Chapter 1 - Introduction to Object-oriented Programming

"C++ is designed to allow you to express ideas, but if you don't have ideas or don't have any clue about how to express them, C++ doesn't offer much help."

— Bjarne Stroustrup

Object-oriented Programming (OOP) is the term used to describe a programming approach based on **objects** and **classes**. The object-oriented paradigm allows us to organise software as a collection of objects that consist of both data and behaviour. This is in contrast to conventional functional programming practice that only loosely connects data and behaviour.

Since the 1980s the word 'object' has appeared in relation to programming languages, with almost all languages developed since 1990 having object-oriented features. Some languages have even had object-oriented features retro-fitted. It is widely accepted that object-oriented programming is the most important and powerful way of creating software.

The object-oriented programming approach encourages:

- Modularisation: where the application can be decomposed into modules.
- Software reuse: where an application can be composed from existing and new modules.

An object-oriented programming language generally supports five main features:

- Classes
- Objects
- Classification
- Polymorphism
- Inheritance

An Object-Oriented Class

If we think of a real-world object, such as a television (as in Figure 1.1), it will have several features and properties:

- We do not have to open the case to use it.
- We have some controls to use it (buttons on the box, or a remote control).
- We can still understand the concept of a television, even if it is connected to a DVD player.
- It is complete when we purchase it, with any external requirements well documented.
- The TV will not crash!

In many ways this compares very well to the notion of a class.

Figure 1.1. The concept of a class - television example.

A class should:

- Provide a well-defined interface - such as the remote control of the television.
- Represent a clear concept - such as the concept of a television.
- Be complete and well-documented - the television should have a plug and should have a manual that documents all features.
- The code should be robust - it should not crash, like the television.

With a functional programming language (like C) we would have the component parts of the television scattered everywhere and we would be responsible for making them work correctly - there would be no case surrounding the electronic components.

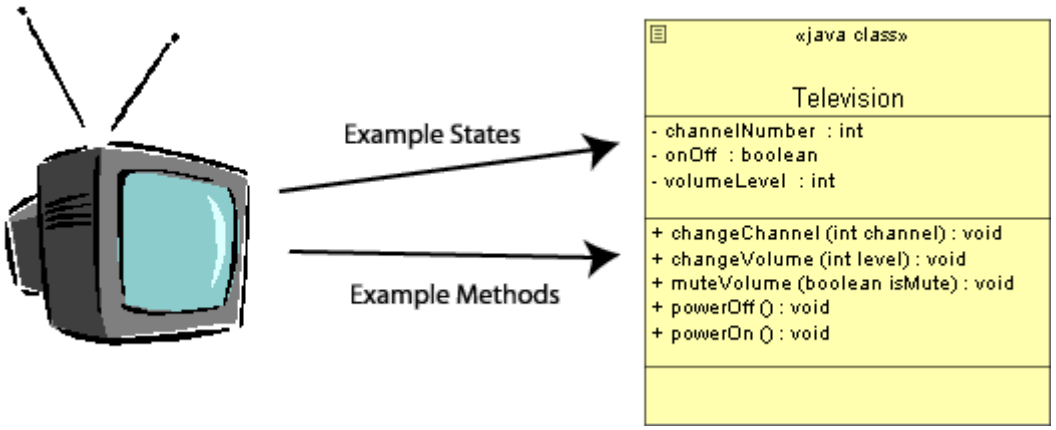
Humans use class based descriptions all the time - what is a duck? (Think about this, we will discuss it soon.)

Classes allow us a way to represent complex structures within a programming language. They have two components:

- **States** - (or data) are the values that the object has.
- **Methods** - (or behaviour) are the ways in which the object can interact with its data, the actions.

The notation used in Figure 1.2 on the right hand side is a Unified Modelling Language (UML) representation of the Television class for object-oriented modelling and programming.

Figure 1.2. The Television class example.



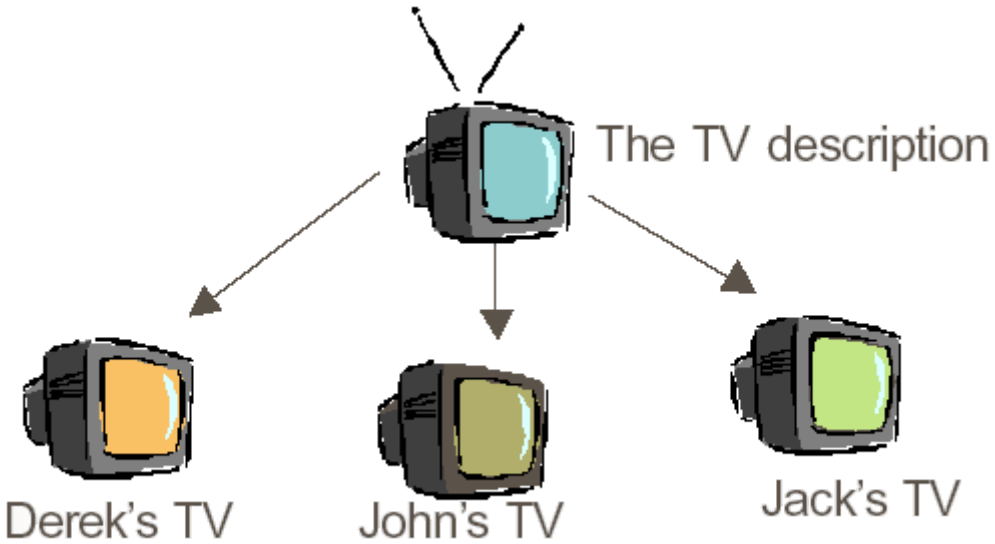
An instance of a class is called an **object**.

An Object

An **object** is an instance of a class. You could think of a class as the description of a concept, and an object as the realisation of this description to create an independent distinguishable entity. For example, in the case of the Television, the class is the set of plans (or blueprints) for a generic television, whereas a television object is the realisation of these plans into a real-world physical television. So there would be one set of plans (the class), but there could be thousands of real-world televisions (objects).

Objects can be concrete (a real-world object, a file on a computer) or could be conceptual (such as a database structure) each with its own individual identity. Figure 1.3 shows an example where the Television class description is 'realised' into several television objects. These objects should have their own identity and are independent from each other. For example, if the channel is changed on one television it will not change on the other televisions.

Figure 1.3. The Television objects example.



Encapsulation

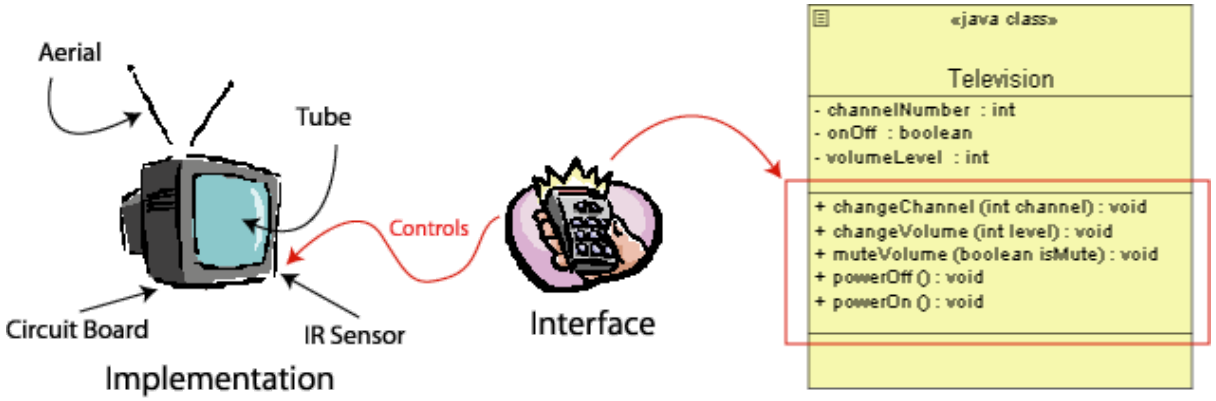
The object-oriented paradigm encourages encapsulation. Encapsulation is used to hide the mechanics of the object, allowing the actual implementation of the object to be hidden, so that we don't need to understand how the object works. All we need to understand is the interface that is provided for us.

You can think of this in the case of the Television class, where the functionality of the television is hidden from us, but we are provided with a remote control, or set of controls for interacting with the television, providing a high level of abstraction. So, as in Figure 1.4 there is no requirement to understand how the signal is decoded from the aerial and converted into a picture to be displayed on the screen before you can use the television.

There is a sub-set of functionality that the user is allowed to call, termed the interface. In the case of the television, this would be the functionality that we could use through the remote control or buttons on the front of the television.

The full implementation of a class is the sum of the public interface plus the private implementation.

Figure 1.4. The Television interface example.



Encapsulation is the term used to describe the way that the interface is separated from the implementation. You can think of encapsulation as "data-hiding", allowing certain parts of an object to be visible, while other parts remain hidden. This has advantages for both the user and the programmer.

For the user (who could be another programmer):

- The user need only understand the interface.
- The user need not understand how the implementation works or was created.

For the programmer:

- The programmer can change the implementation, but need not notify the user.

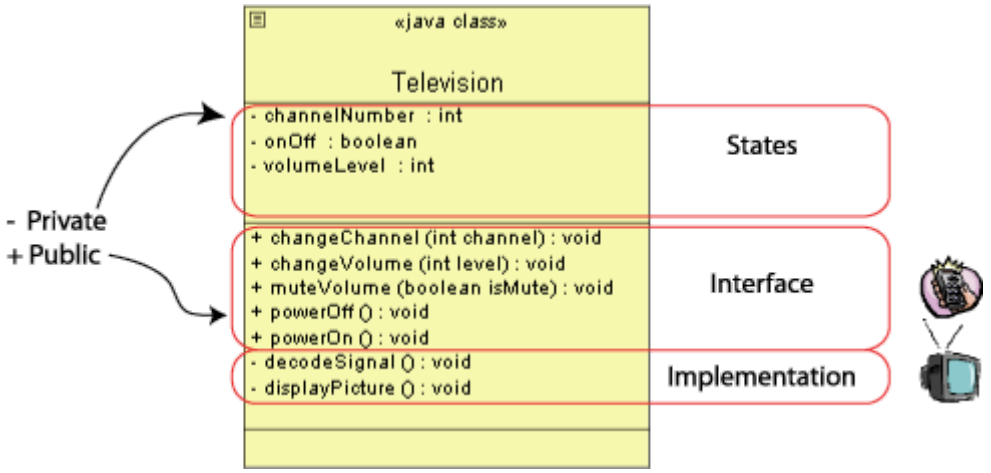
So, providing the programmer does not change the interface in any way, the user will be unaware of any changes, except maybe a minor change in the actual functionality of the application.

We can identify a level of 'hiding' of particular methods or states within a class using the public, private and protected keywords:

- public methods - describe the interface.
- private methods - describe the implementation.

Figure 1.5 shows encapsulation as it relates to the Television class. According to UML notation private methods are denoted with a minus sign and public methods are denoted with a plus sign. The private methods would be methods written that are part of the inner workings of the television, but need not be understood by the user. For example, the user would need to call the powerOn() method but the private displayPicture() method would also be called, but internally as required, not directly by the user. This method is therefore not added to the interface, but hidden internally in the implementation by using the private keyword.

Figure 1.5. The Television class example showing encapsulation.



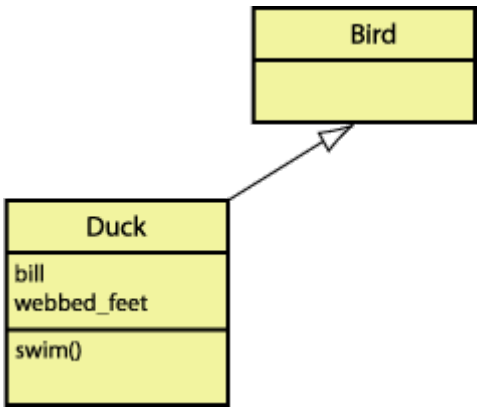
Inheritance

If we have several descriptions with some commonality between these descriptions, we can group the descriptions and their commonality using inheritance to provide a compact representation of these descriptions. The object-oriented programming approach allows us to group the commonalities and create classes that can describe their differences from other classes.

Figure 1.6a. What is a duck[1]?

Humans use this concept in categorising objects and descriptions. For example you may have answered the question - "What is a duck?", with "a bird that swims", or even more accurately, "a bird that swims, with webbed feet, and a bill instead of a beak". So we could say that a Duck is a Bird that swims, so we could describe this as in Figure 1.6. This figure illustrates the inheritance relationship between a Duck and a Bird. In effect we can say that a Duck is a special type of Bird.

Figure 1.6. The Duck class showing inheritance.

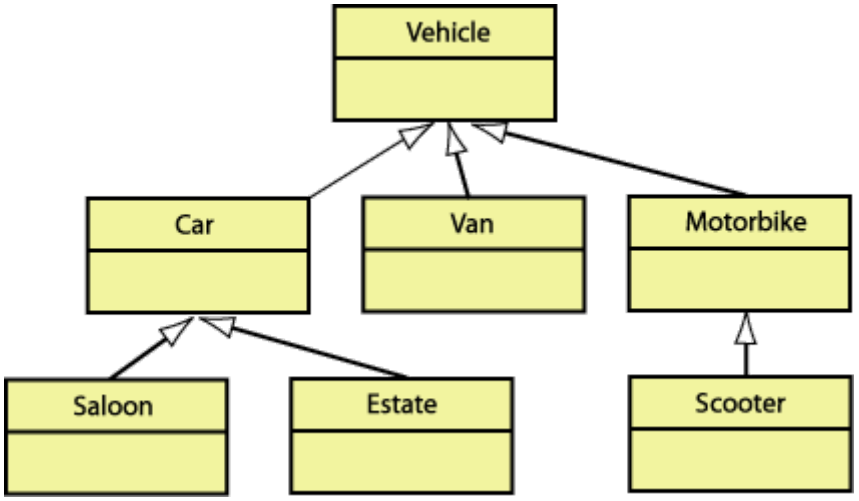


For example: if we were to be given an unstructured group of descriptions such as Car, Saloon, Estate, Van, Vehicle, Motorbike and Scooter, and asked to organise these descriptions by their differences. You might say that a Saloon car is a Car but has a long

[1] Believe it or not, rubber duck debugging is a method of debugging your code where you are encouraged to 'explain it to the duck!' Some software companies even distribute rubber ducks during onboarding for this very purpose. The idea has psychological merit in that it helps you bridge the left/right brain separation. You may have noticed this yourself when explaining a problem to a friend/family member that you identified the problem during your explanation without any input. The rubber duck just acts as a proxy for busy colleagues, and gives cover for talking to yourself!

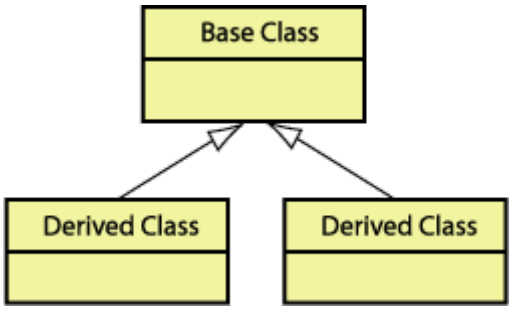
boot, whereas an Estate car is a car with a very large boot. Figure 1.7 shows an example of how we may organise these descriptions using inheritance.

Figure 1.7. The grouped set of classes.



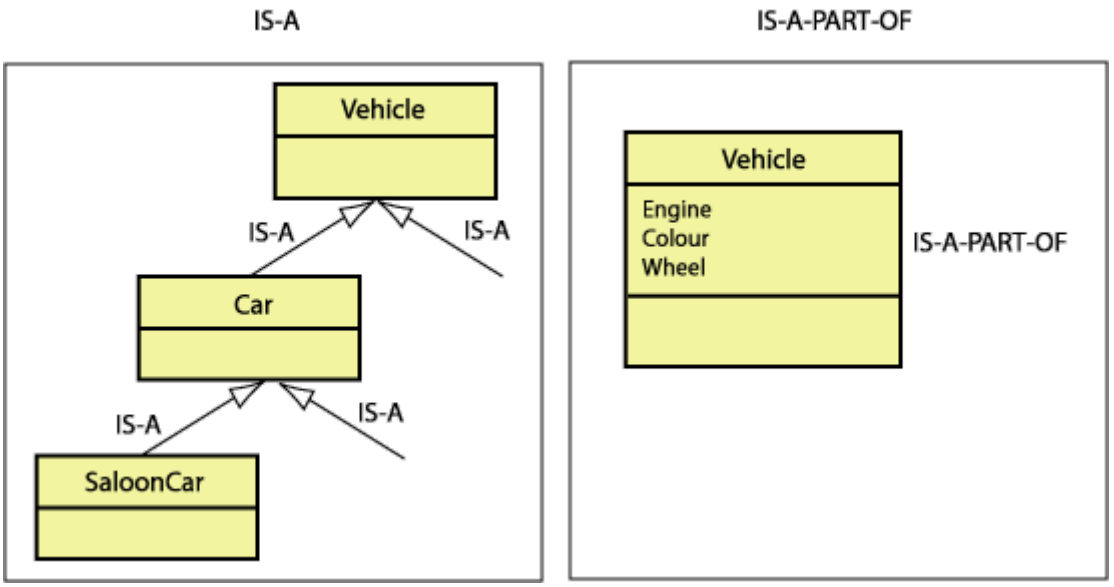
So we can describe this relationship as a child/parent relationship, where Figure 1.8 illustrates the relationship between a base class and a derived class. A derived class inherits from a base class, so in Figure 1.7 the Car class is a child of the Vehicle class, so Car inherits from Vehicle.

Figure 1.8. The Base class and Derived class.



One way to determine that you have organised your classes correctly is to check them using the **"IS-A"** and **"IS-A-PART-OF"** relationship checks. It is easy to confuse objects within a class and children of classes when you first begin programming with an OOP methodology. So, to check the previous relationship between Car and Vehicle, we can see this in Figure 1.9.

Figure 1.9. The IS-A/IS-A-PART-OF relationships and the Vehicle class.



The IS-A relationship describes the inheritance in the figure, where we can say, "A Car IS-A Vehicle" and "A SaloonCar IS-A Car", so all relationships are correct. The IS-A-PART-OF relationship describes the composition (or aggregation) of a class. So in the same figure (Figure 1.9) we can say "An Engine IS-A-PART-OF a Vehicle", or "An Engine, Colour and Wheels IS-A-PART-OF a Vehicle". This is the case even though an Engine is also a class! where there could be many different descriptions of an Engine - petrol, diesel, 1.4, 2.0, 16 valve etc.

So, using inheritance the programmer can:

- Inherit a behaviour and add further specialised behaviour - for example a Car IS A Vehicle with the addition of four Wheel objects, Seats etc.
- Inherit a behaviour and replace it - for example the SaloonCar class will inherit from Car and provide a new "boot" implementation.
- Cut down on the amount of code that needs to be written and debugged - for example in this case only the differences are detailed, a SaloonCar is essentially identical to the Car, with only the differences requiring description.

Polymorphism

When a class inherits from another class it inherits both the states and methods of that class, so in the case of the Car class inheriting from the Vehicle class the Car class inherits the methods of the Vehicle class, such as engineStart(), gearChange(), lightsOn() etc. The Car class will also inherit the states of the Vehicle class, such as isEngineOn, isLightsOn, numberWheels etc.

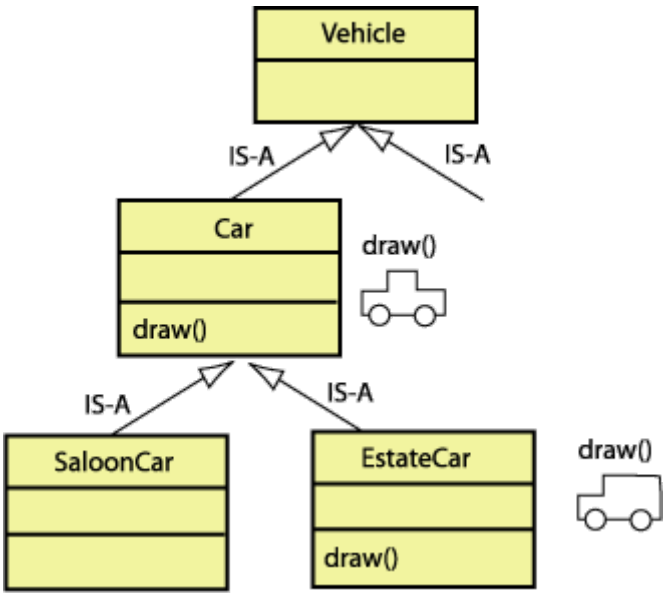
Polymorphism means "multiple forms". In OOP these multiple forms refer to multiple forms of the same method, where the exact same method name can be used in different classes, or the same method name can be used in the same class with slightly different parameters. There are two forms of polymorphism, **overriding** and **overloading**.

Overriding

As discussed, a derived class inherits its methods from the base class. It may be necessary to redefine an inherited method to provide specific behaviour for a derived class - and so alter the implementation. So, overriding is the term used to describe the situation where the same method name is called on two different objects and each object responds differently.

Overriding allows different kinds of objects that share a common behaviour to be used in code that only requires that common behaviour.

Figure 1.10. The overridden `draw()` method.



Consider the previous example of the `Vehicle` class diagram in Figure 1.7. In this case `Car` inherits from `Vehicle` and from this class `Car` there are further derived classes `SaloonCar` and `EstateCar`. If a `draw()`method is added to the `Car` class, that is required to draw a picture of a generic vehicle. This method will not adequately draw an estate car, or other child classes. Overriding allows us to write a specialised `draw()` method for the `EstateCar` class - There is no need to write a new `draw()` method for the `SaloonCar` class as the `Car` class provides a suitable enough `draw()` method. All we have to do is write a new `draw()` method in the `EstateCar` class with the exact same method name. So, Overriding allows:

- A more straightforward API where we can call methods the same name, even though these methods have slightly different functionality.
- A better level of abstraction, in that the implementation mechanics remain hidden.

Overloading

Overloading is the second form of polymorphism. The same method name can be used, but the number of parameters or the types of parameters can differ, allowing the correct method to be chosen by the compiler. For example:

```
add (int x, int y)
add (String x, String y)
```

are two different methods that have the same name and the same number of parameters. However, when we pass two String objects instead of two int variables then we expect different functionality. When we add two int values we expect an intresult - for example 6 + 7 = 13. However, if we passed two String objects we would expect a result of "6" + "7" = "67". In other words the strings should be concatenated.

The number of arguments can also determine which method should be run. For example:

```
channel()
channel(int x)
```

will provide different functionality where the first method may simply display the current channel number, but the second method will set the channel number to the number passed.

Abstract Classes

An abstract class is a class that is incomplete, in that it describes a set of operations, but is missing the actual implementation of these operations. Abstract classes:

- Cannot be instantiated.
- So, can only be used through inheritance.

For example: In the Vehicle class example previously the draw() method may be defined as abstract as it is not really possible to draw a generic vehicle. By doing this we are forcing all derived classes to write a draw() method if they are to be instantiated.

As discussed previously, a class is like a set of plans from which you can create objects. In relation to this analogy, an abstract class is like a set of plans with some part of the plans missing. E.g. it could be a car with no engine - you would not be able to make complete car objects without the missing parts of the plan.

Figure 1.11. The abstract draw() method in the Vehicle class.

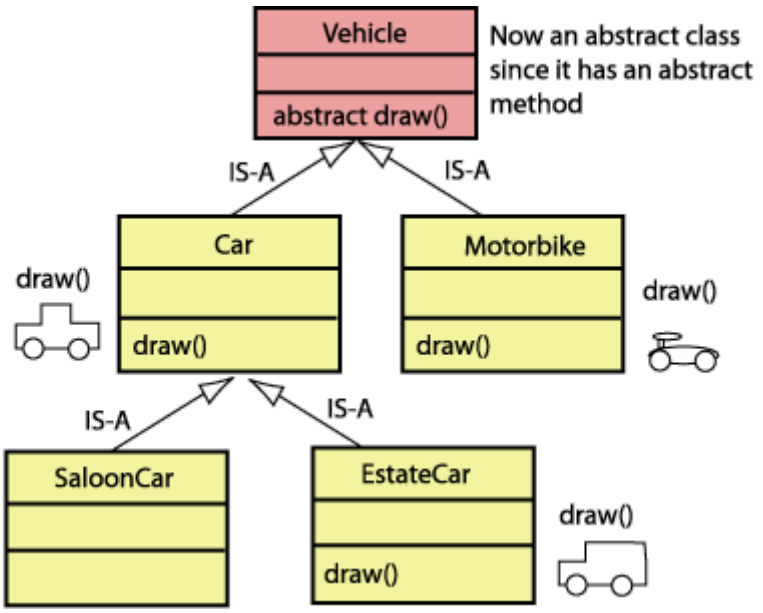
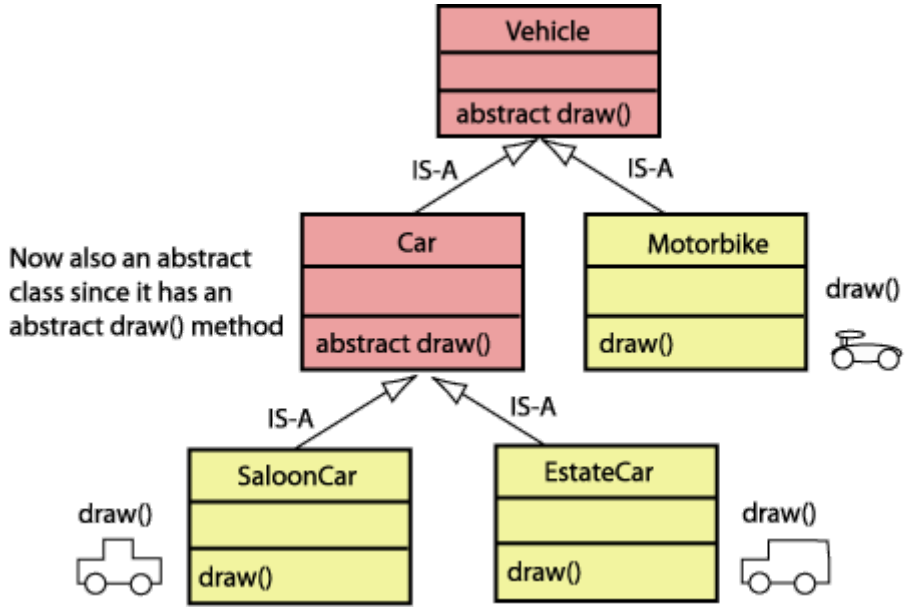


Figure 1.11 illustrates this example. The draw() has been written in all of the classes and has some functionality. The draw() in the Vehicle has been tagged as abstract and so this class cannot be instantiated - i.e. we cannot create an object of the Vehicle class, as it is incomplete. In Figure 1.11 the SaloonCar has no draw() method, but it does inherit a draw() method from the parent Car class. Therefore, it is possible to create objects of SaloonCar.

If we required we could also tag the draw() method as abstract in a derived class, for example we could also have tagged the draw() as abstract in the Car class. This would mean that you could not create an object of the Car class and would pass on responsibility for implementing the draw() method to its children - see Figure 1.12.

Figure 1.12. The abstract draw() method in the Vehicle and Car classes.



Object-Oriented Analysis and Design

As discussed previously, object-oriented programming has been around since the 1990s. Formal design processes when using objects involve many complex stages and are the debate of much research and development.

Why use the object-oriented approach?

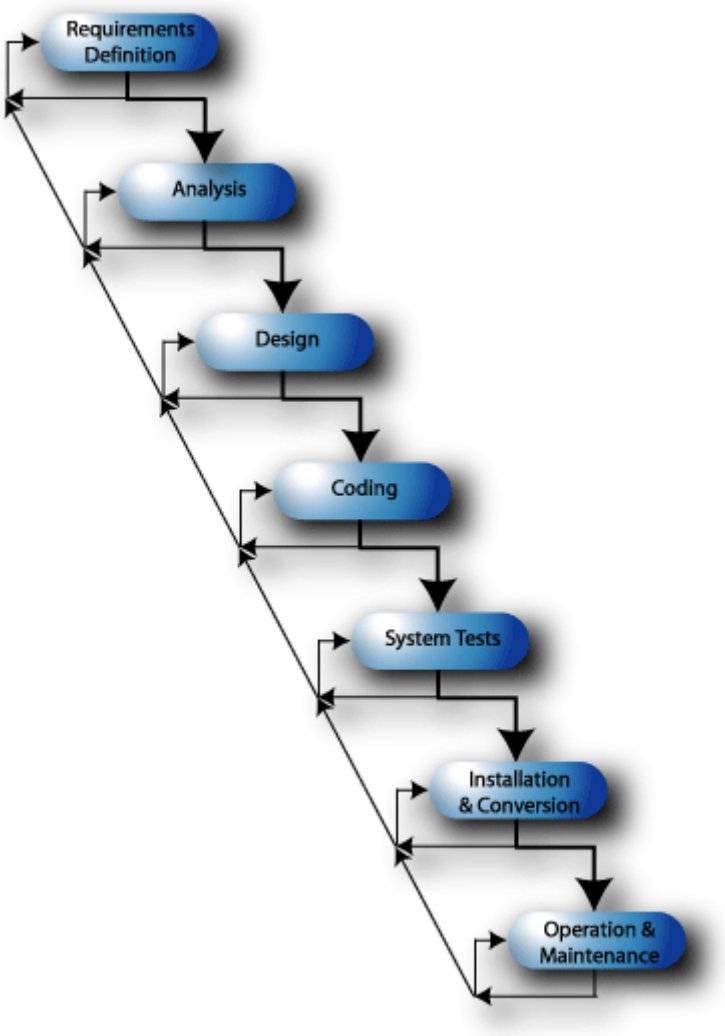
Consider the general cycle that a programmer goes through to solve a programming problem:

- **Formulate the problem** - The programmer must completely understand the problem.
- **Analyse the problem** - The programmer must find the important concepts of the problem.
- **Design** - The programmer must design a solution based on the analysis.
- **Code** - Finally the programmer writes the code to implement the design.

The Waterfall Model

The **Waterfall Model**[1], as illustrated in Figure 1.13, is a linear sequential model that begins with definition and ends with system operation and maintenance. It is the most common software development life cycle model and is particularly useful when specifying overview project plans, as it fits neatly into a Gantt chart format[2].

Figure 1.13. The Waterfall Model



The seven phases in the process as shown in Figure 1.13 are:

- **Requirements Definition**: The customer must define the requirements to allow the developer to understand what is required of the software system. If this development is part of a larger system then other development teams must communicate to develop system interfaces.
- **Analysis**:The requirements must be analysed to form the initial software system model.
- **Design**: The design stage involves the detailed definition of inputs, outputs and processing required of the components of the software system model.
- **Coding**: The design is now coded, requiring quality assurance of inspection, unit testing and integration testing.
- **System Tests**: Once the coding phase is complete, system tests are performed to locate as many software errors as possible. This is carried out by the developer before the software is passed to the client. The client may carry out further tests, or carry out joint tests with the developer.
- **Installation and Conversion**: The software system is installed. As part of a larger system, it may be an upgrade; in which case, further testing may be required to ensure that the conversion to the upgrade does not affect the regular corporate activity.

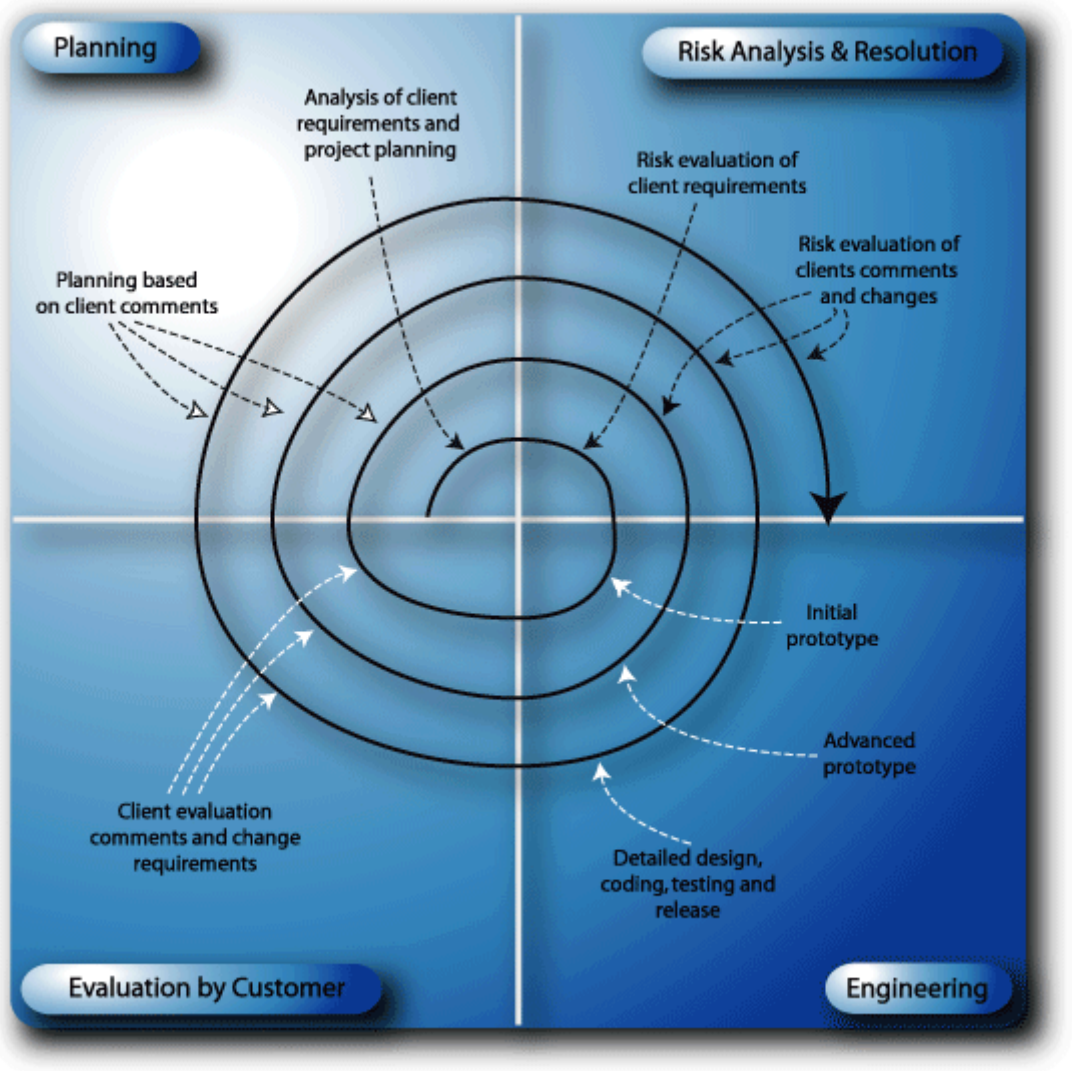
- **Operation and Maintenance**: Software operation begins once it is installed on the client site. Maintenance will be required over the life of the software system once it is installed. This maintenance could be **repair** oriented, to fix a fault identified by the client, **adaptive** to use the current system features to fulfill new requirements, or **perfective** to add new features to improve performance and/or functionality.

