NEAR EAST UNIVERSITY

Faculty of Engineering

Department of Computer Engineering

Generating Time-Table with Object-Oriented Programming

Graduation Project
Com 400

Student: KADİME ALTUNGÜL (980215)

Supervisor: Dr. HALİL ADAHAN

Nicosia-2003

ACKNOWLEDGEMENT

I would like to thank Dr. Halil Adahan for his endless and untiring support and help and his persistence, in the course of the preparation of this project. Under his guidance, I have overcome many difficulties that I faced during the various stages of the preparation of this project.

I would like to thanks all of my friends who helped me to overcome my project especially Com.Eng Mehmet Göğebakan and Menderes Bozkurt, Zafer Akarsu. Finally, I would like to thank my family, especially my parents for providing both moral and financial support, their love and guidance saw me through doubtful times. Their never-ending belief in me and their encouragement has been a crucial and a very strong pillar that has held me together.

They have made countless sacrifices for my betterment. I can't repay them, but I do hope that their endless efforts will bear fruit and that I may lead them, myself and all who surround me to a better future. Also thanks all Teachers who behaved me in patient and understanding during my studying time

Especially to Assoc. Prof. Dr. Doğan Ibrahim, Assoc.Prof. Rahip Abiyev, Prof. Fakhreddin S. Mamedov for Everything they have done by helping.

ABSTRACT

The general characteristics of the object-oriented approach are described, and some key terms are introduced. The development of object-orientation is traced thrugh previous developments in languages and methodologies, and the object-oriented and procedural programming paradigms are compared. The history and development of C++ is explained in the context of a number of other languages which had an influence on it, principally C and Simula 67. Tha areas of syntax covered here are largely those directly inherited from C, since they cover the basic programming concepts of data representation, arithmetic, and functions. However, some aspects of syntax apply to C++ only.

In first chapter is about how we can build abstract models in programs of things in the real world. It discusses how encapsulation links together data and processes, how information hiding limits external access to internal data and processes, what an abstract data type is. It explains that the three componenets of an abstract data type are a unique name, attributes and methods. And also in this chapter we will look at the differences between classes and objects. We will examine what is meant by the three elements of an object, namely identity, state and behaviour.

In the second chapter investigates object lifetimeand visibility within a program, and compares external, automatic, stattic and dynamic objects in terms of these properties. The different roles of class atributes and methods as opposed to object attributes methods are outlined, also the relationship between constructors/destructorsand the metaclass. Aggregations-objects which are composed wholly or partly of other objects. Composition hierarchies describe aggregations are compared with classification hierarchies which describe inheritance, and the characteristics of aggregations are explored.

The various type of polymorphism are introduced and classified. The applications and syntax of operator overloading are explained in this chapter. The use of parameters to give polymorphic behaviour to functions and object methods is disxussed. Tis covers two general types of polymorphism; overloading and genericity.

The various types of container which may be required for different types of object collections. The container classes and the data structures which may be used to implement them. How a heterogenous collection of objects with polymorphic methods may be manged in a container. A generic approach to containers using template classes is demonstated. And also in this chapter we look at multiple inheritance, which allows a class to inherit from more than one base class. Uses of this facility are outlined, and potential conflicts between inherited elements are discussed. The role of virtual base classes and scope resolution in resolving ambiguity are demonstrated. Circumstances where multiple inheritance is necesarry are contrasted with those where other strategies can achieve similar result.

Object persistance and how objects can be made to persist beyond the lifetime of a single program run time, so that they may become common to multiple applications. The characteristics of object-oriented databass as a means for craeting and maintaining persistent objects, the syntax for streams and files is outlined in the context of providing persistent attribute data for bojects. The fundamental elements of any method are summarised, and a numer of different activities within the overall alaysis and design cycle are identified.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

NEU	Near East University
WWW	World Wide Web
LAN	Local Area Network
OOP	Object-Oriented Programming
OOD	Object-Oriented Design
OOA	Object-Oriented Analysis
OOBs	Object-Oriented Databases
GUIs	Graphical User Interfaces
PARC	Palo Alto Research Center
RAM	Random Access Memory
CPL	Combined Programming Language
BCPL	Basic CPL
BNF	Backus Naur Form
CLOS	Common Lisp Object System
GUI	Graphical User Interface
ATD	Abstract Data Type
Auto	Automatic variable
Extern	External variable
MAC	Military Air Command
AKO	A Kind Of
APO	A Part Of
LIFO	First In First Out'
FIFO	First In First Out
ODBMS	Object Database Management System
int	Type of variable (integer)
char	Type pf variable (character)
OOA&D	Object-Oriented Analysis & Designing
CRC	Class/ Responsibility / Collaboration

LIST OF SYMBOLS

~	Tilde
→	Arrow
:	Colon operator (Denote inheritance)
::	Scope resolution operator
{ }	Scope operator
<<	insertain operator
>>	extraction operator
||	Or operator
=	Assignment operator
= =	Equal operator
+	Add (plus) oparetor
/	Division operator
-	Subtract opeartor
&	Ampersand
//	Comment operator (Taken from BCPL)
/*	Comment operator (Taken from C)
|	or'-ing modes
\t	tab
\r	carriage return
\n	newline
? :	Conditional operator
,	Comma operator
*	Pointer (asterix)
( )	Function call Operator
[]	Subscripting operator

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

The book begins with background to object-oriented and C++, outlining their development. Bjarne Stroustrup has this advice about books on object-orientation; 'My rule is: look to see if there is a history section, and look to see if there is a lot of hype about object-oriented programming at the beginning; and if there is no history and there's object-oriented hype, don't touch it!' [Stroustrup in Watts(1), 1992]. In the first sections, there's some history and just a little object-oriented hype!

This followed by a basic introduction to the syntax of C++ (including data types, operators, selections, loops, functions and simple I/O syntax). It is not intended that this should provide an in-depth knowledge of C++, but simply that it should provide a basic level of knowledge appropriate to understanding the examples used in the book and being able to tackle the exercises with functional programs. In the following chapter, new concepts are introduced in the context of previous material. Another chapter cover relatively advanced topics such as building container classes, multiple inheritance, object persistence and approaches to object-oriented analysis and design.

Chapter I

OBJECT ORIENTED STRUCTURE & MODELLING

1.1 Object-Orientation

1.1.1 Objects in Software

The field of object-orientation is a large and growing one, and what it entails can to some extent depend on the context. For example, the facilities of the programming languages vary widely; some languages are more 'object-oriented' than others. Generally, however, object-orientation is about trying to represent the 'objects' that we find in the real world in software. These 'objects' may bevarious types, ranging from the physical to the more conceptual (some sort of container for other objects perhaps).

Shlaer and Mellor in their object-oriented analysis method identify five types of objects:

1. Tangible Things: Physical objects , eg 'car', 'bar code reader'.
2. Roles: Of the people ororganisations, eg 'account holder', 'employer'.
3. Includets: Something happenning at a particular time, eg 'flight', 'transaction'.
4. Interactions: A link between objects, eg 'electrical connection', 'contract'.
5. Specifications: A definition of a set of other objects, eg a descriptionof a particular type of stock item.

While this is only one approach to analysis, clearly some objects are more 'real' than others! A more general distinction in terms of object-oriented programming is made by Stroustrup when the identifies two kinds of object:

1. Those that directly represent the ideas used to describe the application.
2. Those that are 'artifacts of the implementation', objects used as programming tools to implement the code.

We may take a distinction then between objects that are part of an application such as banks, petrol pumps and suctomers, and others that are programming tools such as data structures (stacks, queues etc.) which allow us to implement the application objects.

1.1.1.1 Objects in General

Having said object-orientation is about representing 'objects' in software, we must ask ourselves two questions:

1. What is an object ?
2. Why use objects in programs?

We may answer the first question by saying that, in essence, objects are the elements through which we perceive the world around us. We naturally see our environment as being composed of 'things' which recognisable identities and particular behaviours. All objects have these characteristics of identity and behaviour which enable us to recognise them as discrete things. They also exhibit 'state', which means that we can describe an object is, as well as what it does and what it is called. Another key elementin the object-oriented model is that objects can be classified into types, again in a natural response to our environment. It is instinctive to classify individual objects according to their common characteristics; we recognise for example that all cats are of a single 'type'. Although we may also identify individual differences between specific cats, 'Cat'. We 'know about' all cats because we recognise what they have in common.

We are also able to recognise that some objects composed of other objects, or contain other objects. A 'house' object is composed of other such as bricks,windows, doors, tiles etc. A shopping trolley full of groceries is an object in its own right and a container of other objects. All these human responses to the world are elements of the object-oriented approach.

The basic philosophy of using objects in program is a simple one, that the world is composed of interactin classifiable and identifiable objects, and therefore programs, too, can usefully be structured in this way. Since all software applications serve people living in the real world, then taht software shopuld mirror the real world as closely as possible, both in its workings and im its underyling design. Simply, object-oriented programming is a more 'natural' way to program.

1.1.1.2 Object Processes

What is perhaps most important (and different) abuot the object-oriented approach to software is that it unifies inside 'objects' two things that have traditional been separeted in programming paradigms:

1. Data
2. Processes

The linking of these two aspects of rogramming is known as 'encapsulation', and allows the implementation if 'informating hiding', restricted access to the internal representations of objects. Objects have a 'private' part of hidden internal detail (state data and internal processes), and a 'public' interface which clearly defines the set of possible behaviours of the object. This separation of the public interface of an object and its internal implementation allows use to treat the two as separate parts of the programming process. An object's private implementation details can be defined without reference to other objects or a particular application context, and than the object may be used in an application that utilises only the object's public interface.

Both the public and private part of an object are defined in an 'abstract data type' which defines the common aspects of all objects in a particular 'class', i.e. objects that we classify as one type.

1.1.2 Areas of Object Technology

The general field of object technology can be seen as falling into four areas of application:

1. Object-Oriented Programming (OOP)
2. Object-Oriented Design (OOD)
3. Object-Oriented Analysis (OOA)
4. Object-Oriented Databases (OOBs)

All these necesarily interact some way, particularly object-oriented analysis and design, many methodologies have been proposed, but we can see common elements between them. Object-oriented databases are a very important field baceuse traditional forms of

data storage do not sit aesily with the object-oriented approach, an appropriate and flexilbe ways of storing objects on disk need to be found.

1.1.2.1 A Brief History of Object-Orientation

There have been a number of milestones on the road to object-orientation, both theoretical developments and language implementations. The following table shows some of the most important:

1968: *simula67 – the first object-oriented language*
1972: David Parnas' seminal paper on 'Information Hiding'
1970: Graphical User Interfaces (GUIs) developed using object-orientation
1980: First version of smalltalk
1983: First version of C++
1988: First version of Eiffel

Table 1.1: History of Object-Orientation

Although there has been a claim hat the designers of the 'Minuteman' missile used rudimentary object-oriented techniques as early as 1957 (Graham, 1992), the 'birtd' of object-orientation came in 1968 with simula67, the first object-oriented language, and an extremely influential one. One only has to consider that the key p0ersonalities behind three of the major object-oriented language, Bjarne Stroustrup (C++), Alan Kay (Smalltalk) and Bertrand Meyer (Eiffel), all had backgrounds in Simula to realise its influence. Simula was developed in Norway as a discrete event simulation language, and was the first language to introduce the key object-orinted concepts of classes and inheritance.

Many people have been influential in developing the teoretical basic of object-orientation, but perhaps the most influential singe paper was published by David Parnas, who introduced the idea of 'information hiding'. Parnas prencible was that a programming module should be 'reveal as little as possible about its inner workings'. This basic is crucial to the way that objects hide their internal implementations behind a public interface.

Thoughout the 1970s and 1980s, a great deal of development using object-oriented techniques went on at various research centers. At Xerox PARC, Smalltalk was developed, an object-oriented language that comes with its own WIMP interface. Its interface style in particular has had a strong influence on the subsequent grownt in the development of GUIs for many applications. Other languages also appeared in the 1980s, notably C++ and Eiffel.

Clearly, object-orientation has had a long development path, which has taken off only in recent years partly because the programming languages are very demanding on hardware resources. Another reason for its recent growth has been the popularity of the GUI, which has become the norm for all types of applications in recent years. Since the user interface has become increasingly more sophisticated, the advantages of the object-oriented approach, particularly in terms of reuse, have seenit become the standard method for coding GIUs. Other areas that have grown in importance in recent years are those of concurreency and rapid prototyping, both of which can benefit from the object-oriented approach. Design methodologies such as Booch's 'unified notation' provide us with tools for designing concurrent systems using objects, while in the area of rapid prototyping, the modular nature of an object-oriented program means rapid reuse and modification without compromising robutness or reliability. This is particularly useful in applications where there are new and unknown problems to be solved, it is possible to test various approaches quickly and efficiently when we can easily assemble and disassemble software objects in prototype systems. In addition, the fields of object-orientation and knowledge based systems seem likely to grow closer together as thet increasingly find common theoretical and practical ground [Harmon, 1990].

1.1.3 Basic Terminology and Ideas

The various aspects of object-orientation will be covered in detail throughout the book, but the three key features of the object-oriented approach are often quoted as:

1. Abstraction/ encapsulation
2. Inheritance
3. Polymorphism

1.1.3.1 Abstraction / Encapsulation

Abstraction is the representation of all the essential features of an object, which means its possible states and behaviours. These are 'encapsulated' intoan 'abstract data type' which defines how all objects in a class (type) of objects are to be represented and how they behave.

Encapsulation is the practice of including in an object everyting it needs, (both data and processes) hidden from other objects in the system. The integral state of an object is not directly accessible from outside, and cannot be altered by external changes to the aplication. Similary, changes to the internal implementation details can be made without affecting the external interface.

1.1.3.2 Inheritance

Inheritance means taht one class inherits the characteristics of another class as part of its definition. Inheritance is appropriate when one class is 'a kind of' onother class (e.g: A is a kind of 'text window' is a kind of 'window'). These types of hierarchies are fundamental to object-oriented programming. Some classes may inherit from more than one other class, and this is known as 'multiple inheritance' a 'classificaiton hierarchy' shows the inheritance relationship between classes, basedon the similarities between different classes of objects. Inheritance has two complementary roles:

1. To allow a class to be extended so that its existing functionality can be built on for new applications.
2. To allow similar objects to share their common properties and allowed behaviours.

1.1.3.3 Polymorphism

Polymorphism means 'having many forms', and there many forms of polymorphism! The general description of polymorphism is that it allows different objects to respond to the same message in different ways, the response specific to the type of object. This is achieved by various type of 'overloading', allowing a symbol (an operator like '+', or a function name like 'show') to have more than one meaning depending on the context in which it is used. Polymorphism is particularly important when object-oriented programs are dynamically creating and destroying objects in a classification hierarchy at

run time, so that it is not possible when compiling to predict the number or classes of objects which will receive messages when the program is running. We therefore have to defer decisions about the meaning of a particular symbol until run-time, a practice known as 'dynamic binding'. 'Run-time' polymorphismof this kind is one of the most useful aspects of object-oriented programming.

These three main ideas tend to build on each other, so that once we have encapsulation, then we can build classification hierarchies of inheritance, and from these we can really being to use the power of polymorphism.

1.1.3.4 Aggregation

In addition to the three concepts mentioned (which apply to classes) there are also 'aggregation' relationships (which apply to objects). These apply when objects of one class are composed of objects of another class. Aggregation of this kind are more flexible than inheritance because they can be more readily reorganised. Objects which contain other objects (but are not composed of them) are also forms of aggregation.

1.1.3.5 Object Based, Class Based and Object-Oriented

Because there is some separation between the different object-oriented conceps, we find some languages have more facilities than others for object-orientation, and some texts make a distinction between three types of programming, depending on the facilities available:

1. Object based
2. Class based
3. Object-oriented

Object based programming involves some aspects of encapsularion inside 'objects', which can be created from a specified set of existing classes, but does not provide mechanisms for createing new classes. Class based programming includes facilities for creating classes, but these classes cannot be organised into a classification hierarchy in order to implement inheritance however, the semantic differences between these two types need not further concern us here. The key point is that a truly object-oriented language has all the facilities for encapsulation, inheritance and polymorphism.

It is in fact possible to make many distinctions between the facilities of different object-oriented languages, and to say that one is more or less object-oriented than another. Wegner identifies no less than six 'languages classes' with different levels of 'object-orientedness' but, by most standarts, C++ provides all the crucial elements for applying the object-oriemted paradigm.

1.1.4 Object-Orienred Programming

There are in essence two elements to object-oriented programming:

1. Making Casses: Creating, extending or reusing abstract dta types.
2. Making Objects Interact: Creating onjects from abstract data type and defining their relationships over time.

The key to reusability in object-oriantation is that when we come to a new application, we do not beging by reinventing the wheel, but by looking to see if we can reuse any existing abstract data types in our new program. Some way may be reusable directly. Others may need to be extended by using inheritance to add extra functionality. Others again may need to be created from scratch. 'Domain analysis' is a term used to maen looking at an application area and identifying commonality between the new application and existing applications of a similar type. Common abstract data types can then be identified for reuse or extension.

Once all the abstract data types are identified or coded, then the program can be created by making and destroying objects and defining the messages that are paased between them over time. The way in which a program is approached will depend partly on the language used. In C++ we can use as much or as little of the object-oriented facilities on offer as we like. In a wholly object-oriented language there is no such option.

1.1.4.1 Comparing Approaches

What similarities can we identifiy between the types of programming elements we know from procedural programming, and what we havein an object-oriented programming? As the following table shows, there are several aspects that can be equated [adapted from Wirth, 1990].

Procedural	Object-oriented
Data type	Abstract data type / class
Varible	Object / Instance
Function / Procedure	*Method / Operation / Service*
Function calls	Message passing

Table 1.2: Procedural and Object-Oriented

All the aspects of the procedural paradigm may also be present in an object-orinted program to a greater or lesser degree, but object-orientation provides extensions to the facilities we already have. As well as existingg data types, we can define our own 'abstract data types' which describe all objects of a class. In addition to variables of simple data types, we can create instances of our abstract data types, called 'objects'. Instead of functions and procedures existing part from data adn other procedures, they become part of particular object classes, encapsulated inside abstract data types, and their calling mechanism is known as 'message passing', since objects communicate with each other by calling each others' method via 'messages'. The internal working of an object are no different in many ways to the internal working of a module in a procedural program, so many skills learned using other programming paradigms can be usefully employed in object-oriented programming.

1.1.5 Problems Associated with Object-Orientation

We might ask the question: If object-orientation has so many advantages, why has it taken so long to become established? Of course, like any other method, it has its problems. Many of these are simply because the languages are still in their infancy, an d the analysis and design tools even more so, so there is still much work to be done in providing the support for large scale projects using object-oriented methods. Software developers need proven tools, whether compilers, labraries, databases or CASE tools, and all these are still in relatively early stages of development.

Specific problems encountered when appliying the object-oriented approach to a software problem include:

1. Resource Demands: Since an object-oriented program can require a much greater processing overhead than one written using traditional methods, it may work more slowly. This is not good selling point from customer's perspective.
2. Object Persistence: An object's natural environment is in RAM as a dynamic entity. This is in constant to traditional data storage in files or databases where the natural environment of the data is on external storage. This causes problems when we want objects to persist between run of a program., even more so between different applications.
3. Reusability: It is not easy to produce reusable objects between applications when inheritance is used, since it makes their class closely coupled to the rest of the hierarchy. With inheritance, objects can become too application specific to reuse. It is extremely difficult to link to gether different hierarchies, making it difficult to coordinate very large system.
4. Complexity: The message passing between many objects in a complex application can be difficult ot trace and debug.

Nevertheless, all of thehe problems are easily outweighed by the potential benefits of object technology. As the technique matures and develops, it becomes more and more pervasive in all aspects of computing, to the point where it will become impossible to ignore.

1.2 C, Unix, BCPL and B

The history of C beings back in 1968 when two Bell Laboratories programmers were developing an operating system written in assembler on a DEC PDP-7 computer. This was the birth of the first version of the now widely used Unix operating system. In order for Unix to be more easily portable to other cumputers, it was necessary to replace the assembler with a higher level language, since assembly languages are vvery much tied to particular processors. The first non assembler Unix language was called 'B' and based on a 1967 British computerlanguage called BCPL (Basic CPL), itself based on an earlier language called CPL (Combined Programming Language) developed by Oxford and Cambridge Universities (1962-7). However, since B was interpreted rather than compiled, its performance was inadequate for Unix and it became a tool for writing an assembler for the more powerful PDP-11 to which Unix development was transferred in 1970. One of the key characteristics of BCPL and B is that they are 'typeless'

languages. Instead of dealing with a range of data types, they only address a 'machine word'. In order to improve the execution speed of B, Ritchie produced a compiler which was able to deal with types, over-coming inefficiencies caused by the 'typelessness' of B which meant complex operations identifying machine addresses: C was born! Unix was rewitten in C in 1973 and the rest, as they say, is history.

Because C was developed as a tool for driving an operating system, it had certain characteristics such as speed, compactness and some very low level elements. This made it became widely used, although an ANSI standard for Cwas not completed until 1990. The definetive C text is 'The C programming Language' by brian Kernighan and Dennis Kitchie.

1.2.1 C++

C++ was developed by Bjarne Stroustrup in 1983 at the AT&T Bell laboraties, and is basically a superset of C,though there are one or two inconsistencies. Stroustrup became interested in the object-oriented approach whilst stduying for his PhD at Cambridge University, using Simula 67 to write a distributed systems simulator. However, he had major problems with the resources required by the simulator, and had to write the final version in (interestingly enough) BCPL. In the light of this rather than difficult experience. Strousstrup decided that a proper programming tool was needed for complex projects and that such a tool should :

1. Help the complexity (use classes)
2. Ensure correctness (have strong type checking)
3. Be affordable in term of harware and resources
4. Be open – easy and cheap to investigate existing software libraries and facilities
5. Be portable

The language he developed was firdt called 'C with classes' (in 1980) and later C++. The name derives from the 'increment' operator in C, which is '++'. The effect of the increament opreator is to add one to the oprand. Therefore the name suggests that C++ is an incremental development of C- an extension to the existing syntax, as opposed to a separate language. C++ derives directly from C in terms of its syntax, but also owes much in terms of its facilities to Simula 67 and also Algol 68.

