CHAPTER ONE

SOFTWARE COMPLEXITY AND OBJECT-ORIENTED PROGRAMMING CONCEPTS

This chapter begins with a discussion of the need for programming methods and techniques that deal with the problems of large and complex software systems. The first section of the discussion focuses on the issues of *complexity*, *maintainability* and *reusability* of components within a complex modular software system. In this section we admit that the organisational complexity of biological systems are ideal for the development of modelling techniques to deal with these issues. The traditional approach is to use models base on well established engineering principles. Hence the use of the term "Software Engineering". However, it is interested to note that many engineering principles are modelled on those found in natural biological systems. Typical examples are the concepts of control-feedback and encapsulation.

The second section in this chapter presents a generalised discussion of the principles and concepts of Object Oriented Programming as a method for dealing with large and complex programming problems. Object Oriented Programming is presented as an alternative for dealing with the problems of complexity and maintainability of large and complex software systems. This section draws on current discussion in the debate on Object Oriented Computing.

We are careful to make the point that Object Oriented Programming is not a final solution to the problem of ensuring a correct and efficient algorithmic solution to program problems. Object Oriented Programming focuses mainly on the problems of largeness and complexity. As a programmer, you still need to know how to design algorithmic solutions to specific program problems. However, research in theoretical computer science has over the pass 50 years derived generic solution to many common data manipulation problems. These algorithms are available as generic modules in many Object Oriented Programming languages. They relief the programmer from having to derive a specific solution to such problem. In Object Oriented Programming, you still need to pay attention to the design of algorithms that manipulate data values uniquely or that perform tasks unique to your application. Never the less, the Object Oriented system allows you to do this in an incremental manner. Do not get carried away by the false notion that you do not need to pay careful attention to the design and implementation of control logic.

LEARNING OBJECTIVES

* An understanding of the need to develop methods/techniques to deal with the issues of Complexity and maintainability for large and complex software systems.

* To provide a framework of reasons for the development of Object-Oriented concepts as a means for solving the problems of complexity and maintainability in the development of large software systems.

* Become familiar with the various concepts and terminology used in object-oriented computing.

* To provide a basic understanding of object-oriented concepts

* To provide a framework for understanding reasoned debate on object-oriented   computing.

PROGRAMMING PROBLEM SOLUTION METHODS

In this section I develop an argumentative discussion that looks at and suggest a deeper approach towards the development and maintenance of complex software systems. I look at the main issues involved in the development and maintenance of complex software systems by suggesting an approach, which I think, is a deeper Object Oriented approach. In this approach, I attempt to take advantage of the fact that the Object Oriented approach has result in dramatic improvements because of its modelling similarity to many natural processes.

THE PROBLEM EVOLUTION

Originally, at the start of the development of stable two state devices programming these device was focused on their internal constructions. Programmers paid exact attention to their internal constructions and the use of available resources. However, as the technology evolved new method for programming them have also evolved. This evolution include the development of systems of abstraction at the hardware and software level, speed at which operations are implemented by hardware element and the continued reduce cost of hardware components. The development of Systems of abstraction at the software level includes user interface modelled on real world objects, New compiler techniques, 4th and 5th generation programming languages, and programming methods. These gains in software techniques are closely interrelated and form the basis of our main concern.

COMPARING A SOFTWARE SYSTEM TO AN INTELLIGENT SYSTEM

It is desirable to design and code complex software systems that are intelligent. We begin with a simple definition of the term "Intelligent System". An intelligent system is a system that responds to its external environment in a consistent and logical fashion. Therefore, the system synchronise it self with positive external stimuli or it protects it self from destructive or corrosive external stimuli. By this we mean that the system respond to fulfil its purpose whenever it is presented with positive stimuli and it respond to protect it self when presented with negative stimuli. A positive stimulus is the presence of an external resource viewed by the system as been useful to its functionality or existence. On the other hand a negative stimulus is the presence of an external resource viewed by the system as been detrimental to its functionality or existence. A resource is any form of energy or agent that can affect the state, functionality or existence of a system.

How does this concept relate to a complex software system? Well, a complex software system respond positively to fulfil its purpose whenever presented with a correct stimuli and it respond to a negative stimuli to protect its existence. Hence, the concepts of abstraction and encapsulation are used to define the internal complexity of an object-oriented system. In addition, an intelligent system adjusts to its external environment while maintaining its individuality. For example, a software system should be design so that it does not accept input data values that will cause it to crash or malfunction. In a relative sense biological systems are intelligent because they are pre-programmed to interact with their environment through feedback systems. These feedback systems are design to interact with the external environment so that the system can adjust its internal apparatus to synchronise its whole function to achieve the ultimate purpose of continued existence and perpetuation of it self. It would be nice to be able to design and implement software systems that carry similar virtues. Unfortunately, one cannot always know exactly what events will occur in the future to affect the behaviour of a system. However, one can develop techniques and methods for designing software system base on what one learn from biological systems.

Software systems tend to be large in an attempt to deal with all possible kind of behaviour. However, largeness leads to complexity of structure. The correlation between largeness and complexity is strong. A complex system has a higher probability of malfunction because the chance that an element in the system will malfunction whenever a resource flows through the system is increase with its complexity. However, there are ways to control and deal with the issues of largeness and complexity.

There are two basic approaches in the development of intelligent software systems. One approach place emphasis on the way the internal elements that make up the system are controlled and the second approach places emphasis on the way individual components in the system or the whole system interacts with its external environment. Biological systems interact with their external environment whilst maintaining their individuality. Therefore, we use descriptive terms for biological systems to articulate the problems of complex software systems. Therefore, I present a general solution model base on those features common to biological systems.

It is easy to accept that the following terms can be used to give concise description of biological systems.
- Responsive to their external environment
- Complex
- Self-maintaining and self-perpetuating
- Highly Specialised
- Generic
- Abstract

It would be interested to develop a discussion around each of these attributes that is confined to our main concern in this discussion. Unfortunately, this is not my approach at this point.

Any method for dealing with the solution to programming problems must address the following issues:

- ❑ Complexity
- ❑ Maintainability
- ❑ Genericity
- ❑ Abstraction

These issues are highly interrelated as we explain below. In many respect, programming languages have evolved to deal with these generalise issues. However, this is usually within the context of a specialisation. Specialisation of function coupled with interdependency is a method for dealing with complexity in intelligent systems. In this section, we look at approach methods used for solving some of these issues, their merits and demerits. We also draw a conclusion at the end of each section.

By the logic of a system, we mean those principles that govern the flow of Information, Data, Energy, or any other entity that determines the overall functionality or state of the system or component elements in the system. Therefore, we see that the application of specialisation or generalisation is an essential approach that must be carefully used to control complexity in large multi-functional systems. To give a practical example of this, consider a large interactive software system in which the user is required to enter data values of different kind in various modules. Clearly, there are going to be a lot of similarity between the functions that obtain user input values in these modules and the way they do validation to ensure correct input. Therefore, it is desirable to have a single function within the whole system that obtain and validate user-input data. This calls for a generalisation of the overall function of obtaining input data from the user and the development of a specialised function that implement the generalised solution. Because this generalised function is now used in different parts of the system, its functionality must be abstracted and its interface to the overall system must be generalised and to some extent abstracted. This is just one approach for dealing with this kind of complexity issue in a large software system.

An intelligent system is made robust by the clear definition and separation of functions into sub-systems and the creation of interfaces between components within the system. Therefore, a complex system consists of sub-systems that require the use of resource, which is conveyed to it through interrelated components. A component sub-system may take some resource to create an intermediate resource needed by another component within the system. Therefore, there is also a hierarchy of interdependency within the system. It is the nature of this interdependency that needs to be controlled in order to deal

with the issues of complexity and maintainability. As an example illustration consider a plumbing network sub-system in a larger system. The plumbing network is a programmed system that conveys fluid under pressure to a destination determined by the physical layout of the plumbed network. Therefore, the plumbed network serves it purpose of making resource in a system available to other components within the system in an orderly and systematic manner. We can add extra intelligence to the functionality of the plumbing network by adding feedback loops to the system at essential interfaces like in biological systems. In this way we can get the plumbing network to interact with the external environment of the system and internal components in the system that the plumbing network serves. As a practical example, consider a plumbing system that conveys a reactant fluid to a reactant vessel in a chemical plant. Use of a feed back loop to control pressure in the reactant vessel creates intelligence

COMPLEXITY

The complexity of a programming problem deals with those issues of largeness, the flow and control of data values and the way in which component elements interrelate with each other. The complexity of a large system is understood through the logic that illustrates these issues in the system.