The Waterfall Model is a general model, where in small projects some of the phases can be dropped. In large scale software development projects some of these phases may be split into further phases. At the end of each phase the outcome is evaluated and if it is approved then development can progress to the next phase. If the evaluation is rejected then the last phase must be revisited and in some cases earlier phases may need to be examined. In Figure 1.13 the thicker line shows the likely path if all phases are performing as planned. The thinner lines show a retrace of steps to the same phase or previous phases.

The Spiral Model

The **Spiral Model**[3] was suggested by Boehm (1988) as a methodology for overseeing large scale software development projects that show high prospects for failure. It is an iterative model that builds in risk analysis and formal client participation into prototype development. This model can be illustrated as in Figure 1.14.

Figure 1.14. The Spiral Model



The spiral, as shown in Figure 1.14 of development is iterative, with each iteration involving planning, risk analysis, engineering (from design, to coding, testing, installation and then release) and customer evaluation (including comments, changes and further requirements). More advanced forms of this model are available for dealing with further communication with the client.

The spiral model is particularly suited to large scale software development projects and needs constant review. For smaller projects an agile development model is more suitable.

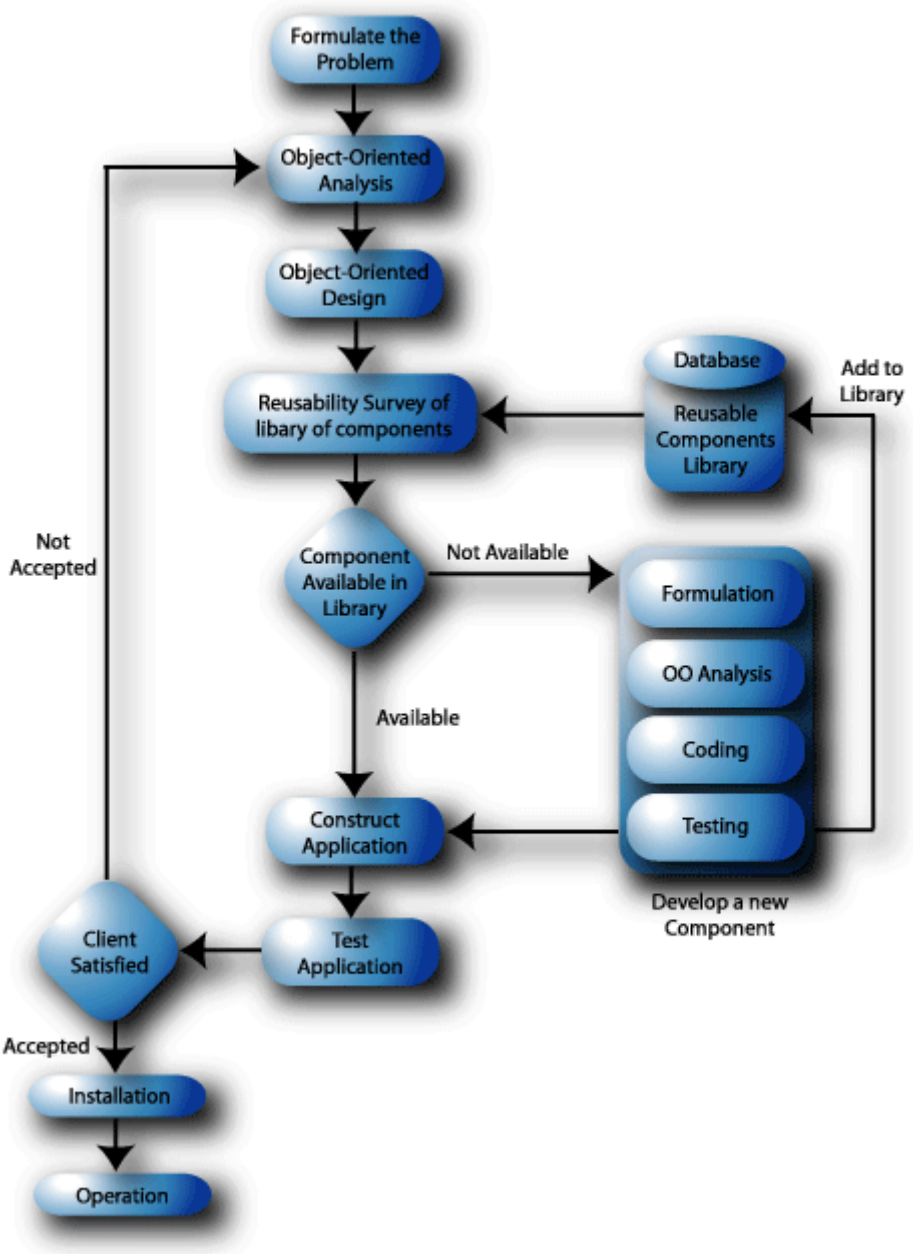
The Object-Oriented Design Model

One object-oriented methodology is based around the re-use of development modules and components. As such, a new development model is required that takes this reuse into account. The object-oriented model as shown in Figure 1.15 builds integration of existing software modules into the system development. A database of reusable components supplies the components for reuse. The object-oriented model starts with the formulation and analysis of the problem. The design phase is followed by a survey of

the component library to see if any of the components can be re-used in the system development. If the component is not available in the library then a new component must be developed, involving formulation, analysis, coding and testing of the module. The new component is added to the library and used to construct the new application.

This model aims to reduce costs by integrating existing modules into development. These modules are usually of a higher quality as they have been tested in the field by other clients and should have been debugged. The development time using this model should be lower as there is less code to write.

Figure 1.15. The Object-Oriented Design Model



The object-oriented model should provide advantages over the other models, especially as the library of components that is developed grows over time.

An Example Design Problem

Task: If we were given the problem; "Write a program to implement a simple savings account"… The account should allow deposits, withdrawals, interest and fees.

Solution: The problem produces many concepts, such as bank account, deposit, withdrawal, balance etc.. that are important to understand. An OO language allows the programmer to bring these concepts right through to the coding step. The savings account may be built with the properties of an account number and balance and with the methods of deposit and withdrawal, in keeping with the concept of the bank account. This allows an almost direct mapping between the design and the coding stages, allowing code that is easy to read and understand (reducing maintenance and development costs).

OOP also allows software reuse! … The concept of this savings account should be understood, independent of the rest of the problem. This general savings account will certainly find re-use in some other financial problem.

So after discussion with the client, the following formulation could be achieved - Design a banking system that contains both teller and ATM interaction with the rules:

- The cashiers and ATMs dispense cash.
- The network is shared by several banks.
- Each transaction involves an account and documentation.
- There are different types of bank accounts.
- There are different kinds of transactions.
- All banks use the same currency.
- Foreign currency transactions are permitted.
- ATMs and tellers require a cash card.

Step 1. Identify Possible Classes

- ATM, cashier, cashier station, software, customer, cash.
- banking network, bank.
- transaction, transaction record.
- account, deposit account, long term savings account, current account.
- withdrawal, lodgement, cheque.
- currency.
- foreign currency, euro cheque.
- cash card, computer system.

Step 2. Remove Vague Classes

- software, computer system, cash.

Step 3. Add New classes that arise!

- currency converter

Step 4. Create Associations

- **Banking Network** (includes cashiers and ATMs)
- **Banks** (holds accounts)
- **Account** (has a balance, a currency, a log of transactions)
- **Transaction** (requires a cash card)
- **Lodgement** (has an account number, an amount)
- **Withdrawal** (has an account number, an amount)
- **Cheque** (is a withdrawal, has a payee, an amount)
- **Eurocheque** (is a cheque, has a currency)
- **ATMs** (accept cash cards, dispense cash)

Step 5. Refine the Classes

Bank:

- has a name
- has accounts
- has a base currency
- has a sort code

Account:

- has an owner
- has a balance
- has an account number
- has a log of transactions

Deposit Account:

- is an account
- has a shared interest rate

Current Account:

- is an account
- has an overdraft limit

Transaction:

- has an account
- has a date
- has a value
- has a bank
- has an account Number
- has a number

Withdrawal:

- is a transaction

Lodgement:

- is a transaction

Cheque:

- is a withdrawal
- has a payee

EuroCheque:

- is a cheque
- has a currency

CurrencyConverter:

Step 6. Visual Representation of the Classes

Figure 1.16. The Bank class.

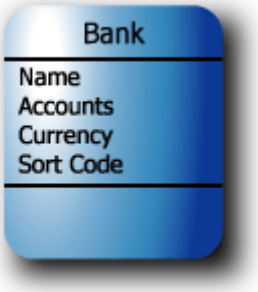


Figure 1.17. The Account class.

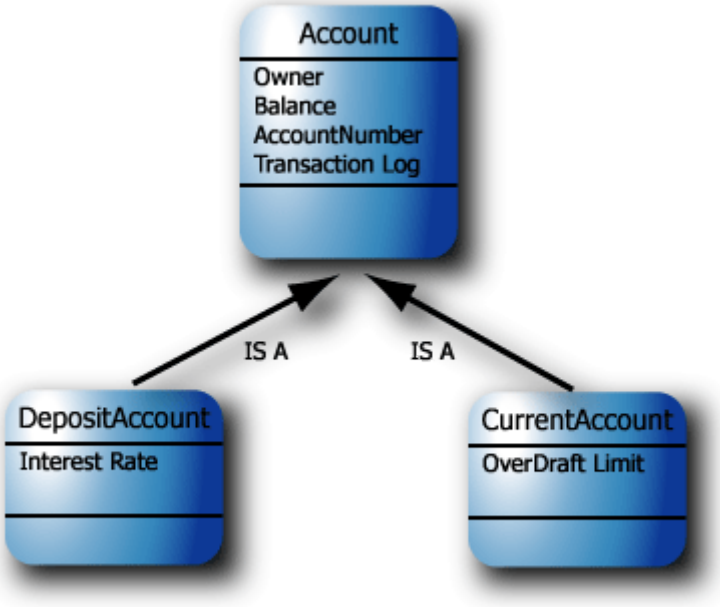
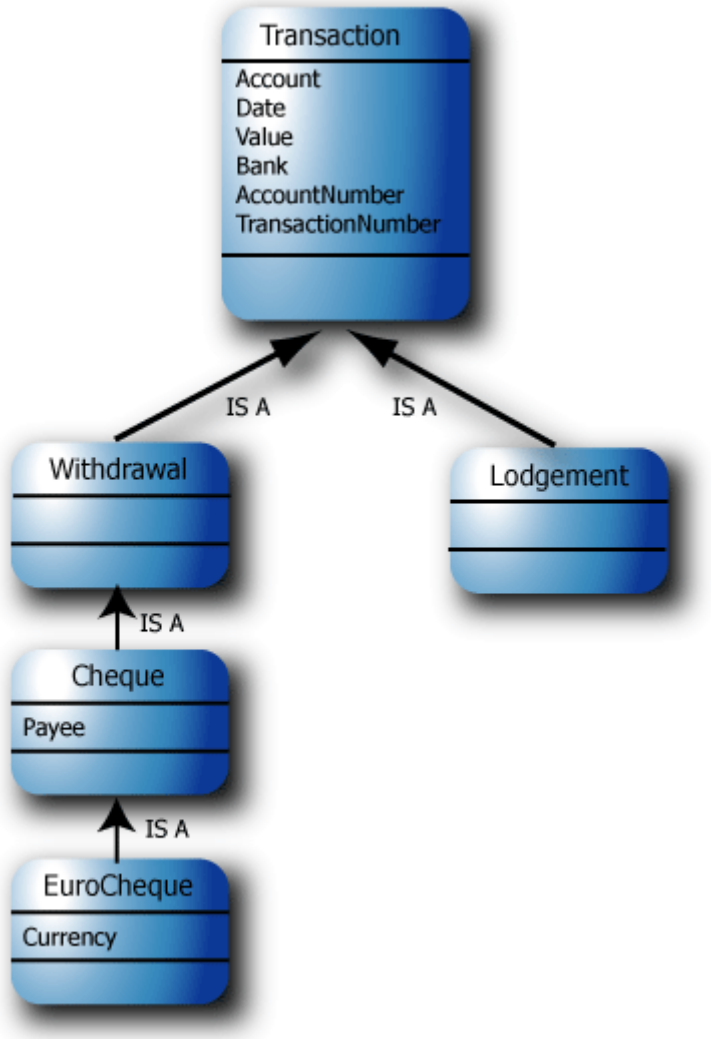


Figure 1.18. The Transaction class.



OOP Assessments

Self-assessments allow you to check your understanding of a topic using multiple choice questions. These self-assessments are corrected on-line and provide explanations for questions that you may have answered incorrectly.

Please find the self assessments on the Loop page for this module.

---

[1] Boehm, B. W. (1981) Software Engineering Economics, Ch. 4 Prentice Hall, Upper Saddle River, NJ.

Royce, W. W. (1970) "Managing the development of large software systems: concepts and techniques", Proceedings of IEEE WESCON, August 1970.

[2] http://en.wikipedia.org/wiki/Gantt

[3] Boehm, B. W. (1988) "A spiral model of software development and enhancement", Computer, 21(5), 61-72.

Chapter 2 - Introduction to C++ (The 'C' of C++)

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off."

— Bjarne Stroustrup

Introduction

C++ is an object-oriented language. So to use C++:

- Engineers define classes and create objects from these classes.
- Compose an application from existing and new C++ modules.
- Decompose the application into modules.

C++ was developed by Bjarne Stroustrup[4] at Bell Labs (now AT&T Labs) during 1983-1985. It is based on the C language (named in 1972) that was developed at AT&T for Unix systems in the early 70's (1969-1973) by Dennis Ritchie. Believe it or not, the C language was a descendent of the B language developed in 1970 by Ken Thompson, which in turn was a descendent of the BCPL language developed by Martin Richards, a Cambridge Student visiting MIT.

As well as adding object-oriented technologies (originally he called it C with Classes), C++ also improves the C language by adding features such as better type checking. The language very quickly came into widespread use and is today one of the best known object-oriented languages. The fast uptake was mainly due to its similarity to the C programming syntax, being an extension to the C language that allowed existing C code to be used when possible. C++ is not a purely object-oriented language, rather it is a hybrid - having both the organisational structure of object-oriented languages, but retaining the efficiencies of C, such as types, pointers etc.

In 1998 the ANSI/ISO (American Standard Institute/International Organization for Standardisation) committee adopted a worldwide uniform language specification that was to remove inconsistencies between the various C++ compilers [5]. Unfortunately not all compilers support this standard, even today. This is a problem with C++, as each vendor can have their own flavour of C++, where code developed on one compiler on one operating system may not necessarily compile on another compiler on another operating system.

This standardisation continues today with C++11 approved by the ISO in 2011 and more new features appearing in compilers today with the approval of C++14 in August 2014. C++23 is the most recent version of C++ programming language revised by ISO/IEC 14882 standard, agreed in Feb 2023. C++26 is under discussion.

[4] Bjarne Stroustrup home page:

http://www.research.att.com/~bs/homepage.html

[5] The final technical vote took place November 14, 1997. After the technical vote, the standard was ratified in 1998 by a 22-0 national vote. The standard is ISO/IEC 14882 and is available at the ANSI Webstore, price $18 (2004) **ANSI Webstore**

A First Application

> "I saw 'cout' being shifted 'Hello world!' times to the left and stopped right there..."
>
> --Steve Gonedes, 2000

The first program in a new language should always be "Hello world!". Line numbers are only present below to assist with lecture discussion.

```
1
2   // Hello World Application
3   #include <iostream>                //1
4   using namespace std;               //1
5
6   int main() {                       //2
7     cout << "Hello world!\n";         //3
8     return 0;                        //4
9   }
```

❶ The iostream.h header file is required for the cout call. The #include is a C++ preprocessor directive [6] , that causes the C++ preprocessor at compile time to substitute the contents of the specified file into your program at that location. Using the #include directive allows you to import libraries of code into your application as required. The size of your application will increase by the size of each header file included, but the inclusion of iostream.h is necessary to perform any input/output calls, such as printing to the screen in C++ (we will discuss this later). The using namespace line is a new feature of C++ which properly imports the header file into the namespace (we will discuss properly later).

❷ The main() function is the starting point for all command line C and C++ applications. The default return type in C++ is int, meaning that this function should return a whole number.

❸ The cout call sends the "Hello World!" string to the standard output stream, usually the screen. The << operator evaluates the parameter that follows it and places it on the output stream. To end a line of output you can use << "\n" or << endl.

❹ The return 0 statement tells this function to return a value of 0 as the whole number to the calling code. This statement is not necessary as a value of 0 will be returned from the main() function by default.

The source for this example is in **`HelloWorld.cpp`** which is attached to the bottom of this page.

You might think that this is the shortest C++ program that you can have - It is not! The shortest valid C++ program is:

```
main()
 {

 }
```

There is no need for libraries, as there is no input/output, there is no need to specify a return type for the main() function as it defaults to a return type of int and an empty method is a valid method. This shows how flexible C++ is in **assuming** default states and values. This is also one of C++'s greatest weaknesses!

In the upcoming and most modern agreed version of C++ (C++23), the Hello World application will be adapted slightly as follows:

```
#include <print>

int main() {
   std::println("Hello World!");
}
```

Few compilers currently support C++23 and g++ only supports the standard with the -std=c++2b compiler setting in early releases. The previous code will continue to function correctly.

Setting up a compiler

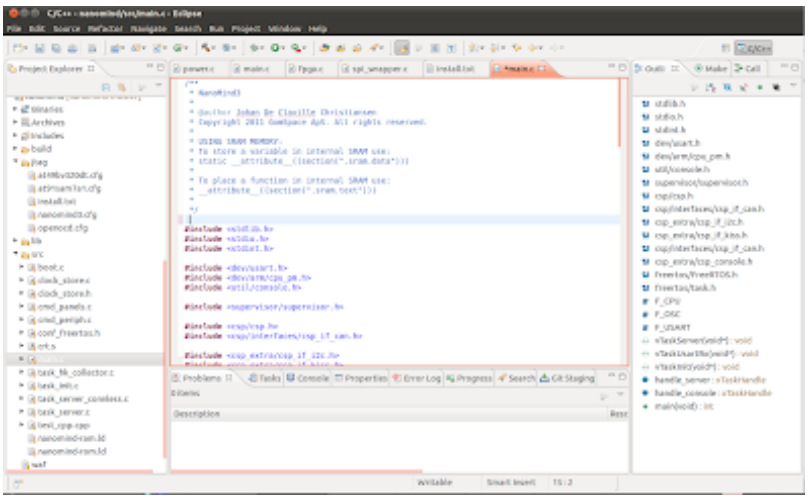
> "C++ : an octopus made by nailing extra legs onto a dog"
>
> --Unknown

There are several C++ compilers that you can use for this course. Once the compiler is ANSI compliant there should be no issue in using it with this module as we just require a standard non-windowing compiler. A Unix/Linux compiler will also work. The current recommended compiler is Eclipse CDT.

Eclipse CDT

"The CDT Project provides a fully functional C and C++ Integrated Development Environment based on the Eclipse platform. Features include: support for project creation and managed build for various toolchains, standard make build, source navigation, various source knowledge tools, such as type hierarchy, call graph, include browser, macro definition browser, code editor with syntax highlighting, folding and hyperlink navigation, source code refactoring and code generation, visual debugging tools, including memory, registers, and disassembly viewers. " - http://www.eclipse.org/cdt/

Figure 2. The Eclipse CDT Integrated Development Environment (IDE)



```
// Hello World Application
#include<iostream.h>
int main()
{
  cout << "Hello World!\n";
}
```

---

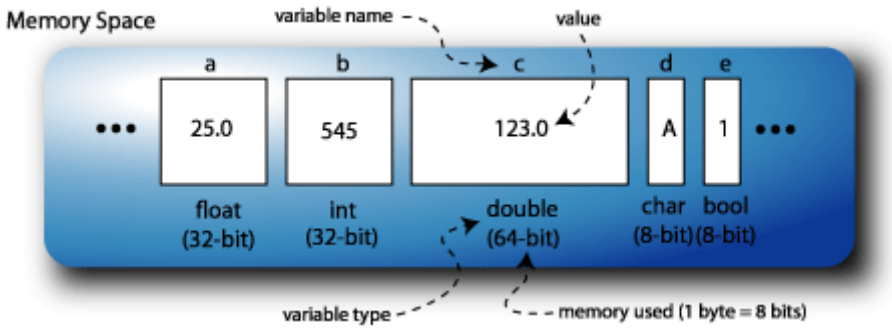
[6] Preprocessor directives are orders for the preprocessor, not for the program itself. They must be specified in a single line of code and should not end with a semicolon.

C++ General Language

Variables

A variable is a data item stored in a block of memory that has been reserved for it. The type of the variable defines the amount of memory reserved. This is illustrated in Figure 2.6.

Figure 2.6. An example memory space with variables defined.



C++ supports many variable types, such as:

- int integers (-5, -1, 0, 1, 3 etc.)
- char characters ('a', 'b', 'c', etc.)
- float floating point numbers (4.5552324, 1.001, -4.5553 etc.)
- double larger more accurate floating point numbers (i.e. to more decimal places or with a larger magnitude)
- long (long int) larger integer value range than int if using 16-bit ints.
- bool contains either the true or false value (i.e. a 1 or 0 respectively).
- short (short int) smaller integer value range than int
- unsigned int 0,1,2,3, etc.
- unsigned long 0,1,2,3,4, etc.

Variables may be defined using these types, as illustrated in Figure 2.6.

```
int main()
{
       float a = 25.0;
       int b = 545;
       double c = 123.0;
       char d = 'A';
       bool e = true;
}
```

The source for this test program is given in **SizeofVariables.cpp** **(at the bottom of the page)** Using these variables we can assign values to them, modify them and print them to the output if required:

```
1
2 // Variables Application
3
4 #include<iostream.h>
5
6 int main()
7 {
8   int x = 7, y = 10;                    //1
9
10   x=2;    // assign variable x a value of 2
11   x++;    // increment x by 1
12   x+=2;   // increment x by 2
13
14   cout << "x equals " << x << endl;     //2
15 }
16
```

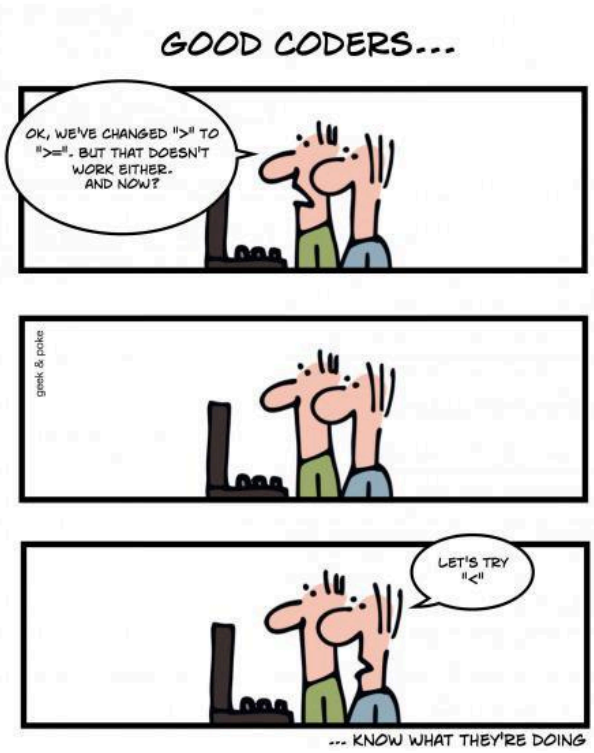
The source for this is in **Variables1.cpp**

❶ You can define several variables on the one line.

❷ At this point, the program will result in an output of

```
x equals 5
```

Notes about variables:



- Variables can be introduced as required!
- cin allows values to be read in.
- C++ will usually complain if you assign a value of one type to a variable of another type of lower resolution.
- Variables can be initialised as they are defined.
- We can use the const statement to protect the value of a variable from change. In embedded applications, const variables may be stored in ROM (rather than RAM) and will only be placed there once, no matter how many times they are used. As such, use const rather than #define whenever possible.
- A volatile variable is one that can change outside of the control of the compiler, such as value changed by hardware, threading or interrupts. We use the volatile keyword to tell the compiler not to perform any form of optimisation on this data. We can also set this value as const volatile to prevent the programmer from changing this value - It can still change, but outside the control of the programmer.

There are certain conversion rules for basic types:

```
1
2  // Using variables with automatic conversion
3
4  #include<iostream.h>
5
6  int main()
7  {
8    int x,y;        //(see 1)
9
10   x = 6.73;       // x becomes 6
11   cout << "x = " << x << "\n";
12
13   char c = 'w';  // (see 1)
14   cout << "c = " << c << "\n";
```

```
15
16    x = c;          // x becomes the integer
17                    // equivalent of 'w' which is 119
18    cout << "x = " << x << "\n";
19
20    y = 2.110;      // y becomes 2
21    double d;       // (see 3)
22    d = y;          // d becomes 2.0
23    cout << "d = " << d << "\n";
24
25    const float pi = 3.14159;
26                   // (see 4)
27  //pi = 223.34;  // would be an error if included
28  }
29
```

The source for this is in **Variables2.cpp**

❶ x and y are being "declared" as variables. Variables in C++ are not automatically initialised to zero, so it would be better practice to use the statement int x=0, y=0;

❷ The variable c is initialised as it is defined.

❸ The variable d is introduced as required.

❹ The pi variable is defined as constant so that it cannot be modified without causing a compile-time error.

This program will output:

```
x = 6
c = w
x = 119
d = 2
```

The byte size of your variables matters and can have a significant impact on your code. For embedded applications floating point operations are very expensive, especially if you do not have a hardware floating point unit (what was once called a maths coprocessor). On the other hand, if you choose incorrect precision then strange things can happen. Take this very short segment of code:

```
int main(){
       float balance = 200000000; // My balance is €200 million
       balance = balance + 1.00;  // Make a deposit of €1
       printf("My balance is now €%12.2f \n", balance);
```

```
}
```

What will this output? Well you might expect that the balance of my bank account would now be €200,000,001, but when compiled using a 32-bit compiler it is actually:

```
My balance is now €200000000.00
```

And I have lost €1. I can live with that, but if it was a 'for loop' with 100,000,000 x €1 lodgements, I would lose them too! Big numbers tend to consume small numbers on embedded devices.

Why does this happen? Well the precision of a 32-bit float is about 7 decimal places and since this number will actually be stored as 2.0e+8, it cannot represent 2.00000001e+8. We could easily solve this by using 64-bit floating point numbers in this case, but please remember that the same problem will recur, just with bigger numbers.

typedef

When working with embedded devices it can be important to define exactly what we mean when working with types that are machine dependent. For example int might be a 32-bit number on one platform and a 16-bit number on a different platform. Clearly, this could result in errors if you were trying to port a code library between two such devices. To assist with this there are standard types available in the <stdint.h> C header file, and are included by default on recent C++ compilers. These include:

	Signed	**Unsigned**
8-bit	`int8_t`	`uint8_t`
16-bit	`int16_t`	`uint16_t`
32-bit	`int32_t`	`uint32_t`
64-bit	`int64_t`	`uint64_t`

Use these types whenever you require portability and readability.

We also sometimes need to give a variable type another name. We can use typedef to reduce the apparent complexity of the code, for example:

```
typedef unsigned char   uchar;
typedef unsigned int    cardinal;
typedef int integer;
etc..
```

We can then just use this defined type as normal:

```
integer x;
```

You should use this carefully and only where the definition of a necessary data type is required. If you type define int as elephant, it may make your code more interesting, but it will make it difficult for another programmer to comprehend.

One side effect in C++ is that if you are defining:

```
int* a,b;
```

it does not create two pointers, rather one int pointer a and one int variable b as the * binds to the right. If you were to use a typedef for this then we would not have the same problem. E.g.

```
typedef int* intPointer;
intPointer a,b;
```

declares two pointers a and b, both of type int.

Volatile Variables

Compilers are designed to optimise your code and generate efficient program code, however this sometimes leads to unintended consequences. For example, take the following segment of code:

```
int a, b;  //using global variables for emphasis

void function(){
   a = 10;
   b = a * 7;
   if (a == 10) {
      cout << "a has the value 10" << endl;
   }
   else {
      cout << "a does not have the value of 10" << endl;
   }
}
```

Since the value of the variable a does not change between when it is assigned and when it is compared, an optimising compiler might reduce this code to something like:

```
int a, b;  //using global variables for emphasis

void function(){
   a = 10;
   b = a * 7;
   if (a == 10) {
      cout << "a has the value 10" << endl;
   }
   else {
      cout << "a does not have the value of 10" << endl;
   }
}
```

Depending on your program and intention, this might not be correct or safe. For example, if you are working on a multithreaded embedded device, or a device where the variable a is actually mapped to an input/output hardware then it could indeed be possible that the variable a is modified external to the function between when the variable is created and the subsequent comparison. Therefore it might not be correct for you to allow the compiler to optimise this code.

The keyword **volatile** allows us to inform the compiler that a variable can be modified outside of the program it is compiling, and that it should not make assumptions about a variable that lead to incorrect optimisation. To fix this in the function above you can simply state:

```
volatile int a;
int b;

void function(){
  ...
}
```

Now, any code that uses the variable a will not optimise the code related to that variable. In embedded applications, this can also be useful for setting up delays that might be needed for serial communication. For example,

```
void delay(){
   volatile int a = 0;
   while (a++<100000) {}
}
```

The loop here would likely be removed by an optimising compiler, but might be needed for an embedded application. Making the variable a in this case volatile prevents this removal.

The level of compiler optimisation is set at compile time. For example, with a simple C++ program:

```
molloyd@desktop:~/een1035$ g++ test.cpp -o test -O3
```

Where O3 sets the most aggressive optimization level. The default, -O0 is the least aggressive option, which is suitable for code development and debugging and then O1 to O3 increase the level of optimisation by performing additional tests on the code, which increases compilation time.

Namespaces

In the development of large C++ applications that involve several different programmers, two programmers could use the same name for a global variable or class, representing related concepts. This would cause complications as such conflicts can have

unpredictable results (more likely compiler errors). So, while individual code segments would work independently, when they are brought together errors may result.

The namespace concept was introduced during the standardisation of C++, to allow the programmer to define a namespace that is limited in scope. When writing C++ applications we can make it explicit that we are using the standard namespace by a "using directive":

```
using namespace std;     // i.e., I want to use the declarations and
                         // definitions in the "Standard Library"
                         // namespace
```

at the beginning of our code segments, however this can be considered poor practice in certain circumstances as the entire contents of the namespace are included. The alternative is to make it explicit that we were calling the standard cout output stream by typing:

```
std::cout << "Hello World!" << std::endl;
```

which states that the cout and endl that we wish to use are both defined in the std standard library.

It is possible to include the exact namespace member to be included by a "using declaration"::

```
using std::cout;
```

would allow us to use cout without including all of the names in the std namespace.

When you include header files you can type #include<iostream.h> or #include<iostream>, but be careful, there is a difference between the two statements. Writing:

```
#include<iostream.h>
```

Is the same as writing:

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

It is possible to create your own namespace, by using the namespace grouping. For example, here I have created a namespace called MolloySpace that contains a function called testAdd. To use this function in the code below you have to use the "using namespace" directive or "using" declaration, otherwise the code will not compile.

```
#include<iostream>

namespace MolloySpace
{
  float testAdd(float a, float b) { return a+b; }
```

```
    class Time
    {
        public: //etc.
    }
}

using std::cout;
using std::endl;

using namespace MolloySpace;
//using MolloySpace::Time;

...
```

The source code for this is in **NameSpaceTest.cpp**

Note

It is good practice to use a noun (such as your surname) as part of the namespace name. You can have namespaces within namespaces (e.g. DerekSpace within MolloySpace, but the notation of MolloySpace::DerekSpace::SomeClass is becoming unwieldy so we can define an alias, eg. namespace DerekMolloySpace = MolloySpace::DerekSpace;

Note

There is a subtle difference in functionality between a using declaration and a using directive when multiple namespaces are used with a common shared name. For example:

Assuming we have two namespaces MolloySpace1 and MolloySpace2, but both namespaces have a function someFunction(). If we use the "using directive":

```
using namespace MolloySpace1;
using namespace MolloySpace2;
```

There would be no error provided we do not use someFunction()

However, if we use the "using declaration" and typed:

```
using MolloySpace1::someFunction;
using MolloySpace2::someFunction;
```

This would result in a compiler error and so would be detected.

Which should you use? To get the full use of namespaces, it is good to avoid:

```
using namespace someNameSpace;
```

Placing such a using directive at the start of every file is the same as placing all definitions in the global namespace - exactly what namespaces were invented to avoid! So, this approach gives little value out of the namespace mechanism. If you place the using directive inside a block, then it only applies to that block; a more sensible approach. The using declaration is better most of the time, inserting statements like:

```
using theSpace::f;
```

at the start of a file, allows you to omit names that are in the namespace but that are not used, avoiding potential name conflicts. It also documents which names you use, and it is not as messy as always qualifying a name with notation of the form theSpace::f.

Comments

Comments are a necessary evil! A good programmer uses useful comments efficiently in their code. In fact, you may find that sometimes it may clear your head to lay out an algorithm by writing the comments first, before writing even one line of code. We have two types of comments in C++ (i) End of line comment and (ii) block comments.

```
x = x * 4.533;  // explain what you are doing
                // end of line commenting

/* An example of block commenting */
y = 5 * 3;

/* TODO: This section needs to be fixed! Derek 25/12/99
j = j * 32.7 + 6;
k = k * j + 1;
*/
```

Important: You cannot nest /* .. */ comments!

There is a third (and very useful) comment form in Java (/** .. */) that allows for the automatic documentation of code.

Methods (or Functions) in C++

Code that is needed in a number of places, should be grouped as a method and called when required. Methods should be kept as short as possible (including the main() function).

So to write a simple function that returns a single value we can use:

```
1
2 // Using Functions/Methods
3
4 #include<iostream>
5 using namespace std;
6
7 float addInterest(float val, float rate)  //1
8 {
9     return val + (val * (rate/100)); //2
10 }
11
12 int main()
13 {
14     float balance = 5000;
15     float iRate = 5.0;
16
17     balance = addInterest(balance, iRate); //3
18
19     cout << "After interest your balance is "
20          << balance << " Euro." << endl;
21 }
22
```

The source code for this is in **Functions.cpp**

❶ The function is defined to return a float value, so it is required to do so. In this case the return value will be the new balance.

❷ The return keyword defines the value to return. Since the return type has been specified as float the return value must also be of float type.

❸ The return value of the function is assigned to the left-hand side of the =. In this case the balance value has been modified to update the balance in the main() function.

This will result in the output:

```
After interest your balance is 5250 Euro.
```

Some points about methods:

- If a function/method has a return type then it must return a value!
- The void keyword implies that no return value is expected. void can also mean "type undefined" (to be discussed later).
- int is the default return type in C++.
- char* is the string type (pointer to characters).

- In C if we specified a function with no parameters, e.g. void someFunction() it actually meant that there was an undetermined number of parameters, thus disabling type checking. In C++ this means that there are zero parameters.

The previous code segment passed values to the function using **pass by value**. In this case you are really passing the value of the variables balance and iRate, so in this case the numbers 5000 and 5.0. It is only possible to have one return value when passing by value (However, this value could be a pointer to an array). If we wish to have multiple return values, or wish to modify the source then we can **pass by reference**.

