Computer Science Department

Introduction to Programming 2

Brief Summary Notes by Steve Benford

Chapter 1 : Programming in the Large

1.1. Goals of the course

PR1 taught you how to write relatively small-scale computer programs to solve a variety of problems. As a result, you should have developed some basic problem solving skills as well as a working knowledge of C++.

So far, you have programmed on your own, producing one-off "stand-alone" programs. This is a far cry from real-world programming which often involves teams of programmers producing massive and complex systems over a long period of time. PR2 changes our focus away from the microscopic nuts-and-bolts level of small-scale programming towards **programming in the large** - i.e. solving large problems as part of a programming team.

As with PR1 the course will cover general programming concepts. It will also introduce new features of C++ to support these concepts. Regular coursework will be a central part of the course. More specific goals include:

- Introducing techniques for dividing large systems into well defined components which can be constructed independently.
- Techniques for writing code which can easily be used by other people in their programs.
- Powerful new techniques for solving a variety of problems.

As you have probably already noticed, the hardest part of programming is often problem-solving, not writing code. This course will attempt to further develop problem solving as well as programming skills. We will also introduce some "software tools" which support the general development of programs.

You should understand that this course does not tell the whole story. First, we do not look at many more advanced programming techniques (e.g. fully object-oriented programming). These are left for the second year course on *Object-Oriented Design*. Third, we do not cover the team-work and project-management aspects of large-scale system building. These are addressed by the **Group Projects** in the second year.

1.2. Overview of PR2

PR2 is divided into the following chapters.

1. Introduction to programming in the large. Discussion of the requirements of building large scale commercial systems in programming teams. Introduction to Data Abstraction as a key technique.
2. C++ Classes as a way of implementing Abstract Data Types. Terminology and structure of classes and objects. Simple examples: the *Counter, Money* and *Set* classes.
3. Interacting classes. Solving more complex problems by building classes which work together. At this level, programs can be seen as a set of interacting, but discrete objects. The main example is a *Cashpoint* simulator which interacts with *Account* and *Money* classes.
4. Better interfaces to classes. Making your classes easier for other programmers to use. Specific issues will include operator and function overloading as well as the use of references. Examples include a *Point* class representing two dimensional coordinates.
5. Pointers revisited. An in depth exploration of pointers, building on the work of PR1. If used carefully, pointers provide us with great flexibility for solving complex problems. This chapter will give thorough coverage of how to declare and manipulate pointers.
6. Building dynamically expandable data types. How to use pointers to gain greater control over memory usage and so become free of the fixed size limitation inherent in data types such as arrays.

Examples include a resizeable array and a *String* class for manipulating text.

7. Linked data types. The construction of complex objects which consist of linked components. The main example will be a *Linked List* data type representing an arbitrarily long list of objects.
8. Case study: the *Line Editor*. This chapter will draw the contents of the course together by looking at the design and implementation of a simple Line Editor which can be used to edit files.

1.3. Commercial programming

Building real systems in a commercial/industrial environment is very different to the kind of programming you have done so far :-

- Programs are many times larger (maybe millions of lines of code).
- As a result of their size, programs are usually written by programming teams, not by individuals. This means that they must be Modular in structure (i.e. easily divided into distinct components).
- Programs should have a long lifetime (not just 1 week) during which they need to be maintained, maybe repaired and/or upgraded.
- New programs should be built on existing code because i) re-using existing code reduces development time and increases profit ii) new products often need to be compatible with old ones.
- Programs need to work properly. This is obviously true for programs which control dangerous machines (e.g. fly-by-wire aircraft) or manipulate huge sums of money. Of course, it really applies to all programs as the company's reputation is critical to good business. The number of bugs in code should be minimised and, when they occur (because they always do), they should be easy to locate and fix.

The discipline of programming in such an environment is called *Software Engineering*. Its main goals are to produce systems which are **modular**, **maintainable**, **extendable**, **work properly** and make maximum **re-use** of existing code.

1.4. Procedural Abstraction

The key to building such systems is ABSTRACTION. We have already seen on the CUA course that the history of programming languages is one of increasing abstraction. Abstraction allows us to form a view of a problem which considers only necessary details and hides away unnecessary nitty-gritty. Abstraction can also be used to give different people different views of the same problem. You have already come across this in PR1.

For example, imagine that I write a C++ function called *power* which raises one number to the power of another. I might show you the function heading with suitable comments describing what the function does. You would then know how to use it in your own program.

```
// raise the number n to the power p and return the answer
// works for floating point n and positive and negative integer p
float power(float n, int p);
```

Note that you don't have to know how it is implemented! In other words, you have an abstract view of the function. I then have to write a specific implementation. Here are two different ones.

```
// implementation using a for loop
float power(float n, int p) {
     int i;
     float res = 1;

     // test for positive or negative power
     if(p >= 0)
          // multiply 1 by n p times
          for(i = 0; i < p; i++)
               res = res * n;
     else
          // divide 1 by ny p times
          for(i = p; i < 0; i++)
```

```
            res = res / n;
    return res;
}


// implementation using a while loop
float power(float n, int p) {
    float res = 1;

    // test whether positive or negative power
    if(p >= 0)
        // multiply 1 by n p times
        while(p--)
            res = res * n;
    else
        // divide 1 by n p times
        while(p++)
            res = res / n;

        return res;
}
```

Note that you are unware of which implementation I use. Even more importantly, I can change the implementation - perhaps make it more efficient - and your code won't have to be changed. We call this *procedural abstraction.*

Put more formally, procedural abstraction is used to define an interface between the user and the implementor of a function resulting in more modular code.

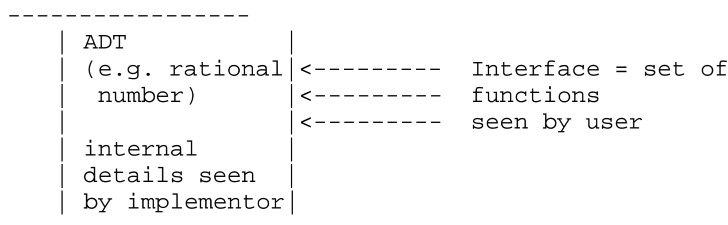
1.5. Data Abstraction

Data Abstraction takes these ideas much further. Instead of defining a single function, a whole new data type is defined.

Each language provides a basic set of types (e.g. float, int, char in C++) and gives the user operations for manipulating variables of these types (e.g. + - * etc). Data Abstraction allows the definition of new types, complete with a set of operations or functions, which appear as if they were part of the language. We call these **Abstract Data Types (ADTs)**.

The idea is that a new ADT is defined so that its users see an interface which provides an abstract view of the type and only its implementors see its internal details. The interface is defined by a set of functions which give access to the type. It is only possible to access the type through its functions.

We can show this by the following picture:



Several points need emphasising:-

- We talk about each ADT having a **user** and an **implementor**. These may represent different members of a programming team. Thus ADTs give us a way of breaking a program into a modular structure.
- The functions exactly specify the interface to the type (i.e. how it can be used).

- The internal details (e.g. any data) cannot be accessed apart from through the functions. In other words, the functions **protect** the ADT from being tampered with, as well as protecting the user from having to know internal details. We call this **encapsulation**.
- Encapsulation reduces the chances of bugs occurring and makes them easier to locate when they do. If a data type gets corrupted you know that it must have been one of its interface functions that caused the problem.
- If we define a general enough set of functions, an ADT can be re-used in many different situations.

Thus, ADTs will form the basis of our approach to programming in the large. Notice, that they represent a general technique, not a specific part of any particular language (although several languages provide good support for building them). In the next chapter we will see how to realise ADTs in C++ using the new mechanism of **classes**.

Chapter 2 : C++ Classes

2.1. Introduction

The last chapter introduced the idea of Abstract Data Types (ADTs) as an approach to building modular systems in programming teams. This chapter deals with the specifics of realising ADTs in C++ through the new concept of **CLASSES**.

The chapter introduces necessary terminology and covers the fundamental stages of defining, implementing and using classes. Some initial guidelines for good class design are also provided.

The chapter follows three examples: a simple *Counter* class, used for counting the occurrences of events, a *Money* class used to manipulate values of pounds and pence, and a more complex *Set* class which allows programmers to work with sets of characters and supports set operations such as union and intersection.

2.2. First example - the Counter Class

The Class is a new concept which can be used to introduce new C++ types for programmers to use. We will introduce classes through a simple example - a generalised Counter which counts the number of occurrences of some event in a program. Effectively, a Counter is an object which starts with an initial value of zero, and whose value can be increased one step at a time. At any point the value can be read by the program, tested whether it is zero or compared against another counter. The value can also be reset to zero at any time.

Our goal is to build a general Counter class that can be used in many different programs. Bear in mind that there are two kinds of people involved with the Counter. Implementors, who design and build it, and its user(s) who use it in their programs. We shall look at each role in turn. Overall, there are three stages to building the class.

1. Definition - the implementors (and maybe users) decide what the class should do and define its interface.
2. Implementation - the implementors implement the class.
3. Use - users use the class in their programs.

Stages 2 and 3 can occur in parallel.

2.2.1. Definition of Counter

Definition of the Counter means exactly specifying its interface (what it does and how to use it) and writing this as a C++ class definition in a **header file**. The name of the header file is usually (but not necessarily) that of the class with the added suffix .h.

