



Module 819

Inheritance and Virtual Functions in C++

Aim

After working through this module you should be able to define and use class hierarchies, employing inheritance and polymorphism *via* virtual functions.

Learning objectives

After working through this module you should be able to:

1. Incorporate inheritance into C++ programs
2. Apply inheritance to create complex class structures in C++
3. Create protected data structures in C++
4. Create private data structures in C++
5. Create pointers to an object, and to an array of objects, in C++
6. Define multiple inheritance in C++
7. Define virtual functions in C++
8. Define and carry out the steps involved in developing a full-scale OOP project in C++
9. List some of the likely future developments in C++

Content

Inheritance in C++

Public and private data structures

Polymorphism

Virtual functions

New developments in C++

Learning Strategy

Read the printed module and the assigned readings and complete the exercises as requested.

Assessment

Completion of exercises and the CML test at the end of the module.

Introduction

Inheritance, like encapsulation, is central to the development of object-oriented programs. The major point of using object inheritance is to support code reusability: it allows you to reuse code from a previous project, but gives you the flexibility to slightly modify it if the old code does not do exactly what you need for the new project. It makes little sense to start every new project from scratch, since some code (especially that dealing with the interface) will certainly be repeated in several programs and you should strive to build on what you did previously.

Moreover, it is easy to make an error if we try to modify the original class, but we are less likely to make an error if we leave the original alone and only add to it. Another reason for using inheritance is if the project requires the use of several classes which are very similar but slightly different.

In this Module we will focus on the mechanism of inheritance and how to build it into a program. A better illustration of why you would use inheritance will be given later where we will discuss some practical applications of object-oriented programming.

The principle of inheritance is available with several modern programming languages and is handled slightly differently by each. C++ allows you to inherit all or part of the members and methods of a class, modify some, and add new ones not available in the parent class. You have complete flexibility, and as usual, the method used with C++ has been selected to result in the most efficient code execution.

Objective 1 After working through this section you should be able to incorporate inheritance into C++ programs

A simple class

Examine the following header file [VEHICLE.H] for a simple class which we will use to begin our study of inheritance.

```
// vehicle header file
#ifndef VEHICLE_H
#define VEHICLE_H
class vehicle {
protected:
    int wheels;
    float weight;
public:
    void initialize(int in_wheels, float in_weight);
    int get_wheels(void);
    float get_weight(void);
    float wheel_loading(void);
};
#endif
```

The class header consists of four simple methods which can be used to manipulate data about our vehicle. What each method does is not especially important at this time. The vehicle class is actually an example of a *base* class or *parent* class, but for now we will use it like any other class to show that it is equivalent to the classes already studied. We will explain the added keyword *protected* shortly. For now, ignore lines 2, 3, and 14 until the end of this section where they will be explained in detail.

The implementation for vehicle

The next program [VEHICLE.CPP] is the implementation of the vehicle class.

```
#include "vehicle.h"
// initialize to any data desired
void vehicle::initialize(int in_wheels, float in_weight)
{
    wheels = in_wheels;
    weight = in_weight;
} // get the number of wheels of this vehicle
int vehicle::get_wheels()
{
    return wheels;
} // return the weight of this vehicle
float vehicle::get_weight()
{
    return weight;
} // return the weight on each wheel
float vehicle::wheel_loading()
{
    return weight/wheels;
}
```

The initialize() method assigns the values input as parameters to the wheels and weight variables. We have methods to return the number of wheels and the weight, and finally, we have one that does a trivial

calculation to return the loading in each wheel. We will have a few examples of methods that do some significant processing later, but at this point we are more interested in seeing how to set up the interfaces to the classes.

As stated above, this is a simple class that will be used in the next program; eventually we will use it as a base class. You should compile this class now in preparation for the next example program, but you cannot execute it as there is no entry point.

Using the vehicle class

The next program [TRANSPRT.CPP] uses the vehicle class in exactly the same manner as we used the time and date classes in the last Module. This supports the argument that the vehicle class is truly nothing more than a normal C++ class, but we are able to use it unmodified as a base class in the next few example files to illustrate inheritance. Inheritance uses an existing class and adds functionality to it to accomplish another, possibly more complex, job.

```
#include <iostream.h>
#include "vehicle.h"
void main()
{
    vehicle car, motorcycle, truck, sedan;
    car.initialize(4, 3000.0);
    motorcycle.initialize(2, 900.0);
    truck.initialize(18, 45000.0);
    sedan.initialize(4, 3000.0);
    cout << "The car has " << car.get_wheels() << " wheels.\n";
    cout << "The truck has a loading of " << truck.wheel_loading()
        << " pounds per wheel.\n";
    cout << "The motorcycle weighs " << motorcycle.get_weight()
        << " pounds.\n";
    cout << "The sedan weighs " << sedan.get_weight()
        << " pounds, and has " << sedan.get_wheels() << " wheels.\n";
}
```

You should have no problem understanding the operation of this program. It declares four objects of the vehicle class, initialises them, and prints out a few of the data points to illustrate that the vehicle class can be used as a *simple* class because it is a simple class - as opposed to calling it a *base* class or *derived* class as we will do shortly.

A derived class

Examine the next file [CAR.H] for our first example of the use of a *derived* (or *child*) class.

```
#ifndef CAR_H
#define CAR_H
#include "vehicle.h"
class car : public vehicle {
    int passenger_load;
public:
    void initialize(int in_wheels, float in_weight, int people = 4);
    int passengers(void);
};
```

```
#endif
```

The vehicle class is inherited due to the ": public vehicle" added to line 4. This derived class named car is composed of all of the information included in the base class vehicle, and all of its own additional information. Even though we did nothing to the class named vehicle, we made it into a base class because of the way we are using it here. To go a step further, even though it will be used as a base class in an example program later in this section, there is no reason it cannot continue to be used as a simple class in the previous example program. In fact, it can be used as a single class and a base class in the same program. The question of whether it is a simple class or a base class is answered by the way it is used.

A discussion of terminology is needed here. When discussing object-oriented programming in general, a class that inherits another is often called a *derived* class or a *child* class, but the most proper term as defined for C++ is a *derived* class. Since these terms are very descriptive, and most writers tend to use the terms interchangeably, we will also use these terms in this tutorial. Likewise the proper C++ terminology for the inherited class is to call it a base class, but parent class and super class are sometimes used.

A *base* class is a rather general class which can cover a wide range of objects, whereas a derived class is somewhat more restricted but at the same time more useful. For example if we had a base class named programming language and a derived class named C++, then we could use the base class to define Pascal, Ada, C++, or any other programming language, but it would not tell us about the use of classes in C++ because it can only give a general view of each language. On the other hand, the derived class named C++ could define the use of classes, but it could not be used to describe the other languages because it is too narrow. A base class tends to be more general, and a derived class is more specific.

In this case, the vehicle base class can be used to declare objects that represent trucks, cars, bicycles, or any number of other vehicles you can think up. The class named car however can only be used to declare an object that is of type car because we have limited the kinds of data that can be intelligently used with it. The car class is therefore more restrictive and specific than the vehicle class. The vehicle class is more general than the car class.

If we wished to get even more specific, we could define a derived class using car as the base class and name it sports_car and include such information as red_line_limit for the tachometer which would be silly for the family station wagon. The car class would therefore be used as a derived class and a base class at the same time, so it should be clear that these names refer to how a class is used.

Declaring a derived class

A derived class is defined by including the header file for the base class in line 3, then the name of the base class is given following the name of the derived class separated by a colon in line 4. [Ignore the keyword *public* immediately following the colon in this line; it is optional and we will study it in detail in the next section.] All objects declared as being of class `car` therefore are composed of the two variables from the class `vehicle` because they inherit those variables, and the single variable declared in the class `car` named `passenger_load`.

An object of this class will have three of the four methods of `vehicle` and the two new ones declared here. The method named `initialize()` which is part of the `vehicle` class will not be available here because it is hidden by the local version of `initialize()` which is a part of the `car` class. The local method will be used if the name is repeated allowing you to customise your new class.

Note once again that the implementation for the base class only needs to be supplied in its compiled form. The source code for the implementation can be hidden for economic reasons to aid software developers. Hiding the source code also allows the practice of information hiding. The header for the base class must be available as a text file since the class definitions are required in order to use the class.

The car class implementation

Examine the next program [CAR.CPP] which is the implementation file for the `car` class.

```
#include "car.h"
void car::initialize(int in_wheels, float in_weight, int people)
{
    passenger_load = people;
    wheels = in_wheels;
    weight = in_weight;
}
int car::passengers(void)
{
    return passenger_load;
}
```

The first thing you should notice is that this file has no indication of the fact that it is a derived class of any other file; that can only be determined by inspecting the header file for the class. Since we can't tell if it is a derived class or not, it is written in exactly the same way as any other class implementation file.

The implementations for the two new methods are written in exactly the same way as methods are written for any other class. Compile the file for subsequent use.

Another derived class

Examine the following header file [TRUCK.H] for an example of a class that uses the vehicle class and adds to it. Of course, it adds different things to it because it will specialise in those things that relate to trucks. In fact it adds two more variables and three methods. Once again, we will cover the keyword public in detail in the next section of this Module.

```
#ifndef TRUCK_H
#define TRUCK_H
#include "vehicle.h"
class truck : public vehicle {
    int passenger_load;
    float payload;
public:
    void init_truck(int how_many = 2, float max_load = 24000.0);
    float efficiency(void);
    int passengers(void);
};
#endif
```

A very important point that must be made is that the car class and the truck class have absolutely nothing to do with each other, their only connection is that they are both derived classes of the same base class. Note that both the car and the truck classes have methods named passengers() but this causes no problems and is perfectly acceptable. If classes are related in some way (and they certainly are if they are both derived classes of a common base class) you would expect them to be doing somewhat similar things. In this situation there is a good possibility that a method name would be repeated in both child classes.

The truck implementation

The next program [TRUCK.CPP] is the implementation of the truck class.

```
#include "truck.h"
void truck::init_truck(int how_many, float max_load)
{
    passenger_load = how_many;
    payload = max_load;
}
float truck::efficiency(void)
{
    return payload / (payload + weight);
}
int truck::passengers(void)
{
    return passenger_load;
}
```

Compile it in preparation for our next example program, which uses all three of the defined classes.

Using all three classes

We complete our examination of the development of the vehicle, car and truck classes by seeing how we might use the three classes in a basic application program [ALLVEHIC.CPP].

```
#include <iostream.h>
#include "vehicle.h"
#include "car.h"
#include "truck.h"
void main()
{
    vehicle unicycle;
    unicycle.initialize(1, 12.5);
    cout << "The unicycle has "
         << unicycle.get_wheels() << " wheel.\n";
    cout << "The unicycle's wheel loading is "
         << unicycle.wheel_loading()
         << " pounds on the single tire.\n";
    cout << "The unicycle weighs "
         << unicycle.get_weight() << " pounds.\n\n";
    car sedan;
    sedan.initialize(4, 3500.0, 5);
    cout << "The sedan carries " << sedan.passengers()
         << " passengers.\n";
    cout << "The sedan weighs " << sedan.get_weight()
         << " pounds.\n";
    cout << "The sedan's wheel loading is "
         << sedan.wheel_loading() << " pounds per tire.\n\n";
    truck semi;
    semi.initialize(18, 12500.0);
    semi.init_truck(1, 33675.0);
    cout << "The semi weighs " << semi.get_weight()
         << " pounds.\n";
    cout << "The semi's efficiency is "
         << 100.0 * semi. efficiency() << " percent.\n";
}
```

This program uses the parent class vehicle to declare objects and also uses the two child classes to declare objects. This was done to illustrate that all three classes can be used in a single program. All three of the header files for the classes are included in lines 2 to 4 so the program can use the components of the classes. Notice that the implementations of the three classes are not in view here and do not need to be in view. This allows the code to be used without access to the source code for the actual implementation of the class. However, it should be clear that the header file definition must be available.

