Preface

Intended readership

This book is primarily intended to be a text for the programming component in an introductory two semester computer science course (some materials are a little advanced and might postponed to later semesters). This intent shows in terms of references to "students", "assignments" and "later studies in computing". However, the book should be equally suited to an individual who wants to learn how to program their own personal computer. If used as a text for an introductory computer science course, the contents of this book would normally be supplemented by a variety of other materials covering everything from "ethics" to "Turing machines".

Assumed knowledge

It is assumed that you are "computer literate". In your school or college, or at work, you will have used word processing packages, and possibly other packages like spreadsheets and data bases. Although most schools do provide at a limited introduction to programming (usually in Pascal or possibly a dialect of Basic), this text does not rely on such past experience.

Programming environment

It is expected that you will use one of the modern "Integrated Development Environments" (IDE) on a personal computer. Examples of such environments include the Borland environment for Intel PCs and the Symantec environment for Macintosh/PowerPC machines. These environments are supplied with comprehensive reference manuals . These manuals give details like how to start programs; they list and explain menu options; they include illustrations showing window layouts with details of all controls. These kinds of reference material are not duplicated in this book.

Content

Part I Introduction

The first section of the book contains some introductory material on computer hardware, operating systems and programming languages. Most of the material should be largely familiar to anyone who has done a "computer literacy" subject at school, or who reads popular computing magazines. The aim of this section is to provide background context knowledge so that you will understand what is happening when you do something like "compile a program" or "link with a library". Even if you feel confident that you know this material, you should skim through this section as it does introduce some terminology employed later in the text.

Hardware

Chapter 1 provides a brief overview of computer hardware giving a fairly standard presentation of computer architecture along with a few details of how data are input and output when being processed by a program.

Machine and assembly languages

Chapter 2 looks at programs in the machine, describing the bit patterns that represent instructions and explaining how these bit patterns get into memory. The chapter includes short sections on assembly language programming and system's software such as assemblers and loaders. The basic structural components of assembly language programs – sequences of instructions, conditional branches to build loops and make selections, and subroutine call and return – are illustrated.

Operating systems

Operating systems are briefly reviewed in Chapter 3. The treatment is partly historical, explaining the development of operating systems. Terms such as "time sharing", "virtual memory", and "multiprogramming" are introduced along with some details of the systems where these ideas first evolved and the reasons why these features were added to operating systems.

High level languages

Chapter 4 has again a somewhat historical viewpoint; it presents the evolution of high level programming languages. Most commonly used programming languages are closely related relying on the same "imperative procedural" programming constructs.

The "imperative procedural languages" share the same view of programs – really the same view as in assembly language. Programs calculate results from data values given as inputs; the input data, intermediate values, and final result values are represented by *variables*. *Expressions* describe how to combine data values; the most common expressions simply define arithmetic manipulations. Variables are updated by assignment of the values calculated by executing instructions implementing the expressions. In addition to *assignment statements*, these languages have *iterative statements* (for loop constructs), *selection statements*, and subroutine call mechanisms. Although there are major differences in style and appearance, diverse languages like FORTRAN, COBOL, and C++ have underlying similarities resulting from their common imperative procedural model. (There are languages, like Prolog, that do embody quite different models.)

Algol family of languages

Languages like Pascal, Modula2, C, C++, Simula, Eiffel really all belong to the same linguistic family; their roots are in a language called Algol-60. They have diverged to varying extents and, to some degree, have each become specialized for particular kinds of application. Nevertheless, they remain closely in structure.

Recursive functions

In addition to touching on the development of programming languages, Chapter 4 also provides a simple introductory example illustrating the idea of recursion.

C++ development environment

Chapter 5 provides a general picture of the kind of "Integrated Development Environment" that you will be using to create and run your programs. The treatment is non-specific; all the environments include:

- an editor (for entry of program code and of data files),
- a compiler (for translating code),
- function libraries (compiled code for standard useful components like code to calculate the square root of a number, or code to translate between a string of printed digits and a number in the form that the computer can manipulate),

- a linker (puts together a complete program by combining your code with components from the libraries),
- a "make system" (this organizes the compilation and linking processes for you; this make system may be fully automatic and largely hidden in the "program" or "project" management facilities provided by the IDE),

and

- a visual debugger (a supplied program that you may use to control the execution of your own program; the debugger will allow you to do things like arrange for your program to stop at places where you want to check on the values of chosen data elements or where you want observe exactly what is happening as each individual statement is executed).

