8 Objects and Classes

8.1 Object-oriented programming

The programming that we have done so far has been procedural — everything is done by functions that are ultimately called from within the main function. This use of functions hides details of the information and instructions needed to perform specific tasks from the rest of the program. Such compartmentalisation of code and data allows easy upgrades to the program and also makes it easier for many programmers to work on a large project.

Object-oriented programming develops these ideas further and imposes a higher level of structure than procedural languages. In general, a problem is broken down into subproblems organised in a hierarchical structure. Each subproblem can be translated into self-contained units called objects, which can contain variables and functions (also called methods) that operate upon those variables.

In their simplest form, objects can be regarded as “custom” data types. For example, a complex number is an object that has a real and an imaginary part. If we want to operate on the new complex number object then we must write new functions to do so.

Object-oriented programming introduces three main ideas:

Encapsulation binds code and data together into an object that cannot be influenced from outside sources except in very tightly controlled ways.

Polymorphism represents the concept of “one interface, multiple methods”. The same interface can be used to do different things for different objects: i.e. define + to add real numbers, and “custom made” complex numbers, matrices, etc.

Inheritance allows one object to acquire the properties of another. The new object has all the properties of the old and may add more of its own. An example would be to define a generic Integrate object and the derived objects Trapezium, Simpsons, GaussianQuadrature, etc, could all inherit from Integrate.

8.2 The C++ class

The C++ class keyword is used to define objects. A very simple example would be to create a bank account class that stores the name and balance of a customer.
The syntax is straightforward, the `class` keyword followed by a name indicates that we are defining a new class. Inside the braces we define all the variables (member data) that make up our new class. A `string` is a C++ object that represents a collection of letters, i.e. words. The `public` keyword indicates that these data can be accessed (and potentially modified) from outside the class. Finally, we **must** finish the declaration by placing a semicolon after the closing brace `};`.

Once defined, the Account class can be used in the same way as any built-in data type. We define objects of the `Account` class as follows:

```cpp
int main() {
    Account ac_001, ac_002;

    ac_001.Name = "John Smith"; ac_001.Balance = 0.0;
    ac_002.Name = "Jane Jones"; ac_002.Balance = 10.0;
}
```

The two objects (variables) `ac_001`, `ac_002` are both instantiations of the `Account` class. Note that the variables within the class are accessed by using the dot operator, but that we must use the name of the specific object, not the class name, `Account.Name` is an error. If the `public` keyword had not been included in our class definition, the above code would not compile; it fails with the error `'Account::Name' cannot access private member declared in class 'Account'`. 
The most general class definition is:

```cpp
class class_name {
private data and functions
  access-specifier:
  data and functions
  access-specifier:
  data and functions
.
.
.
  access-specifier:
  data and functions
}
```  

The object list is optional, but if present it defines objects of the class. Once the class is declared, objects may also be defined anywhere in the code. The access-specifier is one of the three keywords public, private or protected. The default access type of classes is private, which means that the data and functions can only be used by objects of the same class. If the items are public, they are accessible by other parts of the program, as shown above. Finally, the protected keyword is essentially the same as private, but protected members may be used by any derived classes, §8.4. Note that access specification can be changed many times in a class declaration.

Member functions are functions that are defined within the class and can operate on all the class’s member data. An example would be a print_balance function in our Account class.

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


class Account
{
  public:
    string Name;
```
double Balance;

//Print the balance to the screen (cout)
void print_balance()
{
    cout << Name << "'s account has a balance of "
         << Balance << endl;
}
};

int main()
{
    //Declare and initialise the account
    Account ac_001;
    ac_001.Name = "John Smith"; ac_001.Balance = 100.0;

    ac_001.print_balance();
}

Note that the member function is accessed in the same way as member data by using the dot operator.

Another obvious member function for an account class is a deposit() function.

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

class Account
{
public:
    string Name;
double Balance;

//Print the balance to the screen (cout)
void print_balance()
{
    cout << Name << "'s account has a balance of "
         << Balance << endl;
}

//Deposit amount into the account
//Function prototype, "guts" must be defined later
void deposit(const double &);

//Definition of the deposit function
void Account::deposit(const double &amount)
{
    Balance += amount;
}

int main()
{
    //Declare and initialise the account
    Account ac_001;
    ac_001.Name = "John Smith"; ac_001.Balance = 100.0;
    ac_001.deposit(50.0);
    ac_001.print_balance();
}
We can define the "guts" of member functions outside the class provided that we provide a function prototype within the class. A function prototype must specify the name of function, its return type and its arguments, but instead of the definition within braces, we finish the statement with a semicolon. The function body can be defined anywhere else in the program, but must be defined somewhere or the program will not compile. If defined outside a class declaration, a member function must include the class namespace, ClassName::member_function(), to distinguish it from member functions of other classes with the same name. If the arguments of the function are not the same as those of the prototype the program will not compile. The const keyword in the function argument indicates that the function is not allowed to change the argument amount even though it has been passed by reference.

Having written our deposit function, we can make Balance a private variable so that it can only be modified by member functions of the Account class. The following program should produce the same output as the program above, but it does not. Why?

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

class Account
{
    private:
        double Balance;
    public:
        string Name;

    //Print the balance to the screen (cout)
    void print_balance()
    {
        cout << Name << "'s account has a balance of " << Balance << endl;
    }
```
8 OBJECTS AND CLASSES