Here only one object of each class is declared and used, but as many as desired could be declared and used in order to accomplish the programming task at hand. You will notice how clean and uncluttered the source code is, since the classes were developed, debugged, and stored away previously, and the interfaces were kept very simple.

Once again, compiling and executing this program will require the use of the make or project capability of your compiler.

The #Ifndef vehicle_h

When we define the derived class car, we are required to supply it with the full definition of the interface to the vehicle class, since car is a derived class of vehicle and must know all about its parent. We do that by including the vehicle class into the car class, and the car class can be compiled. The vehicle class must also be included in the header file of the truck class for the same reason.

When we get to the main program, we must inform it of the details of all three classes, so all three header files must be included as is done in lines 2 to 4 of ALLVEHIC.CPP, but this leads to a problem. When the preprocessor gets to the car class, it includes the vehicle class because it is listed in the car class header file, but since the vehicle class was already included in line 2 of ALLVEHIC.CPP, it is included twice and we attempt to redefine the class vehicle. Of course it is the same definition, but the system simply will not allow redefinition of a class. We allow the double inclusion of the file and at the same time prevent the double inclusion of the class by building a bridge around it using the word VEHICLE_H. If the word is already defined, the definition is skipped, but if the word is not defined, the definition is included and the word is defined at that time. The end result is the actual inclusion of the class only once, even though the file is included more than once. You should have no trouble understanding the logic of the includes if you spend a little time studying this program sequence.

Even though ANSI-C allows multiple definitions of entities, provided the definitions are identical, C++ does not permit this. The primary reason is because the compiler would have great difficulty in knowing if it has already made a constructor call for the redefined entity, if there is one. A multiple constructor call for a single object could cause great havoc, so C++ was defined to prevent any multiple constructor calls by making it illegal to redefine any entity. This is not a problem in any practical program.

The name VEHICLE_H was chosen as the word because it is the name of the file, with the period replaced by the underline. If the name of the file is used systematically in all of your class definitions, you cannot have a name clash because the filename of every class must be unique. It would be good for you to get into the practice of building the optional skip around all of your classes. All class definition files in the remainder of this tutorial will include this skip around to prevent multiple inclusions and to be an example for you. You should get into the practice of adding the skip around to all of your class headers no matter how trivial they may seem to be.

Practical inheritance

We will inherit the date class into the file below [NEWDATE.H] and add a member variable and a new method to the class.

```
// This class inherits the date class and adds one variable and
// one
// method to it.
#ifndef NEWDATE_H
#define NEWDATE_H
#include "date.h"
class new_date : public date {
protected:
    int day_of_year;           // New member variable
public:
    int get_day_of_year(void); // New method
};
#endif
```

Actually, this is not a good way to add the day_of_year to the date class since it is available in the structure returned from the system call in the date class. However, we are more interested in illustrating inheritance in a practical example than we are in developing a perfect class, so we will live with this inefficiency. You will note that we add one variable and one method to create our new class.

The program below [NEWDATE.CPP] contains the implementation for the added method and should be simple for the student to understand. This class implementation uses the array days[] from the date class implementation since it was defined as a global variable there. The method named get_time_of_day() involves very simple logic but still adjusts for leap years.

```
#include "newdate.h"
extern int days[];
// This routine ignores leap year for simplicity, and adds
// the days in each month for all months less than the
// current month, then adds the days in the current month
// up to today.
int new_date::get_day_of_year(void)
{
    int index = 0;
    day_of_year = 0;
    while (index < month)
        day_of_year += days[index++];
    return (day_of_year += day);
}
```

Finally, the next example program [TRYNDATE.CPP] will use the new class in a very simple way to illustrate that the derived class is as easy to use as the base class; in fact, the main program has no way of knowing that it is using a derived class.

```
#include <iostream.h>
#include "newdate.h"
void main()
{
    new_date now, later, birthday;
    later.set_date(12, 31, 1991);
    birthday.set_date(2, 19, 1991);
    cout << "Today is day " << now.get_day_of_year() << "\n";
    cout << "Dec 31 is day " << later.get_day_of_year() << "\n";
}
```

```
cout << "Feb 19 is day " << birthday.get_day_of_year() << "\n";  
}
```

In order to compile and link this program it will be necessary to link in the object code for the original date class, as well as the object code from the newdate class and the main program.

Exercise 1

Modify ALLVEHIC.CPP by adding a new object of the vehicle class named *bicycle*, including operations equivalent to those for the unicycle object.\

Add a new method to the truck class (in TRUCK.CPP) to return the total weight of the truck and its payload. Add appropriate code to ALLVEHIC.CPP to read the value out and display it on the monitor.

Modify the name class from the previous module to include a variable named *gender* (of type char, taking only the values 'M' or 'F'), including methods to set and retrieve the value of the variable.

Objective 2 After working through this section you should be able to apply inheritance to create complex class structures in C++

In the last section we developed a model using modes of transportation to illustrate the concept of inheritance. In this section we will use that model to illustrate some of the finer points of inheritance and what it can be used for.

Reorganised file structure

A close examination of the first program below [INHERIT1.CPP] will reveal that it is apparently identical to the program developed in the last section (allvehic.cpp) except that the program text is rearranged. The biggest difference is that some of the simpler methods in the classes have been changed to inline code to shorten the file considerably. In a practical programming situation, methods that are this short should be programmed inline since the actual code to return a simple value is shorter than the code required to send a message to a non-inline method.

```
#include <iostream.h>
class vehicle {
protected:
    int wheels;
    float weight;
public:
    void initialize(int in_wheels, float in_weight);
    int get_wheels(void) {return wheels;}
    float get_weight(void) {return weight;}
    float wheel_loading(void) {return weight/wheels;}
};
class car : public vehicle {
    int passenger_load;
public:
    void initialize(int in_wheels, float in_weight, int people = 4);
    int passengers(void) {return passenger_load;}
};
class truck : public vehicle {
    int passenger_load;
    float payload;
public:
    void init_truck(int how_many = 2, float max_load = 24000.0);
    float efficiency(void);
    int passengers(void) {return passenger_load;}
};
void main()
{
    vehicle unicycle;
    unicycle.initialize(1, 12.5);
    cout << "The unicycle has " << unicycle.get_wheels()
        << " wheel.\n";
    cout << "The unicycle's wheel loading is "
        << unicycle.wheel_loading()
        << " pounds on the single tire.\n";
    cout << "The unicycle weighs " << unicycle.get_weight()
        << " pounds.\n\n";
    car sedan;
    sedan.initialize(4, 3500.0, 5);
    cout << "The sedan carries " << sedan.passengers()
        << " passengers.\n";
}
```

```
cout << "The sedan weighs " << sedan.get_weight()
      << " pounds.\n";
cout << "The sedan's wheel loading is "
      << sedan.wheel_loading() << " pounds per tire.\n\n";
truck semi;
semi.initialize(18, 12500.0);
semi.init_truck(1, 33675.0);
cout << "The semi weighs " << semi.get_weight() << " pounds.\n";
cout << "The semi's efficiency is "
      << 100.0 * semi. efficiency() << " percent.\n";
} // initialize to any data desired
void
vehicle::initialize(int in_wheels, float in_weight)
{
    wheels = in_wheels;
    weight = in_weight;
}
void
car::initialize(int in_wheels, float in_weight, int people)
{
    passenger_load = people;
    wheels = in_wheels;
    weight = in_weight;
}
void
truck::init_truck(int how_many, float max_load)
{
    passenger_load = how_many;
    payload = max_load;
}
float
truck::efficiency(void)
{
    return payload / (payload + weight);
}
```

The only other change is the reordering of the classes and associated methods with the classes all defined first, followed by the main program. This puts all class interface definitions on a single page to make the code easier to study. The implementations for the methods are deferred to the end of the file where they are available for quick reference but are not cluttering up the class definitions which are the focus of this section. There is evidently considerable flexibility in the way the classes and methods can be arranged in C++. Of course, this violates the point of separate compilation in C++, and is done here for convenience. The best way to package all of the example programs in this section is like the packaging illustrated in the previous section.

As mentioned before, the two derived classes, car and truck, each have a variable named `passenger_load` which is perfectly legal, and the car class has a method of the same name, `initialize()`, as one defined in the super-class named `vehicle`. The rearrangement of the files in no way prevents this allowable repeating of names.

After you have convinced yourself that this program is truly identical to the program `allvehic.cpp` from the last section, compile and execute it to prove that this arrangement is legal. In this form of code packaging, you will not need a `make` or a project file to compile and execute this code.

The scope operator

Because the method `initialize()` is defined in the derived car class, it hides the method of the same name which is part of the base class, and there may be times you wish to send a message to the method in the base class for use in the derived class object. This can be done by using the scope operator in the following manner in the main program:

```
sedan.vehicle::initialize(4, 3500.0);
```

As you might guess, the number and types of parameters must agree with those of the method in the base class because it will respond to the message.

Hidden methods

Examine the next program [INHERIT2.CPP] and you will notice that it is a repeat of the last example program with a few minor changes.

```
#include <iostream.h>
class vehicle {
protected:
    int wheels;
    float weight;
public:
    void initialize(int in_wheels, float in_weight);
    int get_wheels(void) {return wheels;}
    float get_weight(void) {return weight;}
    float wheel_loading(void) {return weight/wheels;}
};
class car : vehicle {
int passenger_load;
public:
void initialize(int in_wheels, float in_weight, int people = 4);
int passengers(void) {return passenger_load;}
};
class truck : vehicle {
    int passenger_load;
    float payload;
public:
    void init_truck(int how_many = 2, float max_load = 24000.0);
    float efficiency(void);
    int passengers(void) {return passenger_load;}
};
void main()
{
vehicle unicycle;
unicycle.initialize(1, 12.5);
cout << "The unicycle has " << unicycle.get_wheels()
    << " wheel.\n";
cout << "The unicycle's wheel loading is "
    << unicycle.wheel_loading()
    << " pounds on the single tire.\n";
cout << "The unicycle weighs " << unicycle.get_weight()
    << " pounds.\n\n";
car sedan;
sedan.initialize(4, 3500.0, 5);
cout << "The sedan carries " << sedan.passengers()
    << " passengers.\n";
// cout << "The sedan weighs " << sedan.get_weight()
//     << " pounds.\n";
// cout << "The sedan's wheel loading is " <<
//     sedan.wheel_loading() << " pounds per tire.\n\n";
truck semi;
```

```
// semi.initialize(18, 12500.0);
semi.init_truck(1, 33675.0);
// cout << "The semi weighs " << semi.get_weight()
// << " pounds.\n";
// cout << "The semi's efficiency is " <<
// 100.0 * semi. efficiency() << " percent.\n";
}
// initialize to any data desired
void
vehicle::initialize(int in_wheels, float in_weight)
{
wheels = in_wheels;
weight = in_weight;
}
void
car::initialize(int in_wheels, float in_weight, int people)
{
passenger_load = people;
wheels = in_wheels;
weight = in_weight;
}
void
truck::init_truck(int how_many, float max_load)
{
passenger_load = how_many;
payload = max_load;
}
float
truck::efficiency(void)
{
return payload / (payload + weight);
}
```

You will notice that the derived classes named `car` and `truck` do not have the keyword `public` prior to the name of the base class in the first line of each. The keyword `public`, when included *prior* to the base class name, makes all of the methods defined in the base class available for use in the derived class just as if they were defined as part of the derived class. Therefore, in the previous program, we were permitted to call the methods defined as part of the base class from the main program even though we were working with an object of one of the derived classes. One example of when we did this was when we sent a message to the sedan to get its weight in an output statement of the main program.