In our example, the following C++ code is written in the file *Counter.h*.

```
// definition file for a simple counter class
// Steve Benford November 1991

#ifndef COUNTER_H
#define COUNTER_H

class Counter {
    private:
        int count;          // stores the current count value

    public:
     // constructor
        Counter();          // initialise counter to 0

        // access
        void reset();       // reset counter to 0
        int read();         // return current count
        void inc();         // increment count

        // tests
```

```
        int iszero();    // test whether 0
        int equals(Counter c); // test equivalence
};

#endif
```

The class definition starts with the word class, followed by the name of the class and then the rest of the definition between braces. The definition must be terminated with a semi-colon. The definition is divided into two parts.

The **public** part defines the interface to the class in terms of a set of function declarations. These follow the word *public:*. Each declaration gives the name of a function along with its argument and result types (argument names are optional, but it is good style to include them). Another name for these declarations is *function prototypes*. Being in the public part means that users of the class can invoke these functions.

The interface functions define all the actions that users can take on variables of the class. Thus, the *reset* function simply sets the value to zero (hence no argument or result); the *read* function returns the current value as an integer; the *inc* function adds one to the current value; the *iszero* function tests whether the current value is zero and the *equals* function tests against another counter (hence one argument of type Counter). The last two functions return integer results (1 indicates TRUE, 0 indicates FALSE).

The *Counter* interface function is special. It is called the **constructor** function for the class and is used to initialise variables of the class. In this case it will set the initial count value to zero. Most classes will have a constructor function which must have the same name as the class. The constructor doesn't return a result (not even void) but can take arguments. It is automatically invoked whenever a variable of the class is declared in a program.

Note that good comments are an essential guide to how to use the class.

The **private** section defines the data that is needed to implement the class. In the case of the Counter we need only remember the current count value. This requires a single integer variable. Being in the private section means that users of the class are prohibited from directly accessing the variable "count". It can only be accessed through the interface functions.

Now for some terminology. Anything that appears in the class definition is a **member** of the class. In general we refer to **member functions** and **data members**. A member function is also often called a **method** of the class. Thus, when someone talks about "invoking a method in a class" they mean using a class interface function.

A variable of the class which is declared in a users program is called an **object** (or sometimes an **instance** of the class). Thus, the distinction between classes and objects is the same as for types and variables.

You should note that data members and methods can appear in both the public and private sections of a class. It is quite common to see private methods (which can only be used by other methods). It is very rare (and probably very bad practice) to see public data members.

It is a convention that the private section is written before the public section.

You may wonder about the lines:

```
#ifndef COUNTER_H
#define COUNTER_H

#endif
```

These are needed to stop multiple definitions of the same class occurring when a header file is accidently referred to more than once by a user program. We shall see later that this easily happens when classes use other classes (see chapter 3).

The statements are a kind of if-statement for the compiler. The first line says: "has the name COUNTER_H been defined yet?" If the answer is no, everything up to #endif is compiled. This includes defining the name COUNTER_H as the very next statement. If the answer is yes, the compiler ignores everything up to the #endif. The logic of this is that the code between the statements will only be looked at once by the compiler. If you don't understand this, you should at least be aware that YOU MUST ALWAYS PLACE SIMILAR STATEMENTS AROUND CLASS DEFINITIONS THAT YOU WRITE.

You can chose the name to be defined yourself.

To summarise, the general structure of a class definition is:

> **class** *class-name* **{**
>   **private:**
>   *declaration of data members*
>   *and private methods*
>   `public:`
>   *declaration of constructor methods*
>   *declaration of interface methods*
> **};**

Watchout for the semi-colon at the end and also remember the use of #ifndef, #define and #endif in the header file.

2.2.2. Implementation of the Counter Class

Now let's play the role of the implementor. Once it has been defined, the Counter class needs to be implemented. This means implementing all of the functions specified in its definition. It is a good idea to write this implementation in a separate file from the definition. For example, the following code could be placed in a file called *Counter.C*.

```
// simple counter class: implementation of methods
// Steve Benford November 1991
#include "Counter.h"

// construct and initialise to 0
Counter::Counter() { count = 0; }

// reset data member count to 0
void Counter::reset() { count = 0; }

// return current value of data member count
int Counter::read() { return count; }

// increment the value of the data member count
void Counter::inc() { count++; }

// test whether the value of the data member count is zero
int Counter::iszero() {
     if (count == 0)
          return 1;
     else
          return 0;
}

// test whether the value of data member count is equal to that of c
int Counter::equals(Counter c) {
     if(count == c.count)
          return 1;
     else
          return 0;
}
```

The statement *#include "Counter.h"* tells the compiler to insert all of the code from the class definition file at this point of the implementation file. This must be done by any program which refers to the Counter class. You can think of it as merging the two files together. The UNIX pathname of the file appears between double quotes, so in this case it looks in the current directory.

The rest of the file consists of code implementing the interface functions. This is mostly straight forward as the functions are very simple. However, you should note the following points:

- There is no function *main()* so this is not a complete program.
- The code *Counter::* appears before each function name. This is necessary to tell the compiler to which class the function belongs. After all, there might be many "read" functions defined for different classes!
- Each function can refer directly to private data members of the class. Hence the statement *count = 0;* in the reset function.
- The functions can also access the data members of other objects of the same Class. To do this they specify the object name. Thus the equals functions compares its own data member against that of its argument "c" via the statement *if (count == c.count)*.

2.2.3. Using the Counter Class

Now let's assume the users perspective. The user wants to employ the class to solve a specific problem. The following program reads in a sentence of text ending with a full-stop, a character at a time. It counts the number of occurrences of the letter e (lower or uppercase) and prints out the result.

```
// test the counter class
// count all occurrences of the letter 'e' in a sentence
#include <stream.h>

// use the Counter class
#include "Counter.h"

main() {
     Counter count;
     char c;

     // read first character
     cin >> c;

     // loop until a fullstop
     while(c != '.' ) {
             // is the character upper or lowercase 'e'?
          if (( c == 'e')  || (c == 'E'))
               count.inc();
          cin >> c;
     }

     cout << "There were " << count.read() <<
          " letter 'e' in the text\n";
}
```

This program must also include the "Counter.h" header file. The statement *Counter count;* declares an object (variable) of Class (type) Counter, called count. This statement automatically calls the constructor function which initialises its internal data member to zero.

Inside the while loop, we see the statement *count.inc()*. This says invoke the inc() function on the object count. Its effect is to increase the internal value of count by one.

The statement *count.read()* invokes the read() function on count which returns its current value ready to be printed out.

Notice how easy it is to use the Counter class. Notice also how the user is oblivious of its internal implementation (and also how they cannot tamper with internal details!).

The program doesn't use the equals function. However, the following fragment of code shows it might be used.

```
Counter a,b;
if ( a.equals(b) )
```

```
cout << "They are the same!\n";
```

Notice how equals, a binary operation between two objects, is expressed - invoke the method ON one object and pass the other as an argument. This may be a little confusing at first and is important to take some time to understand.

2.3. Compiling classes

A consequence of splitting our code between different files is that the compilation process becomes a little more complex.

The implementor must compile the implementation file (Counter.C). However, because this is incomplete (no main()) they need to tell the compiler to stop short of linking the code into a final executable program. This is done via the **-c** flag to the g++ command. Thus, they enter the command:

g++ -c Counter.C

This places compiled versions of each method in a file called *Counter.o*.

The user must compile their own code and also link in the implementors complied methods. They do this with the command:

g++ myprog.C Counter.o

Where myprog.C is the name of their program file. Note that to use someone elses class you need to i) include the header file in your program ii) include the compiled functions when you compile your program. The following diagram summarises the files and commands involved in the process of implementing and using the Counter class.

```
                  -Counter.h-
                 /           #include Counter.h        #include "Counter.h"
               /
              /                     myprog.C        Counter.C
             |                   |
             |                   |
             |                   | g++ -c Counter.C
             |                   |
   g++ myprog.C Counter.o    Counter.o
             |                  /
             | -----------------
             |/
             |
           a.out
```

If the class user has access to the source code for the class implementation they could also enter the command:

g++ myprog Counter.C

This bundles togther all of the compilation and linking steps in the above sequences of commands into a single command.

2.4. Second example - the Money class

Our second example is rather more complex and will highlight some of the issues involved in designing good classes. Our goal is to define and implement a general purpose Money class which can be used to manipulate pounds and pence values. This class might be very useful for a variety of accounting applications (as we shall see in chapter 3).

2.4.1. Defining the Money class

Bearing in mind that we want a generally useful class, our biggest problem is how to design the right interface. Unfortunately there is no easy formula to be applied. Good class design is a matter of experience and trial-and-error. In spite of the impression that you might get in lectures, it is often an iterative process. This means that there might be many revisions before the definition is satisfactory. However, I will give you some guidelines that represent a good starting point. Remember they are guidelines - not a formula!

First, think what are the general properties of the Money class? Well, it stores pounds and pence values. You can add and subtract money. Subtraction implies that we can have negative values. You can compare money values (<, >, == etc). You might want to change an overall value or just inspect the pounds or pence components. You might want to print out money values in a special format.

My guidline is to think about methods which fall into five general categories: -

1. **Constructors** - how should objects of the class be initialised?
2. **Combination** - methods which combine existing objects into new ones (e.g. addition).
3. **Access** - methods which return of alter internal data members of an object.
4. **Tests** - methods which test or compare objects in some way.
5. **Input/Output** - methods which deal with input and output of objects.

After several iterations, I arrived at the following methods for the Money class.

Constructors: initialise to given pounds and pence values.

Combination: add, subtract.

Access: negate, return pounds, return pence, set pounds and pence, increase and decrease by another Money value.

