Object Oriented Programming

Colin Fyfe,
Department of Computing and Information Systems,
The University of Paisley.

Edition 1.1,
1996

Contents

0.1 Course outline 4

1 Objects and Classes **5**
1.1 Introduction 5
1.2 Encapsulation 6
1.3 Function Overloading 9
1.4 Constructors, destructors and scope 10
1.5 An Example 12
1.6 A Graphical Model 16
1.7 Visual C++ 19
1.8 Laboratory Tasks 19
1.9 Appendix C/C++ Syntax 19

2 Abstract Data Types **23**
2.1 Abstract Data Types 23
2.1.1 Example 24
2.1.2 The Circle class 24
2.2 Object Oriented Abstraction 28
2.2.1 Data abstraction 28
2.2.2 Behaviour Sharing 28
2.2.3 Evolution 28
2.2.4 Correctness 29
2.3 Classes 29
2.4 Member functions 29
2.4.1 This 30
2.5 Abstract Data Class Design 30
2.5.1 What makes a good object 31
2.6 Abstract Classes in C++ 31
2.7 Laboratory Exercises 32

3 Inheritance **33**
3.1 Knowledge Representation 33
3.2 An Example 34
3.3 A Second Bye 35
3.4 Private, Public and Protected 37
3.5 A Graphical Representation 39
3.5.1 Constructors with parameters 41
3.6 Type Hierarchies 41
3.7 Multiple Inheritance 42
3.7.1 Clashes of Member Identifiers 43
3.8 Run-time Decisions 43
3.9 Library programming 44
3.10 A Case Study 44
3.11 Laboratory Exercises 50

4 Functions and Operators **51**
4.1 Polymorphism and Overloading 51
4.1.1 Default parameters 51
4.2 Operators 52
4.3 Constructors and Copy Constructors 54
4.4 Friend Functions 55
4.4.1 Friend Operator Functions 57
4.5 Virtual Functions 58
4.6 Function Templates 61
4.7 More Operators 61
4.8 Inline Functions 63
4.9 Static 65
4.9.1 Static Storage Class 65
4.9.2 Static Data Members 65
4.9.3 Static Member Functions 66
4.10 Laboratory Exercises 66

5 Graphical User Interfaces **67**
5.1 Introduction 67
5.2 Event-driven Applications 67
5.3 Object Oriented Features 68
5.3.1 Encapsulation 68
5.3.2 Inheritance 68
5.3.3 Dynamic Binding 69
5.4 A First Application 69
5.4.1 Windows95 and Visual C++ 69
5.4.2 A Basic Application 69
5.4.3 An enhancement 72
5.5 Using Visual C++ 73
5.6 Documents, Views and Frames 77
5.7 Controls 77
5.8 Practical Exercise 77
5.8.1 The Circle class 78
5.8.2 Create the basic interface 79
5.8.3 Get the information to a Circle 80
5.8.4 Showing the results 81
5.8.5 Conclusion 82
5.9 A Second Application 82
5.9.1 Creating the Interface 82
5.9.2 Mapping the Interface 82
5.9.3 Coding the Functions 83
5.9.4 Conclusion 85
5.10 Further Exercises 85

6 Smalltalk -an Introduction **87**
6.1 Basic Smalltalk Syntax 87
6.1.1 Classes 88
6.1.2 Basic Syntax 88
6.1.3 Message Sending 89
6.1.4 Querying Objects 90
6.2 Using Smalltalk Interactively 90
6.2.1 Standard Output 90
6.2.2 Collection Classes 91
6.2.3 Registering variables 92
6.3 Creating New Classes 92
6.4 Reading Files 94
6.5 Laboratory Tasks 94

7 Smalltalk and Dynamic Binding **95**
7.1 Conditional Statements and Loops 95
7.2 Subclassing 96
7.2.1 The Savings class 96
7.2.2 The Cheque class 97
7.2.3 Refinements to the Class 97
7.3 Run time errors 99
7.4 Polymorphism at work 100
7.5 Conclusion 101
7.6 Laboratory Exercises 102

8 Eiffel and Objective C **103**
8.1 Introduction 103
8.2 Eiffel - the Language 103
8.2.1 Aspects of Eiffel 103
8.3 Programming by Contract 104
8.3.1 Assertions 104
8.3.2 Exceptions 105
8.4 Garbage Collection 106
8.5 Objective C 106
8.6 Dynamic and Static Binding 110
8.7 IC Paks 111
8.8 Conclusion 112

9 Exceptions, Threads and Persistence **113**
9.1 Exceptions 113
9.1.1 The Exception Stack Frame 115
9.1.2 The Terminate Function 118
9.2 Threads 119
9.3 Persistent Objects 123
9.3.1 Serializing an Object 123
9.3.2 Making a Serializable Class 126

0.1 Course outline

This course comprises 12 lectures on Object Orientation:

Week 1 Objects and Classes

Week 2 Abstract Data Types

Week 3 Inheritance

Week 4 Functions and Operators

Week 5 Graphical User Interfaces

Week 6 Java as an Object Oriented Language

Week 7 Introduction to Smalltalk

Week 8 Smalltalk and Dynamic Binding

Week 9 Eiffel and Objective C

Week 10 Exceptions, Threads and Persistence

Week 11 Java Applets

Week 12 Review or Delphi

We will mainly use C++ (Visual C++ Version 4). The first 4 weeks will introduce the basic Object Oriented concepts which are applicable on any OO system. The laboratory work will provide simple exercises using C++. Weeks 5 will give an introduction to creating Windows95 applications using the full power of Visual C++. In week 6 we introduce Java as an object oriented language and show its GUI capabilities. In weeks 7 and 8 we introduce Smalltalk and will be using a public domain Smalltalk interpreter.

From week 9 onwards we will not ask you to complete labwork since for these 4 weeks you will be attempting to complete the coursework assignment which will be given to you in week 8.

In week 9 we will look briefly at 2 other OO languages. In week 10, we will introduce three topics which are sure to gain importance over the next few years - exception handling, multi-threaded programs and persistent objects.

Finally in week 11, we will show how to create www applets using Java.

Assessment will comprise:

- A two hour exam valued at 40% of the total marks
- A software project valued at 60% of the total marks

Lecture 1

Objects and Classes

1.1 Introduction

In this lecture we will investigate the basic concepts of Object Oriented Programming. We will do this in the context of C++ source code. C++ is an object oriented language which is based on the classical non-object oriented language, C. It is in fact a superset of C and any code written in ordinary C can be compiled by the C++ compiler. This has both positive and negative aspects: positive since C++ retains C's legendary efficiency - it is specifically sold as "a better C"; but negative, since it is perfectly possible to be using a C++ compiler without writing object oriented code. Consider the code below:

```
//  program 1
//  a simple C++ program
// THIS IS NOT AN OBJECT ORIENTED PROGRAM

#include <iostream.h>

int main()
{
cout << "Hello world \n";

return 0;
}
```

The first three lines are comments. In C++, any code written between `//` and the new line character is treated as a comment by the compiler. Therefore since we require three lines of comments at the start of this program, we need `//` at the start of the second and third lines too. Blank lines are ignored so the next line of interest is the `#include <iostream.h>` line; this is a preprocessor directive[1] which includes the file "iostream.h", a **header file**, at this point in the code i.e. the contents of the file "iostream.h" are included in their entirity at this point in the source code file of the program. The file-name is surrounded by angled brackets to show that it is a standard header file: it will be searched for in specific libraries of header files. Then the main part of the implementation begins with function main(). At this stage we introduce Program 2 - exactly the same algorithm written in standard C.

```
/*  Program 2
  a simple C program
 Compare this program with the C++ program above  */

 #include <stdio.h>

int main()
{
```

[1] i.e. it is completed before compiling

```
printf("Hello world \n");

return;
}
```

Compare the first few lines of each. C treats everything between `/*` and `*/` as comments. This is also useful in C++ programs for multi-lined comments so we could have used in Program 1:

```
/*  Program 1
  a simple C++ program
 THIS IS NOT AN OBJECT ORIENTED PROGRAM
*/
```

The similarities between the two programs should be striking.eger). The braces { and } are the beginning and end of the block. The I/O in the two programs is slightly different but the structural similarities should be even more striking.

So Program 1 is written in the object oriented language of C++ yet it is not an object oriented program. Why not? The simplest answer is that there are no obvious objects in the code. So we need to discuss the question "What is an object?".

1.2 Encapsulation

If learning the syntax of an object oriented language does not guarantee that you are writing object oriented programs, what does? *This course will concentrate on encouraging you to think objects.* The process of object oriented software construction starts well before the writing of the code. We will design objects to perform particular tasks before implementing these objects in an object oriented language.

In a typical object, we bind together data and code which manipulates the data. This process is known as **encapsulation.** If done well, this process will ensure the integrity of the data from unwanted interference: the creator of an object can ensure that the data within an object is reachable in only certain ways. The object designer can insist that some data and code is private - not accessable from outwith the object - and some data and code is public - meaning that it can be accessed by any program which uses the object.

So an object is determined by

1. Its state - a `person` object may have a piece of data defining its `age` which will change in time.
2. Its behaviour - a `person` object should exhibit well-defined behaviours associated with performing particular functions (such as telling you its age).
3. Its identifier - a `person` object will be uniquely identifiable by e.g. its name "colin".

Let us recreate our first program in an object oriented way. Our requirements for this simple task are really very simple - we wish an object which will print out "Hello world". We can treat the string "Hello world" as data and the action of printing it out as a necessary method which our object must have. So we require

1. To create an object
2. To ensure that the object has the required data
3. To ensure that the object can print out the data

There is an additional requirement which is necessary for any object. This is that when the object has completed its task, the space used by the object must be freed in order that it can be used by other bits of software. This task is done in C++ by a **destructor.** Object creation is the duty of a **constructor.** We declare the object as an instance of the class. The class can be thought of as defining a template which is used to create objects. In C++ and most OO languages, we can distinguish between a class specification or interface and the implementation of the class. The interface declares to the world what an object of the class is and can do. The implementation says how it will do it. An object is known by its interface. The class interface will tell everything the world needs to know about an object of the class.

Let us first examine the specification of the object which is found in the interface file "hello.h".

```
// interface file for the class Hello
#ifndef _HELLO
#define _HELLO

#include <iostream.h>

class Hello
{
char *data;
void initialise(void);
public:
Hello(void);
~Hello(void);
void output(void);
};

#endif
```

The header file defines a new data-type, a class called `Hello`. When we wish to use an object of type `Hello`, we will simply declare an instance just as we would any standard C/C++ type such as int, char, double etc. The definition of the class continues from the opening brace { to the closing } and is followed by a semi-colon.

The class defines 5 **attributes** or **members** which instances of the class will have where we are using the term members to cover both data and functions which act on the data. The 5 members are encapsulated by the class.

The first line of the class defines an instance of type pointer-to-a-char called `data`. `data` will be used to hold the string "Hello world". The default mode for members is private which means that they cannot be accessed directly from outwith the object. Thus the first two members are private. The last three objects appear after the `public:` declaration; they are all available to users of the object. The attribute `Hello(void)` is the constructor of an object of the class; it will be called every time a new object is declared. The method `~Hello(void)` is the destructor of the object; it will be called whenever the object must be destroyed e.g. when the program execution reaches a stage that the object is no longer in scope. There is no return type associated with constructors or destructors. Notice that these methods must be public and must share the name of the class.

The final member `output(void)` is the method which will actually print out "Hello world". Now as the size of our applications increases, we are liable to have a large suite of programs with many interconnecting dependencies in the code. Therefore we wish to ensure that the class is defined once and only once and to do that we use the preprocessor conditon `#ifndef _HELLO` . This checks if the identifier _HELLO has already been defined; if it has then all code till the `#endif` is ignored; if it has not already been defined, it is now defined and all code is included.

Let us now look at the implementation of the class: it will be held in file "hello.cpp". The file-names need not be the same as the class names but it cuts down the cognitive demands on us humans if they are.

```
// implementation file for the class Hello

#include "hello.h"

extern "C" char * strcpy(char *, const char *);

const int MAX = 30;

Hello::Hello()
{
int max;
max = MAX;
data = new char [max];
initialise();
```

```
return;
}

Hello::~Hello(void)
{
  delete data;
}

void Hello::initialise(void)
{
strcpy(data, "Hello world");

return;
}

void Hello::output(void)
{

cout << data ;

return;
}
```

Firstly note that we include the header file which contains the class specification/interface. Each of the methods which were declared in the class specification must be implemented in this file. Each is now attached to the **scope resolution operator** `Hello::`; we can think of this as explicitly identifying each method with its class which will be necessary in large systems where we are very likely to use the same function identifier more than once.

The

```
#include  "hello.h"
```

is a little different in that the filename is surrounded by quotes rather than angled brackets. The compiler will now look for the file "hello.h" in the current directory first (and subsequently in specifically named other directories determined by the **-I** compiler option and the INCLUDE environment variable).

We are going to use the C function `strcpy()` which copies the contents of the string comprising its second argument to the string which is its first argument and so we, in declaring its prototype, declare that the function is declared and implemented externally to this program (and in fact belongs to the language "C").

Within the constructor, `Hello()` we allocate 30 bytes for the attribute `data` and then call the function `initialise()`. Notice that this is a private member and so may not be called outwith the class. However, like `data` it can be used anywhere within the class. `initialise()` is in fact used to ensure that data contains "Hello world".

The destructor is used to tidy up when the object ceases to exist. The only thing which is necessary is to free the memory which was allocated for `data`. The memory allocation and freeing is extremely simple in C++ but you should note that you can only use `delete` when you have used `new` to allocate memory.

The output function performs the action we require. We can now compile this class but we cannot yet make an executable since we have not a function `main()`. We will now create a third file, "sayhello.cpp".

```
#include "hello.h"
#include <stdlib.h>

int main()
{
int i;
Hello aHello;  // declared like any other variable

aHello.output();
```

```
cin >> i; // takes in an integer variable from the user

return 0;
}
```

This file must also include the interface for the `Hello` class since it will use an object of that type. A new object of type `Hello` is created in the first line of function `main()`. At this stage the class constructor is called. Now the object (whose identifier is `aHello` but can be anything we wish provided it complies with C++ identifier-naming conventions) can be used. We only want one thing and so we send it the message `output()`. We have added a line

```
cin >> i;
```

just for the sake of demonstrating integer input to the program.

The object goes out of scope when we `return 0` and so the class destructor is called for `aHello` at this point.

We can now compile and link the file "sayhello.cpp" to get an executable. So to review our progress:

- We have one file , "hello.h" which contains the class interface.
- We have a second file. "hello.cpp" which contains the implementation of the class.
- We have a third file, "sayhello.cpp" which creates and uses an object of the class.

The last two files must include the first.

Phew! This is typical of object oriented programming - the overheads in performing simple acts can be extremely high. So what are the supposed benefits of object orientation?

- Code reuse. We will see soon that, having created a simple class, we can expand the capabilities of objects by creating new classes which inherit all attributes of the original class and add to them. This provides an extremely secure method for code-reuse.
- As well as this type of reuse we have the option of creating and using libraries of objects. Classes in such libraries are classed according to their maturity. A class which has been used often in a lot of different situations is a mature class. New classes must be used with the caveat that they have not been proved to be reliable.
- The class/object design of programs is inherently modular. This forces the traditional benefits of structured design to be built into programs. In particular, it is said that OO programs are much easier to maintain than classical code.
- Using the compiler's facility to make some class attributes private greatly enhances the security of our systems.
- The software models more accurately mirror real-life. We do not in the real world have the conceptual divide between data and methods. Bundling together data with the methods which operate on the data is a more accurate representation of our mental model of the world.

1.3 Function Overloading

We now briefly introduce a very powerful feature of C++ - function overloading. In C++, two or more functions can have the same identifier *provided* the type or the number of their arguments differ. e.g.

```
void Hello::setData(void)
{
strcpy(data,"Hello world");
return;
}

void Hello::setData(char *aData)
```

```
{
if (strlen(aData) < MAX-1)  // which is why MAX is declared globally
strcpy(data,aData);
else cerr << "String too long \n"; // often cerr and cout will both be the screen
return;
}
```

will be treated as two different member functions of the class Hello and will be compiled as two different functions. When a call is made (such as `aHello.setData()`), the compiler will meet no ambiguity and will simply select the correct member based upon the number and type of the arguments used in the call to the function.

This is often used in constructors e.g. if class Hello is defined by

```
class Hello
{
char *data;
void initialise(void);
public:
Hello(void);
Hello(char *aData);
~Hello(void);
void output(void);
void setData(char *aData);
char *getData(void);
};
```

with the additional constructor implemented in the file hello.cpp as

```
Hello::Hello(char *aData)
{

int max;
max = MAX;
data = new char [max];
setData(aData);

return;
}
```

Then our file, "sayhello.cpp", can use both e.g.

```
#include "hello.h"

int main()
{

Hello aHello;   // first constructor
Hello newHello("Hi there");  // second constructor

aHello.output(); //  "Hello world"
newHello.output(); // "Hi there"

return 0;
}
```

1.4 Constructors, destructors and scope

Let us amend our constructor and destructor to output to the screen when they are used. Thus

```
Hello::Hello(char *aData)
{
int max;
max = MAX;
data = new char [max];
setData(aData);

cout >> "inside constructor \n">> data >>'\n';
return;
}

Hello::~Hello(void)
{
  cout >> "inside destructor \n">> data >>'\n';
  delete data;
}
```

Then let us amend our main program as follows

```
#include "hello.h"
#include <stdlib.h>

void afunc(void)
{
Hello newHello("That one");
}

int main()
{

Hello aHello("This one");  // declared like any other variable

afunc();

return 0;
}
```

Then `aHello` is created and exists till the program terminates by returning 0. `newHello` exists from the point when `afunc()` is called till `afunc()` returns to function `main()`. Like any other variable `newHello` has local scope. Therefore we will see output on screen

```
inside constructor
This one
inside constructor
That one
inside destructor
That one
inside destructor
This one
```

The first 2 lines are called when `aHello` is created in function `main()`. The next 2 lines are written when `newHello` is created in function `afunc()`. Now the destructor for `newHello` is called as it goes out of scope when `afunc()` terminates. Finally the destructor for `aHello` is called when the program terminates.

In passing, we note that you will more often see the file "sayhello.cpp" using a function prototype as in

```
#include "hello.h"
#include <stdlib.h>
```

```
void afunc(void);  // the function's prototype

int main()
{

Hello aHello("That one");  // declared like any other variable

afunc();

return 0;
}

void afunc(void)
{
Hello newHello("This one");
}
```

However the calling of constructors and destructors is exactly the same.

1.5 An Example

We will now create and use a more useful object. Our task is to create a system which will tell us whether, given a point's coordinates, the point lies within, outwith or on a given circle.

Traditionally we might have created a function which will take as parameters the centre and radius of the circle and the coordinates of the point. But we are now thinking objects. Often in OO design we wish to put the data at the centre of the design process; in this case we would like a class circle which will tell us if a given point is "inside itself". Therefore each instance of the class must know where it is and how big it is; so the data which will be "hidden" inside each object will be the circle's centre and its radius.

Since we are going to have to create a class circle, let us make the class as general as we are liable to want. What other things might we be interested in from a circle? Probably only two things - its area and its circumference. However these will also be hidden data (though accessible through the usual type of methods). Therefore the class specification is

```
// "circle.h"
// interface for class circle
#ifnef _CIRCLE
#define _CIRCLE


class Circle
{
double centre_x; // x-coordinate of centre
double centre_y; // y-coordinate of centre
double radius;
double area;
double circumference;
double howLong(double first,double second);
public:
Circle(void);
Circle(double first, double second, double third);
void setRadius(double aRadius);
double getRadius(void);
void setCentre(double first,double second);
double getXCentre(void);
double getYCentre(void);
void calculateArea(void);
```

```
double getArea(void);
void calculateCircumference(void);
double getCircumference(void);
int isInside(double first,double second);
int isOn(double first, double second);
int isOutside(double first, double second);
};

#endif
```

Notice that we have two constructors but no destructor; the default destructor will be called when necessary. There are a number of "set and get" functions, all of which are public. Such functions are sometimes known as retrieve and update member functions. The implementation of the class is found in the file "circle.cpp".

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

#include "circle.h"

#ifndef PI
#define PI 3.14
#endif

Circle::Circle(void)
{
// centre_x=centre_y=radius=area=circumference =0.0;
Circle(0.0,0.0,0.0); // will call other constructor
}

Circle::Circle(double firstCoord, double secondCoord, double thisradius)
{
// centre_x = firstCoord;
// centre_y = secondCoord;
// radius = thisradius ;
setCentre(firstCoord, secondCoord);
setRadius(thisradius);

calculateArea();
calculateCircumference();
}

void Circle::setRadius(double thisRad)
{
radius = thisRad;
return;
}

double Circle::getRadius(void)
{
return radius;
}

void Circle::setCentre(double first,double second)
{
centre_x = first;
centre_y = second;

return;
```

```
}

double Circle::getXCentre(void)
{
return centre_x;
}

double Circle::getYCentre(void)
{
return centre_y;
}

void Circle::calculateArea(void)
{
area = PI * radius * radius;
return;
}

double Circle::getArea(void)
{
return area;
}

void Circle::calculateCircumference(void)
{
circumference = 2.0 * PI * radius;
return;
}

double Circle::getCircumference(void)
{
return circumference;
}

double Circle::howLong(double first,double second)
{
double length;
length = sqrt((centre_x-first)*(centre_x-first) + (centre_y-second)*(centre_y-second));
return length;
}

int Circle::isInside(double first,double second)
{
double length;
length = howLong(first,second);
return length < radius;
}

int Circle::isOn(double first, double second)
{
double length;
length = howLong(first,second);
return length == radius;  // careful!! NOT length = radius;
}

int Circle::isOutside(double first, double second)
{
```

```
double length;
length = howLong(first,second);
return length > radius;
}
```

Notice the code which we have commented out but left in position in the first constructor to make comparison easier.

```
// centre_x=centre_y=radius=area=circumference =0.0;
```

This would have been easy to implement but in accordance with the principle of having a single implementation where possible we have reused the other constructor rather than enter these very simple implementation details. In general this strategy means that any changes in the implementation can be isolated. We need only make bottom level changes when the specification of a circle changes. Similarly, in the second constructor, we have the code

```
// centre_x = firstCoord;
// centre_y = secondCoord;
// radius = thisradius ;
```

but have opted to use other member function instead so that again all changes to the implementation can be isolated.

A circle object will be created and used with the following code

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

int main()
{
double xcoord, ycoord, rad;
char ans;

cout <<  "\n Please enter x_coord of circle's centre" ;
cin >> xcoord;
cout <<  "\n Please enter y_coord of circle's centre" ;
cin >> ycoord;
cout <<  "\n Please enter  circle's radius" ;
cin >> rad;

Circle aCirc(xcoord, ycoord, rad);
// C++ allows objects to be declared anywhere not just the start of a block

do
{
cout <<  "\n Please enter x_coord of point" ;
cin >> xcoord;
cout <<  "\n Please enter y_coord of point" ;
cin >> ycoord;

if (aCirc.isInside(xcoord,ycoord)) cout << "It is inside the circle";
else if (aCirc.isOn(xcoord,ycoord)) cout << "It is on the circle";
else if (aCirc.isOutside(xcoord,ycoord)) cout << "It is outside the circle";

cout <<"\n Do you wish to enter another point?";
cin >> ans;
} while (ans =='y' || ans =='Y');
return 0;
}
```

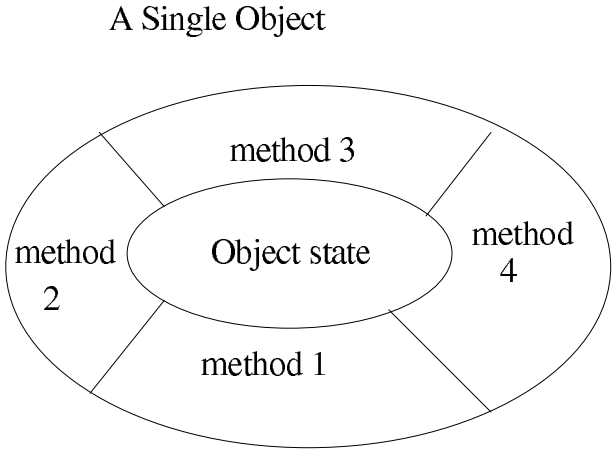

Figure 1.1: A single object: its state is the value of its member variables while the methods outside are the member functions

Notice that C++ allows objects to be declared anywhere not just the start of a block (unlike C). Therefore we postpone creating the new instance of `Circle` until we have its parameters. We also now have a `do ... while` loop which will continue to be executed while the user replies 'y' or 'Y'.

Note that we have used the `private:` mode for the member function `howlong()` since objects outside the class do not need to have access to this member. It is equally possible to declare it in `protected:` mode which means that it is still private with respect to objects outside of the current one but can be viewed by instances of inherited classes (see later). However, there is no need to make the data members `protected` since they have associated public set and get methods which can be used by any derived class. C++ provides an alternative to the dual constructor method used above in that we can specify default parameters for any member e.g. in the interface we can define the constructor

```
Circle(double firstCoord =0.0, double secondCoord =0.0, double thisRad =0.0);
```

which can be called with

```
Circle aCirc; // all 3 parameters are 0.0
Circle nextCirc(1.0,-1.0,2.0); // parameters are as defined
Circle notherC(1,0,-1.0);  // the third parameter only is the default 0.0
Circle newCirc(-9.0);  // only the first parameter is non-zero
```

1.6 A Graphical Model

One way of imagining objects is shown in Figure 1.1. We have a single object which has a state and methods which interact with the state. The state of the object is given by value currently assumed by the member variables while the methods are the member functions.

It is possible to refine this simple description of an object as shown in Figure 1.2: here we have split up the member functions into public members, which can be seen outwith the object, and private and protected members functions which cannot. It is possible that there could be public member variables but this is almost universally bad practice as it is breaking the data-hiding properties of classes.

Figure 1.3 is intended to show a complete system in which we have a number of objects each from a single class. Now all objects can have different state - the actual value of the member variables can be different for each instance - but all share the member functions of the class (regardless of whether these member functions are private, protected or public).