More recent programming languages have features that allow the programmer to focus more on the end user rather on the problem domain. This feature greatly enhances the speed at which software systems can be developed and deployed. Actually, what this really does is that it provides the software developer with a frame of reference from which to view the desired result of the development and maintenance of a software system. From this view, it is easy to define and control the desired definitions of characteristics within the system. These programming languages has predefined general formats that map's specific GUI element onto code that defines the behaviour of these GUI elements in the user paradigm. In addition, both the behaviour of elements of the software system and the user are synchronised because of their common view of functionality. Therefore, the process of writing specific instructions using a script language is abstracted from the programmer to a view that is in keeping with the user expectation. The programmer no longer needs to pay attention to the interpretation of strings of characters in a programming language. Instead, the programmer pays attention to the interpretation of GUI elements in the programming language as they relate directly to the intended user of the system. LogicCoder is a software development tool that implements this concept/principle in a more general form without paying much attention to a specific programming language. However, the focus in LogicCoder is not so much on the end user of a system as it is on the result in designing and coding the control logic of a system.

We define and explain the following two terms
➢ Static behaviour and
➢ Dynamic behaviour

In any system, we can say that a static description relates directly to tangible resources within the system. We define a resource as any entity or agent with the potential to become involved in the dynamics of the system. The dynamics of a system are those interactions and processes that result in a change of state of the system. A system state changes whenever it's static description changes.

The Static behaviour or description of an Object is a description that generalises the use of an object in a system. A static behaviour does not necessarily fully describe the purpose or function of an Object. I define the static behaviour of an object as the highest degree of predictability of the object's state given its exposure to some usable resource or energy. An example of a static behaviour or description is that of a filing cabinet. It is used for filing documents. It is used for other purpose also. A static description can also include dynamic components in a system depending on the user interpretation of the system's state or the user frame of reference in interpreting the system's state. Therefore, a static description is not necessarily inferior to a dynamic description but in some instance may just be a generalisation of a dynamic description without the detail.

The dynamic behaviour or description of an Object is a description that relates directly to an interactive event in the Objects environment. On the other hand, the dynamic description relates directly to the use of resources within the system and the logic that controls their interrelationship to other objects or components within the system. Dynamic objects dispense or consume static objects or resources. Dynamic objects are governed by orderly rules in a well-behaved system.

In a GUI programming system, the programmer assumes that he user is aware of or is familiar with the static description/behaviour of objects to a far greater degree than he does about the dynamic behaviour of objects. The essential complexity issue in these systems is the dynamic behaviour of objects. Therefore, an essential skill needed by a programmer that uses such a system is the definition and implementation of dynamic behaviour. The programmer must also document these behaviours for the benefit of the user and for the purpose of system maintenance. Most OO system consists of component objects which behaviour is highly localised. This localised behaviour is implemented through a principle of abstraction and a highly defined interface. A localise behaviour is a dynamic behaviour associated with an object with little or no implication on the behaviour of other objects in the system, except where interactive behaviour is control through a system of defined interfaces. Actually, at machine level, the internal state data in

particular or component systems interact with each other mainly through the use of RAM and to a lesser extent through external magnetic memory. OO systems control this interaction by use of abstraction, restricted access and the use of defined interfaces.

It is interested to observe that some commercial systems have used OO Computing to an extreme. This extremity exploits the principle of information hiding to control a programmer's creativity in a selfish manner. Therefore, we conclude that some OO systems have taken these principles to an extreme. OO Computing is a method of programming that effectively deals with the issues of complexity and maintainability of large software systems. It is not intended to be a way to restrict a programmer so that he/she uses a system within the confines of its originator. This is a case of manipulative control mainly for the purpose of greed.

In a GUI system, behaviour is defined in terms of real life objects. And to make things better, most objects and their behaviour are represented visually by familiar representation. Therefore, there is no need to use the alphabet of a natural language to describe all behaviours in such a system. The description for the behaviour of an object in such a system is intuitive. Actually, there is no such thing.

LARGENESS

The largeness of a problem has to do with our capacity to deal with the amount of processed information about the problem in order to encapsulate a description or conclusion about the problem. Therefore, largeness has to do with capacity, knowledge and the process of encapsulation.

INPUT STIMULI AND A SYSTEM'S RESPONSE

A stimulus is the appearance of some form of energy or resource that can or cannot be used by the system in a positive manner. For example, a system may respond to an increase in temperature by switching on a cooling mechanism that prevents its internal components from absorbing too much heat energy and so result in damage from overheating. Therefore, some stimuli to a system can result in damage if the system does not have proper response mechanism to deal with them.

By an input stimuli interface, we mean an interface system through which an input resource or a resource (form of energy) seeks to cross the boundary of an enclosed system. This input resource is only allowed to cross the boundary of the system in a controlled manner as determined by the interface between the external environment and internal components within the system. The

internal components within the system use this resource to fulfil the requirements of components within the system without causing harm or damage to the system. Here the concept of an interface overlaps with that of an encapsulated system.

MAINTAINABILITY

The maintainability of a system deals with the issues of growth, change in requirements and repair. The process of system maintenance must deal with the issues of quality assurance. As a system grows its ability to cope with its environment improves. Therefore, maintenance is for improving a systems ability to cope with its requirements. Because it is not always possible to know in advance all the possible input to a system and so device robust response mechanism for them all, there are times when a system response to an external stimuli results in a detrimental effect to the system. Therefore, maintenance also seeks to repair damage and to improve the response system that was not able to cope and so result in the damage. The act or process of repair to a system should be a localised process so that it has little or not effect on component element within the system. On the other hand, the act or process of growth can not in most case be a localised or isolated process. Growth is always an integrated process because of the nature of the interdependency of components within a system. The growth of any one element within the system is for the benefit of all other elements within the system. Hence, the growth of a selected component will necessitate the growth of other components within the system. Growth is also a way to deal with the issue of repair.

By a change in requirement we mean, we mean that a system or its component is now required to respond differently to an external stimuli for which it was not originally programmed (designed) to respond. If a system has no way to respond to external stimuli, under constant input of that stimulus it may break down

GENERICITY

The construction and functionalities of component sub-systems within natural biological systems are great examples of genericity. As a simple to follow first hand illustration, take a look at the nervous and the circulatory systems in the human body. We can simply summarise the function of the nervous system by saying that it receives and supplies nerve impulse to various organs in the body under the central control of the brain. For the circulatory system we can say that it collects and distribute resources to the various organs in the body centrally from the heart. Note the similarity of both descriptions. Careful examination of the physical outlay of both systems reveals a network of

interconnected fibrous organelles designed to convey the required resource. Therefore, the physical outlay of both systems is very similar. Therefore the physical design of one can be easily determine from the design of the other.

The current definition of a generic system in computing is a system that will perform the same basic kind of operation on data values regardless of the data type or structure that is used to represent these data values. Examples of well-used generic systems in digital computer systems are List, Trees, Queues, and so on.

In order for a generic system to be applicable regardless of the data type, it must be defined solely in terms of the operations on units of data values and not in terms of a particular data type or data structure.

ABSTRACTION

I begin with a general definition of the term Encapsulation. This is a key term in the concept of Abstraction.

Encapsulation:

We can define encapsulation in a general manner by saying that it is the process of confining the internal structure of a system to restrict or control the flow of energy or resource into or out it. We give definition that is more specific in the following paragraphs.

Abstraction seems to be a natural function of the human mind in dealing with the real world. An abstracted system hides the detailed complexity of a large or complex system. It views a system in terms of a particular function or purpose. In a general sense, an Abstraction is a concise representation of a complicated system or idea. The detailed implementation of an abstraction is not essential to understanding its purpose and functionality with regards to its external environment. It is OK to view a system in terms of an abstraction as long as the system is encapsulated by a system that separates its internal detail from its external environment and it interacts with the external through a well-defined interface. This leads me to the suggestion of *Circumstantial* and *Relativistic Abstraction*. A Circumstantial Abstraction is an abstraction in which the system of concern is not completely insulated from its external environment. In this case, the abstracted view may not be accurate or complete and so can lead to erroneous conclusions or assumptions. A simple example of this is an electronic system place within a box that does not completely protect the system from its external environment. For example the box covering may not protect the system from magnetic fields that can affect the function of the system. A Relativistic Abstraction is an abstraction that is determined by the viewer's perception of the encapsulated state of a system.