Tests: ==, <, >, <=, >=, !=, is it zero?, is it positive or negative?

I/O: print

These translate into the following C++ class definition which I put in the file *Money.h*.

```
// C++ class definition for money objects (pounds and pence values)
// Steve Benford - November 1991

#ifndef MONEY_H
#define MONEY_H

class Money {
    private:
     int pnce;          // total number of pennies

    public:
     // constructor function
     Money(int pounds, int pence);    // construct with supplied values
                              // 0 <= pence <= 99 and 0 <= pounds
     // combination functions
     Money add(Money m);         // add two money values
     Money subtract(Money m);    // subtract two money values

     // access functions
        void negate();           // change sign of value (+ve <-> -ve)
                          // this allows negative values
     int pounds();          // return number of pounds
     int pence();           // return number of pence
     void set(int pounds, int pence);// set to new value
     void increase(Money m);     // increase current value by m
     void decrease(Money m);     // decrease current value by m

     // test functions      // compare values
     int equals(Money m);        // equal to?
     int less(Money m);     // less than?
     int greater(Money m);       // greater than?
```

```
     int less_equals(Money m);    // less than or equal to?
     int greater_equals(Money m);     // greater than or equal to?
     int is_zero();               // has zero value?
     int is_positive();      // is the value +ve or -ve

     // I/O functions
     void print();           // display in the format
                         // +/-$pounds.pence
};

#endif
```

The most difficult design decision was how to build negative Money values. At first, I thought that the user could supply negative values of pounds in the constructor (e.g. *Money a(-6,40)*). However, initialising a sum such as -40 pence would be impossible this way because the computer interprets -0 as 0. In the end I decided that the constructor could only build positive values and the user would have to subsequently use the negate() method to make them negative. This is clumsy but consistent.

2.4.2. Implementing the Money class

There are several possible implementations of the money class. My first thought was to use two internal integer data members called *pounds* and *pence*. Each method would then manipulate these as appropriate. For example, adding would involve adding the pounds and pence values from two objects. However, because the pence data member should have a maximum value of 99, this would involve some quite complex calculations in all combination and test operations (particularly for subtraction involving negative values). So, in the end I chose a second approach - use only one data member to represent the total number of pence. This simplifies most calculations. The only extra work is now in the print() pounds() and pence() methods. However, I think that this is easier. This kind of trade off is common to a variety of problems. Here is the implementation file for the Money class (in *Money.C*).

```
// Implementation of the Money class
// Steve Benford - November 1991
#include <stream.h>
#include <stdlib.h>
#include "Money.h"

// constructor - check and initialise values
// NOTE: can only initialise to positive value
// negative values must be obtained via the negate function
Money::Money(int pounds, int pence) {

     // check for pounds out of range
     if (pounds < 0) {
          cerr << "Error initialising Money: pounds out of range\n";
          exit(-1);
     }

     // check for pence out of range
     if ( (pence < 0) || (pence > 99) ) {
          cerr << "Error initialising Money: pence out of range\n";
          exit(-2);
     }

     // set data value to total number of pennies
     pnce = (pounds * 100) + pence;
}

// add two Money objects
Money Money::add(Money m) {
     // declare result object
     Money res(0,0);
     res.pnce = pnce + m.pnce;
```

```
     return res;
}

// subtract two money objects
Money Money::subtract(Money m) {
     // declare result object
     Money res(0,0);
     res.pnce = pnce - m.pnce;
     return res;
}

// negate current value
void Money::negate() {
     pnce = -pnce;
}

// return number of pounds (absolute value)
int Money::pounds() {
     return abs(pnce / 100);
}

// return number of pence (absolute value)
int Money::pence() {
     return abs(pnce % 100);
}

// is the value positive
int Money::is_positive() {
     if(pnce >= 0)
          return 1;
     else
          return 0;
}

// set to a new value
void Money::set(int pounds, int pence) {

     // check for pounds out of range
     if (pounds < 0) {
          cerr << "Error initialising Money: pounds out of range\n";
          exit(-1);
     }

     // check for pence out of range
     if ( (pence < 0) || (pence > 99) ) {
          cerr << "Error initialising Money: pence out of range\n";
          exit(-2);
     }

     // set data value to total number of pennies
     pnce = (pounds * 100) + pence;
}

// increase current value
void Money::increase(Money m) {
     pnce = pnce + m.pnce;
}

//decrease current value
void Money::decrease(Money m) {
     pnce = pnce - m.pnce;
}

// are two values equal?
int Money::equals(Money m) {
```

```
    if(pnce = m.pnce)
        return 1;
    else
        return 0;
}

// is one value less than another?
int Money::less(Money m) {
    if(pnce < m.pnce)
        return 1;
    else
        return 0;
}

// is one value greater than another?
int Money::greater(Money m) {
    if(pnce > m.pnce)
        return 1;
    else
        return 0;
}

// is one value less than or equal to another?
int Money::less_equals(Money m) {
    if(pnce <= m.pnce)
        return 1;
    else
        return 0;
}

// is one value greater than or equal to another?
int Money::greater_equals(Money m) {
    if(pnce >= m.pnce)
        return 1;
    else
        return 0;
}

// is a money value zero
int Money::is_zero() {
    if(pnce == 0)
        return 1;
    else
        return 0;
}

// print a money value
void Money::print() {
    int p;
    // display sign and money symbol
    if (!is_positive())
        cout << "-";
    cout << "$";

    // calculate and print pounds and pence values
    cout << abs(pnce / 100) << ".";
    p = abs(pnce % 100);
    if(p < 10)
        cout << "0" << p << "\n";
    else
        cout << p << "\n";
}
```

Note the use of the *abs* function to get absolute (i.e. positive) values and the remainder (%) operation in the print() and pence() functions.

2.4.3. Testing the Money class

The following user program shows how Money objects might be built, added and subtracted. The program does nothing useful other than show the syntax of manipulating Money objects.

```
#include <stream.h>
#include <stdlib.h>
#include "Money.h"

main() {
     int Pounds, Pence;
     char ch;

     // read in first money value
     cout << "enter a - pounds: ";
     cin >> Pounds;
     cout << "pence: ";
     cin >> Pence;
     Money a(Pounds, Pence);
     cout << "negate? ";
     cin >> ch;
     if(ch == 'y')
          a.negate();

     // read in second money value
     cout << "enter b - pounds: ";
     cin >> Pounds;
     cout << "pence: ";
     cin >> Pence;
     Money b(Pounds, Pence);
     cout << "negate? ";
     cin >> ch;
     if(ch == 'y')
          b.negate();

     Money c(0,0);

     // test add function
     cout << "a + b is ";
     c = a.add(b);
     c.print();
     cout << "\n";

     // test subtract function
     cout << "a - b is ";
     c = a.subtract(b);
     c.print();
     cout << "\n";

     // test a comparison operator
     if(a.less_equals(b))
         cout << "a <= b\n";
     else
         cout << "a > b\n";

}
```

2.5. Third example - the Set class

The final example in the chapter is a **Set** class used for manipulating sets of characters. Even though it is a complex example, these notes only present the source code and a few comments about its design. It is left as an "exercise for the student" to work through the example in detail.

A set is a collection of items of some type (in this case single characters). Each item can only appear once in the set and the items are in no particular order. Thus, unlike the arrays and structured from last term,

there is no way of saying "give me the 5th item" or "give me the item called Fred". Instead, the Set provides the following basic operations:

- Include a new element in the set.
- Remove an element from the set.
- Form the *intersection* of two sets (i.e. a new set consisting of the elements in common).
- Form the *union* of two sets (i.e. a new set consisting of the elements in either).
- Form the *difference* of two sets (i.e. a new set consisting of elements in the first but not in the second).
- It is also possible to test whether an element is in the Set.
- It is also possible to compare sets (do they contain the same items, is one a subset of the other etc).

2.5.1. Definition of the Charset class

The following header file contains the class definition for the **Charset** (set of characters) class. Notice how the operations are divided into the five categorise mentioned above.

```
// Definition of a class for a set of characters
// Steve Benford - November 1991

#ifndef CHARSET_H
#define CHARSET_H

// upperbound on the possible number of elements in the set
const int MAXSIZE = 500;

class Charset {
    private:
     char set[MAXSIZE];      // array storing elements in the set
     int n_elems;            // number of elements at any one time

    public:
     // constructor function
     Charset();              // construct an initially empty set

     // access functions
     int include(char c);         // add a character to the set
                       // return status (success or
                       // failure due to array overflow
     int remove(char c);          // remove a character from the set
                       // indicate success or failure due
                       // to element not being in set
     int size();             // how many elements are in the set?

     // combination functions
     Charset unionof(const Charset &s);    // set union
     Charset intersection(const Charset &s); // set intersection
     Charset difference(const Charset &s);// set difference

     // test functions
     int equals(const Charset &s);    // are two sets equal?
     int subset(const Charset &s);    // subset?
     int superset(const Charset &s);  // superset?
     int in(char c) const;        // is an element in the set?
     int isempty();               // is the set empty?
     int disjoint(const Charset &s);  // are the two sets disjoint?

     // I/O functions
       void print();         // display contents in the form { ... }
};

#endif CHARSET_H
```

Notice how we use the *&* symbol in front of the variable name whenever we pass a Charset as a parameter. For example, we see,

```
int equals(const Charset &s);    // are two sets equal?
```

We call this passing paramters by reference (whereas the usual way is called passing by value). This is discussed in detail in chapter 4. For now, you need to know that this reduces the amount of copying of data that the computer does whenever the function is called. Pass by value means that the function uses a copy of the parameter not the original whereas pass by reference means that the code inside the function actually accesses the original value.