//Add amount to the balance (now declared in the class)
void deposit(const double &amount)
{
    Balance += amount;
}

int main()
{
    //Declare and initialise the account
    Account ac_001;
    ac_001.Name = "John Smith"; ac_001.deposit(100.0);
    ac_001.print_balance();
}

The answer is that because we have made Balance private, we cannot access it directly in the main program and it has not been initialised, which means that the result could be anything! The initialisation of private data can only be performed by a special member function — the constructor.

8.3 Constructors and Destructors

Constructor functions are called whenever an object is created and are used to initialise variables within the object. Similarly, destructor functions are called when an object is destroyed and are used to clean up memory or close files that may have been opened by the object.

For any object, the constructor has the same name as the class and the destructor has the same name as the class, prepended by a tilde ~. We now add a constructor and a destructor to our Account class.

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

class Account
{
    private:
        double Balance;
    public:
        string Name;

    //Constructor, initialise the Balance to zero
    Account() {Balance=0.0;}

    //Destructor, print closing Balance
    ~Account()
    {
        cout << "Closing account :";
        print_balance();
    }

    //Print the balance to the screen (cout)
    void print_balance()
    {
        cout << Name << \'s account has a balance of " << Balance << endl;
    }

    //Add amount to the balance
    void Account::deposit(const double &amount)
    {
        Balance += amount;
    }
};
```cpp
int main()
{
    //Declare and initialise the account
    Account ac_001;
    ac_001.Name = "John Smith"; ac_001.deposit(100.0);

    ac_001.print_balance();
}
```

The balance of a new account is initially set to zero and the balance will be printed when the `Account` object is destroyed (goes out of scope). It is possible to pass variables to a constructor and so we could also set an initial balance for a new account. It is **never** possible to pass arguments to a destructor.

```cpp
Account::Account(const double &initial_balance)
{
    Balance = initial_balance;
}
```

```cpp
int main()
{
    //Set the opening balance of the account to 100
    Account ac_001(100.0);

    //Set the opening balance of the account to 50
    Account ac_002 = 50.0;
}
```

The second form of initialisation, normal assignment using `=`, only works if the constructor takes a single argument. Note that if the class constructor takes arguments, it is impossible to create an object without passing those arguments. We can use function overloading, if required, to provide a number of different constructors.
8.4 Inheritance

C++ allows the creation of new, sometimes called derived, classes from existing, or base classes. The idea is that the derived classes should be related to the base classes, perhaps they are more specialised versions of an abstract concept. Let us create a new type of Account called a SavingsAccount that pays interest on the balance.

```cpp
class SavingsAccount : public Account
{
  private:
    double Interest_rate;
  public:

    //Default Constructor, call Account’s default constructor
    SavingsAccount() : Account()
    {
      //Set the interest rate
      Interest_rate = 0.05;
    }

    //Constructor with initial balance, call equivalent constructor
    //of Account
    SavingsAccount(const double &initial_balance) : 
    Account(initial_balance)
    {
      //Set the interest rate
      Interest_rate = 0.05;
    }

    //Add interest to the account
    void add_interest()
    {
```
Balance += Balance*Interest_rate;
}
};

The new SavingsAccount has all the member features of the Account class, but adds new functionality of its own. The following simple program

```cpp
int main()
{
    //Create a new savings account with initial balance of 100
    SavingsAccount sav_ac_1(100.0);
    sav_ac_1.Name = "John Smith";

    //Add interest to the account
    sav_ac_1.add_interest();
}
```

should produce the output

Closing account :John Smith’s account has a balance of 105

but instead it fails to compile and produces the error

'A::Balance' cannot access private member declared
in class 'A'

This is because we declared Balance to be private which means that it can only be used by member functions of the Account class. We could make Balance public, but then any function anywhere in the program could modify it. The solution is to change the Account class so that Balance is protected, meaning that it can be used by Account and any of its derived classes.

```cpp
class Account:
```
The general syntax for creating derived classes is as follows:

```cpp
class class_name : access-specifier base_class_name {};
```

The `access-specifier` can be `public`, `private` or `protected`, just as inside class definitions. The principle of encapsulation cannot be violated, and so the access-specifier only applies to the public and protected members of the base class. Private members of the base class always remain private to that class. Protected members of the base class can be used by derived classes, but cannot be used outside them. Finally, public members of the base class remain public in the derived class, unless overruled by the access-specifier. This can all get quite confusing, but the essence is that data can only be made “more” private by using an access-specifier in front of a class — private data can never be made public.

### 8.4.1 Overloading member functions

In the `SavingsAccount` class, we added extra member functions and data to the base `Account` class. We can also use function overloading to modify the functions defined in the base class. For example, let us create a new `account_type` function that returns a string describing the type of account and modify the `print_balance` function to use of this function.