Of course, the exact layout of an editor window, the compiler options, the libraries etc all do vary a little between different environments.

C++ input and output

Chapter 5 also contains an introduction to input and output. Any interesting program will produce some results that are to be output; most require some input data. The C++ language does not define input and output commands; instead it relies on function libraries. There are different input/output libraries that one may chose to use; the most common is the `iostream` library. The `iostream` library uses some of the more advanced features of C++ (such as object oriented design and "operator overloading"); but it can be used long before you get to really understand how its code works. Essentially, the `iostream` library provides two "objects" – `cout` and `cin`. The `cout` object "owns" the connections to the output device (normally the computer's screen) and provides all services related to output (such as copying character strings so that a program can get its output displayed on the screen, and converting numbers into sequences of printable/displayable digits). Similarly, the `cin` object "owns" the input device (usually this is the keyboard) and takes responsibility for translating keyed characters into values suable by a program. Simple use of the `cin` and `cout` objects is illustrated with some small examples.

Part II Simple programs

The second part introduces the fundamental programming constructs of imperative procedural languages.

Programs are built up from a *main* routine (main program) that utilizes " procedures" (or "functions", or "subroutines" – the names vary); these procedures may be specifically written for a particular application or may come from libraries. But all routines, (main and all its subroutines) are constructed in the same way. So we start by looking at a number of simple programs that just have a single main routine that makes limited use of standard functions from the input/output and mathematical libraries.

The constructs for building an individual routine include:

- variable definitions,

- simple (arithmetic) expressions that combine the values of variables to obtain new results,
- assignment statements that change variables,
- sequences of assignment (and input and output) statements that perform processing tasks,
- declarations of constants,
- iterative constructs for building loops
- selection statements for making choices.

Sequence

Chapter 6 deals with the basics. It focuses on programs that read data values, perform a few computational steps, and print the results. Such programs involve solely a sequence of variable definitions, input statements, assignment statements, and output statements. The data values used may be integers, real numbers, or individual printable characters. These basics are covered in Chapter 6.

Iteration

Chapter 7 adds loop constructs. Usually, a calculation will involve something more complex than a fixed sequence of steps. Instead, you may need to do the same (or similar) processing steps for each element in some collection, or you may need to repeat a particular operation many times.

"Boolean" variables, variables that represent the values true and false, and boolean expressions are also covered. This is a somewhat untidy area of C/C++ at the moment; the proposed standard for C++ will improve things.

C++ has a minor alternative to the normal while loop in its "do … while" construct. This is covered briefly. Then "for loops" are introduced. Two other control statements used with loops – "break" and "continue" – are noted but illustrations of their use is deferred to later more complex examples.

Selection

Selection statements, switch and if, are covered in Chapter 8.

File i/o

Chapter 9 introduces the use of files. Most realistic programs require reasonable amounts of input data (e.g. the names and marks for each of several hundred students enrolled in a course). It is very tiresome to have to type in such data "live" (make a mistake and you may have to do it all again). Even when testing programs, one often has to type in a substantial amount of data; retyping of data for each test run is unacceptable. It is worth learning quite early on how to use file i/o.

Part III Functions and data aggregates

The third part of this text focuses mainly on functions., but also introduces more complex forms of data such as arrays and structs.

Functions

Functions have a number of important roles:

- they allow a processing step, e.g. validated input of a data value, to be coded once and used from many points in a program rather than having the code repeated at each point,

- they capture and preserve expertise; experts can develop functions that implement particular algorithms, e.g. a sort algorithm; once developed, the function can be added to a library from where it can be copied and used in numerous programs;
- they serve as a basis for design.

Arrays

It is rare for one to be working with individual data elements as done in the examples in Part II. Usually, data values form related groups. Arrays represent a simple case of structured data; arrays group sets of data elements that are identical in type. The data elements making up an array are most often simple built in types so you can have arrays of reals, or arrays of integers, or arrays of characters; you can also have arrays of programmer defined structured types. The compiler knows that the only "structure" possessed by an array is that all the data elements belong together (allowing you to write code to do things like calculate how much memory space is used by an array), but at the same time the individual data elements in the array should be capable of being separately referenced so the compiler supports a mechanism that allows you to do things like ask for the "1st", "2nd", …, "199th" element.