A relativistic abstraction of a system may be correct in a given frame of reference and not correct in an alternative frame of reference.

Technical information provided in certain sections of this manual is base on the following assumptions about the knowledge of the reader.

The reader has prior knowledge about
➢ High Level programming languages
➢ The use of language translators

In the object oriented system, objects can be found in an array or in a collection such as a desktop. Objects in the collection have been defined such that they share common characteristics such as: Knowledge of screen location and the ability to be repositioned, activated or deactivated. When a programmer selects an object, there is no information about the object's type or the internal structure of the object. Hence, the precise behaviour of the object when a message is sent to it cannot be predicted. The ability to develop applications in which messages are sent to object of unknown type is a very powerful programming technique. This is what *object-oriented programming* means.

User interfaces such as the Window environment in Microsoft Windows and the desktop in Smalltalk programming environment, are examples where an object's precise type is not always known. For instance, the object displayed on the screen can be scrollbars, text, windows, buttons, icons, menus or other type of controllers. An object responds to messages sent to it in the form of key-presses or mouse clicks. Different things will happen depending on which object is pointed to when the message is sent. For example, if the mouse pointer is place on the minimise button and the left mouse button is press, the application's window will turn into an icon. On the other hand, if the mouse pointer is place on the "**File Save**" icon in a word-processor application, the file currently been edited will be saved to disk.

ENCAPSULATION, INHERITANCE, AND POLYMORPHISM

The objects in a single collection such as the desktop described above are known as *polymorphic objects*. Polymorphic objects share certain characteristics, such as the ability to be taken as a single collection, and the ability to respond to the same messages. Polymorphism is directly related to Inheritance. *Inheritance* allows the programmer to define new objects that inherits the properties and behaviour of previously defined objects. The new object can then be described in terms of how it differs from existing object. For example, a prompt window behaves just like a text editor window, except that the prompt window only holds a single line of text. Inheritance and Polymorphism are complex interrelated concepts that require the understanding of some preliminary concepts. Encapsulation is a system for defining objects in which data and related procedures are encapsulated into a single entity. The concept of encapsulation also explains how objects are created from class definitions and how objects communicate. An object-oriented program is best viewed as a set of communicating objects as opposed to a set of procedures. Emphasising objects instead of action as is typical in control oriented language leads to a very different program structure when object-oriented concepts are used in the design of a program.

PROGRAM STRUCTURE

The structuring techniques used in procedural programming are well established. The basic building block in procedural - programming are Procedures. A well-known design technique is top-down stepwise refinement. In this approach, the programmer starts with a general view of the programming problem. The programmer then recursively or iteratively break each identified task in the overall programming problem solution into smaller more manageable pieces-procedures. The programmer begins with an overall description of what the program is supposed to do in a single statement. For example, the programmer may define the overall tasks of a sales letter program as follows:

Produce Sales Letter

Program Tasks
*1. Display the menu and obtain the user selection.
 2. Determine the selection to be performed.
*3. Display the program instructions.
*4. Prepare and print the letters.
*5. Print the mailing labels.

Each task that begins with an asterisk "*" represents a complex procedure that need to be broken down into a set of simpler procedures. For example, the 4th task could be broken down as follows.
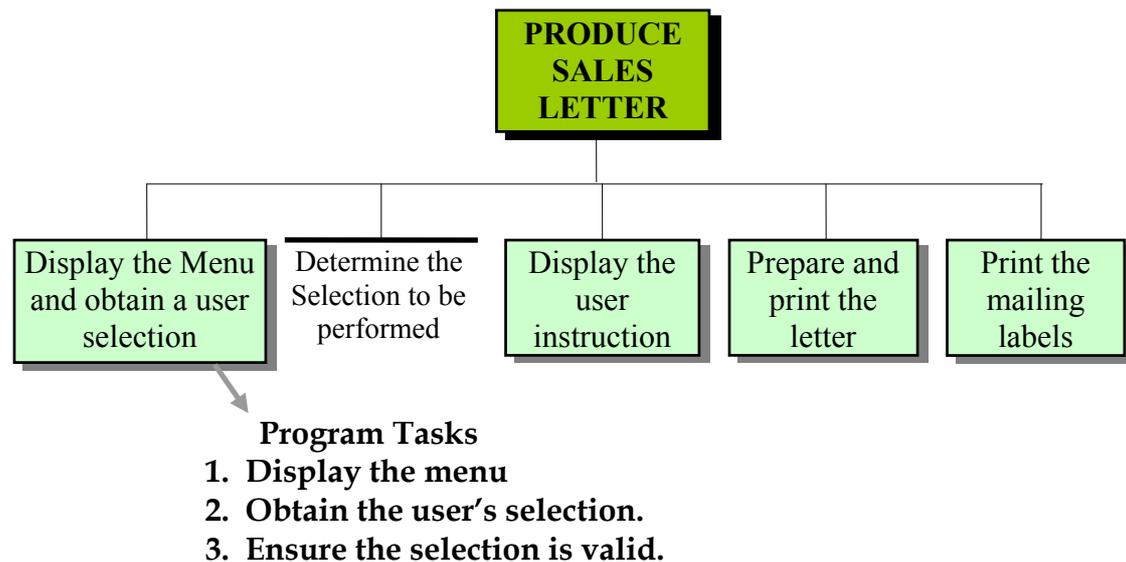
Prepare and Print the Letter
Program Tasks
*1. Obtain valid title and name from the user.
 2.   Obtain the address.
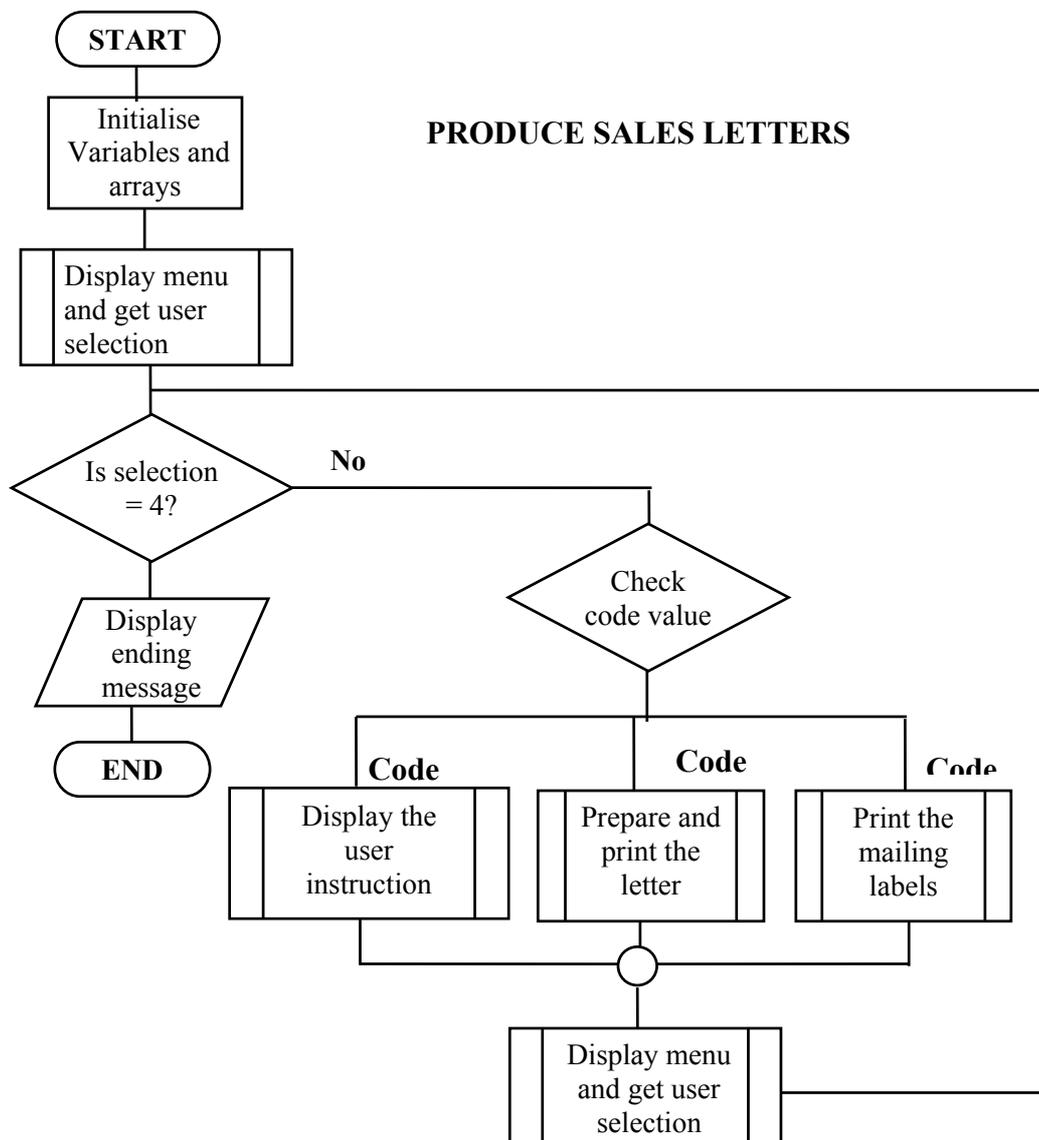*3. Obtain a city or county and valid postcode.
*4. Print the letter.