Stroustrup used C as the basis for his new language because, despite feeling that some acpects of C were problematical, it was good enough as a computational language, and it would enable existing C tools to be used in its development. No doubt another crucial factor was that he was working at AT&T Bell, the home of C, and had access to the Ccompiler and associated software. Also, since this was a commercial rather than academic project, it was not practicable to built a new language entirely from scratch. C's deficiencies 'are in the area of the program oragnisation and support for good design and programming, those then became my ares of work' [Stroustrup in Smith, 1989]. The combination of C's low level programming power and run – time efficiency, plus the higher level constructs added with C++, mean that C++ is able to span a wide spectrum of programming applications. Despite the fact that C++ was developed 'on the cheap', it has proved extremely succussful, with many former c applications migrating to it. Indeed some future development in the Unix operating system may well be developed in C++ rather than C.

C++ is in many ways a 'better C', and is primarily important for its addition of object-orientation. C++ can be used to write procedural code just like C, but its real value lies in the ability to write object-oriented programs. In this sense it is a hybrid anguage, able to produce both traditional procedural and object-oriented programs, or pretty much anything in between. From this point of view it is useful as a tool for an incremental approach to object-oriented programming.

C code can be written in such a way as to be virtually unintelligible, even to theprogrammer who wrote it. This rather unfortunate characteristic is also evident in C++, but it does have a number of improved syntax elements which make it easier to read and write whether or not the object-oriented facilities are used, and clarity of code is largely dependent on common sense. There is not much real gain in writing on one line what can be written much more lucidly on three.

C++ has come in for a lot of criticism from object-orientation 'purists' (such as bertrand Meyer [Watts (2), 1992]) who regard its hybrid approach as clumsy and inadeguate. However, stroustrup counters such criticism by stressing that it is a practical approach, allowing flexibility for working programmers to learn the concepts incrementally if

necessary without losing all the power and efficiency of C code. He is quoted as saying 'It is not right tobe pure. It is right to serve your own and others' needs. Diversity of approaches has been shown to work. There's not just one right way, and anyway I have a problem with the word 'pure', because it makes me think of Stormtroopers. [stroustrup , in Watts (1), 1992].

1.2.2 C++ Keywords and Functions

A keyword is a word which the compiler recognises as being part of the language's built in syntax. A function is a set of source code which is defined by a name. That name is not a keyword since it is not part of the compiler's instruction set.

C++ has a small set of keywords (about 60, depending on version and implementation) and some of those are not used very much (the dreaded 'goto' for example!). Compare this, for example, to COBOL with its vast and ever growing keyword syntax but through extensibility; the facility for a programmer to extend the language with ease. Most parts of a C++ program rely on the use of 'library functions' which are external to the compiler itself. These library functions ar ein effect C++ source code which is available to the programmer to incorporate into his/her own programs. It is possible to continuously build on existing functions to create code at a higher and higher level, so thatapparently very simple programs can be written, calling functions (or in the case of object-oriented programs, creating objects) that do all the serious work invisibly behind the scenes. Indeed, it is very much part of the philosophy of the object-oriented approach that the implementation details of an object in a program are not sen, only its externalbehaviour. A well written object-oriented program should be extremely easy to follow, even if the underlying code which defines a library object's behaviour is very complex indeed. Although there are only a few library functions definedas part of the standard language, the vendors who provide particular implementations of the C++ compiler tend to provide large function libraries for a wide range of applicaiton including graphics, hardware-specific functions and screen interface designs, among many others. These will of course vary widely in their particular contents and syntax.

1.2.3 C++ and Other OO Languages

How does C++ compare with other object-oriented languages? There are a number of object-oriented languages available with their own particular characteristics (some of these languages are listed in figure 1.1, but this is a very partial set). One thing which characterises C++ is its hybird nature, as an orthogonal extension of C. 'Orthogonal' is a commonly used term which strictly speaking means something to do with right angles, but in this context is used to mean that the object-oriented extensions do not affect the existing syntax. C++ is (with a few minor exceptions) orthogonal to C, and it shares this characteristic of being a direct extension of another language with CLOS (Common Lisp Object System), which is based on Lisp with object-oriented facilities. Previous 'object-oriented Lisps' included 'Loops' and 'Flavors' but CLOS has superseded these.

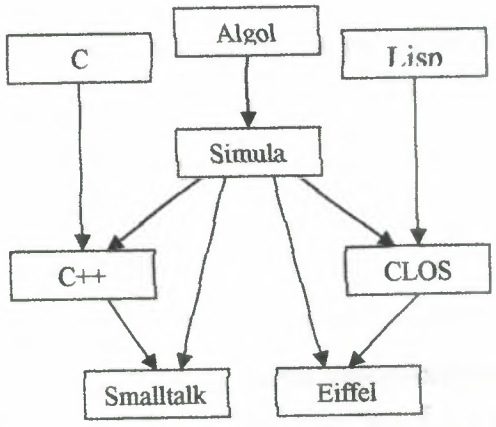


Figure 1.1: Some Object-Oriented Languages and Their Ancestors.

Extended procedural languages contrast sharply with 'pure' object-oriented languages such as Smalltalk and Eiffel whcih are not based onpre-existing sets of syntax (through there has beem some cross-fertilisation between LISP, CLOS and Smalltalk). In Smalltalk 'everything is an object.', and the language is fully integrated with its graphical user interface (GUI). It is impossible to write a Smalltalk program independent of its environment, which comes in two main version –'Smalltalk –80' and 'Smalltalk/V'. A smalltalkprogram is only executable within that environment and is not portable beyond it , which means something of a processing and cost overhead for those wishing to use a Smalltalk application. It is not really possible to learn object-oriented concepts incrementally in Smalltalk; rather one has to go in at the deep and sink or swim!

The Eiffel language is named after the French enginer and the tower he built, which serves as an analogy for object-oriented software construction – 'It is built of a few small, simple parts which you combine' [Meyer in Watts (2), 1992]. It was originally developed by Bertrant Meyer in 1985 and is described in Meyer's book 'Object-Oriented Software Construction' [Meyer, 1988]. Eiffel is , like Smalltalk, entirely based on the implementation of object-oriented concepts and is not an extension of an existing procedural language syntax. It contrasts with it, however, in not being dependent on a graphical environment. Executable Eiffel programs are as portable as those generated by C++. Although Eiffel has yet to find a large user base, it may well emerge as one of the main object-oriented languages.

In the software marketplace, it seems that the close similarity of C++ with an existing commonly used language has been something of an advantage, in that former C programmers are more willing to learn the additional syntax than to learna completely new language from scratch. The ability to either learn or convert to the object-oriented syntax elements incrementally seems to be an aid to the transition to object-orientation for many programmers, though it often leads to rather odd programming practices during the transition which are neither one thing nor the other. Nevertheless, Stroustrup wholly endoreses this approach of taking on board object-oriented syntax one step at a time, 'you do not expect things to happen in one step...It takes a little time but it always happens...going in at the deep end is just asking for failures'. This should be a good maxim to bear in minds as you read this project.

1.3 Functions

When you declare a function you do not have to prototype it! You must give the function definition physically before you call the function. You simply type in the entire definition of the function where you would normally put the prototype.

The programming concept is the inline function. Inline functions are not very important, but it is good to understand them. The basic idea is to save time at a cost in space. Inline functions are a lot like a placeholder. Once you define an inline function, using the 'inline' keyword, whenever you call that function the compiler will replace the function call with the actual code from the function.

How does this make the program go faster? Simple, function calls are simply more time consuming than writing all of the code without functions. To go through your program and replace a function you have used 100 times with the code from the function would be time consuming. Of course, by using the inline function to replace the function calls with code you will also greatly increase the size of your program.

Using the inline keyword is simple, just put it before the name of a function. Then, when you use that function, pretend it is a non-inline function. Inline functions are very good for saving time, but if you use them too often or with large functions you will have a tremendously large program. Sometimes large programs are actually less efficient, and therefore they will run more slowly than before. Inline functions are best for small functions that are called often.

Functions that a programmer writes will generally require a prototype. Just like an blueprint, the prototype tells the compiler what the function will return, what the function will be called, as well as what arguments the function can be passed. When the function returns a value, that the function can be used in the same manner as a variable would be. For example, a variable can be set equal to a function that returns a value between zero and four.

When the programmer actually defines the function, it will begin with the prototype, minus the semi-colon. Then there should always be a bracket followed by code, just as you would write it for the main function.

Return is the keyword used to force the function to return a value. Note that it is possible to have a function that returns no value. In that case, the prototype would have a return type of void.

The most important functional question is why. Functions have many uses. For example, a programmer may have a block of code that he has repeated forty times throughout the program. A function to execute that code would save a great deal of space, and it would also make the program more readable.

Another reason for functions is to break down a complex program into something manageable. For example, take a menu program that runs complex code when a menu choice is selected. The program would probably best be served by making functions for each of the actual menu choices, and then breaking down the complex tasks into smaller, more manageable takes, which could be in their own functions. In this way, a program can be designed that makes sense when read.

1.4 Arrays and Pointers

One of the most common problems in programming in C (and sometimes C++) is the understanding of pointers and arrays. In C (C++) both are highly related with some small but essential differences. You declare a pointer by putting an asterisk between the data type and the name of the variable or function:

```
char *strp;      /* strp is `pointer to char' */
```

You access the content of a pointer by dereferencing it using again the asterisk:

```
*strp = 'a';          /* A single character */
```

As in other languages, you must provide some space for the value to which the pointer points. A pointer to characters can be used to point to a sequence of characters: the string. Strings in C are terminated by a special character NUL (0 or as char '). Thus, you can have strings of any length. Strings are enclosed in double quotes:

You can view str to be a constant pointer pointing to an area of 6 characters. We are not allowed to use it like this:

```
str = "hallo";   /* ERROR */
```

Because this would mean, to change the pointer to point to 'h'. We must copy the string into the provided memory area. We therefore use a function called strcpy( ) which is part of the standard C library.

```
strcpy(str, "hallo"); /* Ok */
```

Note however, that we can use str in any case where a pointer to a character is expected, because it is a (fixed) pointer.

1.5 Selection and Iteration

1.5.1 The if Statement

There are some aspects of programming that are common to all programming languages. One such item that C++ has in common with other programming languages is the if statement. The if statement is used to test for a condition and then execute sections of code based on whether that condition is true or false.

If the conditional expression evaluates to false, the code block associated with the if expression is ignored, and program execution continues with the first statement following the code block. In some cases you want to perform an action if the conditional expression evaluates to true and perform some other action if the conditional expression evaluates to false. In this case you can implement the else statement. You can nest if statements if needed. Nesting is nothing more than following an if statement with one or more additional if statements.

1.5.2 Thrown for a Loop

The loop is a common element in all-programming languages. A loop can be used to iterate through an array, to perform an action a specific number of times, to read a file from disk the possibilities are endless. All loops have these common elements:

- A starting point
- A body, usually enclosed in a brace, which contains the statements to execute on each pass
- An ending point
- A test for a condition that determines when the loop should end
- Optional use of the break and continue statements

The starting point for the loop is one of the C++ loop statements (for, while, or do) followed by an opening brace. The body contains the statements that will execute each time through the loop. The body can contain any valid C++ code. The ending point for the loop is the closing brace.

Most loops work something like this: The loop is entered and the test condition is evaluated. If the test condition is not met, the body of the loop is executed. When program execution reaches the bottom of the loop (usually the closing brace), it jumps back to the top of the loop, where the test condition is again evaluated. If the test condition is not met, the whole process is repeated. If the test condition is met, program execution jumps to the line of code immediately following the loop code block. The exception to this description is the do-while loop, which tests for the condition at the bottom of the loop rather than at the top.

1.5.3 The for Loop

The for loop is probably the most commonly used type of loop. It takes three parameters: the starting number, the test condition that determines when the loop stops, and the increment expression.

The for loop repeatedly executes the block of code indicated by statements as long as the conditional expression, cond_expr, is true (non zero). The state of the loop is initialized by the statement initial. After the execution of statements, the state is modified using the statement indicated by adjust.

1.5.4 Learning About Scope

The term scope refers to the visibility of variables within different parts of your program. Most variables have local scope. This means that the variable is visible only within the code block in which it is declared. The scope-resolution operator tells the compiler, "Give me the global variable x and not the local variable x."

1.5.5 Extern Variables

A real-world application usually has several source files containing the program's code. A global variable declared in one source file is global to that file but is not visible in any other modules. There are times, however, when you need to make a variable visible to all modules in your program. Doing this is a two-step process. First, declare the variable in one source file as you would any global variable. Then, in any other source file that needs to access the global variable, you declare the variable again, this time with the extern keyword: extern int countChickens; The extern keyword tells the compiler, "I'm going to be using a variable in this source file that you will find declared in another

source file." The compiler sorts it all out at compile time and makes sure you get access to the correct variable.

While global variables are convenient, they aren't particularly OOP friendly. Usually there are better solutions. In addition, global variables consume memory for the life of the program. Local variables use up memory only while they are in scope. Use local variables whenever possible, and keep the use of global variables to a minimum.

1.6 Modelling the Real World

1.6.1 Encapsulation

Leaving objects members to public access can bring serious data integrity problem like changing the stored size of an array without adjusting the array size. One way to protect data and methods is to declare them as private. Private members can only be accessed by their class or subclasses. So it is wise to declare private all members that should not be accessible by objects users. Private specification can be applied to the object data and methods as well as to class data and methods. Sometimes, for simplicity or efficiency, it is also useful to have direct access to object members like for the complex number object. Public and private members and methods can be intermixed without any problem.

1.6.2 Information Hidding

We have defined the term 'encapsulation' as meaning that we represent insoftware some kind of unit which, like real world objects, has both state and behaviour. This integration of state and behavior allows us to implement 'information hiding' which means thst encapsulation hides private elements of an object ehind a public interface. This separation between the public and private parts of an encapsulated object has two aspects –protection of the object's state from unforeseen external influences, and hiding of the implementation details used to define an objects behaviour.

Firstly, then, 'information hiding' means that an object's state cannot be altered except by its fixed methods of behaviour. This is crucial, because we are trying to protect the state data inside an object from being changed by unpredictable externalinfluences. If we take the coffee cup example (figure 1.2), we can say that it has a state of 'fullness'

(how much the coffee in the cup). Unless we resort to conjuring tricks, this state can only be affected by specific operations: being filled (from kettle,jug etc.) or being emptied. No other external effect can change the state of 'fullness' The protection of state data is something which object-orientation provides us with in software. In effect, when programming we need the security of knowing that data in our program won't suddenly turn from white to black becasue the sun has gone behing a cloud.

Figure 1.2: A Coffee Cup Encapsulates Both its State and its Behaviour.

This protection of state data behind a 'fire wall' of fixed behaviour makes the life of the programmer easier because it means that the behaviour of software is predictable and less prone to error. It enhances reusablity and maintainabilty baceuse we only have to be aware of an object interface, not its internal structures.

The second aspect of information hiding is that we do not need to know how behaviour happens, only that it does. In software, what goes on inside the programming unit which represents an object is not relevant to the user. We don't need to know how the manufacturer made our caffee cup white, we only need to know that it is white so that it will match our kitchen. This 'public' view of the cup would be uneefected by any change in the manufacturer's process which is 'private', hidden from the user. In the same way, the behaviour of a software 'object' should be uneffected by changes in its internal implementation.

The way in which encapsulation allows information hiding to be applied is often described in terms of a 'daughnut' diagram (figure 1.3), the private state is hidden behind the surrounding public interface, which represents taht behaviour which is externally visible. From 'outside' the object, we cannot see the private, internal elements. What constitutes the pulic interface is defined in the 'abstract data type'. In fact, a jam doughnut is a better analogy than a ring, since its internal state is hiddden!

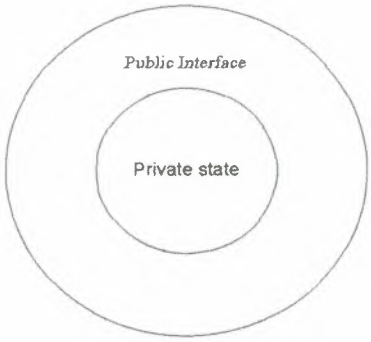


Figure 1.3: 'Doughnut' Diagram of a Hidden.

1.6.3 Abstract Data Types

When using the coffee cup as an example of an object, we assume that we are loking at a particular coffee cup, and are evaluating its particular state and behaviour. However, we are able to identify it as a coffee cup because it is one of many which we recognise as being of a particular type of object. We know the possibilities for the state and behaivour of a particular coffee cup because i fact it shares these with all other coffee cups. It is this commonality of possible state and behaviours which allows us to 'classify' objects as belonging to particular groups, or 'classes', and when we try to model objects in software, we do so by defining the possible states and behaviours of all objects of a particular type. The vehicle for doing this is the 'abstract data type'.

Anyone who has any degree of programming experience will understand what is meant by the term 'data type'. It defines the type of data which a particular variable can hold, it may be an integer, a character, a float, or any of a range of simple data storage representations. However, when we build object-oriented systems, we use more complex data types, known as 'abstract data types', which represent more realistic entities. Whereas the 'integer' data type defines how all integer number are handled in a program, we might be interested in interested in representing a 'bank account' data type, which describes how all bank accounts are handle in a program.

Abstraction is about reducing complexity, ignoring unnecessary detail. An abstract representation of something is supposed to contain the esential features of what is being represented. A realistic painting contains all the details seen by the eye, whereas an abstract painting intends to reflect the essential features of what is being seen. Unfortunately that meaning is not necessarily comunicated to the viewer of the picture!

1.6.4 Objects and Classes

We have already referred to 'objects' as things in the real world, and used the example of a coffee cup as a particular type of object. We instinctively classify objects in term of what we see as their types based on the attributes and behaviours they have in common. The term 'classify' gives as the idea of the 'class'- all objects belong to a particular class of object. The common element of a set of objects which allow use to classify them are those which we try to encompass in an abstract data type. Indeed C++ uses the word 'class' to describe an abstract data type.

1.6.5 Attributes and Methods

If we are to build an abstract data type in a program, we have to identify three things:

1. The abstrack 'thing' we are trying to represent
2. The data which represents the state of thatb 'thing'
3. The behaviour of that 'thing'

The that 'thing' will be the name of the abstrack data type. It may be worth stressing here that the 'thing' we are trying to define is not the object, but a class of objects. The class will be define the common attributes and methods for all objects of the class (figure 1.4).

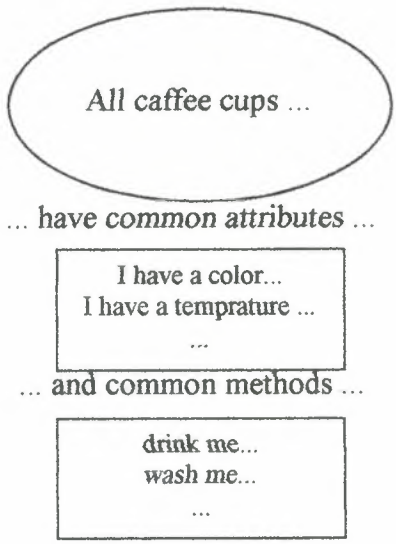


Figure 1.4: All Coffee Cups have Common State Attributes and Behaviour Methods.

1.6.5.1 Attributes

Having decided what our abstrack data type is going to represent, we have to identify what elements go to make up it is state. These are known as 'attributes' in object-oriented terminology, but in programming terms are variables and \ or data structures (e.g arrays, lists etc.). It is important to not that when we identify attributes, we are not specifying values for them. For example, we might decide that our data must have the attribute of the 'colour', but we do not predetermine what that colour might be for any individual coffee cup, only that all coffee cups will have colour of some kind. This is exactly analogous to deciding the record structure, we only specify their names and types, but the data held in each field may be different for each actual record.

1.6.5.2 Methods of Modelling

The behaviour of our 'thing' is going to be defined by it's 'methods' (sometimes called 'perations' or 'services'), which are processing routines related to the data type. In an abstract data type, the methods act as the only path between the user and the state data-it is normal practice to limit access to the data so that it can only be accessed via these methods. There are several types of method, but two in particular are relevant here:

1. 'selector' methods
2. 'modifier' methods

A 'selector' methot is a 'read' method, one which allows access to an attribute, but does not allow that attribute to be altered. This is analogous to the traditional idea of a function (data is uneffected by executing a function). It is also known as a 'get' method. A 'modifier' methot is a 'write' method, allowingattributes to be altered. This is analogous to the traditional idea of a procedure (data is changed when a procedure is executed) and is also known as a 'set' method.

It is up tp the designer of the abstract data type to decide which attributes may be read by selector methods, and which may be altered by modofier methods. If an attribtes has no selector method, then to all intenets and purpoese it is 'invisible' from outside the abstract data type. If it has no modifier method then its state cannot be altered. In practice most attributes will have both 'get' and 'set' methods as a matter of course.

1.6.5.3 The Classes in C++

In C++ we model an abstact data type by using the 'class'. The class has two parts; 'private' and 'public'. The private part o0f a class contains any elements which are to be hidden form public access. The public part of the class defines the 'behaviour' (methods) of any object of the class. In a program, a behaviour means a process which the object is able to perform or undergo.