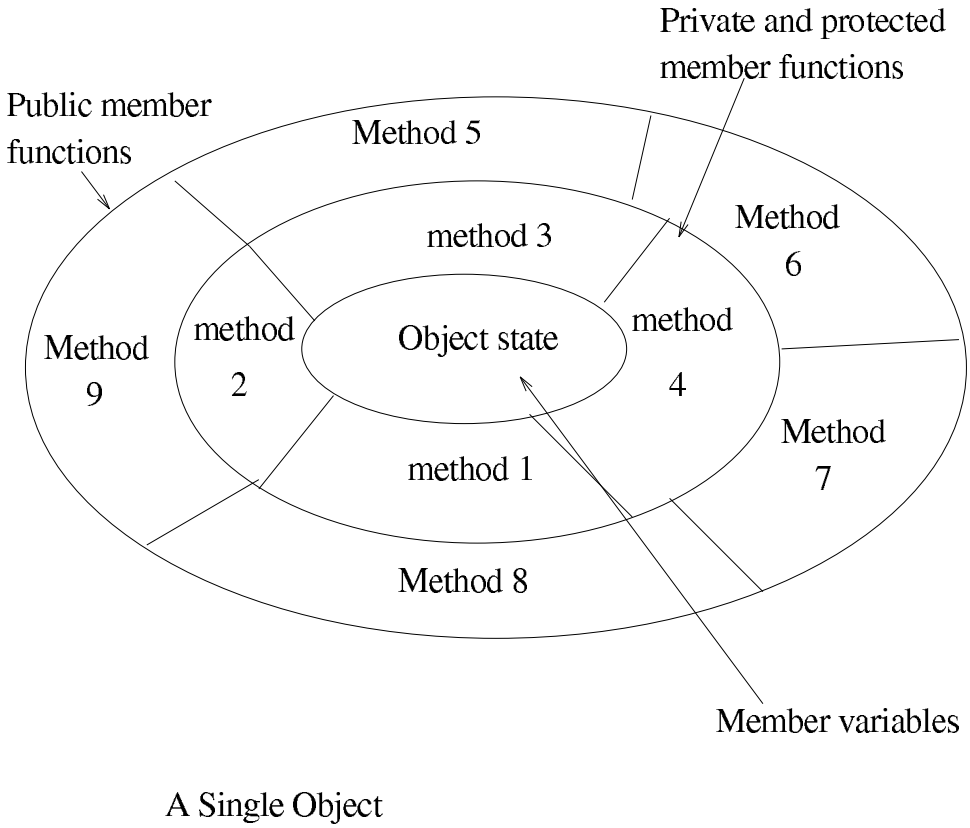

Figure 1.2: A more detailed look at an object: the member functions can be private or protected, in which case they cannot be accessed from outwith the object, or public in which case they can.

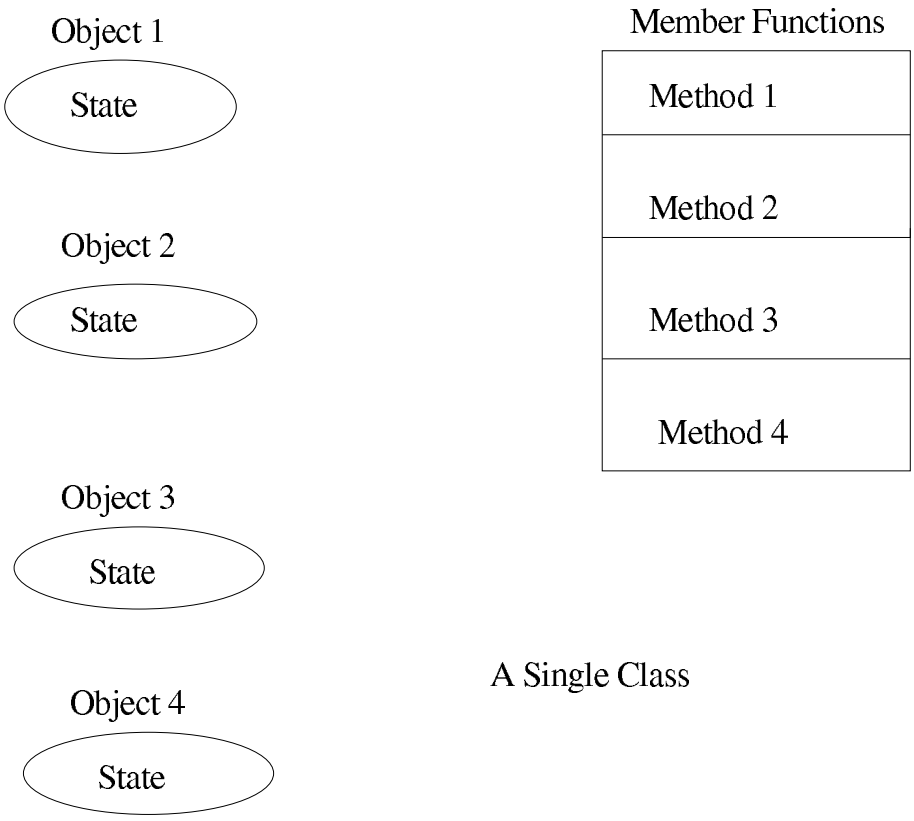

Figure 1.3: A system with 4 objects from a single class. Each object can have different state (member variables have different values) but share member functions.

1.7 Visual C++

You will use Microsoft's Visual C++ environment during this course; this allows you to create files, add them to projects, compile and link programs, debug programs and search through the library of objects. For example, to create the programs in the last section,

1. From `Start`, select `programs`, then `Microsoft Visual C++` and finally `Visual c++`. This brings up the Visual C++ environment.
2. From the `File` menu, we select `new` and select the default `code/files` and type in the code for "circle.h" and then `Save as..` "circle.h"
3. Repeat with "circle.cpp" and "testcirc.cpp"
4. Now again from the `File` menu, we select `new` but this time choose a new `project`. Select `Console Application` from the menu (This runs your (non-Windows) application in a window). You will get a second pop-up window in which you will add "circle.cpp" and "testcirc.cpp" to the project. Note that you do not need to add the header file - it will be automatically included. Close this window.
5. Now from the `Project` menu, you can either compile each of the source files separately or you can `build` the ".exe" file directly (which performs both the compilation and the linking).
6. From the `Project` menu, you may now execute the program.

1.8 Laboratory Tasks

1. Repeat the above example with a `Rectangle` class. Use the Visual C++ compiler to compile the code and create a test program to test the class. Compile and link the test program.
2. Create objects/classes to implement and test
   - A `Queue` class
   - A `Stack` class.
3. Create objects/classes to implement and test an Account class which will have a single member variable containing the instance's balance.

1.9 Appendix C/C++ Syntax

This section will not provide explanations of the code, merely examples since I am assuming that you have the necessary concepts already.

1. Declarations: can appear anywhere in C++ though we will often continue to make them at the start of blocks.

```
int anInt;
float aFloat, nextFloat;
char aChar;
char *ptr;
char ptr[50];
double x, y , z = 0.0;
```

2. Assignments: can appear in the declarations as above or elsewhere;

```
x = y = z = 10.0;
anInt = (int)aChar; // casts the value of aChar to an integer before assigning it
```

3. Control statements:

```
if ( x > 5 ) x--;
else x++;

if (i == 10) { compound statements }

switch (i)
{
case 0:  ....
break;
case 1: ...;
default:
}
```

Careful with

```
if (i=0) ...;
```

which is perfectly good C. It assigns 0 to i and then tests i which is now 0 which is equivalent to false ....
We also have a simplified test in e.g.

```
x = ( x>0 ) ? x : -x ;
```

This performs the test "is $x > 0$ " and if true the first expression is evaluated; if false the second is evaluated. So this is equivalent to assigning x its absolute value.

4. Logical operators:
   - `&&` AND
   - `||` OR
   - `!` NOT

```
if (i > 10 && j != 0) ....
```

5. Loops: first two contain compound statements; third a single function

```
do
{
...
} while (something_is_true);

while (something_is_true)
{
..
}

for (i=0;i<MAX;i++) do_this();
```

`break` breaks you out of the most internal loop construction.
`continue` moves the thread of execution to the end of the current loop but not out of the loop e.g. in the `for` loop above i is simply incremented and the test carried out as before.

6. I/O: cout/cerr is screen, cin is keyboard (unless otherwise defined)

```
cout <<"A String output"<<anInt<<'\n';
cin >>anInt;
```

7. Arrays and pointers: same thing in C but memory must be allocated if using pointers

```
char aString[60]; // space for 60 bytes allocated
char *aString;

aString = new char [60]; // 60 bytes allocated
```

In both cases, `aString` is the address of the first character in the string. Arrays begin indexing at 0. Strings end with the `'\0'`, null character. We should note that memory allocated using the `new` operator continues to be allocated till it is explicitly deallocated using the `delete` operator and so the variable used to reference the memory must be in scope at this time.

```
delete [] aString;
```

8. i++ is equivalent to i = i + 1; i– is i = i - 1 ;

9. Pointers: we can assign pointers to any object or function e.g.

```
int i, *pi, **ppi;
```

is a declaration that i is an integer, pi contains the address of an integer while ppi contains the address of the address of an integer. The actual value of the integer is found by dereferencing pi using the syntax *pi. It can be assigned a value by giving it the address of i e.g. `pi = &i` . Similarly,

```
int *pi1();
```

is the prototype of a function `pi1()` which returns a pointer to an integer while

```
int (*pi2)();
```

defines `pi2` as a pointer to a function which returns an integer.

10. Pointer arithmetic: from the first example above since pi may point to the first integer of an array, *(pi +1) is the contents of the second element of the array. So pi++ changes pi to point to the next integer in the array. Similarly pi+10 is a valid address pointing to the 11th integer in the array which can then be found using *(pi+10).

11. References: whereas C uses pointers, references are only defined for C++ programs. A reference is a pointer to variable (of any type) but has none of the different syntax which we associate with pointers. Thus the following is legitimate C++ code

```
void swap(int &x, int &y)
{
int temp;
temp = x;
x = y;
y = temp;
return;
}
```

References (shown by the ampersand) to x and y are passed into the function `swap()` but, within the body of the function, we do not need to use any special syntax. When the function returns, the values of x and y will have been swapped. To call the vunction we need no special syntax:

```
swap(first,second);
```

Lecture 2

Abstract Data Types

When we write a program to model any situation in the real world, we must abstract from the real world those aspects of the situation which we deem to be important and which we will describe in our model. Any aspect of the real world which is thought to be irrelevant is discarded.

Object oriented design also abstracts from the real world but whereas the emphasis in Structured Design methods is on creating the procedures and functions (the verbs or doing-words of the system), in Object Oriented design we give equal[1] priority to the data (the nouns or state) of the system. When we make such a design based on the data in the system, we are creating an abstract data type.

2.1 Abstract Data Types

All programming languages have types which determine the number and type of operations which are permitted on instances of the type. For example, when we declare in C

```
int a,b;
```

we are implicitly stating that the following operations will be supported:

```
a = b+3 ;
a = a+b;
a = a*b;
```

and so on. An Abstract Data Type(ADT) is a specification for a type with no account taken of the implementation of the type. In the context of the above example, we might consider an ADT called INTEGER. Now an implementation of the ADT INTEGER may be given by the C type `int`, the Ada type `INTEGER` or a user defined sub-type in any language called `PaisleyInt`. It may be that an instance of the type will be held in 32 bit two's complement or in hexadecimal or as matchsticks. These are irrelevant issues as far as the ADT is concerned. The important thing about the ADT is that it should fulfil the semantics of our concept of integers in the real world.

Therefore we can view the ADT as an idealisation which specifies what we can expect from instances of any realisation of the type and what operations are performable by or on instances of any implementation. Thus an ADT INTEGER must ensure that ordering is possible on the integers and that if x is an instance of any realisation, then x should respond with the correct semantics to the operations of add (which may or may not be represented by '+'), subtract etc..

The important points to note are:

1. The actual data structures used to hold the representation of the ADT are not relevant.

2. The ADT must explicitly and publicly declare what operations can be performed on the object

It is thought that OO design better mirrors the real world since it creates abstract data types which are closer to reality than the modular procedures which structured design methods create. The main advantage in such modelling is that the resultant representations exhibit easier understandability.

[1] most books actually say higher

2.1.1 Example

Consider again the circle class in the previous lecture. Then there is one place in which we do not seem to be matching reality very closely. Our concept of a circle is that it is fully determined by its centre and its radius. We do not think in general of its centre as two separate floating point numbers: the separate numbers are merely representations of the centre. This would suggest that we have two ADTs: one which is an abstraction of the concept of CIRCLE and a prior one which is an abstraction of POINT (in a plane).

What do we require from the ADT POINT? An Abstract Data Type POINT must satisfy the following constraints: each instance must consist of two floating point numbers which must be ordered (or otherwise distinguishable from each other) so that we can tell the difference e.g. between the x-coordinate and the y-coordinate. There must be a way of creating a new POINT and of destroying an existing instance. There should be methods of getting the point's coordinates and there should be a way of changing each coordinate.

What changes does this make to our ADT CIRCLE? First and foremost, it means that we can define our CIRCLE in terms of its centre which is an instance of the previously defined ADT, POINT. Subsequently we can ask our CIRCLE what its centre is (it should tell us a POINT), we can change the instance of CIRCLE's centre and so on. Rather than continue the discussion in the abstract, let us consider these changes on the class sturcture of our implementation.

2.1.2 The Circle class

As it was designed in the previous lecture, the Circle class forces instances to get involved in implementation details. Instead of

```
void setCentre(double xcoord,double ycoord);
double getXCentre(void);
double getYCentre(void);
```

we would like

```
void setCentre(Point aPoint);
Point getCentre(void);
```

This suggests that we have a new data-type `Point`. We have said that an Abstract Data Type POINT must satisfy the following criteria: each instance must consist of two floating point numbers which must be ordered and we would expect that the usual set and get methods would be necessary. What other things might we need from an instance of class `Point`? Perhaps the ability to find out the distance between itself and any other point. This suggests the interface file, "point.h", for the class `Point` should be

```
#ifndef _POINT
#define _POINT

class Point
{
double x_coord ;
double y_coord;
public:
Point(double first =0.0, double second =0.0);
double distance(Point aPoint);
void setValue(Point aPoint);
void setX(double anX);
void setY(double aY);
double getX();
double getY();
};
#endif
```

Notice that this specification of the class `Point` satisfies the semantics of the ADT (we have two floating point numbers which are distinguishable from each other) but even this minimal specification has gone beyond the ADT: the class is now specific to C++, is using double floating point precision to hold the values and has 7 member functions. We are at a half-way stage between the ADT and the implementation. The implementation file, "point.cpp" now contains

```
#include "point.h"
#include <math.h>

extern "C" double sqrt(double);

Point::Point(double first, double second)
{
setX(first);
setY(second);
}

double Point::distance(Point aPoint)
{
double dist;
dist = sqrt((x_coord - aPoint.getX())*(x_coord - aPoint.getX()) +
      (y_coord - aPoint.getY())*(y_coord - aPoint.getY()));

return dist;
}

void Point::setValue(Point aPoint)
{
setX(aPoint.getX());
setY(aPoint.getY());

return;
}

void Point::setX(double anX)
{
x_coord =anX;

return;
}

void Point::setY(double aY)
{
y_coord =aY;

return;
}

double Point::getX()
{
return x_coord;
}

double Point::getY()
{
return y_coord;
}
```

This allows us to modify the `Circle` class to

```
// "circle.h"
// interface for class circle
#ifnef _CIRCLE
#define _CIRCLE
```

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

class Circle
{
Point centre;
double radius;
double area;
double circumference;
public:
Circle(Point aPoint, double thisradius);
void setRadius(double aRadius);
double getRadius(void);
void setCentre(Point aPoint);
Point getCentre();
void calculateArea(void);
double getArea(void);
void calculateCircumference(void);
double getCircumference(void);
BOOL isInside(Point aPoint);
BOOL isOn(Point aPoint);
BOOL isOutside(Point aPoint);
};

#endif
```

with the associated implementation file, "circle.cpp"

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

#ifndef PI
#define PI 3.14
#endif

Circle::Circle(Point aCentre, double thisradius)
{
setCentre(aCentre);
setRadius(thisradius);

calculateArea();
calculateCircumference();
}

void Circle::setRadius(double thisRad)
{
radius = thisRad;
return;
}

double Circle::getRadius(void)
{
return radius;
}

void Circle::setCentre(Point aPoint)
{
```

```
centre.setValue(aPoint);

/*     This is conceptually nicer  than
centre.setX( aPoint.getX());
centre.setY( aPoint.getY());
so that we can get the instances of Point to perform
their actions at as high a level as possible.  */
return;
}

Point Circle::getCentre()
{
return centre;
}

void Circle::calculateArea(void)
{
area = PI * radius * radius;
return;
}

double Circle::getArea(void)
{
return area;
}

void Circle::calculateCircumference(void)
{
circumference = 2.0 * PI * radius;
return;
}

double Circle::getCircumference(void)
{
return circumference;
}

BOOL Circle::isInside(Point aPoint)
{
return centre.distance(aPoint) < radius;
}

BOOL Circle::isOn(Point aPoint)
{
return centre.distance(aPoint) == radius;
}

BOOL Circle::isOutsid(Point aPoint)
{
return centre.distance(aPoint) > radius;
}
```

Note first that we must include the header file for the `Point` class in the header file for the `Circle` class since we wish to use its specification of instances of type Point. The compiler will check that we are only using methods to which an instance of type Point can respond. Consider within the member function `setCentre()` the code which we have not commented out. You should be struck by its clarity and simplicity. We will attain maximum benefits from OO by using the highest concepts made available to us in our existing classes.

Compare the members of class `Circle` in lesson 1 with the corresponding members in our current class. The code should far better match our view of reality. In addition, we are beginning to see in this small example the power of OO techniques in mastering complex systems. By devolving responsibilities to the appropriate level of code, we can, in building large systems, deal with a top-level abstraction of the system: we wish to deal with the attributes of circles using this class not become involved with the implementation details of how distances are calculated.

2.2 Object Oriented Abstraction

Object oriented abstraction is based on 4 principles(Blair et al, 1991):

1. data abstraction
2. behaviour sharing
3. evolution
4. correctness

2.2.1 Data abstraction

The programmer is given a higher level of abstraction over both the data and the algorithms to manipulate the data. There are two processes here:

Modularisation which is concerned with breaking down complex systems into a number of smaller blocks or modules. All information about a particular entity in the system is held within a module; therefore a module is self-contained. This gives us the usual benefits in terms of testability and maintainability of the code. In addition, modularisation forces a design approach in which the programmer breaks down the domain into a number of recognisable conceptual entities.

Information Hiding hides the implementation details from the user. This allows the user to work again at a high level since he/she does not need to concern him/herself with the details of the system. The programmer provides methods which control access to the information. This also improves the reliability of the system since it is not possible to access data directly.

2.2.2 Behaviour Sharing

There are three facets to this

Single Identifier For example, we may have a member function in our system called `print_yourself()`. If every class has such a member function we are relieving the user from having to remember a new method's identifier for each class. In addition we have abstracted a behaviour at the whole system level.

Classification Every instance of each class shares the same set of behavours (member functions) though the state of each instance (its member variables) may be different.

Taxonomies Behaviour may be shared by inheritance. A class at the base of an inheritance hierarchy can have a behaviour which will be inherited by derived classes.

2.2.3 Evolution

The need for this is due to the change in requirements of any system. This is particularly the case in dynamically changing environments. However a more general view of evolution is to develop systems incrementally; this can be particularly useful when the original requirements are not tightly defined.

OO design incorporates both elements by evolving a system throughout its life-cycle through a sequence of steps.

2.2.4 Correctness

Correctness refers to the semantics of the objects in the system. We should ensure our abstractions of a system do not leave any loopholes: a user must be confident that every possible behaviour which can be asked of our system must be covered i.e. there must be an implememtation for each behaviour which is asked for and that implementation must satisfy the semantics of the behaviour.

2.3 Classes

Now for any actual implementation, we must create a data-type in the language we are using which models the ADT. At this stage we are moving from design to implementation. For user-defined data-types in OO languages we are usually talking about the creation of a new class. We have seen that the specification of the class is almost a half-way house between the ADT and the implementation: we create the class interface in order to match the semantics of the ADT (which, if our design is correct, matches the semantics of the real situation we are modelling) but now when we create the interface, we must be aware of implementation constraints within the language.

Firstly are there any advantages of classes other than those derived from the use of ADTs?

1. Classes increase the robustness of the system; if the language is strongly typed (as C++ is), the compiler will perform checks that our instances respond to the messages which we have sent them. Less strongly typed OO languages (as we shall see later) will be more prone to run-time errors.
2. Also in such a system, we can perform optimisation at compile-time.
3. A class system separates the implementation from the specification. Thus we can optimise implementation details on a class by class basis.
4. A class system enables the creation of a library of classes.

Notice this distinction between a class and the corresponding ADT - the class may separate implementation details from interface but both must be present in the definition of the class whereas only the interface/specification must be present in the ADT. Further a class can be viewed as a template (a "factory") for creating new instances which should conform to the ADT. Thus we may create three points in a program with the declaration

```
Point a(5.0,9.0), b, c(1.0,1.0);
```

The class constructor creates the three instances and since these instances must conform to the specification for the class, we know that all three instances must respond appropriately to the member functions of the class.

It should be stated at this point that, while all OO languages have classes, not all languages have strong typing e.g. in Objective C, there is a universal identifier which may pertain to any class. Thus when we make the declaration

```
id anObject;
```

we are simply declaring an instance of a class which is to be specified at run time. An advantage of this is that e.g. collection classes are very easy to implement in Objective C: each collection class is simply a collection of ids. This does, however, leave open the possibility of sending a message to an object which does not respond to the message (i.e. whose specification does not include the member function). We will return to this point in more detail later.

2.4 Member functions

Now an ADT is defined not just by the type of the data (integer, circle etc.) which it models but also by the functions to which instances of the data type much respond if the ADT is to match the semantics of the real world.

So a class is defined not only by the data which is held by instances of the class but also by the member functions to which instances of the class must repond. Often OO practitioners speak of applying member functions as message passing. Thus in our previous example, when the instance of type `Circle` receives the

message `isInside(thisPoint)` it will pass the message (see implementation) `distance(thisPoint)` to its instance variable `centre`. We may at this stage imagine a complex multi-object model in which instances of various types are interacting by passing messages to each other. Therefore invoking a function requires

1. The identifier of the function
2. The identifier of the object to which the function is passed
3. The arguments of the function
4. A variable to which the return type (if any) may be attached

In C++, resolution of messages takes place at compile-time; thus after a successful compilation, we can be sure that all objects will respond to the messages which will be sent to them.

2.4.1 This

There is an inherent difficulty with the system as described so far. Each time we create an instance of class Point, memory is allocated for the member variables of the instance i.e. in this case for the two doubles, `x_coord` and `y_coord`. However all instances of class Point share the member functions - we do not create replica member functions for each instance.

Now consider the declaration

```
Point a(1.0,2.0),b(-1.0,9.5);
```

When the new instance a is declared, the constructor for the class Point calls the member functions `setX()` and `setY()`, each of which changes the value of a member variable (`x_coord` and `y_coord` respectively). Now the question arises *how does the flow of control of the program know whose member variables to change?*. Even in our simple example we have two instances of the class `Point` in the system and there looks to be immense potential for confusion. This is resolved by the identifier `this`. When each instance is created, there is also created a pseudovariable `this` which points to the new instance of self. Since it is a pointer, we can use the c syntax construct for referring to members of the current object:

```
Point::Point(double first, double second)
{
this -> setX(first);
this -> setY(second);
}
```

which is exactly equivalent to our existing code

```
Point::Point(double first, double second)
{
setX(first);
setY(second);
}
```

So if it is exactly equivalent, how does the called method `setX()` know that it is to be used by the current object? The answer is that the pointer `this` appears as an unseen first parameter to each member function of the object. Thus in the constructor, `this` is initialised to point at the current object being created.

2.5 Abstract Data Class Design

One very simple way to design ADTs is the use of Class-Responsibility-Collaborator(CRC) cards. This involves stepping through a variety of scenarios which are thought to be typical of the situations which you wish to model. As we come to an object which is necessary to the simulation we create a new card representing the class of the object and write what responsibilities each member of the class has (i.e. what can be expected from instances of the class); there are two basic types of responsibilities: responsibilities for knowing something and responsibilities for doing something. Collaborators (objects from other classes) are necessary in order that the responsibilities of the class can be executed. They represent the answers to the question "With whom are you doing it?".

It is thought that the tangible nature of the CRC cards helps in the role-play used to step through typical situations. It is a group design method rather than one for the individual but its informality limits its usefulness.

Thus in the simulation of a dentist's surgery, we may have one class called Receptionist instances of which have responsibility for taking new patients into the dentist's waiting room (the waiting room may be a new class of type Queue with which the instance of Receptionist must collaborate), for keeping the appointment book (another collaborator class) up to date, etc. Each individual in the design team is expected to take responsibility for one (or more) CRC cards and hence the classes associated with the card.

When this process is complete, the specification of the class should be complete. Programmers must now continue with the implementation but can do so in isolation from each other; however, clearly they must not change the interfaces of the agreed classes.

2.5.1 What makes a good object

When designing a OO system[2],

1. Look out for candidates which are really doing the same thing.
2. Ensure that all candidate classes are essential to the view of the world that the model describes.
3. Reject candidates which are e.g. member variables of larger objects.
4. Group sets of individual objects together as instances of a class where possible.
5. Look for features of objects which point to other possible objects which are not yet in the model.
6. Operations on objects are often (though not always) member functions rather than independent classes.
7. The "how much" test. If you can put "how much" in front of the candidate it may not be a suitable object; if you can put "how many", it is more likely to be.
8. Discard implementation-dependent details

2.6 Abstract Classes in C++

Looking ahead to the next lecture we note that we can subclass any class so that the *derived* class inherits all members of the *base* class. Thus we can define a class Account as follows:

```
class Account
{
double balance;
public:
Account(double d);
virtual double getBalance(void);
virtual void printBalance();
};
```

with implementation file

```
#include "account.h"

Account::Account(double d)
{
balance = d;
return;
}

double Account::getBalance(void)
{
```

[2]This section is based on work by Cook and Daniels

```
return balance;
}

void Account::printBalance(void)
{
cerr <<"Error. Balance not available \n";
return;
}
```

For the moment we will ignore the keyword `virtual`. We can now subclass the `Account` class to get a class `ChequeAccount` and another class `SavingsAccount`. However note that anyone who creates an instance of `Account` and asks it to `printBalance()` will simply get an error message - not at all the semantics he/she might expect from the function.

A better way is to create an abstract class by creating a *pure virtual function*. The abstract class, while it exists as C++ code, is one for which there can be no instances; it merely acts as a definition class for those classes subclassed from it. To create an abstract class, simply redefine the class as:

```
class Account
{
double balance;
public:
Account(double d);
virtual double getBalance(void);
virtual void printBalance()=0;
};
```

The "=0" defines `printBalance()` as a pure virtual function - it has no implementation and so there can never be any instances of the class. The class forms a type of template to which subclasses must conform. Now `ChequeAccount` and `SavingsAccount` must redefine `printBalance()`. If they do not, they in turn are abstract classes and cannot have instances.

2.7 Laboratory Exercises

1. In groups of 4/5, continue the design of the Dentist's surgery.
2. Update your class `Rectangle` to use the class `Point`.
3. Write a C++ abstract class for `Quadrilateral` which could be subclassed by the classes `Square`, `Rectangle` or `Kite`.

Lecture 3

Inheritance

One of the major benefits of OO which is widely touted is code-reuse. One of the major methods of implementing code-reuse is inheritance. We will show that inheritance also means sharing of the inherited member variables which sometimes creates a tension with the encapsulation principles we met in Lecture 1.

3.1 Knowledge Representation

By using inheritance, we can emulate our own representation of knowledge in that we can create a hierarchy or taxonomy of representations. For example, we can represent a vehicle taxonomy as shown in Figure 3.1. Here each elipse can be thought of as a type(class) and the arrows can be thought of as the relationship ISA between instances of the class. Thus an instance of `Fiat` is also an instance of `Car`, and of `Road_vehicle` and of `Vehicle`.