This example is the same as the previous example except this time we are **passing by reference**:

```
1
2  // Using Functions/Methods (with Pass by reference)
3
4  #include<iostream>
5  using namespace std;
6
7  void addInterest(float &val, float rate) //1 2
8  {
9      val = val + (val * (rate/100)); // 3
10 }
11
12 int main()
13 {
14     float balance = 5000;
15     float iRate = 5.0;
16
17     addInterest(balance,iRate); // 4
18
19     cout << "After interest your balance is "
20          << balance << " Euro." << endl;
21 }
22
```

The source code for this is in **Functions2.cpp**

❶ The function is defined not to return a value, so the return type is set as void. We do this to prove that the pass by reference actually works.

❷ You will notice that the first parameter val has been changed so that it has an & in front of it. This is notation to signify that the val parameter is to be passed by reference, not by value.

❸ You will notice the return keyword is not used, as the return type is set to void. The value is not returned, rather it is modified directly. So when the

value of val is modified in the addInterest() method it modifies the value of balance directly.

❹ The balance is passed in the same way, but the function receives the reference in this case, not the value. The balance variable has been updated by the function and the new value is displayed as below

This will result in the output:

```
After interest your balance is 5250 Euro.
```

So the output is exactly the same as in the pass-by-value case. Passing-by-reference is like passing a copy of the name of the variable to the method.

Note

In C++ you can actually leave out the name of a parameter in a function definition. e.g. int add(int a, int b, int) { return a+b; }. You might do this to create a function that will have a third parameter in the future and you may also wish to avoid compiler warnings that result if a variable is defined, but not used in a section of code.

Strings in C++

The C++ language has no built-in type for strings, rather they are treated as an array of the char type terminated by the null character '\0'. A character constant is an integer represented in inverted quotes around an integer value. For example 'A' = 65, 'a' = 97. ANSI/ISO C++ does provide standard C and C++ libraries for the use of strings.

C Style String Processing

To use the C standard library for strings in your application use #include<cstring>. The inclusion of the header file <cstring> actually includes the library<string.h>, but this is the correct notation as it identifies it as a C header (rather than C++).

```
1
2 // C String Example
3 // Note - Just for an example, no "using namespace" directive
4
5 #include<iostream>
6 #include<cstring>
7
8 int main()
9 {
```

```
10      char s[20] = "hello "; //1
11      char t[] = { 'w', 'o', 'r', 'l', 'd', '!', '\0' }; //2
12
13      // modify the strings directly, replace h with H
14      s[0] = 'H'; //3
15
16      // compare strings
17      if (strcmp(t, "world!") == 0) //4
18      {
19        strcpy(t, "World!");   //note capital W //5
20      }
21
22      char *u = strcat (s, t); //6
23
24      // will output "Hello World!"
25      std::cout << u << std::endl;
26      std::cout << "This string is " << strlen(u) //7
27          << " characters long." << std::endl;
28  }
29
```

This application will output:

```
Hello World!
This string is 12 characters long.
```

The source code for this is in **CStringExample.cpp**

❶ The character array can be assigned with an initial string (the null character is automatic). The array is set to 20 characters long to allow for a string concatenation on line 22. Otherwise sufficient memory is not available.

❷ The character array can be assigned explicitly with characters (the null character is required).

❸ The string is an array of characters. This array of characters has the range of 0 to the length of the array.

❹ To compare two character arrays we can use the strcmp() function. It accepts two strings and returns 0 if the strings are the same, less than zero if the first string is lexicographically less than the second string and greater than zero if the first string is lexicographically more than the second string. i.e. does the first string come before or after the second string in the dictionary.

❺ The strcpy() function allows a direct assignment to a new string.

❻ The strcat() function allows a concatenation of strings. It returns a pointer to the string. Note that the string s has been modified by this operation and now contains the string "Hello World!"

❼ The length of a string can be found using strlen() function that returns the length of a string using a value of the int type.

Table 2.1. The C String Functions Reference

Function	Description
char *strcat(char *s, const char *t);	Append string t to string s. The first character of t replaces the NULL character at the end of s. The new value of s is returned.
char *strncat(char *s, const char *t, int n);	Append part of string t to string s. The first character of t replaces the NULL character at the end of s. n determines how many characters to append. The new value of s is returned.
char *strcpy(char *s, const char *t);	Assigns string t to string s. The new value of s is returned.
char *strncpy(char *s, const char *t, int n);	Assigns string t to string s. n determines how many characters to copy. The new value of s is returned.
int strlen(const char *s);	Returns the length of s as an integer (excluding the NULL character).
int strcmp(const char *s, const char *t);	Compare string t to string s. It returns 0 if the strings are the same, less than zero if the first string is lexicographically less than the second string and greater than zero if the first string is lexicographically more than the second string. i.e. does the first string come before or after the second string in the dictionary.
int strncmp(const char *s, const char *t, int n);	Compare the first n characters of the string t to string s. It returns 0 if the strings are the same, less than zero if the first string is lexicographically less than the second string and greater than zero if the first string is lexicographically more than the second string. i.e. does

	the first string come before or after the second string in the dictionary.

char *strtok(char *s, const char *t);	char *ptr, s[] = "Hello World";
	ptr = strtok(s, " "); //to break s into individual words
	
	while (ptr != NULL)
	{
	cout << ptr << std::endl;
	ptr = strtok(NULL, " ");
	}
	// please note that strtok inserts \0 into the " ",
	// so now s = "Hello"

C++ Style String Processing

The standard library provides us with functionality associated with strings such as concatenation provided by the + operator, assignment with the = operator and comparison with the == operator.

```
1
2  // C++ String example
3  // Note - Just for an example, no "using namespace" directive
4
5   #include<iostream>
6   #include<string>
7
8   int main()
9   {
10     // create new string variables
11     std::string s = "hello ";
12     std::string t = "world!";
13
14     // modify the strings directly, replace h with H
15     s[0] = 'H';
16
17     // compare strings
18     if (t == "world!")
19     {
20       t = "World!";     //note capital W
21     }
22
23     std::string u = s + t;
24
25     // will output "Hello World!"
26     std::cout << u << std::endl;
27     std::cout << "This string is " << u.length()
```

```
28          << " characters long." << std::endl;
29  }
30
```

The source code for this is in **`StringExample.cpp`**

❶ The std::string can be replaced by string when using the std namespace. An initial value can be assigned using = operator.

❷ The string can still be treated as an array of characters. This array of characters has the range of 0 to the "length of the string" - 1.

❸ The == operator allows a comparison of the strings, returning true or false.

❹ The = operator allows a direct assignment to a new string.

❺ The + operator allows a concatenation of strings.

❻ The length of a string can be found using s.length() that returns a value of the int type.

This application will also output:

```
Hello World!
This string is 12 characters long.
```

Table 2.2. The C++ String Methods

Method	Description

`append(char *ptr);`
`append(char *ptr, int n);`
`append(string &s, int offset, int n);`
`append(string &s);`
`append(int n, char ch);`
`append(InputIterator Start, InputIterator End);`
	Appends characters to a string from C-style strings, character arrays or other String objects. Please note that we will discuss iterators later on in this module.

`copy(char *cstring, int n, int offset);`	Copies n characters from a C-style string beginning at offset.
`*c_str();`	Returns a pointer to C-style string version of the contents of the Stringobject.

`begin();` `end();`	Returns an iterator to the start/end of the string.
`at(int offset);`	Returns a reference to the character at the specified position. Differs from the subscript operator [], in that bounds are checked.
`clear();`	Clears the entire string.
`empty();`	Tests if a string is empty.

`erase(int pos, int n);`
`erase(iterator first, iterator last);`
`erase(iterator it);`
	Erases characters from the specified positions.

`find(char ch, int offset = 0);`
`find(char *ptr, int offset = 0);`
`find(string &s, int offset = 0);`
	Returns the index of the first character of the substring when found. Otherwise, the special value "npos" is returned.

`find_first_not_of();`
`find_first_of();`
`find_last_not_of();`
`find_last_of();`
	This has the same sets of arguments as find. Finds the index of the first/last character that is/is not in the search string.

`insert(int pos, char *ptr);`
`insert(int pos, string &s);`
`insert(int pos, int count, char ch);`
`insert(iterator it, InputIterator start, InputIterator end);`
	Inserts characters at the specified position.

`push_back(char ch);`	Inserts a character at the end of the string.

`replace(int pos, int n, char *ptr);`
`replace(int pos, int n, string &s);`
`replace(iterator first, iterator last, char *ptr);`
`replace(iterator first, iterator last, string &s);`
	Replaces elements in a string with the specified characters.

`size();`	Returns the number of characters in a String object.
`swap(string &s);`	Swaps two String objects.

Standard Input - cin

We have seen the use of the cout output stream. The cin is the name of the standard input stream. The >> operator allows you to read information from the input stream and place it in the argument that follows it. For example, we can read in a value in the following way:

```
1
2  // cin Example
3  #include<iostream>
4  #include<string>
5  using namespace std;
6
7  int main()
8  {
9    cout << "What is your name?" << endl;
10   string s;
11   cin >> s;
12   cout << "Hello " << s << endl;
13 }
14
```

The source code for this is in **CinExample.cpp**. The output of this application is:

```
What is your name?
 Derek
Hello Derek
```

The >> operator ignores spaces, new-line and tab characters in the typed input. The operator has a different behaviour depending on whether strings or numbers are being entered:

- When reading in an int >> may take a + or - as a leading character and will read numeric characters until a non-integer character is reached, such as a space, letter or decimal point.
- When reading in double/float values a + or - will be accepted as a leading character and it stops at non-numeric characters, but will accept a decimal point. It will accept a leading 0 in front of a value, but it is not required.
- When reading in a string, the >> reads in all characters, but does not read in spaces, new line characters or tab characters.

If you enter an invalid value you can call cin.clear() to clear the stream's fail state. If the same value is entered again then the same problem will recur.

Assignment statements

An assignment takes the form: Variable = Expression. There are some shorthand versions, for example ++x; or x++; is the same as x=x+1;, or x*=2; is the same as x = x * 2;

Note that:

```
int i = 0;
while ( ++i < 10){
  cout << i << endl; // outputs 1 to 9
}
```

Whereas:

```
int i = 0;
while ( i++ < 10){
  cout << i << endl; // outputs 1 to 10
}
```

In the second case the increment takes place after the "less than" comparison. However, be very careful with the for loop case:

```
for(int i=0; i<10; i++){
  cout << i << endl; // outputs 0 to 9
}
```

Whereas:

```
for(int i=0; i<10; ++i){
  cout << i << endl; // outputs 0 to 9
}
```

These are both equivalent as the increment takes place after the statement in the loop has been executed.

Scope of Variables

The scope of a variable is the area in a program where the variable is visible and valid. If we examine the code segment:

```
void someMethod() {
  int y = 5;
  x++; // invalid - x is not defined in someMethod()
  y++; // valid - y now equals 6
}

int main() {
  int x = 1;
  x++; // valid - x now equals 2
  y++; // invalid - y is not defined in main()
}
```

The variable y is defined in someMethod() and so is only valid in that method. x is defined in main() function and so is only valid in that method.

A more complex case can be seen below:

```
main(){
   int x = 7;
   cout << "x = " << x << endl;
   {
     cout << "x = " << x << endl;
     int x = 2;
     cout << "x = " << x << endl;
   }
   cout << "x = " << x << endl;
 }
```

This code segment will result in the output:

```
x = 7
x = 7
x = 2
x = 7
```

The first definition of x is initialised with the value 7. This first x is displayed first and next within the {}. As a new x is then defined within the {} it now has scope and is displayed on the next line with a value of 2. Once we go outside the {} then that x variable is destroyed and scope once again returns to the original x variable, resulting in an output of 7. Although the use of {} to create an inner level of scope might seem unusual it is only the general case of for(){}, if(){}, while(){} etc…

Pointers in C/C++

Every variable has two components:

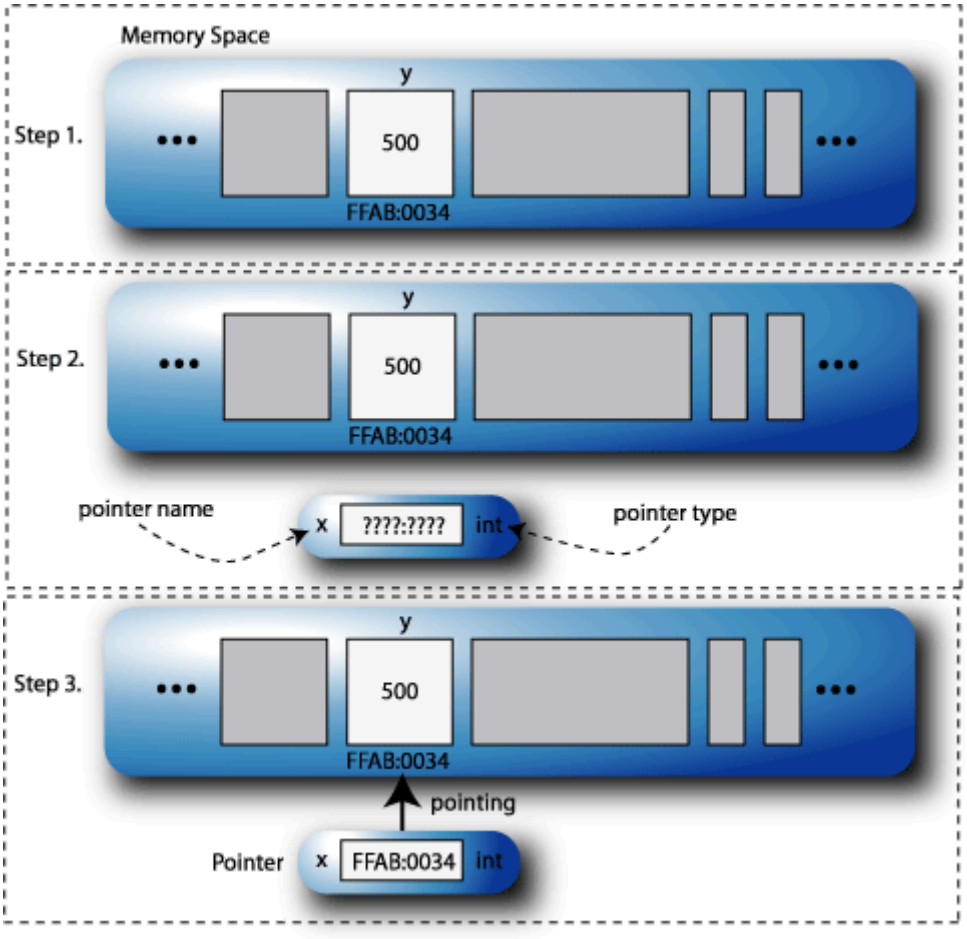
- A value stored in the variable (rvalue), and
- The address of the variable in memory (lvalue).

The & operator returns the **"address of"** the variable. So, if we look at an example piece of code:

```
int y = 500;     // define a variable (step 1)
                 // and initialise it to 500
int *x;          // define the pointer (step 2)
x = &y;          // point it at the address of a variable. (step 3)
```

This example can be illustrated as in Figure 2.7.

Figure 2.7. An example use of pointers.



To find out the value that is "pointed-to" by a pointer x we can use the dereference operator, *x. The "*" can be thought of as the **"value of"** a pointer. So to print out the value of x in this example we could use: cout << "The value of x = " << *x << endl;. In this case we would get an output of The value of x = 500.

Please remember to be aware of the precedence table when using pointers in C++, the section called "Precedence Reference:".This table specifies the correct order in which to use operators, so for example to increase the value at pointer x by 1, you might have used: *x++; and this would be wrong. If you look at the table you will see that ++ comes before * (the dereference *) in the precedence table (both at level 2). This means that the ++ gets applied to the x before the dereference *, increasing the value of the pointer by 1 and then uselessly exposing the value of x. If you change the code to (*x)++; it will work as expected, incrementing the dereferenced x; I would consider it good practice to use as many () as possible to avoid people having to "learn-off" the precedence table.

So there are several operations that we can carry out with the use of pointers:

```
1
2 // Pointer Example
3
```

```
 4 #include<iostream>
 5 using namespace std;
 6
 7 int main()
 8 {
 9    int x[5] = {1,2,3,4,5}; //1
10    int *q, *p = &x[0]; //2
11
12    //increment all values in the array by 1
13    q = p; //3
14    for (int i=0; i<5; i++)
15    {
16       (*q++)++; //4
17    }
18
19    //reset q pointer again
20    q = p; //5
21    for (int i=0; i<5; i++)
22    {
23      (*(q+i))++; //6
24    }
25
26    //do I need to reset q this time? no!
27
28    for (int i=0; i<5; i++)
29    {
30     cout << "x[" << i << "] = " << x[i] <<
31           " at address " << &x[i] <<
32                " and the value of p is " << *(p+i) <<
33           " at address " << p+i << endl; //7
34    }
35 }
36
```

The source code for this is in **PointerExample.cpp**

❶ The array of int x is defined with 5 elements and defined initial values of 1 to 5. i.e. x[0]=1, x[1]=2 etc.

❷ The two pointers p and q are defined using the * notation. The p pointer is initialised to point at the address of the first value in the array, i.e.x[0].

❸ The q pointer is set to point to the same address as the p pointer, i.e. x[0].

❹ For this point it is important to keep in mind the C++ precedence table (the section called "Precedence Reference:"). There is a double increment going on at this stage. The pointer address is being incremented at the same time as the value at the pointer address, but **before** this happens, the increment outside the brackets, i.e. (..)++ causes the value inside the brackets, which is the dereferenced q, i.e. *q, to be incremented.

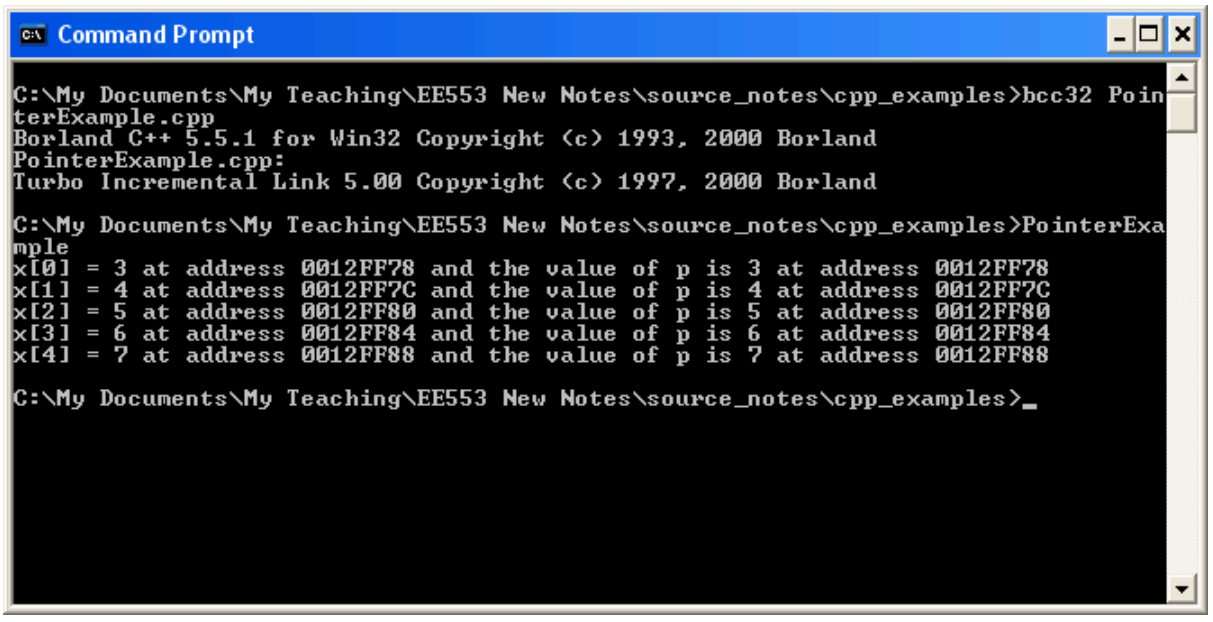
❺ The effect of the previous loop is to move the q pointer from pointing to the first element in the array to pointing to the element after the end of the array. Step 5 resets the q pointer address back to the same address as the pointer p so that it once again points to the first element in the array.

❻ For this point it is once again important to keep in mind the C++ precedence table (the section called "Precedence Reference:"). This loop once again increments even element in the array by 1, but it does it by keeping the q pointer pointing at the first element and offsetting the address, incrementing the value at that address.

❼ This outputs the values of the array and the values at the addresses of p, showing that all the values are the same.

When run, the output of this application can be seen in Figure 2.8.

Figure 2.8. The output from an example use of pointers with arrays.



This example can be further illustrated in Figure 2.9 and Figure 2.10.

Figure 2.9. The pointer example in operation, steps 1 to 4 as in the code sample above.

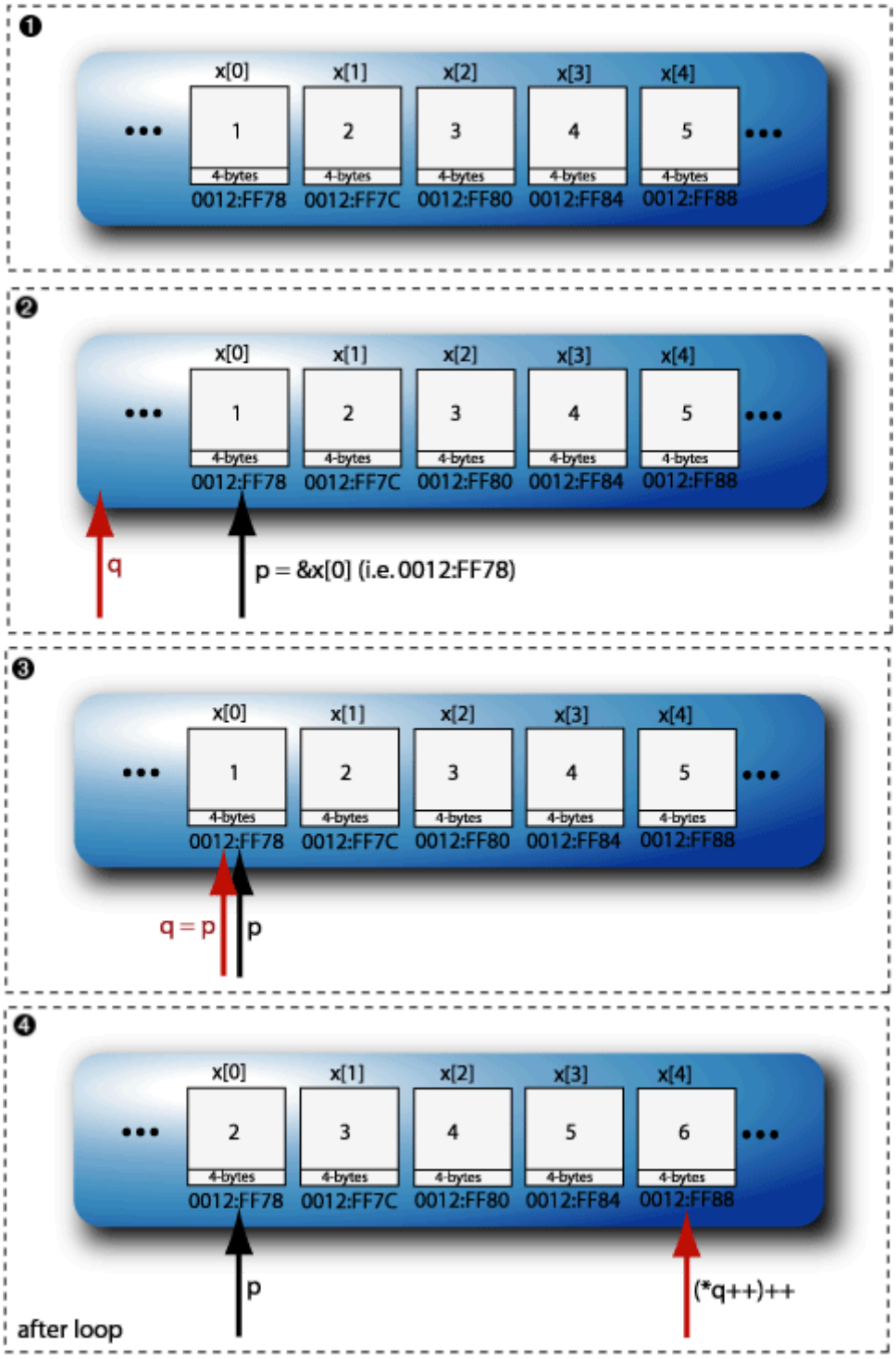
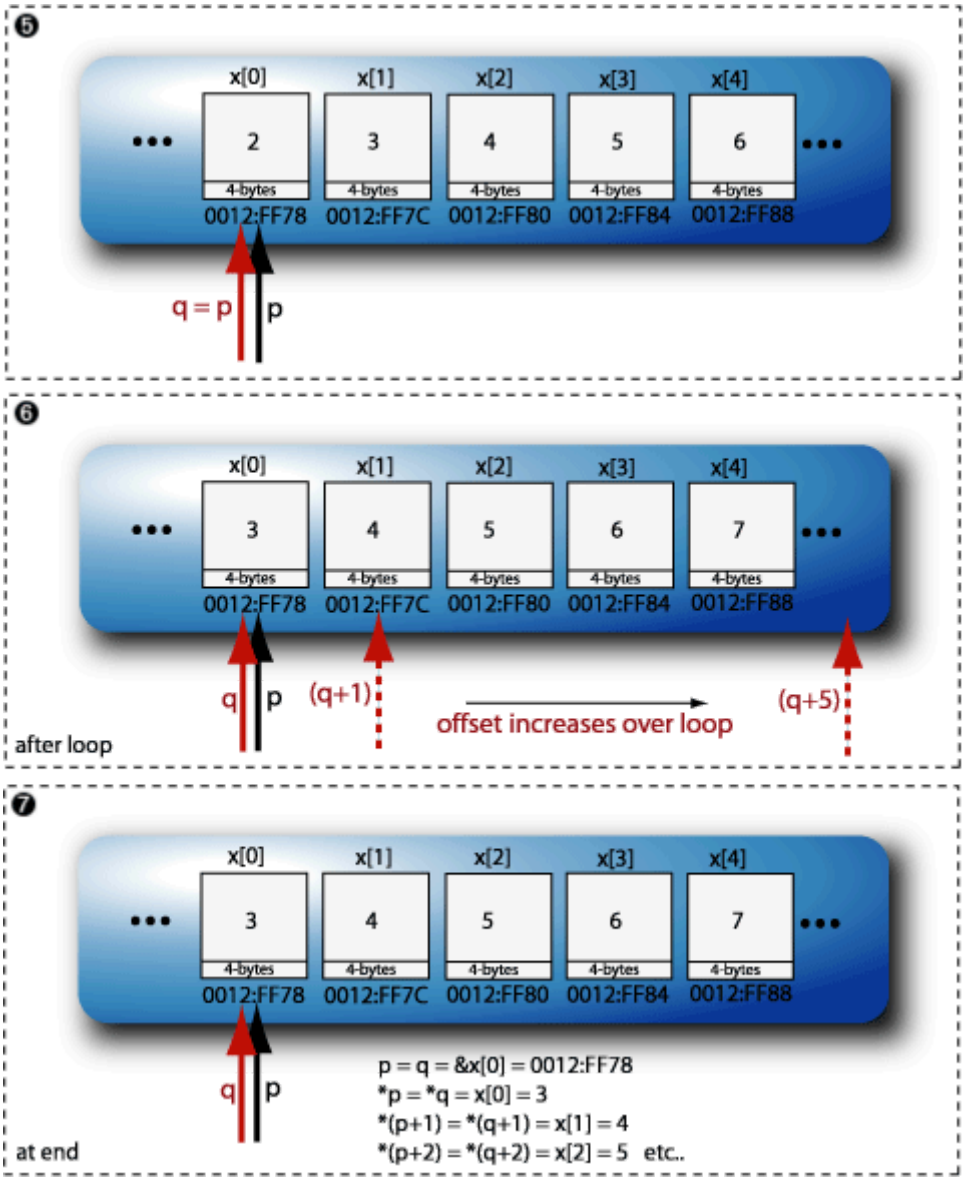
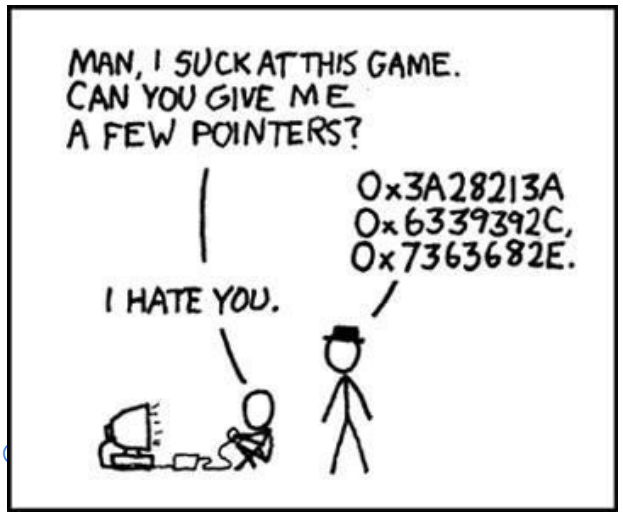


Figure 2.10. The pointer example in operation, steps 5 to 7 as in the code sample above.



Why does a pointer require a type? When we call *(x+1) (the value at the pointer plus one position) the amount of bytes travelled to increase the pointer position by 1 will depend on the data type of x, so if x was of the type int then the "true" memory pointer would travel 4 bytes, whereas if x was of the type double then the "true" memory pointer would travel 8 bytes.



In C and C++ we can convert a variable of one type into another type. This is called casting and we cast using the cast operator () to the left of the data value. When we convert an int into a float the compiler inserts invisible code to do this conversion and we do not have to deal with casts - this is called implicit casting. However, in the situation where for example there is a loss of resolution (eg.

a float to an int, e.g., int x = (int) 200.6;) then an explicit cast is required. Serious difficulties can occur with 'C' style casts in C++ as in certain cases a pointer could be made to consider assigned to a value occupying a larger amount of memory than it actually is. This can damage data surrounding this value if we attempt to change it. We will examine new C++ explicit casts in the next section.

The void* pointer

If you state that a pointer is void in C++ it means that it can point to the address of any type of variable. This feature of the language should probably be avoided unless it is completely necessary as it allows one type to be treated as another type.

```
1
2  // void pointer example
3  int main()
4  {
5     int a = 5;
6     void* p = &a;
7
8     // *p = 6; would be a compile time error. We must cast back
9     // to an int pointer before dereferencing, e.g.
10    *((int *) p) = 6;
11 }
12
```

The void pointer type cannot be dereferenced. In this example (int *) p is the statement that casts p into an int pointer. However, we could just as easily have cast it to any other pointer type, e.g. a float pointer, in which case modifying such a pointer could easily crash the program. void pointers are not used in the general language, but we will have a good use for them later.

A C++ pointer can be assigned to null. This is a special value in the language, often zero, that signifies that the pointer is not pointing at any value. This is not to be confused with a pointer pointing to nothing in particular, i.e. an uninitialised pointer. Comparing two pointers, where one is uninitialised, could result in a positive match by pure chance as the value pointed to by an uninitialised pointer could have any value. This will not happen with a null pointer. We can set a pointer p to null by using the statement p = NULL;. NULL is defined in the standard C header stdlib. This can be included correctly as below:

```
// include the C stdlib.h header file
 #include<cstdlib>

 int main() {
    int *p = NULL;
 }
```

C++ Expressions

Some general points about the C++ language:

C++ variable names

- can begin with a letter or _
- can be followed by letters and digits
- are case sensitive
- cannot use any of the standard language keywords (new, class etc.)

Character Constants

Table 2.3. Character Constants Table

newline \n	horizontal tab \t	vertical tab \v
backspace \b	carriage return\r	form feed \f
bell \a	backslash \\	question mark \?
single quote \'	double quote \"	null character \0

Operators

- Multiplicative operators: a*b a/b a%b (remainder after division, e.g. 10 modulo 4 = 2)
- Additive operators: a+b a-b
- Equality operators (boolean return): a==b (is equal?) a!=b (is not equal?)
- Relational operators (boolean) a<b a<=b a>b a>=b
- Logical operators a==b && a==c (AND) a==c||a<b (OR). Some compilers accept the and, or and not keywords.
- Bitwise operators << (left shift) >> (right shift) [7] & (bitwise AND) ^ (bitwise XOR) | (bitwise OR) ~ (bit complement)
- Increment and decrement a++ a-- ++a --a
- Unary operators -a (negative of a) +i (i) ! (negative logical values) !(a==b) same as (a!=b)

Sizeof operator sizeof(type) (number of bytes of type is returned) sizeof expression (number of bytes of expression returned) So for example:

```
int p[5] = {10, 20, 30, 40, 50};
cout << "Size of int is " << sizeof(int) << " bytes" << endl;
cout << "Size of the array p is " << sizeof p << " bytes" << endl;
```

Will result in:

```
Size of int is 4 bytes
Size of the array p is 20 bytes
```

Conditional operator ?. It has the form conditionalExpression ? trueExpression1 : falseExpression2, for example:

```
// Display the shorter of s and t
cout << (s.length() < t.length() ? s : t);
```

Note that the expressions must have the same type.

Note precedence: a + b * c is the same as a + (b * c) - so be careful and always use ()!

Remember to be careful when comparing values to use == instead of =. If you write if (x=0) then this is an assignment, not an equality test, and the effect will be to assign 0 to x. How does it evaluate the expression to true or false? If the value assigned is zero, it evaluates it as false; if it is anything else, it evaluates it as true.

Control Statements:

if structure (See Figure 2.11):

```
if (expression)
  {statement}
```

```
if (x == y)
   { x = x + 5; }
```

if else structure (See Figure 2.12):

```
if (expression)
   {if true statement}
else
   {if false statement}
```

```
if (x>5)
   { x++; }
else
   { x = x + 5; }
```

while structure (See Figure 2.13):

```
while (expression)
  {statement}
```

```
while(x<3)
   { x++; }
```

do while structure (See Figure 2.14):

```
do
  {statement}
while (expression)
```

```
do
   { x++; }
while (x<3)
```

for structure (See Figure 2.15):

```
for (init, comparison, modifier)
  {statement}
```

```
for (i=0; i<10; i++)
   { someFunct(); }
```

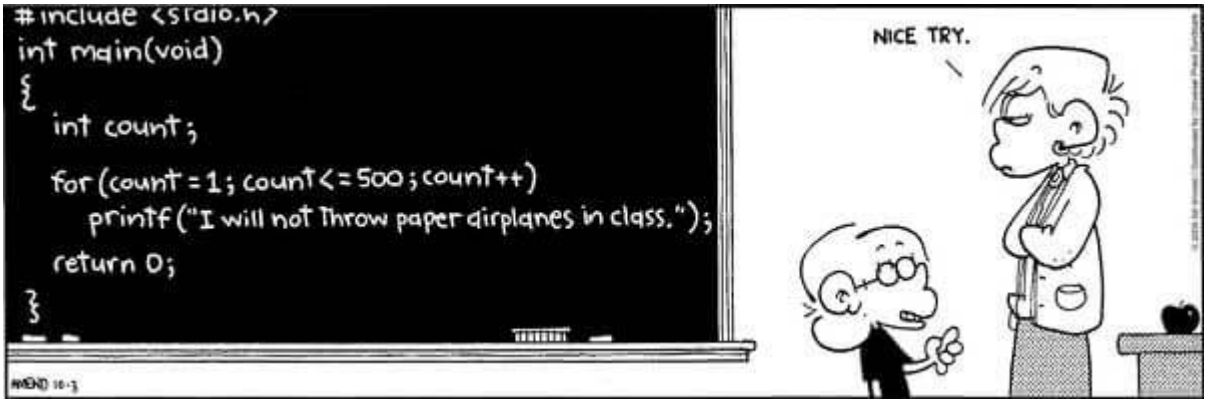
switch structure (See Figure 2.16):

```
switch (int)
  {
```

```
int x = 5;
switch(x)
{
```

```
    case (expression): statement
    case (expression): statement
    default : (expression)
}
```

```
case 1: cout << "test1";
        break;
case 5: cout << "test5";
        break;
default: cout << "none";
}
```



Within all of these control statements we can also control the flow using break and continue keywords. break quits out of the loop/switch, without completing the remaining statements in the loop/switch. continue on the other hand continues directly to the next iteration of loop without executing the remaining statements in the loop. The use of these keywords is frowned upon to some extent in programming as loops must be constructed so that it is safe to skip the remaining statements. For example if the first line of a loop opened a database connection, and the last line closed that connection, a break in the middle of the loop would result in a "lost" open database connection. Below is a short example to show the use of break and continue.