It is normal prctice to put attributes into the private part of the class, where they can only be accessed via methods. Methods themselves appear in the public part of the class to they can be accessed externally and provide the interface to the class. It is possible to put attributes in the public part of the class. It is also possible to put methods in the private part of the class. This is useful for methods which are used by other methods of the same class, ut not appropriate as part of the external interface.

1.7 Classes and Objects

1.7.1 What is a Class, What is an Object?

The role of the class is to act as a kind of blueprint for all the objects of that type. A class defines the types of state data appropriate to the class (the attributes), and also its set of allowed behaviors (the methods). The class is in effect an 'object factory' which allows use to create the new objects of the class, each one of which follows an identical pattern of methods and attributes. A single abstract data type is used to create as many instances of the class as we wish, in the same way that a single simple data type can be used to create many instances of that type.

It might be useful to think of the class as, for example, a machine like one, which makes security badges for visitors to a company. The machine makes a standard pattern of badge and defines thwe-required attributes for each one such as the visitor's name, their own employer and the person they are visiting, perhaps even a photo of the visitor. Each badge which the machine makes is an object, a single 'instance' of the class, and it is each badge which contains the 'state' data for the attributes defined by the class. The class contains the attributes, but it is the objects, which actually contain the state data. Each badge will have a standard behaviour, a standard set of methods defined by the class. Note that these are not different for each object. If the 'class machine' fixes a

safety pin onto the back of the badge, then it will have the methods of the being put on and taken of etc, and all objects will have exactly the same behaviour, even though their state data is different .

Each object created by the class is a single 'instance' of that class, and this is why the term 'instantiation' is used in object-oriented terminology. 'Instantiation' means the creation of a single instance of a class.

1.7.1.1 Class

A class is the implementation of an abstract data type (ADT). It defines attributes and methods, which implement the data structure and operations of the ADT, respectively. Instances of classes are called objects. Consequently, classes define properties and behaviour of sets of objects.

Objects are uniquely identifiable by a name. Therefore you could have two distinguishable objects with the same set of values. This is similar to ``traditional" programming languages where you could have, say two integers i and j both of which equal to ``2". Please notice the use of ``i" and ``j" in the last sentence to name the two integers. We refer to the set of values at a particular time as the state of the object.

1.7.1.2 Object

An object is an instance of a class. It can be uniquely identified by its name and it defines a state which is represented by the values of its attributes at a particular time. The state of the object changes according to the methods, which are applied, to it. We refer to these possible sequences of state changes as the behaviour of the object.

1.7.1.3 Behaviour

The behaviour of an object is defined by the set of methods, which can be applied on it. We now have two main concepts of object-orientation introduced, class and object. Object-oriented programming is therefore the implementation of abstract data types or, in more simple words, the writing of classes. At runtime instances of these classes, the objects, achieve the goal of the program by changing their states. Consequently, you can think of your running program as a collection of objects. The question arises of how

these objects interact? We therefore introduce the concept of a message in the next section.

1.7.2 State, Identity and Behaviour

A class is defined by three elements:

1. A unique Name
2. Attributes
3. methods

In constant, an object is defined by:

1. Identity
2. State
3. Behaviour

In each of these three cses, the poperty of the class relates in some way to the property of the object (figure 1.5). The attributes of the class allow each object to contain state data –one value for each attribute. The methods of the class define the possible behaviours of the object. However, the concept of identity is slighty more complex than the name of the class.

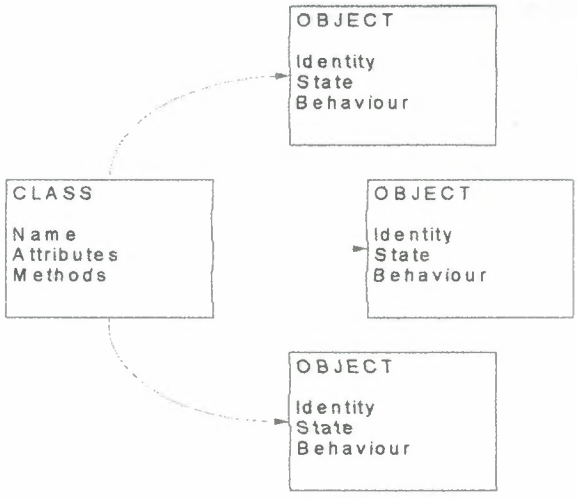


Figure 1.5: Objects have Identity, State and Behaviour.

It is possible for a specific object to be identifiable by a unique name. In an oilfield control system for example there might be a fixed and limited number of individual oil pumps, each one easily identifiable by a unique name, e.g. 'Oil pump 1', 'Oil pump 2' etc. However, this simplicity of idetity is not always the case. In some situations, there

many objects constantly being creadet and destroyed. If it is raining, what identifieseach raimdrop? It would be absurd to suggest that we could name every raindrop., but each does exist and therefore must have some kind of uniqueidentity. In some cses, the idea that each object can have a unique name falls down. What in act differentiates one raindrop from another is simply the cpace that it occupies at a particular time. The identity of any object may be ultimatelydefinable only by this space/time relationship (figure 1.6).

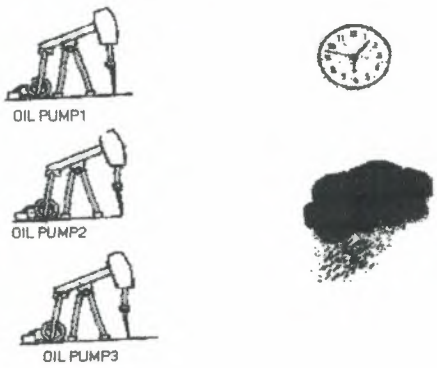


Figure 1.6: Objects of Predictable Number and Lifetimes have Unique Names.

In object-oriented programs which have to keep track of large number of objects which are constantly being instantiated and destroyed, objects will be identified by their memeory locations rather than by unique names. Objects which are predictable are identified, like classes by unique names, but objects which are unpredictable rae identified by location.

It is sometimes possible to differentiated objects by key attributes, in the same way that records in an indexed file or database table can be accessed by some unique data field(s). However, this is not what is meant by object identity, and it is perfectly possible for more than one object to have identical state values for all attributes.

Chapter II

APPLICATION INTERFACE of OOP

2.1 Object Lifetimes and Dynamic Objects

2.1.1 Object Persistance and Visibility

There are a number of factors which affect the perssistence and visibility of an object during the run time of a program. Some objects may exist throughout the running of a program, and may be visible in all modules. Other objects may exist momentarily within the limited scope of a particular method or statement body. Between these two extremes we may have a range of lifetimes and visibility among instantiated objects.

2.1.2 Types of Object

There are four types of object which have we may instantiate in a program. The first three are objects with specific names, put the fourth, dynamic objects, cannot be identified in this way:

1. External (global) Objects: Persistent (in exstence) througout the lifetime of a program and having 'life scope'-visibility througout themodule source file. May be visible in more than onemodule, perhaps visible in all modules (global).
2. Automatic (local) Objects: Persistent and visible only througout the (local) scope in which they are created.
3. Static Objects: Persistent througout a program but only visible within their local scope.
4. Dynamic Objects: Lifetime may be controlled within a particular scope.

These types of object all serve different purposes, and require different forms of language syntax for their creation, access and destruction. In our examples so far, we have only created objects with specific names, but we have also discussed the possibility that some objects cannot have unique names, and are identified only by the space which they occupy at a particular time.

In some programs there may be a fixed number of clearly identifiable objects whose existence is pedictable in all runs of a promram. When objects are pedictable enough to be identifie at a compile time, we are able to give them unique names. Such objects may be either 'external', 'static' or 'automatic' depending on their required persistence and visibility as defined by scope. In contrast, dynamic objects are those which cannot be identified at compile time, either in terms of their number or their identities, and their lifetimes may be controlled independently of scope.

2.1.2.1 External (Global) Objects

An external object is one which is persistent and visible throughout a program module, i.e. its scope is an entire module. It may also be made visible in other modules. Objects, which fall into the category of 'external', would be ones whose numbers and identities remained constant throughout an application. For example, if we are monitoring 6 petrol pumps in a garage forecourt simulation program, then there are unlikely to be changes to this setup when the program is running. Even if a pump is out of action it does not cease to exist, it simply changes its state. In this kind of scenario, the objects can have unique names. They also persist for the lifetime of the program. If they are instantiated as software objects when the program starts up, then they will exist in the program until it shuts down, since the physical pumps also persist in reality (figure 2.1)

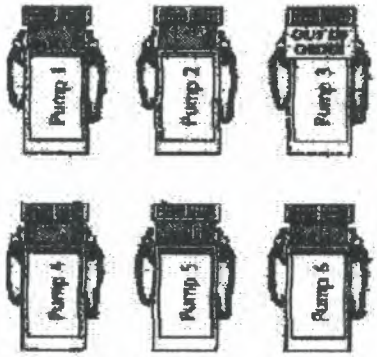


Figure 2.1: These Petrol Pumps have Predictable Lifetimes and Identities.

Such objects may be regarded then as 'external' objects, those that persist throughout the program lifetime. External objects are always global for the module in which they are declared, and are often global throughout all program modules. Depending on implementation we might have objects, which are persistent throughout the lifetime of the program but not global. This is purely an implementation detail relating to 'linkage',

the way in which the linker resolves the visibility of objects between modules. An object with a name which is local to a module is said to have 'internal linkage', whereas those whose names are visible in multiple modules have 'external linkage '.

2.1.2.2 Automatic (Local) Objects

As well as external, global objects, we may also have a number of locally declared 'automatic' objects, objects that exist in a predictable manner for a particular period of time. The key difference between an external and an automatic object is that whereas an automatic object is instantiated within the scope of part of a program module, an external object is instantiated outside of any scope. 'Automatic' objects are automatically destroyed when they fall out of the scope in which they were instantiated.

For example, we might have a system with some form of object-oriented menu driven user interface, with one of the objects in the system a 'help' menu. Since the help menu only needs to be visible to the user when he/she requires help, it does not need to exist for the whole lifetime of the program, but only needs to be instantiated when required, and can be automatically destroyed when no longer needed. Since there will only ever be one help menu, and the circumstances in which it will be required are constant and predictable, then we can give it a unique name, and confine its existence to a particular 'scope' in the program, the part where the help menu is visible.

The existence of an automatic object is therefore delimited by the scope of the part of the program in which it is instantiated. Depending on where it is instantiated. This may mean that the object is in existence for the whole time the program is running, but not necessarily visible throughout run-time. This is because even if an automatic object persist throughout a program module. It cannot be made visible in other modules in the system.

2.1.2.3 Static Objects

Static object s declared like automatic object within a local scope, but it is persistent throughout lifetime of program. In this type of menu, the possible options would be different depending on the current configuration, so that the menu might display 'mono screen' as an option while the current configuration is color, but 'color screen' if the

current state is mono. In cases like this, it is useful if the menu can 'remember' its state from the last time it was in scope.

2.1.3 Dynamic Objects

It may be that objects in a system are not predictable enough to be instantiated as external, static or automatic objects. This is the case when we are unable to predict at compile time:

1. Object Identities
2. Object Quantities
3. Object Lifetime

For example, if we return to the garage forecourt simulation scenario, it would probably be the case that the vehicles present on the forecourt at any one time need to be represented as objects in the system.

In this circumstances, neither external, automatic nor static objects can be used to represent vehicles, since such objects need to be predictable-require unique names, are of fixed number and fixed lifetimes. We need to represent unpredictable combinations of vehicles using objects, which can be dynamically instantiated and destroyed at run time to represent the vehicles entering and leaving the system. The way in which such collection of dynamic objects may be managed will be dealt with in a later chapter, but the first step is to investigate how dynamic objects may be created and destroyed.

2.1.3.1 Creating Dynamic Objects

When creating objects, we cannot give them the unique names, which are possible with other types of objects. This means that we have to have some other way of referencing objects at run time, and in practice this is done with pointers. We have already discussed the use of pointers to memory locations, for example in referencing the address of the first character of a string, and the use of pointers to reference dynamic objects is similar. However, we still need a mechanism to create dynamic objects, and this is through the use of dynamic memory allocation. In C++, a pointer can be directed to an area of dynamically allocation memory at run time in order to reference newly created object. In this way, the constructor can be called via the pointer.

2.1.3.2 Destroying Dynamic Objects: The Destructor Method

We have already looked at the constructor method, and how it creates a software object by reserving memory for the object's attributes. We also know that, whether or not we define it, a constructor is always called when an object is created the default constructor is used if the programmer does not supply one to override it. An object cannot be instantiated without soma form of constructor to reserve memory because the existence of any object depends on it occupying memory space. Indeed, the identity of a dynamic object depends entirely the memory location it occupies at a particular point in time.

If an object cannot be instantiated without a constructor method, it follows that an object cannot be destroyed without a 'destructor' method. This is a method, which allows us to destroy an object, and if we do not define one, a default destructor is called whenever an object is destroyed. Its primary purpose is to free the memory used by the object. Like the constructor, it can be extended by the programmer to perform additional functions if required, though unlike the constructor it cannot take parameters. The destructor executes when the object is destroyed either specifically by the programmer, by falling out of scope, or by the program terminating.

2.1.4 C++ Syntax

In the previous chapter we looked at the creation of objects with unique names, created within the scope of a function. Before investigating dynamic objects, we should constant the instantiation of external, static and automatic objects.

2.1.4.1 The Lifetime of Named Objects

When we create a named object in a program, its lifetime and visibility is controlled only by its scope. Named objects, as we have stated, may be of three types:

1. Automatic Objects: Objects instantiated inside the local scope of a function or other structure with its body defined by braces. They only exist while they are in scope.
2. External Objects: Objects instantiated outside any function body. These have 'file scope' and exist for the lifetime of the program.

3. Static Object: Object instantiated inside local scope and having local visibility, but persisting from their declaration to the end of the program.

In all three cases, we only have control over the instantiation of objects, not over their destruction, which is controlled automatically by the compiler.

2.1.4.2 Automatic Object Declaration

The objects which we instantiated in the previous chapter were automatic objects, declared inside the local scope of a pair of braces. Since braces delimit all block structures in C++, then we can declare automatic objects within any of these. The following object is automatic because it is declared inside the body 'main '. It will be visible anywhere inside 'main', and will persist until 'main' finishes. There is in fact a keyword ('auto'), which may be used to precede automatic objects, but it server no useful purpose since this is the default type of any object declared within a local scope.

2.1.4.3 External Object Declaration

An external object is declared outside the scope of any braces. This declaration of objects is all we are able to do outside of scope; we cannot for example call any methods of 'account1' except inside the braces of a function body. The main reason for doing this would be to make the object visible in other source files which are in the same system, since only externally declared objects can be referenced in other program modules. This is known as 'external linkage'.

2.1.4.4 External Linkage

As noted above, external objects and variables are those declared outside the body of any function. In a system containing several program modules, such objects may be declared as 'extern' in other modules if their visibility needs to extend beyond the file scope in which they are defined. For example, if the external object 'an_object' is declared in one module, it must also be re-declared in other modules which reference the object, but the compiler has to be made aware that it is in fact the same object being referenced in all modules. This can be achieved with the 'extern' keyword (figure 2.2) and is known as 'external linkage', linking the declaration of an object in the scope of one file with the declaration of the same object in separate files. The object must be declared in one module, and then declared as an 'external' object in all other modules

which reference it. In figure 2.2 'an_object' is declared in the source file of 'prog1', and declared as an external object in 'prog2' and 'prog3', allowing the object to be referenced in those programs do.

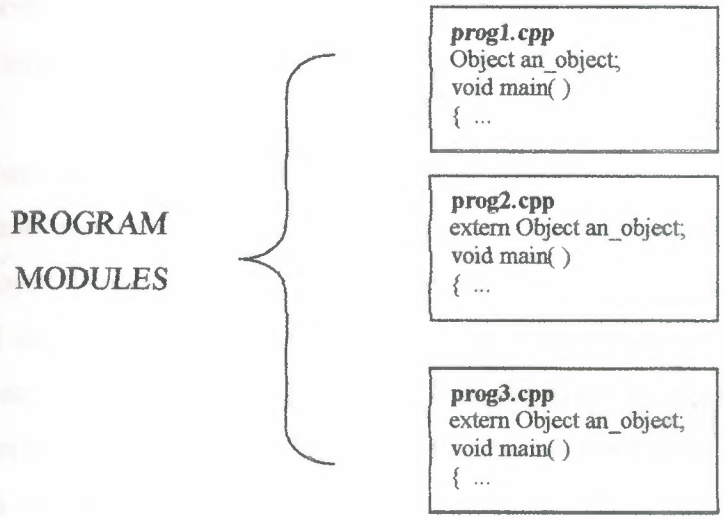


Figure 2.2: An External Object Declared in one Module.

2.1.4.5 Static Object Declaration

A static object is declared, like an automatic object, within a local scope. However it is only initialized once, regardless of the number of times it falls in and out of scope. In the following function, a static object is contrasted with an automatic object. Both will come into scope when the function is executed, but the automatic object will be created and destroyed each time. The static object will be instantiated only once and will persist until the end of the program:

```
void afunction ()
{
BankAccount account1;
static BankAccount account2;
....
}
```

The advantage of a static object is that because it persist, it can retain its state even when it is not in scope. A static object cannot also be declared 'extern', i.e. it can only be visible within one module (source file).

2.1.5 Dynamic Oject Syntax

In this section we have discussed object being dynamically created and destroyed as a program is running. We have previously looked at the syntax for the creation of objects by the constructor method, which reserves memory space for the new object and performs any use-defined initialisation processing.

2.1.5.1 Controlling the Lifetime of Dynamic Objects

External and static objects persist as long as the program in which they are declared and are destroyed when it terminates. An automatic object is definedby its scope, and is destroyed when it passes out of scope. We have no other controlover the lifetimes of these objects, which are destroyed by the compilerat the appropriate time. If we are to exercise more control over the lifetime of objects, we need to be able to destroy objects at will, as well as create them. In other words we need to be able to call the 'destructor' method as well as call the constructor.

The syntax which we have used to call the constructor in previous chapters is only applicable to the instantiation of named objects, and a different form of the constructor call has has to be used to create unnamed, dynamic objects. We also need to be able to explicitly call the destructor, rather than simply allowing objects to fall out of scope. To handle the dynamic memory allocation of data, including objects, C++ includes the operators 'new' and 'delete'. These operators are used to dynamically call object constructor and destructor respectively via the use of pointers. It is important to note that they are operators, not keywords, so they can be 'overloaded'.

2.1.5.2 Creating Dynamic Objects: The 'new' Operator

C++ includes this special memory allocation operator ('new') for use in objects constructors, though it may be used to allocate storage for variables of other types as well. The effect of 'new' is to allocate memory via a pointer of the required type. The syntax is based on the creation of a pointer, and then direction of that pointer to an area of memory, which will contain an object.

This pointer is now able to point to an object of the BankAccount class. The pointer can be directed to any 'BankAccount' instantiated using the 'new' operator:

```
account_pointer=new BankAccount;
```

Or the creation of the pointer and the constructor call can be put into a single line:

```
BankAccount* account_pointer=new BankAccount;
```

'account_pointer' is not the name of an object, but the name of a pointer to an object. It is able to point to any dynamic object of that class, and may be redirected at run-time to point to various dynamic objects instantiated using 'new' do not have names –they simply occupy a memory space which may be referenced by a pointer.

At this point in the program, both pointers now reference the object, and either could be used to send messages to it. Although pointers to dynamic objects may be redirected, it is, however, essential that a dynamic object is being referenced by at least one pointer at any one time; otherwise it will be lost and inaccessible.

2.1.5.3 Calling the Methods of a Dynamic Object

In order to send a message to a dynamic object, the pointer must be de-referenced before the object's public interface can be accessed. Therefore, instead of the dot operator, the de-referencing 'arrow' operator must be used for objects instantiated in this way:

```
Pointer_name-> methodName();
```

There are many advantages to the instantiations of objects dynamically using pointers rather that creating them as named objects, such as the ability to destroy them while they are still in scope, and to handle disparate objects in 'container classes' whereby groups of objects can be handled in organized collections such as stacks, queues and lists. It also allows us to implement 'run time polymorphism'. There are also potential problem areas, such as keeping track of which pointers are currently referencing an object, making sure that all objects have at least one pointer referencing them and taking responsibility for 'cleanup' when objects are no longer required.

2.1.5.4 Destroying Dynamic Objects: The 'delete' Operator

You may have noticed that in the previous two examples, though we created dynamic objects by calling the constructor with 'new', there was no explicit call to the destructor. If dynamic objects are allowed to fall out of scope in this way then there is no automatic

destructor call, and no 'cleanup' of memory. Whilst this does not matter in these specific examples it would matter very much in large-scale programs.

C++ includes the 'delete' operator to destroy objects which have been instantiated dynamically using 'new' –'delete' explicitly calls the destructor, which destroys the object currently referenced by the pointer, as follows:

```
delete pointer_name;
```

2.1.5.5 Directing Pointers to 'NULL'

It is worth noting that a pointer, which is not directed to an object, may point to any random area of memory, so how can we tell if it is referencing an object or not? This problem can be addressed by directing any pointers, which are not currently referencing objects to 'NULL', which is equivalent to the base memory address (address 0). This is a constant declared in the standard header files 'stddef.h' and 'stdlib.h', and is automatically included in 'iostream.h'. Before referring to NULL in a program therefore we must include one of these files.

It is good practice always to initialize pointers to NULL when they are declared. Not only does it allow us to check if a pointer is referencing a valid object, but using 'delete' with a pointer, which is directed to NULL, is guaranteed to be harmless. In contrast, calling 'delete' with an unused pointer referencing some random area of memory is asking for trouble!

