbers are much less so for humans. For one thing, the binary representation of numbers soon becomes very large and unwieldy, even if the quantity being represented is not particularly large. For that reason, humans are more comfortable using number systems with bases other than 2. As you are no doubt aware, the decimal number system that we use in everyday life is the number system with base 10. Two number systems that are more convenient than the binary number system for working with computers are the octal (base 8) and especially the hexadecimal (base 16) number systems.

Decimal numbers have base 10.

Octal numbers have base 8.

Hexadecimal numbers have base 16.

19.4.8.3 Follow-up hands-on activities for number_bases.cpp

□ Activity 1 Copy, study and test the program in `number_bases.cpp`.

□ Activity 2 Copy `number_bases.cpp` to `number_bases1.cpp` and bug it as follows:

a. Change the octal number 014 to 019.

b. Make a change that will test whether the `oct` and `hex` manipulators are *persistent*, i.e., whether they apply to *all* subsequent values that are output until a new such manipulator is encountered, or whether, like `setw`, they apply *only* to the *next* value output. This is equivalent to asking whether *each* instance of `oct` and `hex` used in the program is necessary, or only the *first* instance in each case.

○ Instructor checkpoint 19.8 for evaluating prior work

19.4.9 bit_operators.cpp illustrates some of the C++ bitwise operators

```
//bit_operators.cpp
//Illustrates the effects of some C++ bitwise operators.

#include <iostream>
using namespace std;

int main()
{
    cout << "\nThis program demonstrates the effect of applying some of "
        "the C++\nbitwise operators to an integer value.\n\n";

    int i, j;

    i = 7;
    j = i << 2;  //The "left shift" operator
    cout << i << " left-shifted by 2 becomes " << j << endl;
    cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');

    i = 43;
    j = i >> 3;  //The "right-shift" operator
    cout << endl << i << " right-shifted by 3 becomes " << j << endl;
    cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');

    i = 43;
    j = i & 15;  //The "bitwise and" operator
    cout << endl << i << ", after a \"bitwise and\" with 15, becomes "
         << j << endl;
    cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');

    i = 42;
    j = i | 28;  //The "bitwise or" operator
    cout << endl << i << ", after a \"bitwise or\" with 28, becomes "
         << j << endl;
    cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');

    i = 42;
    j = i ^ 28;  //The "bitwise exclusive-or" operator
    cout << endl << i << ", after a \"bitwise exclusive-or\" with 28, "
         << "becomes " << j << endl;
    cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');

    i = 2147463647;
    j = ~i;      //The "bitwise complement" operator
    cout << endl << i << ", after a \"bitwise complement\", becomes "
         << j << endl;
    cout << "Press Enter to continue ... ";  cin.ignore(80, '\n');
}
```

19.4.9.1 What you see for the first time in bit_operators.cpp

This program illustrates the C++ *bitwise operators* shown in the following table:

```
<<  left shift, with zero fill on the right
>>  right shift, with (usually) sign-bit fill on the left
&   logical and
|   logical or
^   exclusive or
~   complement
```

19.4.9.2 Additional notes and discussion on bit_operators.cpp

These operators allow a C++ programmer to manipulate directly the bits stored in a computer's memory locations. This sort of *low-level programming* (or *bit twiddling*, as it is sometimes called, tongue in cheek), is important for *systems programming*, but the average C++ programmer does not have much call for these operators. You should at least be aware of their existence, and have a basic knowledge of how each works. However, we do not discuss them further. If you encounter a situation where you actually have to use one or more of them, you will want to delve further into their properties (their operator precedence, for example).

A very brief look, just in passing

19.4.9.3 Follow-up hands-on activities for bit_operators.cpp

□ Activity 1 Copy, study and test the program in `bit_operators.cpp`. Try changing some of the numbers and predicting the output for the same operations with your new values.

○ INSTRUCTOR CHECKPOINT 19.9 FOR EVALUATING PRIOR WORK

19.4.10 scope1.cpp to scope6.cpp contain some "pathological" examples to test your understanding of variable scope and lifetime

These programs will test not only your skill, but also your patience and possibly your sanity.

This sequence of programs gives you an opportunity to take a closer look at the notions of scope and lifetime of variables. The programs do not do anything useful or interesting, so there are no "context clues" to help you determine the output values. You must understand certain concepts and be able to relate those concepts to the program flow as you trace each program if you are to determine the correct output.

19.4.10.1 What you see for the first time in scope1.cpp to scope6.cpp

- The use of *global variables* (variables declared outside any function, and also called *external variables*), and how they force you to "look outside" any function that refers to such a variable to see what the function is doing
- The use of *global variables* and *local variables* with the same name
- The use of global variables and function parameters with the same name

  This is not really different from the bulleted item immediately above, since a function parameter has the same scope as a local variable defined in that function.

19.4.10.2 Additional notes and discussion on scope1.cpp to scope6.cpp

And a valuable lesson it is!

One of the lessons to be taken away from your examination of these programs is how hard it can be to follow the execution flow of programs that contain global variables. Beginning programming students are often tempted to use nothing but global variables, since doing so seems to make life easier by making all variables accessible everywhere to all functions. What could be simpler?

In fact, using global variables is often a recipe for disaster, and the first sign that a program is on its way to the scrap heap. Take a lesson from history: Early programming languages sometimes provided nothing but global variables, and there are many horror stories in the literature detailing the many nightmares that arose, at least in part, from their use.

So, learning about and appreciating the dangers of global variable use is important, but understanding the whole (related) notion of *scope* is also critical to a full understanding of C++ program structure. In particular, the *scope of a variable* refers to the region of a program in which that variable is available for use, or can be accessed.

The full scope rules of C++ are quite complex, but the key rule to remember, for the moment, is that once a variable has been declared in a function block, or in any other block for that matter, it is *local* to that block and hence:

- It is available throughout the rest of that block (including any interior blocks, unless it is *overridden* (i.e., re-declared) by the declaration of another variable with the same name in one of those interior blocks).

- But, it is also "hidden" within that block and is unavailable (cannot be accessed) outside that block.

These two basic ideas, together with the realization that, from a scope point of view, both the parameters of a function and any variable declared in the function body are *local to the function* and *unknown* outside the function, will get you through a lot of scope difficulties if you understand them thoroughly.

Thus, implied by all of this is the fact that a variable defined outside any function (i.e., a *global variable*), is *global* in the following sense: Since it is not "hidden" inside any block, it is known from the point of its declaration throughout the rest of the file containing the program and hence is available to all functions that physically follow its declaration in the file. That is, the *scope* of a global variable is the entire file from its point of declaration onward, and therefore (potentially, at least) throughout the entire program if the program is contained in that one file. And therein lies the source of many problems, so situations that fit this description should be avoided.

Scope of variables

Another notion that it is important to grasp is that of the *lifetime* of a variable. This term refers to the length of time that a variable has memory *allocated* to it. The simple rule is this: If a variable is declared in a block (say a function block, or even a while-loop body enclosed in braces), it "comes into existence" (i.e., has memory allocated to it) when the declaration within that block is reached, and "goes out of existence" (i.e., has its memory *deallocated*) when that block finishes executing. Implied by this, for example, is the fact that if a variable is declared as a local variable inside a function it "comes and goes" with each function call, while the lifetime of a variable declared outside any function (i.e., a *global variable*) extends from the beginning to the end of program execution.