In the present program, without the keyword `public` prior to the base class name, the only methods available for objects of the `car` class, are those that are defined as part of the class itself, and therefore we only have the methods named `initialize()` and `passengers()` available for use with objects of class `car`. In this program, the only inheritance is that of variables since the two variables are inherited into objects of class `car`.

When we declare an object of type `car`, according to the definition of the C++ language, it contains three variables. It contains the one defined as part of its class named `passenger_load` and the two that are part of its parent class, `wheels` and `weight`. All are available for direct use within its methods because of the use of the keyword `protected` in the base class. The variables are a part of an object of class `car` when it is declared and are stored as part of the object. We will show you the

details of access to the parent class variables within derived classes shortly. For now, we will return to the use of the subclasses in this example program.

You should have noticed that several of the output statements have been commented out of the main program since they are no longer legal or meaningful operations. Lines 41 through 43 have been commented out because the methods named `get_weight()` and `wheel_loading()` are not inherited into the car class without the keyword `public` in the car class definition. You will also notice that `initialize()` is still available - but this is the one in the car class, not the method of the same name in the vehicle class.

Moving on to the use of the truck class in the main program, we find that lines 46 and 48 are commented out for the same reason as given above, but lines 49 and 50 are commented out for an entirely different reason. Even though the method named `efficiency()` is available and can be called as a part of the truck class, it cannot be used because we have no way to initialise the wheels or weight of the truck objects. We can get the weight of the truck objects, as we have done in line 48, but since the weight has no way to be initialised, the result is meaningless and lines 49 and 50 are commented out.

As you have surely guessed by now, there is a way around all of these problems and we will cover them shortly. In the meantime, be sure to compile and execute this example program to see that your compiler gives the same result. It would be a good exercise for you to reintroduce some of the commented out lines to see what sort of an error message your compiler issues for these errors.

Initialising all data

If you will examine the next example program [INHERIT3.CPP] you will find that we have fixed the initialisation problem that was left undetermined in the last example program.

```
#include <iostream.h>
class vehicle {
protected:
    int wheels;
    float weight;
public:
    void initialize(int in_wheels, float in_weight);
    int get_wheels(void) {return wheels;}
    float get_weight(void) {return weight;}
    float wheel_loading(void) {return weight/wheels;}
};
class car : vehicle {
    int passenger_load;
public:
    void initialize(int in_wheels, float in_weight, int people = 4);
    int passengers(void) {return passenger_load;}
};
class truck : vehicle {
    int passenger_load;
```

```
float payload;
public:
void init_truck(int in_wheels, float in_weight,
               int how_many = 2, float max_load = 24000.0);
float efficiency(void);
int passengers(void) {return passenger_load;}
};
void main()
{
vehicle unicycle;
unicycle.initialize(1, 12.5);
cout << "The unicycle has "
      << unicycle.get_wheels() << " wheel.\n";
cout << "The unicycle's wheel loading is "
      << unicycle.wheel_loading()
      << " pounds on the single tire.\n";
cout << "The unicycle weighs "
      << unicycle.get_weight() << " pounds.\n\n";
car sedan;
sedan.initialize(4, 3500.0, 5);
cout << "The sedan carries " << sedan.passengers()
      << " passengers.\n";
// cout << "The sedan weighs " << sedan.get_weight()
//      << " pounds.\n";
// cout << "The sedan's wheel loading is " <<
// sedan.wheel_loading() << " pounds per tire.\n\n";
truck semi;
// semi.initialize(18, 12500.0);
semi.init_truck(1, 33675.0);
// cout << "The semi weighs " << semi.get_weight()
//      << " pounds.\n";
cout << "The semi's efficiency is "
      << 100.0 * semi.efficiency() << " percent.\n";
}
// initialize to any data desired
void
vehicle::initialize(int in_wheels, float in_weight)
{
wheels = in_wheels;
weight = in_weight;
}
void
car::initialize(int in_wheels, float in_weight, int people)
{
passenger_load = people;
wheels = in_wheels;
weight = in_weight;
}
void
truck::init_truck(int in_wheels, float in_weight,
                 int how_many, float max_load)
{
vehicle::initialize(in_wheels, in_weight);
passenger_load = how_many;
payload = max_load;
}
float
truck::efficiency(void)
{
return payload / (payload + weight);
}
```

The method named `init_truck()` now contains all four of the parameters as input data which get transferred to the four variables. Following the initialisation, it is permissible to call the `semi.efficiency()` method in the main program.

Exercise 2

Remove the comment delimiters from lines 48-49 in INHERIT2.CPP

```
// cout << "The semi's efficiency is " <<  
// 100.0 * semi. efficiency() << " percent.\n";
```

and describe the results of executing the re-compiled program. Remove the comments from line 41-42; describe and explain the consequent compiler error.

```
// cout << "The sedan weighs " << sedan.get_weight()  
// << " pounds.\n";
```

Objective 3 After working through this section you should be able to create protected data structures in C++

The next program [INHERIT4.CPP] contains an example we will use to define *protected* data. Just to make the program more versatile, we have returned to the use of the keyword `public` prior to the name of the parent classes in lines 15 and 23 of the class definitions.

```
#include <iostream.h>
class vehicle {
protected:
    int wheels;
    float weight;
public:
    void initialize(int in_wheels, float in_weight);
    int get_wheels(void) {return wheels;}
    float get_weight(void) {return weight;}
    float wheel_loading(void) {return weight/wheels;}
};
class car : public vehicle {
private:
    int passenger_load;
public:
    void initialize(int in_wheels, float in_weight, int people = 4);
    int passengers(void) {return passenger_load;}
};
class truck : public vehicle {
private:
    int passenger_load;
    float payload;
public:
    void init_truck(int how_many = 2, float max_load = 24000.0);
    float efficiency(void);
    int passengers(void) {return passenger_load;}
};
void main()
{
    vehicle unicycle;
    unicycle.initialize(1, 12.5);
    cout << "The unicycle has " << unicycle.get_wheels()
        << " wheel.\n";
    cout << "The unicycle's wheel loading is "
        << unicycle.wheel_loading()
        << " pounds on the single tire.\n";
    cout << "The unicycle weighs " <<
        unicycle.get_weight() << " pounds.\n\n";
    car sedan;
    sedan.initialize(4, 3500.0, 5);
    cout << "The sedan carries " << sedan.passengers() <<
        " passengers.\n";
    cout << "The sedan weighs " << sedan.get_weight() << "
    pounds.\n";
    cout << "The sedan's wheel loading is " <<
        sedan.wheel_loading() << " pounds per tire.\n\n";
    truck semi;
    semi.initialize(18, 12500.0);
    semi.init_truck(1, 33675.0);
    cout << "The semi weighs " << semi.get_weight() << " pounds.\n";
    cout << "The semi's efficiency is "
        << 100.0 * semi.efficiency() << " percent.\n";
}
// initialize to any data desired
void
vehicle::initialize(int in_wheels, float in_weight)
{
```

```
wheels = in_wheels;
weight = in_weight;
}
void
car::initialize(int in_wheels, float in_weight, int people)
{
    passenger_load = people;
    wheels = in_wheels;
    weight = in_weight;
}
void
truck::init_truck(int how_many, float max_load)
{
    passenger_load = how_many;
    payload = max_load;
}
float
truck::efficiency(void)
{
    return payload / (payload + weight);
}
```

If the data within a base class were totally available in all classes inheriting that base class, it would be a simple matter for a programmer to inherit the superclass into a derived class and have free access to all data in the parent class. This would completely override the protection afforded by the use of information hiding. For this reason, the data in a class are not automatically available to the methods of an inheriting class.

There are times when you may wish to automatically inherit all variables directly into the subclasses and have them act just as though they were defined as a part of those classes also. For this reason, the designer of C++ has provided the keyword `protected`.

In the present example program, the keyword *protected* is given in line 3 so that all of the data of the vehicle class can be directly imported into any derived classes but are not available outside of the class or derived classes. All data are automatically defaulted to private type if no specifier is given. The keyword `private` can be used as illustrated in lines 13 and 20 but adds nothing due to the fact that class members default to private by definition.

You will notice that the variables named `wheels` and `weight` are available to use in the method named `initialize()` just as if they were declared as a part of the car class itself. Table 1 summarizes the rules for the three means of defining variables and methods.

<i>private</i>	the variables and methods are not available to any outside calling routines, and they are not available to any derived classes inheriting this class.
<i>protected</i>	the variables and methods are not available to any outside calling routines, but they are directly available to any derived class inheriting this class.
<i>public</i>	all variables and methods are freely available to all outside calling routines and to all derived classes.

Table 1: Rules For Defining Variables And Methods

You will note that these three means of definition can also be used in a struct type. The only difference with a struct is that everything defaults to public until one of the other keywords is used.

Exercise 3

Explain in your own words the key difference between methods defined as private and those defined as public

Objective 4 After working through this section you should be able to create private data structures in C++

In the program below [INHERIT5.CPP] the data are allowed to use the *private* default. In this program, the data are not available for use in the derived classes, so the only way the data in the base class can be used is through the use of messages to methods in the base class.

```
#include <iostream.h>
class vehicle {
    int wheels;
    float weight;
public:
    void initialize(int in_wheels, float in_weight);
    int get_wheels(void) {return wheels;}
    float get_weight(void) {return weight;}
    float wheel_loading(void) {return weight/wheels;}
};
class car : public vehicle {
private:
    int passenger_load;
public:
    void initialize(int in_wheels, float in_weight, int people = 4);
    int passengers(void) {return passenger_load;}
};
class truck : public vehicle {
private:
    int passenger_load;
    float payload;
public:
    void init_truck(int how_many = 2, float max_load = 24000.0);
    float efficiency(void);
    int passengers(void) {return passenger_load;}
};
void main()
{
    vehicle unicycle;
    unicycle.initialize(1, 12.5);
    cout << "The unicycle has " <<
        unicycle.get_wheels() << " wheel.\n";
    cout << "The unicycle's wheel loading is "
        << unicycle.wheel_loading()
        << " pounds on the single tire.\n";
    cout << "The unicycle weighs "
        << unicycle.get_weight() << " pounds.\n\n";
    car sedan;
    sedan.initialize(4, 3500.0, 5);
    cout << "The sedan carries " << sedan.passengers()
        << " passengers.\n";
    cout << "The sedan weighs " << sedan.get_weight()
        << " pounds.\n";
    cout << "The sedan's wheel loading is "
        << sedan.wheel_loading() << " pounds per tire.\n\n";
    truck semi;
    semi.initialize(18, 12500.0);
    semi.init_truck(1, 33675.0);
    cout << "The semi weighs " << semi.get_weight() << " pounds.\n";
    cout << "The semi's efficiency is "
        << 100.0 * semi.efficiency() << " percent.\n";
}

// initialize to any data desired
void
vehicle::initialize(int in_wheels, float in_weight)
{
    wheels = in_wheels;
    weight = in_weight;
}
```

```
    }  
    void  
    car::initialize(int in_wheels, float in_weight, int people)  
    {  
    passenger_load = people;  
    vehicle::initialize(in_wheels, in_weight);  
    }  
    void  
    truck::init_truck(int how_many, float max_load)  
    {  
    passenger_load = how_many;  
    payload = max_load;  
    }  
    float  
    truck::efficiency(void)  
    {  
    return payload / (payload + get_weight());  
    }
```

It may seem a little odd to have to call methods in the base class to get to the data which is actually a part of the derived class, but that is the way C++ is defined to work. This would indicate to you that you should spend some time thinking about how any class you define will be used. If you think somebody may wish to inherit your class into a new class and expand it, you should make the data members protected so they can be easily used in the new class.