2.1.5.6 Losing Objects

As noted previously, if a pointer to an object created with the 'new' operator is allowed to pass out of scope without the 'delete ' operator being used, then the destructor will not be called. The object will still exist but the 'lost', since it has no pointer to reference it, it will be unreachable. In a large program, 'garbage' objects such as this may eventually cause memory management problems. It is up to the programmer to ensure that every object has at least one pointer referencing it any given time. Of course, a single object may be referenced by many pointers. One potential pitfall to be aware of is that using the same pointer to instantiate many objects is perfectly acceptable, but every time 'new' is used, the pointer is redirected to a new area of memory. It does not

automatically destroy any object, which the pointer may already be referencing. Unless another pointer is already pointing to that object, it will be lost.

2.1.5.7 Defining a Destructor Method

In the above examples, we have been calling the default destructor, which, like the default constructor, does not need to be defined by the programmer. Its sole function is to free the memory previously allocated to the deleted object. However, we may extend the destructor to perform other processes if we wish. Like the constructor, the destructor has certain characteristics, which mark it out from other methods:

1. It takes the same name as the class, preceded by the 'tilde' character (~)
2. It cannot take arguments
3. It cannot return value

For a class called 'Queue' for example, both the constructor and the destructor would also be called 'Queue', but the name of the destructor would be preceded by the tilde, which would also appear in any out of line definition of the destructor method:

```
Class Queue
{
public:
Queue();  //constructor prototype
~Queue(); // destructor prototype
};
//out of line destructor definition
Queue::~Queue
{
// body of destructor method....
}
```

It is a usefull convention for the destructor to follow the constructor in the class declaration, followed by the other methos.

As with the constructor, if the default destructor is all that is required, there is no need to state it explicitly, but ifsome processing is reguired to take place when an object is destroyed, then the destructor has to be specified. Unlike a constructor, a destructor xannot take take parameters and cannot therefore be 'overloaded', i.e. there can only ever be one destructor per class. In contrast there may be more than one constructor for one class. As stated previously, the destructormethod is invoked whenever an object is destroyed. This may be done by the compiler or explicitly by the programmer.

2.2 The Metaclass

2.2.1 The Role of the Metaclass

The metaclass concept encompasses some important aspects of the role of the classes, and therefore provides a useful framework for discussion. The use of 'meta' as a prefix has a wide range of meanings, includng, simply, 'about'. The term 'metaclass' is similar in usage to 'metalanguage' a language used to describe some other language. There are also a munber of terms where 'meta' is used to imply some form of abstraction. From these interpretations we may conclude that the metaclass tells use about the class, and may be seen as an abstraction of the class in the same way that the class is an abstraction of a set of objects.

The metaclass is often described as the 'class of a class', which may be seem a rather odd expression, but it can be explained as follows.The class, as we know, holds the attributes and methods which will apply to the class itself, therefore it is the class of the class! (figure 2.3)

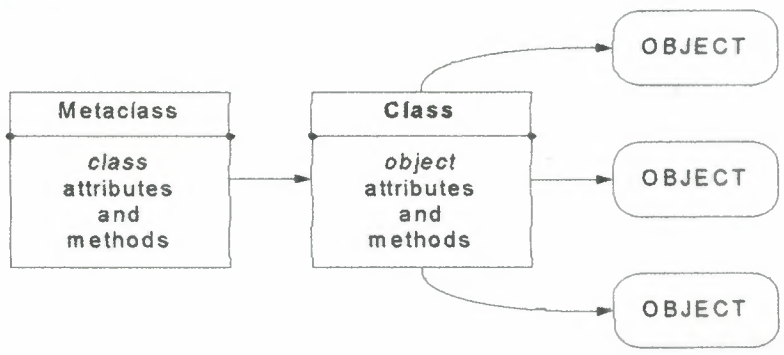


Figure 2.3: The 'Metaclass' is the Class of the Class.

Every class has one (and only one) metaclass, and the metaclass contains those parts of a class which are not appropriate to be duplicated for every specific object. The metaclass may seem a rather esoteric concept, but it is basically a repository for those parts of a class which must exist at the run time of a program, whether or not there areobjects of the class in existence. What kind of things are appropriate to the class, but not to the objects of that class? Let us begin by reviewing the relationship between classes and objects.

2.2.1.1 Limitations of Object Attributes and Methods

A class acts as a kind of blueprint or skeleton for objects of the class. The class does not exist as a specific entity, but is the abstract representation of a user-defined data type. It defines the attributes which all objects in the class will contain to represent their state, and the methods which may be used to send messages to them.

When an object is created, its attributes will have state values, independent of the state values of other objects. For example, if we instantiate 3 'fuel tank' objects, all 3 will have an attributes of 'fuel level', but each object may have a different state value for that attribute (figure 2.4)

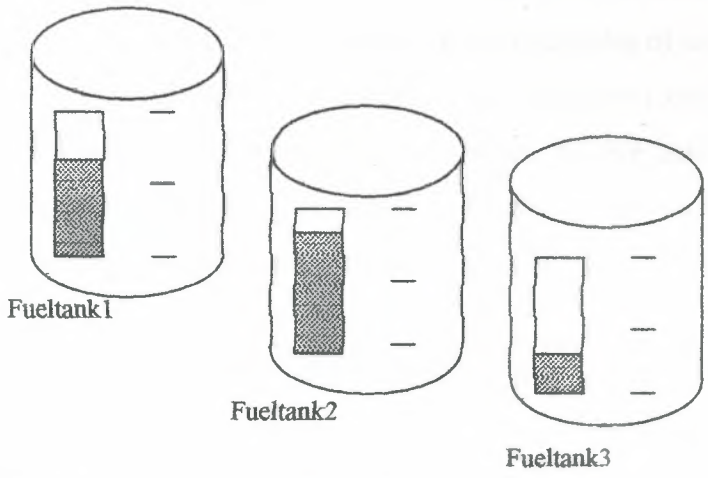


Figure 2.4: Limitations of Object Attributes and Methods

However, sometimes we may want to have a physical representation of something about a whole class of objects rather than about one object in particular. Suppose we want to know how many objects of a particcular class exist at any one time, where do we record this information?

With what we know about classes and objects, we can see that there would be serious drawbacks to having an attribute ai each object which stored this total. It would be possible to implement such an attribute, but to avoid each object containing a different total, that attribute would have to be changed for every object each time a new object was created or destroyed, and would be an unnecessary duplication of the same data.

2.2.1.2 Class Attributes

Consider a large system which keeps track of vehicles passing through a controlled traffic system, representing each vehicle as adynamic object created when that vehicle enters the system and destroyed wheen it leaves. If we wanted to know at any one time how many vehicles were in the system, then each 'vehicle' object would have to have constantly updated count of how many vehicles (objects) currently existed, every single time another vehicle entered or left the controlled traffic area. This is clearly an inappropriate approach - details about the whole class should not be the responsibility of specific objects.

What we need therefore is not an attribute duplicated for every object, but a single attribute for the whole class. It is true of course thet we could have a simple counting varible outside the class itself, but this would undermine the principles of encapsulation by making the attribute simply a global variable, open to manipulation from outside the class. In order to encompass such attributes inside the class, we are able to make a distinction between 'objects attributes' and 'class attributes'. The term 'instance variable' are sometimes used to mean the same thing as 'object attribute' respectiely.

Given that we want to have class attributes as well as object attributes, but need to keep them encapsulated in the abstract data type, where do we put them? We cannot put them in the class since, as we know, all attributes defined in the class are properties of individual objects. The answer, as you may have worked out by now, is to put them in the metaclass.

The metaclass, then, is a repository for class attributes which only have one instance, regardless of how many objects of the class exist. In the traffic system reffered to above, the metaclass might for example hold an attribute called 'vehicle_count' (figure 2.5),

which would be incremented each time a vehicle entered the system and decrement when a vehicle left. Class attributes arein essence any data which tells us about the class as a whole rather than a specific object.

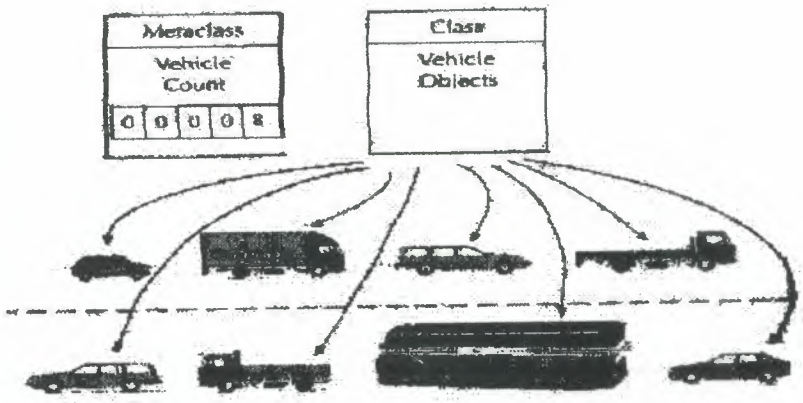


Figure 2.5: The Metaclass Contains Class Attributes.

It is important to not that, despite conceptual differences, the class and metaclass are inextricably linked, and that class attributes are just as 'visible' to objects of the class as object attributes are. Indeed, in C++ the metaclass an d the class are part of the same structure. However, object attributes belong to individual objects and are not accessible by the class.

2.2.1.3 Class methods

Having established that the metaclass contains class attributes, let us examine the role of 'class methods'. Like class attributes, they also exist in the metaclass, and are as visible to objects as they are to the class.

Consider our example of a class attribute which keeps a count of how many objects exist in a 'live' system atb any one time. If we wish to return the current state of this count, we will need some method to do so. It is perfectly acceptable to access class attributes with object methods, as we have stated there is no difference in visibility from an object's point of view between an object attribute and class attribute. However, what happens if there are no objects instantiated when we want to return the count? If there are no objects currently in existence, then we have no mechanism by which to call the method which returns the count, so we cannot access the class attribute! This is where

we need a class method, which is usable by the class directly, whethet or not any objects of the class exist, class methods, like class attributes, reside not in the class but in the metaclass (figure 2.6). Class methods may only access class attributes since object attributes can only be accessed by objects.

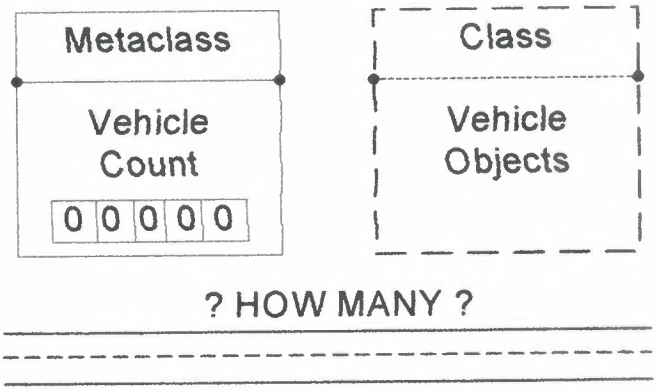


Figure 2.6: Class Methods Allow Access to Class Attributes.

2.2.1.4 Other Components of the Metaclass

Apart from class attributes and methods relating to them, what else can go in the metaclass? In practice, C++ makes no further distinction between parts of the class and the metaclas and as we will see, subsumes both class and object attributes and methods into the 'class' body. However, there is a semantic distinction between ordinary methods and those which create and destroy objects, the constructor and the destructor. Both of these methods are said to reside in the metaclass. The reason for this is simply that since objects do not create objects, then the method which creates them cannot be an object method. Therefore the constructor belongs to the metaclass rather tahn the class.A similar argument follows for destructors; an objectcannot destroy itself, it is destroyed by the permanent instantiation of the class, the metaclass.

Much of the above is a semantic distinction which has few practical implications in C++, and the concept of the metaclass is only a significant element in Smalltalk programming. Other object-oriented languages such as Effiel have little time for it. For our purposes, the main thing is to be clear about why some attributes and methods belong to the metaclass and others belong to the class. C++ does not have separate metaclass constructor in its syntax, but it does make this important distinction.

2.2.1.5 Metadata

A similar and semantically related term is 'metadata'. The term applies to any data which describes other data, such as a class definition or perhaps a database table record struture. The term may be applied to real world data too: 'There are real-world things that describe other real-world things. A part description in a catalog describes manufactured parts. A blueprint describes a house. An engineering drawing describes a system' . a class, as we know, has a metaclass which describes the data associated with it, and the class in turn describes the data associated with objects. The metaclass, then, is the metadata for the class, which in turn in the metadata for the objects. The term 'meta-object' is therefore sometimes useed to refer to the class definition (figure 2.7).

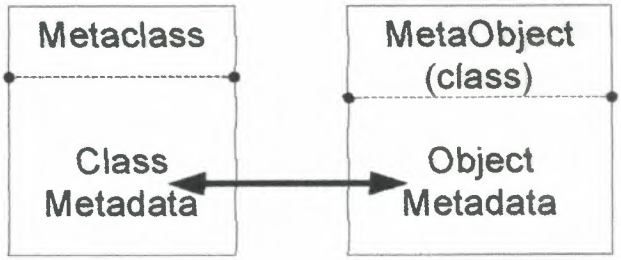


Figure 2.7: The Metaclass Provides 'Metadata' About the class Itself.

'Metadata' may also be taken to mean generic classes and methods ('templates' in C++) which are able to act on a range of types.

2.3 Inheritance

One of the most powerful features of classes in C++ is that they can be extended through inheritance. Inheritance means taking an existing class and adding functionality by deriving a new class from it. The class you start with is called the base class, and the new class you create is called the derived class.

When you derive a class from another class, the new class gets all the functionality of the base class plus whatever new features you add. You can add data members and functions to the new class, but you cannot remove anything from what the base class offers. You'll notice that in the private section there is a line that declares a variable of the Mission class. The Mission class could encapsulate everything that deals with the mission of a military aircraft: the target, navigation waypoints, ingress and egress altitudes and headings, and so on. This illustrates the use of a data member that is an

instance of another class. In fact, you'll see that a lot when programming in C++ Builder. There's something else here that I haven't discussed yet. Note the virtual keyword. This specifies that the function is a virtual function. A virtual function is a function that will be automatically called if a function by that name exists in the derived class. But if you wanted your function to take the functionality of the base class and add to it, you would first call the base class function and then add new code. By calling the base class function, you get the original functionality of the function. You could then add code before or after the base class call to enhance the function.

The scope-resolution operator is required only when you have derived and base class functions with the same name and the same function signature. You can call a public or protected function of the base class at any time without the need for the scope-resolution operator, provided they aren't overridden. For example, if you wanted to check the status of the aircraft prior to takeoff, you could do something like this:

```
void MilitaryPlane::TakeOff(int dir)
{
if (GetStatus( ) != ONRAMP) Land( ); // gotta land first!
Airplane::TakeOff(dir);
// new code goes here }
```

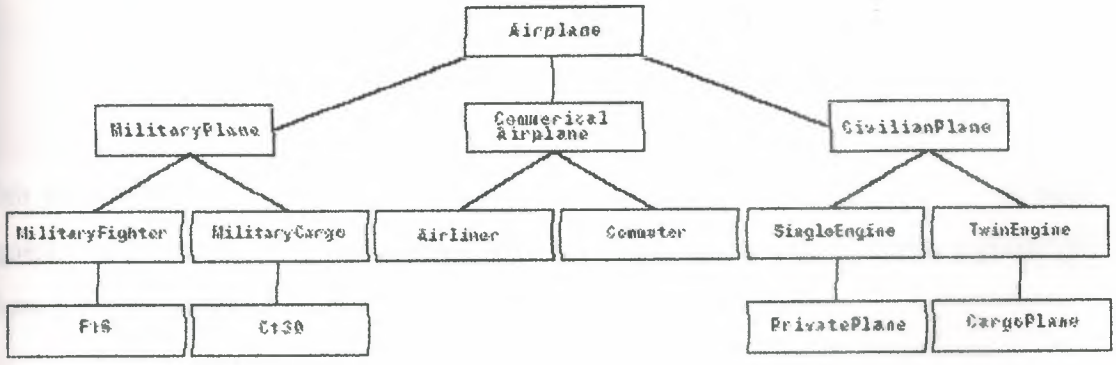


Figure 2.8: The Simple Figure About the Inheritance

You can see from Figure 2.8 that the class called F16 is descended from the class called MilitaryFighter. Ultimately, F16 is derived from Airplane since Airplane is the base class for all classes.

2.3.1 Types of Inheritance

You might notice again the keyword public used in the first line of the class definition. This is necessary because C++ distinguishes two types of inheritance: public and private. As a default, classes are privately derived from each other. Consequently, we must explicitly tell the compiler to use public inheritance. The type of inheritance influences the access rights to elements of the various superclasses. Using public inheritance, everything which is declared private in a superclass remains private in the subclass. Similarly, everything which is public remains public.

The leftmost column lists possible access rights for elements of classes. It also includes a third type protected. This type is used for elements which should be directly usable in subclasses but which should not be accessible from the outside. Thus, one could say elements of this type are between private and public elements in that they can be used within the class hierarchy rooted by the corresponding class.

2.3.2 Constructure

Constructors are methods, which are used to initialize an object at its definition time. Constructors have the same name of the class. They have no return value. As other methods, they can take arguments. Constructors are implicitly called when we define objects of their classes:

```
Point apoint;          // Point::Point( )
Point bpoint(12, 34);  // Point::Point(const int, const int)
```

With constructors we are able to initialize our objects at definition time as we have requested it in section 2 for our singly linked list. We are now able to define a class List where the constructors take care of correctly initializing its objects.

If we want to create a point from another point, hence, copying the properties of one object to a newly created one, we sometimes have to take care of the copy process. For example, consider the class List, which allocates dynamically memory for its elements. If we want to create a second list, which is a copy of the first, we must allocate memory and copy the individual elements

With help of constructors we have fulfilled one of our requirements of implementation of abstract data types: Initialization at definition time. We still need a mechanism, which automatically ``destroys" an object when it gets invalid. Therefore, classes can define destructors.

2.3.3 Destructure

Consider a class List. Elements of the list are dynamically appended and removed. The constructor helps us in creating an initial empty list. However, when we leave the scope of the definition of a list object, we must ensure that the allocated memory is released. We therefore define a special method called destructor which is called once for each object at its destruction time.

Destruction of objects take place when the object leaves its scope of definition or is explicitly destroyed. The latter happens, when we dynamically allocate an object and release it when it is no longer needed. Destructors are declared similar to constructors. Thus, they also use the name prefixed by a tilde (~) of the defining class. Destructors take no arguments. It is even invalid to define one, because destructors are implicitly called at destruction time: You have no chance to specify actual arguments.

2.3.4 Multiple Inheritance

The act of deriving a class from two or more base classes is called multiple inheritance. Multiple inheritance is not used frequently, but it can be very handy when needed.

If a class has a default constructor, it is not strictly necessary to call the base class constructor for that class. In most situations, though, you will call the base class constructor for all ancestor classes. Let me give you one other example. In the United States, the Military Air Command (MAC) is responsible for moving military personnel from place to place. MAC is sort of like the U.S. military's own personal airline. Since personnel are ultimately cargo, this requires a military cargo plane. But since people are special cargo, you can't just throw them in the back of a cargo plane designed to haul freight. So what is needed is a military cargo plane that also has all the amenities of a commercial airliner. Look back to Figure 2.9.

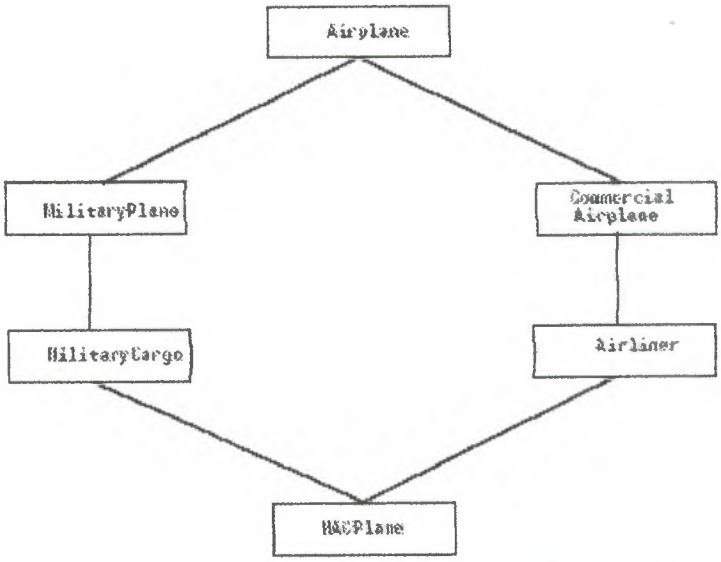


Figure 2.9: An example of Multiple Inheritances

While you may not use multiple inheritance very often, it is a very handy feature to have available when you need it. Classes in VCL do not support multiple inheritance. You can still use multiple inheritance in any classes you write for use in your C++ Builder applications.

2.4 Aggregation

2.4.1 Composition v Classification

In such hierarchies, classes inherit from another classes in order to share or extend functionality. Objects are classified by their position in the hierarchy as 'a kind of' ('AKO') other object – eg an estate car is a kind of car which is in turn a kind of vihicle (figiure 2.10)

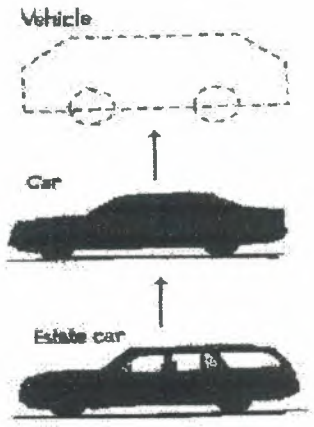


Figure 2.10: In Classification Hierarchies, Each Level is a Kind of the Level Above.