Now instances of the class `Vehicle` may have a BOOLEAN member variable `has_wheels`; this may be assigned the value `TRUE` in constructor of the class `Road_vehicle` so that instances of `Car` will also have this BOOLEAN member `has_wheels` (inherited from `Vehicle`) which will automatically have the value `TRUE` since a new instance of car will automatically call the constructors of `Vehicle` and `Road_vehicle` in that order. Similarly if the `number_of_wheels` variable is set to 4 in `Car`, instances of both `Fiat` and `Rover` will have this value in the variable `number_of_wheels` unless they specifically change it. Note the dual gains here:

1. We have a representation which mirrors our own taxonomy tree

2. We have code-reuse: we do not need to tell each `Rover` it has 4 wheels since this was defined in the class `car`.

This type of inheritance tree is a form of **structural inheritance:** we are specialising as we go down the tree.

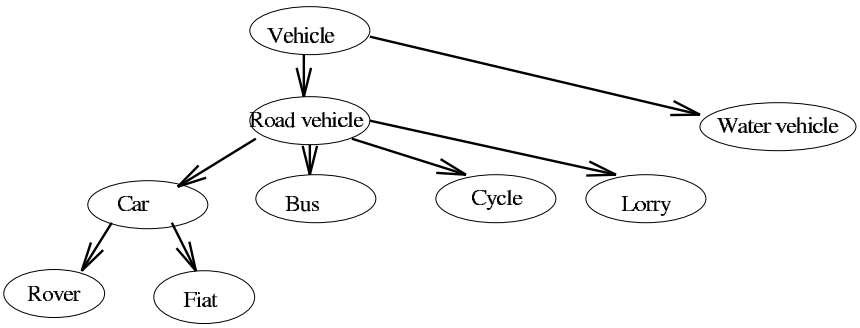

Figure 3.1: A semanctic net showing the relationship ISA in a vehicle hierarchy

3.2 An Example

Consider again the example `Hello` class met in Lecture 1, but now our specification changes: we wish to have a second type of object which will *also* say goodbye. Now we could simply change the initialise method but having done so we would not then be able to use our simple `hello` class again. And further if we wished our objects now to say goodnight we would have to change it again. A better solution is to have a new class inherit the work we have already done with the `hello` class. First we will put the interface for our new class in the file "bye.h":

```
// interface file for the class Bye
#ifndef _BYE
#define _BYE

#include <iostream.h>

class Bye : public Hello
{
char *data2;  // this will contain the bye message
void initialiseBye(); // empty parameter string is same as void
public:
Bye(int anInt);
~Bye();
void tellAll();
};

#endif
```

The class `Bye` is now defined as a subclass of the class `Hello` and all members of the class `Hello` are inherited by an instance of `Bye`. The keyword **public** declares that all public members of `Hello` are inherited as public members of `Bye`. But note that all private members are not accessible in an instance of `Bye` (except through any public members which are inherited).

Class `Bye` now has, as well as the member `data` inherited from `Hello`, a new member `data2` which is also declared as a char *. We can initialise `data2` using a new member function `initialiseBye()` but how do we initialise the member `data` when we have no access to the member `initialise()`, a private member of `Hello`. Recall that the member `initialise()` is called from the constructor of `Hello`. Now the rule is that when a new instance of a derived class is created, the constructor of the base class is called first (which in this case will call `initialise()`) and the constructor of the derived class is called next. So `data` is initialised as before when an instance of `Bye` is declared. Note also that destructors are called in the reverse order.

Now we can write the implementation of the `Bye` class in the file "bye.cpp":

```
// implementation file for the class Bye

#include "bye.h"

extern "C" char * strcpy(char *, const char *);

Bye::Bye(int anInt)
{

data2 = new char [anInt];
initialise();

return;
}

Bye::~Bye(void)
{
  delete data2;  // no need to worry about data
```

```
// Hello's destructor will deal with that
}

void Bye::initialise(void)
{
strcpy(data2, " \n and goodbye \n ");

return;
}

void Bye::tellAll(void)
{

output();  // this is a public member of Hello

cout << data2 ;

return;
}
```

Notice how

- The constructor takes a parameter this time. Therefore the calling program must use a parameter when a new instance of Bye is created.
- `tellAll()` uses a public member of the base class to output `data`.

Now we can have a system with two types of object: an instance of `Hello` and an instance of `Bye`.

```
#include "hello.h"
#include "bye.h"

int main()
{
int i;
Hello aHello;
Bye cheerio(50);     // constructor takes an integer parameter

aHello.output();
cheerio.output(); // ok public member of Bye

cheerio.tellAll();  // but not aHello.tellAll()

cin >> i; // Holds window till user enters integer (visual C++)

return 0;
}
```

Now when the program ends, the destructor for the class `Hello` is called twice - once for `aHello` and once for the `Hello` part of `cheerio`.

3.3 A Second Bye

Creating the derived class as we have done above has one big disadvantage: it means that any user of our classes must remember that e.g. the initialisation function of `Hello` is simply `initialise()` while that of `Bye` is `initialiseBye()`. Now while this does not seem a large constraint on their use, when this situation is met over many messages over a whole library of classes, the cognitive demands put on programmers can be very large. Let us consider a different implementation.

Thus we amend our `Bye` class to

```
#ifndef _BYE
#define _BYE

#include "hello.h"
#include <iostream.h>


class Bye : Hello
{
  char *data2;
  void initialise();
  public:
  Bye(int anInt);
  ~Bye();
  void output();
 };

 #endif
```

with an associated implementation file

```
#include "bye.h"

extern "C" char * strcpy(char *, const char *);

Bye::Bye(int anInt)
{
data2 = new char [anInt];
initialise();
}

Bye::~Bye()
{
   delete [] data2;
 }

 void Bye::initialise()
 {
 strcpy(data2, "\n and goodbye\n");

 Hello::initialise();

  return;
  }

  void Bye::output()
  {
  Hello::output();
  cout << data2;

  return;
  }
```

Consider what happens now when an instance of `Bye` is created. The `Hello` constructor is called which calls the member function `initialise()` of `Hello`. Now the constructor of `Bye` is called which calls its member function `initialise()`. In case we wish to specifically call the `initialise()` member function of `Bye` to initialise both sets of member variables, we have made an explicit call to `Hello::initialise()`. This actually means that when creating a new instance of `Bye`, this function is called twice. We will later investigate methods to avoid this duplication.

A similar action occurs with the `output()` function. There is one difficulty with the story as we have presented it so far and that is that the member function `initialise()` of the `Hello` class has been declard private and so cannot be accessed by the derived class `Bye`. As we shall see in the next section we must declare the function `initialise()` as protected.

```
 #ifndef _HELLO
#define _HELLO

 #include <iostream.h>

class Hello
{
char *data;
protected:
void initialise(void);
public:
Hello(void);
~Hello(void);
void output(void);
};

 #endif
```

3.4 Private, Public and Protected

There are two times when the distinction between private and public members receive attention:

1. when a new class is specified
2. when a class inherits members from a previous class

With respect to the first situation, we have in both of our examples above, declared the data (`data` and `data2`)in the objects to be private. We could have declared them to be public (appearing after the declaration `public:`). This would have enabled us to use lines such as

```
Hello aHello;

strcpy(aHello.data, "hi there");
```

You will see that typically we do not do so since a basic principle of object orientation is **data hiding**. If data is made visible to external objects or other pieces of software, it has the potential for being changed. Data hiding is used to ensure that each piece of data can only be accessed through an interface provided by the object creator. In this example, if we had allowed a user of our object to have access to `data`, there would have been nothing to stop our user from writing

```
strcpy(aHello.data, "hallo there, how's it going, Jim");
```

which string exceeds the length allocated for our `data`. Thus in permitting `data` to be public, we have decreased the robustness of our system. Good object oriented design will have data hidden and allow specific "get and set methods". Thus we may have

```
// interface file for the class Hello
#ifndef _HELLO
#define _HELLO

#include <iostream.h>

class Hello
{
char *data;
```

```
void initialise(void);
public:
Hello(void);
~Hello(void);
void output(void);
void setData(char *aData);
char *getData(void);
};

#endif
```

which will require the following additions to "hello.cpp"

```
extern "C" int strlen(char *);

void Hello::setData(char *aData)
{
if (strlen(aData) < MAX-1)  // which is why MAX is declared globally
strcpy(data,aData);
else cerr << "String too long \n"; // often cerr and cout will both be the screen
return;
}

char *Hello::getData(void)
{
return data;
}
```

Note at this point the shortness of each member; this is typical of object oriented code and is an aid to code reuse.

The second time when public/private issues are raised is when we are inheriting. We could have written

```
class Bye : private Hello
```

which would have resulted in not only the private members of `Hello` being private in an instance of `Bye` but also the public members becoming private. Such newly privatised members are though still accessible to members of `Bye`. Note the difference between the status of the base class members: a member which is declared private within a base class cannot be accessed *at all* by any member function of the derived class whereas the newly privatised members (which were declared public in the base class) are accessible in the derived class.

However there will be times when you will create a class instances of which will contain data types which on the one hand you wish to keep private from other classes and yet to which you wish derived types to have access. To do this declare the data as `protected:`. Thus if `Hello` has an interface,

```
// interface file for the class Hello
#ifndef _HELLO
#define _HELLO

#include <iostream.h>

class Hello
{
protected:
char *data;
void initialise(void);
public:
Hello(void);
~Hello(void);
void output(void);
```

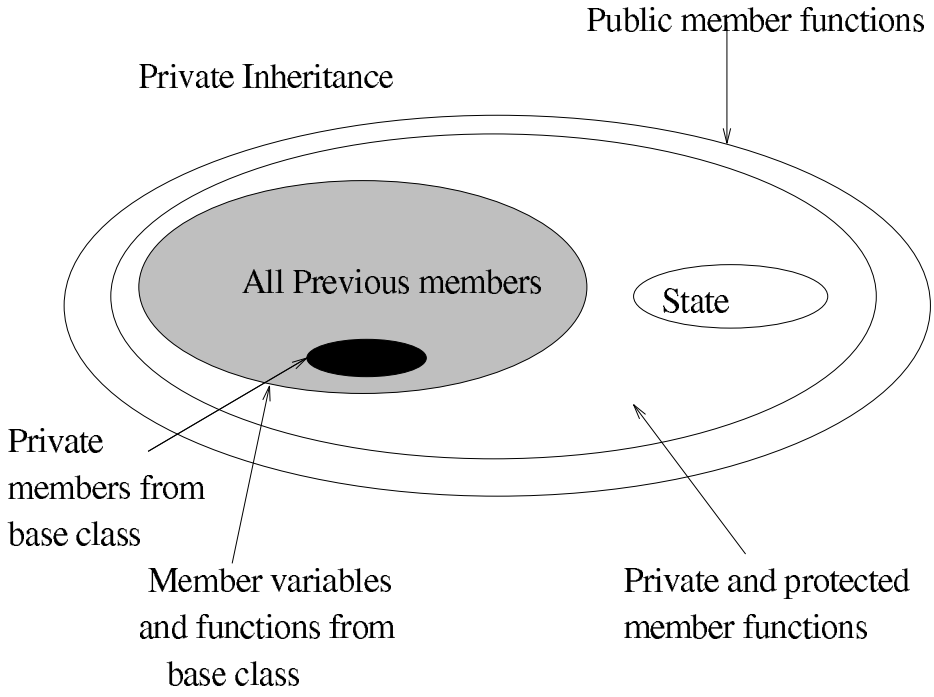

Figure 3.2: Private inheritance is shown here by the dark patch. All members of the base class are now private but the private members of the base class are in a special position: they cannot be accessed by the derived class at all

```
void setData(char *aData);
char *getData(void);
};

#endif
```

As far as objects outwith Hello are concerned, `data` and `initialise()` are both still inaccessible but now instances of classes derived from `Hello` such as `Bye` can access these members.

It is possible to place the keywords, `private, protected` and `public` anywhere inside the interface but we will use the style rules that there will be a single private group of members, followed by a single protected group followed by a single public group (where some groups may not be present).

Finally note that we are now differentiating between two sets of clients for a class:

1. Instantiating clients - objects which create and manipulate instances of the class *but only through those members declared public in the class*

2. Inheriting clients - subclasses which may have access to protected members of the class

3.5 A Graphical Representation

We show a graphical representation of the inheritance mechanism in Figures 3.2 and 3.3.

For private inheritance, all members (variables and functions) are now private and cannot be accessed outwith the instance of the class. However the public members of the base class are in a special position (superprivate) in that they cannot be accessed *at all*; not even the instance can affect or look at its own

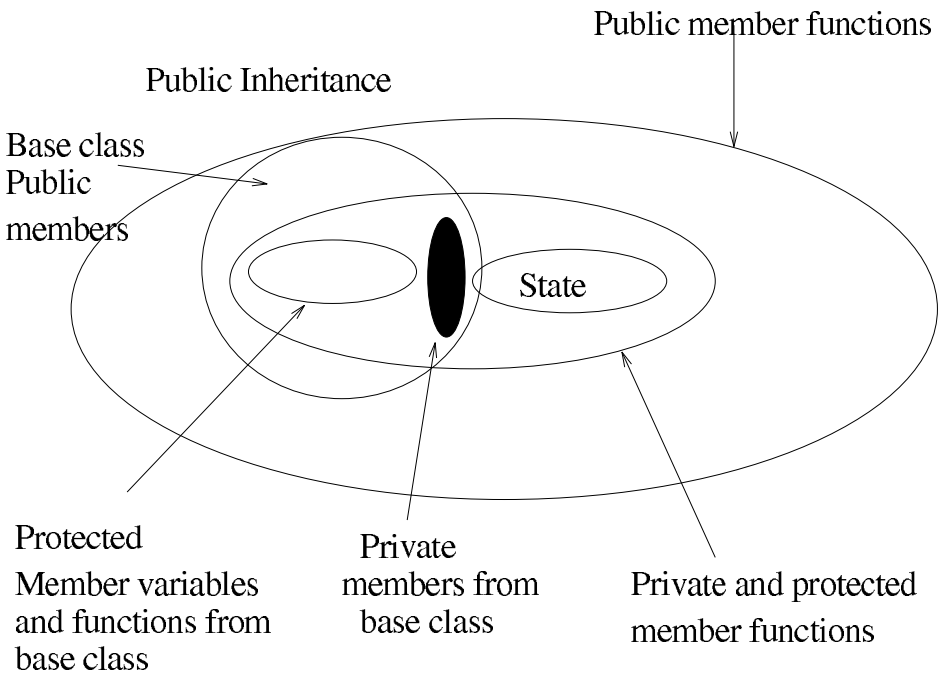

Figure 3.3: The base class is shown by the internal circle. Its private members are still unaccessible in the derived class but all other members are now accessible.

inherited private members. For public inheritance, the base class members retain their visibility but private member functions and variables are still not visible to the instance of the derived class.

3.5.1 Constructors with parameters

Now consider what happens when our base class requires a parameter in its constructor. e.g. when the Hello class has a constructor

```
Hello::Hello(char *aData)
{

int max;
max = MAX;
data = new char [max];
setData(aData);

return;
}
```

Then our derived class, Bye must have data to instantiate the base class. Then Bye's interface is now

```
// interface file for the class Bye
#ifndef _BYE
#define _BYE

#include <iostream.h>

class Bye : public Hello
{
char *data2;  // this will contain the bye message
void initialiseBye(); // empty parameter string is same as void
public:
Bye(int anInt, char *aData);
~Bye();
void tellAll();
};

#endif
```

with associated implementation

```
Bye::Bye(int anInt, char *aData):Hello(aData)
{

data2 = new char [anInt];
initialiseBye();

return;
}
```

so the constructor in Hello receives the required data. Note that the body of the constructor in Bye could also use "aData" since it is simply a normal parameter of the constructor of Bye.

3.6 Type Hierarchies

In the last lecture, we compared the semantics of types and classes. We can derive the properties of inheritance by considering the semantics of subtyping.

We can subtype in most languages. Typically subtypes are valid if the **Principle of Substitutability** is satisfied: T_1 is a valid subtype of T_2 if and only if an instance of T_1 can be used wherever an operation

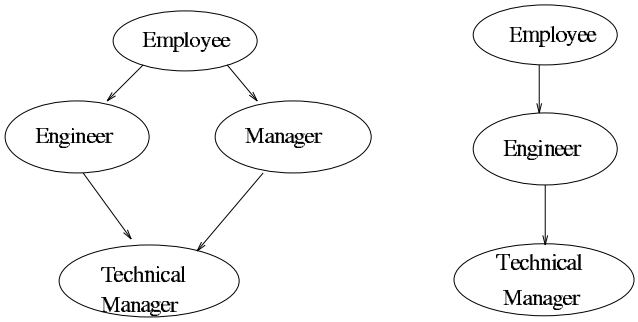

Figure 3.4: Left: An ISA network rather than a tree showing muytiple inheritance. Right: Repeated (but not multiple) inheritance.

or construct expects an instance of T_2. If an operation expects an integer and `Numb` is a subtype of integer then the following is valid:

```
Numb aNumb ;
aNumb := 3 + aNumb;
```

When we subtype types such as the integers, we are creating subsets of the integers - we are restricting the range of appropriate instances. Notice that subtyping is

- Transitive: if T_2 is a subtype of T_1 and T_3 is a subtype of T_2 then T_3 is a subtype of T_1.
- Reflexive : T_1 is a subtype of itself.

So since the relationship is antisymmetric, we can create a partial ordering of types and so a hierarchical tree can be formed[1].

However subtyping by subsetting has its dangers: since a type is a set of objects and the operations that can be performed on the objects, we must ensure that the semantics of each operation is satisfied. For the example given above, if type `Numb` is the set of odd integers, then the operation "+" no longer has the correct semantics since $1 \in$ `Numb` and $3 \in$ `Numb` but it is not true to say that $1 + 3 \in$ `Numb`.

When we subtype complex data types, we are using **inclusion** semantics. If we have a record type Person which contains a field to hold an integer corresponding to the person's age in years and a second field which holds a person's name then a new record type AddressedPerson which contains a field holding a person's address *as well as* the previous two fields is a subtype of the type Person since by the Principle of Sustitutability, each instance of AddressedPerson can be used when a Person might be. With this in mind we can define ADT_1 to be a subtype of ADT_2 if

- The structure of instances of ADT_1 is a subtype of the structure of instances of ADT_2; in C++ classes we would require that every member variable of an instance of ADT_2 appears in the instance of ADT_1.
- The behaviour of ADT_1 in terms of the signature and specification of its member functions must conform to the behaviour of ADT_2.

3.7 Multiple Inheritance

The left half of Figure 3.4 gives a simple example showing how multiple inheritance can arise. Notice that the right half is not a multiple inheritance graph - there is a single chain of inheritance throughout the diagram. The left half is sometimes described as a Directed Acyclic Graph(DAG). Many OO languages do not allow multiple inheritance: its very power makes it an extremely dangerous tool, if misused.

Consider what could happen when an instance of `TechnicalManager` is created: first the constructor for `Engineer` is called which in its turn calls the constructor for `Employee`. Then the constructor for `Manager` is called which in its turn calls the constructor for `Employee`. It seems entirely possible that we will have two copies of the members of `Employee` in each instance of `TechnicalManager`. Therefore when accessing member variables of an instance of `TechnicalManager` when these variables have been defined in `Employee` leads to conflict. To prevent this we inherit the base classes as a **virtual** base classes:

[1]We will later see that multiple inheritance will change this to a directed acyclic graph

```
class Engineer: virtual public Employee
{
...
};

class Manager : virtual public Employee
{
...
};

class TechnicalManager : public Engineer, public Manager
{
...
};
```

Notice that this small change only affects the interface of the classes. The implementation of the classes is unaffected. The net result is that only one copy of the class `Employee` is present in the class `TechnicalManager`. Note also that the class `Employee` still exists as before for instances of `Manager` and `Engineer`.

3.7.1 Clashes of Member Identifiers

It is entirely possible in a large inheritance DAG that the same member identifier will be used more than once. Some languages (e.g. Eiffel) use a specific renaming policy to resolve such conflicts. We can easily resolve them in C++ using the scope resolution operator.

```
void Manager::initialise(void)
{
Employee::initialise();
// now perform the specific actions necessary
// to initialise the members specific to Manager
}
```

Notice however that this may not stop the member `Employee::initialise()` being called more than once if it is also called in `void Engineer::initialise(void)`.

3.8 Run-time Decisions

One of the live issues in OO programming is with respect to what sort of checking you need at compile-time and what can be left to run-time. This issue will feature more strongly in the next and subsequent lectures but for the moment we can set the scene with respect to the inheritance hierarchy.

Consider the following code:

```
class Square : public Rectangle
....

int main()
{
Rectangle *ob1;  // pointer to an instance of Rectangle
Rectangle aRec;  // a new instance of Rectangle
Square aSq; // a new instance of Square

ob1 = &aRec; // making ob1 point to the address of the
// instance of Rectangle

ob1 = &aSq; // making ob1 point to the address of the
// instance of Square
}
```

Here we have defined a class Square as a subclass of Rectangle (and have assumed the appropriate implementation file) and declared a pointer to class Rectangle and an instance of each of Rectangle and Square. As shown in the code fragment, the pointer can be used to point to the object from the derived class as well as to one from the base class. However, three points must be made

1. The pointer can only be used to access those members of the derived object which were inherited from the base class. You could not use `ob1` to access members of `aSq` which were not defined in the class Rectangle.

2. The reverse is not true - you cannot use a pointer to a derived class (Square) to point to an instance of the base class (Rectangle).

3. Note that e.g. `ob1++` will move the pointer along to what it thinks is the next *base* class instance i.e. to the next instance of Rectangle even if it is currently pointing to an instance of Square.

Clearly the potential for mishaps is very great but the potential benefits are equally great.

3.9 Library programming

Some languages come with a built-in library - collection classes such as sets, dictionaries and sorted lists are often an important part of the library. This is not the case in C++ but all commercial compilers come with a set of library classes to which you can add your own. A major task of the new OO programmer is learning the libraries which are available to her/him in the local compiler. Thus, a major new job title will be Software Librarian. A programmer is someone who can join together the appropriate software objects to create an application with the required functionality.

Most compilers nowadays come with an object browser which allows you to jump through the class hierarchy in a hypertext fashion.

3.10 A Case Study

Let us reinvestigate the Queue and Stack class which we wrote last week. We might have created a Queue class like

```
#ifndef QUEUE_H
#define QUEUE_H

#define MAX 100
class Queue
{
float *theQueue;   // the actual Queue
int size ; // maximum size of queue
int currentSize; // current size of queue
public:
Queue(int aSize = MAX);
~Queue(void);
int add(float anyFloat);
float remove(void);
};
#endif
```

with associated implementation file

```
#include "queue1.h"
#include <stdlib.h>
#include <iostream.h>

Queue::Queue(int anInt)
{
```

```
theQueue = new float [anInt];
size = anInt;
currentSize =0;
}

Queue::~Queue(void)
{
delete [] theQueue;
}

int Queue::add(float x)
{
if (currentSize < size)
{
theQueue[currentSize] = x;
currentSize++;
return currentSize;
}
return -1;
}

float Queue::remove(void)
{
float temp;

if (currentSize >0)
{
temp = theQueue[0];
for (int i=0;i<currentSize;i++)
theQueue[i] =theQueue[i+1];
return temp;
}
cerr << "Nothing in Queue " << endl;
return -1.0;
}
```

and a Stack class

```
#ifndef STACK_H
#define STACK_H

#define MAX 100
class Stack
{
float *theStack;   // the actual Stack
int size ; // maximum size of stack
int currentSize; // current size of stack
public:
Stack(int aSize = MAX);
~Stack(void);
int push(float anyFloat);
float pop(void);
};

#endif
```

with its implementation

```
#include "stack1.h"
```

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

Stack::Stack(int anInt)
{
theStack = new float [anInt];
size = anInt;
currentSize =0;
}

Stack::~Stack(void)
{
delete [] theStack;
}

int Stack::push(float x)
{
if (currentSize < size)
{
theStack[currentSize] = x;
currentSize++;
return currentSize;
}
return -1;
}

float Stack::pop(void)
{
if (currentSize >0)
{
/* currentSize--;
return theStack[currentSize]; */
return theStack[--currentSize];
}
cerr << "Nothing in Stack " << endl;
return -1.0;
}
```

But now we notice that there is actually a great deal of similarity between the two classes and decide to use that to create an abstract data type from which both can inherit. One possibility would be the Bag class:

```
#ifndef BAG_H
#define BAG_H

class Bag
{
protected:
float *theBag;
int size;
int currentSize;
public:
 virtual int add(float aFloat) =0;
 virtual float remove(void) = 0;
};

#endif
```

Now the Bag class is an abstract data type (since add() and remove() have no actual implementation i.e. both are pure virtual functions and so no instances of class Bag can be created. Note also that the instance

variables have been made protected so that any derived class will have access to them.

Now we can have our Queue class defined as

```
#ifndef QUEUE_H
#define QUEUE_H

#include "bag.h"

#define MAX 100
class Queue : public Bag
{
public:
Queue(int aSize = MAX);
~Queue(void);
int add(float anyFloat);
float remove(void);
};

#endif
```

Notice that we have explicitly redefined add() and remove() so that we can now create an instance of this class. The implementation file is

```
#include "queue2.h"
#include <stdlib.h>
#include <iostream.h>

Queue::Queue(int anInt)
{
theBag = new float [anInt];
size = anInt;
currentSize =0;
}

Queue::~Queue(void)
{
delete [] theBag;
}

int Queue::add(float x)
{
if (currentSize < size)
{
theBag[currentSize] = x;
currentSize++;
return currentSize;
}
return -1;
}

float Queue::remove(void)
{
float temp;

if (currentSize >0)
{
temp = theBag[0];
for (int i=0;i<currentSize;i++)
```

```
theBag[i] =theBag[i+1];
return temp;
}
cerr << "Nothing in Queue " << endl;
return -1.0;
}
```

You can see that we are accessing the protected member variables in a way which would not have been possible had they been declared private in the Bag class. We require slightly more changes to the Stack class:

```
#ifndef STACK_H
#define STACK_H

#include "bag.h"

#define MAX 100
class Stack :public Bag
{
public:
Stack(int aSize = MAX);
~Stack(void);
int push(float anyFloat);
float pop(void);
int add(float a);
float remove(void);
};

#endif
```

where we required to have new member functions add() and remove() since they were pure virtual functions in the base class and must therefore be overridden in any derived class if there are to be instances in the derived class. However such overriding requires minimal extra work on our part and we can keep our pop() and push() methods:

```
#include "stack2.h"
#include <stdlib.h>
#include <iostream.h>

Stack::Stack(int anInt)
{
theBag = new float [anInt];
size = anInt;
currentSize =0;
}

Stack::~Stack(void)
{
delete [] theBag;
}

int Stack::push(float x)
{
if (currentSize < size)
{
theBag[currentSize] = x;
currentSize++;
return currentSize;
}
```