Structures or "structs"

Application specific groupings of data are defined using "structs" (also known as "structures" or "records"). A struct declaration describes the data that belong together. Thus, in a program to handle sales records one might have a grouping "customer" (that groups data values of name, address, amount owed, date last contacted, ...), while in a word processor program you might have a grouping "format_record" (that groups data values of text font, style, size, ...). Usually, the individual data members that make a struct are of different types; a customer record might contain an array of characters for the name, and several different integers or reals (phone number, amount owed, etc). You do get structs where the data members are of similar types, e.g. a struct to represent a point in three dimensions would have three integer (or real) number data members to represent the different coordinates. Once you have defined the form of a structure, the compiler allows you have variables that are instances of that struct and to do things like assign values to complete structs or individual data members within structs. The major role of structs is this grouping of related data.

Chapter 10 introduces functions. It covers definition of functions and result types. The working a function calls, including recursive functions, is illustrated. Minor features like "inline" and "default arguments" are explained.

The arrays are covered in Chapter 11. The first examples use one dimensional arrays; these include "strings" – the arrays of characters used to represent textual data. Multidimensional arrays are then covered. Example programs illustrating the use of arrays include an encryption program, and a simple version of an "image analysis" program.

Lots of examples

Chapter 12 consists of lots of examples using functions and arrays. Some of these examples are complete programs; one program provides a simple model of heat diffusion, another is an implementation of the game "hangman". Other examples build up little "libraries" of useful functions; these include a "cursor graphics" package for

both Symantec and Borland environments. This package is used in a number of the other examples to provide simple visual outputs.

Functions for standard algorithms

Chapter 13 looks at some problems for which "standard" functions have been developed. These problems involve things like sorting or searching through arrays of integers. The methods for solving such tasks (the algorithms) are now well known; functions implementing these algorithms are available in libraries.

Tools

We diverge slightly with Chapter 14 which looks at two tools that help in the development of programs. The more important of these two tools is a "code coverage tool". A "code coverage tool" helps a programmer check that all parts of a program's code have been executed during test runs; this is an important part of the process of getting a program ready for practical use. Unfortunately, neither of the IDEs includes a code coverage tool; the examples use output from the "tcov" tool on Unix. The second tool is a profiler. Although less useful than code coverage tools, profilers are more widely available and are included in both IDEs. A programmer can use a profiler to determine where a program spends most time. This can be used to focus any work that is to be done on speeding up the program. Most of the programs that you will write initially spend their time largely in the system's input/output libraries so there won't often be much scope for speed improvements.

Design and documentation

Chapter 15 contains some suggestions for how you might go about designing, and documenting your programs. The design approach is the one underlying all the examples using functions that have been presented in Chapters 10…13. The design process is a form of "divide and conquer". You start by considering the overall task the program is to perform and break it down into a separate functions that deal with particular aspects like data input, computational steps, and results display. You then consider each function in turn, analysing it and breaking it down into functions. This "top-down functional decomposition" process is repeated until the functions are sufficiently simple that you can see how to express their processing in terms of the sequence, iterative loop, and selection constructs introduced in Part II. Top down functional decomposition is a limited approach. It works well for programs that basically "read data, process data, print results" and where the data are simple, e.g. an array of numbers like the heat diffusion example in Chapter 12. This design approach gets relegated to playing a minor support role in Part IV where more complex programs are considered.

Enum, struct, and union

Chapter 16 presents enumerated types (enums), structures (structs), and unions. These constructs allow a programmer to define new data types to supplement the integers, reals, and characters that are built in to the language.

Enumerated types are useful when you want to have a data element that must have one of a restricted number of values. For example, you want a "colour" variable that can be 'red', 'blue', or 'yellow'; or a "text_style" variable that can be 'plain', 'bold', or 'italic'. You can just use integers for such data; the advantage of enumerated types is that they give you a mechanism for telling the compiler about the semantics of these "colour" and "text_style" variables. If you define an enumerated "colour" type, the

compiler can check that colour variables are used correctly; the compiler will verify that only colour values like 'red' get assigned to a colour variable.

Technically, in C++ there isn't much difference between a struct and a simple class. In this text, a distinction is made. Structs are used only to achieve a grouping of related data elements. The examples in Chapter 16 illustrate the definition of structs and the ways in which code can manipulate both complete variables of struct types and the individual data members within a struct variable.