There is another form of hierarchy in object-oriented system, which has been described by various terms, including the following:

1. Composition
2. Aggregation
3. Part-Whole
4. A Part Of (APO)
5. Has-a
6. Containment

This type of composition hierarchy, classes do not inherit from another classes, but are composed of ather classes. What this means in practice is that an object of one class may be have its representation defined by the other objects rather then bythe attributes we have used so far. The enclosing class does not inherit any ettributes or methods from these other included classes, so it is not a classification relationship. Rathar, it is arelationship between objects. An object of the enclosing class is composed wholly or partly of objects of other classes. An object which has this characteristic is known as an 'aggregation'

2.4.1.1 Containment v Containers

A commonly used term for this type of class relationship is 'containment', but there is a semantic difference between this term and the idea of 'continer classes' which also may contain objects of other classes, but do not depend them on term for their representation. Therefore, although they can both be seen as types of aggregation, we should draw an important distinctionbetween containment and containers, as follows:

1. In 'containment', a composition hierarchy defines how an object is composed of other objects in a fixed relationship. The aggregate object cannot exist without is components, which will probably be of a fixed and stable number, or at least will vary within a fixed set of possibilities.
2. A 'container' is an object (of a 'container class') which is able to contain other objects. The existence of the container is independent of whether it actually

contains any objects atv a particular time, and contained objects will probably be a dynamic and possible heterogenous collection.

Figure 2.11 contrasts containment and a container using the example of a car. The relationship between the car and the engine is one of containment, the engine is an essential component of the car. A car object which does not consist parly of an engine object is not a car, since it is unable to exhibit the behaviour expected of one. In contrast, the integrity of the car as an object is not affected by what is in the boot, which is simply a 'container', existing independently of its contents. It may contain a suitcase, a tool kit, 500 bananas or nothing at all, but this does not affect the car object. In practice, there is a range of possibilities between these two extremes of fixed cpmponents and flexible containers where the idea of aggregation may be applied.

Figure 2.11: The Car and its Engine have a Containment Reoation Ship.

2.4.1.2 Abstraction and Aggregation

Both inheritance and aggregation relate to levels of abstraction, but in a different way. In a classificaiton hierarchy, base classes are more abstract than derived classes, particulary if the base class is not appropriate for instantiation. 'vehicle' is a more abstract class than 'bus' for example, 'Animal' more abstract than 'Devon Rex Rabbit'. In a composition hierarchy, 'door' is a more abstract (general) concept than 'door lock'. Remember that abstraction is about the essential features of something –we perceive a door object through its behaviour without necessarily being concerned with its components. In the case of containers, the container itself is more abstract than the object which it contains, particularly if that container is an 'artifact of the implementation' such as a tree or list which has no direct real-world parallel. However, the elements of a composition hierarchy must represent instantiated objects at run time

in order to create an object of the aggregate class, whereas a classification hierarchy may contain many 'pure' abstract classes which cannot be instantiated. For example, to make a 'door' object we must also instantiate objects of classes lock, handle, hinge, letterbox etc, but we do not need to instantiate an object of class 'Vehicle' in order to make a 'Bus'.

2.4.1.3 Properties of Aggregation

Rumbaugh [Rumbaugh et al, 1991] notes that there are certain properties associated with the objects in an aggregation:

1. Transitivity: If A is a part of B and B is part of C, then A is part of C.
2. Antisymmetry: If A is part of B, then B is not part of A.
3. Propagation: The environment of the part is the same as that of the assembly.

2.4.1.4 Layers of Aggregation

Aggregation may well exist in several layers, so that objects are composed of component objects which themselves are composed of other objects. A train, for example, is composed of one or more locomotives and a number of coaches or wagons. The locomotives and coaches/wagons are in turn composed of many smaller components (figure 2.12).

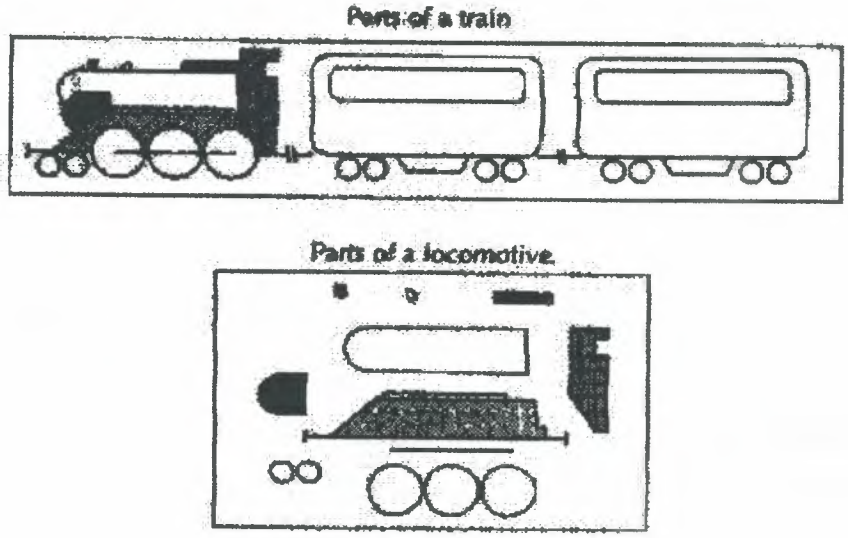


Figure 2.12: Layer of Aggregation.

The objects which comprise parts of a larger object may or may not have an existence independent of the larger object. Aggregation can be fixed, variable or recursive:

1. Fixed : The particular numbers and types of the component parts are predefined.
2. Variable : The number of leves of aggregation are fixed, but the number of parts may vary (like the train)
3. Recursive: The object contains components of its own type, like a russian doll. A more specific example in C++ is the ability for an object to contain a pointer of its own type, allowing it to send messages to other objects of the same class.

2.4.1.5 Partial Aggregations

To say that objects are sometimes, composed of other objects is clearly the case when looking at concrete examples in the real world, but how does it relate to programming, particularly when our objects are often much less concrete than hardware components such as 'wheel','engine','mouse' etc?

The most common use of aggregation in object-oriented programming is in fact a partial application, whereby one class uses one or more objects of another class in order to represent its internal state, but may well have other attributes which are not objects. In many cases these are not the pure 'hardware' aggregations of a parts explosion, where discrete physical components create a larger object, but more abstract collections of objects and attributes which together provide the implementation of a particular class. Often, the contained objects are artifacts of the implementation, objects which are programming tools rather than representation of real world entities. Booch calls this a 'uses for implementation' relationship which is to say that one class uses objects of other classes in order to provide implementation deails for its object methods.

2.4.1.6 Designing Classes Using Aggregation

One of the ways in which aggregation can be useful is in writing classes which contribute towards 'open' object-oriented system. We have stated before that one of the prospective benefits of object technology is the creation of 'software ICs', the sofware equivalent of reusable 'plug in and go' electronic hardware components. However, one of the practical problems which has frustrated this aim is the difficulty of combining elements from different class libraries with their own classification hierarchies.

Inheritance in this context is a rather restrictive device, given the fixed nature of the relationship between classes, and the problem is further compounded by the wide range of potential application environments requiring hardware-specific implementations. How then can reuse be achieved beyond the confines of one class library or one operating environment? It has been suggested that a 'layered' approach may help to achieve this end. This model can be represented by an inverted pyramid (figure 2.13), each level of which represent a category of classes. Object at each level may be constructed from aggregation of objects at lower levels. One aspect of this is that all the components we use, however, simple, are objects, which is a reflection of the Smalltalk philosophy in which everything is an object.

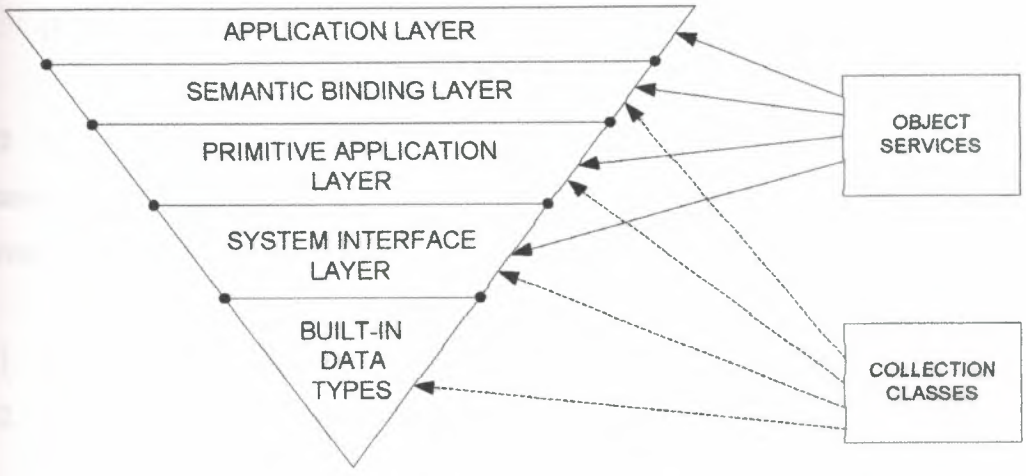


Figure 2.13: Aggregation Models.

Each layer in the model contains classes at a particular level of detail. The layers suggested by Corbett are:

1. Built in Data Types: This is the level which needs to be encapsulated, since it is at the machine representation level that data types vary so much between platforms and operating systems.
2. System Interface Layer: The simple data types are encapsulated at this level as classes in order to be 'open'. The environment-specific representation of these standard data types is hidden behind the interface of the objects. These might include such basic components as strings, enumerated types, booleans, numbers and characters.

3. The Primitive Application Layer: Objects appropriate in scale to be attributes of classes may be modelled here. Modelling attributes as classes in their own right affords the opportunity to add methods to them rather than the enclosing class having responsibility for all attribute behaviour. Classes at this level might be of the scale of name, address, account number etc, composed of types defined in the system interface layer.
4. Semantic Binding Layer: At this level is the aggregation of 'attribute' objects into classes appropriate for applicaiton sized objects. Objects such as Customer, Account and Bank would appear at this level, composed of primitive application layer objects.
5. Applicaiton Layer: At the highest level, we instantiate our large scale objects in the final application.

2.4.2 C++ Syntax

Because there are variations in the way aggregations may be constructed, there are two approaches to their implementation.

1. Objects contain objects.
2. Objects contain pointers to object.

To implement a 'parts explosion' aggregation in C++, classes are defined with objects of other classes inside them. To take an example of a real world object, we might define an aircraft as being composed of a number of fundamental components (figure 2.4.5)

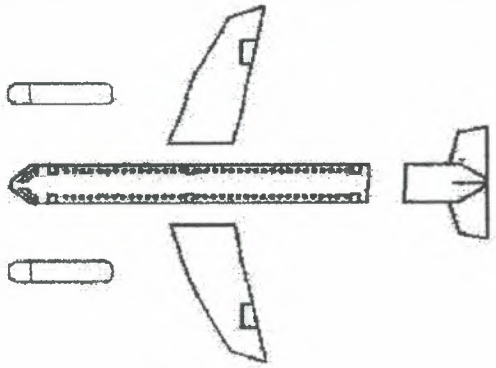


Figure 2.14: A Real World Aggregation.

The aircraft is at this level of analysis is 'fixed' aggregation, because an aircraft object has a fixed set of discrete components. For large scale assemblies, this is typical of the way in which manufacture is broken down between manufacturing plants, possibly in different countries. The intention is , of course, thatthe separetely manufactured components are compatible with each other, allowing the aggregation to function when they are all assembled. This is a useful analogy for the way that software components should be 'composable' into larger systems via compatible interfaces.

A simplified set of major components might be objects of classes PortWing, StarboardWing, Engine, Fuselage and Tailplane. Each of the composing objects would be members of classes defined elsewhere with their own methods. An 'Aircraft' object would be able to call on these methods in defining its own behaviour.

Activities of some composing objects will depend on the states of others. This is an example of 'propagation', whereby the environment of the part is the same as that of the aseembly. In some cases, the behaivour of component objects is constrained by the state of the aggregation.

2.4.2.1 Constructing Aggregations

One aspect of instantiating objects which are aggregations is how to call parameterised constructors of any contained objects. When an object is created, any contained objects must be created at the same time, which implies that their constructor must be caleed. In some cases, none of the composing objects have parameterised constructors, so their instantiation is straightforward. However, If the constructor of contained objects take parameters then we need a syntax which will allow them to be supplied via the constructor of the aggregation. In the following example, a 'car' object is instantiated containingfour 'wheel' objects and one 'engine' object. Since both the 'Wheel' and 'Engine' classes in this caserequire parameters to the constructor of the aggregation using the colon operator.

2.5 Polymorphism

In our pseudo language we are able to declare methods of classes to be virtual, to force their evaluation to be based on object content rather than object type. We can also use this in C++. When using virtual methods some compilers complain if the corresponding

class destructor is not declared virtual as well. This is necessary when using pointers to (virtual) subclasses when it is time to destroy them. As the pointer is declared as superclass normally its destructor would be called. If the destructor is virtual, the destructor of the actual referenced object is called.

The newly introduced operator new creates a new object of the specified type in dynamic memory and returns a pointer to it. The contrary operator to new is delete which explicitly destroys an object referenced by the provided pointer. The various destructor calls only happen, because of the use of virtual destructors. If we would have not declared them virtual, each delete would have only called ~ Colour( ) .

2.6 Operator Overloading

The overloading of operators means making operator symbols which are known to the compiler work with abstract data types defined by the programmer. Operators such as the arithmetic operators '+', '-', '*', and '/' have certain fixed meanings to the compiler- they can be used with the various numeric data types such as int and float.

If we create new data types for use in our programs, then we can also use these symbols to mean operations on our own data types, even though the compiler does not automatically recognise them. This is made possible in object-oriented languages by overloading the meaning of an operator, so that its behaviour can be polymorphic, implementation differently for different classes. This is clearly an extension of the idea of coercion, which allows a single operator to be used with a range of types.

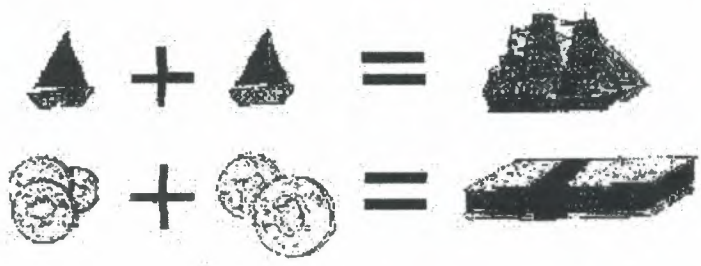


Figure 2.15: A Simple Operator Overloading Examples.

What happens then if we create our own data types, and want to perform operations on them similar to those which are available for the built in numeric data types? In that case we need to explicitly overload the chosen operators so the compiler is aware of

what we expevted to happen when a particular operator is applied to objects of a specific class. For example, if we want to use the addition operator (+) with objects, the compiler will have to interpret statements like:

```
Object1+Object2;
```

It can only do this if we code the mechanism ourselves. In effect, the '+' operator will have to be overloaded to become a method of the class. By overloading operators in this way we can give all the classes inn a system a common interface, allowing us to perform similar operations on a range of different objects (figure 2.15).

2.6.1 Overloading and The Assignment Operator

There is one operator whcih is already overloaded to orkwith objects as wellas simple data types, and that is the assignment ('=') operator.Remember taht we can instantiate an object by making that object's attribute values equla to those of another object which already exist. The syntax, you may recall, is to use the assignment operator:

```
Class_name object2=object1;
```

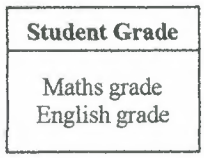
This means that 'object2' is instantiated with the same state as 'object1'. In fact we are not restricted to using the assignment operator in a copy constructor – it has a default behaviour of making on object's attributes equal that of another object. We are able to state:

```
Object2=Object1;
```

And all corresponding attributes will be copied from 'object1' to 'object2'. Although the compiler provides this default behaviour, we may alternatively override this with our own overloaded version of the assignment operator. This is frequently the case where pointers are involved. If the objects are created dynamically for example, then they will be referenced by pointers, and the default behaviour of the assignment operator is simply to redirect pointers. While we may find this acceptable at the object level, there may be pointers inside the object which by default will simply have their address copied to the other object, rather than the data they reference. In such cases, we may well wish to override this default behaviour of redeirecting pointers, and replace it with a mechanism which copies the actual data from one object to another.

2.6.2 Overloading the Addition Operator

Suppose we have created an abstract data type (class) which represent a set of student grades, and create a separate object of this type in a program for each student. Perhaps the objects have just two attributes like this :



Each subjects grade attribute might be an integer which contains a percentage grade. No doubt there would be various methods associated with the class. But what if we wanted to get the average student grade? We would have to add all the student totals together and then divide by the number of students. We might do this by returning the various values from each object and processing them outside the objects, but this would be a bit clumsy and not a very object-oriented approach. How much better it would be if we could say something like:

Grade_total=student1+student2;

And thereby get a total of grades from 'student' and 'student2' stored in 'grade_total'. Since 'gade_total' represents an aggregation of a whole class of objects, it might even reside in the meta-class as a class attribute.

Explicit operator overloading is about this kind of process, whereby an operator's function can be applied to new data types as well as those which are an inherent part of the language. In the 'StudentGrade' class the addition operator would be overloaded to add together the two maths grades to produce a maths total, and add together the two english grades to produce an english total, passing both of these totals to the 'grade_total' object.

2.6.3. The Semantics of Overloaded Operators

In the above example, we are using the addition operator to represent the addition of two objects, which is semantically appropriate for the normal use of the addition operator. In practise, an operator can be overloaded to mean something totally different to the one we might expevt, so that it is possible for instance to overload the '+' sign to

that it would subtract one object from another! Clearly, although this is possible it is very undesirable from the point of view of understandability, so operator overloading tends to be done in the spirit of the operator which is being overloaded. The '+' sign for example is therefore normally overloaded only in the context of adding objects together.

2.6.4 Inheritance of Overloaded Operators

Like other methods, operators which are overloaded in one class inherited by any other classes which derive from it. The exception to this is the '=' operator which, because it has a default behaivour for all objects, can only be explicitly overridden for individual classes and not automatically for their descendants. The userdefined behaivour of any of the other operators may be inherited by descendant classes, but this may well mean that an inappropriate use of the operator is being inherited, perheps more so than when methods are inherited. This is the case because function type methods are often related to single attributes, so that a method returning or setting an attribute. Our addition operator in the 'StudentGrade' example uses all the attributes, and if there were more subjects grades then these would also be part of the process. Therefore, any changes to the attribute set of classes deriving from 'StudentGrade' would require changes to the behaviour of the addition operator.

Let us assume that the StudentGrade class described above represents one set of students who are studying two subjects, but that it also serves as a base class for a class called 'ExtendedGrade'. This class is used to represent students who are stduying a wider range of subjects than just english and maths. For example, they may have a set of five subjects, inheriting two from the base class (figure 2.16)

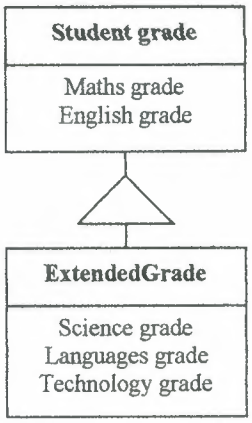


Figure 2.16: One Objects Will Inherit the Overloaded '+' Operaror From Another.

'ExtendedGrade' will inherit the overloaded '+' operator from 'StudentGrade', but this will not be appropriate for adding together objects of the derived class. Although it will work with two 'ExtendedGrade' objects, it can only assign a result appropriate to another object of the 'StudentGrade' class, i.e. it can only generate totals for the maths and english grades. In general terms, although overloaded operators (other than '=') are inherited by derived classes, it is often necessary to redefine their behaviour for all classes in a hierarchy.

2.7 Container Class

2.7.1 Containers as a Form of Aggregation

We mentioned about aggregation, which include a number of object relationships. These may be of three types:

1. Fixed Aggregations: An object is composed of a fixed set of component objects.
2. Variable Aggregations: An object is composed of a variable set of component objects.
3. Containers: An object exists in its own right, but is able to contain objects of other classes.

You may remember that figure 2.17 used a car to illustrate the difference between 'containment' and 'containers'. The engine compartment of a car has a containment relationship with the engine, which is an essential component of the enclosing object. In contrast, the boot exists regardless of its contents, but is able to contain various collections of other objects. You may also recall that one of the characteristics of a container is that it can persist even if it contains nothing.

2.7.1.1 Containers in General

There are containers all around us, and without them life would be very difficult, since containers allow us to collect and organize numbers of other objects. Figure 2.17 shows a few containers of various types.

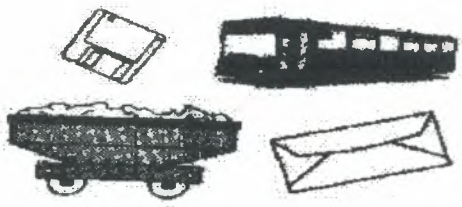


Figure 2.17: Some examples of Containers Which Can Hold Collection of Objects.

A bus, for example, contains passengers, a coal truck contains coal, and envelope may contain a letter and a diskette may contain data files, perhaps with directories providing another level of aggregation. Just from these examples, we can see that there are differences in the characteristics of different containers. Some containers have limitations on what objects they may contain, so that a coal truck might only ever contain coal, whereas an envelope may contain a wide range of items provided they are of a suitable size and weight. Some containers have limited mechanisms for adding and removing objects; People must enter a bus through a door in a particular sequence, which may be dictated by ticket numbers, whereas coal is loaded in a rather more informal manner! The organization of objects within the container may vary too. For example, the organization of files on a diskette must be very strict for us to be able to add files, remove them and access the data on the disk, while a pile of groceries in a shopping trolley has no particular order or control over access. Another aspect of containers is any limitations they may have on the number of contained objects. For some containers the maximum number of contained objects is fixed; Buses have a fixed passenger capacity and diskettes have a fixed size of data storage. In contrast, a bin liner always has enough available expansion for just one more soggy tea bag!