When passing large paramters such as a Charset, pass by reference makes the program run faster and use less of the computer's memory. HOWEVER, this does mean that the function can acually change the original value. We call this a side-effect and sometimes it can be useful and sometimes it can be positively dangerous. Putting the word *const* in front of the parameter as we see in the Charset example prevents side-effects. In other words, the definition

```
int equals(const Charset &s);    // are two sets equal?
```

Can be read as "there is a function called equals with one parameter of type Charset called s. For reasons of efficiency, use pass by reference for s by for reasons of safety disallow any side effects from happening."

There is one further wrinkle to be considered. We shall see below in the implementation file that some of these functions call the member function in. In order for const to work as required, the programmer has to inform the compiler that this function (in) will never try to alter the value of its object. This is why the word const appears after its decalration:

```
int in(char c) const;       // is an element in the set?
```

2.5.2. Implementation of the Charset class

The implementation of the Charset class uses an internal array to store the elements of the set. The integer data member *n_elems* indicates how many elements are in the set and, consequently, how much of the array is currently being used.

It is important to note that using an array introduces a practical restriction on how many elements can be in the set. This is determined by the constant MAXSIZE.

The implementation of the Charset methods involves the kind of array processing that you saw last term (e.g. finding the elements in common). The implementation file (Charset.C) is listed below.

```
// Implementation of the Charset class functions
// Steve Benford - November 1991
#include <stream.h>
#include "Charset.h"

// construct an empty set
Charset::Charset() {
     // set number of elements to 0
     n_elems = 0;
}


// test whether the element c is in this set
int Charset::in(char c) const {
     // search through array containing elements
     for (int i = 0; i < n_elems; i++)
          if ( set[i] == c )
               return 1;
     return 0;
```

```
}

// include an element in a set
int Charset::include(char c) {
     // first make sure that the element is not already in the set
     if (in(c) == 0) {
          // check the array bounds to make sure there is space
          if (n_elems == MAXSIZE) {
               return 0;
          }
          // add the element to the array
          else {
               set[n_elems] = c;
               n_elems++;
               return 1;
          }
     } else // element already in set, do nothing
          return 1;

}

// remove a single character from this set
int Charset::remove(char c) {
     // first look for the element in the array
     for (int i = 0; i < n_elems;  i++)
          // if we find it, then we remove it
          if (set[i] == c) {
                   // shuffle the remaining elements down the array
               for(int j = i+1; j < n_elems; j++)
                    set[j-1] = set [j];
               // decrease number of elements
               n_elems--;
               return 1;
          }
     // if we get to this statement, we didn't find the element
     return 0;
}

// return the union of this set and the given set, s
Charset Charset::unionof(const Charset &s) {
     // build a new set called res
     Charset res;
     int i;

     // include all the elements of this set in res
     for(i=0;i< n_elems; i++)
          res.include(set[i]);
     // include all three elements of the given set, s, in res
     for(i=0;i<s.n_elems; i++)
          res.include(s.set[i]);
     return(res);
}

// return the intersection of this set and the given set, s
Charset Charset::intersection(const Charset &s) {
     // build a new set called res
     Charset res;
     int i;

     // for each of the elements in this set ...
     for(i=0; i<n_elems; i++)
          // ... see if it is in set s ...
          if (s.in(set[i]))
               // ... if so, include it in res.
               res.include(set[i]);
```

```
    return(res);
}

// return the difference between two sets
Charset Charset::difference(const Charset &s) {
    // build a new result set called res
    Charset res;
    // for each of the elements in this set ...
    for(int i = 0; i< n_elems; i++)
        // ... if it isn't in the given set,s ...
        if( s.in(set[i]) == 0)
            // ... then include it in the result set
            res.include(set[i]);
    return(res);
}

// test whether this set is a subset of the given set
int Charset::subset(const Charset &s) {

    // see whether each element of this set is in s
    for(int i=0;i<s.n_elems;i++)
        if(in(s.set[i]) == 0)
            return 0;

    return 1;
}

// test whether this set is a superset of the given set
int Charset::superset(const Charset &s) {
    for(int i=0;i<n_elems;i++)
        if(!s.in(set[i]))
            return 0;

    return 1;
}

// test whether two sets are equivalent (contain the same characters)
int Charset::equals(const Charset &s) {

    // first, are the number of elements the same (quick test)
    if (n_elems != s.n_elems)
        return 0;

    // now, is one a "subset" of the other
    if ( subset(s) )
        // same number of elements and a subset => return true
        return 1;
    else
        // same number of elements but not the same elements
        return 0;
}

// test whether this set is empty
int Charset::isempty() {
    if (n_elems == 0)
        return 1;
    else
        return 0;
}

// test whether this set and the given set have no members in common
int Charset::disjoint(const Charset &s) {
    // our method is to build the intersection and test whether it is empty
    // take the intersection of this set and the given set
    Charset temp = intersection(s);
```

```
    // is it empty?
    if (temp.isempty() == 1)
         return 1;
    else
         return 0;
}

// return the number of elements in the set
int Charset::size() { return n_elems; }

// display the set in the format "{ .... } (n_elems)"
void Charset::print() {

    cout << "{ ";
    // print each element in separated by spaces
    for(int i=0; i< n_elems; i++)
         cout << set[i] << " ";
    // print closing bracket and number of elements
    cout << "}" << " (" << n_elems << ")\n";
}
```

Several points are worth noting about this implementation.

- Removal of an element involves deleting an array element and shuffling the rest of the array one place down.
- The later methods build on the earlier ones. Thus, *include* and *remove* use *in*, and *disjoint* uses *intersection* and *isempty*. This saves alot of implementation effort.
- When one method calls another method on the same object it only needs to give the method name and not the object name as well.

2.5.3. Testing the Charset class

The following simple test program reads text from the keyboard until end of file (control-D if from the keyboard) and prints out all of the characters used in the text. This is gracefully achieved by building up a set of characters.

```
// program to record and then display the set of characters
// used in some input text

#include <stream.h>
#include <stdlib.h>
#include "Charset.h"

main () {
    char c;
    Charset s;

    // read until the end of file (^D for keyboard input)
    while (cin.eof() == 0)  {
         // read a character (ignore whitespace)
         cin >> c;
         // include this in the set
         if(s.include(c) == 0) {
              cerr << "Array overflow in set s\n";
              exit(-3);
         }
    }

    // print the set
    s.print();
}
```

Note how the program tests the results of the include method in case of array overflow.

Chapter 3 : Interacting Classes

3.1. Introduction

Abstract Data Types, implemented as C++ classes, define our approach towards designing and building large systems. Clearly, a large system might involve a combination of many classes and complex classes may be constructed out of simpler ones. This chapter explores these so called "using relationships" among classes and introduces some features of C++ that are necessary to support them.

The chapter also gives some further insights into how to analyse complex problems and approach the design of modular programs.

The chapter is structured around the example of a simplified cashpoint machine which allows users to carry out transactions on bank accounts. In turn, the bank accounts make use of the Money class defined in the previous chapter.

3.2. Simple analysis of the Cashpoint

The goal of our example is to simulate the operation of a cashpoint machine. In order to make the problem manageable, we will make some major simplifications to the real-world scenario.

1. There will only be one cashpoint machine to deal with many accounts (not many machines sharing many accounts).
2. We will not consider ordering statements and new cheque-books through the cashpoint.

The first, and often most difficult, stage of solving a problem like this is to analyse the task and derive an overall program structure. In the last chapter we discussed an approach towards designing class interfaces based around five categories of method (interface function). However, this assumes that we already know which classes we want. In this example we start a stage further back - we need to identify the correct classes for our program before we can specify their interfaces. In other words there is an **analysis** stage before a detailed **design** stage.

As with the last chapter, there is no magic formula to be applied. However, as a starting point we can try writing a brief summary of what the cashpoint simulator does. The summary consists of short sentences with all "nouns" and "verbs" highlighted (in this case using bold and italicised type respectively).

> The **Cashpoint** allows **users** to *access* their **accounts**.
>
> They may *withdraw* **money** or *request* a **balance**.
>
> Each **Account** has an **account number** and a **Personal Identification Number (PIN)**.
>
> These are *Checked* when the user *logs on*.

The nouns suggest possible classes, data members, function arguments and results for the program (cashpoint, user, account, money, balance, account number and PIN).

The verbs suggest possible methods for classes (access, withdraw, request-balance, check).

The next stage is to decide which nouns represent classes (i.e. types of general object) and which represent data members of these classes. My decision was that Cashpoint, Account and Money look like classes; balance, account number and PIN look like data members of account; and that the users are something outside of the program.

Let us now write a brief description of each class and identify some initial methods.

- The **Cashpoint** class provides access to a number of bank accounts. The class supports *access* to accounts, *withdrawal* of money and *balance requests*.
- The **Account** class specifies a single bank account. It maintains data about account numbers, PINs and the current balance.
- The **Money** class we already know.

Of course, this analysis is very basic. The following design stage will involve fleshing out each class with a complete complement of methods and data members to make it sufficiently general to solve the problem (and also to potentially be re-used in future problems).

We have already seen the design of the Money class. The rest of the chapter looks at the design and implementation of the Account and Cashpoint classes.

3.3. The Account Class

Let us briefly apply the five categories of interface function to the Account class:

- Constructors - initialise with zero balance.
- Combination - none appropriate.
- Access - set and return the account number, PIN and balance. The balance is manipulated through two functions - *credit* and *debit*.
- Tests - verify given PIN and account numbers.
- I/O - print out the account details.