You should regard "unions" as a "read-only" feature of C++. For completeness, they are covered in section 16.3 using examples taken from the Xlib graphics library used on Unix. You will need to use unions when working with some C libraries like Xlib, but it may be years before you get to write code where it would be useful to define new unions.

Using structs

Chapter 17 has a few examples illustrating the use of structs. One example illustrates the idea of a "file of records", something commonly required in simple business data processing applications.

Bits and pieces

Next, bit-level operations are introduced. Of course all data are just bits. But usually we think about data as reals, or integers, or characters and forget the bit patterns. But there are a few situations where you actually want to use bits. Chapter 18 presents the bit manipulation facilities of C++ and has some illustrative examples like "hashing" and the use of a bit-vector used to summarise properties of a more elaborate data record. The main example in this chapter is a simple information retrieval system that has a file with the texts of newspaper stories and a separate index file that helps you find stories that use particular keywords.

Part IV A touch of class

The programming approach developed in Part III is fine for straightforward programs that have a single set of data that must be transformed in some way. Some scientific, engineering, and statistical calculations do have just arrays of numbers for data and, essentially, they simply calculate the value of some function of their input data. Most programs are more complex.

There are two complications. Firstly, the data structures become more elaborate; and, secondly, you start encountering problems where you have no idea how many data structures you require until the program is actually running.

Protecting data from arbitrary change

The more elaborate structures involve more than just lots more data fields. There are typically constraints on the values in different data fields. For instance, you might want a data structure that represents a point in two dimensional space with its position recorded using both cartesian (x, y) and polar (r, theta) coordinates. If you change a point's x-coordinate, you really should update its (r, theta) coordinates to match; there is this constraint, they should define the same place in two-dimensional space. Somehow, you need to be able to shield the data values from arbitrary changes. Instead of allowing direct access to the data in a structure, you provide an alternative functional

style of interface that permits changes but arranges that these are done in such a way that all constraints are still satisfied.

Design problems and top down functional decomposition

Large programs contain many different types of data and have to perform many functions. Consider for example, a simple system for maintaining hospital records. This would need data structures representing patients, wards, operating theatre schedules, laboratory tests. These data would be accessed by admissions staff, clerks calculating bills, laboratory staff recording results of blood tests, surgeons wanting lists of their day's tasks. It is difficult to identify the "function" of the program that could serve as the top for the kind of top-down functional decomposition approach illustrated in Part III. Even if you can come up with a design, frequently you will find that it is not very satisfactory. Everything seems interlinked. All the different functions scramble around the same data structures and arrays. Change something in the section used by the pathology lab, and you will find that somehow this alters the way the admissions system works.

Designing around the data

Often it is better to try a different decomposition of the problem, a decomposition that focuses on the data. So you identify important elements in the program: patient, theatre, ward. For each you identify the data owned; a patient owns a name, a ward number, a list of results of lab. tests, …. You also identify the tasks performed; a patient record can handle requests to note the result of another lab. test, report what ward and bed it is associated with, note an addition to its bill and so forth. By considering the data, you can break the overall problem down into more manageable parts. These separate parts are largely independent and can be developed in isolation. Once the parts are working, you make up the overall program by describing interactions, e.g. you can describe how the "laboratory" gives a "patient" a new "lab.-result" to note, and also informs "finance" to arrange additional charging.

Classes

Classes provide the basis for protection of data, and the packaging together of functionality and data. They also serve as the basis for the design of most the programs that you will write in future.

Dynamic data

Sometimes you know how many data elements your programs must manipulate, or at least you can identify a reasonable upper bound. Such knowledge simplifies design; you can plan out arrays to hold the data elements required. But there are situations where you don't know how many data elements you have and where guessing a maximum is inappropriate. All the modern languages in the Algol family allow you to create data structures "dynamically" as and when the program needs them. Often, these dynamic structures are used to build up program models of complex networks.

Beginners class

Chapter 19 introduces C++ classes. The first section covers the syntax, using simple examples like Points and Customers. The following two sections present more elaborate examples. First there is class `Bitmap`. This is a more systematic, more complete version of the idea of a bitmap (set) record structure such as that used to identify use of keywords in the information retrieval example given in Chapter 18. The second class is `Number`. Not your everyday integer, these `Numbers` have hundreds of digits. But you can do all the standard arithmetic operations using them.