The strategy in this approach is to keep breaking down each sub-task until a given sub-task can be express in a manner that easily map's onto procedural statements in the language that will implement the program solution. The process of breaking down a complex program problem into simpler tasks is the same, regardless of how complex the program problem is. The problem solution begins with a general abstraction of the program problem. The abstract program statement is broken down into less abstract program statement(s) that map's more closely onto syntax and semantics of the programming language that is to be used for implementing the program solution. The lowest levels of abstraction in the program problem solution are

those tasks that can be expressed directly as programming language statements. The hierarchical decomposition of a programming problem gives rise to a structure of interrelated procedures like that illustrated below. Therefore, a hierarchy chart is an exact method for dealing with the issues of knowledge and encapsulation when solving the problem of largeness.

```
                         ┌──────────────┐
                         │   PRODUCE    │
                         │    SALES     │
                         │   LETTER     │
                         └──────────────┘
```

| Display the Menu and obtain a user selection | Determine the Selection to be performed | Display the user instruction | Prepare and print the letter | Print the mailing labels |

Program Tasks
1. **Display the menu**
2. **Obtain the user's selection.**
3. **Ensure the selection is valid.**

Figure 1-1
Example hierarchy chart used to illustrate the structural outcome in the design of a procedural program. In the implementation of each task, local data is hidden inside procedures and shared data is passed in arguments.

In such a system illustrated in figure 1-1 you the programmer have to then implement a control logic structure that determine the ways in which components within the whole system interacts with each other. This includes careful attention to the way data values interacts with each other. In most case, this interaction has to be protected by the control logic, which introduces extra consideration during its design. A control logic that can be used to implement the required structure for the program tasks listed in figure 1-1 is given in figure 1-2.

START

Initialise Variables and arrays

PRODUCE SALES LETTERS

Display menu and get user selection

Is selection = 4?

No

Display ending message

Check code value

END

Code

Code

Code

Display the user instruction

Prepare and print the letter

Print the mailing labels

Display menu and get user selection

Figure 1-2
The programmer has to think about the control logic that will implement the various functions in the problem solution illustrated in figure 1-1. In many case, a general solution or alternative implementation for such control logic exist in some Object Oriented system and is available as a generic system. In this case, a user of the Object Oriented system need not think about such an implementation solution.

A hierarchy chart is used to show the relationship between various modules within a complex software system. It shows the hierarchy of relationship between modules that perform simple task and those that perform major tasks in the system. A module is a task that can be accomplished through the

implementation of a specialised function. Therefore, in the example above the task to Display the user instruction is implemented by a single function. On the other hand, the flowchart is used to illustrate the flow of control in the system when it is executed. It shows how control moves from one point to the next in the software system. Both Hierarchy chart and flowchart graphically document the structure of the software system in a detailed manner. Each module in the hierarchy chart requires a separate flowchart. However, by generalisation and the use of functions it is possible to implement a group of tasks by use of a single function. This is achieved by definition of a well-defined interface through which different parameter values pass to the function whenever it is called.

From this we see that functional decomposition is an exact approach in dealing with the complexity of large software systems. Object Oriented programming provides an alternative approach.

In a well-designed Control Oriented system, the relationships between component elements within the system are well defined and clear to see. Data values are also localised within procedural elements and shared data are pass between procedures by way of parameter passing. This creates an interface between the procedures and the originator of these data values. During an instance, procedures only come into existence whenever they are called by other procedures. But procedures must maintain the integrity and correct use of these data values them selves. Therefore, an abstracted procedure is not adequate insulation from the side effects of interacting data values.

In contrast, the structure of a typical object-oriented program can be illustrated by a diagram like that in figure 1-3. The difference in structure is obvious.

Figure 1-3
Example program structure in an object oriented programming system. A program consists of a system of objects that communicate through messages.

Object oriented approach appeal to our natural experience in the evaluation of the organisation of complex systems such as computers, plants, galaxies and large social institutions. Object-oriented decomposition of a program problem has a significantly higher number of advantages in comparison to algorithmic decomposition. An object oriented approach results in smaller systems through the reuse of common mechanisms. Object-oriented systems are also more resilient to changes and are therefore better able to evolve over time because their design is base on stable intermediate forms. There is less risks involved in the use of object-oriented system to build complex software systems because they are design to evolve incrementally from smaller systems in which we already have confidence. Further, more, it gives us greater flexibility in the separation of concern of functionality in a complex system of interacting parts.

CLASSES

The basic motive for implementing classes in an object-oriented language is to provide standard methodical form for object classification base on definition of their components. Classes support the notion of classification. They allow groups of objects to be created which shares identical behaviour. This is implemented by providing templates (classes) for the creation of objects in the system. A more precise definition for classes is as follows: *A class is a template from which objects are created. It contains a definition of the state descriptor and the methods for an object*.

Therefore, the class template gives a complete description of a class in terms of its data structure, internal algorithms and its external interface. The major advantage in this approach is that the implementation of a class need only be done once. A baker's pastry cutter provides a simple and practical analogy of a class. A particular pastry cutter defines the precise size and shape of a particular pastry. For example, one particular cutter may define a circular shape having a 5cm diameter. All pastries created from this particular cutter will have this precise shape and size. To illustrate further, we consider the drink machine mentioned earlier on. We can define a class of drink machine as follows:

Class name: Drinks machines
State variables: Quantity of milk
          Quantity of sugar
          Quantity of water
          Quantity of tea
          Quantity of coffee
Methods: Mechanism and recipe to make tea
          Mechanism and recipe to make coffee

Objects belonging to a class have properties that are common to that class. Each object is an instance of the class to which it belongs. Its internal data values and the methods that can perform operations on these data values determine the internal properties of a class instance. A class instance only has an external property represented as a message through an interface. This external property is only visible to a receiving object. Therefore both the internal and external state of a class objects are controlled. Hence, a class definition must also include a method of instantiation for new class objects.

CREATING INSTANCES OF AN OBJECT

The use of classes makes it necessary to have a mechanism for creating objects of a particular class. Objects can be created dynamically by a process of

instantiation. By instantiation, we mean the process where by an instance of an object is created. Instantiation is implemented by calling on a class to create a new object base on the class template. The new object is initialise at the time of its creation and is now available within its environment for access by other objects. When an object is initialised, its state variables are assigned initial values. The environment of an object is such that each instance of the object within its environment is aware of the class to which it belongs. The general characteristics and attributes of classes in the system also make them object. Hence, there are interfaces that the developer must use when defining classes. One method that is always provided by a class is the method *new*. This method creates instance of a specific class when it is invoked. Hence, sending a message to the appropriate class requesting a new object to be created and initialised creates instance of a new object. Objects created from a particular class are referred to as instances of that class. It should be noted that although all instances of a given class exhibit common behaviour, they are not identical in every respect. However, they are identical in terms of interface and implementation. Each object has it's own state which can change over time and can depend on the previous state. The following examples illustrate how the instantiation process could be used to implement instances of the drink machine.

Machine_A : [Drinks_machine new]
Machine_B : [Drinks_machine new]
Machine_C : [Drinks_machine new]

The logic used in declaring instances of the drink machine will guarantee that the three machines will exhibit common behaviour.