Lifetime of variables

19.4.10.3 scope1.cpp with follow-up hands-on activities

```
//scope1.cpp
//A nonsense program to test your understanding
//of variable scope and lifetime.

#include <iostream>
using namespace std;

void DoSomething(int& x);

int a = 7;  //Global variables (declared outside any function)
int b = 8;
int c = 9;

int main()
{
    cout << endl;
    cout << a << b << c << endl;
    DoSomething(a);
    cout << a << b << c << endl;
}

void DoSomething(int& x)
{
    int b;  //Local variable (with the same name as a global variable)

    a = 1;
    b = 2;  //Which "b" is this, the local or the global?
    x = 3;
    c = 4;
    cout << a << b << c << endl;
}
```

□ Activity 1 Copy, study, and trace the program in `scope1.cpp`. Then predict the output before running the program and comparing the output with your prediction.

□ Activity 2 Copy `scope1.cpp` to `scope1a.cpp` and bug it as follows:

a. Remove the initializations of the global variables `a`, `b` and `c` (but leave the declarations).

____

b. Convert `x` from a reference parameter to a value parameter.

____

c. Replace `DoSomething(a)` with `DoSomething(b)`.

____

d. Replace the identifier `x` by the identifier `a` everywhere in the program.

____

○ Instructor checkpoint 19.10 for evaluating prior work

19.4.10.4 scope2.cpp with follow-up hands-on activities

```
//scope2.cpp
//A nonsense program to test your understanding
//of variable scope and lifetime.

#include <iostream>
using namespace std;

char firstChar, secondChar;  //Global variables

void P1()
{
    char firstChar;   //Local to P1

    firstChar  = ’A’; //Are these
    secondChar = ’B’; //local or global?
}

void P2()
{
    char secondChar;  //Local to P2

    firstChar  = ’C’; //Are these
    secondChar = ’D’; //local or global?
    P1();
    cout << firstChar << secondChar << endl;
}

int main()
{
    cout << endl;
    firstChar = ’E’;  //Are these
    secondChar = ’F’; //local or global?
    P2();
    cout << firstChar << secondChar << endl;
}
```

□ Activity 1 Copy, study, and trace the program in `scope2.cpp`. Then predict the output before running the program and comparing the output with your prediction.

□ Activity 2 Copy `scope2.cpp` to `scope2a.cpp` and bug it as follows:

a. Remove the declaration of the global variable `firstChar`.

____

b. Remove the declaration of the global variable `secondChar`.

____

c. Remove the declaration of the local variable `firstChar` in `P1`.

____

d. Remove the declaration of the local variable `secondChar` in `P2`.

____

○ Instructor checkpoint 19.11 for evaluating prior work

19.4.10.5 scope3.cpp with follow-up hands-on activities

```
//scope3.cpp
//A nonsense program to test your understanding
//of variable scope and lifetime.

#include <iostream>
using namespace std;

int i, j;      //Global variables

void P(int& i) //Parameter i is local to P
{
    int j;    //Local to P

    j = 6;
    cout << i << j << endl;  //Which "i" and "j"
    i = i + j;               //are being used here?
    cout << i << j << endl;
}

int main()
{
    cout << endl;
    i = 3;    //Which "i" and "j"
    j = 4;    //are being used here?
    cout << i << j << endl;
    P(j);
    cout << i << j << endl;
}
```

□ Activity 1 Copy, study, and trace the program in `scope3.cpp`. Then predict the output before running the program and comparing the output with your prediction.

□ Activity 2 Copy `scope3.cpp` to `scope3a.cpp` and bug it as follows:

a. Convert the function parameter to a value parameter.

   ____

b. Change the function call from `P(j)` to `P(i)`.

   ____

c. Convert the function parameter to a value parameter, and change the function call from `P(j)` to `P(i)`.

   ____

○ Instructor checkpoint 19.12 for evaluating prior work

19.4.10.6 scope4.cpp with follow-up hands-on activities

```
//scope4.cpp
//A nonsense program to test your understanding
//of variable scope and lifetime.

#include <iostream>
using namespace std;

void P(int& i, int& j)
{
    i = 3*i;
    cout << i << j << endl;
    j = i + j;
    cout << i << j << endl;
}

int i, j; //Global variables

int main()
{
    cout << endl;
    i = 2;
    j = 3;
    cout << i << j << endl;
    P(j, j);
    cout << i << j << endl;
}
```

□ Activity 1 Copy, study, and trace the program in `scope4.cpp`. Then predict the output before running the program and comparing the output with your prediction.

□ Activity 2 Copy `scope4.cpp` to `scope4a.cpp` and bug it as follows:

a. Remove the first occurrence of `&` in the function's parameter list.

____

b. Remove the second occurrence of `&` in the function's parameter list.

____

c. Remove *both* occurrences of `&` in the function's parameter list.

____

d. Replace the function call `P(j, j)` with `P(i, i)`.

____

e. Replace the function call `P(j, j)` with `P(i, j)`.

____

f. Replace the function call `P(j, j)` with `P(j, i)`.

____

○ INSTRUCTOR CHECKPOINT 19.13 FOR EVALUATING PRIOR WORK

19.4.10.7 scope5.cpp with follow-up hands-on activities

```
//scope5.cpp
//A nonsense program to test your understanding
//of variable scope and lifetime.

#include <iostream>
using namespace std;

int i, j, k, t; //Global variables

void DoSomething(int i, int j)
{
    int t; //Local to "DoSomething"

    t = i;
    i = j;
    j = t;
    cout << i << j << k << t << endl;
}

int main()
{
    cout << endl;
    i = 1;  j = 2;  k = 3;  t = 4;
    cout << i << j << k << t << endl;
    DoSomething(i, j);
    cout << i << j << k << t << endl;
}
```

□ Activity 1 Copy, study, and trace the program in `scope5.cpp`. Then predict the output before running the program and comparing the output with your prediction.

□ Activity 2 Copy `scope5.cpp` to `scope5a.cpp` and bug it as follows:

a. Convert the parameter `i` in `DoSomething` to a reference parameter.

____

b. Convert the parameter `j` in `DoSomething` to a reference parameter.

____

c. Convert both parameters `i` and `j` in `DoSomething` to reference parameters.

____

○ Instructor checkpoint 19.14 for evaluating prior work

19.4.10.8 scope6.cpp with follow-up hands-on activities

```
//scope6.cpp
//A nonsense program to test your understanding
//of variable scope and lifetime.

#include <iostream>
using namespace std;

int j = 3; //Global variable

void DoThis(int& i, int j)
{
    i = 3 * j;
    j = 4 * i;
    cout << i << " " << j << endl;
}

int i = 2; //Global variable

void DoThat(int& i)
{
    j = 2 * i;
    i = j % 3;
    cout << i << " " << j << endl;
}

int main()
{
    cout << endl;
    cin >> i >> j;
    DoThat(i);
    DoThis(j,i);
    cout << i << " " << j << endl;
}
```

□ Activity 1 Copy, study, and trace the program in `scope6.cpp`. Then predict the output before running the program and comparing the output with your prediction. Try input values 5 and 7, then 9 and 8, then some of your own.