Dynamic data and pointers

Chapter 20 looks at dynamic data and "pointers". Dynamic data are data structures created at run time. Obviously in order to use these data, a program has to know where they've been located in memory. That is the role of pointers; pointers hold addresses of dynamic structures.

Chapter 20 starts by explaining how it all works, providing a model of the "heap" (the area of memory used for these data). Next, there is a general introduction to pointers and ways of using pointers. There are a few simple examples illustrating pointers with "strings" (character arrays). Then there is a larger example illustrating the use of dynamic data in a little simulation of aircraft landing at an airport. Aircraft are objects that are created dynamically. The simulation allows aircraft movements to be controlled. If controlled correctly, aircraft land at the airport and, then, the corresponding dynamic data structures are destroyed.

Pointers aren't used solely with dynamic data structures. Chapter 20 explains some other uses. The older C language made much more use of pointers than does C++; you are cautioned against continuing with some of the older C style usages, such as use of pointers rather than array indexing.

The final section of Chapter 20 illustrates how to use pointers and dynamic data structures to build a very simple "network" of interlinked data elements.

Collection classes

Very often programs need to maintain collections of data. For example, a "patient" needs a collection of some kind to store the results of all the lab. tests performed; a "theatre" needs a list of the patients due for surgery that day; the air-traffic control example needed a collection of "aircraft" in the controlled air space.

Collections get used in different ways. You may want to represent a "queue" – first come first served. Alternatively you might have a priority queue, the tasks in the queue have priorities and an urgent task added later may jump to the front of the queue. You might just want a list; the order is arbitrary, you remove things from the list as you choose. Or you might have a collection where the stored items have "keys" (unique identifiers like "driver licence numbers") and you need to find an item given its key.

Just as some algorithms have been standardized and become standard functions in function libraries, so too have some mechanisms for storing collections of data. A collection class will define a set of operations for adding and removing items that are to be stored. It will also define some data structure that it uses internally to keep track of the stored items. Collection classes can be standardized and put into "class libraries". These libraries facilitate reuse, just as did the simpler function libraries.

Chapter 21 introduces a variety of simple collection classes and shows how they are implemented. The examples include queue, priority queue, dynamic array, list, and binary search tree. This introduction is fairly brief. You may well study such structures in more detail in a later "Data Structures and Algorithms" course.

World of Interacting Objects

Chapter 22 contains a couple of simple examples that illustrate the "world of interacting objects" that is characteristic of programs designed using classes.

Intermediate class

Chapter 23 expands on the initial treatment of C++ classes. Additional topics covered include: shared class properties, friends, iterators, operator functions, resource manager classes with destructors, and an introduction to "inheritance". "Inheritance" is

again motivated mainly by considerations of program design; it is shown how you can identify similarities among classes, represent these similarities using an inheritance class hierarchy, and then exploit this hierarchy to simplify the overall design of a program.

Two more trees

Chapter 24 presents two more "collection classes". They are both more sophisticated versions of the binary search tree illustrated in Chapter 21. The first AVL tree performs exactly the same task as earlier the binary tree class, it just performs better. Class BTree shows how a collection of data that is too large to fit into memory can be held on disk but still permit rapid retrieval of information. You may wish to ignore details of the algorithms used for these two examples because they are more complex than others in this text. But, you will probably find class AVL and class BTree quite useful building blocks for later programs.

Templates and Exceptions

Chapters 25 and 26 are both short. Chapter 25 gives a brief introduction to C++'s template facilities. Templates are a mechanism for making code more general purpose. You can write a "template" for a function like a "sort" function; this "template sort" describes how to sort any data for which assignment and the > (greater than) and == (equals) operations are defined. This saves copying and editing of standard code to make minor changes needed to accommodate different data. You can also define "template classes". The most common "template classes" are other "collection classes". There are some advantages in using templates for such classes, though they do tend to complicate syntax and some compilers still don't handle templates particularly well. The advantages relate to better type security (i.e. less chance of run-time errors due to programmer confusion as to the nature of the data being used).

Exceptions, Chapter 26, are intended as an error handling mechanism. The examples used in this book deal with errors, e.g. unreadable input files, by printing an apology and quitting. This isn't the best approach (for example, it wouldn't do for the program that controls Space Shuttle launches). A better mechanism is for the code that encounters an insurmountable problem to stop and return an error report to the code that called it. The calling code can handle the error report and, possibly, can implement a different strategy (e.g. ask the user to select another input file).