The use of an arithmetic expression or a function call to declare the number of elements in an array is a more common practice for dynamic memory allocation. The **malloc( )** function is used in ANSI C to dynamically allocate memory during the run of a program.

MULTI-DIMENSIONAL ARRAYS

A one-dimensional array can be thought of as a row vector. For example the elements of the **Letters** array declared above can be represented as follows.
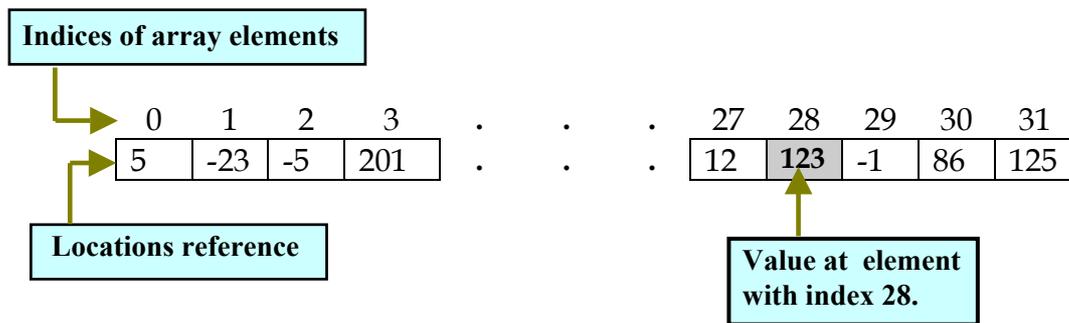
Figure 2-19
Elements of an array in ANSI C are numbered starting
from 0 to the number of elements minus 1.

You can also have multidimensional arrays in ANSI C. For example, you can define a two-dimensional array like the following.

```
float Yearly_earn[12][5];
```

This array consist of **12 × 5 = 60** elements of type **float**. The first element is reference by specifying **Yearly_earn**[0][0] and the last element is reference as **Yearly_earn**[11][4].

POINTERS

Every identifier used in a program has an associated address value. The address of an identifier is the first location in memory that specifies the beginning of the set of locations occupied by the value associated with the identifier. The amount of memory used to store the data value depends on the identifier type. An **int** type uses 2 bytes and a **float** type uses 4 bytes. An array type uses the number of elements in the array times the number of memory location use for each value in the array. A structure type uses memory that is the sum used for each member element in the structure, plus some padding if necessary. Because of this orderly and predictable way in

which data is stored in memory, it is possible to access data values in memory by using a variable that stores the relevant address. A variable that stores a memory address value is called a *pointer*. A pointer can be used to access the individual elements of an array or the individual data member of a structure. They can be used to initialise the array or to retrieve data from it.

ADDING ITEMS TO THE PROJECT FILE

You will now use the **P**roject | **A**dd Item function to add the following files to the project. **drawplnt.cpp**, **setgraph.cpp**, **draw.cpp**. You will then debug, compile, link and run the executable program.
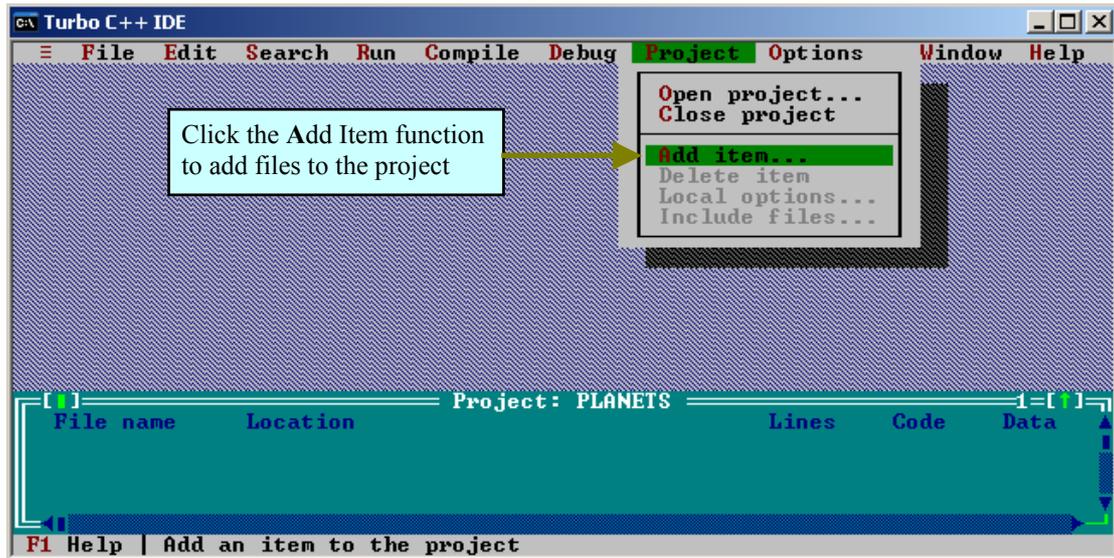
Figure 2-34
**You use the Project | Add Item function to select file items
to add to the project. You can also use the Project sub-menu
to delete items belonging to a project.**

(**21**) Click the **A**dd Items menu option in the **P**roject pull-down sub-menu. Alternatively, use the Up/Down arrow key to highlight this menu item and then press the Enter key once.

The Turbo C++ IDE respond with the **A**dd Items dialog window. Use this dialog window to navigate to folders with file items you want to add to the project. You can use the mouse left button to double click each file item you want to add to the project in a file list or highlight the item and then press the Enter key once. You then click the **D**one button when you are finish adding items to the project file.

(**22**) Use the **F**iles list box to navigate to the **DrawPlanets** directory where the files you created are stored.
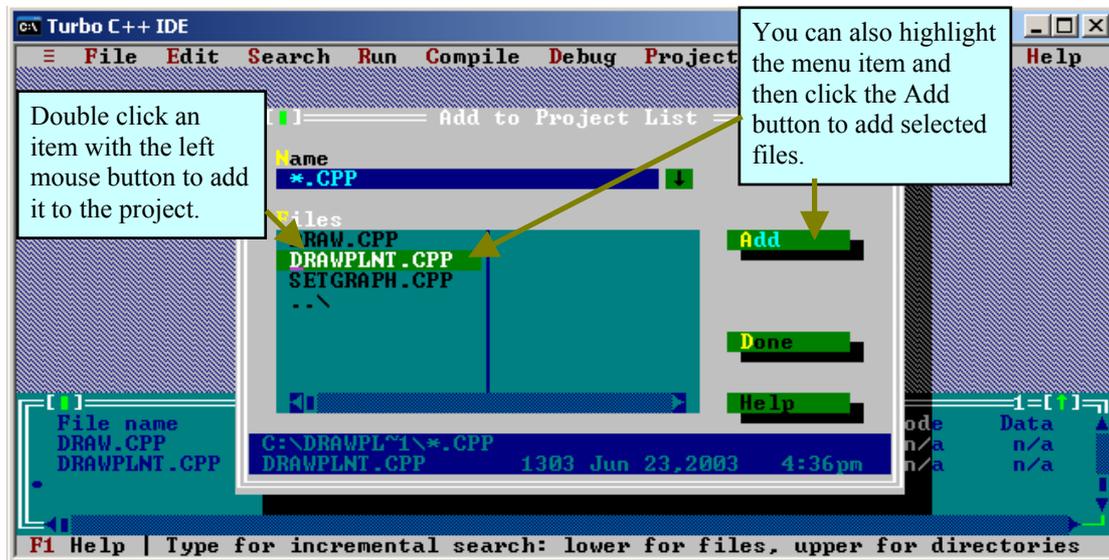
Figure 2-35
The Add to Project List dialog window allows you to select
**items from a file list.**

At this point, the **F**iles list box should look similar to that in figure 2-35.

(**23**) Use the mouse left button to double click each file in the **F**iles list box so that they also appear in the project window. Alternatively, highlight each item and then click the **A**dd button.

(**24**) Use the mouse left button to click the **D**one button when you are finish adding the three **CPP** source files to the project window.

(**25**) After you have added all files to the project, select **O**ptions| **L**inker from the menu bar, and then select **L**ibraries. You are then presented with the **L**ibraries dialog box as illustrated in figure 2-36, select **G**raphics Library and then toggle the check box to on by pressing the space bar or by clicking the option check box with the left mouse button. The option check box is selected whenever an "×" appears in it. It is unselected when it appears empty.