□ Activity 2 Copy `scope6.cpp` to `scope6a.cpp`, choose your own "bugs" to insert, and record both the changes you made and your brief description of the result below (including output where appropriate):

□ Activity 3

□ Activity 4

○ INSTRUCTOR CHECKPOINT 19.15 FOR EVALUATING PRIOR WORK

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 (page 3) 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
and_eq
bitand
bitor
compl
not
not_eq
or
or_eq
xor
xor_eq
```

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 see here and elsewhere when you look at C++ code, some of them much more frequently than others:

```
cin
cout
endl
hex
include
INT_MAX
INT_MIN
iomanip
iostream
main
oct
std
```

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 a "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 lowercase letters each form a contiguous group of characters.
- The group of capital letters comes before the group of lowercase letters.
- The number of code positions separating any given capital letter from its corresponding lowercase 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

Some 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. The first group is enclosed between lines of dashes, the second between lines of equal signs.

```
++   --             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
-------------------------------------------------------------
*    /    %         multiplication and division
+    -              addition and subtraction
>>   <<             input and output operators
<    <=   >   >=    relational
==   !=             more relational
&&                  logical and
||                  logical or
?:                  conditional
=============================================================
=    *=   /=   %=   assignment
+=   -=             more assignment
=============================================================
```

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 it is not that often that we really want to output the value of a conditional expression, but it makes the point.

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"

a. Name things well!

b. Be consistent!

c. Re-format continually! (This will help to *preserve* consistency.)

D.2 Spacing and Alignment

a. Put each program statement on a separate line, unless you can make a very good argument to explain why you didn't. (For example, you may be using a C++ *idom* of some kind.)

b. Use vertical spacing to enhance readability. For example, use one or more blank lines to separate logically distinct parts of a program.

c. Use horizontal spacing to enhance readability. For example, place a blank space on each side of an operator such as `<<` or `+`, unless there is a good reason for *not* doing so. (For example, you have a complicated expression on a long line of code, and omitting some spaces around some operators will emphasize the association between those operators and their operands, perhaps making it more readable than it would be with the extra spaces.)

d. Use an indentation level of four (4) spaces, and always indent in these cases:

  - 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.

e. 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).

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 lowercase letter, and in fact use all lowercase 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.

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`

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 Miscellaneous

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:

a. Use *named constants* whenever appropriate.

b. Until you become a more “advanced” C++ programmer, only use the increment and decrement operators in stand-alone statements.

Appendix E

Guidelines for Structured (Procedural) Program Development

This Appendix contains a summary of our recommended guidelines when you are applying the structured (i.e., procedural) approach to program development. It is worth pointing out that "C++" does not appear in the title. This is intentional, and a reflection of the fact that these guidelines are essentially language-independent.

a. Analyze the problem. Analyze it to death if necessary, but do *not* proceed until you *understand* it.

b. 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.

*Here you describe **what** but not **how**.*

c. Design and develop the algorithm(s) that will accomplish the task at hand.

Now for the how!

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:

The sooner you start coding, the longer it's going to take.

And don't you forget it!

d. 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.

e. 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.

f. 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(s), 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

Introduction to your programming environment

F.1 Objectives

- To locate and visit the physical facilities that you will be using.
- To understand what is meant by a *programming environment*, and to learn what the major components of your own programming environment are.
- To make sure you have the necessary authority to begin exploring your programming environment.

F.2 Overview

If you want to learn how to write *computer programs*, you must become familiar with the *programming environment* used on your computer. This means acquiring at least some knowledge of the *operating system*, an *editor*, the *language translator* for the *programming language* you will be using (a *compiler* for *C++*, in your case), and any local *utility programs* that may make your life easier, as well as the policies put in place by those in charge of your system. This Appendix will help you with that process.

Though there are many different programming environments and we cannot, therefore, supply all the specific details of the information *you* will need to know, we *can* direct your search for the required information by providing questions for which you will need the answers that apply to your actual situation, whatever it may be, and this is the approach that we shall take.

F.3 Questions needing local answers, with follow-up hands-on activities

The questions posed in the margins of this section will have to be answered, wherever and whatever your particular programming environment might be. Fill in the blank space of each cell with the answer(s), for your local environment, to the question(s) posed next to that cell. Even if there is locally available documentation or “on-line” help that tells you all you need to know, and possibly much more, it will be useful to sift through that material and find the answers to the specific questions posed here. There may, of course, be questions other than those given below that you will want to have answered as well, and you are reminded in the hands-on activities at the end of this section to make a list of these to ask your instructor.

Where are the facilities that you will be using located?	Answer
On which days and during which time periods are they available for *your* use?	Answer
With what pieces of equipment will you have actual physical contact—terminals, PCs or workstations, printers, and so on?	Answer
What kind of computer will you be using to do your C++ programming?	Answer
Will you need an account and a password to obtain access to your computer, or computing system? If so, how, where, and from whom do you obtain them?	Answer
What is the name of the operating system on the computer you will be using?	Answer

Answer

What is the name of the editor or other program you will be using to enter your own programs?

Answer

What are the names of any other useful *utility programs* that you will be using?

Answer

You will be programming in C++, but there are many versions, or “dialects” of this language. Which one will you be using and how does it compare with *Standard C++*?

Answer

Are there manuals or any other publicly available documentation that you might need from time to time and, if so, what are they and where are they located?

Answer

Is your computer connected to a network? If so, what kind of network?

Answer

Does your computer or network allow remote access? If so, are there any particular details of which you should be aware?

Answer

Where can you get further information on these and other related topics?

Just to keep track of your accomplishments as you work through this Manual, you may wish to place a check mark in the box that appears at the beginning of each activity as you complete that activity (or your instructor may require that you do so).

□ Activity 1 Visit the room(s) where you will be working when you start to use the computer, and locate as many of the components of your programming environment as you can find.

□ Activity 2 Read all of the posted notices that deal with the use of your facilities. Pay particular attention to the notices regarding printer use. Printers seem to be a major source of problems in programming environments where many users are printing to the same device. Remember to leave your printer in the same condition that you would like to find it in yourself.

□ Activity 3 Make a list below of any questions that will need further clarification from your instructor.

○ Instructor checkpoint 6.1 for evaluating prior work

Appendix G

Introduction to your operating system

G.1 Objectives

- To gain access to the computer and/or account that you will be using.
- To learn how to perform some basic operating system tasks.
- To understand what a *file* is, and to learn how files are named and managed in your operating system.
- To understand where the files associated with this Manual are located, and to learn how to view and/or copy those files.

G.2 List of associated files

- `cs_names.txt` contains some important names in computing science.
- `quotes.txt` contains some quotations from various sources.