Supermarket

Chapter 27 is again just an example. Once again it is a simulation. This one makes limited use of inheritance and employs standard collection classes.

Design and documentation

Chapter 28 looks again at design and documentation, now from the perspective of "object based" programs.

Part V Object oriented programming

Finally section five provides a brief introduction to "object-oriented" (OO) programming.

Polymorphism

The introduction to inheritance in Chapter 23 showed you could exploit similarities to factor details out of code. The example used there was the "document" that owned a collection of "things" such as "text paragraphs, and "circuit components". When it

needed to save its data to file, the document used a loop where it got a pointer to the next "thing" in its collection and told that "thing" to write itself to the file. The document object didn't even know the range of types of "thing" that it could have in its collection; it knew only that all "things" could write themselves to files. The pointer it used to reference a thing was a "polymorphic pointer" – it pointed to "things" that had different shapes (some shaped like text paragraphs, others like circuit components).

Chapter 29 develops a complete program that illustrates this style of design and use of polymorphism. The example is a form of "dungeons and dragons" game. The dungeon is inhabited by numerous creatures drawn from different races. All creatures have essentially the same behaviour (they seek to end the game by eliminating the human player), but each different kind (class) of creature goes about this task in a distinct way. The game program works with a collection of creatures. At appropriate times, the creatures get their chance to proceed – the code is simple "for each creature in creature list do creature.run". Here the reference variable creature is polymorphic, as we move along the list the variable creature may reference first a ghost, then a patrol.

Polymorphism maybe great fun but in itself it isn't the reason why object oriented methodologies are increasing in popularity. So why OO?

Reuse! Reuse!

Really, the reason is that OO represents another step in the progression towards higher levels of design reuse.

Reuse with functions

Function libraries allow you to reuse algorithms. You don't write a sine function, you use sin() from the math library; you don't write a sort routine, you use qsort() from stdlib. Design by top down functional decomposition allows you to consider the overall role of a program and break down, over and over again, until you have functions that either exist in a library or that you can write. You build up a program from a set of reusable (and special purpose) functions that rummage through global data structures.

Reusable classes and object based design

Class libraries and object based programs allow you to reuse abstract data types. You don't have to implement code to interrogate the operating system and build up a representation of today's date, you use an instance of class Date from the class library. You don't implement your own search structure, you use an instance of class AVL or class BTree from some library. Your approach to design changes. Your first step is to identify a partitioning of the overall program system into separate components. You then identify their classes, determining what data each instance of a class must own, and what these objects do. Finally, you build up your program as a unique system of interacting objects (each jealously guarding its own data).

Can we reuse more?

When inheritance was introduced in Chapter 23, it was shown that this was a way of representing and exploiting similarities. Now, many application programs have substantial similarities in their behaviour. Just think about the way the different word processor, spreadsheet, diagram drawing and other programs work on your personal computer. They all have the same arrangements of "window", "menus", "undo/redo" editing operations and so forth. Obviously, they all have a lot of rather similar code. The data drawn in the windows, and the specific menu commands differ, but all the code for picking up a mouse click, recognizing it as a menu selection, finding which menu option must be similar in all the programs.

Reusable patterns of interaction?

If you could abstract out this similar code, you could greatly decrease the cost of developing new applications. The new applications could reuse not just individual classes, but groups of classes whose patterns of interactions are already defined at least in part.

Increasingly, this is being done. Once patterns of interaction, e.g. undo/redo an edit step, have been identified as being similar in many programs, an attempt is made to identify how this pattern can be represented in an abstract form. This leads to the identification of the classes involved and a particular sequence of interaction among them. Thus you might determine that in response to a menu request a Window object creates a Command object to hold the information needed to change (backwards and forwards) the data held by a Document object. You can define these as partially implemented abstract classes with the calls needed for Undo/Redo already encoded.

If you need to implement a similar editing operation in your own program, you can create classes that "inherit" from the partially implemented abstract classes. Your classes are just the same as those defined, except yours hold some data. Although overall patterns of interactions will be defined in the abstract classes, a few operations will have been left undefined. These are the operations that actually create and change data. All you need do is implement those few operations. You reuse the overall pattern of interactions.

Framework class library

The classes needed to build a particular kind of a program can be provided as a "framework". So, nowadays, you frequently get to program using a "framework class library".