```
return -1;
}

float Stack::pop(void)
{
if (currentSize >0)
return theBag[--currentSize];

cerr << "Nothing in Stack " << endl;
return -1.0;
}
int Stack::add(float x)
{
return push(x);
}

float Stack::remove(void)
{
return pop();
}
```

We can use such classes as we might expect:

```
#include "queue2.h"
#include "stack2.h"

#include <stdlib.h>
#include <iostream.h>

int main()
{
int i;
Queue a,b(90);
Stack c,d(50);
Bag *e;

for (i=0;i<10;i++)
{
a.add(float(i));
c.push(float(i));
b.add(float(100-i));
d.push(float(100-i));
}

cout<<"The Queue " << endl;
e = &a;
for (i=0;i<10;i++)
cout << e->remove() << endl;

cout<<"The Stack" << endl;
for (i=0;i<10;i++)
cout << c.pop() << endl;

cout<<"The Stack" << endl;
e =&d;
for (i=0;i<10;i++)
cout << e->remove() << endl;
// but not e->pop();
```

```
return 0;
}
```

Notice that we created a pointer to an instance of the base class and used it to remove an element from the Stack. While the methods pop() and remove() might seem to be the same, in fact we can only use the remove() method with this pointer since we can only use it to access those members which were available in the base class.

3.11 Laboratory Exercises

1. You are going to create a geometry hierarchy:
   (a) You created a `Rectangle` class in Week 1. Subclass it to make a `Square` class.
   (b) Create a program which will create and use instances of both `Rectangle` and `Square` and query each appropriately. Ensure that the `Rectangle` class inherits from the abstract class you created last week.
   (c) Create a `Kite` class which will also inherit from the abstract class `Quadrilateral`. Draw the inheritance hierarchy you have created.
   (d) Now have your `Square` class inherit from the `Kite` class. What changes are necessary to your hierarchy?
2. You are going to create an Account hierarchy:
   (a) Subclass the abstract class `Account` created last week to create a `ChequeAccount` class.
   (b) Subclass the abstract class `Account` to create a `SavingsAccount` class.
   (c) Subclass both the above classes to make a `ChequeSavingsAccount`. Show the class hierarchy
3. Use the browser to browse the Visual C++ window classes.

Lecture 4

Functions and Operators

As we noted in previous weeks, a class is defined by its interface which defines both its member variables and its member functions. We will now consider its member functions more closely.

4.1 Polymorphism and Overloading

Overloading is not an exclusively OO concept. Every procedural language uses overloading to some extent. When we use the basic operator "+" we are using an overloaded operator since "+" can be applied to either integers or floating point types. Typically overloaded functions or operators are overloaded because the data types to which they are bound are from the same family. Thus the "+" operator is overloaded for the basic number types and we can imagine it overloaded for ADTs VECTOR, COMPLEX_NUMBER etc.. OO orientation, however, takes this concept and makes it a central tool in the management of complexity.

Polymorphism is the idea of "one interface, multiple implementations". A particular set of activities may have a single interface to present to the outside world; which particular implementation is used depends (in C++) on the compiler. Notice this difference between true polymorphism and overloading: overloading requires a different parameter profile for the functions, polymorphism uses the same parameter profiles for the functions. We will call the method of polymorphic implementation overriding. This removes the burden from the programmer of having to specify e.g. that it is an integer-output not a float-output that must be used in a particular situation. The programmer need only remember the general interface.

However as we shall see polymorphism is at its most powerful when resolution takes place at run-time rather than compile-time.

4.1.1 Default parameters

One way that C++ obviates the need to have explicit overloading is to provide member functions with default parameters. Thus we can define a member function with

```
int aFunc(int a=0, double c=10.0);
```

and call it with

```
a = aFunc(); // equivalent to aFunc(0,10.0)
a = aFunc(5,10);
a = aFunc(7); // equivalent to aFunc(7,10.0)
```

Note that all default parameters must be to the right of any non-default parameters. Thus

```
int aFunc(char *aName, int a=0, double c=10.0); // OK
int aFunc(int a=0, char *aName,  double c=10.0); // WRONG
```

You can use this method in the overloaded constructor in Lecture 1.

There is a danger of introducing ambiguity with this type of default parameter. For example, if we have

```
int aFunc(int a, double c);
int aFunc(int a);
```

then when we make the call `a = aFunc(5)` , there is no doubt which function will be called. However if there is a default parameter for c, we have introduced a situation where two functions both may satisfy the call.

Note that we have a method of finding the address of particular functions called e.g. with the last two definitions of aFunc(), we can declare

```
int (*fp1)(int);
int (*fp2)(int, double);
```

Now `fp1` is a pointer to a function which takes one integer parameter while `fp2` is a pointer to the function which takes one integer and one double parameter (in that order). Then simply making the assignments

```
fp1 = aFunc;  \\ the address of aFunc(int)
fp2 = aFunc;  \\ the address of aFunc(int, double)
```

4.2 Operators

Operator overloading is a special case of function overloading; the crucial point to remember is that the overloading is done relative to a particular class. Let us consider this in the context of a class to model coordinates:

```
class Coord
{
int x, y ; // the coordinates
public:
Coord(int i = 0, int j =0);
int getX(void);
int getY(void);
Coord operator+(Coord ob2);
};
```

With this syntax we have declared that we are going to overload the "+" operator for the class `Coord`. Notice that we are returning an instance of type `Coord` since we will almost certainly sometime wish to use "+" in the form `coord1 = coord2 + coord3 + coord4`. To implement this we have the file "coord.cpp":

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

#include "coord.h"

Coord::Coord(int i, int j)
{
x = i;
y = j;
return;
}

int Coord::getX(void)
{
return x;
}

int Coord::getY(void)
{
return y;
}
```

```
Coord Coord::operator+(Coord ob2)
{
int temp_x, temp_y;

temp_x = x + ob2.getX();
temp_y = y + ob2.getY();

Coord temp(temp_x, temp_y); // creates a new instance

return temp;
}
```

Several points must be noted at this stage:

1. Most operators can be overloaded.
2. You cannot change the precedence of an operator e.g. * has precedence over + and must remain so for all overloaded implementations.
3. You cannot change the number of operands e.g. `Coord operator+(Coord ob2, Coord ob3);` is illegal.
4. Operators cannot have default parameters

Naturally, while it is permissable to have an overloaded operator perform any function, it is better to keep to semantically similar operations for any overloaded operator.

A little care must be taken with the assignment operator since we wish to allow statements like `a=b=c=d`. Now when we have assigned the value of d to c, we have a second assignment to worry about - b must be assigned to something and that something has to be c's value. Therefore we have

```
Coord operator=(Coord ob2);
```

in "coord.h" and

```
Coord Coord::operator=(Coord ob2)
{
x = ob2.getX();
y = ob2.getY();

return *this;
}
```

That is we are returning the object which has just been assigned the new values from `ob2`.

We can also overload an operator more than once e.g.

```
class Coord
{
int x, y ; // the coordinates
public:
Coord(int i = 0, int j =0);
int getX(void);
int getY(void);
Coord operator+(Coord ob2);
Coord operator+(int anInt);
};
```

where we have two different overloaded versions of the operator +.The implementation file must be augmented with something like

```
Coord Coord::operator+(int anInt)
{
int temp_x, temp_y;
temp_x = x + anInt;
temp_y = y + anInt;

Coord temp(temp_x, temp_y);
return temp;
}
```

There is no confusion for the compiler e.g.

```
main()
{
Coord c1,c2,c3(7,5);
int i =10;

c1 = c1 + c3; // uses first version of +
c2 = c3 + i; // uses new version of +
}
```

Notice though that we cannot yet perform `c2 = i + c3` since we have only overloaded + *with respect to the class Coord* which must appear to the left of the operator.

4.3 Constructors and Copy Constructors

We have previously met constructors and written classes which have overloaded constructors. However there is are situations when the compiler requires to copy an existing object e.g.

- When we use an object as a value parameter to a function e.g

  ```
  Coord ob1(2,8),ob2;
  ...
  dist = howlong(ob1,ob2);
  ```

  Then the program will call a copy constructor to make copies of `ob1` and `ob2` and place these copies on the stack.

- Returning from a function that declares a member of the class as its return value

  ```
  Coord translate(int horiz, int vert);
  ...
  ob3 = translate(3,4);
  ```

  This time the copy constructor is called to copy the object when the return statement of the function is called.

- When we initialise a new object by copying the old

  ```
  Coord ob2(ob1);
  ```

  Note that we do not yet have a constructor which will do this..

If you do not declare a copy constructor, the default copy constructor is called which performs a member-by-member copy. (Note this is recursive in that each member variable's copy constructor is called). An example for our `Coord` class is

```
class Coord
{
int x, y ; // the coordinates
public:
Coord(int i = 0, int j =0); // existing constructor
Coord(Coord& ob1); // the copy constructor
int getX(void);
int getY(void);
};
```

which requires the implementation in "coord.cpp" of

```
Coord::Coord(Coord& ob1)
{
x = ob1.getX();
y = ob1.getY();

}
```

In the constructor we are passing the existing object in by reference. This is a feature of C++ which we do not have in C. It may be thought of as a pointer to the object but without any of the associated pointer operations i.e. within the function to which the reference has been passed, simply treat the object as we would any other object. This is a valuable feature and should be used for our previously defined + and = functions.

4.4 Friend Functions

Friend functions break all the OO rules yet are extremely common in C++. Friend functions are used when you wish a function to have access to private members of the class but still not itself be a member function of the class. One use of friend functions is when you wish a function to have access to the private members of more than one class. For example consider the `Circle` class defined in Lecture 1 and the `Coord` class defined above. Then we may wish to amend the `Coord` class definition as follows:

```
class Circle; // a forward reference
// Circle will continue to be defined in its own header file
class Coord
{
int x, y ; // the coordinates
public:
Coord(int i = 0, int j =0);
int getX(void);
int getY(void);
Coord operator+(Coord ob2);
Coord operator+(int anInt);
friend int is_in(Circle aCirc, Coord aCoord);
};
```

We may put the definition of the friend function in the file "coord.cpp" (or in the main program or any other file which will be linked into the executable) but it still will not have the scope resolution operator :: attached to it

```
int is_in(Circle aCirc, Coord aCoord)
{
double length;
length = aCirc.howLong(aCoord.x, aCoord.y);
return length < radius;
}
```

Notice that this function is reaching inside the instance of `Coord` to view the values of its x and y member variables and it has also looked inside the instance of `Circle` to use its private member function `howLong()`. Therefore obviously we must also update the file "circle.h" to show that `is_in()` is a friend of this class too

```
// "circle.h"
// interface for class circle
#ifnef _CIRCLE
#define _CIRCLE

#include "coord.h"

class Circle
{
double centre_x; // x-coordinate of centre
double centre_y; // y-coordinate of centre
double radius;
double area;
double circumference;
double howLong(double first,double second);
public:
Circle(void);
Circle(double first, double second, double third);
void setRadius(double aRadius);
double getRadius(void);
void setCentre(double first,double second);
double getXCentre(void);
double getYCentre(void);
void calculateArea(void);
double getArea(void);
void calculateCircumference(void);
double getCircumference(void);
int isInside(double first,double second);
int isOn(double first, double second);
int isOutside(double first, double second);
friend int is_in(Circle aCirc, Coord aCoord);
};

#endif
```

An alternative is to have a function a friend of one class and an ordinary member function of the other. Thus we could have declared

```
class Circle; // a forward reference
// Circle will continue to be defined in its own header file
class Coord
{
int x, y ; // the coordinates
public:
Coord(int i = 0, int j =0);
int getX(void);
int getY(void);
Coord operator+(Coord ob2);
Coord operator+(int anInt);
friend int Circle::is_in( Coord aCoord);
};
```

and declared `is_in()` as an ordinary member function of `Circle`.

4.4.1 Friend Operator Functions

However, by far the most frequent use of friend functions is with respect to operators and by far the most frequent use of friend operators is with respect to I/O operations. Firstly, however, consider the class definition

```
class Coord
{
int x, y ; // the coordinates
public:
Coord(int i = 0, int j =0);
int getX(void);
int getY(void);
friend Coord operator+(Coord ob1, Coord ob2);
};
```

Now this allows an implementation

```
Coord operator+(Coord ob1, Coord ob2)
{
Coord temp;
temp.x = ob1.x + ob2.x;
temp.y = ob1.y + ob2.y;

return temp;
}
```

This is extremely succinct compared to the previous code *but it breaks all the rules about encapsulation.* This function has reached into three objects' private members and changed the values of one. There is one situation where we really require this dispensation, however: recall that we can perform

```
Coord + Coord and
Coord + int but not
int + Coord
```

Now by declaring + a friend operator, we can permit the last operation; we may in fact declare all three overloaded operators friend operators:

```
class Coord
{
int x, y ; // the coordinates
public:
Coord(int i = 0, int j =0);
int getX(void);
int getY(void);
friend Coord operator+(Coord ob1, Coord ob2);
friend Coord operator+(Coord ob1, int anInt);
friend Coord operator+(int anInt, Coord ob1);
};
```

which will permit us now to have a main program

```
main()
{
Coord c1,c2,c3(7,5);
int i =10;

c1 = c1 + c3; // uses first version of +
c2 = c3 + i; // uses second version of +
c3 = i + c2; // uses third version of +
}
```

However the most important use of friend operators is to extend the functionality of the I/O system:

```
class Coord
{
int x, y ; // the coordinates
public:
Coord(int i = 0, int j =0);
int getX(void);
int getY(void);
friend ostream &operator<<(ostream &aStream, Coord ob);
};
```

Notice that we have declared the output operator << a friend operator and that this operator returns a reference to a stream (it will actually be the same stream "aStream" which it takes as a parameter). This is to allow concatenation of operations such as

```
cout<< "The object is " << ob1 << '\n';
```

Typically the implementation of this friend function will appear in "coord.cpp" as

```
ostream &operator<<(ostream &aStream, Coord ob)
{
aStream << " ("<<ob.x << "," << ob.y <<") ";
return aStream;
}
```

Note that we use the more general `ostream` in our definition and implementation rather than the specific `cout` even if we are sure that we shall always be using the standard output stream - you never know when the more general code will be necessary. With this limited breaking of encapsulation there is perhaps no great problem, however it is a potential weak spot in our encapsulation.

It is also possible to declare a whole class as a friend in which case, all of the member functions of that class can access the private members of the first class. e.g.

```
class Anyclass
{
...
public:
friend class Notherclass;
...
};
```

You can see friendship drives a coach and horses through encapsulation. But it is worth noting that

- Friendship is not transitive. The friend of a friend is not necessarily a friend unless it is declared to be so.
- Friendship cannot be inherited. A class derived from a friend class is not a friend unless it is declared to be so.

4.5 Virtual Functions

All of the methods described so far have one great drawback - their usefulness is retricted by compile-time checking. What we would really like is the flexibility of "single-interface/ multiple implementations" where the actual implementation is not determined till run-time. This is true run-time polymorphism. In C++, this is achieved by the use of virtual functions.

Consider the following classes which represent a Polyhedron[1] class, a Triangle class and a Rectangle class.

[1] any straight edged figure

```
class Polyhedron
{
int sides;  // number of sides
public:
Polyhedron(int numb = 3);
int get_side(void);
virtual double calculate_perimeter(void);
};
```

This class contains a constructor and two member functions one of which is delared virtual.

```
#include <stdlib.h>
#include <iostream.h>
#include "polyhed.h"

Polyhedron::Polyhedron(int numb)
{
sides = numb;
}

virtual double Polyhedron::calculate_perimeter(void)
{
cout <<"this method is not to be used";
return 0.0;
}

int Polyhedron::get_side(void)
{
return sides;
}
```

We can now subclass Polyhedron and have the subclass redefine the virtual function with a new function.

```
#include "polyhed.h"

class Triangle : public Polyhedron
{
double length;
public:
Triangle(double aLength);
double calculate_perimeter(void);
};
```

Note the redeclaration of `calculate_perimeter(void)` without the use of the virtual keyword. The implementation file is

```
Triangle::Triangle(double aLength): Polyhedron(3)
{
length = aLength;
}

double Triangle::calculate_perimeter(void)
{
return get_side()*length;
}
```

Now we repeat with the Rectangle class.

```
#include "polyhed.h"
```

```
class Rectangle : public Polyhedron
{
double length;
double breadth;
public:
Rectangle(double aLength, double aBreadth);
double calculate_perimeter(void);
};
```

Note that we are again overloading the member function `calculate_perimeter(void)`

```
#include "rectangle.h"

Rectangle::Rectangle(double aLength, double aBreadth) : Polyhedron(4)
{
length = aLength;
breadth = aBreadth;
}

double Rectangle::calculate_perimeter(void)
{
return length+breadth+length+breadth;
}
```

Now we can access the appropriate method by using pointers. Recall from the last lecture that a pointer to an instance of a base class can be used to point to an instance of a derived class. But we noted at that stage that we could not use the pointer to access members which are only defined in the derived class. But now we have defined a (virtual) function `calculate_perimeter()` in the base class and redefined it in each of the derived classes. Thus the following is legitimate code

```
#include <stdlib.h>
#include <iostream.h>
#include "polyhed.h"
#include "rectangle.h"
#include "triangle.h"


int main()
{
Polyhedron *p;
Rectangle first(4.5,5.0);
Triangle second(5.0);

p = &first;
cout << "Rectangle's perimeter is "<< p->calculate_perimeter() <<'\n';

p= &second;
cout << "Triangle's perimeter is "<< p->calculate_perimeter() <<'\n';

return 0;
}
```

Note the crucial difference here between virtual functions and overloading: overloading a function/operator requires that the profile (number and type of parameters) of the overloaded function is different for each version; with virtual functions it is essential that the functions have the same profile. We sometimes call the redefinition of virtual functions overriding rather than overloading.

Two restrictions you should be aware of are:

1. You cannot override constructors

2. The overridden functions must be member functions

Note also that if a derived class had not overridden `calculate_perimeter()` it would have used Polyhedron's version of `calculate_perimeter()`. Now we do not really wish to ever use this version and so we would like to compel every subclass to override `calculate_perimeter()`. To do so, we define `calculate_perimeter()` in Polyhedron as a *pure virtual function* by assigning it the value 0.

```
class Polyhedron
{
int sides;  // number of sides
public:
Polyhedron(int numb = 3);
int get_side(void);
virtual double calculate_perimeter(void) = 0;
};
```

We now say that Polyhedron is an **abstract** class since we cannot have an instance of the class (though we can still create a pointer as before).

4.6 Function Templates

Often the same operation is valid for a great number of different classes. Rather than write out similar code for each different data type, C++ provides a way to write the function once (as a template) and then use it with each of the different types.

```
template< class T > void swap( T &first, T &second)
{
T temp;
temp = first;
first = second;
second = temp;
return;
}
```

This function is delared to be a template function which means that it can be called with members of any class. Each time that it is called, it is done so with two arguments each of which is a member of the same class. They will be passed by reference. The `swap()` function relies on the fact that all classes inherit the default assignment "=" operator or may write their own. Thus calls such as

```
swap(array1, array2);
swap(account1,account2);
```

are both legal where the first may use the default assignment operator and the second uses a user defined = operator.

4.7 More Operators

```
#ifndef COORD_H
#define COORD_H

#include <iostream.h>

class Coord
{
int x;
int y;
public:
Coord(int anX=0,int aY=0);
Coord operator+(Coord &a);
```

```
Coord operator+(int a);
Coord &operator=(const Coord &a);
Coord &operator++(void); // prefix operator
Coord operator++(int dummy) ; // postfix operator
Coord &operator+=(Coord &a); // trivial
Coord &operator+=(int a);
int getX(void) const ;
int getY(void) const;
friend ostream & operator<<(ostream &a, Coord &b);
};

#endif
```

Now the operators "++" or "+=" are a single operator - two keyboard presses but parsed as a single operator by the compiler. Note that the postfix and prefix "++" operators are differentiated by the postfix being declared with an unused integer parameter. Note also that we have declared `getX()` as a `const` function. Such functions ensure that the state of the object (the values in the instance variables) are the same when the function returns as they were when the function was called. The implementation file is

```
#include "coord.h"

Coord::Coord(int a, int b)
{
x=a;
y=b;
return;
}

Coord Coord::operator+(Coord &a)
{
int tempx, tempy;
tempx= x + a.getX();
tempy = y + a.getY();
Coord temp(tempx,tempy);
return temp;
}

Coord Coord::operator+(int a)
{
int tempx, tempy;
tempx = x+a;
tempy = y+a;
Coord temp(tempx, tempy);
return temp;
}

Coord &Coord::operator=(const Coord &a)
{
x = a.getX();
y = a.getY();
return *this;
}

Coord& Coord::operator++(void)
{
x++;
y++;
return *this;
```

```
}

Coord Coord::operator++(int dummy)
{
Coord temp(x,y);
++x;
++y;
return temp;
}

Coord& Coord::operator+=(int a)
{
x +=a;
y +=a;
return *this;
}
Coord &Coord::operator+=(Coord &a)
{
x += a.getX();
y += a.getY();
return *this ;
}

int Coord::getX(void) const
{
return x;
}
int Coord::getY(void) const
{
return y;
}

ostream & operator<<(ostream &b, Coord &a)
{
b << "("<< a.getX() <<" , " << a.getY() << " )" ;
return b;
}
```

Notice that for some operators we must actually return an instance of the type Coord since we declared and used a local variable in performing the calculation. In others we were able to return *this since this is a constant pointer to the current object.

4.8 Inline Functions

An inline function is one in which the code for the function is inserted in the position in the code from which it is called. This saves the expense of a function call for functions which are used regularly. Since the relative expense of saving one function's stack and creating another is only really significant if the function is a short one, inline functions tend to be short ones.

There are two types of inline functions: explicit and implicit. First we repeat the interface file for the Hello class which we met in Lecture 1.

```
// interface file for the class Hello
#ifndef _HELLO
#define _HELLO

#include <iostream.h>
```

```
class Hello
{
char *data;
void initialise(void);
public:
Hello(void);
~Hello(void);
void output(void);
};

#endif
```

Now the `output()` method is only a short method and we might wish that that be created as an inline function. One method is to put the code into the interface:

```
// interface file for the class Hello
#ifndef _HELLO
#define _HELLO

#include <iostream.h>

class Hello
{
char *data;
void initialise(void);
public:
Hello(void);
~Hello(void);
void output(void){cout << data <<'\n'; return};
};

#endif
```

A second method is to leave the interface as it is and use the reserved word `inline` when specifying the function's implementation:

```
inline void Hello::output(void)
{
cout << data << '\n';
return;
}
```

Both methods have the same effect - the code for `output()` is inserted (at the compiler's discretion) into the relevant parts of the calling program. The second method is more OO than the first since the first breaks the distinction between the interface and the implementation of a class. However the first is widely used by C++ programmers.

Notice that it is the compiler which performs the insertion rather than, as with macros, the preprocessor. This means that e.g. type-checking is carried out. In addition, some macros are prone to side-effects; e.g.

```
#define toupper(a) ((a) >= 'a' && ((a) <= 'z') ? ((a) - ('a'-'A')) : (a))

void main()
{
char ch = toupper( getchar());
printf("%c",ch);
}
```

The intention is that the program should take in one character from the keyboard and, if necessary, convert it to uppercase. However `getchar()` is called once before testing whether its value is greater than 'a', once before testing whether its value is less than 'z' and a final time when its value is changed to uppercase. An alternative is

```
inline char toupper(char a)
{
return ((a) >= 'a' && ((a) <= 'z') ? ((a) - ('a'-'A')) : (a));
}
void main()
{
char ch = toupper( getchar());
printf("%c",ch);
}
```

Since the inline function takes a single argument and is evaluated as a whole before returning a value, you can be sure of its action.

4.9 Static

Static is another of those keywords that C++ uses in more than one context. It firstly uses it (as C does) in the context of storage class, and then goes on to reuse it for member variables and functions.

4.9.1 Static Storage Class

The storage class of a variable determines where the variable is allocated. Static variables are created on the data segment rather than the stack or the heap. The lifetime of a static variable lasts from when the definition is reached (the first time) in the execution code till the program finishes executing. Globally declared objects are static by default. Local variables (within a function) have automatic storage and are deleted when the function returns. For objects, then, a static variable means that the constructor and destructor of the object are called just once.

```
int any_func(void)
{
static int number_of_times =0;
...
number_of_times++;
...
}
```

If `any_func()` is called several times, the variable `number_of_times` will give the number of times it has been called.

4.9.2 Static Data Members

```
class Anyclass
{
static int howMany;
...
};
```

The member variable `howMany` is stored in the data segment along with global variables and static local variables but has a slightly different meaning in that it is shared by all members of a class. A static member variable is allocated only once and is used by all members of a class e.g. to keep a count of how many instances there are of the class in existence. An alternative to the above construction is to declare the member *outside* the class definition i.e. in the file anyclass.cpp we will have a definition,

```
int Anyclass::howMany =0;
```

We can then have an increment of the variable every time the constructor is called.

4.9.3 Static Member Functions

```
class Anyclass
{
...
static int howMany(void);
};
```

A static member function can only access member variables or member functions which are also declared static. They are often `set()` or `get()` methods. A static member function does not pass the hidden `this` pointer since it does not need to (indeed cannot) access the data area of individual objects.

Static member functions can be inlined but not virtual.

4.10 Laboratory Exercises

1. Overload the -, * and == operators[2] for the `Coord` class defined earlier. Why is the order within the implementation of - important in a way that is not so important for + or *?

2. The ++ operator is a unary operator in C++ which has the following semantics: a++ is equivalent to a = a +1 ; similarly, a– is equivalent to a = a -1; Implement these unary operators for the Coord class.

3. Create an abstract class Shape with a pure virtual function `double get_area(void)` and two derived classes `Circle` and `Square`. Dynamically test your overridden function.

[2]The == operator will return 1 if the objects are equal and 0 otherwise

Lecture 5

Graphical User Interfaces

There is no doubt that, hand in hand with the drop in cost of computer hardware, the factor which has had most effect in making computer power accessible to the public at large has been the development of Graphical User Interfaces (GUIs).

5.1 Introduction

The popularity of GUIs is based mainly on the fact that they can be easily mentally aligned with simple metaphors - the most common being the desktop metaphor in which the objects which will be manipulated on screen appear as objects on a desktop and the visible actions have a simple correlate to the actions which will take place when they are performed. For example, a file may be copied to the disk on drive A by moving the pointer over the representation of the file and dragging the file onto the representation of drive A.

A GUI is a bit-mapped display in which any particular location is identifiable by its pixel position. A window is a collection of pixels which together form some region of the display. However the location of the window may change and so the actual pixels which comprise the window will change. To minimally specify the location of a window we require the x- and y-coordinate of its bottom left and top right corners.

A window has associated with it a set of resources such as its current size, location, title, icon etc. Most windows are complicated structures; in order not to have to reinvent the wheel with each separate window, windows are arranged in an inheritance hierarchy. Visual C++ provides one such hierarchy and also a browser to investigate the properties of all windows. When you create a new window, you have the option of creating a browser file associated with the window which will be viewable through the browsing mechanism.

5.2 Event-driven Applications

The type of application which uses a GUI is often described as "event-driven". Typically the application will begin by creating a window on screen which requires the user to perform some action such as

- click on a button
- enter a file-name in a dialog box
- select from a menu

Before the user does so, the application will merely wait. This sequence requires that the application has been programmed in advance to respond to certain types of user actions *and* it must be obvious to the user that such actions are required.