Inheriting constructors

The next example program [INHERIT6.CPP] is another variation of our basic program, this time adding constructors.

```
#include <iostream.h>  
class vehicle {  
protected:  
    int wheels;  
    float weight;  
public:  
    vehicle(void) {wheels = 7; weight = 11111.0;}  
    void initialize(int in_wheels, float in_weight);  
    int get_wheels(void) {return wheels;}  
    float get_weight(void) {return weight;}  
    float wheel_loading(void) {return weight/wheels;}  
};  
class car : public vehicle {  
    int passenger_load;  
public:  
    car(void) {passenger_load = 4;}  
    void initialize(int in_wheels, float in_weight, int people = 4);  
    int passengers(void) {return passenger_load;}  
};  
class truck : public vehicle {  
    int passenger_load;  
    float payload;  
public:  
    truck(void) {passenger_load = 3; payload = 22222.0;}  
    void init_truck(int how_many = 2, float max_load = 24000.0);  
    float efficiency(void);  
    int passengers(void) {return passenger_load;}  
};  
void main()  
{  
    vehicle unicycle;  
    // unicycle.initialize(1, 12.5);  
    cout << "The unicycle has " <<
```

```

        unicycle.get_wheels() << " wheel.\n";
    cout << "The unicycle's wheel loading is " <<
        unicycle.wheel_loading() << " pounds on the single
tire.\n";
    cout << "The unicycle weighs " <<
        unicycle.get_weight() << " pounds.\n\n";
car sedan;
// sedan.initialize(4, 3500.0, 5);
    cout << "The sedan carries " << sedan.passengers() <<
        " passengers.\n";
    cout << "The sedan weighs " << sedan.get_weight() << "
pounds.\n";
cout << "The sedan's wheel loading is "
    << sedan.wheel_loading() << " pounds per tire.\n\n";
truck semi;
// semi.initialize(18, 12500.0);
// semi.init_truck(1, 33675.0);
cout << "The semi weighs " << semi.get_weight() << " pounds.\n";
cout << "The semi's efficiency is "
    << 100.0 * semi. efficiency() << " percent.\n";
}
// initialize to any data desired
void
vehicle::initialize(int in_wheels, float in_weight)
{
wheels = in_wheels;
weight = in_weight;
}
void
car::initialize(int in_wheels, float in_weight, int people)
{
passenger_load = people;
wheels = in_wheels;
weight = in_weight;
}
void
truck::init_truck(int how_many, float max_load)
{
passenger_load = how_many;
payload = max_load;
}
float
truck::efficiency(void)
{
return payload / (payload + weight);
}

```

The vehicle class has a constructor to initialise the number of wheels and the weight to the indicated values and has no surprising constructs. The car and truck classes each have a constructor also to initialise their unique variables to some unique values. If you jump ahead to the main program, you will find that the initialising statements are commented out for each of the objects so we must depend on the constructors to initialise the variables. The most important thing to glean from this example is that when one of the constructors is called for a derived class, the constructor is also called for the parent class. In fact, the constructor for the parent class will be called before the constructor for the derived class is called. All of the data will be initialised, including the data inherited from the parent class. We will say much more about constructors used with inheritance in the next section of this Module.

Exercise 4

Add cout statements to each of the constructors in INHERIT5.CPP to output messages to show the order in which messages are sent to the constructors. Describe and explain the results.

Objective 5 After working through this section you should be able to create pointers to an object, and to an array of objects, in C++

The program which follows [INHERIT7.CPP] includes examples of the use of an array of objects and a pointer to an object. In this program, the objects are instantiated from an inherited class and the intent of this program is to illustrate that there is nothing magical about a derived class.

```
#include <iostream.h>
class vehicle {
protected:
    int wheels;
    float weight;
public:
    void initialize(int in_wheels, float in_weight);
    int get_wheels(void) {return wheels;}
    float get_weight(void) {return weight;}
    float wheel_loading(void) {return weight/wheels;}
};
class car : public vehicle {
int passenger_load;
public:
void initialize(int in_wheels, float in_weight, int people = 4);
int passengers(void) {return passenger_load;}
};
class truck : public vehicle {
int passenger_load;
float payload;
public:
void init_truck(int how_many = 2, float max_load = 24000.0);
float efficiency(void);
int passengers(void) {return passenger_load;}
};
void main()
{
vehicle unicycle;
unicycle.initialize(1, 12.5);
cout << "The unicycle has "
    << unicycle.get_wheels() << " wheel.\n";
cout << "The unicycle's wheel loading is "
    << unicycle.wheel_loading()
    << " pounds on the single tire.\n";
cout << "The unicycle weighs "
    << unicycle.get_weight() << " pounds.\n\n";
car sedan[3];
int index;
for (index = 0 ; index < 3 ; index++) {
    sedan[index].initialize(4, 3500.0, 5);
    cout << "The sedan carries " << sedan[index].passengers()
        << " passengers.\n";
    cout << "The sedan weighs " << sedan[index].get_weight()
        << " pounds.\n";
    cout << "The sedan's wheel loading is "
        << sedan[index].wheel_loading()
        << " pounds per tire.\n\n";
}
truck *semi;
semi = new truck;
semi->initialize(18, 12500.0);
semi->init_truck(1, 33675.0);
cout << "The semi weighs " << semi->get_weight() << " pounds.\n";
cout << "The semi's efficiency is "
    << 100.0 * semi->efficiency() << " percent.\n";
delete semi;
}
```

```
    } // initialize to any data desired
void
vehicle::initialize(int in_wheels, float in_weight)
{
wheels = in_wheels;
weight = in_weight;
}
void
car::initialize(int in_wheels, float in_weight, int people)
{
passenger_load = people;
wheels = in_wheels;
weight = in_weight;
}
void
truck::init_truck(int how_many, float max_load)
{
passenger_load = how_many;
payload = max_load;
}
float
truck::efficiency(void)
{
return payload / (payload + weight);
}
```

The program is identical to the first program in this section until we get to the main program where we find an array of 3 objects of class car declared. It should be obvious that any operation that is legal for a simple object is legal for an object that is part of an array, but we must be sure to tell the system which object of the array we are interested in by adding the array subscript. The operation of this portion of the program should be easy to follow, so we will go on to the next construct of interest.

You will notice that we do not declare an object of type truck, but rather a *pointer* to an object of type truck. In order to use the pointer, we must give it something to point at, which we do by dynamically allocating an object. Once the pointer has an object to point to, we can use the object in the same way we would use any object, but we must use the pointer notation to access any of the methods of the object. Finally, we deallocate the object.

Exercise 5

Create a newtime class (in the series date, time, newdate ...) that inherits the time class and adds a new member variable named *seconds_today* and a method to calculate the number of seconds since midnight to fill the variable.

Objective 6 After working through this section you should be able define multiple inheritance in C++

C++ version 2.0 was released by AT&T during the summer of 1989, and the major addition to the language was *multiple inheritance*: the ability to inherit data and methods from more than one class into a subclass. Multiple inheritance and a few of the other additions to the language will be discussed in this section along with some of the expected future directions of the language.

Multiple inheritance

A major recent addition to the C++ language is the ability to inherit methods and variables from two or more parent classes when building a new class. This is called *multiple inheritance*, and is claimed by many people to be an essential component of any object-oriented language. Some writers, however, have expressed doubts as to its usefulness. As if to support this view, it was difficult to think up a good practical example of the use of multiple inheritance as an illustration for this section. In fact, the resulting example really does nothing useful, but it shows the mechanics of the use of multiple inheritance with C++, and that is our primary concern.

The biggest problem with multiple inheritance involves the inheritance of variables or methods from two or more parent classes *with the same name*. Which variable or method should be chosen as the inherited variable or method if two or more have the same name? This will be illustrated in the next few example programs.

Simple multiple inheritance

An examination of the next program [MULTINH1.CPP] will reveal the definition of two very simple classes (in lines 3 through 24) named `moving_van` and `driver`.

```
#include <iostream.h>
class moving_van
{
protected:
    float payload;
    float gross_weight;
    float mpg;
public:
    void initialize(float pl, float gw, float in_mpg)
    {
        payload = pl;
        gross_weight = gw;
        mpg = in_mpg; };
    float efficiency(void)
    {
        return(payload / (payload + gross_weight));
    };
    float cost_per_ton(float fuel_cost)
```

```
    {
        return(fuel_cost / (payload / 2000.0)); }
};
class driver {
protected:
    float hourly_pay;
public:
    void initialize(float pay)
    {
        hourly_pay = pay;
    };
    float cost_per_mile(void)
    {
        return(hourly_pay / 55.0);
    } ;
};
class driven_truck : public moving_van, public driver
{
public:
    void initialize_all(float pl, float gw, float in_mpg,
        float pay)
    {
        payload = pl;
        gross_weight = gw;
        mpg = in_mpg;
        hourly_pay = pay;
    };
    float cost_per_full_day(float cost_of_gas)
    {
        return(8.0 * hourly_pay + * cost_of_gas * 55.0 / mpg);
    };
};
void main()
{
    driven_truck chuck_ford;
    chuck_ford.initialize_all(20000.0, 12000.0, 5.2, 12.50);
    cout << "The efficiency of the Ford is "
        << chuck_ford. efficiency() << "\n";
    cout << "The cost per mile for Chuck to drive is "
        << chuck_ford.cost_per_mile() << "\n";
    cout << "The cost of Chuck driving the Ford for a day is "
        << chuck_ford.cost_per_full_day(1.129) << "\n";
}
```

In order to keep the program as simple as possible, all of the member methods are defined as inline functions. This puts the code for the methods where it is easy to find and study. You will also notice that all variables in both classes are declared to be protected so they will be readily available for use in any class which inherits them. The code for each class is kept very simple so that we can concentrate on studying the interface to the methods rather than spending time trying to understand complex methods.

In line 35 we define another class named `driven_truck` which inherits all of the data and all of the methods from both of the previously defined classes. In the last two sections, we studied how to inherit a single class into another class, and to inherit two or more classes, the same technique is used except that we use a list of inherited classes separated by commas as illustrated in line 35. You will notice that we use the keyword `public` prior to the name of each inherited class in order to be able to freely use the methods within the subclass. In this case, we didn't define any new variables, but we did introduce two new methods into

the subclass in lines 35 to 50. We declared an object named `chuck_ford` which presumably refers to someone named Chuck who is driving a Ford moving van. The object named `chuck_ford` is composed of four variables, three from the `moving_van` class, and one from the `driver` class. Any of these four variables can be manipulated in any of the methods defined within the `driven_truck` class in the same way as in a singly inherited situation. A few examples are given in lines 53 to 60 of the main program, and you should be able to add additional output messages to this program if you understand the principles involved.

All of the rules for private or protected variables and public or private method inheritance as used with single inheritance extends to multiple inheritance.

Duplicated method names

You will notice that both of the parent classes have a method named `initialize()`, and both of these are inherited into the subclass with no difficulty. However, if we attempt to send a message to one of these methods the system will not know to which we are referring. This problem will be solved and illustrated in the next example program.