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

int main()
{
    for(int i=0; i<10; ++i)
    {
        if (i==5) { continue; }
        if (i==7) { break; }
        cout <<"Loop number:" << i << endl;
    }
}
```

The source code for this is in **BreakContinue.cpp**. The output of this application is:

```
Loop Number:0
Loop Number:1
Loop Number:2
Loop Number:3
Loop Number:4
Loop Number:6
```

When i==5 the cout will be skipped and the loop will continue to the next iteration, but when i==7 to loop will exit. The most likely place to see a break statement is in some form of infinite loop, such as while(true){}, so that there is some exit point, as there is no condition to evaluate as false.

I suppose in this discussion I should also begrudgingly mention goto. In 99% of the times you consider the use of goto there is an alternative solution. The goto keyword was added to the C++ language because it was present in C (and was of appeal to Basic programmers, where it was necessary). The use of goto makes programs difficult to follow and difficult to debug, but can be used correctly in limited circumstances. Remember that example above of the break quitting the loop before the database connection is closed; Well one possible solution to that problem is to use goto. For example we could use:

```
int main()
 {
   recordsExist = true;
   while ( recordsExist )
   {
     // open database connection
     // retrieve record

       if ( record invalid )
       {
         recordsExist = false;
         goto endloop;  // skip remaining statements in loop
       }

       // more statements that operate on record
     endloop:
       // close database connection
     }
 }
```

> "The fact that 'goto' can do anything is exactly why we don't use it."
>
> **— Bjarne Stroustrup**

Now, I am certain that there is another solution to this issue (such as a single else), but this is just an example of when you **might** use a goto. Distant gotocalls are not acceptable as it would be impossible to follow the code, locating the destination in a large section of code.

Figure 2.11. The if structure.

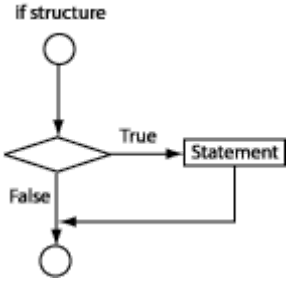


Figure 2.12. The if else structure.

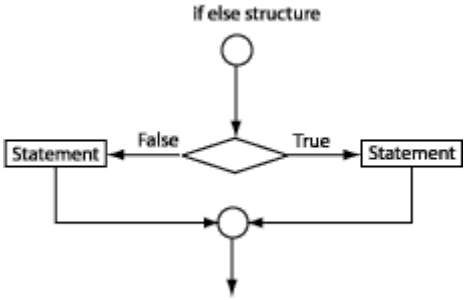


Figure 2.13. The while structure.

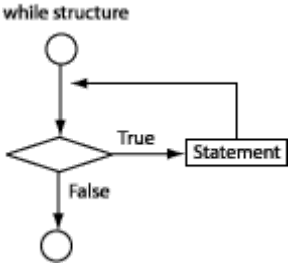


Figure 2.14. The do while structure.

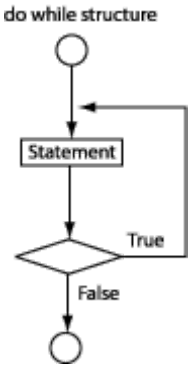


Figure 2.15. The for structure.

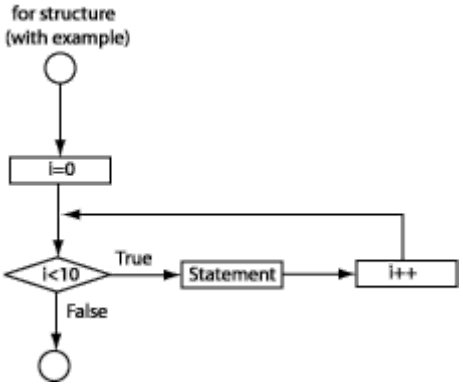
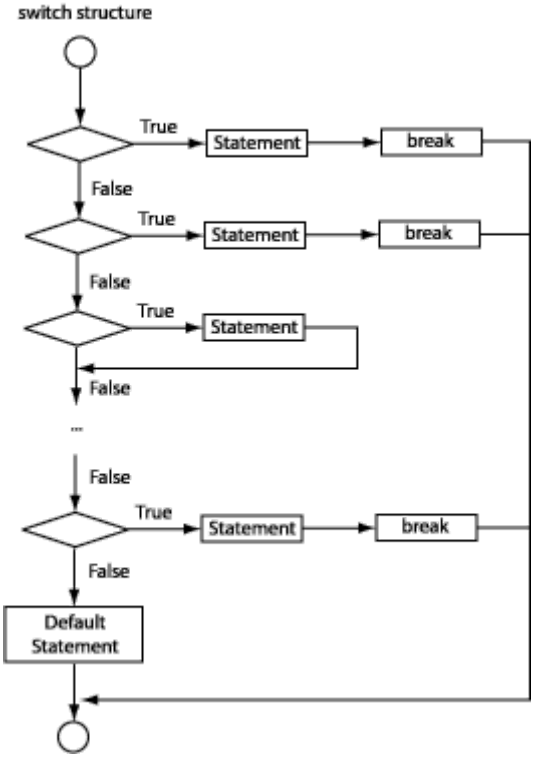


Figure 2.16. The switch structure.



Precedence Reference:

Table 2.4. C++ Operator Precedence Table

Precedence	Operator	Description	Example	Associativity
1	()	Grouping operator	(3+a)*2;	left to right
1	[]	Array Access	a[0]=10;	left to right
1	->	Pointer Member Access	p->balance=0;	left to right
1	.	Object Member Access	account.balance=0;	left to right

1	::	Scoping operator	::value = 1;	left to right
1	++	Post Increment operator	x++;	left to right
1	--	Post Decrement operator	x--;	left to right
2	!	Logical Negation	if(!running)	right to left
2	~	Bitwise Complement	x=~x;	right to left
2	++	Pre Increment operator	++x;	right to left
2	--	Pre Decrement operator	--x;	right to left
2	-	Unary minus	a=-b;	right to left
2	+	Unary plus	a=+b;	right to left
2	*	Dereference	a=*ptr;	right to left
2	&	Address of	ptr=&a[0];	right to left
2	(type)	Cast to a type	x=(int)23.6;	right to left
2	sizeof	Size in bytes	x=sizeof(X);	right to left
3	->*	Member Pointer Selector	ptr->*p=1;	left to right
3	.*	Member Pointer Selector	account.*p=1;	left to right
4	*	Multiplication	a=b*c;	left to right
4	/	Division	a=b/c;	left to right
4	%	Modulus	remainder=b%c;	left to right
5	+	Addition	a=b+c;	left to right
5	-	Subtraction	a=b-c;	left to right
6	<<	Bitwise Right Shift	a=b<<1;	left to right
6	>>	Bitwise Left Shift	a=b>>1;	left to right
7	<	Less than comparison	if(a<b)	left to right
7	<=	Less than Equals comparison	if(a<=b)	left to right
7	>	Greater than comparison	if(a>b)	left to right
7	>=	Greater than Equals comparison	if(a>=b)	left to right
8	==	Equals To	if(a==b)	left to right
8	!=	Not Equals To	if(a!=b)	left to right
9	&	Bitwise AND	a=a&1	left to right
10	^	Bitwise Exclusive OR	a=a^1	left to right
11	|	Bitwise OR	a=a|1	left to right
12	&&	Logical AND	if(a&&b)	left to right
13	||	Logical OR	if(a||b)	left to right
14	?:	Ternary Condition	x=(a<b)?a:b;	right to left
15	=	Assignment Operator	x=5;	right to left
15	+=	Increment and Assign	x+=5;	right to left

15	-=	Decrement and Assign	x-=5;	right to left
15	*=	Multiply and Assign	x*=5;	right to left
15	/=	Divide and Assign	x/=5;	right to left
15	%=	Modulo and Assign	x%=2;	right to left
15	&=	Bitwise AND and Assign	x&=1;	right to left
15	^=	Bitwise XOR and Assign	x^=1;	right to left
15	|=	Bitwise OR and Assign	x|=1;	right to left
15	<<=	Bitwise Shift Left and Assign	x<<=2;	right to left
15	>>=	Bitwise Shift Right and Assign	x>>=2;	right to left
16	,	Sequential Evaluation Operator	for (int i=0, i<10, i++, j++)	left to right

---

[7] The compiler can differentiate this operator from the cout << operator based on the types used.

Chapter 3 - C++ and Object-oriented Programming

> "The problem with using C++ ... is that there's already a strong tendency in the language to require you to know everything before you can do anything."
>
> **-- Larry Wall (Creator of the Perl programming language)**

Classes in C++

A C++ Class contains:

- An **interface** that allows outside user interaction with the class.
- **States** that store the data within the class.
- An **implementation** that provides the actual code implementation of the interface methods, and any other internal workings.

A sample C++ class can be outlined as:

```
// An Example Class

class AnExampleClass { //1
  // state definitions //2
  public: //3
  // interface declarations //4
  private:
  // implementation method declarations //5
}; //6

// member method implementation //7
```

❶ The class keyword lets the compiler know that a class is about to be defined. The class name follows and should begin with a capital letter.

❷ The states are the data values of the class and are usually defined first. Note that variables are defined (not declared). **Definition** means "make this variable or method" here, allocating storage for the name.

❸ The public keyword states that all methods to follow are part of the interface, i.e. publicly visible.

❹ All methods following the public keyword are part of the interface. i.e. the part of the class that you wish to allow users to see outside of the class. Note that the interface methods are declared. **Declaration** is the introduction of a name (identifier) to the compiler. It notifies the compiler that a method or variable of that name has a certain form, and exists somewhere.

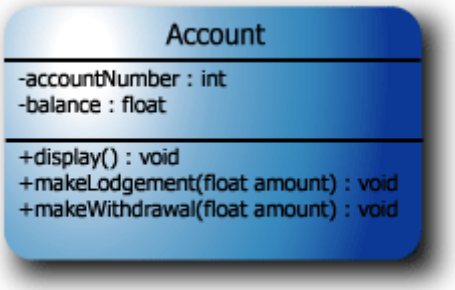
❺ All methods following the private keyword are internal methods of the class and are part of the internal implementation.

❻ The semicolon defines the end of the class declaration. If you leave it out your code will not compile, but worse, the error messages you receive will not make it clear that you have forgotten this semi-colon.

❼ Once the class is defined you can then write the implementation code for the methods. This keeps the class definition short and easy to read. If you wish, these methods can even be written in a separate file.

Here is an example C++ class of a Bank Account class as shown in Figure 3.1, "A Sample Bank Account Class", with private notated with "-" and public notated with "+".

Figure 3.1. A Sample Bank Account Class




```
1
2   // Basic Bank Account Example
3
4   #include<iostream>
5
6   using namespace std;
7
8   class Account {
9
10  private:
11
12    int accountNumber;
13    float balance;
14
15  public:
16
17    virtual void display();
18    virtual void makeLodgement(float);
19    virtual void makeWithdrawal(float);
20  };
21
22  void Account::display(){
```

```
23      cout << "account number: " << accountNumber
24        << " has balance: " << balance << " Euro" << endl;
25    }
26
27    void Account::makeLodgement(float amount){
28      balance = balance + amount;
29    }
30
31    void Account::makeWithdrawal(float amount){
32      balance = balance - amount;
33    }
34
35    int main()
36    {
37      Account a;
38      a.display();  //will output rubbish for the states
39    }
40
```

The source code for this is in **BasicAccount.cpp** The Interface:

- Provides a contract between the users of the class and the implementer.
- The user of a class/object can only manipulate the states via the interface.
- The user cannot modify the state directly.
- The implementor undertakes to provide all the functionality of the interface.
- If the interface must be changed then a new contract must be negotiated with the user. This will cause the user a lot of unnecessary work.

The code above shows how we can define and implement a class. If we wish to use this class, we can create an object of the class and manipulate it directly, for example:

```
int main() {
   Account myAccount; //1

   cout << "Account Details:";
   myAccount.display(); //2

   myAccount.makeLodgement(2300.00); //3
   cout << "Account Details:";
   myAccount.display();

   myAccount.balance = 2300.00; //4
   // ERROR!

   cout << "Account Balance:"
     << myAccount.balance << endl; //5
   // ERROR!
 }
```

This call creates an object of the Account class called myAccount. The balance and account number are not yet set.

❷ The display() method is called directly on the myAccount object. It will display the account details.

❸ The makeLodgement() method is called directly on the myAccount object. It will add 2300 Euro to the balance.

❹ This call is **not allowed**. The balance state is a private state of the Account class, so you cannot modify it directly. It is not part of the interface.

❺ This call is **not allowed**. The balance state is not part of the interface and does not even allow the value to be read.

If you wish to assign a new object to an existing reference, even after initialisation use:

```
int main() {
    Account myAccount; // assigns an object to the reference
    myAccount = Account();  // allows the assignment of a new account object
}
```

Constructors

Constructors allow the initialisation of the state of an object when it is created. Suppose we were to initialise the states of an object using the following format:

```
// Is this OK?
 class Account{
   int myAccountNumber = 242343;
   float myBalance = 0.00;

   public:
     //etc..
 };
```

No this is not sufficient!:

- Even if we were allowed to do this, we cannot leave every instance of this class with the same bank account number and balance.
- We need a way to update the account number and initial balance when the object is being created.

A Constructor can be used for this task:

- It is a member method that has the exact same method name as the class name.
- A constructor must not have a declared return type (not even void).
- A constructor cannot be virtual (to be discussed later).

So using constructors with the Account class:

```
1
2   // Basic Bank Account Example with Constructors
3
4   #include<iostream>
5
6   using namespace std;
7
8   class Account{
9
10    int accountNumber;
11    float balance;
12
13  public:
14
15    Account(float, int); //1
16    virtual void display();
17    virtual void makeLodgement(float);
18    virtual void makeWithdrawal(float);
19  };
20
21 Account::Account(float aBalance, int anAccNumber) //2
22  {
23    accountNumber = anAccNumber;
24    balance = aBalance;
25 }
26
27  void Account::display()
28  {
29    cout << "account number: " << accountNumber
30      << " has balance: " << balance << " Euro" << endl;
31  }
32
33  void Account::makeLodgement(float amount)
34  {
35    balance = balance + amount;
36  }
37
38  void Account::makeWithdrawal(float amount)
39  {
40    balance = balance - amount;
41  }
42
43  int main()
44  {
45    Account anAccount = Account(35.00, 1234); //OK 3
```

```
46      Account testAccount(0.0, 1235);          //OK 4
47      //Account myAccount = Account();         //Wrong! 5
48
49      anAccount.display();
50      testAccount.display();
51    }
52
```

The source code for this is in **BasicAccount2.cpp**

❶ The constructor definition. Note it has the exact same name as the class name and has no return type.

❷ This is the implementation of the constructor. In this case it simply sets the states of the class to the values passed. Once a constructor is supplied it must be used. Every class has an implicit constructor (with no parameters), but once a new constructor is defined the implicit constructor is no longer available.

❸ The anAccount object is created with an account number of 34234324 and a balance of 35.00.

❹ This constructor call is exactly the same as the previous call only with a different notation, so a testAccount object is created with an account number of 34234325 and a balance of 0.00.

❺ This constructor call is not valid as the implicit constructor is no longer available (the one with no parameters) as a new non-default constructor has been defined.

Constructors and Member Initialisation Lists

Instead of the notation used above to set the states of the object, we can also use a member initialisation list, that sets the states within the constructor definition. So, if we use member initialisation lists with the previous constructor, it would look like:

```
// the constructor code implementation
 Account::Account(float aBalance, int anAccountNumber) : //1
     accountNumber(anAccountNumber), balance (aBalance)  //2
 {
      // anything else, place here!
 }
```

❶ The : denotes the use of the Member Initialisation List.

❷ The accountNumber(anAccountNumber) call is the same as using the statement accountNumber = anAccountNumber;

This format may seem complex, but if the class contains an object that must be initialised in the constructor (i.e., IS A PART OF) then the member initialisation list must be used.

Classes and Pointers

There is very little difference between pointers to variables as discussed in the section called "Pointers in C/C++" and pointers to objects. First off, we can define a pointer p to objects of a certain class type using:

```
Account anAccount(5.0, 12345), *p;
```

So, pointer p is a pointer to objects of the Account class. Since the pointer is still just the storage of a memory address there is no need to attempt any type of construction call in the creation of the pointer. When the pointer is first created, it is not pointing to an object of the Account class, nor is an object of the Account class created. The pointer simply points at nothing (well nothing of consequence).

If we wish to make the pointer p point to an object of the Account class, we could use a statement like:

```
Account anAccount(5.0, 12345), *p;
p = &anAccount;

// alternative notation would be (just like pointers to variables):
Account anAccount(5.0, 12345);
Account *p = &anAccount;
```

Previously, several methods were defined for the Account class that allowed operations to be applied to the account object, such as display(), makeLodgement and makeWithdrawal(). How are these methods used when dealing with pointers?

In the same way that we applied the **.** operator to objects we can apply the **->** operator to pointers to objects. The notation is simply a - followed by a >

So for example, to demonstrate the object notation and the pointer notation:

```
//object notation
Account anAccount = Account(35.00, 12344);
anAccount.makeLodgement(20.0);
anAccount.display();

//pointer notation
Account testAccount = Account(5.0, 12345);
Account *testPtr = &testAccount;
testPtr->makeLodgement(20.0);
testPtr->display();
```

A source code example for this segment is listed in **BasicAccountWithPointers.cpp**.

The output from this example would be:

```
account number: 12344 has balance: 55 Euro
account number: 12345 has balance: 25 Euro
```

Note that testPtr->myBalance is exactly the same as (*testPtr).myBalance, but the -> notation is a lot easier to comprehend when combined with other operators.

Class Hierarchy

> "In C we had to code our own bugs. In C++ we can inherit them."
>
> --Unknown, 1991

In an object-oriented language:

- The compiler must understand the class hierarchy.
- If we extend a class, the compiler must combine the extension with the definition of the base class.

In a non-object-oriented language:

- The user must rely on the programmer documentation.
- The user must define each class from scratch.

Task: Develop a new CurrentAccount class that extends the functionality of the Account class.

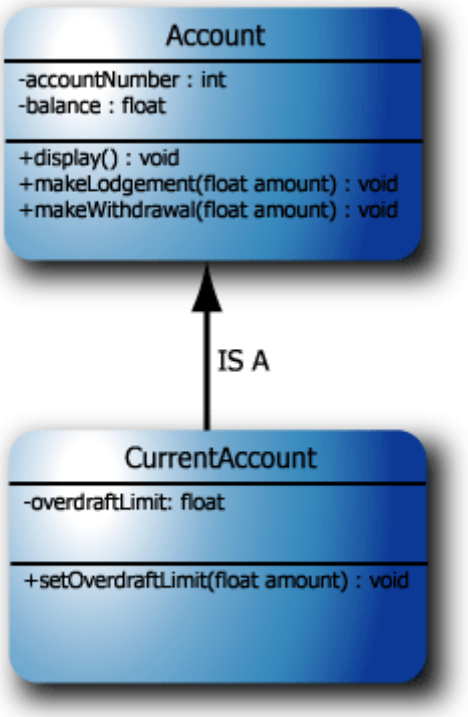
Solution: Define a new class CurrentAccount that inherits the functionality of account, so: Account is the base class of CurrentAccount and equivalently CurrentAccount is the derived class of Account (i.e. CurrentAccount IS A Account).

What additional functionality should the CurrentAccout class have? Well, a current account is usually a non-interest earning account that has overdraft facilities, so we could add:

- An overdraft limit.
- A facility to set this limit.

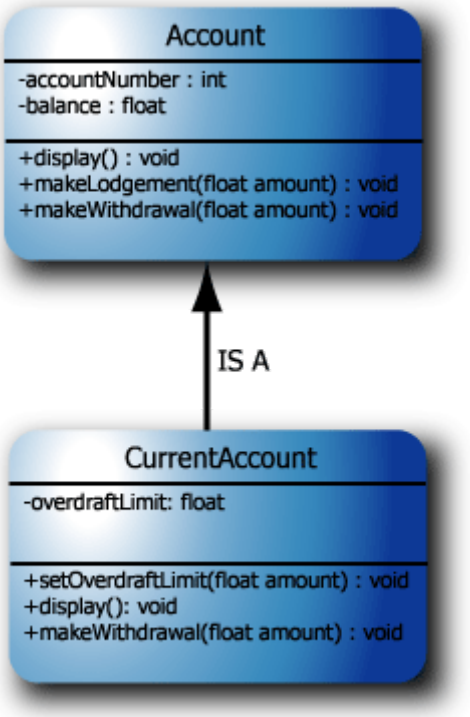
So, this can be visualised as Figure 3.2, "The Account and CurrentAccount class relationship." where an overdraftLimit state and a setOverdraft() method have been added to the CurrentAccount class description. The class also inherits the states and methods of the Account class, i.e. balance, accountNumber, display(), makeLodgement() and makeWithdrawal().

Figure 3.2. The Account and CurrentAccount class relationship.



So, inheritance allows a class to be inherited and extended with new functionality. A problem still remains in that the display() method that we have inherited does not work sufficiently for the CurrentAccount class. The display() method only displays the balance and accountNumber state details - it does not display the overdraftLimit. So we need to replace the behaviour (or override the behaviour) of the display() method with an updated method that also displays the overdraftLimit details. So to override the display method, it can simply be re-defined in the CurrentAccount class. This discussion is also relevant for the makeWithdrawal() method, as it will have to behave slightly differently when a negative balance is reached. This inheritance structure is illustrated as in Figure 3.3, "The extended Account and CurrentAccount class relationship.".

Figure 3.3. The extended Account and CurrentAccount class relationship.



So, inheritance allows a derived class to extend its parent with newly inherited functionality, and also allows functionality to be modified.

The definition of this new CurrentAccount is as follows:

```
class CurrentAccount: public Account
 {
     float overdraftLimit;

   public:

     CurrentAccount(float bal, int actNum, float limit);
     virtual void setOverDraftLimit(float newLimit);
     virtual void display();
     virtual void makeWithdrawal(float amount);
 };
```

A full source code example for this is in **CurrentAccount.cpp** The implementation/definition of these methods is as follows.

```
CurrentAccount::CurrentAccount(float bal, int actNum, float limit):
     Account(bal, actNum), overdraftLimit(limit)
     {}

 void CurrentAccount::display() {
   cout << "Account number: " << accountNumber
        << " has balance: " << balance << " Euro" << endl;
   cout << "  And overdraft limit: " << overdraftLimit << endl;
```

```
}

void CurrentAccount::makeWithdrawal(float amount) {
  if (amount < (balance + overdraftLimit))
  {
    balance = balance - amount;
  }
}

void CurrentAccount::setOverDraftLimit(float limit) {
  overdraftLimit = limit;
}
```

Static and Dynamic Types

The **static type** is the type of an object variable as declared at compile time and the **dynamic type** is the type of an object variable determined at run-time. In most cases the static and dynamic types of an object variable are the same, however, they can differ when we are using pointers to objects. An example of this capability is demonstrated below:

```
int main() {
   Account a = Account(35.00, 34234324);
   Account *ptrA = &a; //1

   CurrentAccount b = CurrentAccount(50.0, 12345, 200.0);
   CurrentAccount *ptrB = &b; //2

   cout << "Displaying ptrA:" << endl;
   ptrA->display(); //3
   cout << "Displaying ptrB:" << endl;
   ptrB->display(); //4

   //ptrB = ptrA;   // not allowed  //5

   ptrA = ptrB; //6
   cout << "Displaying ptrA again:" << endl;
   ptrA->display();  //7
}
```

A full source code example for this is in **CurrentAccount2.cpp**

❶ The Account pointer ptrA is assigned to the address of the Account object a.

❷ The CurrentAccount pointer ptrB is assigned to the address of the CurrentAccount object b.

❸ The display() method is called on the Account object using the ptrA

pointer.

❹ The display() method is called on the CurrentAccount object using the ptrB pointer.

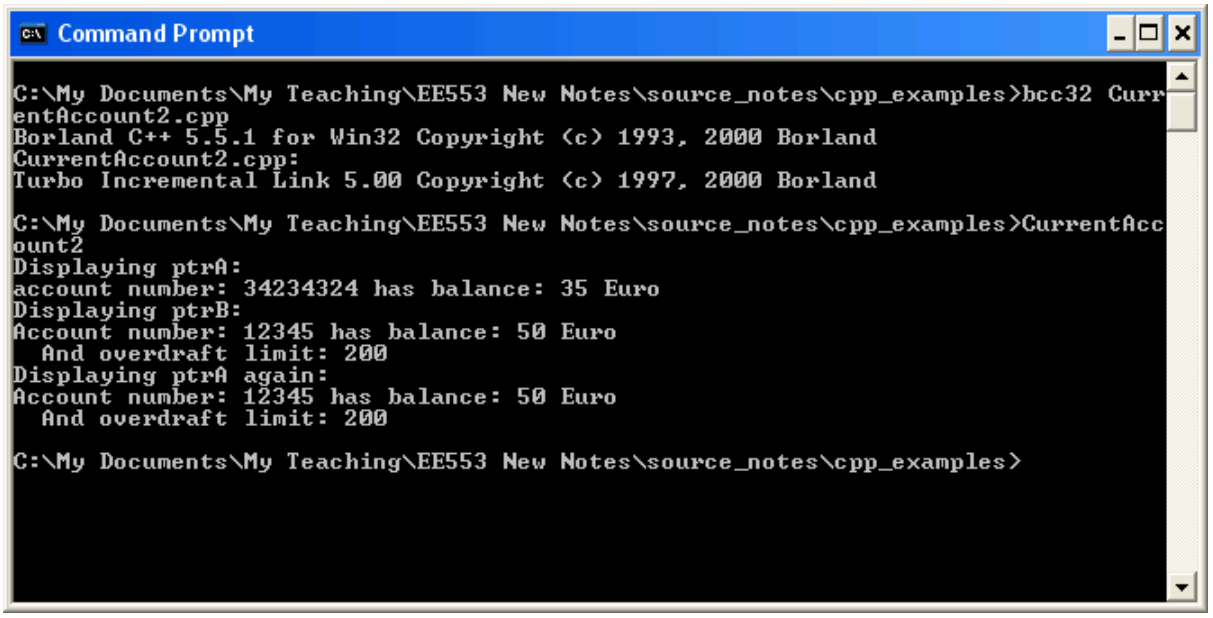
❺ We are not allowed to make the ptrB CurrentAccount pointer point at the Account object.

❻ The ptrA Account pointer is pointed at the CurrentAccount object. This is allowed.

❼ The ptrA Account's display() method is called, which results in the display() method of the CurrentAccount object being called.

The output from this code segment is shown as in Figure 3.4, "The output from the code segment above."

Figure 3.4. The output from the code segment above.



To try to explain this further, see the following figures:

Figure 3.5. The pointers ptrA and ptrB have the same static and dynamic types.

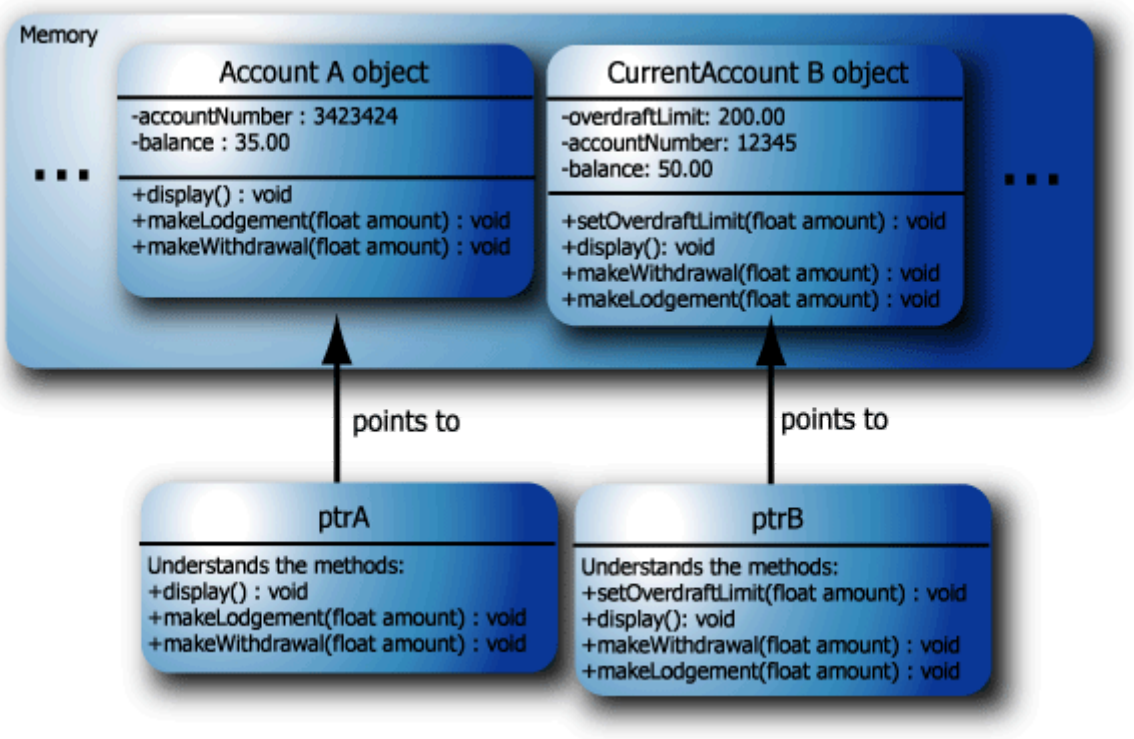


Figure 3.5, "The pointers ptrA and ptrB have the same static and dynamic types." shows the code segment after steps 1 and 2 have taken place. This means that the two objects are created in memory and the two pointers point at their respective objects, so an Account pointer points at the Account object and the CurrentAccount pointer points at the CurrentAccount object.

Figure 3.6. Allowed: The pointer ptrA has static type Account and dynamic type CurrentAccount.

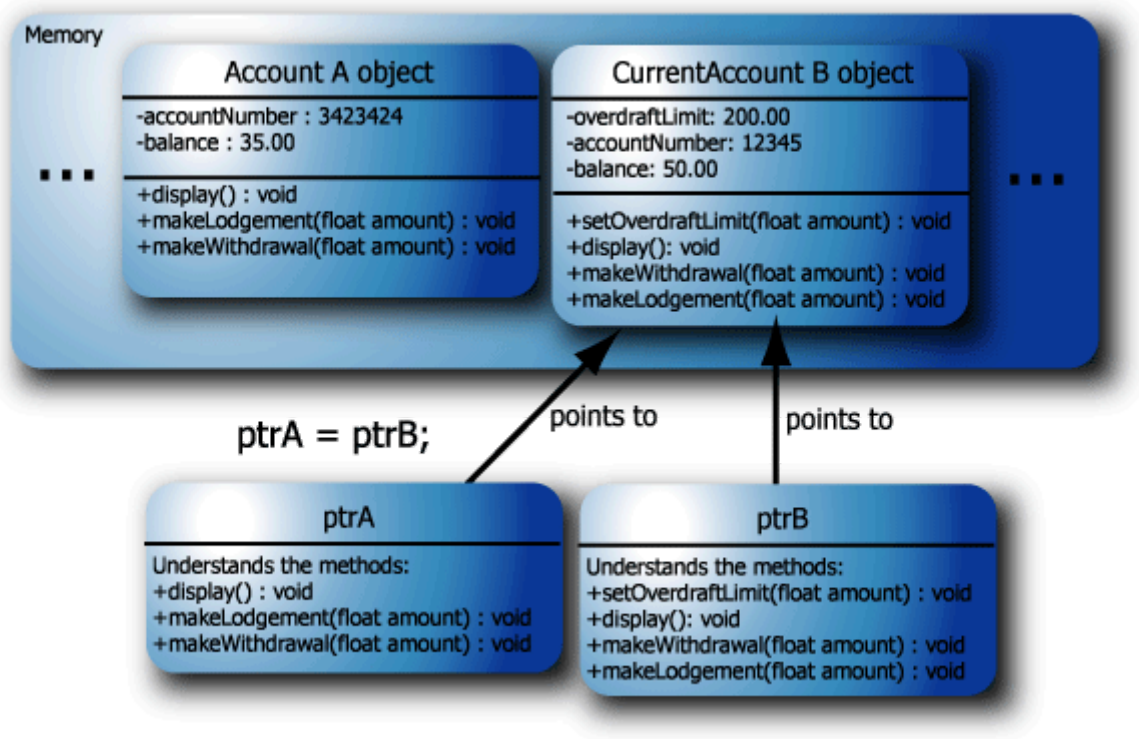


Figure 3.6, "**Allowed:** The pointer ptrA has static type Account and dynamic type CurrentAccount." shows the state of the application after the 6th step takes place. The Account pointer is assigned to the CurrentAccount pointer and this is allowed. To understand this, you can think of the Account pointer as understanding the display(), makeLodgement() and makeWithdrawal() methods. So, we can call any of these methods on the ptrA and the object that we are calling these methods on understands all these methods (and more!). This is guaranteed in this case! why? The CurrentAccountclass is a child of the Account class and so has inherited all the methods from Account. Therefore, calling the methods of the pointer to the Account ptrA works perfectly.

Figure 3.7. Not Allowed: Pointer ptrB has static type CurrentAccount and dynamic type Account.

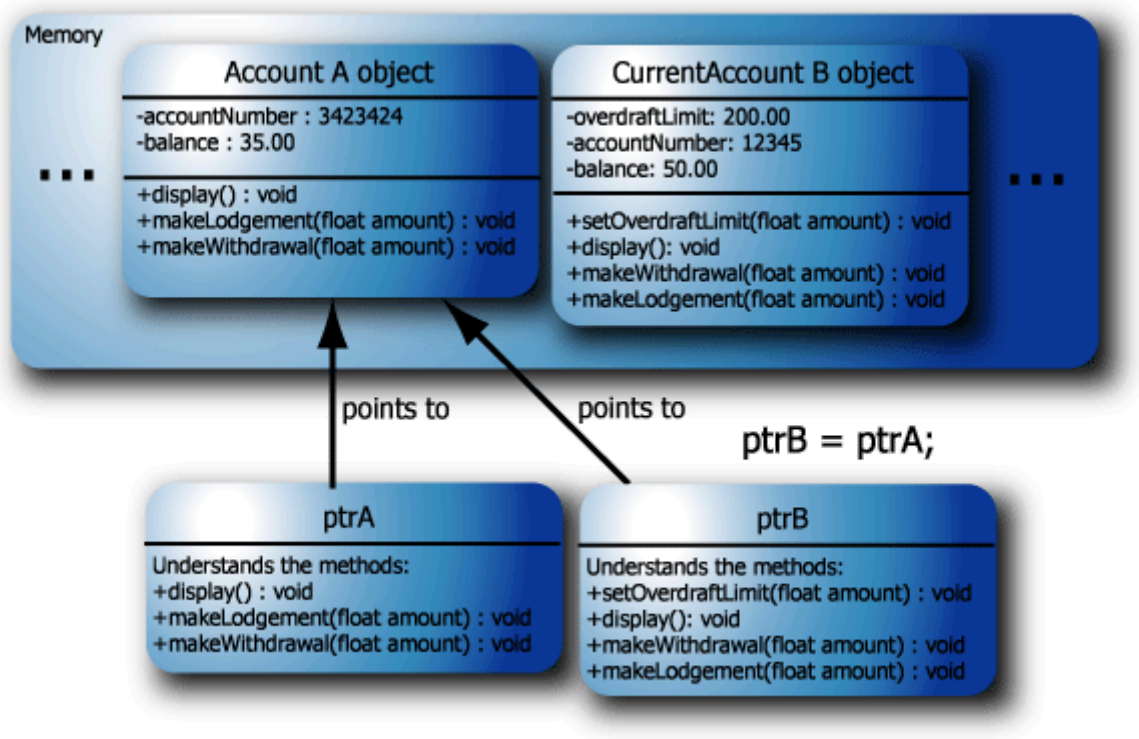


Figure 3.7, "**Not Allowed:** Pointer ptrB has static type CurrentAccount and dynamic type Account." shows the assignment of ptrB = ptrA; which is **not allowed**. If this case was allowed then the CurrentAccount pointer ptrB would be pointing to an Account object. But the CurrentAccountpointer understands the methods display(), makeLodgement(), makeWithdrawal() and setOverdraftLimit(). So what would happen if the setOverdraftLimit() was called on the Account object a? It does not have a setOverdraftLimit() method and so it would fail.

The rule is: **An assignment L=R, is legal only if the static type of R is the same or a derived type of the static type of L.**

One very important point to note in the code segment above is that when the pointer ptrA of static type Account points at the CurrentAccount object and its display() method is called, it is the display() method of the object CurrentAccount is called, not the Account display() method. You can see this in the screen grab of the output, as shown in Figure 3.4, "The output from the code segment above.", where the overdraft limit is displayed the second time ptrA->display() is called. **So the method is called on the dynamic type, not the static type.**

The ability of a variable to change its dynamic type during execution is a useful property of C++, allowing programs to be easier to extend and easier to debug. However, it does have some consequences: Take the last example - The base class Account and the derived class CurrentAccount have different implementations of the display() method. The dynamic type of the object determines which display() method is called. This is called **Dynamic Binding**. With the facility of dynamic binding, CurrentAccount::display() is called. If there was no such facility, Account::display() would always be called.

The notation that I just used for identifying the two display() methods allowed a clear

indication of which display() method was being discussed, either the display() method associated with the Account class or the display() method associated with the CurrentAccount class. The "::" is an actual operator, called the scope resolution operator, and it has very important uses.

For example: The display() method that was developed for the CurrentAccount class displayed the balance, the account number and the overdraft limit. However, the code used to display the balance and the account number is the same as that in the parent class Account::display() method. So code has been replicated - This is unacceptable under the OOP paradigm as later alterations to this segment of code must also be replicated.

The scope resolution operator allow this issue to be resolved, for example, the CurrentAccount class's display() method can be modified to:

```
void CurrentAccount::display()
 {
   Account::display();
   cout << "  And overdraft limit: " << overdraftLimit << endl;
 }
```

Now, the code from the Account's display() method is not being replicated using cut-and-paste, rather it is being called directly. Why is this so important?

Well, this demonstrates that we are able to call the method that we are over-riding (the Account::display() method) from within the new method (CurrentAccount::display()).

- Code does not have to be re-written. Always Good!
- Without this feature - If a later modification needs to be made, for example adding an owner name state, or sort code to the Account class, then all derived class display() methods would have to be updated.
- If a "bug" is found, it need only be fixed once.
- It becomes much easier to manage the code.

In this case, only 2 lines of code have been replicated, but it could have been 100's of lines of code, with 10 different types of derived classes.

Abstract Classes

We discussed abstract classes previously as in the section called “Abstract Classes” in the context of general Object-Oriented Programming. An abstract class in C++ is a class that can have state variables and member methods, however, it provides no implementation for at least one of those member methods, i.e. it is incomplete.

Some C++ compilers expect you to define at least one derived class for each abstract class.

Using the same banking example as before, the use of abstract classes can be demonstrated. Suppose the Account class is modified to add a new abstract method called getAccountType() that returns a string defining the type of account. We can require that every child of the Account class implements this method and we can even

use the method getAccountType() in the Account class in spite of the fact that the method has no implementation. So the Account will look like the following:

```
1
2
3  #include<iostream>
4  #include<string> //1
5
6  using namespace std;
7
8  class Account{
9
10  protected:
11
12     int accountNumber;
13     float balance;
14
15  public:
16
17     Account(float aBalance, int anAccountNumber);
18     virtual string getAccountType() = 0; //2
19     virtual void display();
20     virtual void makeLodgement(float);
21     virtual void makeWithdrawal(float);
22 };
23
24 Account::Account(float aBalance, int anAccNumber)
25 {
26    accountNumber = anAccNumber;
27    balance = aBalance;
28 }
29
30 void Account::display(){
31    cout << "Account type: " << getAccountType() << endl; //3
32    cout << "account number: " << accountNumber
33      << " has balance: " << balance << " Euro" << endl;
34 }
35
36 ...
37
38
```

The full source code for this example is in **AbstractCurrentAccount.cpp**

❶ The "string" header file must be included to allow the use of strings within the application.

❷ The assignment of "= 0" allows the method to be defined as abstract. This means that no implementation will be present for that method, in this case the

getAccountType() method. These are known as **pure virtual functions**.

❸

Even though there is no actual implementation for the getAccountType() method, it can still be used. Since the class has an abstract method, the entire class is abstract and so cannot be instantiated, so no objects can now be created of the Account class. To use this class, a child class **must** exist and it **must** implement the getAccountType() method before an object can be created of that class.

```
1
2
3  class CurrentAccount: public Account
4  {
5       float overdraftLimit;
6
7   public:
8
9       CurrentAccount(float balance, int accountNumber, float limit);
10      virtual string getAccountType(); //1
11      virtual void setOverDraftLimit(float newLimit);
12      virtual void display();
13      virtual void makeWithdrawal(float amount);
14 };
15
16 CurrentAccount::CurrentAccount( float balance, int accountNumber, float
limit):
17      Account(balance, accountNumber), overdraftLimit(limit)
18 {}
19
20 string CurrentAccount::getAccountType()
21   { return "Current Account"; } //2
22
23 void CurrentAccount::display()
24 {
25    Account::display(); //3
26    cout << "  And overdraft limit: " << overdraftLimit << endl;
27 }
28
29 ...
30
31 int main()
32 {
33   //Account a = Account(35.00,34234324);    NOT ALLOWED
34   //Account *ptrA = &a;
35
36   CurrentAccount b = CurrentAccount(50.0, 12345, 200.0);
37   b.display(); //4
```

```
38 }
39
40
```

The full source code for this example is listed in **AbstractCurrentAccount.cpp**

❶ The abstract method getAccountType() must be re-defined in the child class. Note that this time there is no " = 0", stating that there will be an implementation in this class for the defined method.

❷ The implementation is coded. For this class all that occurs is that the string "Current Account" is returned from the method. The implementation for a deposit account would return "Deposit Account" etc.

❸ The Account::display() method is called in the same way as before, so the display() method of the parent is used, which in turn calls the getAccountType() method of the child class.

❹ The call to b.display() calls the display method of CurrentAccount, which in turn calls the display() method of the parent Account class.

C++ Explicit Casts

Casting is one of the most troublesome operations in C/C++ as when you use casts you are directing the compiler to trust you and to forget about type checking. Casts should be used as infrequently as possible and only when there is no alternative. We have two main terms for casting, upcasting and downcasting. Upcasting is casting from a derived class to one of its base classes and downcasting is casting from a base class to one of its derived classes. Remember that the inheritance tree branches down with the base class at the top. We also have the lesser known cross-casting for casting a class to a sibling class.

C++ introduces new explicit casts that identify the rationale behind the cast, clearly identifies that a cast is taking place and confirms type-safe conversions. These casts are:

- static_cast for well-behaved casts such as automatic conversions, narrowing conversions (e.g. a float to int), a conversion from void*, and safe downcasts (moving down the inheritance hierarchy) for non-polymorphic classes. If the base class is a virtual base class then you must use a dynamic_cast
- dynamic_cast is used for type-safe downcasting. The format of a dynamic cast is dynamic_cast <type>(expression) and requires that the expression is a pointer if the type is a pointer type. If a dynamic_cast fails then a null pointer is returned. The dynamic_cast is actually safer than the static_cast as it performs a run-time check, to check for ambiguous casts (in the case of multiple-inheritance).
- const_cast is used for casting away const or volatile.

- reinterpret_cast is the most dangerous cast. It allows us to convert an object into any other object for the purpose of modifying that object - often for low-level "bit-twiddling" as it is known.

Here is an example of all four casts:

```
1
2  // Start reading from main() function to help make it easier to understand
3  #include <iostream>
4  using namespace std;
5
6  // Demo classes with a virtual base class
7  class Account {
8       public:
9              float balance; // for demonstration only!
10             virtual ~Account(){}
11 };
12 class Current: public Account {};
13 class Test { public: float x, y;};
14
15 // Demo Function
16 void someFunction(int& c)
17 {
18   c++;   // c can be modified
19   cout << "c has value " << c << endl; //will output 11
20 }
21
22 int main()
23 {
24    float f = 200.6f;
25     // narrowing conversion, but we have notified the compiler
26    int x = static_cast<int>(f); //1
27    cout << x << endl;
28
29    Account *a = new Current; //upcast - cast derived to base
30    //type-safe downcast - cast base to derived
31    Current *c = dynamic_cast<Current*>(a);  //2
32
33    const int i = 10;     // note i is const
34    //someFunction(i);    // is an error as someFunction
35                          // could modify the value of i
36    someFunction(*const_cast<int*>(&i));  //3
37        // Allow i be passed to the function but it will still remain at 10.
38    cout << "i has value " << i << endl; // will still output 10
39
40    a = new Account;
41    a->balance = 1000;
42     //convert account address to long
43    long addr = reinterpret_cast<long>(a); //4
44
45    // safe to convert long address into an Account
```

```
46      Account* b = reinterpret_cast<Account*>(addr); //5
47      cout << "The balance of b is " << b->balance << endl;
48
49      // could convert to any class regardless of
50      // inheritance - ok this time! (not safe)
51      Current* cur = reinterpret_cast<Current*>(addr); //6
52      cout << "The balance of cur is " << cur->balance << endl;
53
54      // works, but definitely not safe this time!
55      Test* test = reinterpret_cast<Test*>(addr); //7
56      cout << "The value of the float x is " << test->x <<
57           " and float y is " << test->y << endl;
58  }
59
```

The output of this code is:

```
C:\My Documents\My Teaching\OOP Notes\EE553_Output\cpp>casting
 200 //1 //2
 c has value 11
 i has value 10 //3
 The balance of b is 1000 //4 //5
 The balance of cur is 1000 //6
 The value of the float x is 1000 and float y is 5.98875e-39 //7
```

❶ The static_cast here demonstrates the most common casting operation that we have previously performed using the C-style cast. In this case, we are simply converting a float into an int and notifying the compiler of our intention - Use a static_cast for all such conversions. In this example 200.6 will be converted to 200 as output above.

❷ The dynamic_cast here demonstrates downcasting an Account object to a CurrentAccount object. A dynamic_cast must be used when there are virtual base classes (discussed later). In this example there is no output.

❸ The const_cast here takes the const i and allows it to be passed by reference to a function someFunction() that takes a non-constant parameter by reference. In this function we can treat the value just like a non-constant, but it has no impact on the original constant. As shown in the line directly above we would not have been allowed to pass this constant without the const_cast.

❹ The reinterpret_cast here converts the address of the Account object into a long.

❺ The reinterpret_cast is used again, this time to convert a long into a pointer to an Account object. This is dangerous as it is unchecked, but is perfectly

appropriate in this case. The output shows a balance of 1000, so it worked correctly.

❻ The reinterpret_cast is then used to convert the same long pointer into a Current object. Again it is fine in this case, but unchecked.

❼ The reinterpret_cast is finally used to do a very unsafe operation of converting the long into the address of a random Test object. This works perfectly, but it is totally inappropriate as you can see in the output the int value of 1000 has been converted to a float x, but the next block of random memory has also been assigned to the float y. Modifying y would cause unpredictable results, possibly crashing the application.

Method Parameters

Default Parameters

Default parameters can be added to the end of a parameter list. If a value is to be passed to a particular default parameter, then the preceding default parameters must also be passed. For example:

```
// Example Default Parameters

void decrement(int &aValue, int amount=1) {
      aValue = aValue - amount;
}

// To use this method, some examples

int x = 10;
decrement(x,3); // x now has the value 7
decrement(x);   // x now has the value 6
```

The full source code for this example is listed in **DefaultParameters.cpp**

Constant Parameters

To prevent modification of the parameters (when required), the use of constant parameters is good coding practice. It is particularly important when passing parameters by reference that you do not wish to be modified. Use the const keyword in the parameter list:

```
// Example Constant Parameters

void decrement(const int &aValue, int amount=1) {
```

```
        aValue = aValue - amount;  // NOT ALLOWED
}
```

The full source code for this example is in **ConstantParameters.cpp**

In this example the code segment has been modified to set the passed value to constant. When the value is modified it causes an error (at compile time). This code will not compile, and even if it did, it would be pointless having a decrement method that could not decrement.

Const Qualified Objects

Const can also be used to indicate that a method does not modify the calling object -- that the method is **const safe**. This is necessary, even if the method does not have any code that would modify the calling object. For example:

```
class Number{
private:
        int value;
public:
        Number(int value): value(value) {};
        void a() const {};
        void b() {};
        void c() const;
};

void Number::c() const {}   // note const needs to be present here too

int main() {
        Number n(5);  // mutable object
        n.b();

        const Number m(10);  //non-mutable object (i.e., const modified)
        m.a(); // fine
        m.b(); // error as b() could potentially modify the
               // constant modified object m
        return 0;
}
```

- Const and non-const functions can be called by non-const modified objects (i.e., regular mutable objects).
- Non-const functions cannot be called by const modified objects (even if the non-const function does not modify the object.)

Overloading in C++

Overloading was discussed in the section called "Overloading" in the context of general object-oriented programming and applies in the exact same way to the C++ language. In any C++ class the same method name may define many different operations depending on the parameters used. For example:

```
1
2
3   // Overloaded Methods
4
5   #include<iostream>
6   #include<string>
7
8   using namespace std;
9
10  class MyMaths
11  {
12    public:
13      virtual int add(int a, int b);
14      virtual float add(float a, float b);
15      virtual string add(string a, string b);
16  };
17
18  int MyMaths::add(int a, int b)
19  {
20    return (a+b);
21  }
22
23  float MyMaths::add(float a, float b) {
24    return (a+b);
25  }
26
27  string MyMaths::add(string a, string b)
28  {
29    return (a+b);
30  }
31
32  main() {
33    MyMaths m;
34
35    cout << "1 + 2 = " << m.add(1,2) << endl;
36    cout << "1.5 + 2.0 = " << m.add(1.5f,2.0f) << endl;
37    cout << "one + two = " << m.add("one","two") << endl;
38  }
39
```

The full source code for this example is in **overloading.cpp** In this application there are three different add() methods in the same class. Each method differs from the last by having different parameter argument types and so will behave slightly differently depending on the values passed. For example, the call that passes two int values will result in an int result that is the addition of the two integer values. If two string values are passed then the result is a string type result that is the concatenation of the two passed strings. So the output of this code segment will be:

```
1 + 2 = 3
1.5 + 2.0 = 3.5
```

```
one + two = onetwo
```

Note: **A method cannot be distinguished by its return type solely**, so the parameter list must differ in some way (types or number of arguments). Please be aware that this example is fundamentally flawed. If you look at the implementation of my add() methods, they all have exactly the same implementation, i.e. the statement return (a+b). So, how is it possible that the "+" operator is able to work with ints, floats or even strings? Well, this is called operator overloading.

Operator Overloading

C++ allows the programmer to associate specific functionality of a class with the standard predefined operators, such as '+','-','=','==' etc. One class which demonstrates the use of these overloaded operators is the String class, which associates the '+' with concatenation, '=' to assignment etc. We can overload the predefined operators to suit our own classes

For example, in the Account class we can add a + operator, allowing Account objects to be summed in an appropriate way. For example:

```
1
2  // Operator Overloaded Account Example
3
4  #include<iostream>
5  #include<string>
6
7  using namespace std;
8
9  class Account{
10
11   protected:
12
13     int accountNumber;
14     float balance;
15     string owner;
16
17   public:
18
19     Account(string owner, float aBalance, int anAccountNumber);
20     Account(float aBalance, int anAccountNumber);
21     Account(int anAccountNumber);
22
23     Account operator + (Account);
24     bool operator == (Account);
25
26     virtual void display();
27     virtual void makeLodgement(float);
28     virtual void makeWithdrawal(float);
29 };
```

```
30
31 Account::Account(string anOwner, float aBalance, int anAccNumber):
32     accountNumber(anAccNumber), balance(aBalance),
33     owner (anOwner) {}
34
35 Account::Account(float aBalance, int anAccNumber) :
36     accountNumber(anAccNumber), balance(aBalance),
37     owner ("Not Defined") {}
38
39 Account::Account(int anAccNumber):
40     accountNumber(anAccNumber), balance(0.0f),
41     owner ("Not Defined") {}
42
43 Account Account::operator + (Account a){
45     return Account(owner, balance + a.balance, accountNumber);
46 }
47
48 bool Account::operator == (Account a){
50     if ((a.balance == balance) && (a.owner == owner) &&
51           (a.accountNumber == accountNumber))
52     { return true; }
53     else return false;
54 }
56
57 void Account::display(){
58     cout << "account number: " << accountNumber
59       << " has balance: " << balance << " Euro" << endl;
60     cout << "This account is owned by: " << owner << endl;
61 }
62
63 void Account::makeLodgement(float amount){
64     balance = balance + amount;
65 }
66
67 void Account::makeWithdrawal(float amount){
68     balance = balance - amount;
69 }
70
71 int main() {
73   Account a = Account(10.50, 123456);
74   a.display();
75
76   Account b = Account("Derek Molloy", 35.50, 123457);
77   b.display();
78
79   Account c = Account("Derek Molloy", 35.50, 123457);
80   if (b == c) { cout << "b and c are equal!" << endl; }
81
82   Account d = b + a;
83   d.display();
84 }
```

The full source code for this example is in **OperatorOverloadAccount.cpp**

In this example the + operator has been overloaded to add the balance of the object on the RHS of the + to the object on the LHS. It does not add the accountNumber or the owner states (as this would not make sense). The method creates a new Account object and returns it from the method. This is assigned to con the LHS of the assignment, when Account d = b + a; is executed. The output from this code is:

```
C:\My Documents\My Teaching\EEN1035 Notes\source_notes\cpp>
OperatorOverloadAccount

account number: 123456 has balance: 10.5 Euro
This account is owned by: Not Defined
account number: 123457 has balance: 35.5 Euro
This account is owned by: Derek Molloy
b and c are equal!
account number: 123457 has balance: 46 Euro
This account is owned by: Derek Molloy

C:\My Documents\My Teaching\EEN1035 Notes\source_notes\cpp>
```

In the line of code:

```
Account d = b + a;
```

We can look at the statement as two stages. In the first stage b + a is called. Imagine that the + operator was replaced by an add() method. You would call b.add(a)to add an Account object a to an Account object b. Well the + operator is exactly like this - you can think of it as b.+(a) if you like. Now in my example, rather than add the object a to b, I allowed the method to create a new object. So, the first element passed to the operator is the object you are operating on, and the second element is the parameter that you are passing.

In the second stage d = (b + a); the assignment sets the LHS to be equal to the new object returned from the + method.

The reason that the + method has full access to the private data of the Account object a that was passed as a parameter is because the code was written inside the Account class, therefore, having full private access to the states of the Account class.

In this example the == operator allows a comparison of two Account objects, by comparing the states of the objects to see if they are equal. It is up to the programmer to decide what defines objects that are equal. For example, in the Account you may decide that two Account objects are equal if the owner and balance are the same, or you may also require that the accountNumber states are also equal. The == operator can therefore have specific implementation in different classes.

Overloading the Assignment Operator (Advanced)

The assignment operator = (aka the copy operator) is slightly more complicated, because you need to be able to chain operations (e.g., a = b = c), which means that the operator must return a reference to the value assigned.

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

class Number{
private:
	int a;
public:
	Number(int a);
	Number & operator = (const Number & source);
	void display();
};

Number::Number(int a): a(a) {}

void Number::display(){
	cout << "The value of the object at address " << this << " is " << a << endl;
}

Number & Number::operator = (const Number & source){
	if (this!= &source){   // make sure we are not copying from itself
		a = source.a;
	}
	return *this;    // return the reference (value of pointer this)
}

int main(){
	Number x(5), y(4), z(3);
	x.display();
	y.display();
	z.display();
	cout << "Performing the assignment operator now! " << endl;
	z = y = x;
	x.display();
	y.display();
	z.display();
	return 0;
}
```

This will result in the following output:

```
The value of the object 0x62ff0c is 5
The value of the object 0x62ff08 is 4
The value of the object 0x62ff04 is 3
Performing the assignment operator now!
The value of the object 0x62ff0c is 5
The value of the object 0x62ff08 is 5
The value of the object 0x62ff04 is 5
```

Non-Member Operators (Advanced)

Operator overloading is powerful and works very well but it can become complex. For example using the Number class above above you can write an overloaded operator to handle the operation (where n is an object of Number):

```
n = n + 5;
```

You would do this by writing a + operator that accepted a value of type int in this case and appended it to the value of the object n. The int value 5 would be passed as an argument to this operator. However, the following is much more complex:

```
n = 5 + n;
```

Because in this case we would try to call the + operator of the int type (to accept a Number object), which doesn't exist. We can solve this by writing a non-member operator as follows:

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

class Number{
private:
        int a;
public:
        Number(int a);
        void display();
        int getValue() const { return a; }  //getter does not modify object
};

Number::Number(int a): a(a) {}

void Number::display(){
        cout << "The value of the object at address " << this << " is " << a << endl;
}

// This is a non-member operator that can deal with 3+x or x+3. Please note that it
// works because I have added a getter method (Friend functions can help with this).
// This is a function, it is not a method of the class Number

Number operator + (const Number & left, const Number & right){
        return Number( left.getValue() + right.getValue());
}

int main(){
        Number x(5);
        x = 3 + x;
        x = x + 2;
        x.display();
        return 0;
}
```

As expected, this program gives the output:

```
The value of the object at address 0x62ff04 is 10
```

Overloading the pre and post increment operators (Advanced)

An unusual operator that is worth mentioning is the pre-increment (++x), post-increment (x++) operators, as the same operator is used in both cases but the functionality differs depending on the location of the operator. This discussion is consistent for the pre-decrement (--x) and post-decrement operators (x--) also. The following example demonstrates how these operators are implemented in C++ for a basic class called Number. I have kept this class as simple as possible so the overall example is not that useful; however, remember this could be applied in the same way to more complex classes.

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

class Number{
private:
        int a;
public:
        Number(int a);
        Number & operator ++ ();
        Number operator ++ (int);  // int is a dummy value to indicate post increment
                                   // this will always be int for all classes
        void display();
};

Number::Number(int a): a(a) {}

Number & Number::operator ++() {
        a++;
        return *this;  // pre-increment increments before assignment L=R
}

Number Number::operator ++(int){ // again int is a dummy value for the compiler
        Number temp = *this;         // make a copy of the current object
        a++;
        return temp;   // post-increment x = y++; assigns y to x before the increment.
}

void Number::display(){
        cout << "The value of the object at address " << this << " is " << a << endl;
}

int main(){
        Number x(5), y(0), z(0);
        y = x++;   // post increment
        cout << "Displaying y = x++ :" << endl;
        y.display();
        z = ++x;   // pre increment
        cout << "Displaying z = ++x :" << endl;
```

```
        z.display();
        cout << "Displaying x after both operations :" << endl;
        x.display();
        return 0;
}
```

In this example, the two operators are identified from each other using a dummy int value argument. Please note that this will always be int, regardless if your class is called Account or anything else.

Note that the post-increment operator is more computationally expensive than the pre-increment (pre-decrement) as it must make a copy of the source object.

The output of this example is as follows:

```
Displaying y = x++ :
The value of the object at address 0x62ff08 is 5
Displaying z = ++x :
The value of the object at address 0x62ff04 is 7
Displaying x after both operations :
The value of the object at address 0x62ff0c is 7
```

Destructors

> If you think C++ is not overly complicated, just what is a protected abstract virtual base pure virtual private destructor and when was the last time you needed one?
>
> --Tom Cargill

Since the responsibility for the initialisation of the class is in the hands of the programmer of the class, the responsibility for the destruction of the class should also be the responsibility of the programmer. In C++ we can write a destructor for a class that performs some clean-up operations and even notifications. To define a destructor for a class the ~ symbol is used in front of the class name, so for the Account class you would use:

```
class Account{
     // state definition
 public:
     Account(int theNumber, float theBalance);
     virtual ~Account(); //destructor

     // other member methods here.
};

Account::~Account()
{
```

```
    cout << "Account object being destroyed!" << endl;
}
```

Note: A similar destructor is also written for the CurrentAccount class.

The destructor must take no parameters and cannot return a result. A constructor cannot be virtual, but a destructor can (and should) be virtual (we will discuss this later).

When is a destructor called?

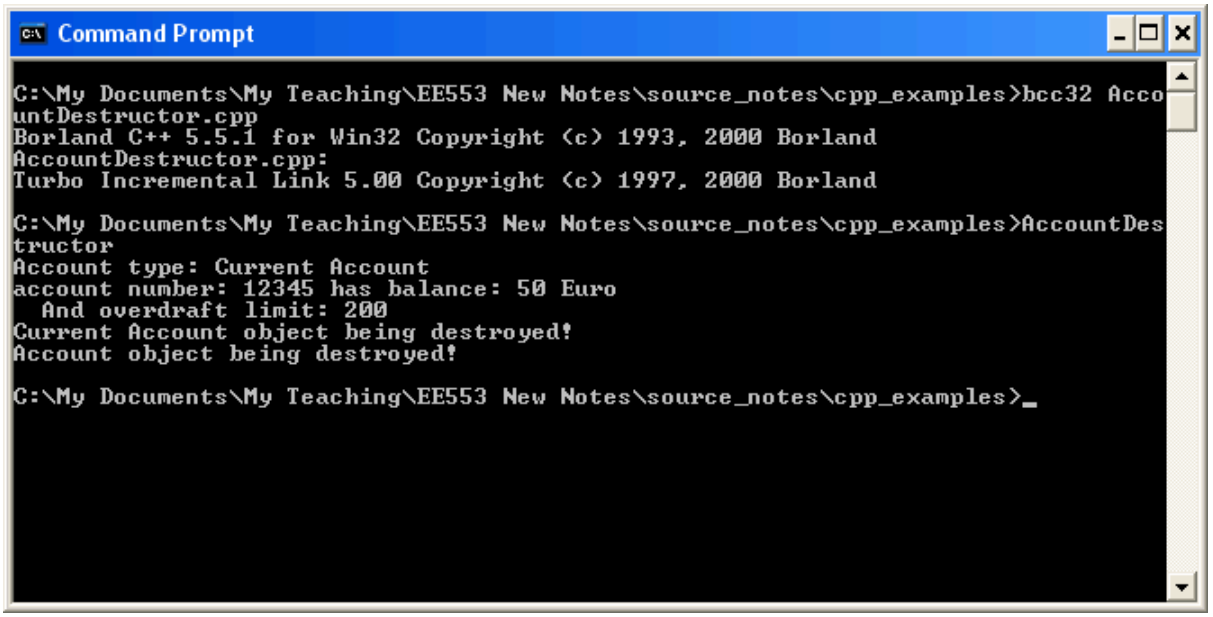
- It is not invoked explicitly.
- It is called when an object goes out of scope.

Suppose an object of the Account class was created as below.

```
// main() method where CurrentAccount is a child of the Account class,
// with a destructor as defined just above.

int main()
{
    CurrentAccount b = CurrentAccount(50.0, 12345, 200.0);
    b.display();
}
```

The full source code for this example is listed in **AccountDestructor.cpp** What would happen? Since CurrentAccount is a child of the Account class then destruction of an object of the CurrentAccount object will result in the Account destructor being called. When this example is run you will get the output as in Figure 3.8, "The output from the Account Destructor Example.".

Figure 3.8. The output from the Account Destructor Example.



In Figure 3.8, "The output from the Account Destructor Example." screen grab it can be seen that the CurrentAccount destructor followed by the Account destructor were called, just after the main() method ended, and just before the application ran to completion.

If you don't declare a destructor, C++ generates an implicit destructor (default destructor), which frees up the memory of the object.

This is suitable for most classes, however a destructor may be required for specific actions:

- Notifying other objects about an imminent destruction.
- Closing open files, database connections, sockets etc.
- Freeing dynamically allocated data. If an object does not free data that is dynamically allocated, then that memory will remain locked until the application terminates.

The problems with losing dynamically allocated memory can be prevented from occurring by using the following structure:

```
class SomeClass{
            // the states
      public:
            SomeClass();
            virtual ~SomeClass(); //The destructor
            // other methods
};

SomeClass::SomeClass(){
     // Allocate extra space here in
     // the constructor!
}

SomeClass::~SomeClass(){
     // De-allocate that extra space!
}
```

In this case the programmer need only worry about freeing the extra space allocated in the constructor, not about the space for the object (or states) itself!

The destructor for the base class must be called when an object of a derived class is destroyed. This happens automatically! (in the order):

- C++ calls the programmer defined destructor for the derived class.
- C++ then calls the programmer defined destructor for the base class.
- C++ then frees the object space itself.

This is the opposite way to how an object is constructed!

Multiple Constructors

A constructor can take parameters to initialise the state. However, it is possible for the initial state to be determined by different sets of parameters.

For example: you might wish to set up an account, by specifying:

- The account number, name, balance.
- The account number, name, (no balance, defaults to 0.0).
- The account number, name, initial balance, and a referee.
- etc.

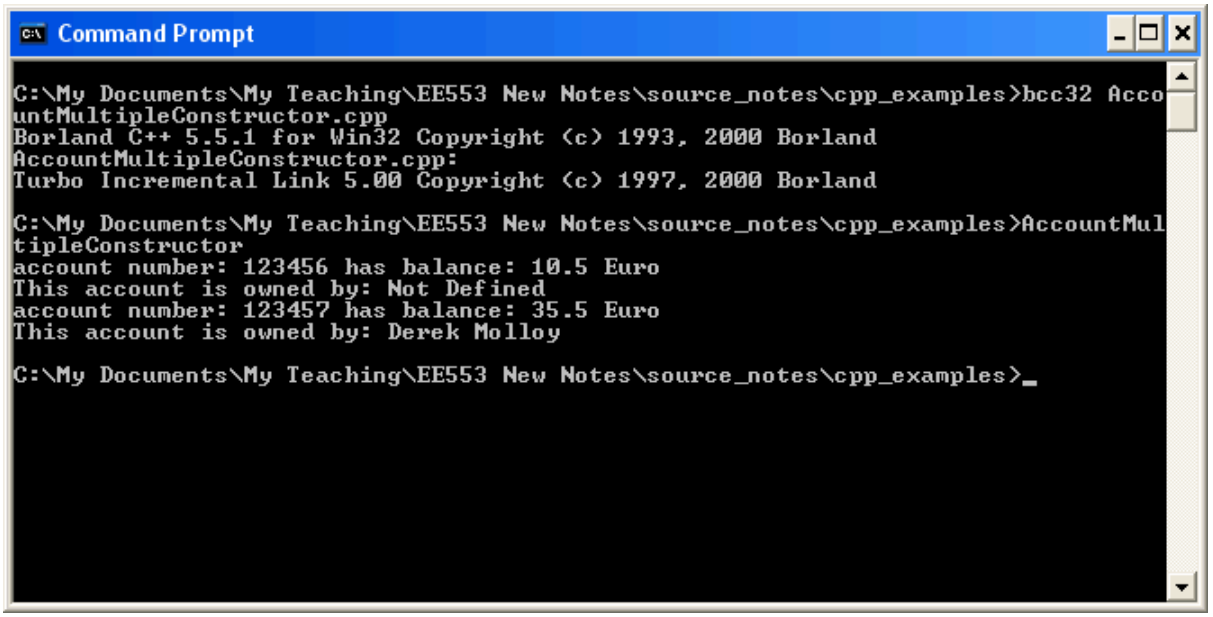
So it is useful that this constructor can be overloaded. So for the Account class:

```
1
2  // Basic Bank Account Example demonstrating multiple constructors
3
4  #include<iostream>
5  #include<string>
6
7  using namespace std;
8
9  class Account{
10
11   protected:
12
13     int accountNumber;
14     float balance;
15     string owner;
16
17   public:
18
19     Account(string owner, float aBalance, int anAccountNumber);
20     Account(float aBalance, int anAccountNumber);
21     Account(int anAccountNumber);
22
23     virtual void display();
24     virtual void makeLodgement(float);
25     virtual void makeWithdrawal(float);
26 };
27
28 Account::Account(string anOwner, float aBalance, int anAccNumber):
29    accountNumber(anAccNumber), balance(aBalance),
30    owner (anOwner) {}
31
32 Account::Account(float aBalance, int anAccNumber) :
33    accountNumber(anAccNumber), balance(aBalance),
34    owner ("Not Defined") {}
35
36 Account::Account(int anAccNumber):
37    accountNumber(anAccNumber), balance(0.0f),
```

```
38     owner ("Not Defined") {}
39
40
41  void Account::display(){;
42     cout << "account number: " <<accountNumber
43          << " has balance: " << balance << " Euro" << endl;
44     cout << "This account is owned by: " << owner << endl;
45  }
46
47  void Account::makeLodgement(float amount){
48     balance = balance + amount;
49  }
50
51  void Account::makeWithdrawal(float amount){
52     balance = balance - amount;
53  }
54
55  int main()
56  {
57   Account a = Account(10.50, 123456);
58   a.display();
59
60   Account b = Account("Derek Molloy", 35.50, 123457);
61   b.display();
62  }
63
```

The full source code for this example is in **AccountMultipleConstructor.cpp**

Figure 3.9. The Account class with Multiple Constructors example output.