This sets the system to link the "**Graphics.lib"** with the source code when it is compiled and linked. Be certain that you set the path argument in calls to **initgraph** to the directory where your .BGI files are. Use the **** symbol instead of **** when setting the path. This ensures compliance with the ANSI C/C++ escape sequence. For example, you type the absolute path "**C:\\TC\\BGI**" or the relative path "**. . .\\BGI**".
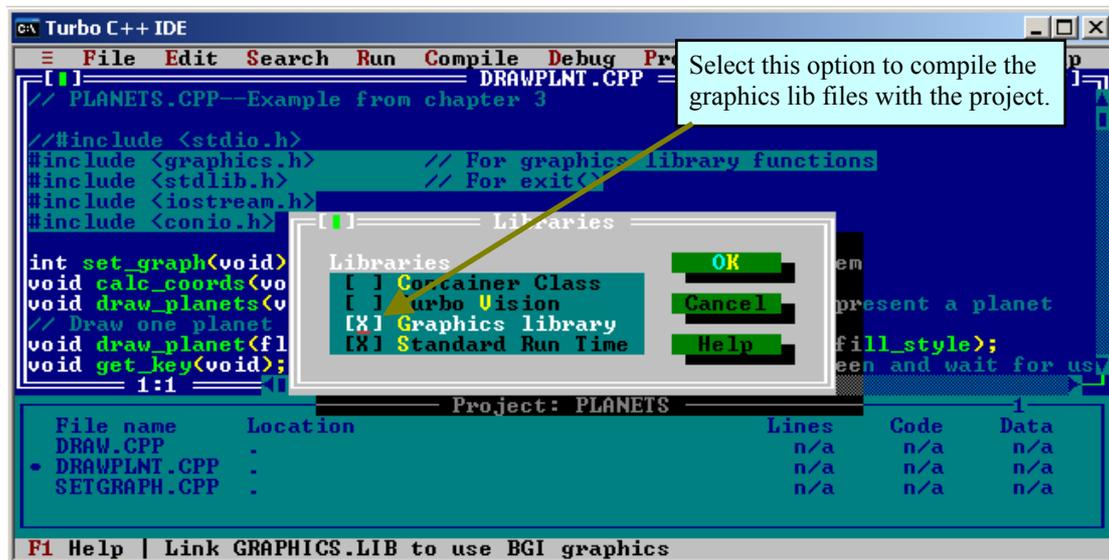
Figure 2-36
You set the system to link Turbo C++ standard graphics
library files when the Linker runs by selecting the Graphics
library option.

CREATING THE EXECUTABLE FILE

(**26**) Execute the **C**ompile | **C**ompile function so that the IDE attempts to compile the program.

The IDE respond with error messages. Most of which are undeclared identifiers in the file **draw.cpp**.

(**27**) If the project window is hidden or does not have the focus then execute the **W**indows | **P**roject function so that the project window appears.

(**28**) Use the mouse left button to double click the **draw.cpp** file in the project window so that the text editor opens with the **draw.cpp** file.

(**29**) Place the cursor at the head of the file and type the following.
#include <graphics.h>.

The set of macros used by the **draw_planet( )** function calls in the "**draw_planets( )**" function are declared in the standard graphics file "**graphics.h**". The include directive for this file in the "**drawplnt.cpp**" file is not visible in the "**draw.cpp"** file. Therefore, we need a separate include preprocess directive in this file.

In addition, the variables **max_x, max_y, y_org, au1, au2** and **erad** declared in "**drawplnt.cpp**" are not visible in this file. You will modify the declaration of these files to make them visible in this file also.

(**30**) Find each of the following variables in the "**drawplnt.cpp**" file, **max_x, max_y, y_org, au1, au2** and **erad**. Place the cursor anywhere on the line where each variable is declared and then press the **Home** key on your keyboard once.

The Turbo C++ IDE places the cursor at the first character of the text line whenever you press the **Home** key. When you press the **End** key, the cursor is place at the last character on the text line.

(**31**) Type the word **extern** followed by a space for each variable.

Placing the key word **extern** in front of a variable name tells the compiler that the variable is global and that it is to be reference externally by code in other files in the system.

(**32**) Execute the **C**ompile | **C**ompile function once more.

The project compiles in this instance

(**33**) Execute the **C**ompile | **B**uild All function so that the IDE creates the executable file.

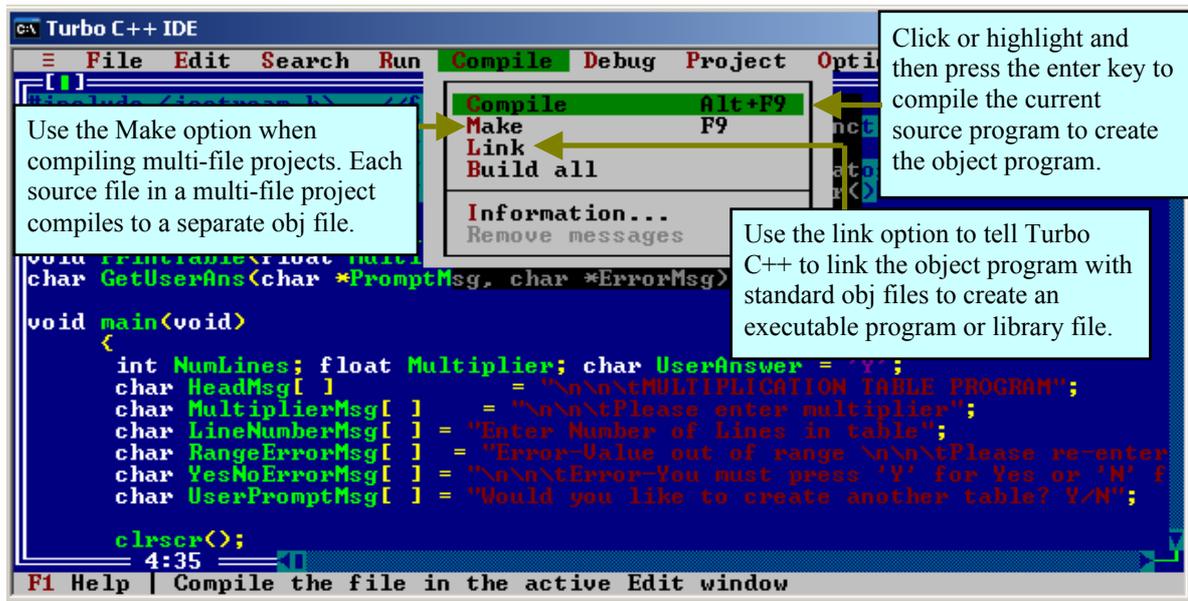# COMPILING THE EXAMPLE PROGRAM

Figure 3-14
You press the Alt+C keys together to get the Compile pull-down sub-menu. You then click the Compile menu item to compile the source code listed in the text editor. You can also use the Alt+F9 short cut key sequence to initiate the compiler.

You will now compile the source program to create an **obj** file. The compiler first checks the source code for errors. If there are any error, the compiler will halt compilation and then present you with a dialog window like that in figure 3-15. You will then use the message window to go to each line of code that has an error in it. You will then edit the source code to remove the error. After the source program compiles correctly to create the **obj** file, you will then link the **obj** file to standard library routines to create a machine executable file. Standard library routines are machine instructions that implement functions found in standard library files such as **stdio.h**, **string.h**, **conio.h** and so on. They are stored in standard locations and in standard **lib** files on the system. Figure 3-13 reviews the stages you must take the source code through to create the object program and the machine executable program.