The two *files* listed above are both *textfiles*, as indicated by the `.txt` file "extension". This means they contain information that is readable by humans when the files are displayed on a screen or printed. Both files should also be available in a *public repository* [1] of some kind on your local system. In fact, this public repository should contain all of the files associated with all of the Modules and Appendices in this Lab Manual, and you must learn the structure

[1] This *public repository* may be anything from a directory (to which you have read access) in your instructor's account, if you are working on a multi-user machine, to a World Wide Web location for which you will need a URL (Uniform Resource Locator) and a web browser to access the files. Your instructor will supply the necessary details so that you may read or copy these files.

of this public repository and how to access the files it contains. You need not do this all at once, of course, but now is a good time to start. Details will be provided by your instructor. Later in this Appendix you will get your own copies of the above files from the public repository. The basic idea is that you should always know how to find and copy each publicly available file referred to in this Lab Manual.

G.3 Overview

Every computer needs an *operating system*. This is a program that runs on the computer and, among other things, manages all resources used by that computer, and handles all interaction between the computer and the "outside world". Thus, it is the operating system that allows other programs to run in response to requests from users, sends print jobs to the printer (or to the appropriate printer if there are several), loads into memory the contents of files stored on disk, and so on. From Appendix F you know what your operating system is. In this Appendix you begin to learn how to use it.

G.4 Questions needing local answers, with follow-up hands-on activities

Once again, the questions posed in the following subsections will have to be answered, wherever and whatever your particular programming environment might be. Enter into the blank space in each cell the answer(s), for your local environment, to the question(s) adjacent to that cell.

Following the question-and-answer cells in each subsection is a sequence of *hands-on activities* that you must complete in order to consolidate your knowledge of the items covered in that subsection. Completing each such group of activities should, of course, be regarded as the minimal effort to be expended in the given context, and any additional activities that suggest themselves should be undertaken. Your goal is to be completely comfortable performing the indicated (and similar) tasks.

G.4.1 Keyboard familiarization and (if applicable) logging in to your computer

What are the main groups of keys on your keyboard, and what is the purpose of each group?

Answer

Answer

Are there key combinations that are important, and if so, which ones and what do they do?

Answer

If you need to log in to your computer, how do you accomplish this?

Answer

How do you know that you have successfully logged in?

Answer

Once you have successfully logged in, how do you log out?

Answer

How do you know that you have successfully logged out?

Answer

What are the dangers of *not* logging out?

□ Activity 1 If you have an account and a corresponding password, begin by logging in to your account. Be sure to complete any procedures and supply all information required by the login process.

□ Activity 2 If you have successfully logged in, log out and then check to make sure you have in fact logged out.

G.4.2 Changing your password (if applicable)

Answer

If you have a password to give you access to your computer account, how do you change this password?

Answer

What form should your new password take?

Why should you change your password, and when (or how often)?

Answer

What should you do if you forget your password?

Answer

□ Activity 1 If you have an account and corresponding password, log in and change your password.

□ Activity 2 When you have changed your password, log out and then log back in to test the new password.

□ Activity 3 Mentally record your new password, but do not write it down. This means that it should be easy to remember, but hard for anyone else to guess.

G.4.3 Communicating with your operating system

If you have changed a password, you have seen how to perform at least one task in your operating system. How, in general, do you tell your operating system to do something?

Answer

If your operating system provides Windows capability, can you list some tasks which you can perform using it?

Answer

If your operating system provides a "command line", can you list some tasks you can perform using it?

Answer

Record here some additional useful actions or commands in your operating system (from local documentation or from your instructor).

Answer

Before beginning the following, or any subsequent similar sequence of hands-on activities, you must of course have already gained access to your computer workspace in the usual way, whether this is by logging in to your account with a username and password, or by some other procedure.

□ Activity 1 Try each of the actions or commands that you listed above.

□ Activity 2 If there is a shortcut or alternate method to perform (or repeat) any of the commands listed above, then try the shortcut method(s) as well.

G.4.4 Files and file naming conventions, pathnames and the "public repository"

Answer

What exactly is a *file*, and what are files used for?

Answer

What are some different kinds of files?

Answer

What naming conventions for files does your operating system use?

Answer

Where are the sample files on your local system? (This is where you should find, among other things, the files for this Lab Manual.) It will be helpful to draw a diagram of the structure in the accompanying space to help fix it in your mind.

Answer

What does a *full pathname* to a subdirectory (and to a file) look like on your system? (Be sure you know how this concept of a *pathname* relates to the concepts of directory and subdirectory on your system.)

What is the location (full pathname, URL, or whatever) to the public repository on your system?

Answer

When responding to the next question, think of your *current directory*, for the moment at least, as the storage space to which you are automatically given access when you turn the computer on and/or log in to your account.

How do you display a list of all files in your own current directory (i.e., your own account)?

Answer

How do you display a list of all files in some other location (the public repository, say)?

Answer

□ Activity 1 Try each of the two actions or commands that you listed above. Display a list of all files in your current directory (there may be none at all at this time).

□ Activity 2 Take a look at the list of all files associated with this Manual, wherever they may be located. Make a note of the location of each of the files `cs_names.txt` and `quotes.txt`, which are associated with this Appendix.

G.4.5 Displaying and printing a file (of text, of course)

Which kinds of files are “safe” to display on your screen, or send to a printer, and which are not?

Answer

How do you display a file on your screen? **First you make sure it is a textfile, and then ...**

Answer

What do you do if you want to see all of a file that is too large to be displayed all at once on your screen?

Answer

Does the location of a file affect the way that you display it? If so, how?

Answer

How do you print a file? **First you make sure it is a textfile, and then ...**

Answer

How do you retrieve a printed file (often called a *hard copy*) from the printer?

Answer

What do you do if you have sent a file to the printer but for some reason want to delete the file from the print queue (i.e. prevent it from printing at all, or discontinue the printing if it has already started)?

Answer

□ Activity 1 Display on your screen the contents of the file `cs_names.txt`. You should see what is shown between the heavy black lines below. Lines like these are occasionally used throughout this manual to mark the beginning and end of file contents, except for C++ program files, each of which appears in its own separate subsection.

```
Filename: csnames.txt

George Boole   Charles Babbage   Augusta Ada Byron   Herman Hollerith
Alan Turing   John V. Atanasoff   Howard Aiken   Grace M. Hopper
John Mauchley   Presper Eckert   John von Neumann   Stanislaw Ulam
Maurice V. Wilkes   John Bardeen   Walter Brattain   William Shockley
John Backus   Donald L. Shell   John Kemeny   Thomas Kurtz
Corrado Bohm   Guiseppe Jacopini   Edsger W. Dijkstra   Harlan B. Mills
Donald E. Knuth   Ted Hoff   Stan Mazer   Robert Noyce   Federico Faggin
Ted Codd   Paul Allen   Bill Gates   Stephen Wozniak   Stephen Jobs
Dan Bricklin   Dan Fylstra   Robert Barnaby   William L. Sydnes
Mitchell Kapor   Tom Button   Alan Cooper   Tim Berners-Lee
Marc Andreessen   Michael Cowpland   Bjarne Stroustrup   Ken Thompson
Dennis Ritchie   Alan Kay   Blaise Pascal   Niklaus Wirth
COBOL   LISP   Smalltalk   APL   FORTRAN   Algol   Simula   PROLOG
Intel   Apple   IBM   DEC   Atari   Microsoft   Corel   Xerox
```

□ Activity 2 Send a copy of the file `cs_names.txt` to your printer and then retrieve the printout, or *hard copy*, of the file from the printer. Be careful to follow any and all relevant posted instructions when removing your printout from the printer.

○ Instructor checkpoint G.1 for evaluating prior work

□ Activity 3 Display on your screen the contents of the file `quotes.txt` in the public repository. The file contents you see should be those shown below. If you cannot see all of the file on your screen, do whatever is necessary to permit you to view the rest of the file. (You do *not* have to see all of the file at once.)