You can see from Figure 3.9, "The Account class with Multiple Constructors example output." that the output for the a state when the display() method is called, displays the string "Not Defined", as assigned in the second constructor.

A destructor **cannot** be overloaded. There can only be one destructor, as destructors do not take any parameters.

In C++, neither constructors nor destructors are inherited by derived classes. i.e. if you define a particular constructor in the base class that takes 3 parameters, then the derived class does not have this constructor unless it explicitly defines it.

In these notes, I use the Stroustrup version of using a private helper function, of the form:

```
class A {
        int x;
        void init(int a) { x=a; ... }
public:
        A(int x) { init(x); }
        A() { init(402); }
        ...
};
```

To do the same thing in C++11 you can write:

```
class A {
        int x;
public:
        A(int x): x(x) { ... }
        A(): A(402) {}
        ...
};
```

Implicit and Explicit Constructors (Advanced Topic)

It is possible to make the default constructor private, preventing it from being called by the programmer. For example:

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

class X{
private:
        int a;
        X(); // made private to prevent default constructor being called
public:
        X(int a);
        void display();
};

X::X(int a): a(a) {}
```

```
void X::display(){
	cout << "The value of a is " << a << endl;
}

int main(){
	X x(5);   // cannot use X x;
	x.display();

	// Since there is no default constructor and therefore only
	//  one constructor C++ permits implicit conversion e.g.,
	X y = 'A';
	y.display();
	return 0;
}
```

This might allow you to require the programmer to use one particular constructor, but it also **allows implicit conversions** to be performed on construction. For example the line to create the object y automatically converts the character 'A' to the number 65 (its ASCII equivalent) giving the output below:

```
The value of a is 5
The value of a is 65
```

If you want to prevent such implicit conversions from being performed, you can use the **keyword explicit** that then imposes that a variable of the correct type is required. For example, if you change the constructor declaration above to the following:

```
class X{
private:
	...
public:
	explicit X(int a);
	...
};
```

Then an error will arise, preventing the implicit conversion from being compiled:

```
error: conversion from 'char' to non-scalar type 'X' requested
```

The Copy Constructor

The copy constructor is a special constructor that is automatically a part of every class. It allows a copy to be made of an object. The default copy constructor creates an identical copy of the object, so states have the exact same values. The default copy constructor can be called simply by:

```
int main() {
   Account a = Account("Derek Molloy",35.00,34234324);
   Account b(a);

   a.display();
   b.display();   // exact same output, both have a balance of 35.00 Euro

   a.makeLodgement(100.0);

   a.display();   // a now has a balance of 135.00
   b.display();   // b has the same balance of 35.00
}
```

The full source code for this example is in **DefaultCopyConstructor.cpp**. This call creates a new Account object, with its own memory space, with the exact same state values as the original Account object.

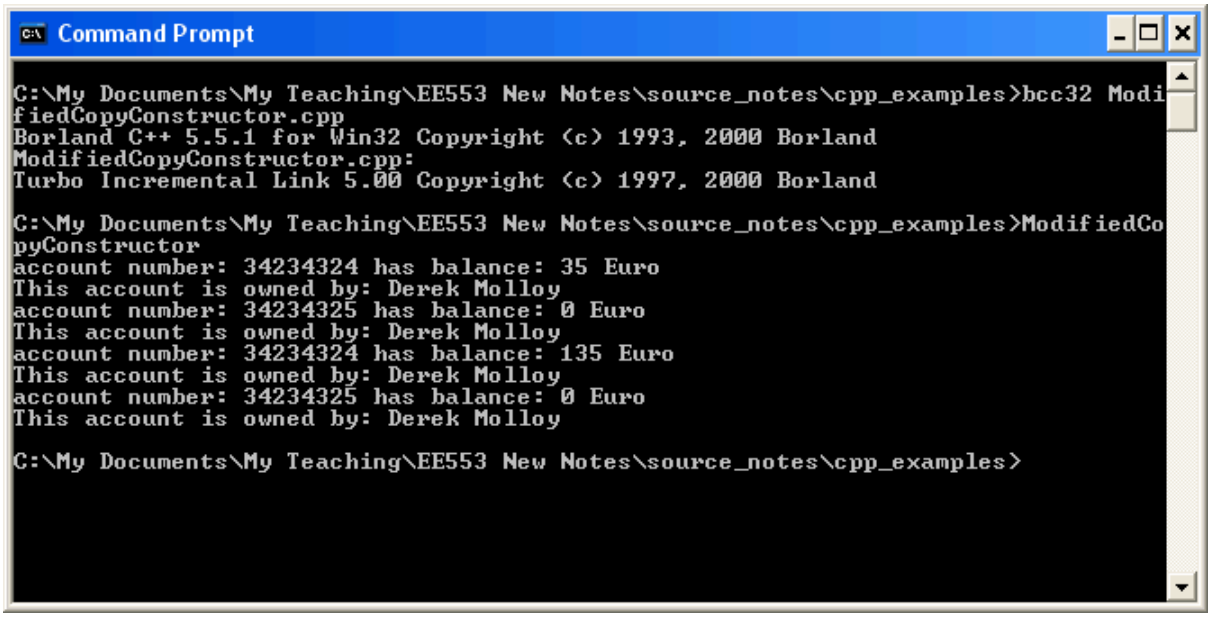
This facility provides limited functionality. For example, if the details of one Account object were being used to create another Account object, creating another account for the same owner, we can modify the copy constructor to provide specific copy behaviour.

```
class Account {

 protected:

  int accountNumber;
  float balance;
  string owner;

 public:

  Account(string owner, float aBalance, int anAccountNumber);
  Account(float aBalance, int anAccountNumber);
  Account(int anAccountNumber);
  Account(const Account &sourceAccount);

  ...

};

...

Account::Account(const Account &sourceAccount):
     accountNumber(sourceAccount.accountNumber + 1),
     balance(0.0f),
     owner (sourceAccount.owner) {}
```

The full source code for this example is in **ModifiedCopyConstructor.cpp**.

The output from this example can be seen in Figure 3.10, "The Output from the Modified Copy Constructor Example." (The same main() method is used). In this case the copy constructor has been modified to copy the account holder name, but to zero the balance of the new account and to set the new account number to be the last one plus 1 (this is not an ideal implementation, just an example).

Figure 3.10. The Output from the Modified Copy Constructor Example.



Separate Compilation

C++ supports separate compilation, where pieces of the program can be compiled independently through the two stage approach of compilation and then linking, so changes to one class would not necessarily require the re-compilation of the other classes. The compiled pieces of code (.o or .obj files)[8] are combined through the use of the linker (in the use of Borland C++ it is **ilink32.exe**). Separate compilation allows programs to be compiled and tested one class at a time, even built into libraries for later use. It is therefore good practice to place each class in a separate source file to take full advantage of separate compilation with the C++ language.

The source code for each class is stored in two files:

- A source file (.cpp) - the implementation of the methods.
- A header file (.h) - the definition of the class.

The header file contains the declarations for the methods contained in the cpp file, allowing for these cpp files to be compiled into libraries. The cpp file will define the methods and by including the header file within the cpp file you will ensure consistency between the declarations and definitions.

So the Account class would take the form of three separate files:

- **Account.h** - That stores the class declaration and definition.
- **Account.cpp** - That stores the method definitions for that class.
- **Application.cpp** - That stores the application, i.e. the main() method for the application.

So the header file (Account.h) will have the form:

```
1
2
3   #include<iostream>
4   #include<string>
5
6   using std::string; //1 // only string is required
7
8   class Account{
9
10    protected:
11
12     int accountNumber;
13     float balance;
14     string owner;
15
16    public:
17
18     Account(string owner, float aBalance, int anAccountNumber);
19     Account(float aBalance, int anAccountNumber);
20     Account(int anAccountNumber);
21     Account(const Account &sourceAccount);
22
23     ...
24  };
25
```

1. You should not place using directives in header files where possible. If we were to write using namespace std; in our header file, all cpp files that include this header would also include this using directive. This would have the effect of turning off namespaces in your project (in this case for std only).

The implementation file (Account.cpp) will have the form:

```
#include "Account.h"

using namespace std;

Account::Account(string anOwner, float aBalance, int anAccNumber):
      accountNumber(anAccNumber), balance(aBalance),
      owner (anOwner) {}

Account::Account(float aBalance, int anAccNumber) :
```

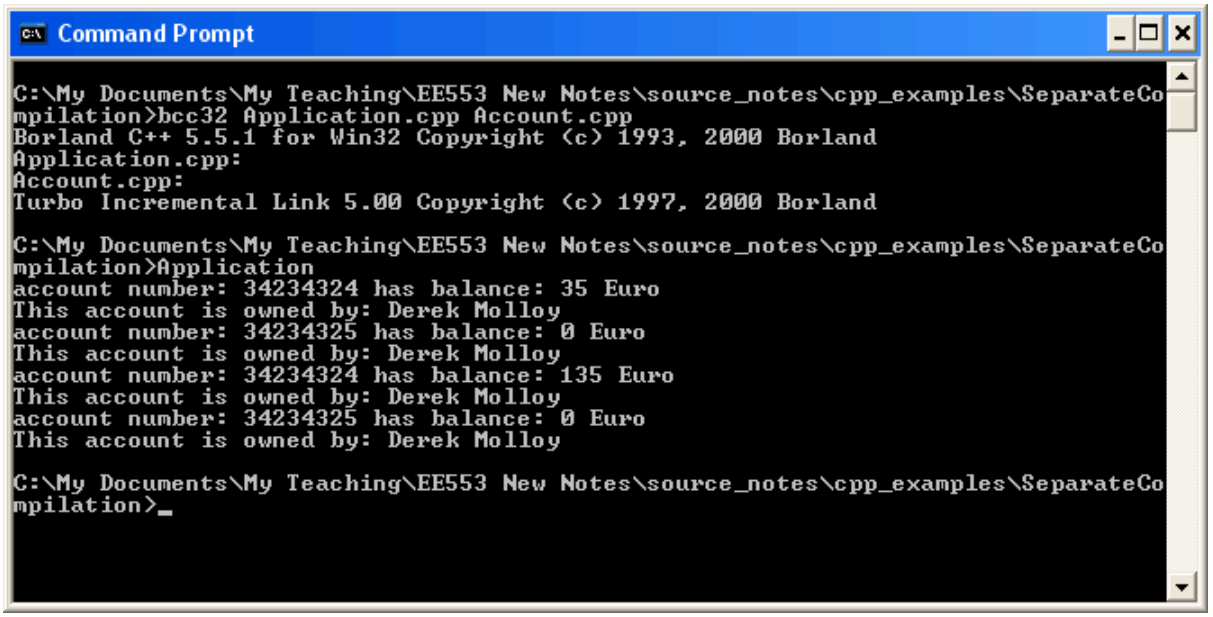
```
      accountNumber(anAccNumber), balance(aBalance),
      owner ("Not Defined") {}

Account::Account(int anAccNumber):
      accountNumber(anAccNumber), balance(0.0f),
      owner ("Not Defined") {}

Account::Account(const Account &sourceAccount):
      accountNumber(sourceAccount.accountNumber + 1),
      balance(0.0f),
      owner (sourceAccount.owner) {}

...
```

And the application (Application.cpp) will have the form:

```
#include "Account.h"

int main() {
  Account a = Account("Derek Molloy",35.00,34234324);

  ...
}
```

To compile the application, you must now specify the files to be used in the compilation. So, to compile all the files at once use: **bcc32 Application.cpp Account.cpp**, where one of the source files contains a main() method. This can be seen in Figure 3.11, "Compilation, and the output from the Separately Compiled Example.".

Figure 3.11. Compilation, and the output from the Separately Compiled Example.



Just before we continue we need to briefly discuss preprocessor directives. Preprocessor directives are orders for the preprocessor, not for the program itself. They must be specified in a single line of code and should not end with a ; (semicolon). Some preprocessor directives are: #include (insert a header file here), #define (define a constant macro)

```
#define PI 3.14
```

#undef (removes definition), #if, #ifdef, #ifndef, #endif, #else, #elif (control directives to remove part of a program depending on the condition),

```
#ifndef MAX_WIDTH
#define MAX_WIDTH 1000
#endif
```

#line (allows control over compile time error messages), #error (allows us to abort compilation if required), e.g.

```
#ifndef __cplusplus
#error You need a C++ compiler for this code!
#endif
```

and #pragma (used for compiler options specific to a particular platform and compiler).

If there are multiple classes, some of which use the same parent, you can use compiler directives to prevent the re-definition of the same class, which would result in a compiler error. These directives can be placed around the class definition such as:

```
#ifndef currentAccount_h  //check not already defined
#define currentAccount_h  //if not, define!

#include "Account.h"    //include the account header

class CurrentAccount: public Account{

  protected:
    float overDraftLimit;
    ...

     public:

    CurrentAccount(int theNumber, char* theOwner, float theOverdraftLimit);
    ...
};

#endif  // currentAcount_h
```

In this case, the compiler directives simply state that if the CurrentAccount class is already defined, so do not redefine it. This is determined by the currentAccount_hvalue, that if undefined is simply defined, and so used as a flag. This process is to ensure that

we have not broken the **C++ single definition rule**: You can declare anything as many times as you want, but you can only define it once.

Here is an example of the complexities that arise with separate compilation. In this example A is the parent of AA and B. AA is a child of A and B is a part-of AA. B is a child of A and AA is a part-of B. I have added the minimum namespace usage necessary in the header files: **testapp.cpp A.h A.cpp AA.h AA.cpp B.h B.cpp**

Fortunately, for the purpose of professional development, tools such as Integrated Development Environments(IDEs) can automatically insert the required preprocessor directives.

There is also an alternative to the include guard structure which is to use a **pragma** called **once** which provides the same functionality. It's a much simpler solution that is not part of the C++ standard, but it is supported by most compilers. It's not part of the standard because it can be confused by strange file system configurations (e.g., those that use symbolic links in the C/C++ include paths), so include guards are still preferred. It has the structure:

```
#pragma once
#include<iostream>

class Student {
...
};
```

Note that the include guards are no longer required.

Alternatives to the preprocessor

Clearly, the preprocessor is very useful for providing include guards but one feature that is weak is the use of the preprocessor for the definition of constants. Again, remember that the preprocessor doesn't 'understand' C++, it simply performs basic editing/administration functions. As such, if you define the following:

```
#define PI 3.14159;
```

the preprocessor will simply replace the letters PI with the value 3.14159 AND the semicolon!! This can be a very difficult bug to identify because it will manifest itself very subtly in your code. For example,

```
area = PI * radius * radius;
```

Would be replaced by the line:

```
area = 3.14159; * radius * radius;
```

for the purpose of compilation. You will see the former when you view your code, but the compiler will see the latter and give quite obscure compile errors.

Thankfully, there is an alternative since C++11 by using the constexpr that allows you to use a constant where const alone would not be permitted. For example, constexpr can be used in switch case statements.

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

// constexpr added in C++11. Note the use of a semicolon
constexpr double PI = 3.14159;
constexpr const char * EEN1035 = "EEN1035 OOP with Embedded Systems";

int main() {
   cout << "Welcome to " << EEN1035 << endl;
   cout << "The value of pi is " << PI << endl;
}
```

```
Welcome to EEN1035 OOP with Embedded Systems
The value of pi is 3.14159
```

---

[8] The .o or .obj filename extensions are called object files, but importantly this has nothing to do with object, as in object-oriented, rather it simply means object, as in goal or aim.

Dynamic Creation of Objects

Pointers can be used to manage storage that is allocated manually while the program is executing. The new and delete keywords allow this manual allocation and deallocation of memory in the C++ language. This facility is very powerful. For example, we can create the memory space to manipulate an image or to open a temporary record at any point in the code. This feature can, however, lead to serious difficulties.

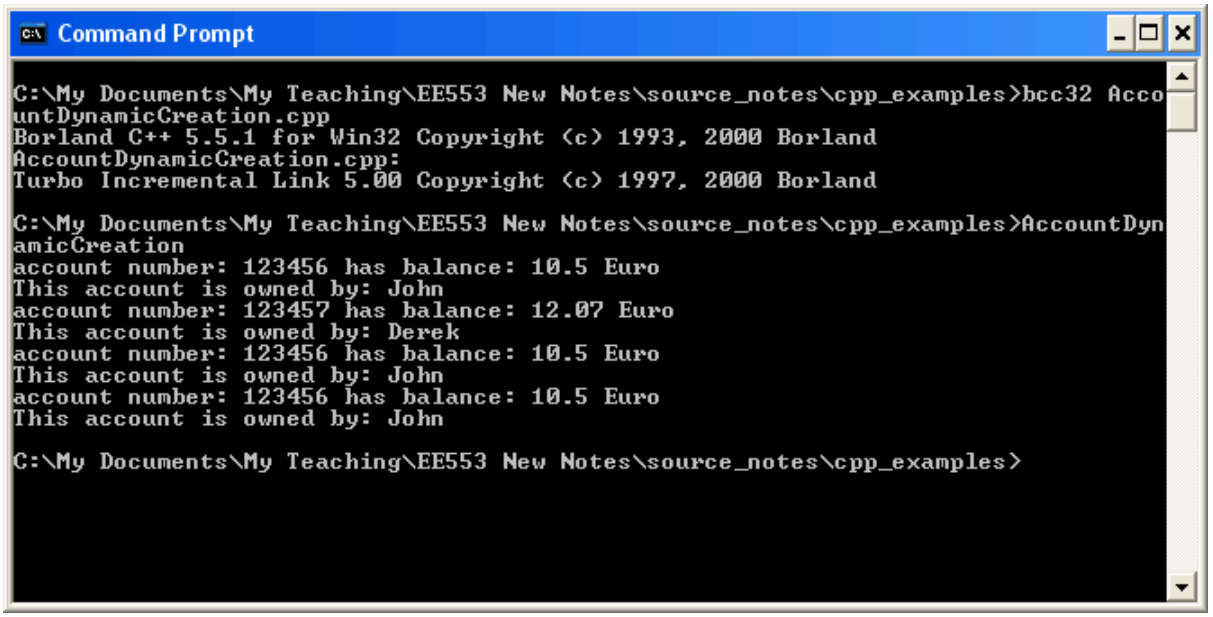
For an example of problems that can arise, see this code segment:

```
int main() {
   Account *a = new Account("John", 10.50, 123456);
   Account *b = new Account("Derek", 12.07, 123457);

   a->display();
   b->display();

   b = a;

   a->display();
   b->display();
 }
```

The full source code for this example is here - **AccountDynamicCreation.cpp**.

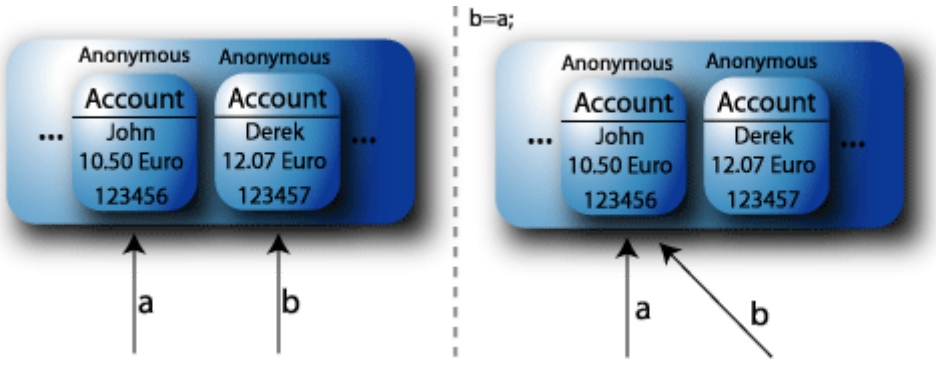
When run, this application gives the output as in Figure 3.12, "The Account Dynamic Allocation Example.". The code dynamically creates two Account objects and assigns them to the a and b Account pointers. The display() method is called on both objects resulting in the display of two account details, John and Derek's accounts. The assignment b = a; is the key problem here. The only reference we have to these two "anonymous" objects is the two pointers. If one pointer is assigned to the other then any reference we had to the second object (Derek's account) is lost. This results in the object being lost in memory, and not capable of being deallocated. This loss is called a **memory leak**. Since both pointers are now pointing at the same Account object, the same details will be displayed when the display() method is called on each pointer.

Figure 3.12. The Account Dynamic Allocation Example.



This can be further illustrated as in Figure 3.13, "The Account Dynamic Allocation Illustration.".

Figure 3.13. The Account Dynamic Allocation Illustration.



After the assignment b=a the pointer b is pointing to the first Account object. The second object "Derek's" Account has no reference or pointer and so is lost. The memory associated with this object will only be deallocated (by the operating system) after the application has run to completion.

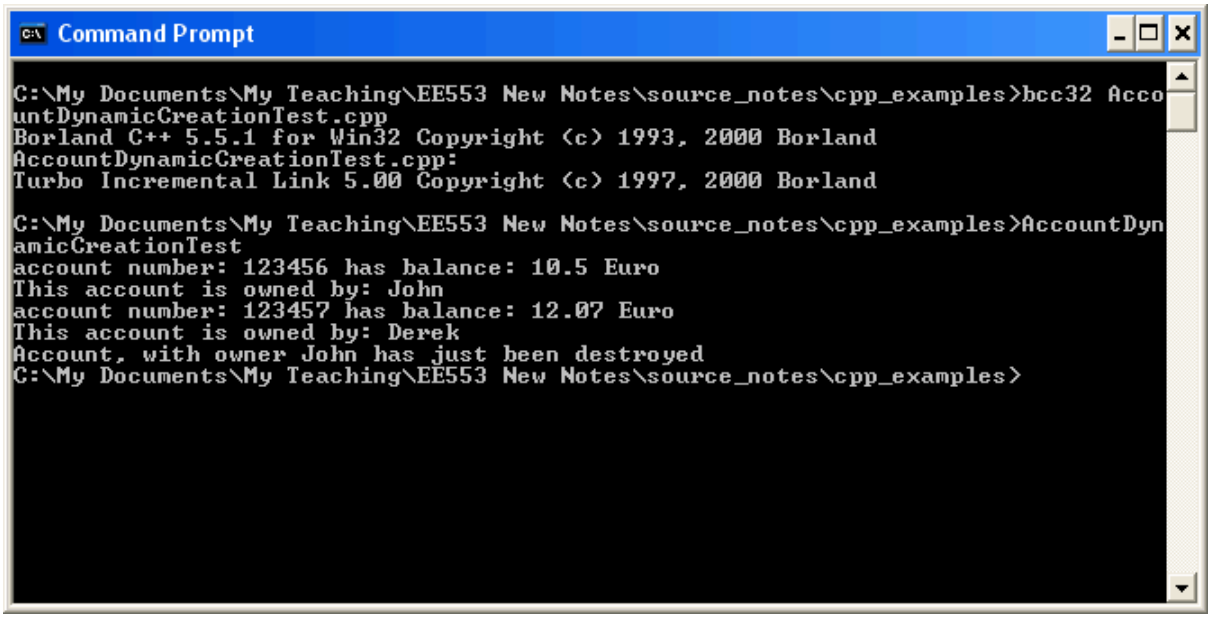
A further problem with the dynamic allocation/deallocation of data is that the delete call must be invoked by the programmer before the destructor of the class is called. For example, when a destructor is added to the code segment above that simply displays that an account is being destroyed (it could be doing something important like creating a paper record), we could use the code:

```
int main()
 {
   Account *a = new Account("John",10.50, 123456);
   Account *b = new Account("Derek",12.07, 123457);

   a->display();
   b->display();

   b = a;

   delete a;  // deletes John's account
 }
```

(The full source code for this example is here - **AccountDynamicCreationTest.cpp**).

Execution of the application will result in the output, as shown in Figure 3.14, "The Account Dynamic Deallocation First Test.":

Figure 3.14. The Account Dynamic Deallocation First Test.



So the destructor is called on the Account object with John's details, but the destructor is never called on the Account object with Derek's details.

Suppose the code is altered to call a delete on both pointers a and b, like:

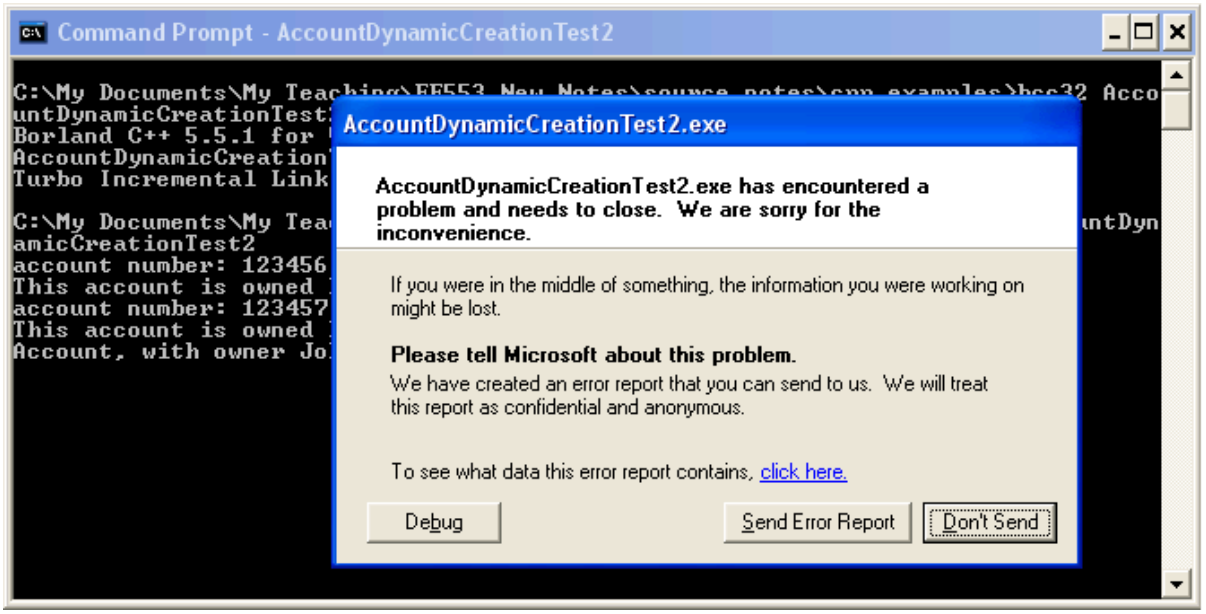
```
int main() {
   Account *a = new Account("John",10.50, 123456);
```

```
    Account *b = new Account("Derek",12.07, 123457);

    a->display();
    b->display();

    b = a;

    delete a;  // deletes John's account
    delete b;  // tries to delete John's account again
}
```

(The full source code for this example is here - **AccountDynamicCreationTest2.cpp** - Caution, this code when run may crash your computer.)

What would the output be? Well it would be as in Figure 3.15, "The Account Dynamic Deallocation Second Test.":

Figure 3.15. The Account Dynamic Deallocation Second Test.



If you run this application on Microsoft Windows, Microsoft will apologize for the inconvenience for the error! You can tell them about the problem, but they probably don't really care!

So what is the problem? Well since both pointers a and b point at the same Account object, the first delete call deletes the object referenced by the pointer (John's account), but the second call to delete tries to delete something that is not there (John's account again), thus causing the crash.

When applying delete to a pointer to an object, it calls the destructor for the object before releasing its storage.

Arrays of Objects

"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."

--Stan Kelly-Bootle

An array is an indexed collection of objects. Each object in an array has the same type.

```
ElementType theArray[numberofElements];
Account allTheAccounts[2000];
```

Each element is initialised by calling the implicit constructor (default constructor). If there is no implicit constructor, we could use (provided we use the correct number of elements):

```
int main()
{
  Account someAccounts[3] = {
        Account("Tom", 20.00, 123456),
        Account("Richard", 55.00, 123457),
        Account("Harry", 99.00, 123458) };  //1
  Account *p = &someAccounts[0]; //2

  someAccounts[2].display();    //displays Harry //3

  p->display();           //displays Tom //4
  (++p)->display();     //displays Richard //5
  (p+1)->display();     //displays Harry //6

  //Warning! The pointer p now points at Richard, not Tom! //7
}
```

The full source code for this example is listed in **AccountArrayExample.cpp**.

❶ The array has defined and initialised 3 elements, by calling their constructors. The someAccounts array has 3 elements (0 to 2).

❷ The pointer p is set to point at the address of the first element in the someAccounts array - element 0.

❸ The array can be indexed directly using the someAccounts array variable, so someAccounts[2] is a reference to the last element in the array and can be operated on in the same way as any other object, using the . operator.

❹ The pointer p is currently pointing at the first element in the array, someAccounts[0] and when display() is called on the pointer p the details of Tom's account will be displayed.

❺ The pointer p has been pre-incremented, and now points at the second element in the array someAccounts[1]. If (p++)->display(); had been used instead, then the post-increment would have incremented the index after the display() was called, and so it would have displayed "Tom's" details instead of "Richard's" details. Note that the pointer index has been incremented by one and now points directly at the second element in the array.

❻ This call displays the details of the last element in the array "Harry's" details, as the pointer p points at the second element in the array someAccounts[1] , but is further offset by one (not incremented by 1!).

❼ The pointer remains pointing at the second element in the array someAccounts[1]. It is important to use incrementing and offsetting correctly when using pointers, otherwise your applications will not work correctly.

Pointers can be used to allocate arrays of objects using the new keyword.

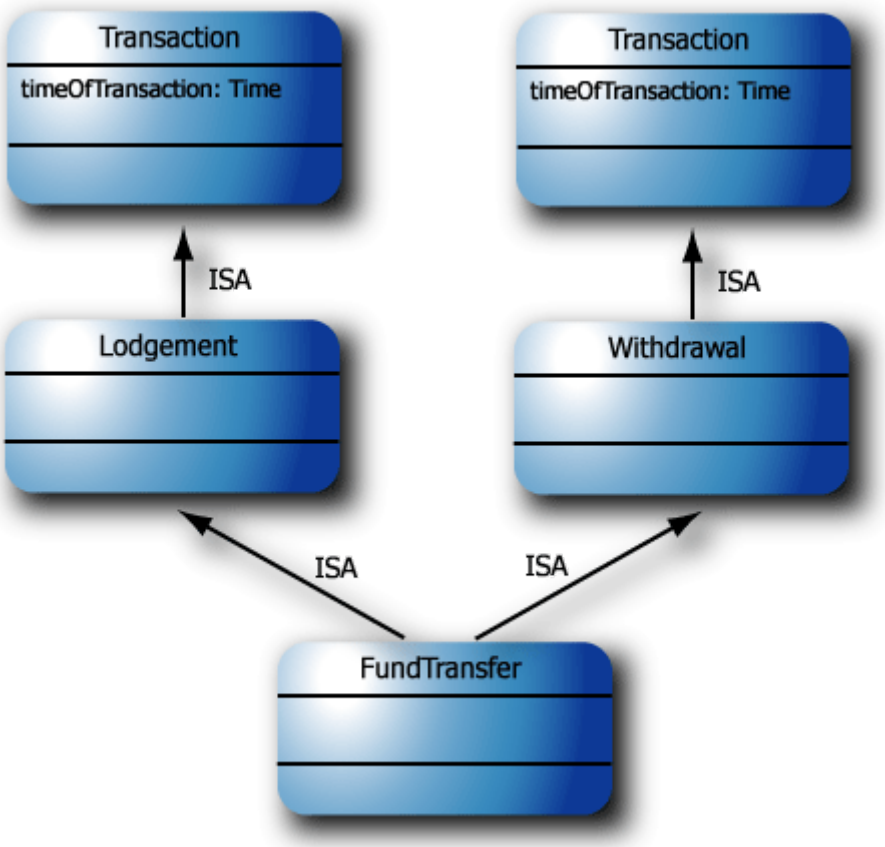
```
Account *ptr;
ptr = new Account[10];
```

A pointer to an array of objects has the exact same form as a pointer to an object. There is no way to distinguish between a pointer to a type and a pointer to an array of that type, except by careful coding. To destroy the array use delete[] ptr; If ptr points to an array then delete ptr; is undefined. Here is an example of using delete[] on an array of your own objects: **using_delete.cpp**

Multiple Inheritance

C++ allows the use of multiple inheritance, where one class can have more than one base class. Remember back to the Transaction, Lodgement and Withdrawal classes that we discussed previously. If we wished to create a FundsTransfer class, it would have properties of both a Lodgement and a Withdrawal. So the following structure could be devised, as in Figure 3.16, "Multiple Inheritance with the FundTransfer class.".

Figure 3.16. Multiple Inheritance with the FundTransfer class.



This is a valid structure as a Lodgement IS A Transaction, a Withdrawal IS A Transaction and at different times during execution a FundTransfer IS A Lodgement and a Withdrawal.

Two separate Transaction objects are being created, and they do not interfere with each other (they have their own state). When using these Transaction objects you must be careful to resolve the ambiguity that results from having two objects of the same class. Suppose, a method was written in the FundTransfer class to calculate the time taken for the physical cash transfer to take place (the time between the withdrawal and the lodgement), you could use a code segment like:

```
class FundTransfer: public Lodgement,
                    public Withdrawal
 {
    private:
        // states..

    public:

      virtual int getTimeTaken()
      {
         return ( Lodgement::timeOfTransaction -
                  Withdrawal::timeOfTransaction );
      }
 };
```

In this segment of code the multiple parents are comma separated and the getTimeTaken() returns the time taken as an int (could be milliseconds). In this method, the ambiguity is resolved between the two timeOfTransaction states that are inherited by the FundTransfer class, by specifying the parent that resulted in the inheritance of that state. So, Lodgement::timeOfTransaction is the timeOfTransaction state inherited through the Lodgement parent.

A common base class may not be inherited twice!