Most compilers come with a library which contains basic interface classes which respond to simple (mouse-driven) actions. Creating a window in a GUI system generally involves the following four steps

1. Define the window - what type of window, what resources it will have etc.
2. Register the window with a window manager. There must be a central resource that can identify individual windows.
3. Draw the window.

4. Manage events such as requests from a user to redraw the window, resize it, button presses etc.

The final stage involves setting up a message loop. When a GUI application is running it will continually be sent messages from users. These messages will be stored in a queue till they can be stored and processed. Every time the application is ready to process another message, it calls for the next message in the queue. Each message (in MS Windows) is a structure containing information such as

- the identity of the window for which the message is intended
- an identifier of the message type
- any parameters which the message might take
- the time the message was received
- the location of the pixel on screen where the message was received

If there is no message in the queue then control passes back to the (Windows) operating system which will continually poll the message queue. Windows 3.x is non-preemptive multitasking in which the application retains control till it has finished processing.

However the trend in computing is towards distributed systems which involve **servers** which provide access to a variety of shared system resources to **clients**, the applications which use these resources. Therefore we can have several users at different terminals all using the same resources. Equally we can have a single application running different **threads** on different workstations. This is best achieved using a different type of operating system - one which uses preemptive multitasking in which the operating system allocates resources to applications on a need basis. Windows95 uses preemptive multitasking.

5.3 Object Oriented Features

We will briefly examine some implications of OO GUI applications in the light of the previous 4 lectures.

5.3.1 Encapsulation

We have stated that each window in a GUI will have a certain state comprising not only such simple features as its size, location, colour - foreground and background - but also more complex state features such as whether text can be entered in the window, whether it can be resized, whether it responds to left-button events and so on. These comprise the member variables of the window object.

All of these are defined by resources allocated to the window when it is created and some may be changed dynamically by actions from the user; for example, using the mouse to drag and drop the window, changes the location resource of the window. Therefore attached to the window must be member functions which can note this change of state and also member functions which actually perform the action (such as redraw in the new position).

Our screen typically has several windows displayed at any one time. To some extent, they may interact (overlap etc.) and yet each has private members which need not be known to (and so cannot be changed by) any external object. Further many objects are of the same type - all may have a name field, pull-down menus from the top, scroll-bars etc. Therefore we use a single class from which several instances can be generated. Note also that each object on screen is liable to comprise a number of sub-objects which are themselves instances of reusable classes - menus, scroll-bars, text fields etc. which can all be used by a variety of windows. Some data - that which is common to multiple instances of the class - will be **static.**

It should be clear that object orientation is a natural way to view window creation. The gain in conceptual conciseness more than outweighs any decrease in low-level programming efficiency.

5.3.2 Inheritance

All window environments have a library composed of a hierarchy of increasingly complex windows. We initially define a base window structure which is common to all windows and then subclass the base window, giving it new capabilities as needed.

The clear advantage of such a hierarchy is that e.g. we do not have to write a new initialisation member function to redraw the window each time we create a new window type. You will soon get a chance to browse the Visual C++ library.

5.3.3 Dynamic Binding

The ability to send different types of objects the same message at run-time is a major advantage of object orientation. While this is available to some extent in more traditional languages, it usually involves lengthy case statements to process diverse messages. As noted in the last lecture, it is then essential that the base class of the window hierarchy has the appropriate virtual members.

5.4 A First Application

We will split the creation of this application into 4 sections:

1. We first discuss the Windows95 system in general terms
2. We then give code to create a window of our own under Windows95
3. We give the code for a small enhancement of our first window
4. We use the Visual C++ system to create a window without our having to write code

5.4.1 Windows95 and Visual C++

We will during this lecture and the next concentrate on Visual C++ and Windows95. Windows95 has a set of functions which access operating system features known collectively as the Application Programming Interface (API) which our applications will use. In particular, there is a set of library functions known as the Dynamic Link Libraries (DLLs) which store some API functions in a relocatable format. Since these are relocatable functions, they do not need to be linked to each application as it is compiled but need only be loaded at run-time so saving disk space and relieving us of the need to recompile code when the libraries are updated.

When our application is running the user will be interacting with the interface by pressing buttons or keys or by moving the mouse. Each application[1] in Windows95 has its own event queue. Windows95 will call your program using a *window function* with the type of message which has been sent by the user. Your program is expected to handle all messages though perhaps with a default handler supplied by Windows. The function which windows uses to communicate with your application must return an instance of type `lresult callback` and the function itself is generally known as a *callback* function. The callback function is typically a large `switch` statement linking many different types of user actions to the appropriate code needed to respond to that action. All Window95 applications must establish a message loop inside the `main()` (actually `WinMain()`) function which will read any message from the application's event queue and despatch it back to Windows95 which will then call the window function. This central role of Windows95 allows it to maintain its ability to schedule CPU time.

Visual C++ comes with a set of functions which wrap the equivalent Windows API functions. Thus in writing our applications we do not need to go out of the Visual C++ system. We will begin, though, by writing an application from scratch to illustrate the main stages that would be needed if we did not have the Visual C++ system to interact with Windows for us.

5.4.2 A Basic Application

```
#include <windows.h>

class Myclass
{
WNDCLASS wcl;
char szWinName[20];
HWND hwnd;
void getResources(HINSTANCE hThisInst);
void register(void);
void create_window(HINSTANCE hThisInst);
public :
```

[1] Actually each thread

```
Myclass(HINSTANCE hThisInst);
int WINAPI perform(HINSTANCE hThisInst, int nWinMode);
void display_yourself(int nWinMode);
friend LRESULT CALLBACK WindowFunc(HWND,UINT,WPARAM,LPARAM);
};
```

The above header file defines a new class which will be a very basic windows class. `WNDCLASS` is a structure containing all the necessary fields which determine a window's appearance and capabilities. szWinName will hold the name of our class. hwnd is a handle to the current instance of the class. We have created one constructor which will be used to set the resources needed in the structure `wcl`, register the window with Windows95 and then create the window itself. `perform()` will display the window and set up the event loop. The function `WindowFunc()` is the *window function* described in the previous section. It is declared as a friend function in order that it can have access to our class' private members. You will note that we have used a number of new data types; these are defined in "windows.h"

```
#include <fstream.h>
#include <string.h>
#include "myclass.hpp"

Myclass::Myclass(HINSTANCE hThisInst)
{
getResources(hThisInst);
register();
createWindow( hThisInst);
}

void Myclass::getResources(HINSTANCE hThisInst)
{
/* first define the window class.
wcl is a struct which contains information necessary
to register and create the window */
strcpy(szWinName, "PaisleyWin");
wcl.hInstance = hThisInst;   // handle to this instance
wcl.lpszClassName = szWinName; // window class name
wcl.lpfnWndProc = WindowFunc;   // window function for this class
wcl.style = 0;                  // default style
wcl.hIcon = LoadIcon(NULL, IDI_APPLICATION);    // default icon
wcl.hCursor = LoadCursor(NULL, IDC_ARROW);      // default cursor
wcl.lpszMenuName = NULL;                        // no menu
wcl.cbClsExtra = 0;                             // no extra information needed
wcl.cbWndExtra = 0;
wcl.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH); // white background
return;
}


void Myclass::register(void)
{
/* register and check registration */
if (!RegisterClass(&wcl)) cout <<"Error in registration\n";
return;
}

void Myclass::createWindow()
{
/* now we can create the window */
      hwnd = CreateWindow(szWinName, // name of window class
```

```
 "Colin Object", // Title of object as displayed on title bar
 WS_OVERLAPPEDWINDOW, // normal window style
 CW_USEDEFAULT,  // X-coord - Windows95 can decide
 CW_USEDEFAULT,  // Ycoord - ditto
 CW_USEDEFAULT, // width - ditto
 CW_USEDEFAULT, // height - ditto
 HWND_DESKTOP,  // parent
 NULL, // no menu
 hThisInst,// handle of this instance
 NULL); // no addional arguments
return;
}



void Myclass::display_yourself(int nWinMode)
{
/* now display the window */
ShowWindow(hwnd, nWinMode);
UpdateWindow(hwnd);
return;
}


int WINAPI Myclass::perform(HINSTANCE hThisInst, int nWinMode)
{

MSG msg;
display_yourself(nWinMode);

// Now create the message loop
while (GetMessage(&msg, NULL, 0 , 0))
{ TranslateMessage(&msg);
DispatchMessage(&msg);
}

return msg.wParam;
}

/* This function will be called by Windows95 and passed messages from
the message queue . It is a callback function   */
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
switch(message)
{
case WM_DESTROY:PostQuitMessage(0);      // terminate the program
  break;
  // now let windows 95 deal with the other messages
default: return DefWindowProc(hwnd, message,wParam,lParam);
}
return 0;
}
```

In the above code, we are using defaults (macros defined in "windows.h") wherever possible. The `GetMessage()` loop will continue to repeat till there are no messages in the application's queue when control will pass back to Windows95. Each message is translated and dispatched to Windows95 which will hold it till it can be

passed to the callback function.

Inside the callback function we are only handling one event - a call from the user to terminate the application. All other input events will be handled by Windows95 using the default processing functions.

We will also have a definition file with this program. This one was generated automatically by Windows when I compiled the above code.

```
NAME       SECOND
EXETYPE    WINDOWS
CODE       PRELOAD MOVEABLE DISCARDABLE
DATA       PRELOAD MOVEABLE MULTIPLE
HEAPSIZE   1024

EXPORTS
;    ===List your explicitly exported functions here===
```

The name of my executable is "SECOND", it is a "WINDOWS" object and has the associated properties described in the file. Now we will need a main program which creates the instance.

```
#include ''myclass.hpp''

int WINAPI WinMain(HINSTANCE hThisInst, HINSTANCE hPrevInst, LPSTR lpszArgs, int nWinMode)
{
Myclass thisInst(hThisInst);

return thisInst.perform(hThisInst,nWinMode);
}
```

If we compile and link these files we get a perfectly good Windows application; it doesn't do anything yet but it will appear on screen. Let us now get it to do something e.g. to write out text.

5.4.3 An enhancement

We have described the creation of an application as

1. Define the window - what type of window, what resources it will have etc.
2. Register the window with a window manager.
3. Draw the window.
4. Manage events.

Now we have an application to which we wish to give an enhanced functionality i.e. it is only the fourth stage which we wish to enhance. We wish the window to take responsibility for keypresses, an event which has been defined by Microsoft as "WM_CHAR". To do so, we need only amend the callback function as follows

```
/* This function will be called by Windows95 and passed messages from
the message queue . It is a callback function   */
LRESULT CALLBACK WindowFunc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
char str[10];
HDC hdc;
switch(message)
{
case WM_CHAR: // this is how we process a keystroke
hdc = GetDC(hwnd); // get device context
TextOut(hdc,1,1,''  '' ,2); // erase previous character
sprintf(str,''%c'',(char)wParam); // put character into str
TextOut(hdc,1,1,str,strlen(str)); // output character
ReleaseDC(hwnd,hdc); // good practice, limited resource
```

```
break;
case WM_DESTROY:PostQuitMessage(0);      // terminate the program
  break;
  // now let windows 95 deal with the other messages
default: return DefWindowProc(hwnd, message,wParam,lParam);
}
return 0;
}
```

The application is now dealing with two type of events - a request from the user to terminate and a keypress. Similarly we can add event handlers to deal with events such as

WM_PAINT We will usually redraw the window when this is asked for

WM_CLOSE We should close the window for this one

WM_LBUTTONUP The left button has been raised and we should perform an action which, hopefully, the user will be expecting when he raises the left button.

You can see that writing such applications is code-intensive, memory-demanding and with a high potential for errors. Fortunately, Visual C++ provides us with a shortcut which takes the need to write a lot of code from us.

5.5 Using Visual C++

The Visual C++ workbench does all the hard work for you! You merely ask for a new project and can select from menus a number of options. The code given below was generated by Visual C++ workbench without selecting a single option - it is the default code for a project I called "THIRD" - the only 5 letters I typed into the environment. Visual C++ created the following header file

```
// third.h : main header file for the THIRD application
//

#ifndef __AFXWIN_H__
#error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols

/////////////////////////////////////////////////////////////////////////////
// CThirdApp:
// See third.cpp for the implementation of this class
//

class CThirdApp : public CWinApp
{
public:
CThirdApp();

// Overrides
virtual BOOL InitInstance();

// Implementation

//{{AFX_MSG(CThirdApp)
afx_msg void OnAppAbout();
// NOTE - the ClassWizard will add and remove member functions here.
//    DO NOT EDIT what you see in these blocks of generated code !
```

```
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////
```

Daunting isn't it. Yet it was created automatically and we do not need to do anything to it. Indeed there is a specific warning that we should not.

We have a new class with a constructor and two other member functions, `InitInstance()` and `OnAppAbout()`. There is also `DECLARE_MESSAGE_MAP()` which, we will see, is declared for each new class created. Let us look at the implementation file:

```
// third.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "third.h"

#include "mainfrm.h"
#include "thirddoc.h"
#include "thirdvw.h"

#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CThirdApp

BEGIN_MESSAGE_MAP(CThirdApp, CWinApp)
//{{AFX_MSG_MAP(CThirdApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
// NOTE - the ClassWizard will add and remove mapping macros here.
//    DO NOT EDIT what you see in these blocks of generated code!
//}}AFX_MSG_MAP
// Standard file based document commands
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
// Standard print setup command
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CThirdApp construction

CThirdApp::CThirdApp()
{
// TODO: add construction code here,
// Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CThirdApp object

CThirdApp NEAR theApp;

/////////////////////////////////////////////////////////////////////////////
```

```
// CThirdApp initialization

BOOL CThirdApp::InitInstance()
{
// Standard initialization
// If you are not using these features and wish to reduce the size
//  of your final executable, you should remove from the following
//  the specific initialization routines you do not need.

SetDialogBkColor();        // Set dialog background color to gray
LoadStdProfileSettings();  // Load standard INI file options (including MRU)

// Register the application's document templates.  Document templates
//  serve as the connection between documents, frame windows and views.

CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
IDR_THIRDTYPE,
RUNTIME_CLASS(CThirdDoc),
RUNTIME_CLASS(CMDIChildWnd),        // standard MDI child frame
RUNTIME_CLASS(CThirdView));
AddDocTemplate(pDocTemplate);
// create main MDI Frame window
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
return FALSE;
m_pMainWnd = pMainFrame;

// create a new (empty) document
OnFileNew();

if (m_lpCmdLine[0] != '\0')
{
// TODO: add command line processing here
}

// The main window has been initialized, so show and update it.
pMainFrame->ShowWindow(m_nCmdShow);
pMainFrame->UpdateWindow();

return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
CAboutDlg();

// Dialog Data
//{{AFX_DATA(CAboutDlg)
enum { IDD = IDD_ABOUTBOX };
//}}AFX_DATA

// Implementation
```

```
protected:
virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
//{{AFX_MSG(CAboutDlg)
// No message handlers
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
//{{AFX_DATA_INIT(CAboutDlg)
//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CAboutDlg)
//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
//{{AFX_MSG_MAP(CAboutDlg)
// No message handlers
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CThirdApp::OnAppAbout()
{
CAboutDlg aboutDlg;
aboutDlg.DoModal();
}

/////////////////////////////////////////////////////////////////////////////
// CThirdApp commands
```

There is a constructor in which we are invited (`TO DO:`) to place any additional code for the construction of an object. Notice that there is a suggestion that any major initialisation code should be placed in `InitInstance()`. In `InitInstance()`, we set the resources for the object, create a new object and display it. However, there is no code for the callback function. This has been taken care of by the `MESSAGE_MAP` which appears at the start of the file. This acts just like the large switch statement in the callback function and is read and parsed dynamically to give the appropriate response. Notice that the inheritance hierarchy is used explicitly in that e.g. `ID_FILE_OPEN`, the macro defined with the event of asking for a file to be opened, is associated with the base class function, `CWinApp::OnFileOpen()`.

Finally notice that the `About...` dialog pop-up box is created as a separate class at the end of this file. It has little functionality yet it is constructive to watch how it is used. The `MESSAGE_MAP` links the message `ID_APP_ABOUT`, which will be sent when the user selects `About..` from the `Help` menu, with the function `onAppAbout()`. `onAppAbout()` creates a new instance of `CAboutDialog` and this calls its base class, `CDialog`'s member function, `DoModal()` which pops up a dialog box with the `About` information. But how does the `About` message get to be displayed. The clue is in the constructor for the `CAboutDialog` which calls the constructor for the `CDialog` class with the parameter `IDD` which has been initialised to `IDD_ABOUTBOX`, a standard macro which contains the description of the about dialog box.

There are also for this class two other classes, `CThirdDoc` and `CThirdView` which are used for document handling and viewing respectively. We will now briefly investigate these and the resource and definition files associated with the project.

5.6 Documents, Views and Frames

The executable file created for the above application was built from compiled versions of 4 files. We have

1. The actual application class which is held in "third.cpp". The application object controls the other objects in the application and initialises the creation of the other classes.

2. A document class whose implementation is in "thirddoc.cpp". A document is a data object which the user may view or update during a session. Typically such objects are held in files between sessions. This will (usually) be a subclass of `CDocument` or `COleDocument`.

3. A view class whose implementation is in "thirdvw.cpp". The view is a window object through which the user interacts with a document. This will usually be a subclass of `CView` or its subclasses such as `CScrollView` or `CEditView`.

4. A frame window which forms the external window shell for the view. This is held in "mainfrm.cpp".

Each of these objects must communicate with the others and so each has member functions which return pointers to the others. The objects together are held in a framework which uses the same simple message loop as before though we do not have to get involved with many of the messages since they are handled by the framework automatically for us. For example, since the framework can close windows and exit from the application, we do not have to write member functions that handle such commands. Notice that the `WinMain()` function is hidden from us: it actually exists in the `Run()` member function of `CWinApp`.

When a message is sent by the user it is first of all received by the main frame window, which will attempt to despatch it to the appropriate object to deal with; as a simplification we can think of the objects being tried in the order `view`, then `document` then `application` then `frame window`. If none of these objects specifically handles the event, it will be passed to Windows' default handler.

Whether an object will respond to a message or not depends on whether the message appears in its message map. We see from the above that our `CThirdApp` object above has a mapping from the message `ID_FILE_NEW` to its parent's member function `CWinApp::OnFileNew()`. Each object has its own message map which pairs a message or command with the name of its handler function. Notice however that message maps are inherited too so that `CThirdApp` objects automatically have all the functionality associated with the `CWinApp` class. This is set up in the first line `BEGIN_MESSAGE_MAP(CThirdApp, CWinApp)`. All message maps are parsed by the C++ preprocessor.

5.7 Controls

"Controls" is the term used by Microsoft as the generic name for visual objects which will input user data and/or output program results. There are several types of controls including

- Static graphic
- Static text and edit box
- Buttons such as Pushbutton, Radio button, Checkbox
- Combinations such as List box, Group box and Combo box
- Horizontal and vertical scroll bars

These can be created and edited by selecting the resource editor part of the project page. We will give 2 examples of using controls in this lecture. We will build them in stages and the notes for this lecture will emphasise these stages.

5.8 Practical Exercise

In this lecture we will create a Windows95 application with a Windows GUI. We shall use the circle class which we created back in Lecture 1 but put a modern interface on it.

5.8.1 The Circle class

First the circle class' definition is repeated here for convenience.

```
// "circle.h"
// interface for class circle
#ifnef _CIRCLE
#define _CIRCLE


class Circle
{
double centre_x; // x-coordinate of centre
double centre_y; // y-coordinate of centre
double radius;
double area;
double circumference;
double howLong(double first,double second);
public:
Circle(void);
Circle(double first, double second, double third);
void setRadius(double aRadius);
double getRadius(void);
void setCentre(double first,double second);
double getXCentre(void);
double getYCentre(void);
void calculateArea(void);
double getArea(void);
void calculateCircumference(void);
double getCircumference(void);
int isInside(double first,double second);
int isOn(double first, double second);
int isOutside(double first, double second);
};

#endif
```

We have already designed our `Circle` class and do not wish to redesign it; our decision now is as to how to add in our interface classes:

1. Add new member variables to the circle class where these variables are subclassed from `CWinApp`

2. Add the instance of `Circle` to the interface class

3. Keep as two separate objects and give them references to each other

The last choice is often thought to be best since it leads to code in which the interface or the interfaced object can be independently modified. However, Microsoft suggests that the standard way to do this is the second and so we shall follow that method.

We must make one change to our existing code and that is to add the include file "stdafx.h" to the "circle.cpp" file *before* the include file "circle.h".

```
#include ''stdafx.h''
#include ''circle.h''
```

This is necessary because the visual applications require some standard headers; an alternative would have been to compile the `Circle` class using different options.

5.8.2 Create the basic interface

Our interface should have 3 Edit boxes for the user to enter the x- and y-coordinates of the centre and the radius. Each Edit box should have a static text explanation of what is required of the user. In detail,

1. Select `File` and `new`. Opt for `project`. Give the project name as `circ`. We will need a `Dialog based` interfaces but all the rest of the defaults can be accepted. Create the application. It can be built and run (though of course it has no functionality yet).

   When a new dialog box is created, Visual C++ assigns it a unique symbol name, ID value and caption all of which are editable though, in practice, it is usually only necessary to change the last of these.

2. We wish to edit resources so we view the file "temp.rc". If the control resource editor is not in view, select `View, toolbars` and click on the `resource` box associated with `controls`. Click on `IDD_CIRC_DIALOG`. We wish to create a new static text with text "centre: x-coordinate". Select the `Aa` button. You can either drag and drop the static text or simply point and click on the dialog to get the new text to the correct position. If you decide to remove any object having created it simply select the object by pressing once on it and select `Edit` and the `delete`. Start to type in the text for this object. A pop-up window appears giving you the choice to edit other parameters associated with the new text object but we ignore these. When you press return the window disappears and the new static text object appears on the dialog. It can be repositioned by drag and drop or resized by clicking on the sizing handles and moving the mouse. Now select the `ab` button to get an Edit box. Repeat for the other two static text objects "centre: y-coordinate" and "radius". It perhaps looks better if the `OK` and `Cancel` buttons are at the bottom of the dialog.

3. Select `Project` and `build` followed by `execute` to have a look at your handiwork. Notice that the edit boxes will accept any text input.

4. Now we wish to give our application some functionality: Visual C++ will create variables for us which will mirror the contents of the Edit boxes *and* will provide us with the functions which will do the transfer from the Edit boxes to these variables for us. Select `Project` and `Class Wizard`. A tabbed pop-up page will appear. Select the `Member Variables` page. Click on `IC_EDIT1` to highlight it and `Add variable...` You are given a new page with the cursor at the name of the variable (which the Microsoft convention states should begin with m_). The box IC_EDIT1 is the identifier of the box in which we wish to put the circle's centre's x-coordinate. We therefore call this variable `m_xcoord` and change its type to double. Repeat with IC_EDIT2 and IC_EDIT3. Since we wish to limit the radius' size give it a maximum size of 100 and minimum of 0.

5. Build the application and run it. Notice now that you get an error message if you enter anything other than numbers into the Edit boxes. In addition, you can only enter numbers between 0 and 100 in the radius box. You will also find that the tab keypress moves you between the boxes.

6. Now we will examine the code which visual C++ has given to us. Open "circdlg.h". We now have public member variables declared withing an `AFX_DATA` clause

   ```
   // Dialog Data
   //{{ AFX_DATA(CCircDlg)
   enum{IDD=IDD_CIRC_DIALOG};
   double m_xcoord;
   double m_ycoord;
   double m_radius;
   //}}AFX_DATA
   ```

   The AFX_DATA declares that this data can be used in the dynamic data exchange routines built into the API[2]. Now look at the code in "circdlg.cpp". In the constructor of `CCircDlg`, the values of these variables are initialised to 0 again within an `AFX_DATA_INIT` clause. The actual data exchange is done by the member function `DoDataExchange()` which, again, has been written for us. It mainly contains an AFX_DATA_MAP clause which associates each variable with the corresponding Edit box contents with calls to an API function, `DDX_Text()` e.g.

[2]When we refer to the API functions in this lecture, we are really referring to the Visual C++ wrappers for the MS Windows API functions

```
DDX_Text(pDX, IDC_EDIT1, m_xcoord);
```

The pDX variable is a structure which contains information about the type of data being transferred and the direction of the transfer.

Notice that the upper and lower limits which we built into the radius have appeared as

```
DDV_MinMaxDouble(pDX,m_radius, 0.,100.);
```

Before going on we should consider how little code writing we have done. By using our knowledge of the library of material available to us we have quickly constructed a robust interface.

5.8.3 Get the information to a Circle

We now have an interface which is transferring information from the user to variables which we can manipulate in the dialog class. We must now link it to the actual centre and radius of an instance of the circle class.

1. If you have not already done so, copy "circle.h" and "circle.cpp" to the current directory. Add the line `#include ''stdafx.h''` before `#include ''circle.h''` in the file "circle.cpp".
2. In "circdlg.h", `#include ''circle.h''`.
3. Add a new private member variable to the class `CircDlg`:

   ```
   private:
   Circle theCircle;
   ```

   and a new public member function

   ```
   public:
   void renewCircle(void);
   ```

4. Open "circdlg.cpp"; we define the implementation of the new function as

   ```
   void CCircDlg::renewCircle(void)
   {
   theCircle.setRadius(m_radius);
   theCircle.setCentre(m_xcoord, m_ycoord);
   return;
   }
   ```

5. We could call the function `renewCircle()` from the end of the function `DoDataExchange()` i.e. just after the user's data has been assigned to the variables `m_xcoord` etc. Try this. The difficulty with this is that this function is called only when the user presses the `OK` button which promptly closes the application.
6. The `OK` button calls a function `OnOK()` which we inherit from the base class, `CDialog`. Let us override that function with

   ```
   void CCircDlg::OnOK(void)
   {
   if (!UpdateData(TRUE))
   {
   TRACE0(''Update data failed during transfer of data to circle\n'');
   return;
   }
   renewCircle();

   return;
   }
   ```

which we must clearly declare in the interface file, "circdlg.h". The `UpdateData()` function is an API function which calls the `DoDataExchange()` function. If it fails, the `TRACE0` macro will be called for debugging purposes. If it succeeds we will call the function `renewCircle();`.

7. Build and run the application. Pressing the OK button does not now cause the window to close since it is now calling our `OnOK()` function which does not include that feature.

5.8.4 Showing the results

Now we have the interface communicating with the `Circle` object but not the other way round. The function `UpdateData()` which we introduced last section takes one parameter. If the parameter is TRUE, it calls the `DoDataExchange()` function with a structure which states that communication is to be from user to dialog; if the parameter is FALSE, the communication is in the opposite direction.