This leads to the following definition file (*Account.h*).

```
// Account.h - definition of a class to represent simple bank accounts
// No interest paid!
// Steve Benford Novemer 1991
#ifndef ACCOUNT_H
#define ACCOUNT_H

#include "Money.h"            // use the Money class

class Account {
    private:
     int account_id;       // account number
     Money bal;            // current balance
     int PIN;        // Personal Identification Number
    public:
     // constructor
     Account();          // create an account
                    // with balance of 0 and no initial PIN

     // access
     int set_account_id(int id);
                    // set the account id for this account
     int get_account_id(); // return the account number
     int allocate_pin();   // allocate a new PIN number and return it
     int check_pin(int p); // check supplied PIN number - return true/false
     void credit(Money m); // increase balance by amount m
     int debit(Money m);   // decrease balance by amount m
                    // and return the amount debited
     Money balance();      // return current balance

     // I/O
     void print();         // print account details
};

#endif
```

Notice that the Account class contains a data member of type Money. Here we see one possible relationship between classes - one class can be built on another. Note also that the Money class appears in arguments and results of Account’s methods. This means that any program using the Account class should know how to use the Money class as well.

The design carefully ensures that the PIN number is only released outside the class once, when it is first allocated (so the account owner can be told what it is). It is allocated internally and can be checked, but not read, by other programs. Here we are using the encapsulation property of classes to increase security.

Here is the implementation of the Account class.

```
// Account.c - implementation of account class
#include <stream.h>
#include <stdlib.h>
#include "Account.h"
#include "Money.h"

// construct a new account with a balance of 0 and no initial PIN number
// notice how Account calls the constructor for Money
Account::Account(): bal(0,0) {
     PIN = 0;
}

// set the id for this account
int Account::set_account_id(int id) {
     account_id = id;
}

// return the account number for this account
int Account::get_account_id() {
     return account_id;

}

// allocate a random PIN number in the range 1000 to 9999
int Account::allocate_pin() {
     PIN = (rand() % 9000) + 1000;
     // tell the outside world what the PIN was
     // the only time it goes outside this account!
     return PIN;
}

// check the given number against the PIN
// notice that the PIN is not returned
int Account::check_pin(int p) {
     if (p == PIN)
          return 1;
     else
          return 0;
}

// credit an amount m to the account
void Account::credit(Money m) {
     bal.increase(m);
}

// debit an amount m from the account (no overdrafts!)
// return success or failure
int Account::debit(Money m)  {
     if(m.less_equals(bal)) {
          bal.decrease(m);
          return 1;
     } else
          return 0;
}

// return the current balance
Money Account::balance() {
     return bal;
}

// print account details (but not the PIN)
void Account::print() {
     cout << "Account id: " << account_id << " Balance: ";
     bal.print();
}
```

Several things need to be explained about this implementation.

The function *allocate_pin()* allocates a random pin number in the range 1000 to 9999. It uses the standard function *rand()* to generate a random number in the range 0 to RAND_MAX (usually 32767) and then modifies this to get it in the desired range (note how the remainder operation, %, is used). If you want to use this function in your own programs you need to include the header file <stdlib.h>.

See how the *credit* and *debit* functions call methods on the balance object (and also how debit won't let the account go overdrawn).

3.4. Constructors for Nested Objects

Look at the constructor function for Account. It has to initialise data members including *bal*, an object of class Money. This requires special treatment - the constructor for the data member *bal* is called directly after the function heading before its main body.

In general, THE CONSTRUCTOR FOR A CLASS HAS TO EXPLICITLY CALL THE CONSTRUCTORS FOR ANY CONTAINED CLASSES. These calls are made directly after the function heading (following a colon) and must take place in the order that the data members are declared in the class definition.

The one exception to this rule is if the constructor for the inner object has no arguments (e.g. the Counter() class from chapter 2). In this case it will be called automatically without the programmer having to worry.

It is useful to be aware that objects are constructed from the inside outwards - this means that the constructors for any inner objects are called before those for the outer object. This is illustrated by the following example program.

```
// demonstrate constructors for contained classes
#include <stream.h>

class Jim {
    private:
     int a;
    public:
     Jim(int i);
     // rest of class ...
};

Jim::Jim(int i) {
     cout << "Jim's constructor called\n";
     a = i;
}

class Jane {
    private:
     Jim a;
     int b;
    public:
     Jane(int i, int j);
     // rest of class ...
};

Jane::Jane(int i, int j): a(i) {
     cout << "Jane's constructor called\n";
     b = j;
}

main() {
     Jane(1,2);
}
```

Observe how the constructor for Jane has to call that for Jim in order to build the data member "a".

The following output is generated when the program is compiled and run, showing the order in which the various constructor functions are called.

```
Jim's constructor called
Jane's constructor called
```

3.5. The Cashpoint class

The definition of the Cashpoint class is as follows.

```
// Definition for a cashpoint controller class which looks after a set
// of accounts and manages withdrawals and balance requests
// statement and cheque book requests are not supported in this example
// the example also doesn't consider how accounts are created and closed
// or how different cashpoints can be used to access the same account
// Steve benford - November 1991

#ifndef CASHPOINT_H
#define CASHPOINT_H

// use the account class
#include "Account.h"

const int MAX_ACCOUNTS = 10;              // maximum number of accounts

// status codes returned by cashpoint functions
const int SUCCESS = 1; // operation carried out successfully
const int BUSY = -1;   // cashpoint currently dealing with another account
const int AUTH_FAILURE = -2;  // users PIN not verified
const int NO_ACCOUNT_OPEN = -3; // can't do operation as no account is open
const int INSUFFICIENT_FUNDS = -4; // not enough money for withdrawal
const int NO_SUCH_ACCOUNT = -5; // there is no account for the given id


class Cashpoint {
     private:
          Account accounts[MAX_ACCOUNTS];   // managed accounts
          int current;             // position of current account
                              // being dealt with (-1 = none)
     public:
          Cashpoint();             // initialise accounts
          int log_on(int id, int pin); // user logs on and gives
                              // account number and PIN
          int log_off();           // user logs off
          int withdraw(Money amount);       // withdraw some money
          int balance(Money &amount);       // put balance of account into
                              // "amount" and return status
                              // NOTE: pass by reference!!
};

#endif
```

The interface functions to the cashpoint allow the user to log on to a specific account (checking their PIN in the process), to withdraw money, obtain a balance of account and then log off.

Any of these could produce an error (e.g. trying to withdraw money before logging on to an account). The handling of these errors needs careful thought. A first reaction might be to print out error messages on the standard error channel from within the methods. This is a BAD idea because the Account class has no conception of what the user interface to the Cashpoint looks like (is it a dumb terminal or a PC with windows?). Instead, it is much better if the Account methods return a status code so that the program (or object) that called them can decide what to do. As a result, the Account class definition also defines the status codes that can be returned by its methods. These take the form of constant named integer values. This style of error handling is generally good practice.

Notice that becuase its result is a status code, the *balance* method actually passes the balance value back through the reference parameter *Money &amount*. This is an example of "pass by reference". In this case,

we are using pass by reference so that we can have the deliberate side-effect of passing a second value back out of the function. Because we want a side-effect we mustn't use the word const in the declaration of this paramter.

The implementation of the Cashpoint class is included below.

```
// Implementation of the Cashpoint class
// Steve benford - November 1991
#include "Cashpoint.h"
#include <stream.h>

// initialise accounts - not normally done in a cashpoint
// but necessary for this example
Cashpoint::Cashpoint() {
     int pin;

     // initial balance for accounts created
     // as 1000 pounds
     Money m(1000,0);

     for(int i = 0; i< MAX_ACCOUNTS; i++) {
          // set account number
          accounts[i].set_account_id(i);

          // allocate an initial PIN
          pin = accounts[i].allocate_pin();
          cout << "Account id " << i << " has PIN " << pin << "\n";

          // give an initial balance
          accounts[i].credit(m);
     }

     // note that there is no account currently in use
     current = NO_ACCOUNT_OPEN;
}

// user logs on - find account and check PIN
int Cashpoint::log_on(int id, int pin) {
     // make sure no other account is in use
     if (current != NO_ACCOUNT_OPEN)
          return BUSY;

     // look for an account matching the id
     for(int i = 0; i < MAX_ACCOUNTS; i++)
          if(accounts[i].get_account_id() == id)
               // if we found one, check the PIN
               if(accounts[i].check_pin(pin) == 1) {
                    // if PIN matches, open account for access
                    current = i;
                    return SUCCESS;
               } else  // PIN didn't match
                    return AUTH_FAILURE;


     // if we get here then no account is open and the ID must be wrong
     return NO_SUCH_ACCOUNT;
}

// log off
int Cashpoint::log_off() {

     // make sure that an account is open
     if(current == NO_ACCOUNT_OPEN) {
          return NO_ACCOUNT_OPEN;
     } else {
```

```
            // close the account
            current = NO_ACCOUNT_OPEN;
            return SUCCESS;
        }
    }

    // withdraw some money
    int Cashpoint::withdraw(Money amount) {

        // first check that an account is open
        if(current == NO_ACCOUNT_OPEN)
            return NO_ACCOUNT_OPEN;

        // withdraw the money
        if(accounts[current].debit(amount) == 1)
            // indicate that the withdrawal is OK
            return SUCCESS;
        else
            return INSUFFICIENT_FUNDS;
    }

    // return the balance of account AND indication of success/failure
    // the balance is assigned to the reference parameter "m"
    // the status is returned as the function result
    int Cashpoint::balance(Money &m) {

        // first check that an account is open
        if(current == NO_ACCOUNT_OPEN)
            // if not, return suitable error_code
            return NO_ACCOUNT_OPEN;
         else {
            // assign balance to parameter "m" and return SUCCESS
            m = accounts[current].balance();
            return SUCCESS;
        }
    }
```

The implementation of the Cashpoint simulator accesses an array of Account objects declared as in internal data member.

```
Account accounts[MAX_ACCOUNTS];   // managed accounts
```

The constructor for Cashpoint has to initialise the objects in this array. Of course, in the real-world, the accounts would be set up by some other program, not by the cashpoint machine each time it was started up. Also notice the syntax for invoking a method on an object stored in this array:-

```
accounts[i].set_account_id(i);
```

This example says, "get the ith element of the accounts array (an object of class Account) and invoke the set_account_id method on it".