This same notation may be used to resolve ambiguity when referring to the class directly. If the Transaction had a display() method, and it was to be called directly, we could use:

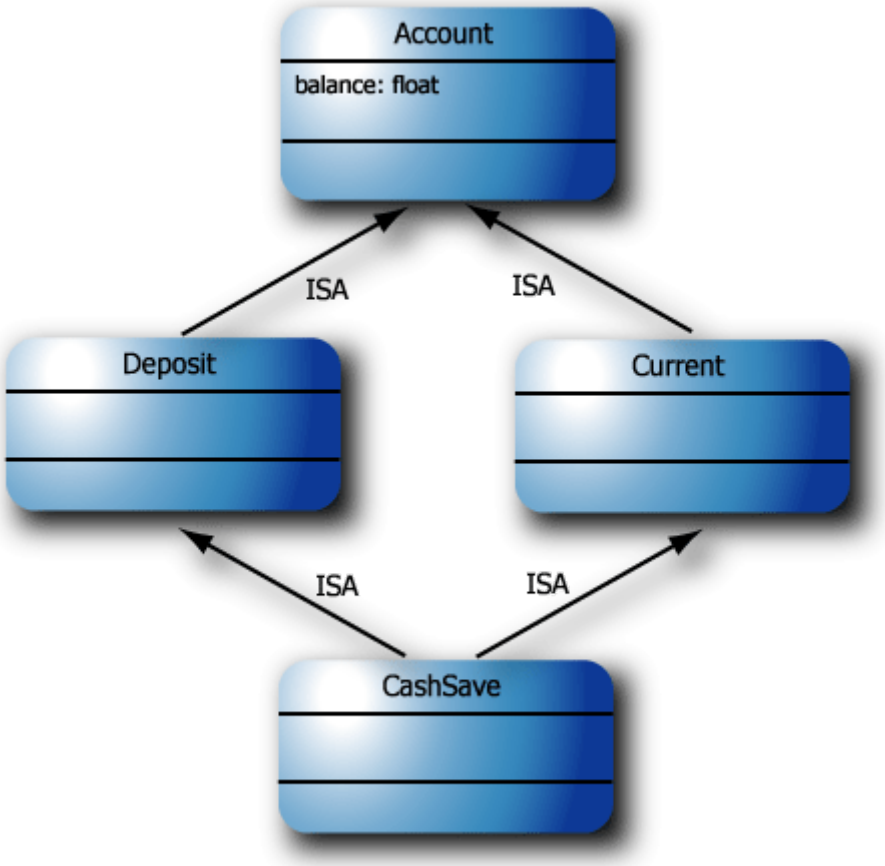
```
FundTransfer *fundPtr;
fundPtr -> display();           // not allowed - ambiguous
fundPtr -> Lodgement::display();     // OK!
fundPtr -> Withdrawal::display();    // OK!
```

This is awkward and does not give us an appropriate level of abstraction. It would be more appropriate to declare a new display() method in the FundTransfer class to avoid this ambiguity.

In the previous example it made sense to have two individual Transaction objects, each with its own state, otherwise it would not have been possible to calculate the time taken. This is not always the case. For example:

If the task was to create a special Current Account that had all the benefits of a Deposit account and all the benefits of a Current account, then a structure as shown in Figure 3.17, "Multiple Inheritance with the CashSave class." could be devised. We could call this class CashSave (or some other name like that).

Figure 3.17. Multiple Inheritance with the CashSave class.



Since Account contains the balance state, and CashSave is a type of Account, we do not want the balance state to be duplicated - having two different balance states! To fix this the CashSave class should only have one instance of its common indirect base class. This can be achieved by declaring the base class to be virtual.

```
class Current: public virtual Account
 {
   // etc..
 };

 class Deposit: public virtual Account {
     // etc..
 };

 class CashSave: public virtual Current,
                 public virtual Deposit{
   // etc.. virtual is not required here!
 };
```

In effect, we are declaring that if either the Current class or the Deposit class should be used in a multiple inheritance hierarchy, then they should share the same instance of the Account class with any other class that has virtual inheritance of Account.

So, this simple alteration to the parent results in a shared parent object. The only difficulty with this design is that the alteration must be made when the Current and Deposit classes are being designed. The programmer/designer must therefore be aware of child classes that are to be created in the future. This is not always possible as the marketing department may not have conceived of a "Cashsave" account until many years after the original design.

Now that we have an example of a virtual base class, remember to use a dynamic_cast in this case.

Advanced Note: Within the 'diamond problem' when a non-default constructor is not available in the virtual base class (A below), it is necessary to add a call to the virtual base class constructor from the ultimate child class (ABC below). This is because virtual inheritance means that the virtual base class (A) behaves as a direct parent of the child class (ABC) rather than being inherited via an indirect parent (AB or AC below). This would be implicit when a default constructor is available in the parent (A).

```
class A {  // virtual base class
public:
    //A() {}   // no default constructor
    A(int x)  {}
};

class AB : virtual public A {   // indirect parent
public:
    AB(int x): A(x)   { }
};

class AC : virtual public A {   // indirect parent
public:
    AC(int x): A(x) {}
};

class ABC : public AB, public AC  {  // ultimate child
public:
    ABC(int x): A(x), AB(x), AC(x)   {}  // note call to A(x)
};
```

Friend Functions

"C++ - Where only your friends can access your private parts"

--Unknown, 1991

In C++ normal access control has the keywords:

- private - only accessible within the class.
- protected - only accessible within the class and its derived classes.

- public - accessible anywhere.

A class can declare a function to be a friend, allowing that function to access private and protected members (friendship is granted! Not acquired.). For example:

```
class SomeClass {
   private:
     int x;
     friend void someFunction(SomeClass &);

   public:
     // the interface
};

// This function is somewhere!
void someFunction(SomeClass &a)  {
  a.x = 5;    //allowed!
}
```

It is important to note that someFunction() is not a member method of the class SomeClass. i.e. It does not have scope within the class SomeClass.

So if we tried:

```
SomeClass b;
b.someFunction();
```

This is illegal, as someFunction() is not a member method of the SomeClass class (and it is not public).

Friend functions are useful as:

- Friend functions avoid adding unnecessary public methods to the interface.
- Prevent states from having to be made public.

However, the overuse of friend functions can make the code very complex and hard to follow.

We can add all the methods of a class as friends of another:

```
class A {
  private:
    int x;
    friend class B;
  public:
    //methods here
};
```

Friendship is not inherited:

```
class B {
```

```
    x(A &a) {  //some method that is passed an object of A
      a.x++;  //fine
    }
};

class C: public B {  // class C is a child of B
    y(A &a){ //some method that is passed an object of A
      a.x++;  //illegal
    }
};
```

Friendship is not transitive:

```
class AA  {
  friend class BB;
};

class BB  {
  friend class CC;
};

class CC  {
  // Not a friend of AA
};
```

Static States of Classes

Each instance of a class has its own states. However, we also have static states allowing us to share a variable between every instance of a class. You can think of a static state as being a part of the class rather than the objects.

Here is a good example of the use of static states in the Account class.

```
1
2   // Static Member Example
3
4   #include<iostream>
5   #include<string>
6
7   using namespace std;
8
9   class Account {
10
11    protected:
12
13      static int nextAccountNumber; //1
14      int accountNumber;
15      float balance;
16      string owner;
```

```
17
18    private:
19
20      void construct(); //2
21
22    public:
23
24      Account(string owner, float aBalance);
25      Account(float aBalance);
26
27      virtual void display();
28      virtual void makeLodgement(float);
29      virtual void makeWithdrawal(float);
30  };
31
32  int Account::nextAccountNumber = 123456; //3
33
34  Account::Account(string anOwner, float aBalance):
35      balance(aBalance),
36      owner (anOwner) { construct(); } //4
37
38  Account::Account(float aBalance) :
39      balance(aBalance),
40      owner ("Not Defined") {construct(); }
41
42  void Account::construct()
43  {
44    accountNumber = nextAccountNumber++; //5
45  }
46
47  void Account::display(){
48    cout << "account number: " << accountNumber
49         << " has balance: " << balance << " Euro" << endl;
50    cout << "This account is owned by: " << owner << endl;
51  }
52
53  void Account::makeLodgement(float amount){
54      balance = balance + amount;
55  }
56
57  void Account::makeWithdrawal(float amount){
58      balance = balance - amount;
59  }
60
61  int main()
62  {
63    Account *a = new Account("John", 10.50);
64    Account *b = new Account("Derek", 12.70);
65
66    a->display();
67    b->display();
```

```
68   }
69
70
```

(The full source code for this example is here - **StaticMemberTest.cpp**)

❶ A state can simply be defined as static by placing the static keyword in front of its declaration.

❷ One of the problems with C++ and multiple constructors is that there is no way to call one constructor directly from the other (like in Java). To prevent us from having to duplicate code we can create a private method (I called it construct()) that is called directly from the actual constructors of the class. If any alterations need to be made to the construct() method then they will impact on all of the constructors.

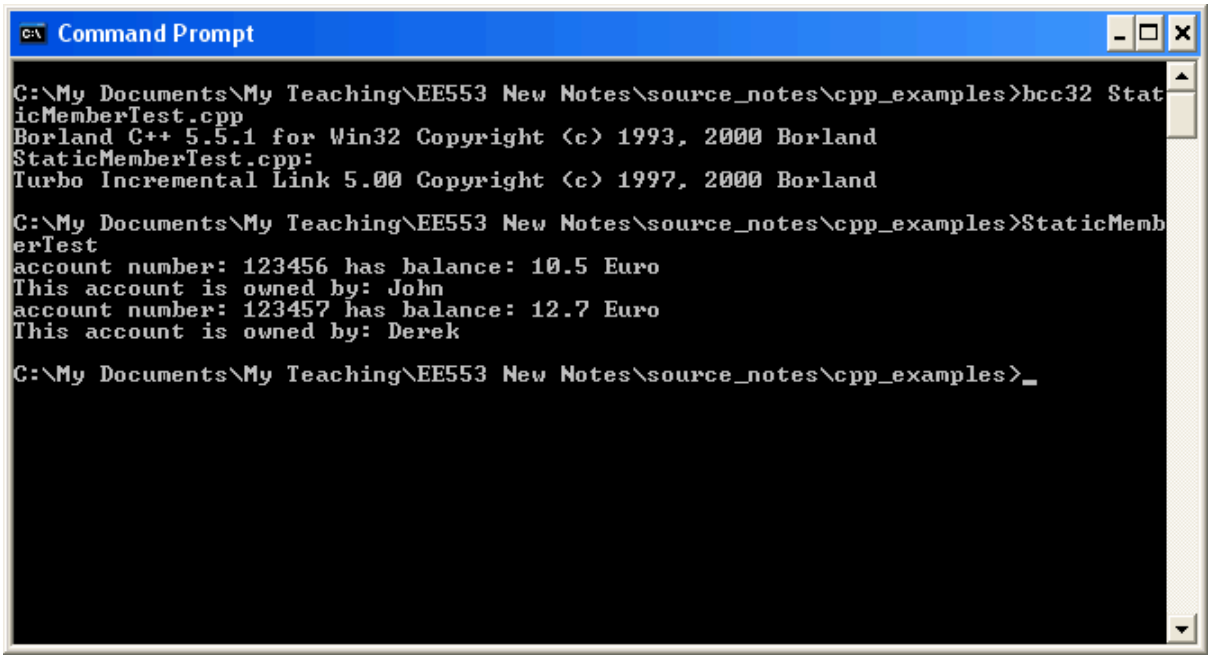
❸ In C++ we are required to define and associate a value with the static value in the implementation of the class. Note that the static keyword is not used again at this point.

❹ Note that in the two constructors the construct() method is called which sets the accountNumber value.

❺ The private construct() method sets the accountNumber state to the nextAccountNumber and increments the nextAccountNumber after the assignment. Since the static state is shared with every Account object is incremented on each and every object. This means that when every object is created it will have its own individual account number, without having to pass the value to the constructor.

The output of this example can be seen in Figure 3.18, “The Account class with static state example output.”.

Figure 3.18. The Account class with static state example output.



Static variables can be public, private or protected.

Static member methods can be defined in the same way (e.g. static int getNextAccountNumber();). Static member methods cannot access non-static states and they cannot be virtual, as they are related to the class and not the object.

Scope Resolution Operator

We have seen the use of the scope resolution operator (::) to explicitly specify the class identity of a method or state. We can also use the scope resolution operator to access global variables, for example:

```
int x;

 class A {
   int x;
   virtual void someMethod() {
     x++;    //increments the A::x
     ::x++;  //increments the global x
   }
 };
```

What about using overloading and hiding?.. Example:

```
class A {
   public:
     virtual int f(int);
 };

 class B: public A {
   public:
```

```
    virtual int f(char *);
};

int main() {
  B b;
  b.f(2);       //Error - B::f(char *) hides A::f(int)
  b.A::f(2);    //fine
  b.f("test");  //fine
}
```

Structs and Unions in C++

Structs in C++

The C programming language had the concept of a structure, grouping data together to form a complex data type (for example: complex numbers, Person record). C++ extends this concept by allowing methods to be associated with this data, so for example in C we could:

```
struct A {
  char* someString;
  int someint;
};

// use like
A testA;
A *ptrtoA = new A;

testA.someString = "something";
ptrtoA->someint = 4;
```

In C++ we can add methods:

```
struct Account {
    Account( ... );
    virtual void makeLodgement( ... ); //etc..
  private:
    int theAccountNumber;
    float theBalance;
};
```

Structs and Classes are very similar in C++; however, members in a struct are **public by default** and members in a class are private by default. The 'C' language only allows public member data within structs (there is no concept of private or protected in C).

We can construct any class from a struct and vice-versa. It is a question of style, however a class is more commonly used for a data structure that has associated methods.

Unions in C++

C++ has a structure that allows us to have different types of data within the same variable. A union is particularly memory efficient as it calculates the minimum amount of memory to store the largest element in the structure. Therefore, contained data overlaps in memory.

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

union StudentID {
     char* byName;    // memory for a pointer OR
     int byNumber;    // memory for an int
};

int main() {
   StudentID s;
   s.byNumber = 12345;
   cout << s.byNumber << endl;  //outputs 12345
   s.byName = "Hello";
   cout << s.byName << endl;   //outputs Hello
   cout << s.byNumber << endl; //outputs 4198928
}
```

Non-Virtual Methods

Omitting the virtual keyword declares a method as non-virtual. Simply speaking, declaring a method as virtual enables overriding, declaring a method as non-virtual disables overriding.

```
theAccount->display();
```

The display() method that is called depends on whether theAccount pointer references a generic Account, Deposit account or Current account (the dynamic type - sometimes called late binding).

So why are non-virtual methods required? If in the rare circumstance of declaring a class that you know will have no derived classes, you can use non-virtual methods. When virtual methods are used the compiler does not know for certain the type of an object until run-time, so the compiler inserts additional code to evaluate the dynamic type at run-time and call the correct method. Removing this feature allows slightly faster program execution. Non-virtual can also be used if for some reason you wish that a method cannot be overridden (e.g. the constructor of an object).

Most Object Oriented Programming languages only support virtual methods, but strangely enough **in C++ non-virtual is the default** setting!

So in the Virtual case:

```
...
```

```
class Account {
  public:
    virtual string getType() { return "Generic Account"; };
};

class Current: public Account {
  public:
    virtual string getType() { return "Current Account"; };
};

class Deposit: public Account {
  public:
    virtual string getType() { return "Deposit Account"; };
};

int main() {
  // Note that all pointers have the static type Account
  Account *a = new Account();
  Account *b = new Current();
  Account *c = new Deposit();

  cout << "Pointer a Displayed: " << a->getType() << endl;
  cout << "Pointer b Displayed: " << b->getType() << endl;
  cout << "Pointer c Displayed: " << c->getType() << endl;
}
```

(The full source code for this "virtual" example is here - **NonVirtualTest.cpp**) This program will result in the output:

```
Pointer a Displayed: Generic Account
Pointer b Displayed: Current Account
Pointer c Displayed: Deposit Account
```

Since the getType() method is virtual the getType() method most closely associated with the object will be called, not the getType() method associated with the pointer. Note that you only need to declare the method as virtual in the base class declaration and it will be virtual in all child classes, regardless if the virtual keyword is used or not. Redefining a virtual method is called overriding.

In the **non-virtual** case:

```
..

class Account {
  public:
    string getType() { return "Generic Account"; };
};
```

```
class Current: public Account {
  public:
    string getType() { return "Current Account"; };
};

class Deposit: public Account {
  public:
    string getType() { return "Deposit Account"; };
};

int main()
{
  Account *a = new Account();
  Account *b = new Current();
  Account *c = new Deposit();

  cout << "Pointer a Displayed: " << a->getType() << endl;
  cout << "Pointer b Displayed: " << b->getType() << endl;
  cout << "Pointer c Displayed: " << c->getType() << endl;
}
```

(The full source code for this "non-virtual" example is here - **NonVirtualTest2.cpp**) This program will result in the output:

```
Pointer a Displayed: Generic Account
Pointer b Displayed: Generic Account
Pointer c Displayed: Generic Account
```

Since the methods are now non-virtual the getType() associated with the pointer will always be called. So, the Account::getType() will always be called.

The destructor is always virtual as we require different actions in destructing an object, for example in:

- Current account - on destruction, send a statement.
- Deposit account - on destruction, calculate interest and then send a statement.

So, for example:

```
class Deposit :: public Account  {
    //states
  public:
    virtual ~Deposit(); //destructor - always virtual
};

Deposit::~Deposit() {
  //calculate interest
  //send statement.
}
```

Accessor/Mutators and Inline Functions

Accessor/Mutators

Do not remove methods for the sake of efficiency - use accessor/mutator methods/functions instead. For example:

```
class Account {
   protected:
     int accountNumber;
   public:
     // an example accessor/mutator method
     int getAccountNumber() { return accountNumber; }
     void setAccountNumber(int number) { accountNumber = number; }
};
```

This is better than making the accountNumber state public, retaining encapsulation. We can provide full access to a state through methods, rather than directly to the state by making it public, by providing an **accessor** and a **mutator**. For example for the accountNumber state in the Account class, these would look like

```
int getAccountNumber(); //accessor
void setAccountNumber(int newNumber); //mutator
```

The main advantage of this format is that access can be controlled to the accountNumber state, so, if additional functionality is required at a later stage, it can be achieved, without affecting the developers using the Account class.

The compiler may choose to 'inline' these methods depending on the compiler optimization settings. Inline is ignored with virtual methods. So, what does inline mean?

Inline Functions

We have looked at the use of macros before and their use in defining constants and their support roles in managing the single-definition rule. Macros can be used to define functions that are inlined by the compiler. Effectively, the compiler will copy and paste the macro function into the code at each location where it is called.

In the code below, the compiler will simply replace the square(5) with 5*5:

```
#define square(x) x*x

int main() {
     int y = square(5);
}
```

Unfortunately you have to be extremely careful with macros. Take for example the simple call to:

```
int main(){
     int y = square(3+4);
}
```

While you might expect 7x7 = 49, this code will return the value of 3 + 4 * 3 + 4 = 19, as it will be evaluated as 3 + (4 * 3) + 4 = 19 due to the rules of precedence. Macro functions are inlined and offer performance advantages, but also be careful with code 'bloat' as calling a large macro 100 times will create 100 copies of the macro in your code.

In my opinion using the **inline keyword** is much safer than using macros, particularly if you are passing compound arguments as above. Use macros for very simple functions only.

Depending on the optimisation options, the compiler will generally decide whether a function should be inlined or not. That said, the inline keyword suggests to the compiler that a particular function should be inlined. The accessors and mutators above are likely to be inlined given their brevity and the fact that the 'cost' of calling them as functions (stack operations etc.) is much greater than the cost of the function code to be executed.

It is also possible to force the compiler to inline a function, but this is not recommended unless you are certain that the function should be inlined.

Here is an example of both:

```
int square_forced(int)__attribute__((always_inline));

int square_forced(int x) {  //this will always be inlined
       return x*x;
}

int inline square_suggested(int x) {
       return x*x;
}

int main() {
     int y = square_forced(5);
     int z = square_suggested(5);
     cout << "The value of y is " << y << " and z is " << z << endl;
}
```

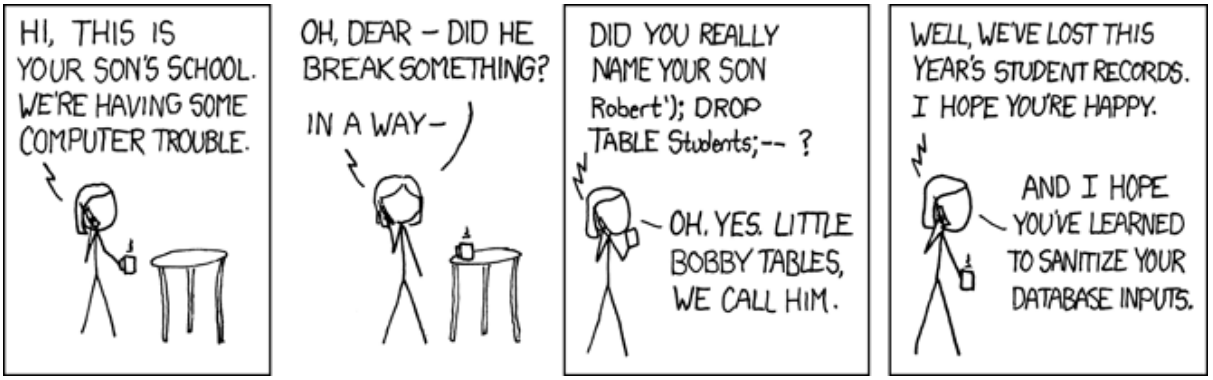
Which gives the expected output:

```
The value of y is 25 and z is 25
```

The forced version gives a warning on my C++ compiler that "always inlinable function might not be inlinable," which can be due to the optimization level specified in the build settings. The g++ compiler documentation states:

Generally, functions are not inlined unless optimization is specified. For functions declared inline, this attribute inlines the function independent of any restrictions that otherwise apply to inlining. Failure to inline such a function is diagnosed as an error. Note that if such a function is called indirectly the compiler may or may not inline it depending on optimization level and a failure to inline an indirect call may or may not be diagnosed.

My advice is to leave inlining to the compiler unless absolutely critically necessary.



APPENDIX A Fully Integrated Example

This section contains full implementations of the previous examples to allow you to compile and modify as required. The source code examples are at the bottom of the page.

The Account Class

Figure 3.19. The full working Account class.

```
/cygdrive/c/temp
$ C++ Account.cpp -o Account

molloyd@PORTALM /cygdrive/c/temp
$ ./Account
Account Details are - Account number: 1235 has balance: 1000
Enter amount to lodge: 500
Account Details are - Account number: 1235 has balance: 1500
Enter amount to withdraw: 1200.00
Account Details are - Account number: 1235 has balance: 300
Creating a second account
Account Details are - Account number: 1236 has balance: 30

molloyd@PORTALM /cygdrive/c/temp
$ ./Account
Account Details are - Account number: 1235 has balance: 1000
Enter amount to lodge: 200
Account Details are - Account number: 1235 has balance: 1200
Enter amount to withdraw: 1500
Withdrawal Failed!
Account Details are - Account number: 1235 has balance: 1200
Creating a second account
Account Details are - Account number: 1236 has balance: 30

molloyd@PORTALM /cygdrive/c/temp
$
```

The source code for this example is in **Account.cpp**

The Account and CurrentAccount Classes

This example extends the Account class to create a new class called CurrentAccount.

Figure 3.20. The full working Account and CurrentAccount classes.

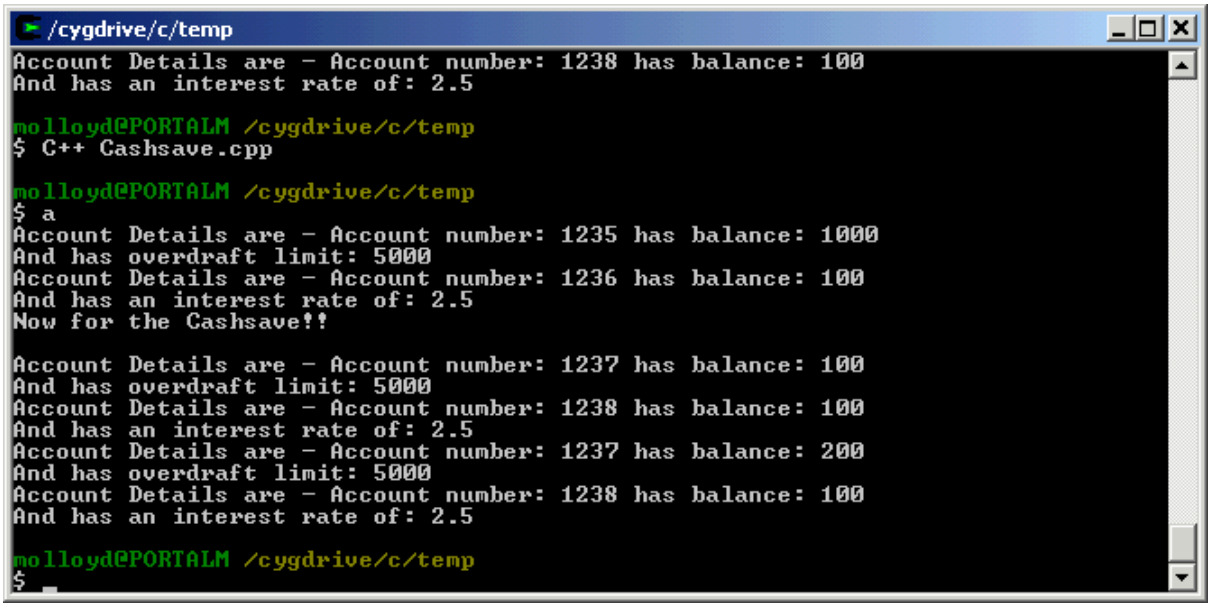
```
/cygdrive/c/temp
molloyd@PORTALM /cygdrive/c/temp
$ a
Account Details are - Account number: 1235 has balance: 1000
And has overdraft limit: 5000
Enter amount to lodge: 1000
Account Details are - Account number: 1235 has balance: 2000
And has overdraft limit: 5000
Enter amount to withdraw: 6000
Account Details are - Account number: 1235 has balance: -4000
And has overdraft limit: 5000

molloyd@PORTALM /cygdrive/c/temp
$ a
Account Details are - Account number: 1235 has balance: 1000
And has overdraft limit: 5000
Enter amount to lodge: 1000
Account Details are - Account number: 1235 has balance: 2000
And has overdraft limit: 5000
Enter amount to withdraw: 9000
Withdrawal Failed!
Account Details are - Account number: 1235 has balance: 2000
And has overdraft limit: 5000

molloyd@PORTALM /cygdrive/c/temp
$
```

The source code for this example is in **AccountAndCurrent.cpp**

Multiple Inheritance Example (with problems)

This incomplete example shows the use of multiple inheritance in the creation of a "Cashsave" class that has parent classes of both DepositAccount and CurrentAccount classes.

Figure 3.21. The incomplete Cashsave class example.



The source code for this example is in **Cashsave.cpp**

Problems with this example are:

- Multiple inheritance has caused the Account class to be instantiated twice, once as the parent of CurrentAccount and once as the parent of DepositAccount.
- I made a lodgement to the cashsave account, but, I have to specify which makeLodgement() method I wish to call - CurrentAccount::makeLodgement() or DepositAccount::makeLodgement() - now I have two balances!

The Multiple Inheritance Example Fixed

This example fixes the example in the section called "Multiple Inheritance Example (with problems)" for this specific banking example.

Figure 3.22. The full working CashSave class example.

```
/cygdrive/c/temp
Cashsave2.cpp:29: candidates are: Account::Account(float)
Cashsave2.cpp:25:                 Account::Account(const Account &)

molloyd@PORTALM /cygdrive/c/temp
$ C++ Cashsave2.cpp

molloyd@PORTALM /cygdrive/c/temp
$ a
Account Details are - Account number: 1235 has balance: 1000
And has overdraft limit: 5000
Account Details are - Account number: 1236 has balance: 100
And has an interest rate of: 2.5
Now for the Cashsave!!

Account Details are - Account number: 1237 has balance: 100
And has overdraft limit: 5000
Account Details are - Account number: 1237 has balance: 100
And has an interest rate of: 2.5
Account Details are - Account number: 1237 has balance: 200
And has overdraft limit: 5000
Account Details are - Account number: 1237 has balance: 200
And has an interest rate of: 2.5

molloyd@PORTALM /cygdrive/c/temp
$
```

The source code for this example is in **CashSave2.cpp**

The problem in the section called "Multiple Inheritance Example (with problems)" has been fixed by inserting the keyword virtual before Account in the definition of the CurrentAccount and DepositAccount classes.

A Final Working Version of the Cashsave example

This example adds additional functionality to the Cashsave class as defined in the section called "The Multiple Inheritance Example Fixed".

Figure 3.23. The final working CashSave class example.

```
/cygdrive/c/temp
Account.exe                 Tortuga2.wmv                    multiple.dsp
Account.gif                 a.exe                           multiple.dsw
AccountAndCurrent.cpp       current\ archive.pst            multiple.ncb
CashSave.cpp                deck.prn.pdf                    multiple.opt
CashSave2.cpp               dx                              multiple.plg
CashSave3.cpp               dx_sdk                          surf99mobile
CurrentAccount.gif          ee_dcu_white_background.gif     test.cpp
Debug                       gcc.gif                         vstudio6sp5
Javaman                     hello.cpp
Microsoft\ Office\ XP.exe   hello.exe

molloyd@PORTALM /cygdrive/c/temp
$ C++ Cashsave3.cpp

molloyd@PORTALM /cygdrive/c/temp
$ a
Account Details are - Account number: 1235 has balance: 1200
And has overdraft limit of: 5000 and deposit interest rate of: 2.5
Make widthdrawal of 200
Calculate interest
Account Details are - Account number: 1235 has balance: 1025
And has overdraft limit of: 5000 and deposit interest rate of: 2.5

molloyd@PORTALM /cygdrive/c/temp
$
```

The source code for this example is in **CashSave3.cpp**

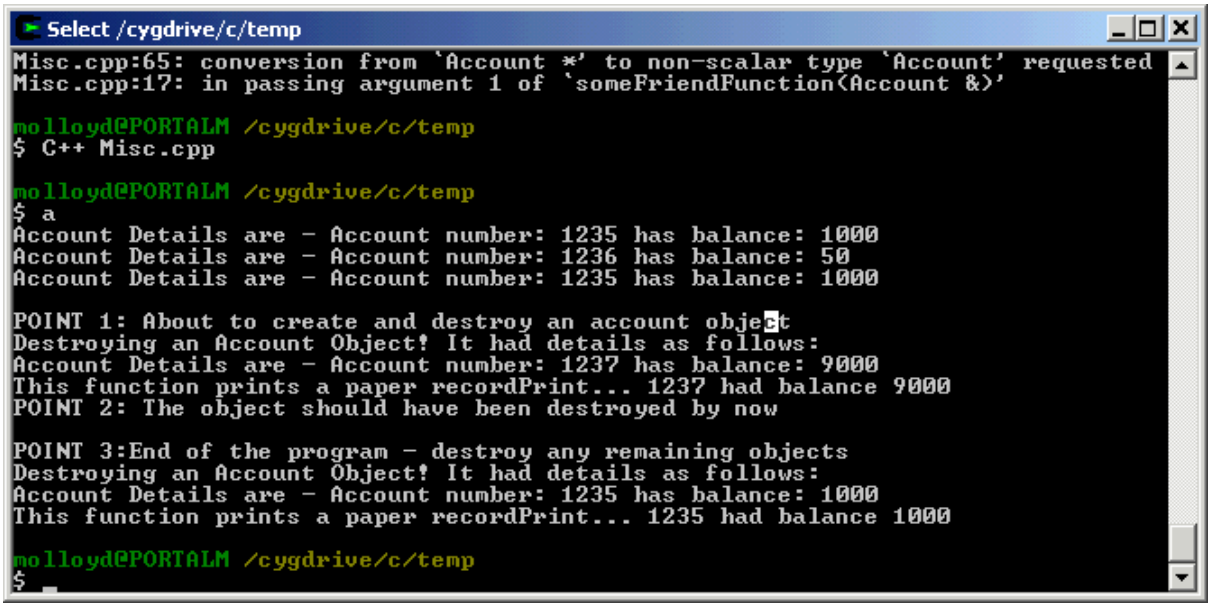
The modifications are:

- A more correct display() method.
- Corrections to the access modifiers of overdraftLimit and interestRate states.

This example shows that the inherited methods from both CurrentAccount and DepositAccount are working correctly for this new child CashSaveAccount class.

Friend Methods, Pointers to Objects and Destructors

Figure 3.24. Friend Methods, Pointers to Objects and Destructors



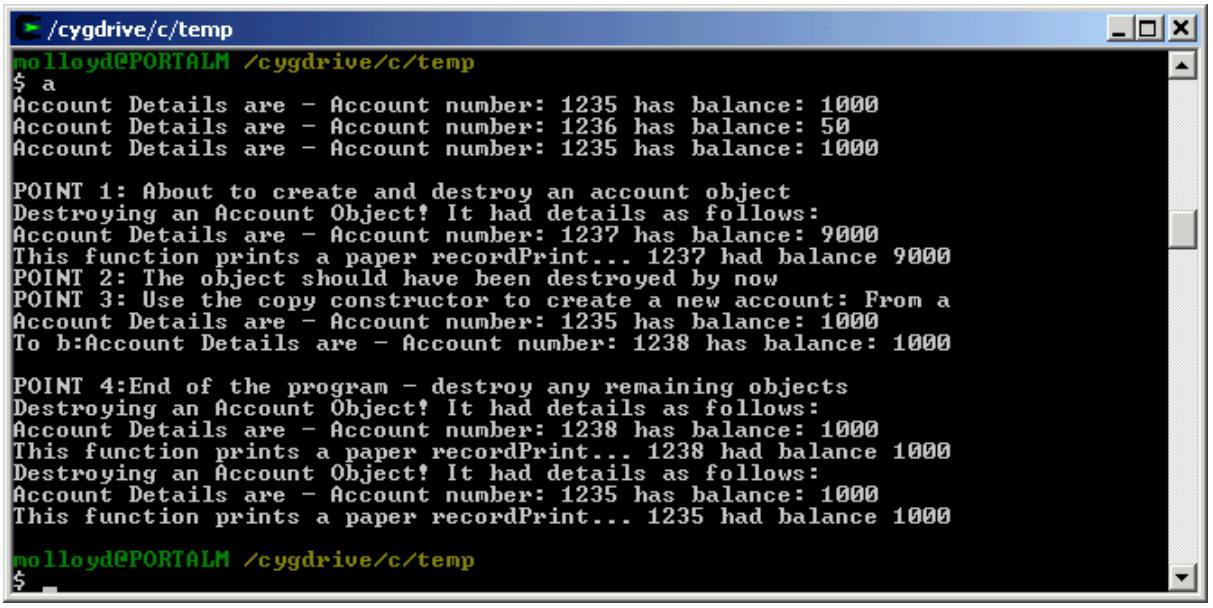
The source code for this example is in **MiscExamples.cpp**

Watch very carefully in the code as the objects go out of scope. See how the destructor is called and see the way that the destructor calls a friend method - could be an external printing method. See the way that this friend method has full access to the private states of the object. See also the way that the object that I created anonymously goes out of scope without having its destructor called. The memory is lost. It is only regained for allocation by the operating system when the program has run to completion.

Now with a Copy Constructor!

This example adds a copy constructor with specific behaviour to the example given in the section called "Friend Methods, Pointers to Objects and Destructors".

Figure 3.25. A copy constructor example.

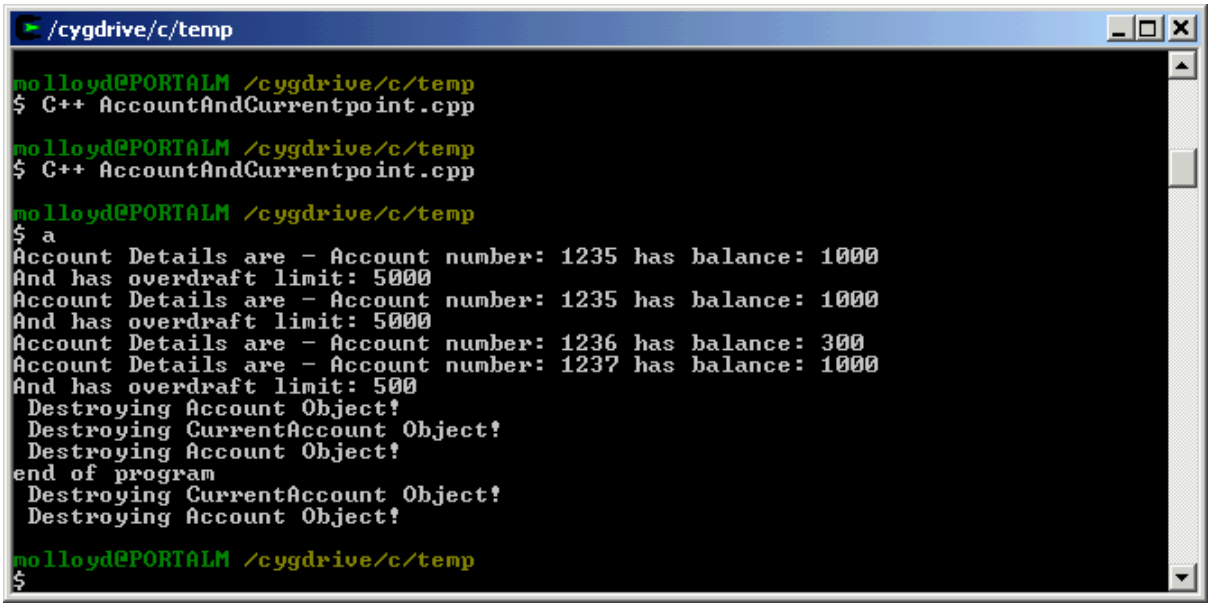


The source code for this example is in **MiscExamples2.cpp**

Destructors and Arrays.

This example adds a destructor with specific behaviour to the example given in the section called "Now with a Copy Constructor!", demonstrating destructors when using arrays of objects and static/dynamic types.

Figure 3.26. Destructors and Arrays Example.



The source code for this example is in **AccountAndCurrentpoint.cpp**

APPENDIX B Worked Examples

There is not now and never will be a language in which it is the least bit difficult to write bad programs.

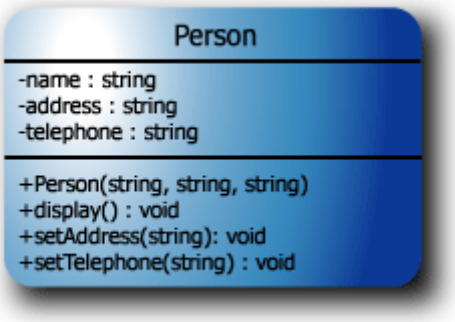
--Unknown, 1991

This exercise will build a set of classes to describe the staff/student hierarchy in DCU. You should follow the various exercises in order, as one is dependent on the next.

The Person class Exercise

Task: Write a generic Person class, as illustrated in Figure 3.27, "The Person class." that builds the basic functionality of the general person role in DCU.

Figure 3.27. The Person class.



The output of my version can be seen below, where the setTelephone() method is called:

```
C:\Temp>Exercise1

Name: Derek Molloy
Address: DCU, Glasnevin, D9
Telephone: 7005355

Setting Telephone Number

Name: Derek Molloy
Address: DCU, Glasnevin, D9
Telephone: 01-7005355
```

Solution: My version can be seen in **Exercise1.cpp**. It is at the bottom of the page - Please do not look at the solution until you have had a good attempt at the exercise.

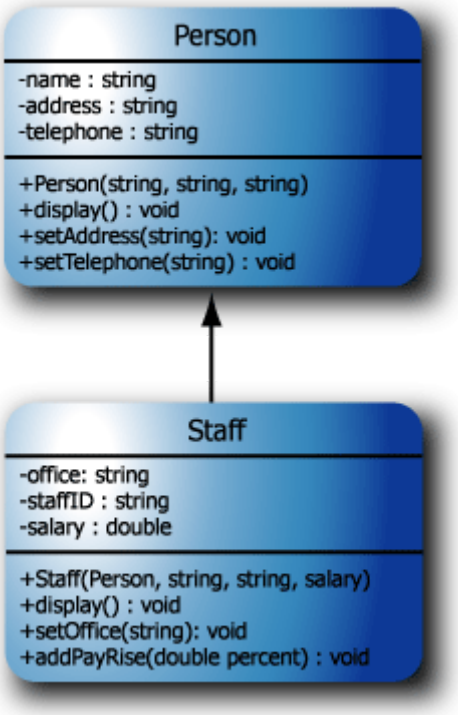
The Staff class Exercise

"Within C++, there is a much smaller and cleaner language struggling to get out."

--Bjarne Stroustrup

Task: Update the previous exercise to create a child of the Person class, called Staff. This child class should have the functionality as illustrated in Figure 3.28, "The Person and Staff class hierarchy." that builds the basic functionality of the general person role in DCU.

Figure 3.28. The Person and Staff class hierarchy.



The output of my version can be seen below, where the addPayRise() method is called to give a pay rise of 5%:

```
C:\Temp>Exercise2
Name: Derek Molloy
Address: DCU, Glasnevin, D9
Telephone: 7005355

Creating Staff Member:

Name: Derek Molloy
Address: DCU, Glasnevin, D9
```

```
Telephone: 7005355
Office: S356
ID Number: 94971056
Salary: 12000 Euro

Payrise of 5%:

Name: Derek Molloy
Address: DCU, Glasnevin, D9
Telephone: 7005355
Office: S356
ID Number: 94971056
Salary: 12600 Euro
```

Solution: My version can be seen in **Exercise2.cpp**. Please do not look at the solution until you have had a good attempt at the exercise.

Chapter 4 - Standard Libraries

Templates

In this section we will look at the use of the C++ standard libraries in more detail, including a brief discussion on templates and the use of the Standard Template Library (STL).

What are templates?

C++ templates are functions that can handle different data types without requiring that we write separate code for each of them. To perform a similar operation on several kinds of data types, a programmer need not write different versions by overloading a function. Instead, the programmer can write a C++ template based function that will work with all data types.

There are two types of templates in C++, function templates and class templates. First off, we will examine function templates.

Function Templates

There are many occasions where we might need to write the same functions for different data types. A simple example can be the addition of two variables, for example, of the int, float or double types. The function requirement will be to return the summed value with the correct return type that is based on the type of the input variables. If we start writing one function for each of the data types, then we will end up with 4 to 5 different functions that all contain very similar code. This goes against everything we have learned so far in this module.

C++ templates are a useful facility in these situations. When we use C++ function templates, only one function "signature" needs to be created. The C++ compiler will automatically generate the required functions for handling the individual data types. This makes programming much easier - Here is an example:

Example: Here is a short example of an add() function. If the requirement is to use this add() function for both int and float types, then two functions are to be created, one for each of the data types (overloading).

```
int add(int a,int b) { return a+b;} // without using function templates
float add(float a, float b) { return a+b;} // without using function templates
```

If there were more data types to be handled, more functions would have to be added. Using C++ function templates for this task makes the entire process a lot easier, by reducing the code to a single C++ function template. Here is the function template code for the same add() functions:

```
template <class T>  //The class keyword here means any
                    //standard OR user-defined type
 T add(T a, T b)    //C++ function template example
 {
   return a+b;
 }
```

Now (as before), logically this example is a bit simplistic as the "+" operator does the same thing anyway -- however, it is a good start. C++ function templates can be used wherever the same functionality has to be performed with a number of data types. Though very useful, experience tells us that lots of care should be taken to test the C++ template functions during development. A well written C++ template will go a long way in saving development time.

Templates are different from classes though -- when you instantiate a template the compiler has to see its definition before it can generate its source code. The usual approach is to define the entire template in a header file and then #include it in every source file that uses this template, i.e., do not separate the template into a .h and .cpp file. This brings up an (arguably) interesting point, since a class template must be in a header file it means that the source code is always exposed, even if you are selling your product to an untrusted client.

Class Templates

Class templates are often used to build type-safe containers. Class templates allow us to create a class that works with/on any user-defined class (type), so the obvious application is for advanced storage. This functionality is provided by C++ class templates, also called parameterized types, and the technique is called generic programming. Here is a short example of a made-up class called Storage that provides basic storage of values of a user-defined type:

```
// First Template Example
// by Derek Molloy - Template Example

#include<iostream>
using namespace std;

template<class T>
class Storage{
  T values[100]; // can store 100 values of type T (quite basic)

  public:
    T& operator [](int index){
      return values[index];
    }
};

int main(){
  Storage<int>   intArray;
  Storage<float> floatArray;
```

```
    for (int i=0; i<10; i++){
      intArray[i] = i * i;
      floatArray[i] = (float)i/2.1234 ;
    }

    for (int i=0; i<10; i++){
      cout << " intArray value   = " <<   intArray[i]
        << " floatArray value = " << floatArray[i] << endl;
    }
  }
```

The full source code for this example is listed in **ClassTemplates1.cpp**

This first example is a little basic, as it is fixed to contain only 100 elements of the user-defined type. We can remove this hard coding, by rewriting the code to:

```
// Second Template Example
// by Derek Molloy - More advanced template example.

#include<iostream.h>
using namespace std;

template<class T, int size>
class Storage{
  T values[size]; // can store size values of type T

  public:
    T& operator [](int index){
      return values[index];
    }
};

int main(){
    Storage<int,10> intArray;
    Storage<float,20> floatArray;

    for (int i=0; i<10; i++) {
      intArray[i] = i * i;
      floatArray[i] = (float)i/2.1234 ;
    }

    for (int i=0; i<10; i++) {
      cout << " intArray value = " << intArray[i]
           << " floatArray value = " << floatArray[i] << endl;
    }
  }
```

The full source code for this example is listed in **ClassTemplates2.cpp**

That is not quite everything to do with templates:

- **Inheritance:** Class templates can inherit properties from other class templates or even non-templates.
- **Friends:** We can use friend functions with templates.
- **Static:** A class template can have static members, but these static members are shared amongst each specialisation of the class template. For example, if you had a class template as above and you used it for int and float, then there would be one static member for the int specialisation and one for the float specialisation.

Standard Template Library (STL)

STL Introduction

The Standard Template Library (STL) uses C++ templates to implement a standardised extensible and reusable code base for the development of complex software applications. STL aims to encourage the programmer to build code that is solid, robust and reusable. STL is extensible, allowing the programmer to add their own containers and algorithms. All C++ library entities are declared or defined in one or more standard headers. To make use of a library entity in a program, write an #include directive that names the relevant standard header (out of the 50+ available).

As discussed before, you can include the contents of a standard header by naming it in an include directive. For example:

```
#include <iostream>   // include I/O facilities
```

All names other than operator delete and operator new in the C++ library headers are defined in the std namespace, or in a namespace nested within the std namespace. You refer to the name cin, for example, as std::cin. Alternatively, you can write the declaration:

```
using namespace std;
```

which brings all library names into the current namespace. If you write this declaration immediately after all include directives, you hoist the names into the global namespace. Unless specifically indicated otherwise, you may not define names in the std namespace, or in a namespace nested within the std namespace.

The downside of STL is that it is not suitable for very low power embedded devices -- e.g., IoT sensor leaf nodes -- as there is significant overhead in the use of templates and iterators. Importantly, this includes the use of classes such as the std::string which should be avoided on such low-power processors. For devices such as the RPi or BeagleBone this is not an issue as the program size cost is of the order of 1-2MB with respect to standard C strings.

The components of STL can be described and categorised as follows:

- **Containers:** Data structures with memory management capability.
- **Iterators:** Logic that binds containers and algorithms together.

- **Algorithms:** Provide a mechanism for manipulating data through the iterator interface.

STL Containers

In previous exercises we have examined the use of an array for the storage of objects. The array discussed is a container, but one that lacks advanced memory management capabilities. For example, if we had an array of 10 elements and the middle one needed to be removed, how would the array be altered to fill this gap, remember this gap, remove this gap? How could we extend it if we needed to add 12 elements? Well, more than likely, the programmer would have to write the logic for these memory management routines.

Containers are data structures that contain memory management capabilities. There are two categories of containers: sequential containers and associative containers. Some examples of sequential containers are:

- **vector:** A flexible array of elements.
- **deque:** A dynamic array that can grow at both ends.
- **list:** A doubly linked list.

STL vectors are a good solution to the array problem just discussed. They are as easy to declare as an array and are automatically dynamically resized during program execution.

Associative containers associate a value with a name and some examples are:

- **map:** A collection of name-value pairs, sorted by the keys value.
- **set:** A collection of elements sorted by their own value.
- **multimap:** Same as a map, only duplicates are permitted.
- **multiset:** Same as a set, only duplicates are permitted.

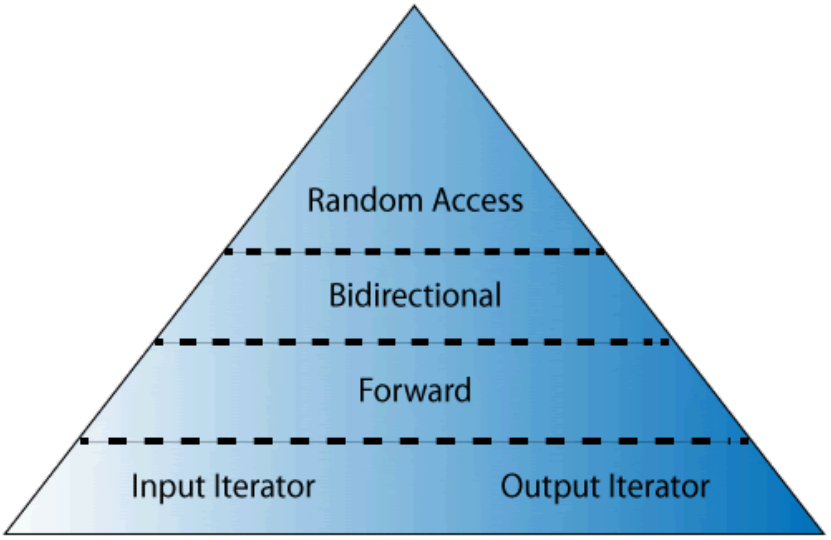
STL Iterators

The container 'contains' the elements together, and the iterator provides a mechanism for traversing the different types of those containers. Even though each container may have a completely different data structure, the iterator can provide the level of abstraction necessary to build a generic interface to traverse any type of container.

- Input iterators
- Output iterators
- Forward iterators
- Bidirectional iterators
- Random iterators

Figure 4.1 relates these iterators by their relative complexity.

Figure 4.1. STL Iterator Categories



Iterator Categories	Provider
Input	istream
Output	ostream
Forward	
Bidirectional	list, map, multimap, set
Random Access	array, vector, deque

Iterators are abstract, requiring a class that implements the methods for use as the iterator. Input and Output iterators are the simplest form of iterators (being simple iterator interfaces to the iostream classes, with the random access iterator being the most complex, and by definition, it has implemented the other iterator categories.

STL Algorithms

Algorithms provide a means of accessing and manipulating the data in containers using the iterator interface. The algorithms can even implement a for_each loop.

The algorithms can be classified as:

- Non-modifying algorithms: e.g., for_each, count, search ...
- Modifying algorithms: for_each, copy, transform ...
- Removing algorithms: remove, remove_if, ...
- Mutating algorithms: reverse, rotate, ...
- Sorting algorithms: sort, partial_sort, ...
- Sorted range algorithms: binary_search, merge ...
- Numeric algorithms: accumulate, partial_sum ...

A for_each example

The for_each loop condenses the for loop into a single line. This single line iterates through the vector and executes a function at every element within the vector. The function outputFunction receives the element type the vector was specialised with at

compile time. The function can do anything, but in this case, it simply prints to the standard output.

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

// declare a simple function that outputs an int
void outputFunction(int x){
     cout << x << endl;
}

int main(void){
     int x;
     vector<int> vect;  // declare a vector container of ints

     cout << "Please enter a list of numbers:" << endl;
     while(cin >> x){
       vect.push_back(x);
     }

     sort(vect.begin(), vect.end());   // sort the array
     cout << endl << "Sorted output of numbers:" << endl;

     // loop through vector with for_each loop and execute
     // outputFunction() for each element
     for_each(vect.begin(), vect.end(), outputFunction);
}
```

This can be applied to an object of our own type - an example is listed in **for_each_fish.cpp** and reproduced in the next example. This is a basic example of the use of vectors with your own type. If you wish to use STL algorithms then your type must provide certain additional functionality to allow you to define the behaviour of your type with certain operations, in particular '<' and '=='. an example is given in **for_each_fish_with_sort.cpp** that demonstrates the use of the required overloaded operators. Note that the const after a method declaration states that this method can be called on a constant object, as it bans the method from modifying the state of Fish through compiler checks ("…. in read-only structure" error).

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;

class Fish
{
```

```
	string name;
public:
	Fish(string);
	virtual void display();
};

Fish::Fish(string nm): name(nm) {}

void Fish::display()
{
	cout << "I am a fish and my name is " << name << endl;
}

void outputFunction(Fish x)
{
	x.display();
}

int main(void)
{
	string x;
	vector<Fish> vect;  // declare a vector container of Fish

	cout << "Please enter five Fish:" << endl;
	for(int i=0; i<5; i++)
	{
	cin >> x;
	vect.push_back(Fish(x));
	}

	// loop through vector with for_each loop and execute
	// outputFunction() for each element

	for_each(vect.begin(), vect.end(), outputFunction);
}
```

And with the sort function, the overloaded operators need to be added:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>

using namespace std;

class Fish
{
	string name;
```

```
public:
    Fish();
    Fish(string);
    virtual void display();

    bool operator < (const Fish & f) const {
      return (name < f.name ? true : false ); }

    bool operator == (const Fish & f) const {
      return (name == f.name ? true : false ); }
};

...

int main(void)
{
      string x;
      vector<Fish> vect;  // declare a vector container of Fish

      ...

      sort(vect.begin(), vect.end());
      for_each(vect.begin(), vect.end(), outputFunction);
}
```

Functors

What is a functor, or function object? Essentially, it is a class declared with the () operator overloaded. First, we declare a class. Next, we overload the () operator with as many parameters as we want, and then we simply instantiate it. Why would we do something so complicated? Bjarne Stroustrup, the creator of C++, says that, "Function objects often execute faster." Since C++ was developed with robustness and efficiency in mind, the creators of STL felt that the use of functors would be an ideal way to maintain the performance for the algorithms implemented.

Implementing the previous example using functors:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// declare a function object using a struct
struct func  {
  void operator()(int x) {
    cout << x << endl;
  }
```

```
};

int main(void) {
  int x;
  vector<int> vect;  // declare a vector container of ints

  cout << "Please enter a list of numbers:" << endl;
  while(cin >> x){
    vect.push_back(x);
  }
  sort(vect.begin(), vect.end());   // sort the array
  cout << endl << "Sorted output of numbers:" << endl;

  // loop through vector with for_each loop and execute the function
  //  object for each element
  for_each(vect.begin(), vect.end(), func());
}
```

So in this example the function object funct() was passed to the for_each() algorithm, just like an object! This is a very useful facility in C++. as we can use the same for_each() algorithm for an unlimited number of applications.

Finally, for completeness, we can implement the previous example using functors and templates:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// declare a function object using a template
template<class T> class func  {
  public:
    void operator()(T x){
      cout << x << endl;
    }
};

int main(void){
  int x;
  vector<int> vect;  // declare a vector container of ints

  cout << "Please enter a list of numbers:" << endl;
  while(cin >> x){
    vect.push_back(x);
  }

  sort(vect.begin(), vect.end());   // sort the array
  cout << endl << "Sorted output of numbers:" << endl;
```

```
    // loop through vector with for_each loop and execute the function
    // object for each element
    for_each(vect.begin(), vect.end(), func<int>());
  }
```

Smart Pointers

Smart pointers were introduced in C++11 (with further changes in C++14) and will work with all modern C++ compilers. Smart pointers provide a structured way to manage memory, which is absent with regular pointers, and is why they have gained popular usage.

Effectively, smart pointers are template classes that use overloading so that they behave like regular pointers. Take for example, the following function call f() that returns a pointer:

```
T * f(); // a function that returns a pointer.
```

This is not very well defined -- it doesn't tell if this is the only version of this pointer, and thereby it is safe for you to use delete on the pointer object, or if it is one of many shared pointers and using delete could have catastrophic impacts on your code.

Smart pointers have the form:

```
unique_ptr<T> & f(); // Only one possible pointer that cannot be copied
shared_ptr<T> f();  // Multiple copies of this pointer exist
```

To use smart pointers you need to include the memory header file, i.e.,

```
#include <memory>
```

We will look at these pointers and another "weak" pointer in the following sections using examples.

The Unique Pointer

The unique pointer is a smart pointer that can never be copied -- only one copy can ever exist. It is created just like you would any other template object.

The following example has a demonstration class Student. Please focus on the main() function and the test() function below and review the inline comments.

```
// Note that this test function takes a reference to a smart pointer. You
// cannot pass by value, as this would try to create a copy of the unique pointer.
void test(unique_ptr<Student> & ptr){
```

```
        cout << "Function received smart pointer to object " << ptr->getName() << endl;
}

int main() {
        // C++11 allows creation of unique pointer
        unique_ptr<Student> p(new Student("Joe", 1234));
        test(p);

        // C++17 allows another template function called make_unique, which also
        //  calls the constructor of Student.
        //  here I am using the auto type to set the type automatically for q
        auto q = make_unique<Student>("Jack",1235);
        test(q);

        // To destroy the object and replace the object you use reset
        //  The object "Joe" will be destroyed at this point
        p.reset(new Student("Derek", 1236));

        // We cannot copy a unique pointer, but we can move it
        //  auto r = p; // copy is not permitted for unique pointers
        cout << "Moving p to a new pointer r" << endl;
        auto r = move(p);
        // At this point p is now null -- i.e., not pointing at an object
        //  The pointer r is now pointing at Derek

        // To destroy the object at q
        cout << "Destroying the object at pointer q" << endl;
        q.reset();

        // You can use release() to release the object from managed memory
        //  but the destructor is not called -- e.g., q.release();

        cout << "End of main() " << endl;
}
```

This example program will give the following output:

```
Constructor called on Joe
Function received smart pointer to object Joe
Constructor called on Jack
Function received smart pointer to object Jack
Constructor called on Derek
Destructor called on Joe
Moving q to a new pointer r
Destroying the object at pointer q
Destructor called on Jack
End of main()
Destructor called on Derek
```

You will notice that the test function is called on p and q, which are created in different ways. You will also see that the line of code r = q; is not permitted and will cause a compiler error as unique pointers may not be copied. Notice also that the destructor is automatically called on the “Derek” object smart pointer when it goes out of scope.

The Shared Pointer

The shared pointer behaves just like the unique pointer except that you can create copies. The shared pointer provides basic garbage collection functionality. The shared pointer is the most commonly used smart pointer as it makes memory management relatively easy and you can have multiple pointers to the same object.

```
#include <iostream>
#include <memory>   // required for smart pointers
#include <string>
using namespace std;

class Student{
private:
        string name;
        int id;
public:
        Student(string name, int id): name(name), id(id) {
                  cout << "Constructor called on " << name << endl; }
        virtual void display();
        string getName() { return name; }
        int getID() { return id; }
        ~Student() { cout << "Destructor called on " << name << endl; }
};

void Student::display(){
        cout << "A student with name " << name << " and id " << id << endl;
}

void test(shared_ptr<Student> & ptr){
        cout << "[ Function received smart pointer to object " << ptr->getName() << endl;
        // Display the reference count for the object pointed to by ptr
        cout << "[ There is(are) " << ptr.use_count() << " reference(s) to the object" << endl;
}

int main() {
        // C++11 on allows you to create a shared pointer to an object of the class Student
        shared_ptr<Student> p(new Student("Joe", 1234));
        test(p);

        // C++17 allows another template function called make_shared, which also
        //  calls the constructor of Student.
        //  here I am using the auto type to set the type automatically for q
        auto q = make_shared<Student>("Jack",1235);
        test(q);

        // To destroy the object and replace the object you use reset
        //  The object "Joe" will be destroyed at this point
        p.reset(new Student("Derek", 1236));

        // We can copy a shared pointer
        cout << "Creating a copy of p to a new pointer r" << endl;
        auto r = p; // copy is permitted for shared pointers
        test(r);

        // To destroy the object at q
        cout << "Destroying the object at pointer q" << endl;
        q.reset();
        // You can use release() to release the object from managed memory
        //  but the destructor is not called -- e.g., q.release();
```

```
        cout << "End of main() " << endl;
}
```

The output of this code is as follows:

```
Constructor called on Joe
[ Function received smart pointer to object Joe
[ There are 1 references to the object
Constructor called on Jack
[ Function received smart pointer to object Jack
[ There are 1 references to the object
Constructor called on Derek
Destructor called on Joe
Creating a copy of p to a new pointer r
[ Function received smart pointer to object Derek
[ There are 2 references to the object
Destroying the object at pointer q
Destructor called on Jack
End of main()
Destructor called on Derek
```

Notice that the reference count was 2 the when the smart pointer r was passed to the function.

Also notice that the objects were destroyed correctly. There are exactly three constructors being called and exactly three destructors being called automatically, even though at one stage there were two pointers to the same object.

The Weak Pointer

The weak pointer is a special type of shared pointer that is not counted in the reference count, which can be useful if you need a pointer that does not affect the memory management of the object. A weak pointer is created from a shared pointer

The following is an example that uses the class from the shared pointer example.

```
int main() {
        shared_ptr<Student> p(new Student("Joe", 1234));

        // Make a copy of p
        auto q = p;
        cout << "The reference count is now " << p.use_count() << endl;

        // Make a weak pointer r
        auto r = weak_ptr<Student>(p);
        // Note there is no increase in the reference count
        cout << "The reference count is now " << p.use_count() << endl;

        // To use the weak pointer you must obtain a 'lock' and extract the shared
        //  pointer. This has the effect of creating another reference!
```

```
        auto temp = r.lock();
        test(temp);

        cout << "End of main() " << endl;
}
```

This code gives the following output:

```
Constructor called on Joe
The reference count is now 2
The reference count is now 2
[ Function received smart pointer to object Joe
[ There is(are) 3 reference(s) to the object
End of main()
Destructor called on Joe
```

Why would you use these? This is typically used when you want to prevent the creation of circular references.

For example, if you had a class called Student and a class called Module. The Student class might have pointers to Module objects that a student is registered for, and the Module object might have pointers to the students that are registered for that module. This condition would prevent memory management from ever releasing the memory for a Module object or Student objects, as the reference count could never be zero.

In this case you could use a weak pointer from the Module to each Student object, which would break the circular reference.

My advice is that you should use a shared_ptr by default and only use the weak_ptr if you need to break a circular reference. The unique_ptr is used only when you need a one-to-one pointer-object relationship.

Custom Deleters

Custom deleters allow you control over how the object attached to a smart pointer is destroyed, which can be very helpful in managing memory. The following example adds in a custom deleter function, which expects a pointer to a Student object. We have access to its methods in this custom deleter function.

```
#include <iostream>
#include <memory>   // required for smart pointers
#include <string>
using namespace std;

class Student{
private:
        string name;
        int id;
public:
        Student(string name, int id): name(name), id(id) {}
        string getName() const { return name; }
```

```
        int getID() const { return id; }
        virtual ~Student() { cout << "Student destructor called " << endl; }
};

void deleter(const Student *s){
        cout << "[ Custom deleter for the Student class" << endl;
        cout << "[ Doing something for " << s->getName() << " object " << endl;
        cout << "[ For example calc grades, print transcript " << endl;
        delete s;
        cout << "[ The destructor should have been called by now " << endl;
}

int main() {
        shared_ptr<Student> p(new Student("Joe", 1234), deleter);
        p.reset();
        cout << "End of main() " << endl;
}
```

Note that the custom deleter is attached to the smart pointer when it is created. You are not able to use the make_shared call to add in the custom deleter.

The output from this example is as follows, where you can see that the custom deleter function was invoked when the p.reset() call was performed:

```
[ Custom deleter for the Student class
[ Doing something for Joe object
[ For example calc grades, print transcript
Student destructor called
[ The destructor should have been called by now
End of main()
```

Note that the custom deleter is now responsible for deleting the object, which is performed by the call to delete s; If you omit this line then the Student destructor will not be called.

Move Semantics

Examine the following section of code, where we pass an object to a function by value as follows:

```
Student function(Student a) {
   // do something
   return a;
}

main(){
    Student x;
    Student y = function(x);
}
```

When function() is called the Student object x is passed by value and a new object, called a (with scope local to the function) is created using the Student class' copy

constructor. When the object a is returned by the function by value a second temporary object is constructed. This could involve a substantial computational cost.

The move semantic allows the data to be associated with the copy without having to perform a byte-by-byte copy. This is achieved using an **rvalue reference** of the form:

```
Student & p // typical lvalue reference (left side reference)
Student && p // new rvalue reference (right side reference)
```

The only difference between the rvalue reference and the lvalue reference is that an rvalue reference can be moved and an lvalue reference cannot.

The following is an example to use the STL move() and swap() template functions that take advantage of the new rvalue reference, as follows:

To implement this in your own class you need your own move constructor. If there is no move constructor then the copy constructor is called instead. The move constructor must have a **noexcept** keyword which prevents exceptions from leaving the object in an undefined state.

```
#include <iostream>
#include <memory>   // required for smart pointers
#include <string>
#include <utility>  // required for move and swap
using namespace std;

class Student{
private:
        string name;
        int id;
public:
        Student(string name, int id): name(name), id(id) {
            cout << "Constructor called" << endl; }
        Student(Student &&) noexcept;
        virtual void display();
        string getName() const { return name; }
        int getID() const { return id; }
        ~Student() { cout << "Destructor called for " << name << endl; }
};

Student::Student(Student && source) noexcept {  //Move constructor
        cout << "Move constructor called " << endl;
        name = std::move(source.name);
        id = source.id;
}

void Student::display(){
        cout << "A student with name " << name << " and ID " << id << endl;
}

int main() {
        Student a = Student("Derek", 1234);
```

```
        Student b = move(a);
        b.display();
        cout << "End of main()" << endl;
}
```

When we execute this code you will see the output:

```
Constructor called
Move constructor called
A student with name Derek and ID 1234
End of main()
Destructor called for Derek
Destructor called for
```

Interestingly, there was only ever one call to the Student constructor, despite it being passed to the move() function by value. If we examine the STL move function you will see the following definition, which includes the use of the rvalue:

```
std::move(_Tp && _T)
```

It is also possible to add an assignment operator that uses the move semantic, so that a call to:

```
c = move(b);
```

Would use the assignment move operator. The operator for the student class would have the following format:

```
class Student{
private:
        string name;
        int id;
public:
        ...
        Student(Student &&) noexcept;
        Student & operator = (Student &&) noexcept;
        ...
};

Student::Student(Student && source) noexcept {  //Move constructor
        cout << "Move constructor called " << endl;
        name = std::move(source.name);
        id = source.id;
}

// Move assignment operator.
Student & Student::operator = (Student && source) noexcept {
        cout << "Move assignment operator performed " << endl;
        name = std::move(source.name);
```

```
        id = source.id;
        return *this;
}

void Student::display(){
        cout << "A student with name " << name << " and ID " << id << endl;
}

int main() {
        Student a = Student("Derek", 1234);
        Student b = move(a);
        Student c = Student("Joe", 1235);
        c = move(b);
        c.display();
        cout << "End of main()" << endl;
}
```

Giving the output:

```
Constructor called
Move constructor called
Constructor called
Move assignment operator performed
A student with name Derek and ID 1234
End of main()
Destructor called for Derek
Destructor called for
Destructor called for
```

Here is another stripped down example of the move constructor in use:

#include <iostream>
#include <vector>

```
class MyClass {
public:
    std::vector<int> data;

    // Constructor
    MyClass(size_t size) : data(size) {
        std::cout << "Constructor called\n";
    }

    // Move constructor
    MyClass(MyClass&& other) noexcept : data(std::move(other.data)) {
        std::cout << "Move constructor called\n";
    }
};

int main() {
    MyClass obj1(100);      // Normal constructor
```

```
    MyClass obj2(std::move(obj1)); // Move constructor

    return 0;
}
```

In this case `MyClass obj1(100);` creates an object with a vector of size 100 using the regular constructor. `MyClass obj2(std::move(obj1));` transfers the ownership of resources from `obj1` to `obj2` using the move constructor. The `std::move` function converts `obj1` into an rvalue reference, signalling that its resources can be moved. This will give the output:

```
Constructor called
Move constructor called
```

So, move semantics allow for efficient transfer of resources (like memory) from one object to another without copying. This is particularly useful for optimising performance when dealing with large data structures or resources. When an object is moved, its resources are "transferred" rather than copied. The object being moved from is left in a valid but unspecified state. This is achieved using move constructors and move assignment operators. Move semantics typically involve rvalue references (`&&`), which refer to temporary objects that can be safely "moved from," avoiding expensive deep copies and improving efficiency.

Lambda Functions/Expression

Introduced in C++11, a lambda function (lambda expression) is an anonymous function (it has no name) with the ability to refer to operators outside of its own level of scope.

If you have a function that you only need to use once, you might just use the code inline where it is called.

Take for example a simple functor example:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct display {
      void operator()(int x){
            cout << "[" << x << "]";
      }
};

int main() {
   vector<int> v {10, 50, 90, 20, 30};
```

```
    std::for_each(v.begin(),v.end(), display());
}
```

This will give the following output:

```
[10][50][90][20][30]
```

Suppose that the function object is only ever used once. We can use an anonymous lambda function instead, which would have the form:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
    vector<int> v {10, 50, 90, 20, 30};
    std::for_each(v.begin(),v.end(), [](int x) { cout << "[" << x << "]"; } );
}
```

Which gives the exact same output. At their simplest, lambda functions are just anonymous function objects/functors and the syntax is neater, particularly if the functor is only ever to be used once.

One nice feature of lambda functions is that they can **capture variables** that are outside of the lambda function. For example, let's say that our lambda function must multiply every value by a variable factor z. You cannot just write:

```
int main() {
  vector<int> v {10, 50, 90, 20, 30};
  int z = 3;
  std::for_each(v.begin(),v.end(), [](int x) { cout << "[" << x*z << "]"; } );
}
```

As, z is outside the lambda function -- i.e., it is not captured. Instead you must list in the capture (the []) any that are to be captured. So the final version would look like this:

```
int main() {
    vector<int> v {10, 50, 90, 20, 30};
    int z = 3;
    std::for_each(v.begin(),v.end(), [z](int x) { cout << "[" << x*z << "]"; } );
}
```

And would give the output:

```
[30][150][270][60][90]
```

You can capture by reference and by value. For example:

- [z] captures z by value

- [&z] captures z by reference
- [&] captures all variables used in the lambda by reference
- [=] captures all variables used in the lambda by value
- [&, z] capture all variables by reference, except z which is captured by value
- [&x, z] capture x by reference and z by value

For example:

```
int main() {
   vector<int> v {10, 50, 90, 20, 30};
   int z = 3;
   int sum = 0;
   std::for_each(v.begin(),v.end(), [&](int x) { cout << "[" << x*z << "]";
                   sum+=x*z;});
    cout << "The sum is " << sum << endl;
}
```

Gives the output:

```
[30][150][270][60][90]The sum is 600
```

If the function was to return a value you can define this using a -> notation. For example, if the lambda function above returned an int, it would have the syntax:

```
[z](int x)->int {
       cout << "[" << x*z << "]";
       return x*z; }
```

Clearly this is not usually required for the for_each call in this case.