It should first be noted that we have not declared any objects of the two parent classes in the main program. Since the two parent classes are simply normal classes themselves, it should be apparent that there is nothing different about them, and they can be used to define and manipulate objects in the usual fashion.

More duplicate method names

The second example program in this section [MULTINH2.CPP] illustrates the use of classes with duplicate method names being inherited into a derived class.

```
#include <iostream.h>
class moving_van {
protected:
    float payload;
    float gross_weight;
    float mpg;
public:
    void initialize(float pl, float gw, float in_mpg)
    {
        payload = pl;
        gross_weight = gw;
        mpg = in_mpg;
    };
    float efficiency(void)
    {
        return(payload / (payload + gross_weight));
    };
    float cost_per_ton(float fuel_cost)
    {
        return(fuel_cost / (payload / 2000.0));
    };
    float cost_per_full_day(float cost_of_gas)
```

```
    {
        return(8.0 * cost_of_gas * 55.0 / mpg);
    };
};
class driver {
protected:
    float hourly_pay;
public:
    void initialize(float pay)
    {
        hourly_pay = pay;
    };
    float cost_per_mile(void)
    {
        return(hourly_pay / 55.0);
    };
    float cost_per_full_day(float overtime_premium)
    {
        return(8.0 * hourly_pay);
    };
};
class driven_truck : public moving_van, public driver {
public:
    void initialize_all(float pl, float gw, float in_mpg, float pay)
    {
        payload = pl;
        gross_weight = gw;
        mpg = in_mpg;
        hourly_pay = pay; };
    float cost_per_full_day(float cost_of_gas)
    {
        return(8.0 * hourly_pay + 8.0 * cost_of_gas * 55.0 / mpg);
    };
};
void main()
{
    driven_truck chuck_ford;
    chuck_ford.initialize_all(20000.0, 12000.0, 5.2, 12.50);
    cout << "The efficiency of the Ford is "
        << chuck_ford. efficiency() << "\n";
    cout << "The cost per mile for Chuck to drive is "
        << chuck_ford.cost_per_mile() << "\n";
    cout << "The cost per day for the Ford is "
        << chuck_ford.moving_van::cost_per_full_day(1.129) <<
        "\n";
    cout << "The cost of Chuck for a full day is "
        << chuck_ford.driver::cost_per_full_day(15.75)
        << "\n";
    cout << "The cost of Chuck driving the Ford for a day is "
        << chuck_ford.driven_truck::cost_per_full_day(1.129)
        << "\n";
}
```

If you study the code, you will find that a new method has been added to all three of the classes named `cost_per_full_day()`. This was done intentionally to illustrate how the same method name can be used in all three classes. The class definitions are straightforward, and the methods are simply named and defined as shown. The problem comes when we wish to use one of the methods since they are all the same name, have the same numbers and types of parameters, and identical return types. This prevents our using an overloading rule to disambiguate the message sent to one or more of the methods.

The process used to disambiguate the method calls is illustrated in lines 64, 67, and 70 of the main program. The solution is to prepend the class name to the method name with the double colon as used in the method

implementation definition. This is referred to as *qualifying* the method name. Qualification is not necessary in line 58 since it is the method in the derived class and it will take precedence over the other method names. Actually, you could qualify all method calls, but if the names are unique, the compiler can do it for you and make your code easier to write and read.

You will notice that there is a slight discrepancy in the results given in lines 59 to 71, since the first two values do not add up to the third value exactly. This is due to the limited precision of the float variable but should cause no real problem.

Duplicated variable names

If you examine the following example program [MULTINH3.CPP] you will notice that each base class has a variable with the same name.

```
#include <iostream.h>
class moving_van {
protected:
    float payload;
    float weight;
    float mpg;
public:
    void initialize(float pl, float gw, float in_mpg)
    {
        payload = pl;
        weight = gw;
        mpg = in_mpg;
    };
    float efficiency(void)
    {
        return(payload / (payload + weight));
    };
    float cost_per_ton(float fuel_cost)
    {
        return(fuel_cost / (payload / 2000.0));
    };
};
class driver {
protected:
    float hourly_pay;
    float weight;
public:
    void initialize(float pay, float in_weight)
    {
        hourly_pay = pay;
        weight = in_weight;
    };
    float cost_per_mile(void)
    {
        return(hourly_pay / 55.0);
    };
    float drivers_weight(void)
    {
        return(weight);
    };
};
class driven_truck : public moving_van, public driver
{
public:
    void initialize_all(float pl, float gw, float in_mpg,
        float pay)
```

```

    {
        payload = pl;
        moving_van::weight = gw;
        mpg = in_mpg;
        hourly_pay = pay;
    };
    float cost_per_full_day(float cost_of_gas)
    {
        return(8.0 * hourly_pay + 8.0 * cost_of_gas * 55.0 / mpg);
    };
    float total_weight(void)
    {
        return(moving_van::weight + driver::weight);
    };
};
void main()
{
    driven_truck chuck_ford;
    chuck_ford.initialize_all(20000.0, 12000.0, 5.2, 12.50);
    chuck_ford.driver::initialize(15.50, 250.0);
    cout << "The efficiency of the Ford is "
        << chuck_ford. efficiency() << "\n";
    cout << "The cost per mile for Chuck to drive is "
        << chuck_ford. cost_per_mile() << "\n";
    cout << "The cost of Chuck driving the Ford for a day is "
        << chuck_ford. cost_per_full_day(1.129) << "\n";
    cout << "The total weight is " << chuck_ford. total_weight()
        << "\n";
}

```

According to the rules of inheritance, an object of the `driven_truck` class will have two variables with the same name, `weight`. This would be a problem if it weren't for the fact that C++ has defined a method of accessing each one in a well defined way; we will use qualification. The last half dozen lines illustrate the use of the variables. It may be obvious, but it should be explicitly stated, that there is no reason that the subclass itself cannot have a variable of the same name as those inherited from the parent classes. In order to access it, no qualification would be required.

It should be apparent to you that once you understand single inheritance, multiple inheritance is nothing more than an extension of the same rules. Of course, if you inherit two methods or variables of the same name, you must use qualification to allow the compiler to select the correct one.

Practical multiple inheritance

The next example program [DATETIME.H] is a practical example of multiple inheritance, using the date and time classes from Module 818.

```

#ifndef DATETIME_H
#define DATETIME_H
#include "newdate.h"
#include "time.h"
class datetime : public new_date, public time_of_day
{
public:
    datetime(void) { ; }; // Default to todays date and time
    datetime(int M, int D, int Y, int H, int Mn, int S) :
        new_date(), // Member initializer
        _date(), // Member initializer

```

```
        time_of_day(H, Mn, S)           // Member initializer
    {
        set_date(M, D, Y);             // Constructor body
    };
};
#endif
```

There is a good deal to be learned from this very short header file since it is our first example of *member initialisation*. There are two constructors for this class, the first being a very simple constructor that does nothing in itself as is evident from an examination of line 8. This constructor allows the constructors to be executed for the classes `new_date` and `time_of_day`. In both cases a constructor will be executed that requires no parameters, and such a constructor is available for each of these two classes.

The second constructor is more interesting since it does not simply use the default constructor, but instead passes some of the input parameters to the inherited class constructors. Following the colon in line 7 are two member initialisers which are used to initialise members of this class. Since the two parent classes are inherited, they are also members of this class and can be initialised as shown. Each of the member initialisers is actually a call to a constructor of the parent classes and it should be evident that there must be a constructor with the proper number of input parameters to respond to the messages given. You will note that in line 8 we are actually calling the constructor with no parameters given explicitly. If we chose, we could simply let the system call that constructor automatically, but this gives us an explicit comment on what is happening.

Member initialisers

Actually, we can use the member initialiser to initialise class members also. If we had a class member of type `int` named `member_var`, we could initialise it also by mentioning the name of the member followed by the value we desired to initialise it to in parentheses. If we wished to initialise it to the value 14 we could use the following line of code in the member initialiser list:

```
    member_var(13);
```

After all member initialisation the normal constructor code is executed, which in this case is given in line 14.

Order of member initialisation

The order of member initialisation may seem a bit strange, but it does follow a few simple rules. The order of member initialisation does not follow the order given by the initialisation list, but another very strict order over which you have complete control. All inherited classes are initialised first in the order they are listed in the class header. If lines 10 and 12 were reversed, class `new_date` would still be initialised first because it is mentioned first in line 5. It has been mentioned that C++

respects its elders and initialises its parents prior to itself. That should be a useful memory aid in the use of member initialisers.

Next, all local class members are initialised in the order in which they are declared in the class, not the order in which they are declared in the initialisation list. Actually, it would probably be good practice to not use the member initialiser to initialise class members but instead to initialise them in the normal constructor code.

Finally, after the member initialisers are all executed in the proper order, the main body of the constructor is executed in the normal manner.

Using the new class

The next example program [USEDTTM.CPP] uses the datetime class we just built, and like our previous examples, the main program is kept very simple and straight forward. You will note that the default constructor is used for the object named now, and the constructor with the member initialisers is used with the objects named birthday and special.

```
#include <iostream.h>
#include "datetime.h"
datetime now, birthday(10, 18, 1938, 1, 30, 17);
datetime special( 2, 19, 1950, 13, 30, 0);
void main()
{
cout << "Now = " << now.get_date_string()
      << " " << now.get_time_string()
      << " and is day " << now.get_day_of_year() << "\n";
cout << "Birthday = " << birthday.get_date_string() << " "
      << birthday.get_time_string() << " and is day "
      << birthday.get_day_of_year() << "\n";
cout << "Special = " << special.get_date_string() << " "
      << special.get_time_string() << " and is day "
      << special.get_day_of_year() << "\n";
}
```

Exercise 6

Describe how potential name conflicts for variables and methods are handled in classes employing multiple inheritance.

Objective 7 After working through this section you should be able to define virtual functions in C++

Objects are polymorphic (exhibit *polymorphism*) if they have some similarities but are still somewhat different. We have already studied operator overloading and function overloading in this and earlier Modules, and they are a subtle form of polymorphism since in both cases, a single entity is used to refer to two or more things. The use of virtual functions can be a great aid in programming some kinds of projects as you will see in these two sections.

A simple program with inheritance

Examine the first example program [VIRTUAL1.CPP] for the basic program outline we will use for all discussion in this section. Since this program has nothing to do with virtual functions, the name may be somewhat misleading; it is so named because it is part of a series of programs intended to illustrate the use of virtual functions. The last program in this section will illustrate the proper use of virtual functions.

```
#include <iostream.h>
class vehicle
{
int wheels;
float weight;
public:
    void message(void)
    {
        cout << "Vehicle message\n";
    }
};
class car : public vehicle
{
int passenger_load;
public:
    void message(void)
    {
        cout << "Car message\n";
    }
};
class truck : public vehicle
{
int passenger_load;
float payload;
public:
    int passengers(void)
    {
        return passenger_load;
    }
};
class boat : public vehicle
{
int passenger_load;
public:
    int passengers(void)
    {
        return passenger_load;
    }
    void message(void)
    {
```

```
        cout << "Boat message\n";
    }
};
void main()
{
vehicle unicycle;
car sedan;
truck semi;
boat sailboat;
unicycle.message();
sedan.message();
semi.message();
sailboat.message();
}
```

This first program is very simple and you will recognise it as being somewhat similar to the programs studied in the last section except that this program is greatly simplified in order to focus on the virtual function. You will notice that many of the methods from the last section have been dropped from this example for simplicity, and a new method has been added to the parent class, the method named `message()` in line 7.