When the Borland Turbo C++ IDE is installed on your system, compilation options in the system are set to default values. You can change these compilation options by selecting the **Option** menu item on the main menu bar and then select the **Compilation** menu item in the **Option** pull-down menu. Compilation option includes C++ options, optimisation option among other options. You will not change any default option on the system at this point.
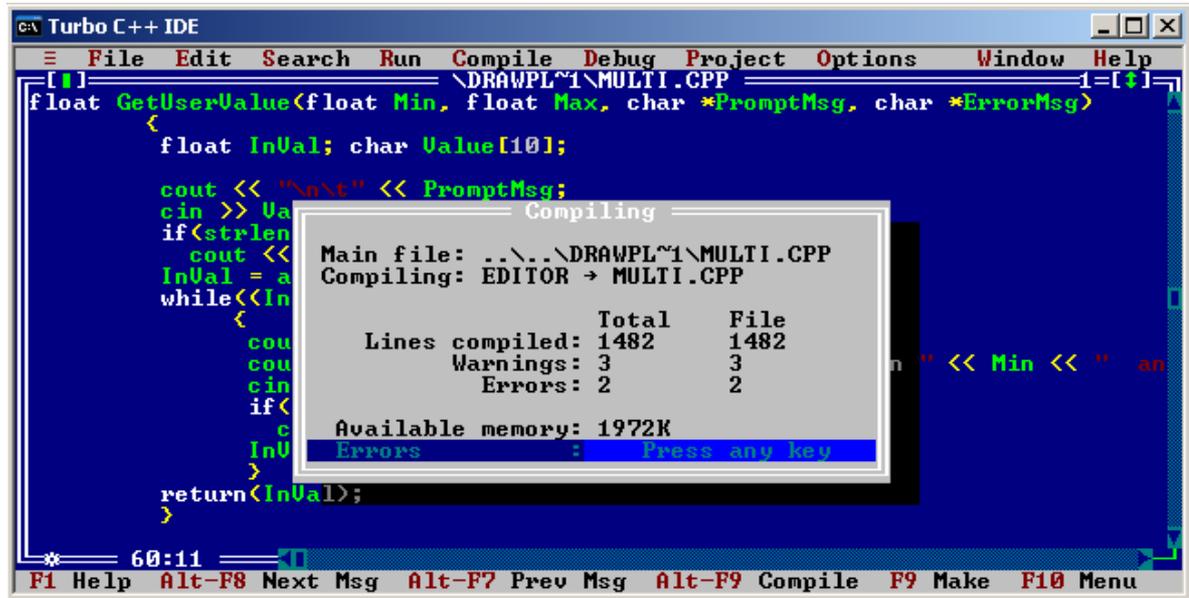
Figure 3-15
The Turbo C++ IDE presents a report dialog window at the
end of running the compile function.

(**10**) Click the **Compile** menu item on the IDE main menu bar so that the **Compile** pull-down menu appears as illustrated in figure 3-14. Alternatively, use the Right/Left arrow keys to highlight the **Compile** menu item and then press the Enter key once. You can also press the **Alt+C** keys to display this pull-down menu.
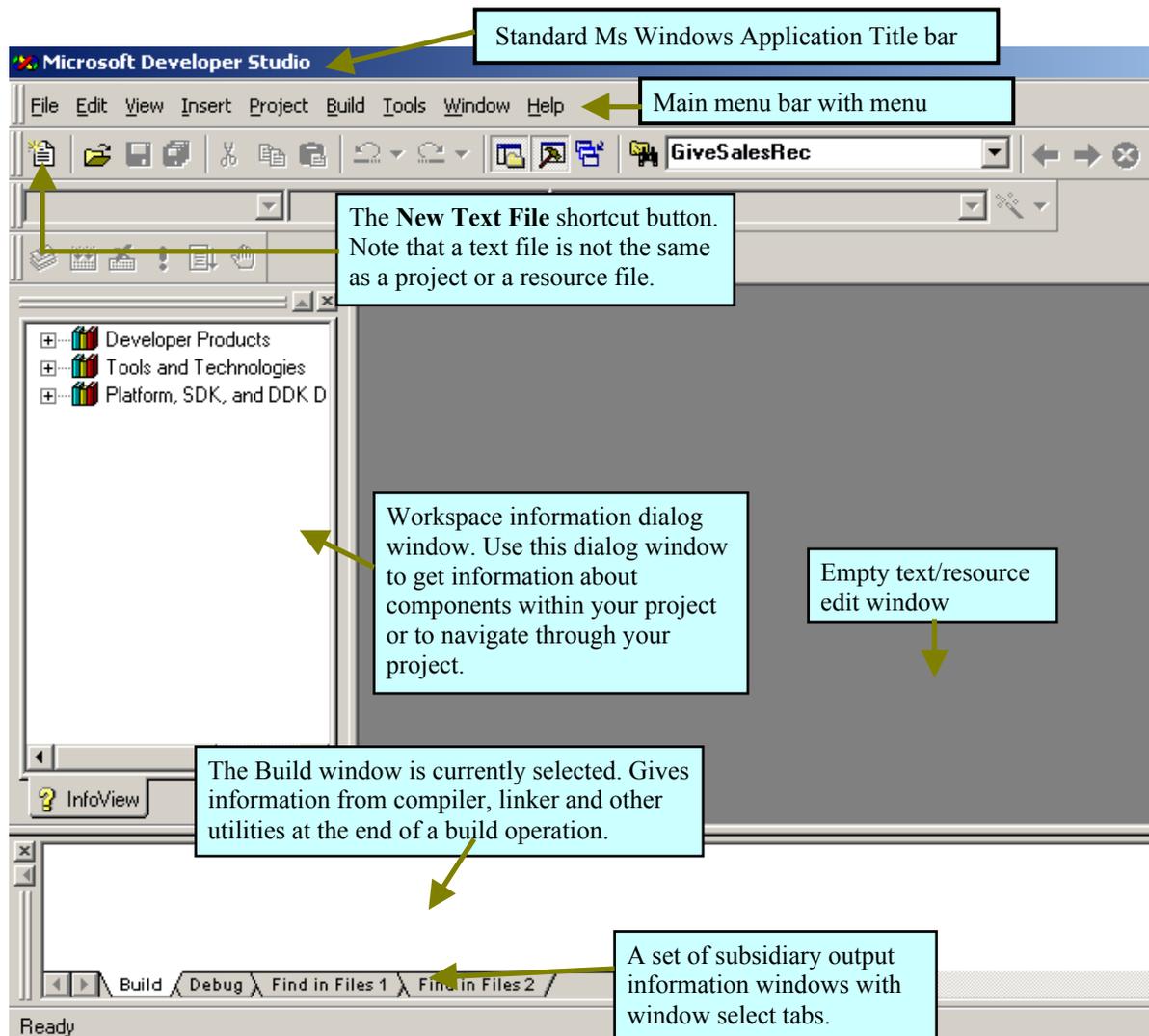
(**11**) Click the **Compile** item in the pull-down menu so that the compile function executes.

When the compilation function executes, it presents a dialog box that specify the errors in the source program or it specify that the compilation has been successful. In addition, it specify the number of lines compiled, the number of warnings, the name of the file that is compiled and the amount of available memory on the system. The available memory is the amount of memory made available to the executable file by the Turbo C++ IDE. At this point, your document window should look similar to that in figure 3-15.

(**12**) Press any key on the keyboard so that the Message window appears. Press any key once more so that the IDE switches to the first line with error in the text editor window

# THE MICROSOFT VISUAL C++ INTEGRATED DEVELOPMENT ENVIRONMEMNT

The Microsoft Visual C++ IDE is design to allow you to create executable programs that will run in a Microsoft environment. Therefore, the implementation of programs on this system is more specific to the Microsoft Windows ™ operating system.

Standard Ms Windows Application Title bar

Main menu bar with menu

The **New Text File** shortcut button. Note that a text file is not the same as a project or a resource file.

Workspace information dialog window. Use this dialog window to get information about components within your project or to navigate through your project.

Empty text/resource edit window

The Build window is currently selected. Gives information from compiler, linker and other utilities at the end of a build operation.

A set of subsidiary output information windows with window select tabs.

Figure 3-23
The Microsoft Visual C++ IDE provides functions that allow you to develop software systems that will run under the Microsoft Windows Operating system.