```
Filename: quotes.txt

- Confidence is the feeling you have before you understand the situation.
- Health is merely the slowest possible rate at which one can die.
- People will buy anything that's one to a customer.
- Just because you're paranoid doesn't mean they AREN'T after you.
- Good leaders being scarce, following yourself is allowed.
- If you can survive death, you can probably survive anything.
- You don't have to drain the swamp to deal with the alligators.
- It's easier to seek forgiveness than permission.
- Experience is what causes a person to make new mistakes instead of old ones.
- Experience is that wonderful thing that helps you recognize your
  mistakes when you make them again.
- Heuristics are bug-ridden by definition.  If they didn't have bugs,
  then they'd be algorithms.

Grabel's Law: 2 is not equal to 3 -- not even for large values of 2.

As soon as we started programming, we found to our surprise that it
wasn't as easy to get programs right as we had thought.  Debugging had
to be discovered.  I can remember the exact instant when I realized that
a large part of my life from then on was going to be spent in finding
mistakes in my own programs. (Maurice Wilkes discovers debugging, 1949)

Anyone can hold the helm when the sea is calm. (Publilius Syrus)

Be careful of reading health books; you might die of a misprint. (Mark Twain)

Confession is good for the soul only in the sense that a tweed coat is
good for dandruff. (Peter de Vries)

Democracy is the recurrent suspicion that more than half of the people
are right more than half of the time. (E. B. White)

I generally avoid temptation unless I can't resist it. (Mae West)

No one can make you feel inferior without your consent. (Eleanor Roosevelt)

I just need enough to tide me over until I need more. (Bill Hoest)

I have the simplest tastes.  I am always satisfied with the best. (Oscar Wilde)

If I had any humility I would be perfect. (Ted Turner)

If all the world's a stage, I want to operate the trap door. (Paul Beatty)

If all the world's economists were laid end to end, we wouldn't reach a
conclusion. (William Baumol)

Injustice anywhere is a threat to justice everywhere. (Martin Luther King, Jr.)

Innovation is hard to schedule. (Dan Fylstra)

If you live in a country run by committee, be on the committee. (Graham Summer)