Throughout this section we will be studying the operation of the method named `message()` in the base class and the derived classes. For that reason, there is a method named `message()` in the car class as well as in the new class named boat in lines 31 to 43. You will also notice that there is a lack of a method named `message()` in the truck class. This has been done on purpose to illustrate the use of the virtual method, or if you prefer, you can refer to it as a virtual function. You will recall that the method named `message()` from the base class is available in the truck class because the method from the base class is inherited with the keyword `public` included in line 15. You will also notice that the use of the keyword `public` in lines 12 and 21 actually do nothing because the only method available in the base class is also available in the derived classes. There are no methods actually inherited. Leaving the keyword in the header poses no problem however, so it will be left there for your study.

The method named `message()` in the base class and in the derived classes has been kept very simple on purpose. Once again, we are interested in the technique of the virtual method rather than a long complicated example.

The main program is as simple as the classes, one object of each of the classes is declared in lines 46 to 49 and the method named `message()` is called once for each object. The result of executing the program indicates that the method for each is called except for the object named `semi`, which has no method named `message()`. As discussed in the last section, the method named `message()` from the parent class is called and the data output to the monitor indicates that this did happen since it displays "Vehicle message" for the object named `semi`.

The data for the objects is of no concern in this section so all data is allowed to default to private type and none is inherited into the derived classes. Some of the data is left in the example program simply to make the classes look like classes. Based on your experience with C++ you will realise that the data could be removed since it is not used.

Adding the keyword VIRTUAL

As you examine the next example program [VIRTUAL2.CPP] you will notice that there is one small change in line 7. The keyword virtual has been added to the declaration of the method named message() in the parent class.

```
#include <iostream.h>
class vehicle
{
    int wheels;
    float weight;
public:
    virtual void message(void)
    {
        cout << "Vehicle message\n";
    }
};
class car : public vehicle
{
    int passenger_load;
public:
    void message(void)
    {
        cout << "Car message\n";
    }
};
class truck : public vehicle
{
    int passenger_load;
    float payload;
public:
    int passengers(void)
    {
        return passenger_load;
    }
};
class boat : public vehicle
{
    int passenger_load;
public:
    int passengers(void)
    {
        return passenger_load;
    }
    void message(void)
    {
        cout << "Boat message\n";
    }
};
void main()
{
    vehicle unicycle;
    car sedan;
    truck semi;
    boat sailboat;
    unicycle.message();
    sedan.message();
    semi.message();
}
```

```
sailboat.message();  
// unicycle = sedan;  
}
```

It may be a bit disappointing to learn that this program operates just like the last example program. This is because we are using objects directly and virtual methods have nothing to do with objects, only with pointers to objects. There is an additional comment in the last line of the main program indicating that, since all four objects are of different classes, it is impossible to assign any object to any other object in this program. We will soon see that some pointer assignments are permitted between objects.

Using object pointers

The next example program [VIRTUAL3.CPP] is a repeat of the first program but with a different main program.

```
#include <iostream.h>  
class vehicle  
{  
    int wheels;  
    float weight;  
public:  
    void message(void)  
    {  
        cout << "Vehicle message\n";  
    }  
};  
class car : public vehicle  
{  
    int passenger_load;  
public:  
    void message(void)  
    {  
        cout << "Car message\n";  
    }  
};  
class truck : public vehicle  
{  
    int passenger_load;  
    float payload;  
public:  
    int passengers(void)  
    {  
        return passenger_load;  
    }  
};  
class boat : public vehicle  
{  
    int passenger_load;  
public:  
    int passengers(void)  
    {  
        return passenger_load;  
    }  
    void message(void)  
    {  
        cout << "Boat message\n";  
    }  
};  
void main()  
{  
    vehicle *unicycle;
```

```
car      *sedan;
truck   *semi;
boat    *sailboat;
unicycle = new vehicle;
unicycle->message();
sedan = new car;
sedan->message();
semi = new truck;
semi->message();
sailboat = new boat;
sailboat->message();
}
```

In this program the keyword `virtual` has been removed from the method declaration in the parent class in line 7 and the main program declares pointers to the objects rather than declaring the objects themselves in lines 46 to 49. Since we only declared pointers to the objects we find it necessary to allocate the objects before using them by using the `new` operator in lines 50 to 57. Running the program we find that even though we are using pointers to the objects we have done nothing different than what we did in the first program. Upon execution, we find that the program operates in exactly the same manner as the first example program in this section. This should not be surprising because a pointer to a method can be used to operate on an object in the same manner as an object itself can be manipulated.

You should have noted that we failed to deallocate the objects prior to terminating the program. As always, in such a simple program, it does not matter greatly because the heap will be cleaned up automatically when we return to the operating system. Still, deallocation is really the programmers responsibility.

A pointer and a virtual function

The next example program [VIRTUAL4.CPP] is identical to the last program except for the addition of the keyword `virtual` to line 7 once again.

```
#include <iostream.h>
class vehicle
{
    int wheels;
    float weight;
public:
    virtual void message(void)
    {
        cout << "Vehicle message\n";
    }
};
class car : public vehicle
{
    int passenger_load;
public:
    void message(void)
    {
        cout << "Car message\n";
    }
};
class truck : public vehicle
{
```

```
        int passenger_load;
        float payload;
public:
    int passengers(void)
    {
        return passenger_load;
    }
};
class boat : public vehicle
{
    int passenger_load;
public:
    int passengers(void)
    {
        return passenger_load;
    }
    void message(void)
    {
        cout << "Boat message\n";
    }
};
void main()
{
    vehicle *unicycle;
    car *sedan;
    truck *semi;
    boat *sailboat;
    unicycle = new vehicle;
    unicycle->message();
    sedan = new car;
    sedan->message();
    semi = new truck;
    semi->message();
    sailboat = new boat;
    sailboat->message();
}
```

This program, including the keyword `virtual`, is still identical to the last program. Once again we are simply using pointers to each of the objects, and in every case the pointer is of the same type as the object to which it points. You will begin to see some changes in the next example program. The last four programs were meant to show you what virtual functions do *not do*. The next two will show you what virtual functions actually *do*.

A single pointer to the parent class

In the next program [VIRTUAL5.CPP] we *almost* use a virtual method.

```
#include <iostream.h>
class vehicle
{
    int wheels;
    float weight;
public:
    void message(void)
    {
        cout << "Vehicle message\n";
    }
};
class car : public vehicle
{
    int passenger_load;
public:
```

```

        void message(void)
        {
            cout << "Car message\n";
        }
    };
    class truck : public vehicle
    {
        int passenger_load;
        float payload;
    public:
        int passengers(void)
        {
            return passenger_load;
        }
    };
    class boat : public vehicle
    {
        int passenger_load;
    public:
        int passengers(void)
        {
            return passenger_load;
        }
        void message(void)
        {
            cout << "Boat message\n";
        }
    };
    void main()
    {
        vehicle *unicycle;
        unicycle = new vehicle;
        unicycle->message();
        delete unicycle;
        unicycle = new car;
        unicycle->message();
        delete unicycle;
        unicycle = new truck;
        unicycle->message();
        delete unicycle;
        unicycle = new boat;
        unicycle->message();
        delete unicycle;
    }

```

You will notice that this is another copy of our program with the keyword `virtual` omitted from line 7 and with a totally different main program. In this program, we only declare a single pointer to a class and the pointer is pointing to the base class of the class hierarchy. We will use the single pointer to refer to each of the four classes and observe what output the method named `message()` produces.

A little digression is in order to understand how we can use a pointer which has been declared to point to one class, to actually refer to another class. If we referred to a `vehicle` (in the real world, not necessarily in this program), we could be referring to a car, a truck, a motorcycle, or any other kinds of transportation, because we are referring to a very general form of an object. If however, we were to refer to a car, we are excluding trucks, motorcycles, and all other kinds of transportation, because we are referring to a car specifically. The more general term of `vehicle` can therefore refer to many kinds of vehicles, but the more specific term of `car` can only refer to a single kind of vehicle, namely a car.

We can apply the same thought process in C++ and say that if we have a pointer to a vehicle (remembering that a pointer is actually a reference), we can use that pointer to refer to any of the more specific objects, and that is indeed legal in C++ according to the definition of the language. In a like manner, if we have a pointer to a car, we cannot use that pointer to reference any of the other classes including the vehicle class because the pointer to the car class is too specific and restricted to be used on any of the other classes.

The C++ pointer rule

A pointer declared as pointing to a base class can be used to point to an object of a derived class of that base class, but a pointer to a derived class cannot be used to point to an object of the base class or to any of the other derived classes of the base class.

In our program, therefore, we are allowed to declare a pointer to the vehicle class which is the base class, and use that pointer to refer to objects of either the base class or any of the derived classes. This is exactly what we do in the main program. We declare a single pointer which points to the vehicle class and use it to point to objects of each of the classes in the same order as in the last four programs. In each case, we allocate the object, send a message to the method named `message()` and deallocate the object before going on to the next class. You will notice that when we send the four messages, we are sending the message to the same method, namely the method named `message()` which is a part of the vehicle base class. This is because the pointer has a class associated with it. Even though the pointer is actually pointing to four different classes in this program, the program acts as if the pointer is always pointing to an object of the parent class because the pointer is of the type of the parent class.

An Actual Virtual Function

We finally come to an example program [VIRTUAL6.CPP] with a *virtual* function that operates as a virtual function and exhibits dynamic binding, or *polymorphism*.

```
#include <iostream.h>
class vehicle
{
    int wheels;
    float weight;
public:
    virtual void message(void)
    {
        cout << "Vehicle message\n";
    }
};
class car : public vehicle
{
    int passenger_load;
```

```
public:
    void message(void)
    {
        cout << "Car message\n";
    }
};
class truck : public vehicle
{
    int passenger_load;
    float payload;
public:
    int passengers(void)
    {
        return passenger_load;
    }
};
class boat : public vehicle
{
    int passenger_load;
public:
    int passengers(void)
    {
        return passenger_load;
    }
    void message(void)
    {
        cout << "Boat message\n";
    }
};
void main()
{
    vehicle *unicycle;
    unicycle = new vehicle;
    unicycle->message();
    delete unicycle;
    unicycle = new car;
    unicycle->message();
    delete unicycle;
    unicycle = new truck;
    unicycle->message();
    delete unicycle;
    unicycle = new boat;
    unicycle->message();
    delete unicycle;
}
```

This program is identical to the last example program except that the keyword `virtual` is added to line 7 to make the method named `message()` a virtual function. You will notice that the keyword `virtual` only appears in the base class, all classes that derive this class will have the corresponding method automatically declared `virtual` by the system. In this program, we will once again use the single pointer to the base class and allocate, use, then delete an object of each of the four available classes using the identical code we used in the last program. However, because of the addition of the keyword `virtual` in line 7, this program acts entirely different from the last example program.

Since the method named `message()` is declared to be a virtual method in its declaration in the base class, anytime we refer to this method with a pointer to the base class, we actually execute the method associated with one of the derived classes if there is a method available in the derived class and if the pointer is actually pointing to that derived class. When the program is executed, the output reflects the same output we

saw in the other cases when we were actually calling the methods in the derived classes, but now we are using a pointer of the base class type to make the calls.