UDERSTANDING THE MS VISUAL C++ IDE

Figure 3-23 illustrates essential elements of the Ms Visual C++ IDE and their function.
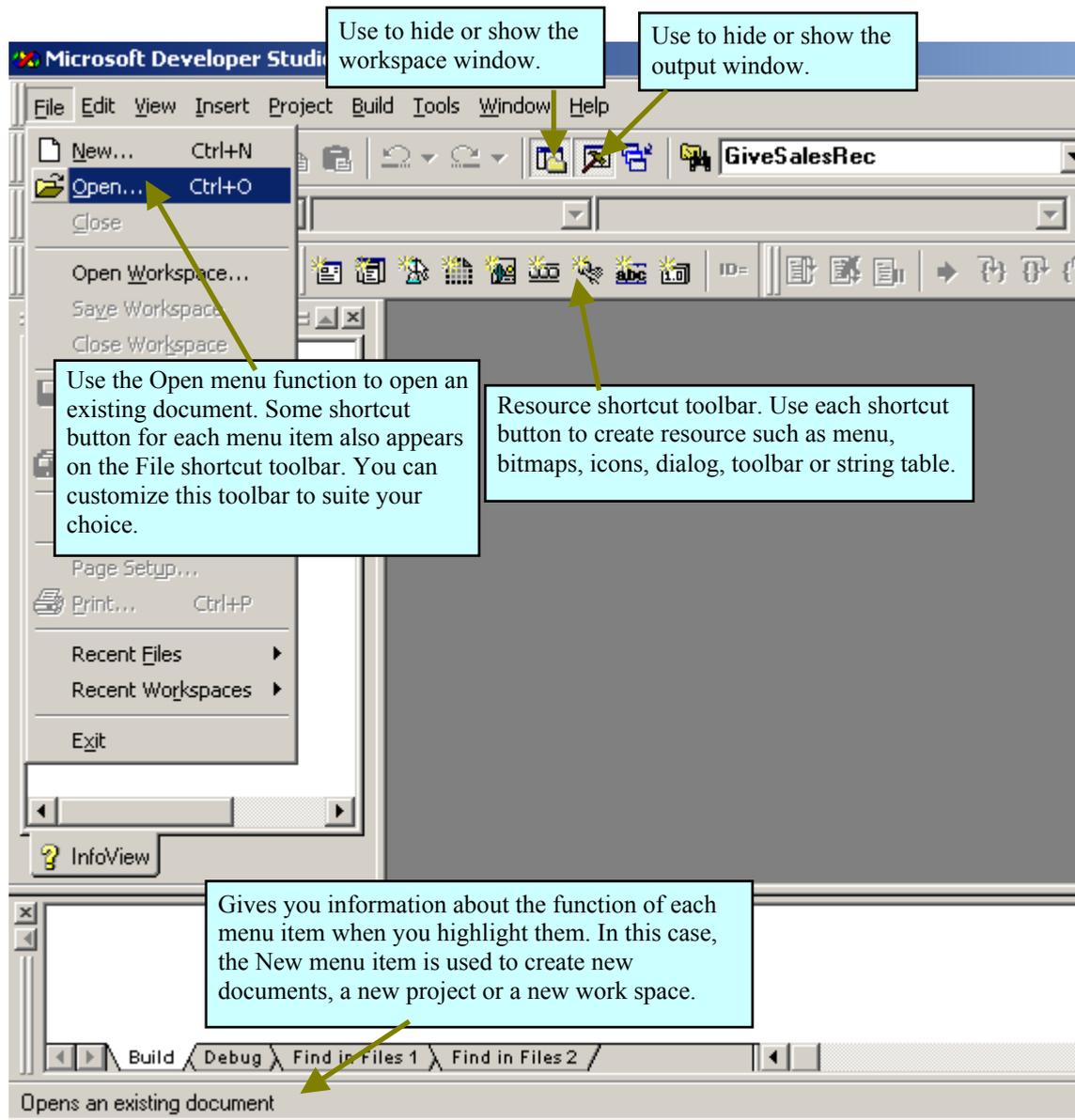
Figure 3-24
The File pull-down menu provides a set of functions that
allow you to perform standard file handing operations when
creating and editing a project. Note that some menu options
are greyed as a result of the state of the IDE.

You will use the Microsoft Visual C++ IDE to create an executable program similar to that you created with the Borland Turbo C++ IDE. Therefore, you will create a console application. Carefully study each element of the MSVC

IDE Graphical User Interface (GUI) as explained in the following figures. You must follow each step-by-step instruction as they are given.

Note from figure 3-23 that some menu items are not available in the IDE. The reason for this is intuitive, you can not perform certain operations on an object that does not exist or is not available. For example, you can not perform text edit operations on a text document that does not exist. In this case, you get access to the file through the interface provided by the IDE.

HANDS-ON EXERCISES 3-2

(1) Roll the mouse pointer over each menu item on the main menu bar so that their pull-down menu list appears.

You will use the **N**ew menu item in the **F**ile pull-down menu to create a new project space for a console application.

(2) Use the mouse left button to click the **F**ile menu item on the main menu bar so that the **F**ile pull-down menu appears.

(3) Click the **N**ew menu item in the **F**ile pull-down menu so that the **N**ew dialog window appears as illustrated in figure 3-25.

The New dialog window allows you to select the type of Ms Windows application you want to develop. Notice that the default platform is Win32. As a consequent, the MSVC IDE will create an application that will run on a windows 32 bit platform. A 32 bit platform is a hardware system that allow the operating system to reference memory locations with the use of 32 bit address values. Therefore, up to 2^{32} different physical memory locations can be access.
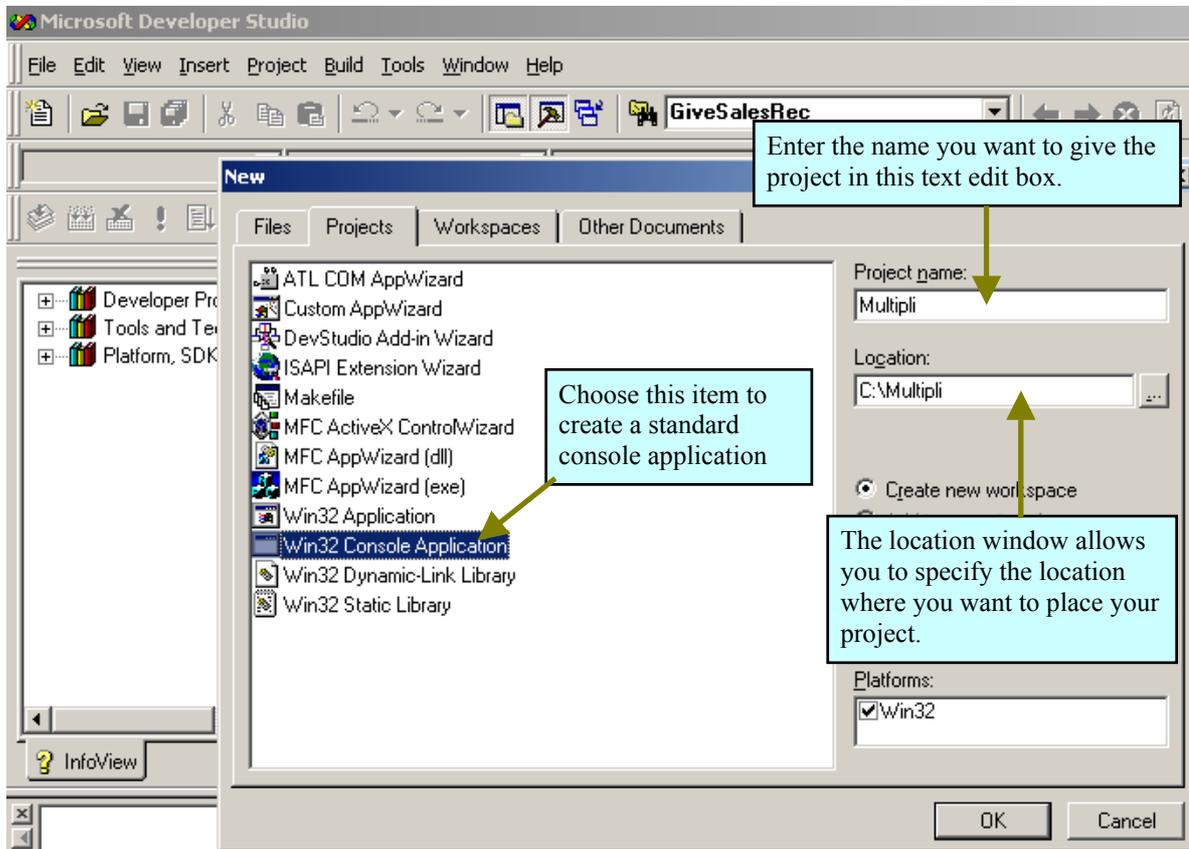
Figure 3-25
The New dialog window allows you to create or add new file
resources to a project.

(1) Type the following in the Project Name text editor box.

Notice that the MSVC IDE set the default location of the project file in the root of the C directory. When you type the project name, MSVC creates a project folder with the same name in the project location directory path.

(**5**) Click the OK button on the **N**ew dialog window.

The dialog window appears as illustrated in figure 3-26. When you choose to create a project space, the MSVC IDE present you with a dialog window that allow you to select configuration options compatible with the Ms Windows operating system.