2.7.1.2 Containers in Software

In this chapter, we will look at some containers which may be implemented I ware. These range from the simplest types, such as containers for a fixed number of objects of a single class, to more complex structures which may collections of objects of many classes.

A container in an object-oriented program is usually instantiated as an object of a container class. A container class object encapsulates inside it the mechanism for containing objects of other classes, and providing the necessary behaviour for adding, removing and accessing the objects it contains. Incidentally, the use of an 'amorphous blob' to represent an object, found in a number of illustrations in this topic, is adapted from the design notation of Grady Booch [Booch, 1991].

Like all classes, a container class gives us the opportunity for reuse, so that container objects can be usefully instantiated in many programs. This frees us fruit having to recreate complex data structures in every programs requiring, the management of a collection of objects.

2.7.2 Container Types

There ate many different types of container, and container class libraries provided with most compilers have large hierarchies of container classes various behaviours and internal implementations. C++ has no standard container classes so they will vary between vendors. We can, however, build our own container classes quite simply. Containers appear in all object-oriented languages, but in all cases we can generalized and contrast a few of the major types of container and their characteristics. Most libraries have a range of class definitions, some encapsulating simple data structure, such as arrays and strings, others providing generic types of collection and some which provides specific access protocols. The following table indicates some of the which tend to be modeled as classes in container class libraries.

Container type	*Ordered?*	*Fixed size?*	*Duplicates allowed?*	*Can contain different type?*
Array	Yes	Yes	Yes	Yes
String	Yes	Yes	Yes	No (char only)
Set	No	No	No	Yes
Ordered	Yes	No	Yes	Yes
Sorted	Yes	No	Yes	Yes

Stack	Yes	No	Yes	Yes
Queue	Yes	No	Yes	Yes
Bag	No	No	Yes	Yes

Table 2.1: Classes in Container Class Libraries

Container classes like these are abstract concepts which are implemented using traditional data structures-such as arrays, lists, trees and hash tables. Once they are encapsulated into classes, however, we do not have to be aware of their implementation details. There are also a lot of containers in the table because some definitions are to do with structure some with contents and some with access methods. It is possible, for example to implement a stack using an array to contain objects, which comprise a set.

2.7.2.1 The Array

Arrays have many advantages; Being instances of a data structure built into the compiler they are easily declared, easily referenced and relatively easy to handle in a program. Since the referencing is done independently of the contained objects, it is no problem to have duplicates contained, and we can use base class pointers to allow an array to container objects of different (derived) classes. This of course moans that the array contains the pointers to the objects, not the objects themselves. The only major drawback of the array is that it must be of a fixed size; as we know, an array can only be declared with a given size it cannot be dynamically resized once declared. It can of course be dynamically allocated at run time, but once it has been created its size is fixed. The only way of resizing an array at run time is to create a new array of the required size and then copy into it all the data from the original array.

In an object-oriented system, arrays are often modeled as classes rather than simply used as data structures. This encapsulates the mechanisms, which directly manipulate the array behind a simpler object interface.

2.7.2.2 The String

It may seem odd to call a string 'a container, but of course this is how we model a string in C++, as a container of individual character. We have also used arrays to implement

this set of characters, so it is perhaps true to say that a string is an abstract container, modeled using a concrete instance of a container. In other words, a string is a concept, which exists in our own minds, but we model its behaviour by using a physical data structure. This combination of the abstract and the concrete occurs again and again with containers, we are modeling conceptual containers, but in order to do so we have to at some level define a physical data structure which is able to implement the required behaviour; All the containers listed in the table have to be created using other forms of structure. The string in fact is one of the most popular candidates for making into a class, since objects of a 'string' class may find use in almost any program.

2.7.2.3 The Set

The key aspect of a set is that there can be no duplicates in it, and there is no concept of order in the collection of contained items. We might regard this as important when creating objects which must be unique in terms of 'their state data. It is possible, of course, to implement a collection of pointers which all reference the same object, so the construction of a set requires certain restrictions on what objects may be put in it. One-way of defining a set is to say that it will discard duplicate additions, so that one might be used for example to collect the uniquely occurring words in a document.

Because the set (like the string} is a concept rather than concrete data structure, it has to be implemented using whatever data structures are appropriate. Since the workings of a container class ob object are always encapsulated behind its public interface, the choice of algorithmic solutions does not impinge on the user.

2.7.2.4 The Ordered Collection

An ordered collection is a very general type of container which could be modeled using a range of 'internal' implementations. Since it is ordered, there simply has to be some mechanism for referencing object a in the container, in the required order, usually the order in which the objects were originally added. Unlike an array, there is no fixed size to an ordered collection. It is worth bearing in mind that certain operations are possible on ordered collections, which are not possible on unordered collections, such as sets. With an ordered collection we can access elements according to their position, such as finding the first or last element, operations which would clearly not apply to unordered collections.

2.7.2.5 The Sorted Collection

A sorted collection is simply an ordered collection, which is sorted on some key. Again, this is a very general idea which may be implemented in various but one important aspect of it is the way its sorting mechanism work. For example, if we are to put objects into a sorted collection we have to make sure that objects are in fact storable, perhaps by being able to respond to an overloaded '>' operator.

2.7.2.6 The Stack

The stack is a classic data structure which may be based on an array or an ordered collection. What defines a stack is not so much its internal structure but its access methods. A stack is a 'LIFO' structure, which means that objects are put into a stack object in order and only retrievable from it in reverse order (figure 2.18)

2.7.3 The Use of Containers

The primary value of container in an object-oriented program is that they give use control over collections of objects, particularly dynamic objects that are unpredictable in terms of their persistence. When we want to model objects in a varying collection at run time, we need simple mechanism for creating accessing destroying them without having to explicitly deal with the programming algorithms, which allow dynamic objects to be handled. A container class takes responsibility for managing our collections of objects, and all we have to do is to use the methods provided for the container. A typical container might have the kind of methods shown below, allowing us to create new objects manipulate the objects which happen to be in the container at a particular time and remove objects from the container, perhaps to destroy them. This type of container is fairly generic, alt kinds of object collections can undergo these operations.

Container
Add object
Remove object
View object
Find object
Sort object

2.7.3.1 Iterator Methods

We have talked about a number of types of object method in previous topic:

Selector methods, modifier methods, constructors and destructors. With container classes, we also have what are known as 'iterator' methods, which allow us to iterate through the contents of the container in order to access all the objects inside it. This process is also known as 'enumeration' [Cox, 1986] though it is important to draw a distinction between this and the 'enumerated types' which may be implemented in C++. Many iterator methods rely on alt the contained objects being able to respond polymorplilcally to the same messages; For example, we might have a container of graphics objects which are to be drawn on the screen. An iterator method would simply address all the objects in the container in turn, sending the message 'draw' (figure 2.18).

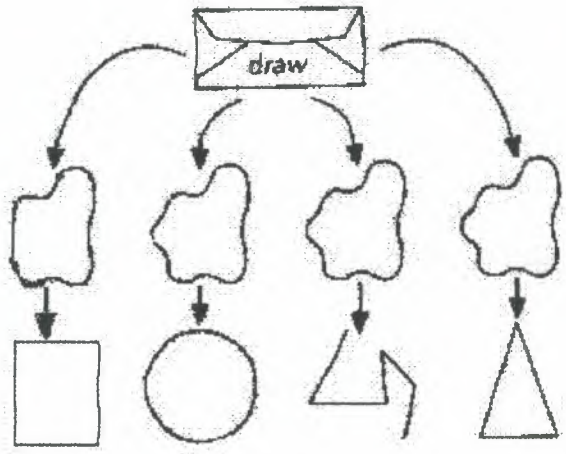


Figure 2.18: A Collection of Graphics Objects are All Sent the Message 'draw'.

Other iterators might destroy all the objects in a container, or scan all the objects looking for a particular one. The 'sort objects' method in the example container class outline would be a type of iterators, since all objects would need to be accessed in order to be sorted.

2.7.4 Linked List of Objects

As we have stated, an array is not the most flexible mechanism for implementing containers. A better approach in many cases is the linked list, which allows the size of the container to vary as objects are added and removed, since it does not necessitate the

storage of objects in a fixed physical sequence. A list is implemented using pointers, one pointer reference. The first object in the list, and then each object points to the next object in the list.

A doubly linked list is where each object has two pointers to reference the objects on either side of it, and two pointers are used to reference the two ends of the list. This makes certain operations such as deletions easier to accomplish.

The next example program demonstrates the certain of a singly linked list of objects of a single class. This is what is known as an 'intrusive' list because the objects themselves have to point to each other, and therefore must contain a pointer of their own class. The contained objects are contain a (private) pointer as follows:

```
ListObject* previous_object;
```

They also be methods to direct this pointer and return the object, address it references. For the purposes of identification when the program runs, object also has a single integer attribute. At this stage, we are not modeling a container class, simply demonstrating the implementation mechanisms for putting objects in lists. The program itself simply instantiates five objects and adds them to the list in turn, iterating through the list at the end to display their attribute values.

2.7.4.1 Iterating Through a List of Objects

In order to iterate through a linked list of objects, each object must be accessed via the object which is pointing to it, starting with the head pointer and terminating when a NULL pointer is reached.

2.7.5 Creating a Container Class

A container encapsulates its implementation behind its object methods. To build a container class, we need to hide the data structure used to manage the collection of objects. It is also preferable if being in a container does not impinge more than necessary on the contained objects. Figure 2.19 (overleaf) shows how a non-intrusive list is organized; the objects themselves do not contain pointers to other objects of a 'link' class. It is these links, which point to each other, and the head pointer always references the last link in the list, not the last contained object.

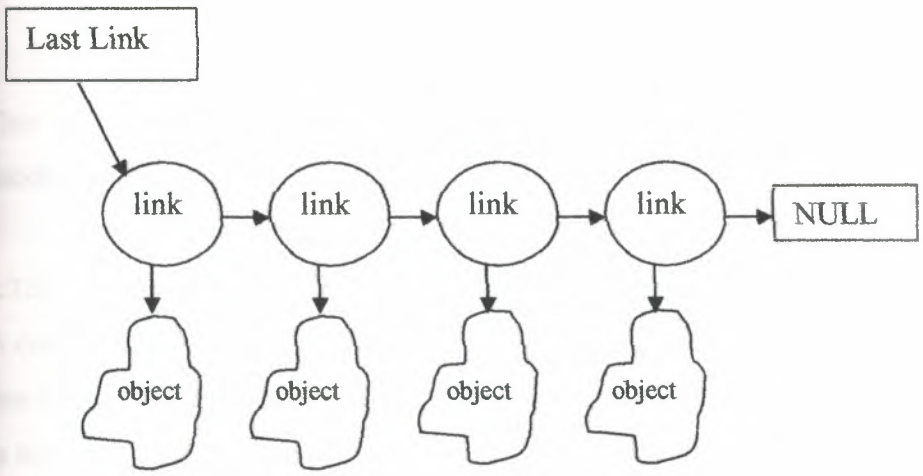


Figure 2.19: A Non-Intrusive List.

There are three classes used in the container class:

1. List Object: The class of the objects which container.
2. Link: The class of the links which inside the container pointers, one to point the another to reference a class.
3. Container: The container class, which objects of class 'ListObject' and manage them using 'link' objects.

2.7.5.1 Adding a New Object to a Container

Since containers generally use pointers to reference the contained objects, adding a new object means passing a pointer to the container via a method. We can do this by instantiating an object outside the container and then passing it as a parameter to a suitable method. This example assumes a container called 'a_container' with a method called 'addObject' which takes as a parameter a pointer of the class 'Object'. We might add a new object like this:

```
Object temp;
temp = new Object;
a_container.add(temp);
```

However, this is rather clumsy, since we are instantiating an object outside the container when it needs to be inside. A better way is to use 'new' inside the argument list of the

method itself, as follows:

```
a_container.addObject(new Object);
```

This will instantiate the object as part of the method call, and remove the need for another pointer outside the container.

2.7.5.2 Deleting an Object in a List

A container, which only allows us to add objects, is rather limited, so our container also has a method to remove objects. Unfortunately, deleting objects from a singly linked list is not very easy. If we simply destroy an object which is in a list, theft the object which it references will be lost, along with any other objects further down the list. Therefore the deletion of an object requires some redirection of pointers.. One way of approaching it (though not the only way) is to handle the removal of the object at the head of the list differently to the removal of objects further along the list. This is because removing the object referenced by the head pointer is relatively simple.

We can see from the code that this is not very easy to express in C++. A doubly linked list provides an easier means of deleting an object, because that objects references the objects on either side of it. This means that both the adjacent objects' pointers can be accessed and redirected. Of course the trade off is that it is a little harder to 'implement a doubly linked list in the first place.

2.7.6 A Container for Heterogenous Objects

The container class described in the previous pat of this chapter is able to contain objects of the 'ListObject' class. It could also, of cours, contain objects of any other class derived from 'ListObject'because, as we know, a base class pointer can be used to instantiate, reference and destroy objects of any derived class.

A queue such as this might be used for example in a simulation of a toll booth on a bridge or motorway to estimate waiting times with different traffic flows. The container is implemented here using an array for simplicity, though other data structures could easily be used.

If we are going to use a base class pointer in a container to reference different objects at run time, then we must make sure that all the objects put into the container have a consistent interface, with polymorphic methods which can be dynamically bound. In the example program, the 'showDetails' method is defined separetly in all three classes and made 'virtual' so that it too can be be dynamically bound. Not only is this necessary for memory management but in this case the derived calss destructors have additional functionality which depends on them being bound at run time.

2.7.7 Container Classes Using Templates

In polymorphism, we introduced the template as a way of creating generic function in C++. A generic function is, you may remember, one which is able to process parameter arguments of different types with a single implementation.

It is also possible to create generic classes, which may be instantiated by parameter of different types. Typically, classes instantiated in this way are container classes, with the class of the objects to be contained represented by a generic type name. The syntax for a container class is- as follows:

```
template <class type_name>class class_name
{
// class definition
```

The use of the 'template' and 'class' keywords has been seen before in the definition of generic template functions, likewise the pointed brackets around the type name. If we wanted to create a generic stack class, then, we might declare the class thus:

```
template <class T> class Stack
{
// etc.
```

The stack class wilt be used to instantiate stack objects, and the classes of objects which the stack may contain are not predetermined by the class definition.

Whenever, we wish to refer to the data type or class of the elements which are to be contained on the stack, the alias 'T' is used. A generic pointer for example would be:

```
T* a_pointer
```

2.7.7.1 Method of a Template Class

Since we will probably wish to define a number of methods for the container class, we also need to know the syntax for defining a method of a template class out of line. This takes the following rather convoluted form:

template <class type_name>return.type class_name <type_name>::
method_name (parameter list ..)

It looks a little simpler in practice. If we assume that our stack class has a 'push' method then the declarator (the first line) looks like this:

template <class T>void Stack<T>::push(T object_in)
{
/ / method definition

2.7.7.2 Instantiating a Template Class Object

Since an object-of a template class is instantiated according to some other data type being provided, the constructor call is a little different to that normally data type which is referred to as 'T' in the class definition must be given after the name of the class, using the familiar pointed brackets:

Class_name<data_type>object_name(parameter list ..)

For a 'Stack' object instantiated to contain a maximum of 20 integers, the constructor call would be:

Stack<int>aStack(20);

2.8 Persistent Objects, Streams and Files

2.8.1 Object Persistance

In topic 'object lifetimes and dynamic objects', we discussed the persistence of objects within a program. The lifetime of an object can vary from a momentary existence inside the body of a function or method, to persistence for the life of the program. However, in all case we have only looked at objects, which exist at run time. None of previously instantiated Objects have been able to persist after the program has finished running, but in practice there are three levels at which objects may persist:

1. Objects persisting during a run of a program.
2. Object, persisting between runs of a single program.
3. Objects persisting between different programs.

Objects which only exist while a program is running are known as 'transient objects, they have no existence independent of a single program run time. Those whose lifetimes extend beyond the boundaries of a single program run are known as 'persistent objects'.

2.8.1.1 Storing Sbjects

In order for an object to be persistent, it must be stored on disk in some form. The problem with objects is that they do no fit easily with the traditional formats of stored data. As we have previously discussed, traditional to programming separate processes and data, so that data is easily stored inpmndent1y of any associated processes. This is not the case with objects, because objects have two aspects:

1. The data associated with attributes.
2. The processes associated with methods.

An object's attributes are unique to that object and therefore may be stored as a set of data similar to a record in a traditional file. However, methods are part of the class, shared by all other objects of the class, these are not so easily stored. If an object is to retain its integrity, both its state and its class need to be stored on disk not just its state alone. We therefore have to implementation of persistent objects:

1. Store attribute data independently of methods. This means shifting from object-oriented approach to a more traditional file based approach storing object data, and back again when re-loading it.
2. Use an object-oriented database.

2.8.1.2 Storing Objects in Traditional Files

It is not possible to store all aspects of an object when using traditional types of file organization, but it is possible to maintain 'pseudo-persistent objects' by storing attribute data in files. While a program is running, objects can write their state data out

to disk as a record or series of records, and then this data can be reloaded later during another run of the program. Semantically, we do not really have a persistent object, because only the state of the object has been stored. As we know, an object comprises three parts, state, identity and behaviour. The state can be saved, and the behaviour can to an extent be maintained by the class definition, but the identity of an object is rather different. In fact, when we rebuild an object from state data stored in a file, we are recreating another object with the same state as the original, rather than maintaining the existence of the original object. However, for most practical purposes this kind of data storage is adequate, though it puts the onus on the programmer to ensure that objects retain their integrity when their classes are represented in source code and their states are saved elsewhere in data files.

2.8.1.3 Object-Oriented Databases

Object-oriented databases allow us to store both the class and state of an object between programs. They take the responsibility for maintaining the links between stored object behaviour and state away from the programmer, and manage objects outside of programs with their public and private elements intact. They also simplify the whole process of rendering objects persistent by performing such tasks invisibly.

As well as recognizing that persistence has to do with time we should be aware that it also has to do with space. Object-oriented databases have developed to address two rather distinct needs:

1. Providing databases which can be used to generate applications in their own right.
2. Providing persistent objects for applications generated in other programming languages.

Between these two extremes lie a number of variations and range of definitions as to what might constitute an object-oriented database. The mast important characteristic of an object-oriented database is that it is able to store objects 'whole', rather than disassembling them into constituent data sets.

In terms of the way in which these databases work, there are two general types; those

which have taken an evolutionary path, extending traditional relational databases to handle objects, and those which are fully object-oriented from the ground up.

There is no single model for object-oriented databases, so the interpretation of what constitutes an Object Database Management System (ODBMS) is dependent on a particular vendor's approach. There are a number of general comparisons, which can be made between various database models [Chauhri, 1993]. In one respect there are a range of data models, and also a range of architectures. These include object managers, extend database systems,database programming languages and relational object shells.

There are many issues involved in the debate about the value of object-oriented databases and the ways in which they may be implemented, but from our point of view their role is simple-storing the objects we create in programs. If we do not have an object-oriented database at our disposal, then storing and retrieving objects is that much harder.

2.8.2 Stream

A stream is a general term for a data flow, which may be to and from a file, or to and from screen and keyboard, or to and from other 'sinks' and 'sources' of data. An object-oriented stream library contains a number of classes, each of which is appropriate to a different kind of stream. In C++, the 'if stream' class for example provides us with appropriate methods for managing input disk files.

The 'iostream.h' stream library defined by Stroustrup [1991] is provided as a simple basis for larger stream libraries provided by compiler vendors. As such it is fairly limited, but does provide enough for the storage and retrieval of object attribute data. The library is intended to be and used both with simple data types and user defined types. It is also based on the idea of stream being redirectable, so the syntax which we have used to accept keyboard input and produce screen output is easlly applicable, to file I/O. All stream classes are in a classification hierarchy based on the 'ios' class, and provide all the objects, operators and methods needed for console I/O and file handling.

2.8.2.1 Stream Operators and Methods

The operators and methods defined by Stroustrup for use with stream objects are as

follows:

Output: << (insertion operator)
put(char)
Input: >> (extraction operator)
get(char&)
get(char*, int, char)

These may be used to handle various types of data such as single characters, numeric data types, strings, strings with embedded spaces and object attribute data.

2.8.2.2 Stream Output

In previous examples, we have seen many statements like the following:

```
cout<<"hello"
```

but where does this syntax come from? In fact, 'cout' is an object of the 'ostream' class (as defined in iostream.h) and '<<' is an overloaded operator of that class able to handle a range of data types. We can further overload this operator to output our own data types. Alternatively, the 'put' method puts a single character into the output stream.

2.8.2.3 Stream Input

Like 'cout', 'cin' is an object, but in this case it is an object of the 'istream' class which relates to the input stream. As you might expect, '>>' is an overloaded operator of the istream class. Again we can overload this operator to allow the input at objects of user defined types (classes).

When using the input the input stream, remember that the 'cin' object using the '>>' operator treats whitespace as a delimiter, so that more than one item can be input in a single statement provided they are separated by white space:

```
int x, y, z;
cin>>x>>y>>z;
```

This will read a sequence of three withespace-separated integers. This is a bit limiting, since we sometimes want to input text which includes embedded spaces. In such cases

we can use the 'get' method of the 'istream' class. 'get' comes in two versions:

1. The first version of 'get' can be get a single 'char':
   prototype: istream& get(char& c);

2. The second version of 'get' a string which may contain spaces. The default terminating character is the newline ('\n') but this can be overridden by another terminating character to accept multi-line input if required:
   prototype: istream& get(char* p, int n, char='\n');

When 'get' is used to read a string of characters from the input stream, it always leaves a '\0' character at the next position stream position. This can lead to problems when we next want to read character or string because the '\0' character will automatically terminate the next attempt to read from the stream. Therefore we often need to read past this terminating character with a single char 'get' before reading the next set of data.