I hate quotations. (Ralph Waldo Emerson)
```

□ Activity 4 Send a copy of the file `quotes.txt` to your printer and then retrieve the hard copy of the file from the printer. Once again, be careful to follow instructions when removing your printout from the printer.

G.4.6 Copying, renaming, appending and deleting files

Answer

How do you copy a file from some other location to your own directory (with the same name or a different name)?

Answer

How do you make a copy of a file in your own directory?

Answer

How do you rename a file?

Answer

How do you append two textfiles? That is, how do you add a copy of one textfile to the end of a second textfile? (This implies that the second textfile is altered but the first one is not.)

Answer

How do you create a brand new textfile that consists of a copy of one textfile appended to the end of a second textfile? (This implies that both of the original files are unchanged.)

Answer

How do you delete a file?

Answer

Are there any kinds of files that you will want to delete regularly, in order to "clean up" your account?

□ Activity 1 Make a copy, for your own directory, of the file `cs_names.txt`, and give it the same name.

□ Activity 2 Make another copy, for your own directory, of the file `cs_names.txt`, and give it the name `cs_stuff.txt`.

□ Activity 3 Make a copy, for your own directory, of the file `quotes.txt`, and give it the name `quips.txt`.

□ Activity 4 Make a copy of the file `quips.txt`, which is in your own directory, and call it `quotes.doc`.

□ Activity 5 Rename the file `quotes.doc` to `quotes.txt`.

□ Activity 6 Create a new file called `both1.txt` by appending a copy of `quotes.txt` to the end of a copy of `cs_names.txt`.

□ Activity 7 Create a second new file by appending a copy of `cs_names.txt` to the end of a copy of `quotes.txt`. Call the new file `both2.txt`.

□ Activity 8 Append a copy of `quotes.txt` to the end of cs_stuff.txt and then rename cs_stuff.txt to `names_and_quotes.txt`.

□ Activity 9 Make a list of the files you should now have in your directory, as well as their contents, and then check to make sure that you do in fact have those files and their contents are what you expected.

○ Instructor checkpoint G.2 for evaluating prior work

□ Activity 10 Delete, in turn, each of the files you have created in the previous sequence of activities, checking after each deletion that your directory contains the files that you think it should.

G.4.7 Creating a short textfile without an editor

Normally you would use a program called an editor (discussed in detail in a later Appendix) to create a textfile. However, some operating systems provide a way to enter text into a file on disk directly from the operating system itself. If your operating system does not permit this, you may simply ignore this subsection. In any case, such a method for textfile creation should only be used for creating very short files "on the fly".

Does your operating system permit direct creation of a textfile without an editor? If so, how is it done?

Answer

□ Activity 1 Create a short textfile containing your name and address. Call it `my_info.txt`. Make sure that each line is indented four spaces and that all lines are aligned on the left.

□ Activity 2 Print a hard copy of `my_info.txt`.

○ Instructor checkpoint G.3 for evaluating prior work

G.4.8 Other useful commands or procedures available in your operating system

There are, of course, many other useful commands or procedures for performing various tasks in your operating system, and your instructor may or may not give you some additional ones now. In any case, you should come back here and make a note of each new one that you encounter as you proceed.

Answer

Are there some additional commands or procedures that you could list right now?

G.4.9 Getting help on your operating system

How can you get help with any questions you might have on your operating system?

Answer

□ Activity 1 Check the help sources for your operating system, and add at least two commands or procedures to your list (in the preceding subsection) of “other useful operating system commands or procedures”.

□ Activity 2 Experiment with each of the new commands or procedures you discovered and listed in the previous activity until you are completely comfortable with its use.

Appendix H

Useful local utilities

H.1 Objectives

- To begin the process of discovering what useful local *utility programs* are available to you in your programming environment.
- To find out what each of these utilities will do for you, and how to use it.

H.2 Associated files

Any particular utility available to you may or may not have one or more files associated with it, or it may simply be a command that you give to the operating system. Since each programming environment is different, you will have to find out from your instructor or from local documentation exactly what is available.

H.3 Overview

Programmers like to make life easy for themselves, and for other programmers. This is why in every programming environment you will find various local *utility programs* that automate certain tedious tasks, and often depend critically on the local operating system and/or on other parts of the local programming environment. You should always look for such utilities when you are beginning work in a new environment. A little experience will give you a better idea of what to look for, and this Appendix is designed to get you involved with whatever may be available in your own particular local programming environment. Examples of such utilities might include a facility for the electronic submission of programming assignments, a tool for the "capture" of certain kinds of screen activity, and so on.

H.4 Questions needing local answers, with follow-up hands-on activities

In this section you must fill in each of the following templates (or as many as you need) to record the necessary information for each of the utilities you will be using. Following each template are several check-boxes, opposite each of which you can enter a description of a hands-on activity that you have made up (or perhaps transcribe one provided by your instructor) for helping you become familiar with the given utility.

We should say a word or two about the cell in the template that asks the question, "Where is this utility found, or where does it have to be placed?" This question may be interpreted in a couple of ways. If a given utility is available simply by entering a command at the operating system command line prompt, then this cell can simply be ignored when completing the template. On the other hand it may be that another utility requires you to know where one or more files are located, where you yourself must place one or more files, or both. If so, then that is the information that should be placed in this cell.

Some typical kinds of utilities that may or may not be available in your programming environment are:

- An *e-mail program* for communicating with other class members and/or your instructor(s)
- An *electronic passin utility* for submitting your homework electronically
- A *file transfer utility* for *uploading/downloading* files from your campus computer system to your home or laptop computer, if you have one
- A *compression utility* for making your programs "smaller", so that they take up less storage space in your account
- An *on-line database* from which you can access your current marks and other status information in the course

Remember that this Appendix just *begins* the process of identifying useful local utilities. You will probably want to return to this Appendix from time to time and fill out another template, and you should in fact do so whenever you encounter another useful utility program during the course. If you are really lucky, your collection of useful utilities will be so long that this Appendix will not accommodate them all!

What is the name of the first utility?
Where is this utility found, or where does it have to be placed?
What does this utility do for you?
How do you use this utility?
Are there any special features or instructions that you should know about?
For further information:

Enter hands-on activities here:

□ Activity 1

□ Activity 2

□ Activity 3

Additional Notes or Comments:

What is the name of the next utility?
Where is this utility found, or where does it have to be placed?
What does this utility do for you?
How do you use this utility?
Are there any special features or instructions that you should know about?
For further information:

Enter hands-on activities here:

□ Activity 1

□ Activity 2

□ Activity 3

Additional Notes or Comments:

What is the name of the next utility?
Where is this utility found, or where does it have to be placed?
What does this utility do for you?
How do you use this utility?
Are there any special features or instructions that you should know about?
For further information:

Enter hands-on activities here:

□ Activity 1

□ Activity 2

□ Activity 3

Additional Notes or Comments:

What is the name of the next utility?
Where is this utility found, or where does it have to be placed?
What does this utility do for you?
How do you use this utility?
Are there any special features or instructions that you should know about?
For further information:

Enter hands-on activities here:

□ Activity 1

□ Activity 2

□ Activity 3

Additional Notes or Comments:

Appendix I

Introduction to your editor

I.1 Objectives

- To learn the difference between a *buffer in memory* and a *file on disk*.
- To learn how to start up your editor and how to shut it down, with and without saving on disk the work done since the last "save".
- To learn how to enter text into a buffer in memory.
- To learn how to move the cursor from one part of a buffer to another.
- To learn how to move and copy text from one part of a buffer to another.
- To learn how to find or "find and replace" particular words or phrases in a buffer.
- To learn a number of other useful editing commands and procedures.

I.2 List of associated files

- `singers.txt` contains a list of names of some performing artists.
- `pascal.txt` contains notes on the man and the language named after him.

I.3 Overview

An *editor* is itself a program, and a very important tool for any programmer. It is the program that allows a programmer to enter the *source code* (instructions for the computer) for his or her programs into the computer. There are many different editors available, and you must become familiar, at the earliest opportunity, with the one you will be using.

By the way, do not confuse an editor with a *word processor*. They are very different things. A word processor (such as *Microsoft Word*, or *Wordperfect*) can usually be used as an editor, but it is usually not wise to do so.

In this Appendix you will learn the basic commands you need to enter, and later modify, the programs you will be writing. Most editors have many features, and you should try to keep learning more and more new ways to do things with your editor as time goes on. Doing so will enhance your productivity, and allow you to develop your programs in much less time than would otherwise be the case.

I.4 Questions needing local answers, with follow-up hands-on activities

Here we go again. Once more the questions posed next to the cells in the following subsections will have to be answered, wherever and whatever your particular programming environment might be. Enter into each cell the answer(s), for your local environment, to the question(s) adjacent to that cell.

In some programming environments the editor is a stand-alone program, while in others it is just one part of a more complex program (usually called an *integrated development environment*, or *IDE* for short). If you have such a system, in what follows whenever we refer to the editor we mean "the editor part of your IDE".

I.4.1 Buffers in memory, files on disk, starting and stopping your editor, and text insertion

What is the difference between a *file on disk* and a *buffer in memory*?

Answer

Do you have a stand-alone editor, or is your editor part of an *IDE*?

Answer

How do you use your editor to edit an already-existing file?

Answer

How do you use your editor to create a brand new file?

Answer

Answer

How do you know your editor has started successfully?

Answer

How do you leave the editor and at the same time save the work you have been doing?

Answer

How do you leave the editor without saving the work you have done?

Answer

Does your editor have different “modes”, such as full-screen editing mode, line-editing mode, command mode or text-entry mode? If so, how do you change from one mode to another?

Answer

How do you insert text when you are editing a file?

Answer

Does your editor distinguish between “insert mode” and “overstrike mode” during text insertion, and, if so, how?

Answer

How do you insert *line feeds*, *form feeds* (*page breaks*), and other special characters into your textfiles? And why would you want to do this?

□ Activity 1 Use your editor to create a new file called `heroes1.txt` containing a list of your four favorite musical artists or groups. Put each one on a separate line and indent each line four spaces.

□ Activity 2 Exit from the editor, saving the file, and then check to make sure the file is in your current directory.

□ Activity 3 Load the file `heroes1.txt` into your editor and then enter another artist or group so that you now have a total of five artists or groups. Put the new one on a separate line and indent as before.

□ Activity 4 Exit from the editor, saving the revised file, and then check to make sure you have what you thought you should have in the location where you thought it would be.

□ Activity 5 Load `heroes1.txt` into your editor again and enter yet another artist or group, but this time leave the editor without saving, and again check the result.

I.4.2 Navigation (moving the cursor from one part of the buffer to another)

How do you move the cursor forward and backward by a character, a word or a line at a time?

Answer

How do you move the cursor greater distances (to the beginning or end of a line, or several words or lines at a time, for example)?

Answer

How do you "scroll through" a buffer, both forwards and backwards?

Answer

How do you move to a particular location in a file? To the end or beginning of the file, or to a particular line, for example?

Answer

□ Activity 1 Make a copy of the file `singers.txt`, and display it, either in your editor, or using any other method of which you are now aware. You should see what lies between the heavy lines below.