3.6. The Interface Program

So far we have seen a Money class, an Account class which uses the Money class and a Cashpoint class which uses the Account and Money classes. To complete the program we need a "user interface" which communicates with the user and invokes the appropriate methods on a Cashpoint object. This is where the function *main()* will be defined.

```
// Interface to the Cashpoint object
// Steve Benford - November 1991

#include "Cashpoint.h"
#include <stream.h>
```

```
// function to process menu transactions on a cashpoint object
void do_transactions(Cashpoint &c);

// function to convert error codes into error messages
void print_error(int error);

// Function to work out how many notes of which denominations
// should be given out
void give_money(Money m);

main() {
     // set up cashpoint object
     Cashpoint cpt;

     int id, pin;  // account identification
     int status;   // return status of operations

     // log users on and off (infinite loop)
     while (1) {
          // prompt for and read the account id and pin
          cout << "please enter your account number: ";
          cin >> id;
          cout << "Please enter your PIN: ";
          cin >> pin;

          // try to log on
          status = cpt.log_on(id, pin);
          if(status == SUCCESS) {
               // do_transactions();
               do_transactions(cpt);
               // log this person off
               cpt.log_off();
          } else
               // print out a suitable error message
               print_error(status);
     } // while
} // main

// process the transactions for a specific session
void do_transactions(Cashpoint &c) {

     // menu selection
     char ch;

     // return status of operations
     int status;

     Money amount(0,0);
     int n;

     while(1) {
          // show menu of options
          cout << "\nw withdraw some money";
          cout << "\nb balance of account";
          cout << "\nq quit";
          cout << "> ";

          cin >> ch;

          // process selection
          switch(ch) {
               case 'w':
                    cout << "Enter amount (multiples of 5 only) >";
                    cin >> n;
                    // check the user entered a multiple of 5
```

```
                    if(n % 5 != 0) {
                         cerr << n << "not a multiple of 5\n";
                         break;
                    }
                    // set the value of amount and withdraw it
                    amount.set(n,0);
                    status = c.withdraw(amount);
                    if(status == SUCCESS)
                         give_money(amount);
                    else
                         print_error(status);
                    break;
               case 'b':
                    // look up the balance
                    status = c.balance(amount);
                    if(status == SUCCESS) {
                         cout << "your balance is currently ";
                         amount.print();
                         cout << "\n";
                    } else
                         print_error(status);
                    break;
               case 'q':
                    cout << "returning\n";
                    return;
               default:
                    cerr << ch << " is not a valid selection\n";
          } // switch
     } // while
} // do_transactions


// give out money!
// calculate the number of 50, 20, 10 and 5 pound notes needed to
// make up the total amount given.
void give_money(Money m) {
     int pnds = m.pounds();

     cout << "here's your money ...\n";
     // note the use of integer divide (/) and modulo (%) operations
     // to obtain the quotient and remainder of divisions.
     cout << pnds << "\n";

     cout << pnds / 50 << " fifty pound notes\n";
     pnds = pnds % 50;
     cout << pnds / 20 << " twenty pound notes\n";
     pnds = pnds % 20;
     cout << pnds / 10 << " ten pound notes\n";
     pnds = pnds % 10;
     cout << pnds /  5 << " five pound notes\n";
}

// process error codes and print out messages
void print_error(int error) {

     switch(error) {
          case SUCCESS:
               cout << "Operation completed successfully\n";
               break;
          case BUSY:
               cout << "Cashpoint busy with another account\n";
               break;
          case NO_ACCOUNT_OPEN:
               cout << "There is no account currently open\n";
               break;
```

```
            case NO_SUCH_ACCOUNT:
                cout << "There is no account with this ID\n";
                break;
            case AUTH_FAILURE:
                cout << "Invalid PIN\n";
                break;
            case INSUFFICIENT_FUNDS:
                cout << "There is not enough money in your account\n";
                break;
            default:
                cout << "Unknown error code generated by cashpoint!\n";
        } //switch
    } //print_error
```

The function *main()* sets up a Cashpoint object and then follows a continuous while loop, logging the user onto an account. When the user is logged on, the function *do_transactions* is called. This sits in a second continuous loop, presenting users with a menu of choices, reading their selection, processing it though a switch statement and invoking the appropriate methods on the cashpoint object. Logging out of the account returns to the outer while loop in the function main.

Two other useful functions are defined: *give_money* prints out a combination of 50, 20, 10 and 5 pounds notes which make up the amount withdrawn. *Print_error* converts status codes from the Cashpoint into error messages.

One feature of this design is that we could easily build alternative user interfaces to the same Cashpoint class.

3.7. File relationships and compilation

The overall system has been divided up into a number of modules representing different C++ classes. Provided that the header (.h) files are well defined, each implementation could be written independently by a different member of a programming team.

The following commands would be needed to compile the various components into a final executable program.

Compile the Money implementation:

    g++ -c Money.C

Compile the Account implementation:

    g++ -c Account.C

Compile the Cashpoint implementation:

    g++ -c Cashpoint.C

Compile the interface and link in all the other compiled code:

    g++ Interface.C Cashpoint.o Account.o Money.o

Note that the order of the .o files in the last command is significant!

Chapter 4 : Better Interfaces to Classes

4.1. Introduction

This chapter explores methods of making your own C++ classes easier and more intuitive for other programmers to use. By the end of the chapter you should be able to write classes which appear as if they were an integrated part of the C++ language.

Major topics include:

- Operator overloading - redefining existing C++ operators (e.g. +, - etc) to apply to your own classes.
- Using function overloading to provide multiple constructors for a class.
- Special techniques required to overload the input (>>) and output (<<) operators.
- Friends to a class.
- The use of references in function arguments.

The chapter is based around the example of a *Point* class for representing and manipulating two dimensional coordinates.

4.2. Operator overloading

Chapter one suggested that Abstract Data Types could support an extensible programming language where new general purpose data types could easily be introduced. Of course, the new ADTs should be as natural to use as the "inbuilt" data types (e.g. int and float in C++). However, this is not true of the ADTs we have seen so far. For example, the *Money* class of chapter two defines the method *add* to combine two money objects together.

```
Money add(Money m);
```

The following fragment of code uses this method:

```
Money a(1,30), b(2,40), c(0,0);
c = a.add(b);
```

Compare this with the addition of two integers:

```
Int a = 0, b = 1, c;
c = a + b;
```

The integer addition is much more natural because:

i) it uses the standard + operator;

ii) it has a more natural sense of symmetry.

C++ overcomes this problem by allowing us to redefine standard operators to apply to our own classes. This is called **operator overloading**. The basic mechanism is quite simple. When defining the Money class, instead of calling our method *add* we could call it *operator+*.

```
Money operator+(Money m);
```

A user program could now include the following code.

```
Money a(1,30), b(2,40), c(0,0);
c = a + b;
```

The compiler would recognise that the + operator is being applied to Money objects and would automatically convert this statement to:

```
c = a.operator+(b);
```

It is important to understand that this conversion actually takes place behind the scenes (see section 4.5 on overloading input/output operators).

Operator overloading is very flexible, and the operators that can be overloaded include the following.

```
+ - * / %
< > <= >= == !=
&& || !
```

We shall come across more later on in the course. The great danger with operator overloading is that implementors might overload operators to behave in unpredicatable counter-intuitive ways. Consequently, in order to avoid possible confusion, C++ places several constraints on the use of operator overloading.

First, the overloaded operator must retain the same "arity" as its standard definitions. For example, * must always be a binary operator and ! must always be unary. Of course, + and - can be both unary or binary. Thus, the following code fragment is illegal because it attempts to overload the + operator to have three arguments.

```
Money operator+(Money a, Money b); // THIS IS ILLEGAL!!!
```

Remember, there is always a first hidden argument which is the object on which the method is actually invoked!

Second, the priority of the operators will always remain the same. Thus, * always has higher priority than +, even when overloaded.

In general I would advise great caution when overloading. Only overload an operator when its effect clearly remains the same (e.g. * should always multiply things). Avoid making your classes harder to use by arbitrary overloading just for the sake of it.

As an additional comment, the argument types for overloaded operators can be anything you like provided the above rules are obeyed. Thus, I could define the following method in the Money class to add an integer number to a Money value.

```
Money operator+(int i);
```

4.3. An example - the Point Class

This section illustrates operator overloading through a *Point* class representing two dimensional coordinates (e.g. (x,y) map references). This class might be very useful for solving a range of geometry problems or in graphics applications which have to deal with screen positions. You might also like to think about how it could be extended to three dimensional coordinates or even to a generalised *Vector* class.