2.8.2.4 Checking Data Types

Since we often wan to do something with input data such as analyses its type for processing or error detection, it is useful to be able to identify the data type of a character. This can be done by standard functions available in a standard C header file called 'ctype.h'. The most useful are:

Function	***Returns TURE (1) for:***
int isalpha (char)	a – z or A - Z
int isdigit(char)	0 - 9
int isspace(char)	Space, tab ('\t'), carriage return ('\r'), newline ('\n'), or formfeed ('\f')
int isasci(char)	In the ASCII table (ie >=0 and <=127)

Table 2.2: Checking Data Types.

Both 'ostream' and 'istream' are derived classes of the 'io,' class. This class provides us with some useful methods, including these:

Method	Purpose
int width(int) char fill(char) int precision(int) long setf(long)	Sets a field width – minimum only (will not truncate) Sets a fill character Sets the precision of floating point numbers Sets the format of a field

Table 2.3: Method / Purpose.

2.8.2.5 Manipulators

One problem with the 'io' format specifiers is that each formatting statement has to be made separately from all the others. This means that there is no logical connection between the separate operations. As an alternative approach, we can we 'manipulators' to put the formatting statements directly into the input or output operations. Manipulators are defined in a file called 'iomanip.h', and they have a similar role to the ios format specifiers. Although their functions do not exactly match with the formatting methods of the ios class, many are similar, as the following table shows:

Manipulator	Purpose
Setw(int) Setfill(char) Setprecision(int) Setiosflag(long)	Sets a field width – minimum only (with not truncate) Sets a fill character Stes the precision of floating point numbers Calls an ios formatting method

Table 2.4: Manipulator / Purpose.

2.8.2.6 Overloading 'istream' and 'ostream' Operators

In order to handle the input and output of objects, we need to able to treat them in the same way as other data types. In general terms, to overloaded the insertion operator ('<<') the method prototype would be:

```
ostream& operator<<(ostream&, class_type);
```

Where 'class_type' is a parameter of some user-defined class. The method returns a reference to an 'ostream' object, and also takes one as parameter. The extraction operator ('>>') can be similarly overloaded, but note that the parameter of the class for which the operator is being overloaded must be passed by reference, not by value as it is

for the insertion operator:

istream& operator>>(istream&, class_type);

2.8.3 Files

The most important classes for file handling in the stream class hierarchy are:

ifstream (input files)
ofstream (output files)
fstream (files for both input and output)

To open a file for output (writing), then, we instantiate an object of the 'ofstream' class. Similarly, a file is opened for input (reading) with an object of the 'ifstream' class. In order to open a file, an object of the appropriate stream class must be instantiated, and assocoated with a disk file name. These activites may be done in a single statement or separately, by creating the object first and then assigning it to a filename.

2.8.3.1 Opening Files

The two forms of the syntax for opening files are as follows:

1. Firs Syntax version: This version instantiates the object and opens the file in one statement:

   stream_type object_name (filename, opening_mode);

We do not have to state the opening mode, since it has default values. The default value for an object of the 'ofstream' class is for the file to be opened for output (writing).

2. Second Syntax version: This version instantiates the object in one statement, and uses the 'open' method to open the file:

   stream_type object_name;

The opening mode is defined by an integer value, but is usually specified via an enumerated type in the 'ios' class which allows us to name the opening modes using the scope resolution operator. The possible opening modes are as follows:

Opening mode flag	Effect
ios::in	open for input (dafault for ifstream)
ios::*out*	*open for output (default for ofstream)*
ios::ate	open and seek end of file
ios:app	append all output
ios::trunc	destroys contents of existing file by truncating it to position 0
ios::nocreate	open fails if the file does not exists
ios::noreplace	open fails if the file exists

For example, let us suppose that we want to open a file for output, but ensure that all new output is appended to the end of any existing data in the file. In this case, we would want to open the file to append as follows:

ofstream outfile("test2.dat", ios::app);

We may find that we need to specify more than one opening mode. This can be achieved by 'or'-ing modes using the '|' character. This example shows how an output file can be opened to truncate the file but for the open to fail if the file does not already exist:

Ofstream outfile("test.dat", ios:trunc | ios::nocreate);

However, this is only necessary if the file object does not fall out of scope at the point where it must be closed. Otherwise the file object destructor will close the file automatically.

2.8.3.2 'ostream' File Methods

Two of the methods of the 'ostream' class allow us to manipulate the file pointer of an output file. The methods are:

Seekp(streampos);

Tellp( );

Where 'streampos' represents the character position in a file. 'seekp' moves the file pointer to the given position, whereas 'tellp' returns the current position.

2.8.3.3 'istream' File Methods

With an input stream, we can look at the stream position referenced by the file pointer. The first method determines the position of the next character to be reed without

actually reading it:

int peek( );

These two methods are the corollary to 'seekp' and 'tellp' in the ostream due, because they move and report the pos Won of the file pointer respectively:

Seekg(streampos);

Tellg( );

2.8.3.4 Objects of the 'fstream' Class

Objects of the 'fstream' class inherit from the 'iostream' class. 'iostream' multiply inherits from both istream and ostream so an 'fstream' object has all the methods of both istream and ostream objects.

An 'fstream' object has separate file pointer positions for reading and writing we may for example, use both 'seekp' and 'seekg' with an fstream file. It can opened in a range of ways, so we often need to specify more than one opening mode. This file for example is opened for both input and output:

fstream in_and_out("iofile.dat",ios::in | ios::out);

2.8.3.5 Detecting the End Of File(eof)

When handling input files, it is essential that we are able to detect the end of file. This may be done with method of the ios class called 'eof' which returns 1 (true) when the end of file is detected.

2.8.3.6 File Handling With Character i/o File Pointers

The following program demonstrates an 'fstream' file being used for character I/O. It indicates how the file pointer may be used, and how single characters may be written to and read from files. Although objects are written and read in a rather different way, manipulating a character at a time is a useful and flexible way of handling data.

In this case, a string is written to file, and then the file is before the data is read back In and displayed. Be aware, however, that not all compiler's stream libraries behave in exactly the same way.

2.9 Object-Orienred Analysis And Design

2.9.1 The Need for Analysis and Design Methods

In the first chapter we discussed the software crisis and the need for some means to handle the development of large complex systems. Object-oriented programming alone is not enough, it has to be used the context of a coherent overall design which can integrate all the objects identified in a system together. Indeed, object-oriented programming alone can only add to the problem if it is not applied in the context of a semantically appropriate systems analysis.

We can only master complexity by putting it into a formal framework, which allows us to concentrate on small parts of a system while being confident that each part will relate appropriately to the other system components. Ultimately, however, we need methods because real problems are not easy to solve. The scenario presented later in this project may be simple, but any real world system would of necessity be infinitely more complex

2.9.1.1 Components of an Object-Oriented Analysis and Design Method

While there are many different methods for object-oriented analysis and design, they all have to address the fundamental components of any object-oriented system. Any method must address (at 1east) the following:

1. Identifying objects.
2. Identifying classes (classifying the object).
3. Defining object behaviours (methods).
4. Structures using generalization and specialization (inheritance).
5. Structures using aggregation (composition, containers).
6. Object state representation (attributes, and events which change state).
7. Message passing between objects (association and visibility).

The means we use to tackle these problems can very between methods, and there are two schools of thought; One which stresses the behaviour of objects and another which stresses state. This divide between the 'responsibility centered' and 'data centered' approaches mirrors in some ways the traditional divide between data driven and process

driven structured methods. However, there are many common elements between the approaches advocated by different authors. Indeed, some developers of object-oriented analysis and design methods such as Booch and Rumbaugh are keen to stress the similarities between their approaches rather than the differences, and match' elements of different methodologies.

2.9.1.2 The Difference Between Analysis and Design

One of the claims which is made for object-orientation is that it smoothes the transition between analysis and design, and between design and implementation. Coad and Yourdon in their analysis and design text use a simple 'chasm' diagram to imply that in traditional approaches, each of these three stages is discrete, to the point where the transition from one to another is difficult (figure 2.20).

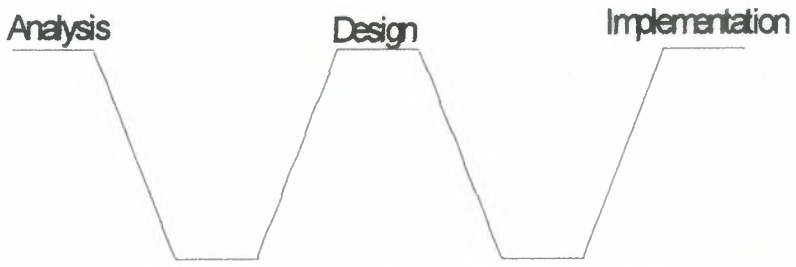


Figure 2.20: 'Chasm' Diagram Adapted From Coad and Yourdon.

With an object-oriented approach, the analysis and design cycle can be seen as a single process, albeit one which has different concerns at different levels of detail. We may contrast traditional analysis and design approaches which use different models at different stages of development with the object-oriented approach which progressively expands the same model of a system. Booch talks about the process as being a 'round-trip gestalt' - a continuous iteration through the stages of analysis and design, including the use of prototyping and refinement of the existing design as necessary.

In object-oriented analysis, we are attempting to create a model of problem by identifying the objects, which exist in the problem domain. In the design phase, we are defining how these abstractions can be made to exist and interact in software by creating an overall framework for the system. Depending on the scale of the problem, these activities may merge into seamless process.

2.9.2 Classifying Objects

Since the first step in any analysis must be to identify objects which must to classes, then the way in which classification is done is important. As Booch explains, there are only three general approaches to classification:

1. Classical Categorization
2. Conceptual Clustering
3. Prototype Theory

In all three we look for common properties and behaviours, but in increasingly abstract ways. In classical categorization for instance it is important that the properties of objects are measurable, so that it is clear to which category an object belongs, but conceptual clustering involves less easily measured similarities. Occasionally we come up against abstractions, which cannot be clearly defined either in terms of their properties or concepts, a game is an example of this, what exactly defines a game? This is the area of prototype theory, where we can only look for 'family resemblances' between one abstraction and another. All forms of categorization are, of course, always going to be domain specific, since nature does not categories, it is purely a human activity which attempts to make sense of our environment Therefore something which is a valid category in one context may not be in another.

2.9.2.1 Identifying Classes

When we apply various tools to the analysis of a system, we are frequently applying some form of classification to the objects identified in the system. These may well be unconscious responses by analysts and users, and it is important that any assumptions about the classification of objects are identified and, if necessary, challenged. Remember that our model of the system is not right or wrong, but simply one way of analyzing problem any classifications we apply might equally be replaced by different but equally valid ones. Certainly when we reach the later stages of design and maybe have to decide whether to use classes in complex hierarchies, delegations, mixins and aggregations, then we may well find that a number of different classifications are possible.

2.9.2.2 Defining System Boundaries

The first step in any analysis must be to define the system boundaries, to exclude from our design any aspect of the overall application, which is not directly part of the system. A 'user' is a typical example of an 'object' which is dearly outside the system we arc trying to build Although we will need to be aware of the messages being sent to and from a 'user', There is no reason to model the user as an object. Before we can begin to successfully analyse a system, we should understand that some of the 'objects', which we talk about in the analysis, are external and will therefore not become objects in the implementation.

Given that one of the primary elements of an object-oriented design method is to encompass reuse, it is important that this aspect should be emphasised at an early stage. Whether, or not we are able to reuse software components at the implementation stage, there is no reason why reuse should not be applied at the analysis and design stages. Domain analysis is simply a detailed study of the problem area, and one aspect of this involves looking at similar systems to the one we are analysing and finding out what objects have been identified in them. Since there are not so many different types of application, the chances of finding a useful precedent are quite high. It would be re-inventing due wheel, for example, to approach a bank cash machine application without reference to the many existing similar systems.

Other sources, of information may be developers, users and libraries. A domain analysis might take the following step:

1. Consult with domain experts to get a general model of the system. A 'domain expert' is not necessarily -a software engineer but someone who knows the application area, perhaps as a user. A domain expert in cash machine systems might simply be someone who spends a lot of cash!
2. Look at existing systems in the problem domain.
3. Compare existing systems for similarities and differences.
4. Refine the original model in the light of this comparison.

How much access we might have to existing systems depends on a particular context. Some developers may have access to other similar applications written in the same

company or organization, in which case a great deal of information can be accessed. In other cases, very little domain information may be available. Domain analysis does not have to depend on information supplied by people written sources can be equally useful.

2.9.2.3 Actors and Use Cases

Many object-oriented analysis methods start from the problem statement, but do not suggest how that problem statement might be arrived at. Therefore one aspect of OOA&D which is rather weak is that of requirements analysis. The use case model is based on scenarios provided by 'actors' at a very early stage of the analysis, which collectively describe what the system is intended to do. An actor represents something outside the system which interacts with it, and is generally a role adopted by an end-user. These scenarios are then analysed using 'storyboards' to identify the objects in the system, their responsibilities and how they collaborate with other objects.

2.9.2.4 CRC Cards

A similar approach to the use-case storyboard is the CRC card, used by a number of authors including Wirfs-Brock and Beck & Cunningham. CRC stands for 'Class/ Responsibility / Collaboration', and is a useful way of 'brainstorming ' the application. A CRC card is simply an index card or sticky note divided into three sections: the name of the class, its responsibilities and its collaborators (figure 2.21). A responsibility is written as a short verb phrase containing an active verb phrase containing an active verb, such as 'turn on the pump', or 'print the account balance'. A collaborator is an object which will send or be sent messages when responsibilities are carded out.

C.R.C. (Class, Responsibility & Collaboration) Card

Class: AIRCRAFT	
Responsibilities:	Collaborators:
Take off Land	*Airport* Pilot

Figure 2.21: A Typical CRC Card.

Because CRC cards can be moved around, torn up, replaced and written on, the are a

very flexible tool for analysis. Developers can 'anthropomorphise' the behaviours of objects by 'becoming' a particular card and walking through scenarios. 'It is not unusual to see a designer with a card in each hand, waving them about making a strong identification with the objects while describing their collaboration. By defining what an object does, and when it needs to do it, we can begin to define the required behaviors (methods) of an object, and thecontexts in which it needs to send or receive messages to and from other objects. The use of CRC cards can be easily integrated into other methods alongside, for example, text analysis.

The approach also allows cards to be physically arranged so that inheritance and aggregation hierarchies begin to arise naturally from the physical location of cats. Cards representing closely collaborating objects can be overlapped, parts of an aggregation can be arranged below the enclosing object, generalizations at the higher levels of a hierarchy can be placed at the top of a pile of the containing specialized objects. Again, detail is added to the design by acting out scenarios identifying object behavior in particular situation themselves.

2.9.2.5 Text Analysis

A general approach first suggested by Abbott [Booch, 1991] is the textual analysis of a problem description. This approach is tried as pail of a number of methods, since it can be done early in the analysis phase. All that is required is some kind of initial problem description which some authors suggest may be as small as a single paragraph. The initial problem description may have been arrived at by tome other analysis tool, such as interviewing, or may simply be a specification provided by a domain expert or end-user. Rumbaugh emphasises is that the problem description should be 'a statement of needs, not a proposal for a solution...The problem statement is just a starting point for understanding the problem, not an immutable document' [Rumbaugh et al, 1991]. In a text analysis, any nouns identified correspond to objects or attributes, and verbs correspond to associations or operations (Figure 2.22). An 'operation' in the analysis phase is what we call a method in the implementation; strictly speaking a method is the implementation of an operation.

A much deeper textual analysis than the simple 'noun vs. verb' approach is possible, but to some extent limited by the vagaries of English (Craham suggests that Chinese might

be a better tool for this kind of analysis!). One problem is that, as Booch says, 'any noun can be verb, and any verb can be nouned' [Booch 1994]. We might equally find the verb phrase 'to report' or the noun phrase 'a report' in a problem description referring to the same thing, so is it an object or an operation? This potential ambiguity might seem to invalidate the process entirely, but it is a useful first step in identifying candidate objects and their behaviours, perhaps to build into some initial scenarios for use with CRC cards or use-case storyboards.

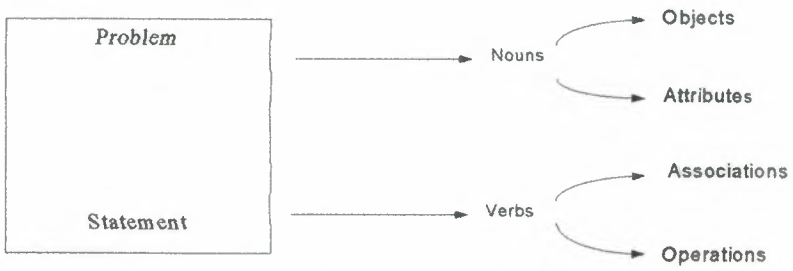


Figure 2.22: A Text Analysis.

2.9.3 Identifying Abstractions

There are, as Booch notes, two processes in the identification of the abstractions in a system, 'discovery' and 'invention' [Booch 1994]. Discovery is the Identification of objects which are found in the problem domain, perhaps referred to by domain experts and whose descriptions are part of the problem itself. In contrast we have to invent other abstractions - those which allow us to implementation. We might say that we discover objects in the analysis phase and invent objects in the design phase. As we noted in the very first chapter, Sroustrup refers to 'artifacts of the implementation', objects like databases, stacks and screen managers which belong to the application rather than the problem.

2.9.3.1 Refining Abstractions

Once an abstraction has been identified, it must be refined role in the overall systems have been clearly defined. Any object we identify will have to be seen in relation to other abstractions, in terms of class hierarchies and aggregations. Given that some classes are abstract, and simply exist t provide generalizations of other classes, the appropriate must be found. A class which is too abstract may lead to large differences between it and the next level down in the hierarchy. In contrast, classes, which are too specialized, may lead to duplication and redundancy. In the end, the 'granularity' of

abstractions is based on the best models available for high cohesion and low coupling. In addition to these two qualities we might also consider there others

Sufficient: does the abstraction do enough to be sufficient for its purpose ?

Completeness: does its interface cover enough aspects of the abstraction for it to have common usage.

Primitiveness: do all aspects of its interface have to access the underlying representation in order to be implemented?

The optimum level for all of these characteristics is to a degree subjective, ultimately developers have to apply their own judgement.

2.9.3.2 Naming Conventions

One important aspect of refining abstractions is to name them appropriately. As Beck & Cunningham state 'The class name of an object creates a vocabulary for discussing a design' [Beck & Cunningham, 1989]. Clearly, if in an appropriate name is chosen then this may cause problems in understanding the role of that class in the system and in relation to other classes. This principles applies right through from the initial naming of classes to the design and implementation stages. At a lower level of detail, Booch suggests the following approach to naming all aspects of classes and objects [Booch, 1994].

1. Name objects with proper nouns.
2. Classes should be named with common nouns.
3. Modifier methods should be named with active verbs.
4. Selector methods should imply a query, or be named with verbs of the form 'to be'.

Names are particularly important when deciding on the semantics of inheritance and aggregation. If the names of our classes in an inheritance hierarchy do not easily suit the 'is a kind of' relationship, then we need to decide if either the names or the hierarchy may be inappropriate. To suggest a simplistic example, we might have a class called 'Rectangle' inheriting from a class called 'Line'. However, the statement 'a rectangle is a kind of line' does not read too easily. Perhaps we need an intermediate class called

'Polygon', and another more abstract base clan called 'GraphicsObject' from which both it and 'Line' inherit. Clearly, the appropriate model depends on the requirements of the problem, but thinking about names in this way can help to clarify the issues involved in defining abstractions.

Similarly, if an aggregation does flat easily fit with an 'a part of' description, then again either the names or the structure may need revision. Relationships which have been modeled as aggregations but are better expressed as associations might be revealed by such analyses. To say that 'a trailer is a part of a lorry' might appear valid, but we' may find that in fact different trailers and tractors combine together at different times to make a 'Lorry' and that both tractors and trailers are objects in their own right.

2.9.3.3 Associations and Mechanism

Objects which do not send messages to each other cannot combine to create a useful system, so the associations between objects which define these messages are a crucial part of the analysis and design. Booch uses the term 'mechanisms' to describe the way that objects collaborate together to provide some high level behaviour required by the system. Without mechanisms, a software system has no organisation, so it is important that the developer is able to identify situations in which objects must collaborate. These can be analysed via scenarios, identifying which objects collaborate in a particular process and which methods e.ch object requires in order to do so. Rumbaugh provides us with a series of tools for this process, beginning with the scenarios themselves, then moving on the event trace diagrams, which formalise the message passing between the objects in a given scenario. Finally, an event flow diagram is used to summarise all the messages passed between objects and therefore the methods needed for objects to send and respond to messages. Although there are some differences in emphasis between methodologists the way that these links between objects are defined, they must ultimately provide the means for objects to communicate the one another.

Chapter III

DESIGN and IMPLEMENTATION of a STUDENT TABLE

3.1 Introduction

Many software vendors create time table generating programs for universities, for administrative use. But most of them are specific to the universities, faculties, and departments needs. There has not been any generalization of program structure when generating student time tables in general.

Implementation of a time table generating program. The design factor issues are considered general for each department, and faculty. When designing a time-table generating program the most important issue to be considered is the conflicting of lectures that arise due to irregular students. This case is analyzed for optimum solution, and placed on the time table.