1. Open "circ.mak", the project file and select "circ.rc" the resources and from that `IDD_CIRC_DIALOG`.
2. Enter two new static text objects with the legends "Area" and "Circumference" respectively and beside each an Edit box.
3. Select `Project` and `Class Wizard..` and choose `IDC_EDIT4` and `Add variable...` Give the new variable the identifier `m_area` and change its type to double. Repeat with `IDC_EDIT5` and `m_circumf`.
4. Again this can be built and run.
5. As before we find that the two new variables have been declared as public member variables in the AFX_DATA clause of the class definition and initialised in the class constructor. Similarly the AFX_DATA_MAP has been updated to include the new variables.
6. In the function `renewCircle()` add the lines

```
theCircle.calculateArea();
theCircle.calculateCircumference();
m_area = theCircle.getArea();
m_circumf = theCircle.getCircumference();
```

7. We now must ensure that when the user presses the `OK` button the results are displayed in the appropriate windows. The `DoDataExchange()` function is already exchanging data in one direction for us. Fortunately it can be used in the other direction also. We do this by using the function `UpdateData()` with the argument FALSE.

```
void CCircDlg::OnOK(void)
{
if (!UpdateData(TRUE))
{
TRACE0(''Update data failed during transfer of data to circle\n'');
return;
}
renewCircle();

if (!UpdateData(FALSE))
{
TRACE0(''Udate data failed during transfer fo data to interface\n'');
return;
}

return;
}
```

8. Finally that `Cancel` button now seems wrong. It should say `Finish`. So select the project window, `colin.mak`, and from the resources select the `IDD_CIRC_DIALOG`. Double click on the `Cancel` button and the change parameters window appears. Change the string which appears on the button.

5.8.5 Conclusion

We have created a simple application with a minimum of code writing. It should be clear how to easily add additional Edit boxes to deal with input of other points and how to ask the circle if such points lie on it etc.

The method has used just a single class of controls for getting from and giving information to a user. The speed of creating such an interface *given a library of interface classes* adds weight to the claim that OO is ideal for *fast prototyping.*

5.9 A Second Application

We will now construct a second application. While there will inevitably be differences with the first application, we should note the striking similarities between these two applications. This should suggest, rightly, that creating this type of application is really only making variations on a theme.

We will create a class to manage a user's bank accounts. He will have 3 accounts -a current account, a deposit account and an investment account. He will be able to deposit and withdraw sums from each. Since the aim of this lecture is an investigation of simple controls we will use different types of controls in this lecture but you will find that the method is very similar to that used for the first application.

5.9.1 Creating the Interface

1. Select `new, project` of type `dialog based` and call the project `account. Create` the project noting how similar the files constructed are to those in the first application.

2. From the Project window (account.mak) select `account.rc`, then `dialog` and finally `IDD_ACCOUNT_DIALOG`. Use the `TODO:` box to get an appropriate title for the dialog "ACCOUNT MANAGER".

3. Select 3 radio buttons. Place then in a vertical line on the left side of the dialog. Place a group box round them. Click twice on the group box and give it the title "Type of account". Click twice on `Radio1` and name it `current`. Name `Radio2 deposit` and `Radio3 investment`.

4. Select an Edit box and place it centrally in the dialog. Select Static text and place it just above the Edit box giving it the title "Balance".

5. Select two more Edit boxes placing them side by side and under each place a pushbutton. Click twice on the pushbutton and rename it `Deposit`. Rename the other `Withdraw`.

6. Build and execute the application. You will note that the radio buttons are mutually exclusive already (though of course they do not do anything yet). The only two buttons which do anything are `OK` and `cancel` which inherit functions from the `CDialog` class.

5.9.2 Mapping the Interface

We will now as before associate the Edit boxes with variables and the buttons with functions. The sole difference this time is that we have new buttons to manage.

1. Select `Project` and `Class Wizard`. Choose `IDC_BUTTON1` select `BN_SINGLE_CLICKED` and `Add Function`. Change the name in the popup window to `OnDeposit`. We shall see that there will be an entry in the `AFX_MAP` which associates the single clicking of this button with member function `OnDeposit()` of the class `CAccountDlg`. Similarly associate `IDC_BUTTON2` with `OnWithdraw()`.

2. Select `IDC_RADIO1`, select `BN_SINGLE_CLICKED` and `Add Function`. Add the identifier `OnCurrentAccount`. Similarly associate `IDC_RADIO2` with `OnDepositAccount` and `IDC_RADIO3` with `OnInvestmentAccount`.

3. Select the `Member Variables` page. Select `IDC_EDIT1` and `Add variable`. Insert the identifier `m_balance` and change its type to float. Similarly associate `IDC_EDIT2` with `m_deposit` and `IDC_EDIT3` with `m_withdraw`.

5.9.3 Coding the Functions

1. Open the file "accountdlg.h". We will find that Visual C++ has created 3 new member variables (again defined within the `AFX_DATA` clause). There is also this time, within the `AFX_MSG` clause, the functions which we have just associated with the button presses. You will find declared:

   ```
   afx_msg void onDeposit();
   afx_msg void onWithdraw();
   afx_msg void onCurrentAccount();
   afx_msg void onDepositAccount();
   afx_msg void onInvestmentAccount();
   ```

   These look just like ordinary function declarations but are prefixed by the `afx_msg` type name. This is a preprocessor identifier and is replaced by white space in the code which reaches the compiler. However we cannot use these member functions unless through an `AFX_MSG_MAP` clause.

2. Open the file "accountdlg.cpp" and check that the new member variables are initialised in the `OnInitDialog()` method and that the new member functions appear (although they are currently simply stubs). Note that `AFX_MSG_MAP` for the CAccountDlg contains all these functions correctly mapped from the appropriate buttons.

3. In "accountdlg.h" create a new global enumeration type

   ```
   enum ACCOUNTTYPE {current, deposit, investment};
   ```

   and then inside the class definition declare new private variables

   ```
   private:
   ACCOUNTTYPE theAccount;
   float currentBalance, depositBalance, investmentBalance;
   ```

4. In "accountdlg.cpp" initialise these new variables within the `OnInitDialog()` method.

   ```
   currrentBalance = depositBalance = investmentBalance =0.0f;
   ```

5. When the user presses the `Current` account radio button we wish to show the balance as we have it at present for the current account. In the `OnCurrentAccount()` method enter

   ```
   theAccount = current;
   m_balance = currentBalance;
   m_deposit = m_withdraw = 0.0f;
   UpdateData(FALSE);
   ```

   Recall that the `UpdateData()` method calls the `DoDataExchange()` method with a structure which indicates in which direction data is to be exchanged. When its argument is FALSE, information flows from the member variables `m_anthing` to their associated Edit boxes.

6. Make the obvious corresponding changes to `OnDepositAccount()` and `OnInvestmentAccount()`.

7. When the user presses the `Deposit` button we wish to increase the balance of his currently selected account by the amount of the Edit box above the `Deposit` button. Therefore we `UpdateData(TRUE)` to get the contents of this Edit box, carry out the calculations and then `UpdateData(FALSE)` to return the balance information to the user.

   ```
   void CAccountDlg::OnDeposit()
   {
   if (!UpdateData(TRUE))
   {
   TRACE0(''Update data failed during transfer of deposit to account\n'');
   return;
   ```

```
}
switch(theAccount)
{
case current:
    currentBalance += m_deposit;
    m_balance = currentBalance;
    break;
case deposit:
    depositBalance += m_deposit;
    m_balance = depositBalance;
    break;
case investment:
    investmentBalance += m_deposit;
    m_balance = investmentBalance;
    break;
}
m_withdraw = 0.0f;

if (!UpdateData(FALSE))
{
TRACE0(``Update data failed during update of interface\n'');
return;
}
```

8. Similarly update `OnWithdraw()`:

```
void CAccountDlg::OnWithdraw()
{
if (!UpdateData(TRUE))
{
TRACE0(``Update data failed during withdrawal from account\n'');
return;
}
switch(theAccount)
{
case current:
    currentBalance -= m_withdraw;
    m_balance = currentBalance;
    break;
case deposit:
    depositBalance -= m_withdraw;
    m_balance = depositBalance;
    break;
case investment:
    investmentBalance -= m_withdraw;
    m_balance = investmentBalance;
    break;
}
m_deposit = 0.0f;

if (!UpdateData(FALSE))
{
TRACE0(``Update data failed during update of interface\n'');
return;
}
```

9. Compile and run the application. Make sure that you can understand the flow of messages caused by any user action.

5.9.4 Conclusion

These two very simple examples should have highlighted the main features of creating this type of application:

1. Create a new (dialog based) project. Visual C++ creates all necessary files.
2. Use the resource editor to add/delete resources as necessary
3. Use the `Class Wizard` to associate the resources with members of the classes.
4. Add specific code to perform the particular functions you need.
5. Compile, link and execute the code

5.10 Further Exercises

1. Make an interface for your `Rectangle` class. Add the additional functionality of asking whether a point is on it, in it or outside it.
2. Make an interface for your `Account` class to allow a user to enter or withdraw funds.

Lecture 6

Smalltalk -an Introduction

Smalltalk is one of the oldest OO languages. It was designed specifically to make the process of abstracting a model from the real world easy to do; this is achieved by making the syntax of Smalltalk as simple as possible. A user need only learn a very limited set of instructions to make it possible to create a working application. However the mindset required to develop Smalltalk programs is even further away from that needed for imperative languages than that needed for C++.

The simple syntax of programs does make *rapid prototyping* possible: the argument for rapid prototyping is that, since a user rarely knows in advance precisely what functionality is required in his application, it makes little sense to devote lots of time to intricate initial versions of the program. The tactic then is to give a user a quickly constructed, cut-down version of the application with which he/she can play to get a feel for the possibilities inherent in the application.

If the system is inherently wrong for the functionality that was desired we will have no hesitation in junking it - better to do so than to continue to add patches to a faultily designed system. If the basic design is good, we can merely add bells and whistles as they become necessary.

Smalltalk is an interpreted language - not designed for efficiency but for ease of use. This is sometimes described as a strong point of Smalltalk since it allows run-time interaction with users and facilitates dynamic changes to applications.

Smalltalk is typeless - we do not declare any object as being of a particular type. In C++, the compiler sets aside a section of memory for each instance of a class as it is declared. We can then associate the value of the instance with its identifier through a tag which indicates its type. If you have only a small number of types you can have a small (e.g. one byte) tag. However in Smalltalk you have a potentially unlimited number of classes and so the tag method could become cumbersome. Therefore the method used is to create for each class an object(the class object) which maintains information about the class.

Objects in Smalltalk do not go out of scope and so we have a potentially unlimited set of objects. Therefore a sophisticated memory manager is an important part of Smalltalk.

We will be using a public domain Smalltalk and so will describe its flavour of Smalltalk. This is not so very different from commercial versions of the language but lacks such features as GUI classes, object browser, debugger and so on. It is not therefore fair to compare this system with Visual C++.

6.1 Basic Smalltalk Syntax

To keep the language syntax as concise as possible, the developers made everything in Smalltalk an object: variables, executable procedures (member functions which in Smalltalk are known as "messages"), stack frames (when functions are called) etc. Every class is descended from the class `Object` either indirectly or directly (if you do not declare a new class' superclass, it is, by default, `Object`).

To perform any function, you must send an object a message (but only one which it "knows how to perform"). Thus

```
anObject print
```

is sending the message `print` to an object, `anObject`. `anObject` can be an object of a user-defined class such as `Circle` or of a class which is built in to the library such as `Object` or `Integer`.

6.1.1 Classes

Some of the basic Smalltalk classes are

Object
- **UndefinedObject**
- **Symbol**
- **Boolean**
  - **True**
  - **False**
- **Magnitude**
  - **Char**
  - **Number**
    - **Integer**
    - **Float**
  - **Radian**
  - **Point**
- **Random**
- **Collection**
  - **Bag**
  - **Set**
  - **KeyedCollection**
    - **Dictionary**
      - **Smalltalk**
    - **SequencableCollection**
      - **Interval**
      - **List**
        - **Semaphore**
      - **File**
      - **ArrayedCollection**
        - **Array**
        - **ByteArray**
        - **String**
  - **Block**
  - **Class**
  - **Process**

The diagram is intended to display that `Object` is at the root of all classes and e.g. that `Dictionary` is derived from `KeyedCollection` which is derived from `Collection` which is derived from `Object`. Note this major difference with C++ - these classes are part of the language and guaranteed to be available with every smalltalk interpreter.

There are also some pseudo-variables which are defined throughout the system:

- **self**: a pointer to the current object (cf `this`)
- **super**: which will temporarily be thought of as a pointer to the superclass of the current class.
- **nil**: the single instance of `UndefinedObject` (cf `NULL`)
- **true**: the single instance of `True`
- **false**: the single instance of `False`

The first two are simplifications which will require to be expanded upon later.

6.1.2 Basic Syntax

A string is any set of characters within single quotes e.g. 'a string'. We can concatenate stings by using the comma operator e.g. 'a ','string'.

An array is identified by the # sign followed by the array members in brackets e.g. #(1 2 3 4) or #(this is an array) have both four elements.

In the above we have instances of class `String` and `Array`. Another important class is the `Symbol` class whose members are symbols written as the # sign followed by any sequence of characters. e.g. #aSymbol #anOther #a1 #***. Notice that symbols can neither be broken down into parts (unlike strings) nor compared.

A block of code is delimited by square brackets [] as in

```
[i := i+1. i print]
```

This decreases the value of i by one and prints out this new value. Notice that the block contains two expressions separated by the "." If the block begins with one or more ":" s the expressions following the colons act like parameters passed into the block. Thus

```
[:a:b | a print. b print. (a+b) print]
```

prints out the values of a, b and a+b where the first two of these are passed into the block.

"This is a comment".

6.1.3 Message Sending

Typical messages are

```
thisObj print
nextArray add: notherObj
newSet add: thisObj at: newObj
```

Each message is composed of three parts:

a receiver - the object which is getting and must respond to the message. In the above cases the receivers are `thisObj, nextArray, newSet`.

a selector - the identifier representing the message. In the above, we have a *unary* selector, `print` which takes no arguments, a *binary* message selector `add:` and a ternary selector `add:at:`.

an argument - of which there may be zero or more.

It is important to be clear why the receiver is deemed to be part of the message: the actual implementation of the message depends on all three parts since e.g.

```
thisArray doThis
```

is a different message from

```
thisString doThis
```

since the receivers are different and will typically use different code in implementing the message. The sole exception to this is if the `doThis` message is inherited from a common base class such as `Object`.

Sometimes it is useful to distinguish between **keyword** messages such as those above and **operator** messages such as the arithmetic operators. The arithmetic operators have higher precedence than the keyword messages but not so high as the unary messages. Thus

```
7 between: 2 sqrt and: 4+2
```

is parsed as

```
7 between: (2 sqrt) and: (4+2)
```

i.e. the object 2 is sent the message `sqrt` and returns 1.41..., the object 4 is sent the message `+` with the argument 2 and returns 6, and then the object 7 is sent the message `between:and:` with the arguments $\sqrt{2}$ and 6 and will return `false`. When messages have equal precedence, they will be activated from left to right; we must beware of this for numerical operations e.g.

```
3 + 5 * 2
```

will be implemented as

```
(3 + 5) * 2
```

6.1.4 Querying Objects

Because there is no compile-time type-checking in Smalltalk, it is often necessary to check an object's type dynamically.

We can query objects using methods described in `Object`. For example,

```
3 class
```

asks the object 3 to which class it belongs. We can ask its class what class it is derived from by

```
3 class superClass
```

or equivalently

```
Integer superClass
```

We can ask an object directly if it is a member of a particular class using

```
3 isMemberOf: Integer
```

or if it is in some way derived from a class using

```
3 isKindOf: Number
```

Both of these examples return `true`. We can also query objects as to whether they respond to particular selectors e.g.

```
3 respondsTo: #+
3 respondsTo: #between:and:
3 respondsTo: #sqrt
```

asks the object 3 if it responds to the messages `+`, `between:and:` and `sqrt`. We can alternatively query the class directly as to whether instances respond to the messages by

```
Integer respondsTo: #+
```

Finally each class has a class comment which is a string which describes its instances e.g.

```
(Object comment) printNl !
```

gives an output of

```
I am the root of the Smalltalk class system.
All classes are subclasses of me
```

6.2 Using Smalltalk Interactively

The first and most obvious way to use the smalltalk environment is in an interactive way. We call up the smalltalk environment by simply clicking twice on the `smltk` file in the directory `mst111`.

The Smalltalk environment will be loaded and if it is the first time that you have done this the standard library will be compiled and added to the current environment. On subsequent occasions, this environment will be automatically loaded.

6.2.1 Standard Output

```
'Hello, world' printNl !
```

The Smalltalk environment collects all text input from standard input till it reaches a "!" at which point it interprets and executes the code. Here we are creating an instance of the String class and sending it a message to print itself. This is a typical Smalltalk ruse - each object has the responsibility of knowing how to print itself. The `printNl` message (literally print and new line) calls the method `printOn:` with the argument "stdout". Since `printOn:` is defined (at least) for the Object class, we see that this method is guaranteed to have some implementation for each object. We will later see that `printOn:` is a method which is very commonly overridden in each new class we create. The thing we must emphasise here is that we did

not write code which spelled out how to print the String object; we used the fact that the object knew how to print itself. It does this by going to a table (the actual table is determined by the object's type) and searching for an entry `printNl`. If it does not find the message in the class table it will search in the class' superclass' table etc. If it reaches the top of the class hierarchy and still does not find the message, you will get an error message e.g.

```
'Hello, world' show !
```

will print out an error message

```
'Hello, world'  did not understand the selector 'show'
```

In each case there is also information on the speed of execution which we shall ignore.

Now one advantage of such a method is that we can use the same method for every object in our system. This cuts down on our cognitive loading. e.g. the two statements

```
3.14 printNl !
5 printNl !
```

are also executed since floating point numbers and integers also know how to perform the method `printNl`. Now we might guess that the code which prints out an integer is very different from that required to print out a string but we, as users, need never get involved in the details - each object deals with its own printing.

Since white space is ignored, we can equally well write

```
'Hello, world'
printNl
!
```

which has exactly the same effect as the first statement. This is sometimes useful when we have complex expressions to evaluate and do not wish anything other than the final answer.

6.2.2 Collection Classes

One of the most useful features of OO languages/compilers is the pre-defined collection classes. To create a new instance of such a class in Smalltalk, we use the `new:` method where the argument is the number of elements in the class. Thus we have

```
x := Array new: 20 !
```

Notice first that we are now sending the message `new:` to the *class* Array. This is explainable by noting that `Class` is a sub-class of `Object` i.e. every instance of Class (e.g. the class String, the class Array, the class Object etc.) is itself an object in the system and can handle messages just as any other object in the system can. If this "chicken-egg" seems too circular we must just accept it. The main use of classes is the generation of instances of the class which is what is happening here: we are creating a new instance of Array with space for 20 members. Collection classes typically start at 1 and count upwards[1]. You will note that we did not have to specify what type of object would be found in the array (contrast this with the highly-typed language C++). We can now put new objects into the array:

```
x at: 1 put: 1234 !
```

which puts the integer 1234 into the first position of the array and

```
(x at: 5) printNl !
```

which prints out the 5^{th} member of the array. Notice that the method `at:` takes only one argument while `at:put:` takes two arguments - the number of colons determines the number of arguments. Some Smalltalk interpreters have error checks built in to check array bounds, the Gnu one doesn't.

Each collection class has the semantics we would expect from the class:

```
x := Set new:10 !
x add: 3 ; add: 'hello' ; add: true ; add: 3 ; add: 3 !
```

[1]Though it is possible to override this

The ";" syntax merely allows us to send more than one message to the first receiving object "x". Thus we are asking x to add the integer 3 (three times), the string 'hello' and the pseudovariable `true`. If we now write

```
x printNl !
```

we will get

```
Set ( 3 'hello' true)
```

We can also remove objects from a Set (or Array etc.) using

```
x remove: 3 !
```

and ask if the set includes a particular object using

```
( x includes: 5) printNl !
```

A very important collection class is the Dictionary class with which we store a set of Associations e.g.

```
x := Dictionary new !
x at: 'dog' put: 'puppy' !
x at: 'cat' put: 'kitten' !
x at: 'horse' put: 'foal' !
(x at: 'horse' ) printNl !
```

creates a new instance of Dictionary and puts 3 Associations within it: each association consists of a key-value pair where the key is specified by the first argument of the `at:put:` method and the value is the second. We then query the dictionary and expect to have the string 'foal' returned. We can ask the dictionary to print itself out (but this becomes increasingly unrealistic the larger the dictionary becomes).

6.2.3 Registering variables

We have used the variable "x" extensively above; however there is a prior need to register the variable with the system. This is done using

```
Smalltalk at: #x put: 0 !
```

Here we are registering the symbol x with the system and initialising it to 0. Smalltalk is a pseudo-variable which has been preset to an instance of Dictionary. If you know what you wish x to represent you may use e.g.

```
Smalltalk at: #x put: (Array new:100) !
```

Notice that the lack of typing in Smalltalk has allowed us to associate x with an integer, a string, a set or an array without any need for redeclaration.

6.3 Creating New Classes

This section owes much to the public domain tutorial "A tutorial for GNU Smalltalk" by Andrew Valencia.

We will now create a new class of object to model a bank account. We will call the new class Account. An instance of Account will have one instance variable showing the balance in the account. Thus we write

```
Object subclass: #Account
instanceVariableNames: 'balance'
classVariableNames: ''
poolDictionaries: ''
category: nil !
```

We are here telling the Object class (the ultimate root of the system) that we are adding a new subclass `Account` instances of which will have a variable `balance`. We will document our new class using the `comment:` message.

```
Account comment: ' I represent a place to deposit and withdraw money' !
```

We will now create a method for instance creation:

```
!Account class methodsFor: 'instance creation' !

new
| r |
r := super new.
r init.
^r
!!
```

The first line states that the messages which are defined in this section are messages which will be sent to the *class* Account (rather than instances of the class). Thus

```
 x := Account new !
```

will construct a new instance of the class Account. The string 'instance creation' is for documentation purposes. The actual method begins with the identifier `new` and continues till the "!!". We create a local variable "r" and assign it to the value returned when the method `new` is sent to the object "super". Now `super` is the same object to which the current message is sent but any message sent to super is to be looked for first in the superclass (in this case `Object`) of the new class. Notice that this method returns an instance of `Account` not `Object` - the `super` merely says begin to look for the method in the superclass of the current class. We then send to r the method `init` (not yet defined) and return r, the new instance of the class `Account`. Notice that the separator "." is not necessary after `^r` .

Now we must initialise our new instance:

```
!Account methodsFor: 'instance initialisation' !
init
balance := 0
!!
```

Notice that this is an instance (not a class) method. Since we do not return anything the default object is returned; the default object is the receiver. This allows us to concatenate methods such as:

```
 x := Account new !
((x init) deposit:50) spend:10 !
```

You can alternatively specifically return `self`.

Notice that we have not had to declare "balance" to be private; the only methods which can access "balance" are those which we provide.

Now we will add methods for changing the value of our balance:

```
!Account methodsFor: 'transferring money' !
spend: amount
balance := balance - amount
!
deposit: amount
balance := balance + amount
!!
```

Note that we have never stated what type of Object "balance" should be yet we are here sending it messages "+" and "-" with arguments "amount" each time. Certainly we are initialising "balance" to 0 but have not restricted the variable "amount" in any way - this must be a run-time check with the concommitant possibility of a run-time error. If we now try to look at an instance of `Account` using

```
x printNl !
```

we get the uninformative message

```
an Account
```

which is provided by `Account`'s superclass, `Object`. We must specifically add methods such as

```
!Account methodsFor: 'printing' !
printOn: stream

super printOn: stream.
' with balance: ' printOn: stream.
balance printOn: stream
!!
```

Notice the "wing and a prayer" type of programming again. We first use the `printOn:` method from the super class, `Object`. This gives us "an Account". We then create a String object containing " with balance:" and ask it to print itself on "stream". Finally we ask `balance` to print itself on stream. We are relying on all objects currently in the system responding to the method `printOn:`.

6.4 Reading Files

The method of interactive processing is very good for small programs but quickly becomes a bind if you have to type (without errors) a large program. The simplest way to do so is to type in a file (perhaps using simple Notepad) and then enter at the `st>` prompt something like:

```
FileStream fileIn: 'myfile.st' !
```

The full absolute address of the file can be used if you are not writing in the current directory from which `smltk` is being run.

This method is perfect for creating and compiling long classes but is just as viable for the simple file

```
Smalltalk at: #anything put: 0 !
anything := Set new !
anything add: 'one' !
anything printNl !
```

Reading this file will result in the output `Set('One')`. An example of a file which prints out the Smalltalk hierarchy is the file "prthier.st" in the directory "mst111". We will look at file handling in more detail next week.

6.5 Laboratory Tasks

1. Interactively test the smalltalk interpreter by performing simple number and string manipulations and printing out the results
2. Test the procedures for querying objects and asking classes for their comments.
3. Create a new `Circle` class with the same functionality as that described in Lecture 1.
4. Create a GasBill class which has instance variables `currentMeterReading`, `previousMeterReading`, `rate` and `standingCharge`.
5. Load the file "prthier.st" and examine its output. Read the source code for this file and go on to examine the code in "Object.st", "String.st" and "Array.st". A more sophisticated Smalltalk would have an object browser to facilitate this.

Lecture 7

Smalltalk and Dynamic Binding

Before we look at the issues raised by dynamic binding we will first introduce two programming constructs which programming languages "must" have.

7.1 Conditional Statements and Loops

There is no "if" statement in Smalltalk!!!
There is no "for" loop in Smalltalk!!!

However we clearly require to have statements executed conditionally; the method Smalltalk uses is to create messages for objects which return `true` or `false` and then test for these. e.g. in our Account class, we may wish to check that the balance does not become negative. Thus, we might have

```
!Account methodsFor: 'transferring money' !
spend: amount

(balance < amount) ifFalse: [balance := balance - amount]
!!
```

Here we are relying on the fact that the method "<" is defined for instances of type `Number` and returns either `true` or `false`. This returned `Boolean` responds to the message `ifFalse:` so that the following block is only executed if the instance of `Boolean` is in fact `false`. There is, of course, a corresponding `ifTrue:` message.

Loops are created by sending the message `to:do:` to an integer which is the initial value of the loop. e.g.

```
1 to: 10 do: [ :x | x printNl] !
```

will print out

```
1
2
3
4
5
6
7
8
9
10
```

The actual parameter sent into the block is the current value of the counting variable. Similarly we can change the increments by using the method `to:by:do:`

```
1 to: 10 by: 2 do: [ :x | x printNl] !
```

and can count down from the initial value by making the `by:` argument negative.

7.2 Subclassing

Let us now attempt to create more refined subclasses for the `Account` class created in the last lecture. We will create a `Savings` class which models a savings account and a `Cheque` class which models a cheque account.