First, we will very briefly skip through the design of the Point class (in reality, a long process). A Point consists of a pair of floating point numbers called X and Y. Considering the five categories of interface function, we arrive at the following:-

Constructor: construct from two floating point numbers.

Combination: binary addition and subtraction between Point objects; unary negation of Point objects; scaler multiplication and division of Point objects.

Tests: all comparisons (<, >, <=, >=, ==, !=); Test whether the value is (0,0).

Access: Return the x and y values; return the magnitude (distance from the origin); return the angle from the X axis; set to new values of x and y.

I/O: display and read in the standard form (X,Y); draw on the X/Y plane.

Clearly, many of these methods are suitable candidates for operator overloading, resulting in the following class definition (file *Point.h*).

```
// Point.h - definition of a Point class to represent coordinates
// Steve Benford  - November 1991
#ifndef POINT_H
```

```
#define POINT_H

class Point {
    private:
      float X,Y;           // coordinates stored as floats
    public:
     // constructor
     Point(float x, float y);    // build a Point with initial value

     // combination
     Point operator+(Point &p);      // add two points together
     Point operator-(Point &p);      // subtract two points
     Point operator*(float f);       // scaler multiplication
     Point operator/(float f);       // scalar division
     Point operator+();              // unary +
     Point operator-();              // unary -

     // tests
     int operator<(Point &p);        // less than
     int operator>(Point &p);        // greater than
     int operator<=(Point &p);       // less than or equal to
     int operator>=(Point &p);       // greater than or equal to
     int operator==(Point &p);       // equal to
     int operator!=(Point &p);       // not equal to
     int operator!();                // is zero

     // access
     float x();                      // return x coordinate
     float y();                      // return y coordinate
     float magnitude();              // return magnitude
     int angle();                    // return angle from x axis
                       // in the range -180 < angle <= 180
     void set(float x, float y);     // set to new value

     // I/O
     void print();                   // print as (x,y)
     void draw();                    // draw on axis
     void read();                    // read from input as (x,y)
};

#endif
```

Notice that all Point arguments to methods have been passed as *reference arguments*. This is for reasons of efficiency (see section 4.7).

The implementation of the Point class is given below.

```
// Point.C - implementation of the Point class
// Steve Benford - November 1991

#include <stream.h>
// use the maths library for trig functions
#include <math.h>
#include "Point.h"

// construct a new point with initial values
Point::Point(float x,float y) {
     X = x;
     Y = y;
}

// add this point to given point, p, and return result
Point Point::operator+(Point &p) {
     // notice how we pass expressions to the constructor for result
     Point result(X + p.X, Y + p.Y);
```

```
     return result;
}


// subtract the given point, p, from this point and return the result
Point Point::operator-(Point &p) {
     Point result(X - p.X, Y - p.Y);
     return result;
}

// scaler multiply for this point
Point Point::operator*(float f) {
     Point result(X*f, Y*f);
     return result;
}

// scaler divide of this point
Point Point::operator/(float f) {
     Point result(X/f, Y/f);
     return result;
}

// unary + (change sign to +ve)
Point Point::operator+() {
     Point result(+X, +Y);
     return result;
}

// unary - (change sign to opposite)
Point Point::operator-() {
     Point result(-X, -Y);
     return result;
}


// return the value of the x coordiante
float Point::x() {
     return X;
}

// return the value of the y coordinate
float Point::y() {
     return Y;
}

// return the magnitude of the point
float Point::magnitude() {
     return sqrt(X*X + Y*Y);
}

// return the nearest whole number angle of the point with the X axis
// return it in degrees in the range -180 < angle <= 180
int Point::angle() {
     float degrees;

     // atan2 returns radians so we must convert to degrees
     // by multiplying by 57/32
     degrees = 57.32 * atan2(Y,X);
     return int(degrees);
}

// set new coordinates for this point
void Point::set(float x, float y) {
     X = x;
     Y = y;
```

```
}

// test whether this point has greater magnitude than the given point p
int Point::operator>(Point &p) {
     if(magnitude() > p.magnitude())
          return 1;
     else
          return 0;
}

// test less than
int Point::operator<(Point &p) {
     if(magnitude() < p.magnitude())
          return 1;
     else
          return 0;
}

// greater equals
int Point::operator>=(Point &p) {
     if(magnitude() >= p.magnitude())
          return 1;
     else
          return 0;
}

// less equals
int Point::operator<=(Point &p) {
     if( magnitude() <= p.magnitude())
          return 1;
     else
          return 0;
}

// are two points equivalent
int Point::operator==(Point &p) {
     if ((X == p.X) && (Y == p.Y))
          return 1;
     else
          return 0;
}

// are two points not equivalent
int Point::operator!=(Point &p) {
     if ((X != p.X) || (Y != p.Y))
          return 1;
     else
          return 0;
}

// is this point equal to the origin (0,0)
int Point::operator!() {
     if ((X == 0) && (Y == 0))
          return 1;
     else
          return 0;
}

// print this point in the form "(x,y)"
void Point::print() {
     cout << "(" << X << "," << Y << ")";
}

// draw this point on the X,Y plane
// not implemented yet!!!
```

```
void Point::draw() {
     cout << "(" << X << "," << Y << ")";
}

// read in this point in the form "(x,y)"
void Point::read() {
     char l,c,r;
     // read in brackets, comma and values
     cin >> l >> X >> c >> Y >> r;
     // check for a format error
     if ( (l != '(') || (c != ',') || (r != ')') ) {
          // if so, give warning and set to (0,0)
          cerr << "Point::read input format error\n";
          X = 0;
          Y = 0;
     }
}
```

Notice how we pass expressions to constructors as a convenient shorthand. For example, the implementation of *operator+* includes :-

```
     Point result(X + p.X, Y + p.Y);
```

Observe that the comparison operators are easily built on top of the *magnitude* function.

The standard mathematics function *atan2* is used to find the angle (in radians) of the point from the X-axis given its tangent. This is then converted to degrees. In order to use this function, the header file <math.h> is included. In addition the **-lm** flag must be passed to the compiler in order to include the binary for this function when a user program is compiled.

The *read()* function carefully checks the input format of a Point. However, *draw()* has not been implemented properly (I have just copied *print()*). You might like to think about how to do this yourselves.

The following program tests the Point class. It reads in a series of Points representing a route. The +, - and *magnitude* methods are then used to calculate the total distance travelled and also the distance from start to finish. Notice how easy it is to use the Point class.

```
// File route.C
// Read in a starting point and a series of coordinates.
// Calculate the total distance travelled as well as the distance
// between start and finish points

#include <stream.h>
#include "Point.h"
#include <math.h>

main() {
     // initial, previous and next points on the journey
     Point start(0,0), prev(0,0), next(0,0);

     char c;
     float distance = 0;     // record total distance travelled


     cout << "Please enter your initial point\n";
     start.read();
     prev = start;

     do {
          cout << "enter the next point on your journey\n";
          next.read();
          distance = distance + (next - prev).magnitude();
          prev = next;
```

```
            cout << "do you wish to enter another point? (y/n)\n";
                cin >> c;
        } while( c == 'y');

        cout << "Total distance travelled " << distance << "\n";
        cout << "Distance between start and finish " <<
            (next - start).magnitude() << "\n";
}
```

Compiling this program requires the command: *g++ route.C Point.o -lm* (don't forget the maths library).

4.4. Function overloading

The general term "overloading" means using the same name to mean different things in different contexts. To recap, the same name can be used for different functions provided that the compiler can unambiguously match each function call from the number and types of its arguments. Remember, the type of the result and the names of the arguments are not taken into account. Thus,

```
int max(int a, int b);
int max(int a, int b, int c);
float max(float a, float b);
```

can be distinguished as separate functions. But,

```
float max(int a, int b);
int max(int x, int y);
```

would both clash with:

```
int max(int a, int b);
```

An important use of function overloading is in defining multiple constructors for a class. We often want a choice of how to initialise objects. For example, a Point could be initialised to the value (0,0), or to two specific values (x,y) or to the value of another Point. We could offer the user a choice by implementing the following three constructors:

```
Point(float x, float y);        // initialise to given values
Point();                        // initialise to 0
Point(Point &p);                // initialise to another Point
```

The following declarations would all be acceptable in a users program with the compiler matching arguments in order to determine which constructor to use.

```
Point a;
Point b(2.1, -3);
Point c(a);
```

In fact, the user could also write:

```
Point d = a;
```

and the compiler would recognise a call to *Point(Point &p);*. We call this last constructor the **copy constructor** because it copies one object to another of the same type.

We could also get the effect of the two constructors *Point()* and *Point(float x, float y)* by using default parameters. In this case we would define the constructor:

```
Point(float x = 0, float y = 0);
```

It is a matter of personal taste which approach you chose. It is left as an exercise for the student to look back over previous classes and to define additional constructors where appropriate.

4.5. Overloading input and output operators

The test program for the Point class demonstrates that operator overloading makes classes much easier to use. However, input and output still require specialised methods such as *print()* and *read()*. Ideally, we would like to use the insertion (<<) and extraction (>>) operators on stream objects (*cin, cout, cerr*) just as we do with inbuilt types. This can be achieved by overloading >> and <<. However, it is rather more complex than for other operators.

A user of the Point class wishes to write code such as:

```
Point a, b, c;
cin >> a;
cin >> b;
cout << a + b;
```

Consider the statement *cin >> a* in more detail. This would use the overloaded operator >> to read in a Point object. Section 4.2 told us that the compiler would map this onto the statement *cin.operator>>(a)*. This implies that we have to implement a new method *operator>>* which can be invoked on the object *cin*. Now cin is an object of class *iostream* (stands for "input stream") which is defined in the file <stream.h>. Unfortunately, this class definition is not ours to modify so we cannot write the overloaded member function!