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

//Base Account class
class Account
{
    protected:
        double Balance;

    public:
        // Function to overload type function
        string account_type() const
        {
            return "Savings Account";
        }

        // Function to print balance
        void print_balance() const
        {
            cout << "Your balance is: " << Balance << endl;
        }

    // Other member functions...
};
```
public:
    string Name;

    // Constructor that sets initial balance
    Account(const double &initial_balance) {Balance = initial_balance;}

    // Return the type of account as a string
    string account_type() {return "Basic Account";}

    // Print the balance to the screen (cout)
    void print_balance()
    {
        cout << Name << "'s " << account_type() << " has a balance of " << Balance << endl;
    }
};

// A savings account inherits from the Account class
class SavingsAccount : public Account
{
private:
    double Interest_rate;

public:
    // Constructor calls Account's constructor
    SavingsAccount(const double &initial_balance) : Account(initial_balance)

{
    // Set the interest rate
    Interest_rate = 0.05;
}

// Overload the account type
string account_type() {return "Savings Account";}

int main()
{
    // Create a savings account
    SavingsAccount sav_01(100.0); sav_01.Name = "Fred Philips";
    // Create a standard account
    Account ac_01(50.0); ac_01.Name = "Fred Philips";

    // Print the account names
    cout << ac_01.account_type() << endl;
    cout << sav_01.account_type() << endl;

    // Print the balance of both accounts
    ac_01.print_balance();
    sav_01.print_balance();
}

The result of the program is the output

Basic Account
Savings Account
Fred Philips’s Basic Account has a balance of 50
Fred Philips’s Basic Account has a balance of 100
What has happened? The function `account_type` has been correctly overloaded when called directly, but not when it is called indirectly from within the member function `print_balance`. The problem is that `print_balance` is a member function of the base `Account` class so that when it is compiled it “does not know” that the function `account_type` will be overloaded in `SavingsAccount`. We can indicate to the compiler that the function might be overloaded by using the `virtual` keyword in the initial definition of the function.

```cpp
virtual string account_type() {return "Basic Account";}  
```

After this simple modification, the program produces the result

Basic Account
Savings Account
Fred Philips’s Basic Account has a balance of 50
Fred Philips’s Savings Account has a balance of 100

If you are writing member functions that are going to be overloaded you should nearly always make them `virtual`. It is also possible to define a “pure virtual” function; that is an interface for a function that must be implemented for every derived class. The syntax for a “pure virtual” function is

```cpp
virtual string account_type()=0;  
```

If the “pure virtual” `account_type()` function is not overloaded in the `SavingsAccount` class, the program will not compile.

### 8.4.2 Multiple inheritance

A class may be derived from more than one base class, in which case the parent classes are separated by commas in the definition:

```cpp
class derived: public base1, public base2 {};
```

One use of multiple inheritance is to “bolt together” functionality from different objects and it is possible to create very complex hierarchies of objects. One of the hardest parts of object-oriented programming is creating a simple, but complete, object model.
8.5 Pointers to objects

Pointers to objects can be declared in exactly the same way as any other data type, `Account* ac_pt` creates a pointer to an account object. Note that member functions and data must be accessed from an object pointer by using the arrow operator, `->`. An important feature of pointers to objects is that an object of any derived class can be assigned to a pointer to a base class. Provided that all overloaded functions have been defined as virtual functions the correct version of the function will be called.

```cpp
int main()
{
    //An array of three pointers to accounts
    Account* ac_pt[3]= {0,0,0};

    //Allocate a standard account to the first pointer
    ac_pt[0] = new Account(100.0); ac_pt[0]->Name = "John";

    //Allocate a savings account to the second pointer
    ac_pt[1] = new SavingsAccount(500.0); ac_pt[1]->Name = "Mary";

    //Loop over all accounts and print the balance
    for(unsigned i=0;i<3;i++)
    {
        //Check that an object has been allocated
        if(ac_pt[i] != 0)
        {
            ac_pt[i]->print_balance();
        }
    }

    //We should delete the objects here to be safe, but because it is
    //the end of the program, they will go out of scope and be cleaned
```
The result of the program is

John’s Basic Account has a balance of 100
Mary’s Savings Account has a balance of 500

The Excel object model relies on the extensive use of pointers to objects. For example, if we have a pointer to a particular Excel application called XL, then we obtain a pointer to the Worksheet “Sheet1” in the active Workbook as follows:

XL->ActiveWorkbook->Worksheets->Item["Sheet1"];  

Note that a pointer to a worksheet is itself an object defined in the Excel namespace.

8.6 The this pointer

Every C++ object contains a pointer to its own location in memory. This pointer is accessed using the this keyword in member functions of the class, e.g.

Account::print balance()
{
    cout << this->Name << "’s account has a balance of "
    << this->Balance << endl;
}

In most member functions, the this pointer is used implicitly by the compiler and it is not necessary to use the this keyword explicitly. An important exception is when working in classes derived from a templated base class. Nonetheless, the this pointer is essential in certain applications, for example, when a member function returns a pointer to the object itself.