7.2.1 The Savings class

```
Account subclass: #Savings
instanceVariableNames: 'interest'
classVariableNames: ''
poolDictionaries: ''
category: nil !
```

The instance variable `interest` will allow us to accumulate interest. We will require to write methods to create a new instance (this will be a class method), to initialise the `interest` variable, to add our interest to the `balance` variable and to then reset the value of the `interest` variable. These methods are written as

```
!Savings methodsFor: 'initialisation' !
init

interest := 0.
^super init
!

!Savings methodsFor: 'interest' !
interest: amount

interest := interest + amount
!

addInterest

self deposit: interest.
self clearInterest
!

clearInterest

|oldInterest|

oldInterest := interest.
interest := 0.
^oldInterest
!!
```

In the `init` message we are initialising the new variable `interest` first and then calling the initialisation procedures in the new class' superclass, `Account` to perform any other initialisiation which is necessary. Notice that we have not defined a new `new` method; when the `Savings` class receives the message `new` it will use that method defined in the `Account` class. When a new instance is created it will be sent the message `init` which, because the new instance is an instance of `Savings`, will be the method defined in `Savings`. This is quite different from C++ in that, in that language, if a superclass' constructor calls an `init()` function, it is the superclass' `init()` function which is called.

Now look at the `addInterest` method: it sends two messages to the current object (the receiver of the message `addInterest`), one of which is defined in `Account` the other defined in `Savings`. We could have simply used

```
balance := balance + interest
```

instead of calling the `deposit:` method. But in doing so we would have created a second way of changing the balance. As things stand, there is only one method which can add something to `balance` and that is

through the `deposit:` method. The advantage of this is that if we decide later to change the implementation e.g. by counting the number of deposits made we need only make the change in one place. Each calling procedure automatically gets the new updated method.

Finally the message `clearInterest` sets the value of `interest` to 0 and returns the previous value. This again is a typical OO procedure - when changing the value of a variable return the previous value to the caller to do with as he wishes. We simply ignored it when we used the message inside `addInterest`.

7.2.2 The Cheque class

```
Account subclass: #Cheque
instanceVariableNames: 'chequeNumber chequesLeft'
classVariableNames: ''
poolDictionaries: ''
category: nil !
```

Again we do not need a `new` method since we can use `Account`'s method. We only need to create a new `init` method.

```
!Cheque methodsFor: 'Initialisation' !
init

chequesLeft := 0.
^super init
!!
```

We have elected to only initialise the variable `chequesLeft` since if there are no cheques left, the `chequeNumber` need not be defined. Now we need methods for writing cheques:

```
!Cheque methodsFor: 'spending' !
newCheques: aNumber count: chequeCount

chequeNumber := aNumber.
chequesLeft := chequeCount
!

writeCheque: amount

| num |

num := chequeNumber.
chequeNumber := chequeNumber +1.
chequesLeft := chequesLeft -1.
self spend: amount.
^num
!!
```

`newCheques:count:` starts us off and `writeCheque:` performs the necessary bookkeeping before using the `spend:` method defined in `Account`.

7.2.3 Refinements to the Class

We stated earlier that Smalltalk was designed to make fast prototyping possible. Let us now state that the previous version of the `Cheque` class is just such a prototype; we will now improve on that prototype by incrementally adding features.

Firstly we would like some checks to, for example, make sure that we do not write cheques when we have none left. This suggests the following method:

```
!Cheque methodsFor: 'spending' !
writeCheque: amount
```

```
| num |

(chequesLeft < 1) ifTrue:[ ^self error: 'Out of Cheques'].

num := chequeNumber.
chequeNumber := chequeNumber +1.
chequesLeft := chequesLeft -1.
self spend: amount.
^num
!!
```

Notice the inherent advantage of the modular OO approach is that we can make changes to the body of a message but, provided the interface does not change, we need take no other considerations into account.

Now we might consider the advantages of keeping a note of all cheques written. An instance of Dictionary might be the easiest implementation of this function and so we create a new instance variable named `history` which will be a Dictionary.

```
Account subclass: #Cheque
instanceVariableNames: 'chequeNumber chequesLeft history'
classVariableNames: ''
poolDictionaries: ''
category: nil !
```

This requires a small change to our initialisation method:

```
!Cheque methodsFor: 'Initialisation' !
init

chequesLeft := 0.
history := Dictionary new.
^super init
!!
```

Notice again how inherently modular the process is: we changed only the internal code of the `init` method and are relying on the `new` method to work with the class `Dictionary`. Now we can use the new Dictionary to keep a note of what cheques have been spent:

```
!Cheque methodsFor: 'spending' !
writeCheque: amount

| num |

" Keep previous check to ensure we have cheques left"
(chequesLeft < 1) ifTrue:[ ^self error: 'Out of Cheques'].

"make sure we have not already used this cheque"
num := chequeNumber.
(history includesKey:num) ifTrue:[ ^self error: 'Duplicate cheque number'].

"Take a note of this cheque's number"
history at: num put: amount.

"Update as before"
chequeNumber := chequeNumber +1.
chequesLeft := chequesLeft -1.
self spend: amount.
^num
!!
```

Since we now have a slightly more complex implementation, we have added comments. Notice how we have used the existing functionality of objects wherever possible. Now we have incrementally improved the method `writeCheque:` twice. Each time we have needed only pay attention to the code held within the method; each time we have used the existing functionality of objects wherever possible. A final demonstration of this will be found in the new method we will write for printing out the history of cheques written.

```
!Cheque methodsFor: 'printing' !
printOn: stream

super printOn: stream.
' , cheques left: ' printOn: stream.
chequesLeft printOn: stream.
' , cheques written: ' printOn: stream.
(history size) printOn: stream

!!
```

Notice how little code we had to write to get this output *and* how much reliance we put on currently existing code for printing in `String`, `Integer` and superclasses of the `Cheque` class.

7.3 Run time errors

The value of the typeless Smalltalk system with respect to polymorphism is clear. However there is an inherent possibility of run-time errors with the system. In practice this is often the case and so there is a need for methods which establish an object's class and which methods it responds to.

Let us now create a simple run-time error:

```
Smalltalk at: #x put: (Dictionary new) !
x at: 1 !
```

Now, since we have put no value with key "1" into the Dictionary, we will get an error:

```
Dictionary new: 32 "<0x4118d8>" error: key not found

MethodContext>>#value
Dictionary>>#at:ifAbsent:
Dictionary>>#at:
UndefinedObject>>#executeStatements
UndefinedObject>>nil
```

The first statement should be clear: the new Dictionary, with default size 32, is stating that it cannot find the relevant key. The subsequent lines show as a set of stack calls how the system attempted to answer the call. We read upwards in chronological order till we get to the most recent call. Thus the undefined object, x, was asked to execute statements. Going up a line we see that it was a Dictionary object which was asked to respond to `at:`. This method called the method `at:ifAbsent:`. So we might guess that the code in the Dictionary class looks something like

```
!Dictionary methodsFor: 'finding values'!
at: key
^self at: key ifAbsent: [^self error: 'key not found']
!!
```

now since the key is not found in the current Dictionary, we require a method such as

```
!Dictionary methodsFor: 'finding values'!
at: key ifAbsent: errCodeBlock

... methods for looking for key ...

(keyNotFound) ifTrue: [^(errCodeBlock value)]
....
!!
```

Now we see that the `at:ifAbsent` method uses an error block and this in fact is the `MethodContext` in the last (most recent) stack call and it is this which receives the `value` method.

7.4 Polymorphism at work

Let us consider adding a new type of number to the system: a class Complex which models complex numbers of the form $a + ib$ where $i = \sqrt{-1}$. One might begin by worrying if the values of a and b are integers or floats but, with "wing-and-a-prayer" programming we will just leave that issue aside for now. We will see that the issue will resolve itself and magically disappear since we will simply use messages which are defined both for integers and floats. Let us begin by subclassing `Number`.

```
Number subclass: #Complex
instanceVariableNames: 'realpart imagpart'
classVariableNames: ''
poolDictionaries: ''
category: nil !

!Complex class methodsFor: 'creating' !
new
^self error: 'use real:imaginary'
!

new: aValue
^self new
!

real: r imaginary: i
^(super new) setReal: r setImag: i
!!

!Complex methodsFor: 'creating -- private' !
setReal : r setImag: i
realpart := r.
imagpart := i.
^self
!!

!Complex methodsFor: ' value returning' !
real
^realpart
!

imaginary
^imagpart
!!

!Complex methodsFor: 'mathematics' !
+ val

^Complex real: (realpart + (val real)) imaginary: (imagpart +(val imaginary))
!

- val

^Complex real: (realpart - (val real)) imaginary: (imagpart - (val imaginary))
!
```

```
* val

^Complex real:( (realpart *(val real)) - (imagpart *(val imaginary)))
imaginary:( (realpart * (val imaginary)) + (imagpart *(val real)))
!

/ val
| d r i |
d:= ((val real)*(val real)) +((val imaginary)*(val imaginary)).
r:= ((realpart * (val real)) +(imagpart *(val imaginary)))/d.
i:= ((imagpart *(val real)) - (realpart * (val imaginary)))/ d.
^Complex real:r imaginary:i
!!
```

This should be reasonably straightforward:

- The first few methods are for generating new instances and are class methods. We wish to insist that a new instance uses the `real:imaginary:` message which creates a new instance of `Complex` using the `new` method defined in `Number` and then sets the `setReal:setImag:` message which is an instance method since by this time a new instance has been created.
- We then have two "get" methods which return the individual real and imaginary parts of the numbers.
- The arithmetic parts of the class each create a new instance of the class `Complex` using the arithmetic operations which we presume are defined elsewhere. Here we see how the potential difficulty with integer and floating point real and imaginary parts disappears. We simply ask each object to respond to the appropriate messages.

There is one difficulty with the class as it stands and that is that we expect all subclasses of `Number` to be comparable using $>, <$ etc. The semantics of equality are straightforward but the others are not so easily implemented for complex numbers and so, in this case, the inheritance hierarchy is working against us. We now simply instruct `Complex` not to let instances respond to such messages:

```
!Complex methodsFor: 'Comparison' !
= val
^((realpart = (val real)) & (imagpart = (val imaginary)))
!

> val
^self shouldNotImplement
!

>= val
^self shouldNotImplement
!

< val
^self shouldNotImplement
!

<= val
^self shouldNotImplement
!!
```

7.5 Conclusion

Smalltalk was one of the first OO languages. It is often said to be more OO than say C++ in that

1. it inherently embodies data-hiding: no object's instance data can be accessed at all unless the object creator creates methods to do so.

2. it comes with a standard set of libraries. Every Smalltalk programmer in the world is using the same basic set of standard libraries.
3. all methods are over-ridable without any special syntax being necessary.
4. all binding of selectors to particular implementations is done dynamically: it is the combination of receiver and selector which decides which particular method is invoked
5. the source code is smaller and more readable

7.6 Laboratory Exercises

1. Create a class hierarchy comprising `Quadrilateral, Rectangle, Square` objects of which will have 4 sides, area and perimeter. The methods used should include `findArea, area, findPerimeter, perimeter, printOn:`

Lecture 8

Eiffel and Objective C

8.1 Introduction

Eiffel has been designed, almost single-handedly, by the software engineer, Bertrand Meyer. His various texts on the topic provide an in-depth, almost pedantic treatment of the creation and use of reusable software components - an emphasis shared with Brad Cox, who developed Objective C. Interestingly both languages share a simplicity of implementation, libraries of classes and an emphasis on library programming. Eiffel however differs from Objective C in that it allows multiple inheritance, is strongly typed and has in-built garbage collection. Both languages have methods for dealing with persistent objects.

Eiffel environments also comes with an ability to cluster groups of classes into distinct universes. While this is outwith the language definition, it seems to have been adhered to by all developers of Eiffel products. Objective C on the other hand clusters Classes using ICpaks.

8.2 Eiffel - the Language

Eiffel is designed on 3 basic principles:

1. Reusability: software should be usable on different projects
2. Extensibility: it should be easy to modify software as requirements change
3. Reliability: an engineering approach to software components should guarantee their reliability

However, the language is also designed to give

1. Efficiency: in the real world, constraints of time and space matter
2. Openness: cooperation with code written in other languages is essential particularly where legacy systems are already in existence.
3. Portability: platform independence

8.2.1 Aspects of Eiffel

Here we simply skim over aspects of Eiffel to give a feel for the language; it is to some extent a half-way house between Smalltalk and C++

1. Eiffel allows multiple inheritance. Features (member functions or variables) inherited may be
   - `renamed` i.e. the implementation kept but the identifier of the feature changed
   - `redefined` by giving the feature a different implementation
2. There is a base class `Any` from which all other classes inherit.
3. Features can be made visible to any class, any group of classes or no classes
4. Abstract data types can be created by using `deferred` classes

8.3 Programming by Contract

A key idea in Eiffel is that of "Programming by Contract" in which every piece of software is deemed, through its interface, to specify what it does and to *guarantee* that it will perform as its interface declares. However, each piece of software in its turn has the right to expect certain preconditions before it is used. We can illustrate the concept with an elementary example from C++. e.g.

```
int CClass::aFunc(int anInt);
```

guarantees that `aFunc` will return an integer. However, `aFunc()` should only be called with an integer parameter. Eiffel makes this idea a little more specific by enforcing contracts.

8.3.1 Assertions

An assertion is an Eiffel term which ensures the correctness of software. There are 4 main types:

Preconditions A precondition to a routine (member function) is the requirements that clients must satisfy before they call the routine. Preconditions are defined by the keyword `require`.

Postconditions A postcondition is the conditions that the routine guarantees on its return (provided the preconditions were satisfied on entry). Postconditons are defined by the keyword `ensure`.

Invariants An invariant is attached to a class and must be satisfied by every instance of the class *whenever* the instance is externally accessable. i.e. it may be broken during a call to an instance's member function but on return from that call, the instance can be guaranteed to satisfy the invariance condition(s). An invariance can be viewed as a global consistency constraint imposed on all routines of the class.

Check Check clauses can be used anywhere to check the value of variables while the program is executing.

There are also special forms for checking the values of expressions within loops. An example from Meyer will illustrate the concepts:

```
class account creation
make    --like a constructor
feature
balance : INTEGER;
minimum_balance : INTEGER is 1000;

deposit(sum: INTEGER) is
require -- precondition
sum >= 0
do
add(sum)
ensure -- postcondition
balance = old balance + sum
end;  -- deposit


withdraw(sum: INTEGER) is
require
sum >=0;
sum <= balance - minimum_balance
do
add(-sum)
ensure
balance = old balance - sum
end;  -- withdraw

feature {NONE)  -- like private features
add(sum:INTEGER) is
```

```
do
balance := balance + sum
end; -- add

make(initial : INTEGER) is
require
initial >= minimum_balance
do
balance := initial
end; -- make

invariant
balance >= minimum_balance
end -- class account
```

This piece of code describes a new class which has member functions/features `deposit()`, `withdraw()`, `add()`, and `make()`, which is the constructor. A `NONE` feature is private, all others here are public. It is possible to make features visible to only some classes. Every feature must satisfy the `invariant` clause, i.e. each time we return from a feature the `balance` must be greater than the `minimum_balance`. This is an essential part of the class' contract. The assertions spell out the contract which this piece of software is prepared to honour when it works with any other piece of software. The preconditions must be obeyed by any caller of the class' routines; in turn each caller can be assured that the postconditions will be satisfied on return from the routine.

Notice the `old` identifier which specifies the value of the variable to which it is attached before the routine was called.

Assertions have other uses than ensuring that the software semantics is correct.

- Eiffel comes with inbuilt documentation-producing tools which create programmers' manuals specifying all assertions i.e. specifying the interface ("observable behaviour") rather than implementation details.
- Since assertions can be evaluated at run-time, they can be used for debugging which requires that we handle any assertion which is found to be false.
- They serve as the basis of exception handling (see below)

Finally we note that the above code minus the assertions is perfectly good Eiffel code in that it will compile and run. However assertions are at the core of the Eiffel philosophy of software engineering and not using them in Eiffel code is considered extremely bad practice.

8.3.2 Exceptions

However, it may be that in our desire to ensure correctness (in the form of assertions) we have created a system which falls over during each run when a user mistypes a value and an exception is triggered. We wish if possible to create a robust system which has the ability to recover from assertions failing. This is provided by the `rescue` clause which is often used in conjunction with a `retry` expression which has the semantics "try the routine again".

The exception may be handled within the routine or by default passed back to the calling program to handle. Again an example from Meyer will suffice

```
attempt_transmission(message : STRING) is
local
failures : INTEGER
do
if failures < 5 then
transmit(message);
successful := true
else
successful := false
end
```

```
rescue
failures := failures +1;
retry
end;  --attempt_transmission
```

This code tries to transmit a message. The call to the routine `transmit(message)` may however fail and so trigger an exception yet we do not necessarily wish the program to fail; we therefore create a `rescue` clause which gets the `transmit(message)` routine called up to 5 times. It is important to realise that the `rescue` clause never attempts to perform the task of the failing routine - that might break the semantics of the design. It merely creates the conditions that the code which performs the design has a second bite at the cherry. If this does not work the exception will be handed on to the caller of `attempt_transmission()` which may have its own `rescue` clause and may simply decide to ignore the failure.

8.4 Garbage Collection

A major issue with respect to memory management in large software systems is what to do with objects which are no longer used by the system. Firstly we have to decide when an object is useless. The simplest way to decide if an object is useless is to find out if it can be reached from the base of the system. (The base of the system is the equivalent of `main()` in any system - what is called first when the program begins execution). Notice that we must ensure that the object cannot be reached either directly or indirectly by any means.

In addition we note that when we have found a useless object, it is often worth following its references to see if there are other useless objects within the system. We could leave this task to programmers but this involves substantial overheads:

1. each programmer must keep a tally of which objects are currently being used and also
2. there must be liaison between programmers since the fact that an object is not being used by one programmer does not imply that it is not being used by all

This clearly violates any design principle asserting modularity. Methods of garbage collection by programmers usually use a means of counting references to objects and deleting those which do not have any references pointing to them; this has been shown to be extremely inefficient and further, it does not get over the problem of disjoint cycles - sets of objects which are unreachable from the main program or currently executing thread but which all reference each other.

Eiffel comes with a built-in Memory class which performs garbage collection. It can be turned on and off. There is no absolute requirement for Eiffel applications to come with this facility but all are encouraged to do so. The major disadvantage of a garbage collection system is the overheads associated with it. Therefore, garbage collectors should be

efficient - obviously

incremental - it should work in small bursts of activity cleaning up as much as possible in each burst. This also implies that it should not just be called when memory has been filled up but it should be triggered frequently

tunable - we should be able to turn the garbage collector on and off again so that in times of high CPU usage, it does not interfere with the application's execution.

8.5 Objective C

Objective C, like C++, is a superset of the C language. Code is preprocessed by the Objective C preprocessor and compiled by the Objective C compiler to give C code which can be compiled by the host C compiler.

Objective C adds to the language exactly one data type, `id` and exactly one new expression, the **message expression** and requires a special syntax for defining classes. The basic C types are available in Objective C and all other objects of any class whatever are of type `id`. (An `id` is a pointer to an instance of a class.) Message expressions are delimited by square brackets and return a value whose type is determined by the programmer. Each message expression consists of two parts: the receiver and the message (or selector). e.g.

[<receiver> <selector>]. The receiver must be an expression which evaluates to type id. The selector is the message identifier plus any arguments it may take. We typically use the header file / implementation file which we used in C++.

Thus a header file (taken from the "Thirty Minute Tutorial" for Objective C) might looks like

```
#import "Object.h"

@interface Fruit : Object
{
char * colour;
int diameter;
}

+create;
-colour:(char *) aColour;
-(char *) colour;
-diameter: (int) aSize;
-(int) diameter;
-grow;
@end
```

This declares a new class `Fruit` which inherits from the base of the class hierarchy, `Object`. Instances of the class have two instance variables, `colour` and `diameter`. These are private and cannot be accessed directly. There are then 6 methods declared: the first is preceded by a "+" and is a class method (a "factory" method) - it will be sent to the class to create a new instance of the class. The other methods are preceded by a "-" sign which signifies that they are instance methods. The syntax is that e.g. the the method `colour:` takes one argument, a char * for which we will use the identifier `aColour`. There is no return type stated for `colour:` and so it will return an `id`. The next method `colour` takes no arguments and returns a char *. We now recognise these as a pair of "set and get" methods.

Now the implementation file

```
#import "Fruit.h"

@implementation Fruit : Object
{
char * colour;
int diameter;
}

// instance creation using superclass' new method
+create
{
id newInstance;

newInstance = [self new]; // create new instance
[newInstance diameter:1]; // set the diameter
[newInstance colour: "green"];

return newInstance;
}

// set the colour
-colour:(char *) aColour
{
colour = aColour;
return self;
}
```

```
-(char *) colour
{
return colour;
}

// set the diameter
-diameter: (int) aSize
{
diameter = aSize;
return self;
}

// get the diameter
-(int) diameter
{
return diameter;
}

// tell the object to increase its size
-grow
{
diameter = diameter +1 ;
return self;
}

@end
```

We recognise the same syntax for comments as in C++. We can omit the `:Object` and the instance variable declarations in the implementation files but many people prefer to have them here as well as in the interface file.

The factory method `create` uses `Object`'s `new` method to create a new instance and having done so it sends the new instance two instance methods to give it an initial diameter and colour. We now specifically return the new instance from the constructor.

Notice in the methods which do not specifically return any object, `self` is returned. As might be expected this is a pointer to the current object and is useful for concatenating expressions. Most of the rest of the code should be self-explanatory.

Let us now subclass the Fruit class to get an `Apple` class.

```
#import "Fruit.h"

@interface Apple : Fruit
{
char * flavour;
}

+create;
-flavour: (char *)aFlavour;
-(char *) flavour;
-flavour:(char *)aFlavour diameter:(int)aSize colour:(char *)aColour;
-grow;
@end
```

with associated implementation file

```
#import "Apple.h"

@implementation Apple

+create
```

```
{
id newInstance;
newInstance = [self new];
[newInstance flavour:"Delicious" diameter:3 colour:"green"];

return newInstance;
}

-flavour: (char *) aFlavour
{
flavour = aFlavour;
return self;
}

-(char *)flavour
{
return flavour;
}

-flavour:(char *)aFlavour diameter: (int)aSize colour: (char*) aColour
{
[self flavour:aFlavour];
[self diameter:aSize];
[self colour:aColour];

return self;
}

-grow
{
diameter = diameter +2;

return self;
}
@end
```

Several things worth noting here:

- Note that after we create a new instance we call a single initialising method which acts like a container for three other methods
- We do not need to redefine the methods `colour:` or `diameter:` for example since they are inherited in the class `Apple` from the class `Fruit`.
- The `grow` method does not require any new syntax. When the selector `grow` is sent to an object, it is that object's class' instance methods which are first searched for a match and only if no match is found is the superclass' instance methods searched.

Now we show a short program which uses these new classes.

```
#import "Fruit.h"
#import "Apple.h"

main()
{
id aFruit, anApple;

aFruit = [ Fruit create];
```

```
printf("aFruit address %lx \n diameter is %d \n colour is %s",aFruit,
 [aFruit diameter],[aFruit colour]);
[aFruit grow];
printf("aFruit address %lx \n diameter is now %d",aFruit, [aFruit diameter]);

anApple = [ Apple create];
printf("anApple address %lx \n diameter is %d \n colour is %s \n flavour is %s",
anApple, [anApple diameter],[anApple colour],[anApple flavour]);


[[[anApple colour:"red"] flavour:"Mackintosh"] diameter:10];

printf("anApple address %lx \n diameter is %d \n colour is %s \n flavour is %s",
anApple, [anApple diameter],[anApple colour],[anApple flavour]);
}
```

Now we create two instances of our new classes. Notice that in the first `printf()` statement, we are using the methods `[anApple diameter]` etc., just as we would use any other expression. Since we know this method will return an integer we need only ensure that the code which uses this method is expecting an integer to be returned. Similarly with the other `printf()` statements.

Now look at the expression `[[[anApple colour:"red"] flavour:"Mackintosh"] diameter:10];` in which we are using the fact that each single method `colour:` and `flavour:` returns `self` and so we are safe in sending it the next selector. This line could have been written

```
 [anApple colour:"red"];
 [anApple flavour:"Mackintosh"];
[anApple diameter:10];
```

8.6 Dynamic and Static Binding

Note that both `anApple` and `aFruit` were declared as `ids`. Thus we have run-time binding here - the compiler cannot check that the method will be accepted by the object since it does not know to which class the object belongs. Sometimes when the system is mature and you know exactly what type of object will receive a particular message you can specify the class of the object exactly. Thus we could have declared

```
Fruit aFruit;
Apple anApple;
```

This gives us static binding and provides compile-time checking. Thus with Objective C we can have some of the benefits (particularly efficiency) of static binding and some of the benefits (particularly flexibility) of dynamic binding simply by changing the declaration of objects. However as with Smalltalk the dynamic linking does have the danger of run-time errors e.g if we send the message `dothis` to `anApple` we would have a stack backtrace looking like

```
[Object dothis] @3377 (non-method)
[ Fruit dothis] @3357 (non-method)
[2b567=Apple dothis] @3345 (non-method)
```

showing the same type of stack unfolding which we met in Smalltalk; the method `dothis` has been passed all the way through the object hierarchy and found no match anywhere.

As with Smalltalk, the creators of the language have felt it necessary to create methods such as `isKindOf:`, `class`, `superclass`, `isMemberOf:` and `respondsTo:` with the same semantics such methods had in Smalltalk. Another useful method is the `perform:` method which takes a selector (which is also an object) as argument and can be used to dynamically change which message is sent to an object.

Also as with Smalltalk, we can start searches for matches with a selector using the identifier `super` which has exactly the same semantics as in Smalltalk.

Persistence of objects in large systems has been tackled by providing `AsciiFiler`, a class which is used to store objects to file. The method `storeOn:` is sent to the object but the method which is used to retrieve the object from the file, `readFrom:` is sent to the class e.g.

```
[anApple storeOn:"apple.dat"];
newApple = [Apple readFrom:"apple.dat"];
```

Thus the object must know how to store itself to file but the class has the responsibility for knowing how to restore instances of the class from files. It is features such as this which makes the class hierarchy more prominent than in C++ where we tend to forget about class hierarchy and talk about object hierarchy.

Unfortunately `AsciiFiler` can be a little temperamental.

8.7 IC Paks

The Objective C compiler comes with a standard library, ICPak101, Some of which is shown below.

```
Object
. Array
. . IdArray
. . IntArray
. . BytArray
. Cltn
. . OrdCltn
. . Stack
. . Set
. . . Dictionary
. . . Bag
. BalNode
. . SortCltn
. String
. Point
. Rectangle
. Assoc
. AsciiFiler
. ObjGraph
. Unknown
. IPSequence
. . Sequence
```

The similarity with the equivalent classes in Smalltalk should be striking. In particular, note the emphasis on collection classes which is facilitated by the fact that every object is of type `id` which we might guess as a C pointer to struct.

There exist other ICPaks also e.g. ICPak201 is a library of visual classes but these must be bought separately.

The name ICPaks comes from the designer's view that software should be created from reusable components which will form something like integrated circuit boards i.e. so that, when you come to a particular problem, you will reach into the library to choose an object whose class already exists in the library. Thus programming becomes more like library management or hardware assembling - the programmer simply takes a pre-constructed piece of software and inserts it in the appropriate way into his code. In time libraries will become known by their maturity - a mature class is one which has been used regularly in the past without mishap and can be relied upon to perform reliably in the future.

Thus Cox envisages a market place in which you have the choice of buying classes freely and in which the maturity of the class is a factor in its selling price: you are liable to pay more for a class which has been proved to be reliable and robust over many years in a variety of different applications than you would pay for a newly-written class.