```
1  Filename: singers.txt
2
3                 4.         Jennings, Waylon
4
5      2.   Presley,    Elvis
6
7
8           1.  The Oak Ridge Boys
9
10
11
12       5. Jones, Tom
13
14                 3. Gayle, Crystal
```

□ Activity 2 Load your copy of `singers.txt` into your editor and experiment with cursor movement by moving backward and forward a character, a word, and a line at a time; by moving several words or lines at a time; by moving to a particular line, say line 7; and by moving to the beginning and end of the file. Practice these and other cursor movements until you are comfortable with these minimal "navigational" procedures.

○ Instructor checkpoint I.1 for evaluating prior work

I.4.3 Deleting text from the buffer

Answer

How do you delete a character, a word, or a line from the buffer?

Answer

How do you delete larger amounts of text (several characters, words or lines, for example, or a larger contiguous "block" of characters)?

□ Activity 1 Make a copy of the file `pascal.txt` and display it in your editor. You should see the file contents shown below.

```
Filename: pascal.txt

Pascal, The Man
---------------
The programming language Pascal was named after Blaise Pascal.  He was a
French mathematician, engineer, scientist and religious philosopher.
Pascal was born in 1623 in Auvergne in central France, and died in 1662.
At the age of 18, he designed a mechanical computing machine capable of
performing simple arithmetic calculations. The machine was "just" a type
of adding machine, and not the kind of programmable device that would be
called a computer in the sense of today's meaning of the term.
Nonetheless, the machine attracted much attention and became a prototype
for a number of later computing devices.  Pascal had a number of models
constructed, and attempted to sell his invention.  Unfortunately, its
high price doomed the machine to financial failure.

His calculating machine was only one of Pascal's many contributions to
science and engineering.  He designed the first public transportation
system for the city of Paris, which used horse-drawn carriages.  He also
made important contributions to many branches of mathematics, including
geometry, probability theory and hydrodynamics.

Pascal was also a prominent figure in the religious philosophy of his
time. His last and most enduring religious work is his Pensees.


Pascal, The Programming Language
--------------------------------
The programming language Pascal was introduced in 1971 by the Swiss
computer scientist Niklaus Wirth.  It was originally intended as a
general-purpose, high-level language for teaching the concepts of
structured programming and top-down design.  Pascal's simplicity,
elegance, and embodiment of structured programming principles have made
it quite popular with a wide audience.

Among all the computer languages widely used in recent years, Pascal is
probably the best for demonstrating what structured programming is all
about.  It is simple, straightforward, and easy to learn, and it imposes
rules that encourage good programming habits.  In addition, Pascal is a
versatile and powerful language that helps users avoid programming
errors.  Consequently, large, complex, relatively error-free programs
are easier to write in Pascal than in many other languages.