Chapter 30 provides something of the flavour of an OO library by building a little framework. It isn't much, just a framework for programs that involve filling in and searching records that are displayed on the screen. For display, it uses the cursor graphics functions from Chapter 12. For storage, it uses the BTree class from Chapter 24.

Chapter 31 introduces the OWL (Object Windows Library) and TCL (THINK Class Library) framework class libraries. These frameworks (and similar frameworks like MFC, Microsoft Foundation Classes) are provided as part of the integrated development environments. They represent the reusable models for "Windows" and "Macintosh" programs. They embody many of the standard patterns that have now been identified as reusable structures for large scale programs.

Examples

You can't learn to program by learning some rules that define the syntax of a programming language and reading a few isolated code fragments.

You need to see a programming language in action. You have to see complete programs developed using the language constructs, as covered at a particular stage of your course, in order to understand how programs can be created.

It is not sufficient just to read the code of a final program. You need to see how the programmer got from a problem specification to that code. You have to learn how to "design" programs and how to record your design decisions.

Design

Design is an iterative process. You look at the problem. You isolate some aspects that can be considered further in isolation. You then work on each of these aspects, either finding ways of decomposing them into still simpler isolatable aspects or identifying how the work required could be expressed in terms of code.

Different problems suit different design styles. The earlier examples in this text are mostly programs that have the form "read some data, apply a function that computes something from those data, output the result". These problems are well suited to a design style known as top down functional decomposition. The aspect that you focus on is the main "function" of the program and you break this function down into simpler auxiliary functions. You specify what these functions do. You identify what data they need to exchange or share. Finally, you write some code.

Examples of functional design

Examples illustrating this approach are given in Chapters 12 and 17. They include things like a "Hangman" game, Conway's Life program, a program that keeps "customer records" in a file on disk. Each example has a fairly detailed discussion of design as well as code.

The later examples are more suited to "object based" design. You start by identifying "objects" that will be used in your program, and identify how these objects interact. You focus then on the different kinds, or classes, of objects that you need; trying to characterise each in terms of what such objects own and what they do. You elaborate on your initial ideas of how these objects interact. Your ideas about the classes objects get defined more formally. You elaborate the descriptions of the functions that define their behaviours. Finally, you write some code. You may not have to write all the code. As you developed your program design you may have discovered opportunities to reuse existing classes.

Examples of object based design

Chapters 20 and 22 contain examples, they include things like simulations of aircraft landing at airports and a system for maintaining files of reference cards (a kind of "Filofax" system). Again these examples, particularly those in Chapter 22, contain fairly lengthy discussions of design.

Examples of object oriented design

The final examples use "object oriented design". These extend on the earlier object based examples showing how more advanced language features can provide a cleaner overall design based on exploiting similarities that may exist among the different kinds of objects needed in a program. The example in Chapter 27 illustrates a simulation that makes relatively limited use of object oriented techniques. More elaborate examples are given in Chapters 29 and 30.

These examples should "bring you up to speed". You won't quite be ready to exploit the full power of your computer, its development system, and the "framework class libraries" that it includes. But you will have got far enough to understand the principles of those frameworks so that you can proceed on with one of the books that covers programming for your chosen system.

Many of the examples don't include the full code. The code, and code for "useful components" like the collection classes, is available. (You need to be able to use the "ftp" file exchange program on the Internet. You should find the code at the site named ftp.cs.uow.edu.au; the directories containing the code will be in "/pub/oop/ABC"). While you can simply fetch the code, it is better if you implement the examples for yourself using the code given and the design outlines for any components for which no code is given.

Note on filenames in examples

The development environments differ in their rules regarding filenames. In this text, it is assumed that there are no restrictions on name length; if your system restricts filenames, you will have to change those names that are too long.

The environments use suffixes on filenames to distinguish content. For example, a file for data may be supposed to have a name that ends with the suffix ".dat". Two different kinds of files are used hold program code. "Header" files (which contain just things like lists of function names and constants) typically have names that end with the suffix ".h". Different systems use different suffixes for the files containing the main code. Code files may be supposed to end with the suffix ".cp", or ".cc", or ".cpp", or ".CC" or something else. The text uses the suffix ".cp"; you will have to change this to the suffix appropriate to your system.

Table of Contents

Part IV

Part V

Figures

Part I Introduction