The second critical point to be considered is the common lectures taken by students from all other departments and to be placed in the time-table. There are many other factors affecting this case of time table generating program, but many conflicts arise when the number of courses increase, and number of irregular students increase.

3. 2 Design Phase

3.2.1 Collection of Data

The program is designed to work only for the Engineering faculty having three departments. They are Electrical and Electronics Engineering, Computer Engineering, and Mechanical Engineering Departments. Since the program applies for all these departments it can easily be applied to any other departments under different faculties.

The first data to be collected was all the courses offered by each department for a full year degree. This record is saved in a file, where the main program can access the records in generating the time table program. The second file is the Technical elective courses offered to all departments, and the third file is the non technical elective courses.

The collection of data section is flexible, since if there are any changes in the courses, the files could be easily updated, without any changes in the program structure. This also implies that for any faculty or department just the names of the files have to be given to the program to execute.

The contents of the files are organized as follows:

1. The Main Courses (course.txt): This file contains information regarding courses offered for the departments. The file contains the course code, the course explanation and the credits of the course. The credits indicate the workload of the course to be fitted in the time table program. The full explanation of the courses is recorded in a separate file called course_exp.txt which contains the full course descriptions of the courses. In this file the course code is also linked to the explanation so that a link list is formed when anyone accesses the course.txt file. This is necessary such that if the lecturer decides to update the contents of the course later, he can by updating this file.
2. The Technical Electives (telective.txt): This file contains information regarding technical elective courses with course codes, and explanation of each. The full explanation of the technical elective courses is recorded into telective_ext.txt. Same principles apply to both of the files as in main courses.
3. The Non-Technical Electives (ntelective.txt): This file contains information regarding non-technical elective courses with course codes, and explanation of each. The full explanation of each technical elective course is recorded into ntelective_ext.txt. Same principles apply to both of the files as in main courses.

3.2.2 Program Design

In this part the program has to be designed based on the courses offered in a semester. The program operates on the contents of the files in the semester. We assume a list of the courses that will be offered in a semester, and the program, given on the input course codes, selects from the necessary input file, and gets the credits and the courses offered in a semester.

Upon receiving the credits, the program places the course code in the time table, and deal locates the space required data for that course from the time table. This division of

space is divided into years. For example for Computer Engineering, there exists four years, and the courses can be nested year by year.

In the time table there exists 40 course hours, and the program is capable of placing $40 \times n$ courses in the time table, where n is the number of independent courses to be placed differentiated without clashes on the same time. The number n is determined by the program in such a place that it might differ from one course hour to another. For example on Monday first two hours there might be 8 courses thought between 08:30-10:00 am, and on Tuesday there might be 6 different courses thought between 08:30-10:00 am. It is totally dependent on the program, and the number of courses offered in a semester.

To one specific location of lecture hour, different departments, different faculties can place their courses, as long as there aren't any clashes in the time table. The sample of division of courses has been categorized for Computer Engineering, Electrical and Electronic Engineering, and Mechanical Engineering. The courses offered have a specific structure that can be applied to all faculties and departments. The following format is used as shown in Figure 3.1.

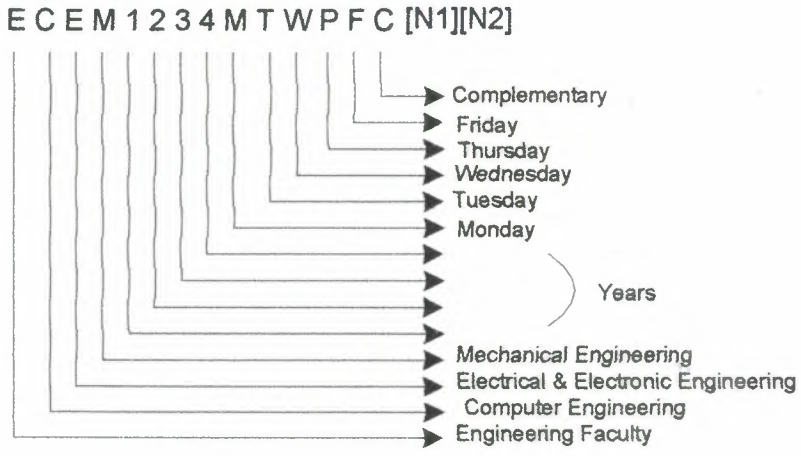


Figure 3.1: Lecture Array Assignment Format Structure.

In Figure 3.1 the lectures format is given the above format where N1 represents the lecture number, and N2 the course hour to be placed in the time table. Both N1 and N2

can take any number of values depending on the number of courses offered in a semester and how long the day it will be.

All the departments will be offered a lecture course format according to the one given in Figure 3.1. As an example for Computer Engineering the code layout will be EC1M: Engineering, Computer, Year 1, and Monday. For Electrical Engineering EE2TC: Engineering, Electrical, Year 2, Tuesday, and Complementary. The complementary field is necessary for the allocation of the courses given.

The program works in multiple ways as, the courses offered in a semester for all the departments will be entered, and the program will divide these courses into separate files, depending on the faculty. Then the program reads these files, and places them on the time table without crashes. Let's assume that COM 211 course is offered. The program will access the necessary file, and allocate to array EC1M[N1][N2] and automatically the array EC2MC[N1][N2] becomes allocated to indicate that that place for the time table has been allocated, and no second year course with Computer Engineering can be placed there.

The array structure of all engineering faculties has been shown in Table 3.1 below.

EC1M	EC1T	EC1W	EC1P	EC1F
EC1MC	EC1TC	EC1WC	EC1PC	EC1FC
EC2M	EC2T	EC2W	EC2P	EC2F
EC2MC	EC2TC	EC2WC	EC2PC	EC2FC
EC3M	EC3T	EC3W	EC3P	EC3F
EC3MC	EC3TC	EC3WC	EC3PC	EC3FC
EC4M	EC4T	EC4W	EC4P	EC4F
EC4MC	EC4TC	EC4WC	EC4PC	EC4FC

a. For Computer Engineering.

EE1M	EE1T	EE1W	EE1P	EE1F
EE1MC	EE1TC	EE1WC	EE1PC	EE1FC
EE2M	EE2T	EE2W	EE2P	EE2F
EE2MC	EE2TC	EE2WC	EE2PC	EE2FC

EE3M	EE3T	EE3W	EE3P	EE3F
EEMCE	EE3TC	EE3WC	EE3PC	EE3FC
EEM	EE4T	EE4W	EE4P	EE4F
EEMC	EE4TC	EE4WC	EE4PC	EE4FC

b. For Electrical & Electronics Engineering.

EM1M	EM1T	EM1W	EM1P	EM1F
EM1MC	EM1TC	EM1WC	EM1PC	EM1FC
EM2M	EM2T	EM2W	EM2P	EM2F
EM2MC	EM2TC	EM2WC	EM2PC	EM2FC
EM3M	EM3T	EM3W	EM3P	EM3F
EMMCE	EM3TC	EM3WC	EM3PC	EM3FC
EMM	EM4T	EM4W	EM4P	EM4F
EMMC	EM4TC	EM4WC	EM4PC	EM4FC

c. For Mechanical Engineering

Table 3.1: Array Structure for All Departments.

The optimization of the case study of the program is also studied, by giving specific parameters to the program as change of course(s) to be applied to the program. The program than rearranges it self to make the necessary changes in the new time table.

CONCLUSIONS

The object-oriented approach reflects our natural perceptions of the world as being composed of objects, classifiable into general types. Objects comprise private data public processes. Object technology encompasses the field of analysis, design, programming and databases. Object-oriented programming is a two stage process, first identifying and creating classes, and secondly creating objects of these classes and defining the messages that pass between them over time. The two part of first chapter have covered all the basic tools needed to build C++ programs. Other elements of C++ syntax, particularly those specific to the application of objevt-oriented programming. A class is an 'object factory' which can instantiate many objects of that class. Each object encapsulates its own state data. The 'costructor' method creates an object by reserving memory space for it. It may also perform other programmer-defined tasks.

Objects may be external, static, automatic or dynamic. External objects have global visibility in at least one program module, and persist for the lifetime of the program. The lifetime and visibility automatic objects is delimited by scope. Static objects also have their visibility delimited by scope, persist from their instantiations to the ned of the program. A destructor (like the constructor) has the same name as its class, but is identified by the tilde character. We can use 'new' and 'delete' to dynamically instantiate and destroy objects at run time.

The metaclass holds class attributes and methods. Metadata is about other data. Inheritance allows classes to inherit attributes and methods from other classes in a classification hierarchy. In C++ 'protected' keyword allows the methods of a derived class access to its inherited attributes. Some forms of aggregation are fixed, constant relationships, with others are containers for unpredictable sets of objects. Between these two extremes there may ben many combinations of aggregiate class. Polymorphism means 'having namy forms'. In an object-oriented program, methods and operatorscan have many forms by being 'overloaded' in various ways. In object-oriented systems, we can classiyf those types of polymorphism which relate to methods and operators together, because they exbihit similar characteristics, including the ability to be dynamically bound at run time.

Operators may be overloaded to work with user defined data types (objects). This is an extension of the idea of coercion, which allows a single operator to be applied to a range of data types. The assignment operator has a default behaviour when applied to objects, copying the attribute values from one object to another. This behaviour may need to be overridden where objects contain pointers. Operators are inherit by derived classes in a similar way to other methods. However, this is not the case with the assignment operator, derived classes do not inherit overloaded assignment operators. A container is a type of aggregation. Containers in software give us control over collections of dynamic objects. Container classes provide us with with objects of various types, such as string, stacks, queues and ordered collections. Containers have iterator methods to allow various form of access to the contained objects. Encapsulating collections of objects inside containers simplifies the interface between the objects and the data structures which are used to contain them. Generic container classes can be constructed using templates, allowing container objects to be instantiated for different types of contained object.

Objects may need persist beyond a single program run time, perhaps between different programs. Object data can be stored in traditional file formays, but an object-oriented database is a preferable storage medium since it maintains the integrity of objects. To handle objects of our own classes in stream I/O, we can overload the 'istream' and 'ostream' operators to work with user-defined types, or create our own file handling methods. Attributes may be written out as fields in larger records or as single data records. Object-oriented programs of any scale require object-oriented analysis and design. The transition from analysis to design is to an extent a seamless process, but analysis is in general terms looking at the problem, and design is providing a framework for the solution.

The time table program is a very sophisticated and complex program, and is very useful for Universities in arranging time tables with minimum number of crashes using optimization. The program structure is designed flexibly in such a way that the array structure allows opening of new departments and faculties to be added inside the program. This flexibility allows more usage of the program university(s) wide, when generating time tables.

REFERENCES

[Parsons, 1994]
'Object-Oriented Programming with C++'
David Parsons
Dp Publications Ltd, Aldine Place, London, 1994

William Ford and William Topp.
Data Structures with C++.
Prentice-Hall, Inc., 1996.

Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, Inc., 1977.

Bjarne Stroustrup.
The C++ Programming Language.
Addison-Wesley, 2nd edition, 1991.

[Blair et al, 1991]
'Object-Oriented languages, systems and Applications'
Gordon Blair, John Gallagter, David Hutchison and Doug Shepherd
Pitman, London 1991

[Booch,1991]
'Object-Oriented Design With Applications'
Grady Booch
Benjamin/Cummings, Redwood City, Cajif. 1991

[Booch,1991]
'Object-Oriented Design With Applications' (2nd Edition)
Grady Booch
Benjamin/Cummings, Redwood City, Calif. 1994

[Coad/Yourdon, 1990]
'Object-Oriented Analysis'
Peter Coad and Edward Yourdon
Yourdon Press, Prentice-Hall, New Jersey 1990

[Cox 1986]
'Object-Oriented Programming – Am Evolutionary Approach'
Brad J. Cox
Addison-Wesley, Reading, Mass. 1986

[Graham, 1991]
'Object-Oriented Mrthods'
lan Graham
Addison-Wesley, Workingham 1991

[Meyer, 1988]
'Object-Oriented Software Construction'
Bertrand Meyer
Prentice-Hall, Hemel Hempstead, 1988

[Shlear & Mellor, 1988]
'Object-Oriented Systems Analysis – Modelling the World in Data'
Sally Shlear, Stephen J. Mellor
Yourdon Press, New Jersey, 1988

[Stroustrup, 1991]
'The C++ Programming Language' (2nd Edition)
Bjarne Stroustrup
Addison-Wesley, Reading Mass. 1991

APPENDIX

```
#define airplaneH
#define airplaneH
#define AIRLINER 0
#define COMMUTER 1
#define PRIVATE 2
#define TAKINGOFF 0
#define CRUISING 1
#define LANDING 2
#define ONRAMP 3
#define MSG_CHANGE 0
#define MSG_TAKEOFF 1
#define MSG_LAND 2
#define MSG_REPORT 3
class Airplane {
public:
Airplane(const char* _name, int _type = AIRLINER);
~Airplane();
virtual int GetStatus(char* statusString);
int GetStatus()
{
return status;
}
int Speed()
{
return speed;
}
int Heading()
{
return heading;
}
```

```
int Altitude()
{
return altitude;
}
void ReportStatus();
bool SendMessage(int msg, char* response,
int spd = -1, int dir = -1, int alt = -1);
char* name;
protected:
virtual void TakeOff(int dir);
virtual void Land();
private:
int speed;
int altitude;
int heading;
int status;
int type;
int ceiling;
};
#endif
-------------------------------
#include <stdio.h>
#include <iostream.h>
#include "airplane.h"
// Constructor performs initialization
Airplane::Airplane(const char* _name, int _type) :
type(_type),
status(ONRAMP),
speed(0),
altitude(0),
heading(0)
{
switch (type) {
case AIRLINER : ceiling = 35000; break;
```

```
case COMMUTER : ceiling = 20000; break;
case PRIVATE : ceiling = 8000;
}
name = new char[50];
strcpy(name, _name);
}
// Destructor performs cleanup.
Airplane::~Airplane()
{
delete[ ] name;
}
// Gets a message from the user.
bool
Airplane::SendMessage(int msg, char* response,
int spd, int dir, int alt)
{
// Check for bad commands.
if (spd > 500) {
strcpy(response, "Speed cannot be more than 500.");
return false;
}
if (dir > 360) {
strcpy(response, "Heading cannot be over 360 degrees.");
return false;
}
if (alt < 100 && alt != -1) {
strcpy(response, "I'd crash, bonehead!");
return false;
}
if (alt > ceiling) {
strcpy(response, "I can't go that high.");
return false;
}
// Do something base on which command was sent.
```

```
switch (msg) {
case MSG_TAKEOFF : {
// Can't take off if already in the air!
if (status != ONRAMP) {
strcpy(response, "I'm already in the air!");
return false;
}
TakeOff(dir);
break;
}
case MSG_CHANGE : {
// Can't change anything if on the ground.
if (status == ONRAMP) {
strcpy(response, "I'm on the ground");
return false;
}
// Only change if a non-negative value was passed.
if (spd != -1) speed = spd;
if (dir != -1) heading = dir;
if (alt != -1) altitude = alt;
status == CRUISING;
break;
}
case MSG_LAND : {
if (status == ONRAMP) {
strcpy(response, "I'm already on the ground.");
return false;
}
Land();
break;
}
case MSG_REPORT : ReportStatus();
}
// Standard reponse if all went well.
```

```
strcpy(response, "Roger.");
return true;
}
Perform takeoff.
void
Airplane::TakeOff(int dir)
{
heading = dir;
status = TAKINGOFF;
}
// Perform landing.
void
Airplane::Land()
{
speed = heading = altitude = 0;
status == ONRAMP;
}
// Build a string to report the airplane's status.
int
Airplane::GetStatus(char* statusString)
{
sprintf(statusString, "%s, Altitude: %d, Heading: %d, "
"Speed: %d\n", name, altitude, heading, speed);
return status;
}
// Get the status string and output it to the screen.
void
Airplane::ReportStatus()
{
char buff[100];
GetStatus(buff);
cout << endl << buff << endl;
}
```

```
#include <stdlib.h>
#include <iostream.h>
#include <conio.h>
#pragma hdrstop
USERES("Airport.res");
USEUNIT("airplane.cpp");
#include "airplane.h"
int getInput(int max);
void getItems(int& speed, int& dir, int& alt);
int main(int argc, char **argv)
{
char returnMsg[100];
Set up an array of Airplanes and create
// three Airplane objects.
Airplane* planes[3];
planes[0] = new Airplane("TWA 1040");
planes[1] = new Airplane("United Express 749", COMMUTER);
planes[2] = new Airplane("Cessna 3238T", PRIVATE);
Start the loop.
do {
int plane, message, speed, altitude, direction;
speed = altitude = direction = -1;
Get a plane to whom a message will be sent.
// List all of the planes and let the user pick one.
cout << endl << "Who do you want to send a message to?";
cout << endl << endl << "0. Quit" << endl;
for (int i=0;i<3;i++)
cout << i + 1 << ". " << planes[i]->name << endl;
// Call the getInput() function to get the plane number.
plane = getInput(4);
// If the user chose item 0 then break out of the loop.
if (plane == -1) break;
// The plane acknowledges.
```

```
cout << endl << planes[plane]->name << ", roger.";
cout << endl << endl;
// Allow the user to choose a message to send.
cout << "What message do you want to send?" << endl;
cout << endl << "0. Quit" << endl;;
cout << "1. State Change" << endl;
cout << "2. Take Off" << endl;
cout << "3. Land" << endl;
cout << "4. Report Status" << endl;
message = getInput(5);
// Break out of the loop if the user chose 0.
if (message == -1) break;
// If the user chose item 1 then we need to get input
// for the new speed, direction, and altitude. Call
// the getItems() function to do that.
if (message == 0)
getItems(speed, direction, altitude);
// Send the plane the message.
bool goodMsg = planes[plane]->SendMessage(
message, returnMsg, speed, direction, altitude);
// Something was wrong with the message
if (!goodMsg) cout << endl << "Unable to comply.";
// Display the plane's response.
cout << endl << returnMsg << endl;
}
while (1);
// Delete the Airplane objects.
for (int i=0;i<3;i++) delete planes[i];
}
int getInput(int max)
{
int choice;
do {
choice = getch();
```

```
choice -= 49;
} while (choice < -1 || choice > max);
return choice;
}
void getItems(int& speed, int& dir, int& alt)
{
cout << endl << "Enter new speed: ";
getch();
cin >> speed;
cout << "Enter new heading: ";
cin >> dir;
cout << "Enter new altitude: ";
cin >> alt;
cout << endl;
}
```

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#pragma hdrstop
#include "structur.h"
void displayRecord(int, mailingListRecord mlRec);
int main(int, char**)
{
// create an array of mailingListRecord structures
mailingListRecord listArray[3];
cout << endl;
int index = 0;
// get three records
do {
cout << "First Name: ";
cin.getline(listArray[index].firstName,
sizeof(listArray[index].firstName) - 1);
cout << "Last Name: ";
```

```
cin.getline(listArray[index].lastName,
sizeof(listArray[index].lastName) - 1);
cout << "Address: ";
cin.getline(listArray[index].address,
sizeof(listArray[index].address) - 1);
cout << "City: ";
cin.getline(listArray[index].city,
sizeof(listArray[index].city) - 1);
cout << "State: ";
cin.getline(listArray[index].state,
sizeof(listArray[index].state) - 1);
char buff[10];
cout << "Zip: ";
cin.getline(buff, sizeof(buff) - 1);
listArray[index].zip = atoi(buff);
index++;
cout << endl;
}
while (index < 3);
// clear the screen
clrscr();
// display the three records
for (int i=0;i<3;i++) {
displayRecord(i, listArray[i]);
}
// ask the user to choose a record
cout << "Choose a record: ";
char rec;
// be sure only 1, 2, or 3 was selected
do {
rec = getch();
rec -= 49;
} while (rec < 0 || rec > 2);
// assign the selected record to a temporary variable
```

```
mailingListRecord temp = listArray[rec];
clrscr();
cout << endl;
// display the selected record
displayRecord(rec, temp);
getch();
return 0;
}
void displayRecord(int num, mailingListRecord mlRec)
{
cout << "Record " << num + 1 << ":" << endl;
cout << "Name: " << mlRec.firstName << " ";
cout << mlRec.lastName;
cout << endl;
cout << "Address: " << mlRec.address;
cout << endl << " ";
cout << mlRec.city << ", ";
cout << mlRec.state << " ";
cout << mlRec.zip;
cout << endl << endl;
}
---------------------------------------
# include<string.h>
class Bankaccount
{
Private:
İnt acc_number;
Char acc_holder[20];
Float curr_balance;
Public:
İnt getacc_number( );
Char* getacc_holder( );
Float getcurr_balance( );
Void setacc_number(int number_in);
```

```
Void setacc_holder(char* holder_in);
Void deposit(float amount);
Void withdrawal(float amount);
};
int Bankaccount::getacc_number( )
{ return acc_number; }
char* Bankaccount::getacc_holder( )
{ return acc_holder; }
float Bankaccount::getcurr_balance( )
{ return curr_balance; }
void Bankaccount::setacc_number( int number_in)
{ acc_number=number_in; }
void Bankaccount::setacc_holder(char* holder_in)
{ strncpy(acc_holder, holder_in,19);
   acc_holder[19]='\0';
}
void Bankaccount::deposit(float amount)
{ curr_balance=curr_balance+amount; }
void Bankaccount::withdrawal(float amount)
{ curr_balance=curr_balance-amount;  }
---------------------------------------
#include<iostream.h>
#include 'Bankacct.h'
void main( )
{
Bankaccount account1, account2, account3;
account1.setacc_number(100);
account2.setacc_number(110);
account3.setacc_number(120);
cout<<"Account numbers are."<<endl;
cout<<account1.setaccc_number( )<<endl;
cout<<account2.setaccc_number( )<<endl;
cout<<account3.setaccc_number( )<<endl; }
```