This takes us on to issues of Intellectual Property Rights. e.g. if someone writes a class which you use to create a new best selling class, are they entitled to expect a cut from future profits or have they, in selling to you, given you *carte blanche* to use the class in any way you choose? Further does their class come with any guarantee of correctness and robustness and if so what is their responsibility when an application several layers up appears to break this class? These are still live issues.

Another issue is the depth to which libraries should extend. Most libraries at present are only at most 4 or 5 classes deep. The inheritance structure rarely goes beyond this level. Is this an inherent limit, something

	Eiffel	Objective C
Typeing	Strong	None
Assertions	Yes	Not explicit
Compile-time check	Yes	Compiler option
Multiple Inheritance	Yes	no
Groups of Classes	Clusters	ICpaks
Libraries	Yes	Yes
Persistence	Yes	Yes
Garbage Collection	Yes	No

Table 8.1: Comparison of Languages

which is an inevitable consequence of human frailty or can we in time create sophisticated libraries of many levels. Some software designers, among them Cox, have stated that a broad rather than deep library is liable to be of more use.

8.8 Conclusion

Two eminent software engineers have begun with the same aim: to create a language which will exploit the full capabilities of object orientation when used to implement large software systems. However their languages are very different.

Lecture 9

Exceptions, Threads and Persistence

We will finish the course with a brief look at three topics which we expect to assume greater importance in the future. We have met two of these topics when using other languages but will use C++ for this chapter:

1. Exceptions: we have seen that polymorphism is at its most powerful when used with dynamic binding. This increases the possibility of run-time crashes and so means of preventing these are becoming more necessary.

2. Threads: Windows95 is a pre-emptive multi-tasking system; the operating system determines how much resources each of the currently operating processes should be allocated. If the projected increase in multi-processor machines happens, it will give an even more powerful boost to multi-threaded programs.

3. Persistence: Object Orientation is inherently about pieces of software which have state. We would like sometimes to be able to retrieve the state of an object when we return to the object after it has been put to one side for some time. Therefore we must have a way of retrieving the data in an object when it has been saved.

We will briefly cover each of these topics. The source code for the programs discussed here are in `Cis\apps\Fyfe-ci0\oo\C` in the directories `Except, Threads, Persist`.

9.1 Exceptions

We will discuss the Microsoft version of the ANSI C++ Exception Handling mechanisms. (Structured) C exceptions are of type `unsigned int` but the C++ mechanism is more flexible in that it can handle exceptions of any type. You can mix C and C++ exceptions but I wouldn't.

To raise an exception, you **throw** the exception. A designated exception handler then catches the exception.

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

int main()
{
char *buf;
int i;

try
{
buf = new char  [64];
if (buf ==0) throw "Memory allocation failed";
}

catch(char * aString)
```

```
{
cerr << "Exception raised:" << aString << endl;
}

//...
 cout << "finished this section";
 cin >> i;
return 0;
}
```

The `try` clause is a guarded section of code. If an exception is thrown in this clause it will be caught by the subsequent catch clause(s). Here we are trying to allocate memory but if this fails we are throwing an exception of type `const char *`. If the exception is thrown, it will be caught by the `catch` clause and then execution is resumed after that clause; if no exception is thrown, the `catch` clause will not be executed. I have suggested in the code that you change the test to `if (buf !=0)` simply to see the exception being thrown and caught. As it stands if an exception is thrown you get the output

```
Exception raised:  Memory allocation failed
```

Notice that one difference between this exception handling and that introduced in Eiffel is that it is non-resumable - once the flow of execution leaves the `try` block, it never returns to that block.

Note that we can throw any type of C++ data (including instances of our own classes) and we can have any number of `catch` clauses to catch different types of exceptions. A "catch-all" clause is one with an ellipses(...) which will catch any exception. In the following code, we throw a float and have three `catch` clauses. The float will be caught by the last.

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

int main()
{
char *buf;
int i;

try
{
buf = new char  [64];
if (buf !=0) throw 5.8 ; // "Memory allocation failed";
}

catch (int anInt)
{
cerr << "Exception number "<< anInt<< " raised" << endl;
}
catch(char * aString)
{
cerr << "Exception raised:" << aString << endl;
}
catch(...)
{
cerr <<"Exception raised"<<endl;
}

//...
 cout << "finished this section";
 cin >> i;
return 0;
}
```

9.1.1 The Exception Stack Frame

One advantage of exception handling C++ style is the automatic cleaning up which goes on when an exception is called. All local object destructors are automatically called. The context which exists between the `throw` site and the `catch` handler is the "exception stack frame". As the `throw` command is called, a new exception object is created and the stack for the local `try` clause is unwound. The compiler then looks for an appropriate `catch` clause. If it finds one the exception is handled and execution resumes after the `catch` clause; if it does not find a handler of the correct type, the exception is propagated outward to any enclosing `try` clause. This is repeated till it reaches the outermost `try` clause. If no `catch` clause of the correct type is found, the predefined run-time function `terminate()` is called and your program will exit (but see later).

To illustrate the flow of execution, we will first create two donothing classes:

```
#ifndef CLASS1_H
#define CLASS1_H
#include <iostream.h>
#include <stdlib.h>
class Class1
{
public:
Class1();
~Class1();
const char *GiveReason(void);
};
#endif
-------------------------------------------------
#include "class1.h"

Class1::Class1(void)
{
cout <<"Class 1 's constructor" << endl;
return;
}

Class1::~Class1(void)
{
cout <<"Class 1 's destructor" << endl;
return;
}

const char *Class1::GiveReason(void)
{
return "Error given in Class 1" ;
}
-------------------------------------------------

#ifndef CLASS2_H
#define CLASS2_H

#include <iostream.h>
#include <stdlib.h>

class Class2
{
public:
Class2();
~Class2();
```

```
};

#endif
-----------------------------------------------------

#include "class2.h"

Class2::Class2(void)
{
cout <<"Class 2 's constructor" << endl;
return;
}

Class2::~Class2(void)
{
cout <<"Class 2 's destructor" << endl;
return;
}
```

Now we will throw an exception of type `Class1` in a `try` block which includes a call to a function:

```
include "class1.h"
#include "class2.h"
#include <stdlib.h>
#include <iostream.h>

void anyFunc(void);

int main()
{
int i;
try
{
cout <<"Try block" << endl;
anyFunc();
}

catch(Class1 anObj)
{
cout <<"In catch handler in main()" << endl;
cout << anObj.GiveReason();
cout << endl;
}
cout <<" Back to main execution" << endl;
cin >> i;
return i;

}

void anyFunc(void)
{
Class1 obj1;
Class2 obj2;
cout <<"Inside anyFunc()" << endl;
throw obj1;
}
```

When we run this code we get the output:

```
Try block
Class 1's constructor
Class 2's constructor
Inside anyFunc()
Class 2's destructor
Class 1's destructor
In catch handler in main()
Error given in Class1
Class 1's destructor
Class 1's destructor
Back to main execution
```

Notice that before reaching the `catch` clause in the main, the destructors for *both* `Class1` and `Class2` are called. Yet `obj1` is necessary in the main program since it is thrown as an exception and must be handled. Therefore a copy of `obj1` is made available to the `main()` function. Secondly the `catch` clause requires a copy and so a second copy is made for it. Thus two copies are now in existence and so the destructor for the class is called twice as the `catch` clause is exited.

We now have to worry about the effect of a second exception being thrown when a first one is being propagated outwards. In this situation, the `terminate` function is automatically called. However a `catch` clause can itself throw *the same exception* as shown in this example:

```
#include "class1.h"
#include "class2.h"
#include <stdlib.h>
#include <iostream.h>

void anyFunc(void);
void anyFunc2(void);
int main()
{
int i;

try
{
cout <<"Try block" << endl;
anyFunc2();
}

catch(...)
{
cout <<"In catch handler in main()" << endl;
cout <<"I'll catch anything thrown at me" << endl;
}
cout <<" Back to main execution" << endl;

cin >> i;
  return i;
 }

void anyFunc2(void)
{
try
{
cout <<"Try block in anyFunc2()" << endl;
anyFunc();
}

catch(Class1 *anyObj)
```

```
{
cout <<"In catch handler in anyFunc2()" << endl;
cout << anyObj->GiveReason();
cout << endl;
throw;
}
return;
}

void anyFunc(void)
{
Class2 obj2;
Class1 obj1;
cout <<"Inside anyFunc()" << endl;
throw &obj1;
}
```

Note that the `throw` statement takes no arguments: it must be the same object which is thrown. This program gives the output

```
Try block
Try block in anyFunc2()
Class 2's constructor
Class 1's constructor
Inside anyFunc()
Class 1's destructor
Class 2's destructor
In catch handler in anyFunc2()
Error given in Class1
Class 1's destructor
In catch handler in main()
Class 1's destructor
Back to main execution
```

Program "except5.cpp" illustrates the fact that you can throw references to objects rather than the objects themselves.

9.1.2 The Terminate Function

We have said that unhandled exceptions (or equivalently multiple simultaneous exceptions) will call the predefined `terminate` function. The default action for this function is to abort your program. It is however possible to redirect `terminate` by using the `set_terminate` function with the name of the function to be called as its single argument:

```
#include "class1.h"
#include "class2.h"
// note the extra includes
#include <eh.h>
#include <stdlib.h>
#include <iostream.h>
#include <process.h>

void anyFunc(void);
void anyFunc2(void);
void term_func(void);

int main()
{
int i;
```

```
set_terminate(term_func);
try
{
cout <<"Try block" << endl;
anyFunc2();
}
catch(int anInt)
{
cout <<"In catch handler in main()" << endl;
cout <<"I'll only catch integer exceptions" <<anInt<< endl;
}
cout <<" Back to main execution" << endl;
cin >> i;
  return i;
 }
void term_func(void)
{
int i;
cout << " This will be executed";
cout << " before abnormal termination of your program";
cin >> i;
exit(0);
}
void anyFunc2(void)
{
try
{
cout <<"Try block in anyFunc2()" << endl;
anyFunc();
}
catch(Class1 *anyObj)
{
cout <<"In catch handler in anyFunc2()" << endl;
cout << anyObj->GiveReason();
cout << endl;
throw;
}
return;
}
void anyFunc(void)
{
Class2 obj2;
Class1 obj1;
cout <<"Inside anyFunc()" << endl;
throw &obj1;
 }
```

Notice that `anyFunc2()` also throws the reference to an object of type `Class1` but there is no function in `main()` which can handle the exception. Therefore the `terminate` function is called which outputs an error message before exiting from the application.

9.2 Threads

A thread is a path of execution through a program: it is also the smallest executable piece of code comprising its own stack, the state of the CPU registers and an entry in the execution list of the system scheduler. Each thread shares all of the process' resources such as open files, semaphores and dynamically allocated memory and so care must be taken when using such resources. The scheduler gives a thread execution control and will

determine how long each thread should run. On multiprocessor machines, the scheduler can move processes to different processors to balance CPU load.

Each thread in a process operates independently and will be unaware of each other unless you tell each about the other. We will use (an adaption of) Microsoft's bounce program to illustrate multi-threaded programming:

```
#define _MT   // instead of compiler option /MT
// will compile as console application but
// must add under Project Settings... Link
// the library LIBCMT.LIB

#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>

#define MAX_THREADS 32
#define getrandom(min,max) ((rand()%(int)(((max)+1)-(min))) + (min))

void kbdFunc(void);
void bounceProc(void *myId);
void clearScreen(void);
void shutDown(void);
void writeTitle(int threadNum);

HANDLE hConsoleOut;
HANDLE hRunMutex;
HANDLE hScreenMutex;
int threadNr;
CONSOLE_SCREEN_BUFFER_INFO  csbiInfo;
int main()
{
hConsoleOut = GetStdHandle(STD_OUTPUT_HANDLE);
GetConsoleScreenBufferInfo(hConsoleOut, &csbiInfo);
// now our functions
clearScreen();
writeTitle(0);
// mutual exclusion flag is cleared
hScreenMutex = CreateMutex(NULL, FALSE, NULL);
// mutual exclusion flag set
hRunMutex = CreateMutex(NULL, TRUE, NULL);
threadNr = 0;

// set up kbdFunc() loop
kbdFunc();

// tidy up
CloseHandle(hScreenMutex);
CloseHandle(hRunMutex);
CloseHandle(hConsoleOut);

return 0;
}
void shutDown(void)
{
```

```
 while (threadNr >0)
 {
  ReleaseMutex(hRunMutex);
threadNr-- ;
  }

  WaitForSingleObject(hScreenMutex, INFINITE);
  clearScreen();
  return;
}

void kbdFunc(void)
{
int keyInfo;

do
{
keyInfo = _getch();
if (tolower (keyInfo) ==’a’ && threadNr < MAX_THREADS)
 {
  threadNr++;
// NOW create new thread
_beginthread(&bounceProc, 0, &threadNr);
writeTitle(threadNr);
}
} while (tolower(keyInfo) != ’q’);
shutDown();
return;
}

void bounceProc(void *myishID)
{
char myCell, oldCell;
WORD myAttrib, oldAttrib;
char blankCell = 0x20;
COORD coords, delta;
COORD  old = {0,0};
DWORD dummy;
char * myID = (char *)myishID; // little cast seems to be necessary

srand((unsigned)  *myID * 3);
coords.X= getrandom(0,csbiInfo.dwSize.X -1);
coords.Y= getrandom(0,csbiInfo.dwSize.Y -1);
delta.X= getrandom(-3,3);
delta.Y = getrandom(-3,3);

if (*myID > 16) myCell = 0x01;
else myCell = 0x02;
myAttrib = *myID & 0x0F;

do
{
// wait for display and then lock it
WaitForSingleObject(hScreenMutex, INFINITE);
// blank out old positions
ReadConsoleOutputCharacter(hConsoleOut, &oldCell, 1, old, &dummy);
ReadConsoleOutputAttribute(hConsoleOut, &oldAttrib, 1, old, &dummy);
```

```
if ((oldCell == myCell) && (oldAttrib == myAttrib))
WriteConsoleOutputCharacter(hConsoleOut, &blankCell, 1, old , &dummy);

// draw new face at current positions
WriteConsoleOutputCharacter(hConsoleOut, &myCell, 1, coords, &dummy);
WriteConsoleOutputAttribute(hConsoleOut, &myAttrib, 1, coords, &dummy);

// release lock
ReleaseMutex(hScreenMutex);

// change coords for next time round
old.X = coords.X;
old.Y = coords.Y;
coords.X += delta.X;
coords.Y += delta.Y;

// make sure we are not leaving the screen
if (coords.X < 0 || coords.X > csbiInfo.dwSize.X)
{
delta.X = - delta.X;
Beep(400,50);
}
if (coords.Y < 0 || coords.Y > csbiInfo.dwSize.Y)
{
delta.Y = - delta.Y;
Beep(600,50);
}
// keep trying while runMutex is still taken
} while (WaitForSingleObject(hRunMutex, 75L)==WAIT_TIMEOUT);
return;
}

void writeTitle(int threadNum)
{
char nThreadMsg[80];
sprintf(nThreadMsg, "Threads running: %02d. Press 'A' to start a thread \
, 'Q' to quit", threadNum);
SetConsoleTitle(nThreadMsg);
return;
}

void clearScreen(void)
{
DWORD dummy;
COORD home = {0,0};
FillConsoleOutputCharacter(hConsoleOut, ' ' , csbiInfo.dwSize.X *csbiInfo.dwSize.Y, home, &dummy);
return;
}
```

The points to note are

1. We need to use the multithread support libraries: this was done by defining the compiler variable "_MT" and adding the library libcmt.lib to the textedit box in Project, Settings, Link.

2. Since we need to synchronise resource use such as file-handling, heap memory usage etc. we create a Mutual Exclusion zone (mutex) to allow threads to execute asynchronously. Thus a shared resource is locked and unlocked. In this example, the `WaitForSingleObject(hScreenMutex,INFINITE)` instruc-

tion gains control of the display and locks it. After drawing a new face, the `ReleaseMutex(hScreenMutex)` is called to free the display for other processes.

3. Therefore the mutex must be created first (in the main() program) and cleaned up in the end.

4. The function `kbdFunc()` creates new threads with `_beginthread()`. Each new thead is begun with a call to the `bounceProc()` function and given a `threadNr`.

5. Note that it is impossible for a thread's x-coordinate to be updated without its y-coordinate also being updated. Each time a thread is ready to write to the screen it calls the `WaitForSingleObject(hScreenMutex,INFINI` instruction. If hScreenMutex is clear, the wait function sets it so that other threads cannot interfere with the display.

6. The second argument in `_beginthread` specifies the stack size for the thread. The zero gives a default value.

7. Your threads should have as little static data as possible so reducing the need for mutexes.

9.3 Persistent Objects

One of the most interesting issues still being researched with respect to Object Oriented languages is the issue of making objects persistent. Since objects have inherent state and we will often wish to use that state during different sessions it is important to have some means of recording the member variables of an object between sessions. We will demonstrate the method which Visual C++ uses to perform that operation however we would cation that the examples we show are very simple and not to be taken as suggesting that all the difficulties with respect to persistance have been solved: we do not have here a system in which many interacting objects containing references to each other is saved between sessions.

We will first look at an example of using the persistence mechanism for C++ and then look at how to make a class persistent. Visual C++ uses the term "serialization" for the process of reading or writing an object to or from a persistent storage medium, typically a disk file. The core idea is that each object will be able to write its current state to file. The object can later be recreated by reading its state from storage. Serialization handles all the details of object pointers and circular references for you. The serialization is done via an object of type CArchive which is an intermediary between the object to be serialized and the storage medium. This frees the object to be stored from having to know any details about the storage medium. It is important to note that the CArchive class is for binary-format serialised objects and is completely different from normal file handling classes.

9.3.1 Serializing an Object

Consider the small `dialog`-based application which was created just as we did in Lecture 6. It aims to take in from the user three values, two of which will be ordinary member variables (known as m_age and m_name) and a member variable which will be the name of the file into which the persistent information is stored. The header file is

```
// persidlg.h : header file
//
/////////////////////////////////////////////////////////////////////////////
// CPersis2Dlg dialog

class CPersis2Dlg : public CDialog
{
// Construction
public:
CPersis2Dlg(CWnd* pParent = NULL); // standard constructor
// Dialog Data
//{{AFX_DATA(CPersis2Dlg)
enum { IDD = IDD_PERSIS2_DIALOG };
CString m_name;
int m_age;
```

```
CString m_filename;
//}}AFX_DATA

// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CPersis2Dlg)
protected:
virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
//}}AFX_VIRTUAL
// Implementation
protected:
HICON m_hIcon;
// Generated message map functions
//{{AFX_MSG(CPersis2Dlg)
virtual BOOL OnInitDialog();
afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
afx_msg void OnPaint();
afx_msg HCURSOR OnQueryDragIcon();
afx_msg void OnSave();
afx_msg void OnRetrieve();
afx_msg void OnInitialise();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

As usual it looks complicated but was all created automatically (see Lecture 6). Notice that the three member variables we mentioned have been declared within an AFX_DATA clause (so that we can use predefined functions to exchange their contents with the contents of the edit boxes to which they are attached). I have removed the standard parts of the implementation file. The sections on which you should concentrate are

```
// persidlg.cpp : implementation file
//
#include "stdafx.h"
#include "persis2.h"
#include "persidlg.h"
#ifdef _DEBUG
#undef THIS_FILE
static char BASED_CODE THIS_FILE[] = __FILE__;
#endif

CPersis2Dlg::CPersis2Dlg(CWnd* pParent /*=NULL*/)
: CDialog(CPersis2Dlg::IDD, pParent)
{
//{{AFX_DATA_INIT(CPersis2Dlg)
m_name = _T("");
m_age = 0;
m_filename = _T("");
//}}AFX_DATA_INIT
// Note that LoadIcon does not require a subsequent DestroyIcon in Win32
m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}

void CPersis2Dlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CPersis2Dlg)
DDX_Text(pDX, IDC_EDIT1, m_name);
DDX_Text(pDX, IDC_EDIT2, m_age);
DDX_Text(pDX, IDC_EDIT3, m_filename);
```

```
//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CPersis2Dlg, CDialog)
//{{AFX_MSG_MAP(CPersis2Dlg)
ON_WM_SYSCOMMAND()
ON_WM_PAINT()
ON_WM_QUERYDRAGICON()
ON_BN_CLICKED(IDC_BUTTON1, OnSave)
ON_BN_CLICKED(IDC_BUTTON2, OnRetrieve)
ON_BN_CLICKED(IDC_BUTTON3, OnInitialise)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

void CPersis2Dlg::OnSave()
{
if (!UpdateData(TRUE))
{
TRACE0("Update data failed during save operation\n");
return;
}
CFile theFile;
theFile.Open(m_filename,CFile::modeCreate |CFile::modeWrite);

CArchive archive(&theFile, CArchive::store);
archive << m_name;

archive.Close();
theFile.Close();
return;
}

void CPersis2Dlg::OnRetrieve()
{
if (!UpdateData(TRUE))
{
TRACE0("Update data failed during retrieve operation\n");
return;
}
CFile theFile;
theFile.Open(m_filename,CFile::modeRead);

CArchive archive(&theFile, CArchive::load);
archive >> m_name;

if  (!UpdateData(FALSE))
{
TRACE0("Update INTERFACE failed during retrieve operation\n");
return;
}

archive.Close();
theFile.Close();
return;
}

void CPersis2Dlg::OnInitialise()
```

```
{
m_name = " ";
m_filename = " ";
m_age =0;
if (!UpdateData(FALSE))
{
TRACE0("Update data failed during INITIALISATION\n");
return;
}
return;

}
```

We have an AFX_DATA_INIT clause in the constructor which shows that I have accepted the default values for these new variables. The AFX_DATA_MAP allows us to link the values of the relevant member variables with the contents of the appropriate edit boxes and the AFX_MSG_MAP links the member functions with the appropriate button clicks. The `onSave(), onRetrieve, onInitialise()` functions were the only ones in which I had to enter code. As previously, we use the predefined function `UpdataData()` with the parameter set to true or false to perform `DoDataExchange()` function in the correct direction.

In `onSave()`, I create a new CFile, open the file determined by the member variable m_filename (creating the file and opening it in write mode) and then create a new CArchive using a reference to the CFile. Notice that the CArchive is opened in store mode and so we often see in system-prepared C++ code

```
if (archive.IsStoring())
{
// do the storing
}
else
{
// do the retrieving
}
```

We must then of course close both the CFile and the CArchive. The `onRetrieve()` and `onInitialise()` member functions should now be self-explanatrory.

Finally we should note that we could and should have placed the write and,*particularly* the read instructions inside a `try` clause.

9.3.2 Making a Serializable Class

You may have wondered why we did not just save the m_age variable at the same time as the m_name variable. The reason is simple: data of type int do not know how to save themselves using a CArchive object!. The only data types that have been predefined to know how to save themselves are CObject*,WORD,DWORD,double,SIZE,CSize,CSt and CTimeSpan.

However we can easily create a new serializable class by inheriting from CObject e.g. suppose we wish to serialise some of the classes we previously used such as Mystrng and Client.

```
#ifndef MYSTRING_H
#define MYSTRING_H
#include <iostream.h>
#include <stdlib.h>
#include <afx.h>
#define DEFAULTLENGTH 30

class Mystring : public CObject
{
char *theString;
int length;
void init(char *aString, int aLength);
 public:
```

```
  DECLARE_SERIAL(Mystring,)
  Mystring(int anInt);
Mystring(void);
Mystring(char *aString);
Mystring(Mystring &aStringObject);
Mystring(char *aString, int aLength);
~Mystring(void);
void setString(char *aString);
char *getString(void);
int getLength(void);
void print(ostream &aStream);
friend ostream &operator<<(ostream &aStream, Mystring &aString);
void Serialize(CArchive &archive);
friend CArchive &operator<<(CArchive &archive, Mystring &aString);
friend CArchive &operator>>(CArchive &archive, Mystring &aString);
};
#endif
```

Notice that we have added one new member function `Serialize()` (using the same identifier as Visual C++ applications since we are overloading a function defined in CObject), two new friend functions (which look very similar to the type of overloaded operators which we have used for I/O) and a single statement- the `DECLARE_SERIAL(Mystring,)` macro. The class now also inherits from CObject. It should be pointed out at this stage that we require a constructor which takes no arguments which is used when objects are being recreated from disk. It may sometimes be better to make this constructor private (`serialize()` still has access to it).

We now have the implementation file (and will only print the new sections)

```
IMPLEMENT_SERIAL(Mystring, CObject, 1)

void Mystring::Serialize(CArchive &archive)
{
CObject::Serialize(archive);
 CString aString(theString);

if (archive.IsStoring())
archive << aString ;
else archive >> aString ;
return;
}

CArchive &operator<<(CArchive  &archive, Mystring &aString)
{
aString.Serialize(archive);
return archive;
}

CArchive &operator>>(CArchive  &archive, Mystring &aString)
{
aString.Serialize(archive);
return archive;
}
```

The IMPLEMENT_SERIAL macro must appear in the implementation file; the first two arguments are the name of the class and the name of its immediate superclass; the third argument is a "schema number" which is essentially a version number for the class(must be an integer greater than or equal to 0).

The `serialize()`method first calls CObject's `serialize()` method and then creates a new instance of type CString (which knows how to serialise itself) and uses its functionality to store the contents of theString.

I have overloaded the "<<" and ">>" operators for completeness (though Microsoft does not recommend this (see below)). We can then use the serialization for any new class which uses a Mystring such as our Client class.

```
ifndef CLIENT_H
#define CLIENT_H
#include <iostream.h>
#include <stdlib.h>
#include <afx.h>

#include "mystring.h"

class Client  :public CObject
{
Mystring name;
int age;
public:
DECLARE_SERIAL(Client,)
Client(void);
Client(Mystring &aString, int anAge =0);
Client(char *aString, int anAge =0);
~Client(void);
friend ostream & operator<<(ostream &aStream, Client &aClient);
Client &operator=(Client &aClient);
void setAge(int anAge);
int getAge(void);
void setName(Mystring &aName);
Mystring &getName(void);
void Serialize(CArchive &archive);
};

#endif

void Client::Serialize(CArchive &archive)
 {
  CObject::Serialize(archive);
name.Serialize(archive);

return;
}
```

However there is one additional consideration which should be taken into account when subclassing from CObject. When we use a CArchive>> operator to construct a CObject (or its descendents) from disk, it uses information previously written to the file by the storing archive using CRuntimeClass information. Therefore if you need to the loading archive to dynamically recreate an object use << to store the object and >> to retrieve the object. If you know the exact class of the object in advance use the `Serialize()` function. Therefore, in the above we use `name.Serialize()`.

However, had we declared the member variable name as a `Mystring *` and dynamically allocated a pointer to an instance of Mystring *or its descendents* to name, we should have used the CArchive << and >> operators to store the run-time information which details specifically what type of object has been stored. Thus we would have had

```
void Client::Serialize(CArchive &archive)
 {
  CObject::Serialize(archive);
if (archive.IsStroring())
archive << name;
```

```
else archive >> name;

return;
}
```

Such a method allows a polymorphic reconstruction of persistent objects.