You will notice that in lines 48, 51, 54, and 57, even though the code is identical in each line, the system is making the decision of which method to actually call based on the type of the pointer when each message is sent. The decision of which method to call is not made during the time when the code is compiled but when the code is executed. This is dynamic binding and can be very useful in some programming situations. In fact, there are only three different calls made because the class named truck does not have a method named `message()`, so the system simply uses the method from the base class to satisfy the message passed. For this reason, a virtual function must have an implementation available in the base class which will be used if there is not one available in one or more of the derived classes. Note that the message is actually sent to a pointer to the object, but this is splitting hairs and should not be overly emphasised at this time.

It is not obvious, but the structure of the virtual function in the base class and each of the derived classes is identical. The return type and the number and types of the parameters must be identical for all since a single statement can be used to call any of them.

This program probably does not seem to do much when you first approach it, but dynamic binding is a very useful construct. Its use is illustrated in the next section with a simple program that uses the technique of dynamic binding to implement a personnel list for a small company.

If the keyword `virtual` is used, the system will use late binding which is done at run time, but if the keyword is not included, early binding will be used. What these words actually mean is that with late binding, the compiler does not know which method will actually respond to the message because the type of the pointer is not known at compile time. With early binding, however, the compiler decides at compile time what method will respond to the message sent to the pointer.

Exercise 7

Modify `VIRTUAL3.CPP` to deallocate the objects before the program terminates.

Add a `message()` method to the truck class in `VIRTUAL6.CPP`. Describe how this differs from defaulting to the equivalent method in the parent class.

Objective 8 After working through this section you should be able define and carry out the steps involved in developing a full-scale OOP project in C++

This section will actually be a continuation of the topics covered in the last section, but this will be a fuller explanation of what virtual functions are and how they can be used. We will do this by presenting a simple database program with a virtual function to show how it can be used, then we will go on to illustrate a more complex use of the virtual function in a manner that finally illustrates its utility and the reason for its existence.

Starting an OOP project

You may have noticed that we always seem to begin our an object-oriented program by identifying an object (in this case a class of objects and even some subordinate objects) which we completely define. When we get to the main program we then have the relatively simple job of carrying out the remaining tasks, which are completed using standard procedural programming techniques. This is the way to begin any object oriented programming project, by first identifying a few objects that can be separated conveniently from the rest of the code, programming them, then writing the main program. It should be added that, for your first project using objects, you should not try to make everything an object. Select a few objects and after gaining experience with object oriented programming techniques, use more objects on future projects. Most programmers use too many objects for their first project and write very obtuse, unreadable code.

The person header file

Examine the following header file [PERSON.H] for the definition file for the person class. This class definition should cause you no problems since there is nothing new here.

```
#ifndef PERSON_H
#define PERSON_H
class person {
protected:          // Make these variables available to the
                    subclasses
    char name[25];
    int salary;
public:
    virtual void display(void);
};
#endif
```

The only thing that should be mentioned about this class is that the protected mode is used for the variables so that they are readily available in the derived classes which will inherit this class.

The person implementation

The implementation for the person class is given here [PERSON.CPP] and it is a little strange in the way it is written and used. The intent of this program is that the virtual method named display() in this file will never be used, but it is required by the C++ compiler to be used for a default in case some of the subclasses do not have this function available. In the main program we will be careful to never call this function due to the nature of the program we are writing. Keep in mind that C++ requires an implementation of all virtual functions even if they are never used. In this case the message is obviously intended to be output as an error message.

```
#include <iostream.h>
#include "person.h"
        // This method should never be called.  If it is ever
        // called, it is considered an error.
void
person::display(void)
{
cout << "person::display - missing subclass method\n";
}
```

The supervisor header

The next header file [SUPERVSR.H] contains the class definitions for the three derived classes: supervisor, programmer, and secretary. These were all placed in a single file for two reasons. The first reason is to simply illustrate to you that this can be done, and secondly, to allow some of the files to be combined on the disk and to require fewer compilations by you prior to executing the resulting program. This is actually a good way to combine these files since they are all derived classes of a common class. It is a matter of style or personal taste.

```
#ifndef SUPERVSR_H
#define SUPERVSR_H
// This defines three subclasses of the parent type person.
// Different data are stored for the different job
// classifications to illustrate that it can be done.
#include "person.h"
class supervisor : public person
{
char title[25];
public:
void init_data(char in_name[], int in_salary,
char in_title[]);
void display(void);
};
class programmer : public person
{
char title[25];
char language[25];
public:
void init_data(char in_name[], int in_salary,
char in_title[],
char in_language[]);
void display(void);
};
class secretary : public person
{
```

```

    char shorthand;
    int typing_speed;
public:
    void init_data(char in_name[], int in_salary,
                  char in_shorthand, char in_typing_speed);
    void display(void);
};
#endif

```

You will notice that all three of these classes contain a method named `display()` and all have the same return value of `void`, and all have the same number of parameters as the parent class's method of the same name. All of this equality is required because they will all be called by the same call statement. You will also notice that the other method in each class has the same name, but different numbers and types of formal parameters which prevents this method from being used as a virtual method.

The supervisor implementation

The next program [SUPERVSR.CPP] contains the implementation for the three classes. If you spend a little time studying the code, you will find that each of the methods named `init_data()` simply initialises all fields to those passed in as the actual arguments in a very simple manner. The method named `display()`, however, outputs the stored data in different ways for each class since the data is so different in each of the classes. Even though the interface to these three methods is identical, the actual code is significantly different. There is no reason other code besides output could not have been used, but the output is so visible when the program is executed that it was chosen for this illustration.

```

#include "supervsr.h"
#include <iostream.h>
#include <stdio.h>
#include <string.h>
// In all cases, init_data assigns values to the class variables
// and display outputs the values to the monitor for inspection.
void
supervisor::init_data(char in_name[], int in_salary,
                      char in_title[])
{
    strcpy(name, in_name);
    salary = in_salary;
    strcpy(title, in_title);
}
void
supervisor::display(void)
{
    cout << "Supervisor --> " << name << "'s salary is " << salary
         << " and is the " << title << ".\n\n";
}
void programmer::init_data(char in_name[], int in_salary,
                           char in_title[], char in_language[])
{
    strcpy(name, in_name);
    salary = in_salary;
    strcpy(title, in_title);
    strcpy(language, in_language);
}
void programmer::display(void)

```

```
{
cout << "Programmer --> " << name << "'s salary is " << salary
    << " and is " << title << ".\n";
cout << "      " << name << "'s specialty is "
    << language << ".\n\n";
}
void secretary::init_data(char in_name[], int in_salary,
                        char in_shorthand, char in_typing_speed)
{
strcpy(name,in_name);
salary = in_salary;
shorthand = in_shorthand;
typing_speed = in_typing_speed;
}
void secretary::display(void)
{
cout << "Secretary ---> " << name << "'s salary is "
    << salary << ".\n";
cout << "      " << name << " types " << typing_speed
    << " per minute and can ";
if (!shorthand) cout << "not ";
cout << "take shorthand.\n\n";
}
```

This file should be compiled now in preparation for the next example program, which will use all the four classes defined in these four files.

The first calling program

The next program [EMPLOYEE.CPP] is the first that uses the classes developed in this section, and you will find that it is a very simple program.

```
#include <iostream.h>
#include "person.h"
#include "supervsr.h"
person *staff[10];
void main()
{
supervisor *suppt;
programmer *progpt;
secretary *secpt;
cout << "XYZ Staff -- note salary is monthly.\n\n";
suppt = new supervisor;
suppt->init_data("Big John", 5100, "President");
staff[0] = suppt;
progpt = new programmer;
progpt->init_data("Joe Hacker", 3500, "debugger", "Pascal");
staff[1] = progpt;
progpt = new programmer;
progpt->init_data("OOP Wizard", 7700, "senior analyst", "C++");
staff[2] = progpt;
secpt = new secretary;
secpt->init_data("Tillie Typer", 2200, 1, 85);
staff[3] = secpt;
suppt = new supervisor;
suppt->init_data("Tom talker", 5430, "Sales manager");
staff[4] = suppt;
progpt = new programmer;
progpt->init_data("Dave Debugger", 5725, "code maintainer",
                "assembly language");
staff[5] = progpt;
for (int index = 0 ; index < 6 ; index++ )
    staff[index]->display();
cout << "End of employee list.\n";
}
```

We begin with an array of ten pointers, each pointing to the base class. As you recall from the last section, this is very important when using virtual functions: the pointer *must* point to the base class. The pointers that will be stored in this array will all point to objects of the derived classes, however. When we use the resulting pointers to refer to the methods, the system will choose the method at run time, not at compile time, as nearly all of our other programs have been doing.

We allocate six objects in lines 11 to 29, initialise them to some values using the methods named `init_data()`, then assign the pointers to the members of the array of pointers to `person`. Finally, in line 31, we call the methods named `display()` to display the stored data on the monitor. You will notice that even though we only use one method call in line 31, we actually send messages to each of the three methods named `display()` in the subclasses. This is true dynamic binding because if we were to change the values of some of the pointers in the array, we would then call different methods with the same pointers.

A pure virtual function

The *pure* virtual function is also available. You can use a pure virtual function in the present example program by changing line 9 of `PERSON.H` to read as follows:

```
virtual void display(void) = 0;
```

You must then eliminate `PERSON.CPP` from the project or make sequence. An implementation for a pure virtual function cannot exist in the base class.

Every derived class must include a function for each pure virtual function which is inherited into the derived class. This assures that there will be a function available for each call and none will ever need to be answered by the base class. You are not permitted to create an object of any class which contains one or more pure virtual functions because there is nothing to answer a message if one is sent to the pure virtual method. The compiler will enforce the two rules mentioned in this paragraph.

The linked list class

Examination of the next program [`ELEMLIST.H`] will reveal the definition of two more classes which will be used to build a linked list of employees to illustrate a more practical way to use the dynamic binding we have been studying in this section.

```
#ifndef ELEMLIST_H
#define ELEMLIST_H
#define NULL 0
#include "person.h"
class employee_list;           // Forward declaration
class employee_element
```

```
{
    // One element of the linked list
    person *employee_data;
    employee_element *next_employee;
public:
    employee_element(person *new_employee)
    {
        next_employee = NULL;
        employee_data = new_employee;
    };
    friend class employee_list;
};
class employee_list
{
    // The linked list
    employee_element *start;
    employee_element *end_of_list;
public:
    employee_list()
    {
        start = NULL;
    }
    void add_person(person *new_employee);
    void display_list(void);
};
#endif
```

The two classes were put in the same file because they work together so closely and neither is of much value without the other. You will notice that the elements of the linked list do not contain any data, only a pointer to the person class that we developed for the last program, so that the linked list will be composed of elements of the person class without modifying the class itself.

There are two interesting constructs used here that must be pointed out before going on to the next program. The first is the partial declaration given in line 5 which allows us to refer to the class named `employee_list` before we actually define it. The complete declaration for the class is given in lines 18 to 29. The second construct of interest is the friend class listed in line 16 where we give the entire class named `employee_list` free access to the variables which are a part of the `employee_element` class. This is necessary because the method named `add_person()` must access the pointers contained in `employee_element`. We could have defined an additional method as a part of `employee_element` and used this method to refer to the pointers but it was felt that these two classes work so well together that it is not a problem to open a window between the classes. We still have complete privacy from all other programs and classes declared as parts of this program.

Note that the single method included in the `employee_element` class is implemented in inline code. Two of the methods of `employee_list` are still open so we need an implementation for this class.

The linked list implementation

The implementation for the linked list classes [ELEMMLIST.CPP] should be self explanatory if you understand how a singly linked list operates.