Designed to be a teaching language, Pascal was not intended to be
employed outside of the academic world, in which it became a very
influential language. However, despite some problems, it was also used
quite widely in the "real world" for a number of years.
```

□ Activity 2 Load your copy of `pascal.txt` into your editor.

□ Activity 3 Practice deleting characters, words, lines and larger blocks of text until all text has been removed from the buffer.

□ Activity 4 Quit the editor without saving so that the original file on disk remains intact.

□ Activity 5 Repeat the previous three steps until you are comfortable with the delete commands.

I.4.4 Moving or copying text by "cutting and pasting"

Answer

How do you move, or copy, a character, a word, or a line by "cutting and pasting"?

Answer

How do you cut and paste larger amounts of text?

□ Activity 1 Make a copy of `pascal.txt` and call it `pascal1.txt`.

□ Activity 2 Load `pascal1.txt` into your editor.

□ Activity 3 Practice moving and copying characters, words, lines and larger blocks of text until you are comfortable using all relevant procedures.

□ Activity 4 Quit the editor without saving your changes.

□ Activity 5 Load `pascal1.txt` into your editor again.

□ Activity 6 In line 24, change the phrase "last and most enduring" to "most enduring and last".

□ Activity 7 Exchange each line of dashes with the line that precedes it.

□ Activity 8 Exchange the last two paragraphs in the file, retaining a blank line between them.

□ Activity 9 Finally, exchange the order of the two major sections of the file (i.e., the section entitled `Pascal, The Programming Language` must precede the section entitled `Pascal, The Man`).

□ Activity 10 Exit from the editor, and this time save your changes.

○ INSTRUCTOR CHECKPOINT I.2 FOR EVALUATING PRIOR WORK

I.4.5 Finding specific text, and "find and replace"

How do you find a particular word or phrase in a buffer, when searching forward and backward?

Answer

Are searches case-sensitive or case-insensitive by default, and how do you change between case-sensitive and case-insensitive searches?

Answer

How do you "find and replace" one, several, or all occurrences of a particular word or phrase?

Answer

How do you find, or find and replace, a "special character", such as a *tab* character, for example?

Answer

□ Activity 1 Load the file `pascal.txt` into your editor and make sure the cursor is at the beginning of the buffer.

□ Activity 2 Search forward for each of the following comma-separated words, phrases, or other text items in turn:

`the, The, 1971, forward, is a, years, language Pascal, today's`

□ Activity 3 Place your cursor at the end of the buffer and then search backward for each of the following comma separated words, phrases, or other text items in turn:

`Wednesday, large, in the, 62, ---, real, Pascal, PASCAL`

□ Activity 4 Try as many other searching examples, of your own choosing, as it takes to make you comfortable with the searching process (forward and backward, case-sensitive and case-insensitive, as appropriate), then quit the editor without saving any changes (there should not be any changes to save if all you have done is search for various things).

□ Activity 5 Make another copy of `pascal.txt` and call it `pascal2.txt`.

□ Activity 6 Load the file `pascal2.txt` into your editor.

□ Activity 7 Find the year 1971 and replace it with 1791.

□ Activity 8 Replace each occurrence of the word "religious" in the buffer with the word "occult".

□ Activity 9 Replace each occurrence of "the" in the buffer with "these".

Ask yourself the following question, and be sure you know the answer: What does the previous activity tell me about the care I should take when doing a "global" search-and-replace, i.e., a replacement of all instances of a certain string with some other string. (By a string we simply mean a given sequence of characters considered as a single entity.)

□ Activity 10 Finally, exit from the editor, saving your changes to `pascal2.txt`.

○ Instructor checkpoint I.3 for evaluating prior work

I.4.6 Avoiding disasters

You should, of course, *always remember to save your work frequently* when you are entering text with an editor, since if something goes wrong you may or may not be able to recover all or even part of the work you have done since the last time you saved your entered text to a disk file.

How do you save your current work without leaving the editor?

Answer

Can you recover your work if some disaster (like a computer crash) occurs before you have saved it? If so, how?

Answer

What should you do if you try to leave the editor and save your changes and you get a message saying you can't because you are "out of disk space"?

Answer

□ Activity 1 Try the following experiment: Start your editor with a brand new file called `numbers.txt`. Then enter on line 1 the word "one", on line 2 the word "two" and on line 3, the word "three", writing the contents of the file to disk after entering the contents of each line. Now on lines 4 to 10 enter the words "four" to "ten", without writing the contents of the buffer to disk, and then quit without saving any changes to the file (which is like simulating a computer crash at this point).

□ Activity 2 Now ask yourself (and this time *record* your answer below) the following question: What do I now have in the file `numbers.txt`, and what does this tell me about saving my work regularly to disk when I am editing a file?

○ Instructor checkpoint I.4 for evaluating prior work

I.4.7 Additional miscellaneous editor commands

Answer

How do you repeat an editing command you have just given?

Answer

How do you change the case of a letter, a word, or a phrase?

Answer

How do you “refresh” your screen if it gets “messed up” in some way (by an incoming mail message if you are on a network, for example)?

Answer

How do you write out the entire buffer you are editing to a file with a different name that you choose?

Answer

How do you write out just a part of the buffer you are editing to a separate file on disk?

Answer

How do you indent one or several lines of a buffer by “one level of indentation”?

Answer

How do you “unindent” one or several lines of a buffer by one level of indentation?

□ Activity 1 Be sure to test each of the above commands, after you have entered it in the cell, until you are thoroughly familiar with its operation.

I.4.8 Getting help on your editor

You should plan to study whatever local "guide to the editor" may be available frequently from now on to increase your command repertoire and thereby make yourself more productive when working on your C++ programs.

How can you get help with any questions you might have on your editor?

Answer

□ Activity 1 Go to your source of help for your editor and find at least two additional useful commands that you have not yet encountered. Record these commands in the space below, and continue to add to the list as time goes on.

Record the date as you add each new editor command below, and be prepared to show the list to your instructor from time to time, should he or she ask to see it. This list will be evidence for your instructor, and reassurance for you, that you have indeed continued to add to your knowledge of your editor and its commands.

Appendix J

Customizing your programming environment with operational shortcuts and file organization

J.1 Objectives

- To learn about the possibilities for customizing your operating system and your editor to fit your personal work habits, or to conform to the programming environment requirements of your class or lab group.
- To organize, in appropriate *subdirectories*, the files you have already copied or created, and to create a *subdirectory hierarchy* for keeping all files organized as your work progresses.
- To begin the ongoing process of creating and/or recording useful shortcuts to help you increase your program development productivity.

J.2 Associated files

The files associated with this Appendix fall into two categories:

- first, the particular customization files specific to your system, the details of which you will record in the appropriate places as you proceed through this Appendix
- second, all the files you have copied from the public repository, together with the new files you have created while working with those copied files, and which now need to be "organized"

J.3 Overview

As you continue to work in your programming environment, you will find yourself repeating certain commands, or actions, or procedures, over and over again. Whenever you notice this phenomenon, you should ask yourself: Is there a better way? Can I create a shortcut of some kind?

Customize your working environment if you can.

You will often find that certain programs can be customized to provide such shortcuts, and, more generally, to better fit your own particular working habits. Sometimes this is accomplished by choosing certain options within the program itself, and other times it is done by entering certain options in a separate file that is read by the program when it starts. For example, it is often possible to customize either or both of the operating system and the editor to suit your tastes.

Organize your files in well-named subdirectories.

Another thing you will notice before too long is that, as you do the hands on activities, files will start to accumulate in your personal workspace on the computer, and it will make your life much simpler if you organize these files into appropriately named *subdirectories*. Your instructor may require that your files be organized in a particular way, or you may have the freedom to choose your own organizational scheme, but the importance to your peace of mind of collecting related files and placing them together in a subdirectory whose name reflects the nature of the collection cannot be overemphasized.

J.4 Questions needing local answers, with follow-up hands-on activities

Some of the following subsections contain explicit hands-on activities, others just contain a couple of Activity lablels, each followed by some blank space in which you can either write your own description of a (platform-dependent) hands-on activity for working with the topic of that subsection, or perhaps transcribe one provided by your instructor.

J.4.1 Customizing your operating system (optional)

Does your operating system have a *startup command file* or a *configuration file* of some kind which permits you to customize your operating system for your particular work habits? If not, then you should simply ignore this subsection. If so, what is its name and where must it reside?
Are the contents of this file “activated” automatically when you start up your computer (or log in to your account), or must you do something extra to activate them? If so, what?
What are some of the things provided by this file, and how do you use them?
For further information:

Enter hands-on activities here:

□ Activity 1

□ Activity 2

J.4.2 Customizing your editor (optional)

Does your editor have a startup command file or configuration file of some kind which permits you to customize your editor for your particular work habits? If not, then you should simply ignore this subsection. If so, what is its name and where must it reside?
Are the contents of this file "activated" automatically when you start up your editor, or must you do something extra to activate them? If so, what?
What are some of the things provided by this file, and how do you use them?
For further information:

Enter hands-on activities here:

□ Activity 1

□ Activity 2

J.4.3 Creating and using subdirectories on your system

Answer

How do you create a *subdirectory* (also, on some systems, called a *folder*) on your system?

Answer

How do you delete a subdirectory on your system?

Answer

How do you make a particular subdirectory your *current working directory*?

Answer

How do you move a file from one location (subdirectory) to another?

Answer

How do you copy a file from one location (subdirectory) to another?

Answer

How do you move or copy several files at once from one location (subdirectory) to another?

□ Activity 1 Create a subdirectory called `test` in your current working directory.

□ Activity 2 Move your copy of `hello.cpp` into the `test` subdirectory.

□ Activity 3 Copy all of your files with a `.txt` extension into the test subdirectory.

□ Activity 4 Make the test subdirectory your current working directory and check to see that its contents are consistent with the preceding activities.

○ INSTRUCTOR CHECKPOINT J.1 FOR EVALUATING PRIOR WORK

□ Activity 5 Move `hello.cpp` back to its original location.

□ Activity 6 Delete the subdirectory `test` and its remaining contents.

J.4.4 Organizing your files

Your instructor may require that you use a particular scheme or structure for organizing and storing your files as you work through the various Modules. Even if no specific structure is required in your case, you would be well advised to design such a structure for your own use.

Sketch in the accompanying space a hierarchical (i.e., "inverted-tree-like") diagram of the subdirectory structure you will be using to organize and store your files.

Answer

□ Activity 1 Create, in your account or in your workspace, the subdirectory hierarchy that you have sketched above.

□ Activity 2 Move each of the files that you have copied or created, while completing previous hands-on activities, into the appropriate subdirectory created above, so that the organization of your directory or account is "up to date".

○ INSTRUCTOR CHECKPOINT J.2 FOR EVALUATING PRIOR WORK

J.4.5 Getting more information on customization

Answer

Where can you find further information on the topics of this Appendix?

J.4.6 Other useful shortcuts

Answer

Can you find any additional useful shortcuts that you can use in your programming environment? If so, make a note of them now, and be sure to come back and add to the list as and when you discover new ones.

□ Activity 1 Test each of the shortcuts you record above until you are thoroughly familiar with how it works.

Index