All new elements are added to the end of the current list. This was done to keep it simple but a sorting mechanism could be added to sort the employees by name if desired. The method to display the list simply traverses the list and calls the method named `display()` in line 20 once for each element of the list.

```
#include "elemlist.h"
void
employee_list::add_person(person *new_employee)
{
    employee_element *temp;

    temp = new employee_element(new_employee);
    if (start == NULL)
        start = end_of_list = temp;
    else {
        end_of_list->next_employee = temp;
        end_of_list = temp;
    }
}
void
employee_list::display_list(void)
{
    employee_element *temp;

    temp = start;
    do {
        temp->employee_data->display();
        temp = temp->next_employee;
    } while (temp != NULL);
}
```

In this entire class there is no mention made of the existence of the three derived classes; only the base class named `person` is mentioned. The linked list therefore has no hint that the three subclasses even exist, but in spite of that, we will see this class send messages to the three subclasses as they are passed through this logic. That is exactly what dynamic binding is, and we will have a little more to say about it after we examine the calling program.

A linked list application

At this time you should examine the final example program in this section [EMPLOYEE2.CPP]; it is our best example of dynamic binding, even if the program is still very simple.

```
#include <iostream.h>
#include "person.h"
#include "supervsr.h"
#include "elemlist.h"
employee_list list;
void main()
{
    supervisor *suppt;
    programmer *progpt;
    secretary *secp;
    cout << "XYZ Staff -- note salary is monthly.\n\n";
    suppt = new supervisor;
    suppt->init_data("Big John", 5100, "President");
    list.add_person(suppt);
    progpt = new programmer;
    progpt->init_data("Joe Hacker", 3500, "debugger", "Pascal");
    list.add_person(progpt);
}
```

```
progpt = new programmer;
progpt->init_data("OOP Wizard", 7700, "senior analyst", "C++");
list.add_person(progpt);
secpt = new secretary;
secpt->init_data("Tillie Typer", 2200, 1, 85);
list.add_person(secpt);
suppt = new supervisor;
suppt->init_data("Tom talker", 5430, "Sales manager");
list.add_person(suppt);
progpt = new programmer;
progpt->init_data("Dave Debugger", 5725, "code maintainer",
                "assembly language");
list.add_person(progpt);
                // Now display the entire list
list.display_list();
cout << "End of employee list.\n";
}
```

This program is very similar to the example program EMPLOYEE.CPP with a few changes to emphasise dynamic binding. In line 5 we declare an object of the class employee_list to begin our linked list. This is the only copy of the list we will need for this program. For each of the elements, we allocate the data, fill it, and send it to the linked list to be added to the list where we allocate another linked list element to point to the new data, and add it to the list. The code is very similar to the last program down through line 28.

In line 34 we send a message to the display_list() method which outputs the entire list of personnel. You will notice that the linked list class defined in the files named ELEMLIST.H and ELEMLIST.CPP are never informed in any way that the subclasses even exist but they dutifully pass the pointers to these subclasses to the correct methods and the program runs as expected. If you changed PERSON.H to use a pure virtual function, it will still work with this program just as we discussed earlier.

Now that we have the program completely debugged and working, suppose that we wished to add another class to the program, for example a class named consultant because we wished to include some consultants in our business. We would have to write the class of course and the methods within the classes, but the linked list doesn't need to know that the new class is added, so it does not require any changes in order to update the program to handle consultant class objects. In this particular case, the linked list is very small and easy to understand, but suppose the code was very long and complex as with a large database. It would be very difficult to update every reference to the subclasses and add another subclass to every list where they were referred to, and this operation would be error prone. In the present example program, the linked list would not even have to be recompiled in order to add the new functionality.

It should be clear that it would be possible to actually define new types, dynamically allocate them, and begin using them even while the program was executing if we properly partitioned the code into

executable units operating in parallel. This would not be easy, but it could be done for a large database that was tracking the inventory for a large retail store, or even for an airlines reservation system. You probably have little difficulty understanding the use of dynamically allocated memory for data, but dynamically allocating classes or types is new and difficult to grasp, but the possibility is there with dynamic binding.

Exercise 8

Add a new class named *consultant* to SUPERVSR.H and SUPERVSR.CPP, then modify to EMPLOYEE2.CPP to use the new class.

Objective 9 After working through this section you should be able to list some of the likely future developments in C++

A committee to write an ANSI standard for C++ was formed and first met in the Spring of 1990. They have recently completed the first version of the standard. Many small changes have been added during the past three years that barely affect the casual programmer, or even the heavy user of the language. You can be sure that the language will evolve slowly and surely into a very useable and reliable language. There are two areas, however, that should be discussed in a little detail because they will add so much to the language in future years. Those two topics are *parameterised types* and *exception handling*.

Parameterised types

Many times, when developing a program, you wish to perform some operation on more than one data type. For example you may wish to sort a list of integers, another list of floating point numbers, and a list of alphabetic strings. It seems silly to have to write a separate sort function for each of the three types when all three are sorted in the same logical way. With parameterised types (also called *templates*), you will be able to write a single sort routine that is capable of sorting all three of the lists.

This is already available in the Ada language as the generic package or procedure. Because it is available in Ada, there is a software components industry that provides programmers with pre-written and thoroughly debugged software routines that work with many different types. When this is generally available in C++, there will be a components industry for C++ and precoded, debugged and efficient source code will be available off the shelf to perform many of the standard operations. These operations will include such things as sorts, queues, stacks, lists, etc.

Borland International has included templates in version 3.0 of Borland C++. The next three example programs illustrate the use of templates with Borland's compiler, but may not work with other compilers.

```
#include <stdio.h>
template <class ANY_TYPE>
ANY_TYPE maximum(ANY_TYPE a, ANY_TYPE b)
{
    return (a > b) ? a : b;
}
void main()
{
    int x = 12, y = -7;
    float real = 3.1415;
    char ch = 'A';
    printf("%8d\n", maximum(x, y));
    printf("%8d\n", maximum(-34, y));
}
```

```

printf("%8.3f\n", maximum(real, float(y)));
printf("%8.3f\n", maximum(real, float(x)));
printf("%c\n", maximum(ch, 'X'));
}

```

The above program [TEMPLAT1.CPP] is the first example of the use of a template. This program is so simple it seems silly to even bother with it but it will illustrate the use of the parameterised type. The template is given in lines 2 to 5 with the first line indicating that it is a template with a single type to be replaced, the type ANY_TYPE. This type can be replaced by any type which can be used in the comparison operation in line 5. If you have defined a class, and you have overloaded the operator ">", then this template can be used with objects of your class. Thus, you do not have to write a maximum function for each type or class in your program. This function is included automatically for each type it is called with in the program, and the code itself should be very easy to understand.

You may have realised that nearly the same effect can be achieved through use of a macro, except that when a macro is used, the strict type checking is not done. Because of this, and because of the availability of the inline method capability in C++, the macros that were covered in Module 813 are almost never used by experienced C++ programmers.

A class template

The next example program [TEMPLAT2.CPP] is a little more involved since it provides a template for an entire class rather than a single function. The template code is given in lines 3 to 12 and a little study will show that this is an entire class definition. This is a very 'weak' stack class since there is nothing to prevent popping data from an empty stack, and there is no indication of a full stack. Our aim, however, is to illustrate the use of the parameterised type and to do so using the simplest possible class.

```

#include <stdio.h>
const int MAXSIZE= 128;
template<class ANY_TYPE>
class stack
{
    ANY_TYPE array[MAXSIZE];
    int stack_pointer;
public:
    stack(void) { stack_pointer = 0; };
    void push(ANY_TYPE in_dat){ array[stack_pointer++] = in_dat;
};
    ANY_TYPE pop(void          { return array[--stack_pointer]; };
    int empty(void)           { return (stack_pointer == 0); };
}
char name[] = "John Herkimer Doe";
void main()
{
    int x = 12, y = -7;
    float real = 3.1415;
    stack<int> int_stack;
    stack<float> float_stack;
    stack<char *> string_stack;
}

```

```
int_stack.push(x);
int_stack.push(y);
int_stack.push(77);
float_stack.push(real);
float_stack.push(-12.345);
float_stack.push(100.01);
string_stack.push("This is line 1");
string_stack.push("This is the second line");
string_stack.push("This is the third line");
string_stack.push(name);
printf("Integer stack ---> ");
printf("%8d ", int_stack.pop());
printf("%8d ", int_stack.pop());
printf("%8d\n", int_stack.pop());
printf(" Float stack ---> ");
printf("%8.3f ", float_stack.pop());
printf("%8.3f ", float_stack.pop());
printf("%8.3f\n", float_stack.pop());
printf("\n Strings\n");
do {
    printf("%s\n", string_stack.pop());
} while (!string_stack.empty());
}
```

In the main program we create an object named `int_stack` in line 20 which will be a stack designed to store integers, and another object named `float_stack` in line 21 which is designed to store float type values. In both cases, we enclose the type we desire this object to work with in "`<>`" brackets, and the system creates the object by first replacing all instances of `ANY_TYPE` with the desired type, then creating the object of that type. You will note that any type can be used that has an assignment capability since lines 10 and 11 use the assignment operator on the parameterised type.

Even though the strings are all of differing lengths, we can even use the stack to store a stack of strings if we only store a pointer to the strings and not the entire string. This is illustrated in the object named `string_stack` declared in line 22 and used later in the program.

Reusing the stack class

The program below [TEMPLAT3.CPP] uses the same class with the template as defined in the last program but in this case, it uses the date class developed earlier as the stack members. More specifically, it uses a pointer to the date class as the stack member.

```
#include <stdio.h>
#include "date.h"
const int MAXSIZE = 128;
template<class ANY_TYPE>
class stack
{
    ANY_TYPE array[MAXSIZE];
    int stack_pointer;
public:
    stack(void) { stack_pointer = 0; };
    void push(ANY_TYPE in_dat){ array[stack_pointer++] = in_dat;
};
    ANY_TYPE pop(void)      { return array[--stack_pointer]; };
    int empty(void)        { return (stack_pointer == 0); };
}
```

```

char name[] = "John Herkimer Doe";
void main()
{
    stack<char *> string_stack;
    stack<date *> class_stack;
    date cow, pig, dog, extra;
    class_stack.push(&cow);
    class_stack.push(&pig);
    class_stack.push(&dog);
    class_stack.push(&extra);
    string_stack.push("This is line 1");
    string_stack.push("This is the second line");
    string_stack.push("This is the third line");
    string_stack.push(name);
    for (int index = 0 ; index < 5 ; index++) {
        extra = *class_stack.pop();
        printf("Date = %d %d %d\n", extra.get_month(),
            extra.get_day(), extra.get_year());
    };
    printf("\n      Strings\n");
    do {
        printf("%s\n", string_stack.pop());
    } while (!string_stack.empty());
}

```

Because class assignment is legal, you could also store the actual class in the stack rather than just the pointer to it. To do so, however, would be very inefficient since the entire class would be copied into the stack each time it is pushed and the entire class would be copied out again when it was popped. Use of the pointer is a little more general, so it was illustrated here for your benefit.

All three of the previous programs can be compiled and executed if you have Borland C++ version 3.0 (or later). Other compilers may not work with these programs since parameterised types are not yet a part of the C++ specification.

Exception handling

A future version of C++ will have some form of exception handling to allow the programmer to trap errors and prevent the system from completely shutting down when a fatal error occurs. The Ada language, for example, allows the programmer to trap any errors that occur (even system errors), execute some recovery code, and continue with program execution in a well-defined way. AT&T and the ANSI-C++ committee have announced that some form of exception handling will be implemented in the next C++ standard, but have not stated what form it will take.

Exercise 9

Define the difference between a *template* and a *class template*.