One approach to this problem might be to define *operator>>* on the Point object. However, the user would have to write *a >> cin*. This is completely counter-intuitive and therefore a bad idea.

The solution to our problem lies in a new C++ technique called **friends**. We can define the following **non-member function** called *operator>>*.

```
iostream& operator>>(iostream& is, Point p);
```

We call this a non-member function because it is not defined inside a specific class (in contrast to member-functions). The functions you saw in PR1 were all non-member functions. The first argument of the function is an object of class iostream (e.g. *cin*) the second is an object of class Point. The user can now write:

```
Point a;
cin >> a;
```

And the compiler will map this into:

```
operator>>(cin, a);
```

In the general case we can apply operator overloading to both member and non-member functions. It is also true that any overloaded operator can be defined as either a member or non-member function. This choice is largely a matter of personal preference (the course examples use member functions where possible).

Notice that our overloaded operator returns a result of class iostream. This makes it possible for the user to write statements such as:

```
Point a, b;
int i, j;

cin >> a >> i >> b >> j;
```

Effectively these will be executed as a series of nested statements, with the result of each (*cin*) being used as the left side argument of the next.

```
(((cin >> a) >> i) >> b) >> j;
```

The implementation of this operator is as follows.

```
// overloaded input operator (friend function)
// "used to be read"
istream& operator>>(istream& is, Point& p) {
     char l,c,r;
     // read from input stream
     is >> l >> p.X >> c >> p.Y >> r;
     // check format
     if ( (l != '(') || (c != ',') || (r != ')') ) {
          cerr << "Point::read input format error\n";
          p.X = 0;
          p.Y = 0;
     }
     // NOTE: we need to return the input stream
     return is;
}
```

Notice how it resembles the *read()* function defined previously. Also notice how the input stream given as its argument is returned as its result. The prefix *Point::*does not appears before the function name because this is not a method of class Point.

4.6. Friends

There is still a problem with our overloaded input operator. Chapter 2 stated that a major feature of classes is that they don't allow external functions to access the private data inside the class. This means that our non-member function *operator>>* will not actually be allowed to alter that values p.X and p.Y.

This problem can be solved by using a technique called **friends**. In the class definition for Point we can specify that our new function is a "friend" of the class through the following statement.

```
friend istream &operator>>(istream& is, Point& p);
```

This gives the function permission to access private data members and functions of the class. In other words, being a friend to a class over-rides the protection mechanism.

In general, any non-member function can be declared as a friend. Futhermore, whole classes can be declared as friends meaning that any of their methods can access private members (more about this in chapter 7).

In general, use of the friends technique is bad form. Over-riding the protection mechanism provided by classes increases the chances of bugs and results in badly structured programs. I would discourage it except for exceptional circumstances such as overloading input and output operators. Treat friends as you would gotos!

The following is a revised class definition for the Point class which defines overloaded input and output operators to replace the member functions *print()* and *read()*. Note the use of the friends mechanism.

```
// File PointIO.h
// REVISED Point.h - definition of a class to represent coordinates
// revision replaces the member functions "print" and "read" with
// overloaded ">>" and "<<" operators.
// Steve Benford  - November 1991

#ifndef POINT_H
#define POINT_H

class Point {
    private:
       float X,Y;            // coordinates stored as floats
    public:
```

```
    // constructor
    Point(float x, float y);    // build a Point with initial value

    // combination
    Point operator+(Point &p);      // add two points together
    Point operator-(Point &p);      // subtract two points
    Point operator*(float f);       // scaler multiplication
    Point operator/(float f);       // scalar division
    Point operator+();              // unary +
    Point operator-();              // unary -

    // tests
    int operator<(Point &p);        // less than
    int operator>(Point &p);        // greater than
    int operator<=(Point &p);       // less than or equal to
    int operator>=(Point &p);       // greater than or equal to
    int operator==(Point &p);       // equal to
    int operator!=(Point &p);       // not equal to
    int operator!();                // is zero

    // access
    float x();                      // return x coordinate
    float y();                      // return y coordinate
    float magnitude();              // return magnitude
    int angle();                    // return angle from x axis
                      // in the range -180 < angle <= 180
    void set(float x, float y);     // set to new value

    // I/O
    // overloaded output operator
    friend ostream &operator<<(ostream& os, Point p);
    // overloaded input operator
    friend istream &operator>>(istream& is, Point& p);

    void read();                    // read from input as (x,y)
    void draw();           // draw on the X/Y plane
};

#endif
```

Here is the implementation of the non-member function *operator<<*.

```
// overloaded output operator (friend function)
// used to be "print"
ostream& operator<<(ostream& os, Point p) {
    // print on output stream
    os << "(" << p.X << "," << p.Y << ")";
    // return output stream
    return os;
}
```

Finally, here is the revised "route" program showing the use of the new operators.

```
// Read in a starting point and a series of coordinates.
// Calculate the total distance travelled as well as the distance
// between start and finish points

#include <stream.h>
#include "PointIO.h"
#include <math.h>

main() {
    // initial, previous and next points on the journey
```

```
        Point start(0,0), prev(0,0), next(0,0);

        char c;
        float distance = 0;     // record total distance travelled


        cout << "Please enter your initial point\n";
        cin >> start;
        prev = start;

        do {
             cout << "enter the next point on your journey\n";
             cin >> next;
             distance = distance + (next - prev).magnitude();
             prev = next;

             cout << "do you wish to enter another point? (y/n)\n";
                 cin >> c;
        } while( c == 'y');

        cout << "Total distance travelled " << distance << "\n";
        cout << "Distance between start and finish " <<
             (next - start).magnitude() << "\n";
}
```

4.7. References

The examples in the chapter have been using the reference symbol (**&**) to pass function arguments and results. References were introduced in PR1 as a means to alter the argument passing mechanism and to enable "side-effects" where a function can alter values in the outside program. The *swap* function was given as a classic example of this use of references.

Copying objects during the default "pass-by-value" mechanism can be very time consuming for complex classes. Using references avoids this copying and so also increase the speed of function calls. Thus, a second use of references is to increase the execution speed of programs that pass classes as function arguments. This is the use we are seeing in the Point class.

This use of references is generally a good idea, provided that the user and implementor are aware of possible side-effects that may take place.

Chapter 5 : Pointers

5.1. Introduction

This chapter introduces the concept of **Pointers** in C++. The use of pointers allows programmers to directly control the allocation of memory within a program. This makes possible the implementation of much more flexible ADTs which overcome the "fixed size" limitations inherent in types such as arrays. For example, pointers can be used to build dynamically expandable arrays or classes of "linked" objects such as lists, queues, stacks or trees.

The great flexibility of Pointers comes at some cost. First, the software engineer needs to be explicitly aware of how memory is allocated to the program. Secondly, the syntax of pointers, at least in C++, can be rather formidable. For these reasons, this entire chapter is devoted to the basics of using pointers. The early examples may often seem a bit trivial, overly complex or, dare I say, "pointless". However, it is worth establishing a thorough grounding before progressing to the really useful examples in later chapters.

This chapter covers the following topics:

- What is a pointer?
- Declaring and assigning pointers.
- Allocating and freeing memory with the operators **new** and **delete**.
- Pointers to objects.
- Pointers as function arguments.
- Pointers, pointer arithmetic and arrays.

5.2. Introducing Pointers

First, let's go back to basics and consider the properties of *variables*. A variable can be viewed as " a place where data is stored in a program" and has three key properties - its *name* (an identifier chosen by the programmer) its *type* (what kind of data it stores) and its current *value*.

The computer stores the value somewhere in its memory. This will typically be in a set of memory locations starting at some memory address. Thus, each variable also has an *address* property describing where its value is stored in computer memory. Whenever the programmer uses the name of the variable, the computer maps this into its address. However, the details of how this is done are hidden away from the programmer. In particular:

- The computer automatically allocates memory to store the value when the variable is first defined.
- The computer automatically frees this memory when the variable is destroyed (e.g. at the end of a program or function).

For simple programs, it is beneficial for programmers to be shielded from these details. However, sometimes this becomes a bit restrictive. For example, the fixed size of arrays results from the computer allocating a fixed number of memory locations to store the array when it is first declared. More advanced programming often requires the programmer to take control of their own memory allocation in order to build more flexible data types (e.g. dynamically expandable). This is achieved through the use of pointers.

A **Pointer** is a new kind of variable whose value is a memory address. Thus, a pointer can store the location of some data in memory. We say that it "points" at other values.

Like normal variables, pointers are "strongly typed". This means that an "integer pointer" can only store the addresses of integer values and a "char pointer" can only contain the addresses of char values.

The following program declares and uses a pointer named *p*. The program achieves nothing useful apart from illustrating the syntax for dealing with pointers.

```
// declare and use pointers example 1
#include <stream.h>

main() {
     int i, j, *p;
```

```
        i = 1;
        j = 2;

        p = &i;

        cout << "contents of i are: " << *p << "\n";

        p = &j;

        cout << "contents of j are: " << *p << "\n";

    }
```

The statement *int *p* declares a variable called p whose type is "pointer to integer". Initially, p doesn't point at anything. We can draw this as the following:

```
     ----
    p|  |----> ?
     ----
```

The statement *p = &i* stores the address of the variable i in p. In other words, it makes p "point at" i. We draw this in the following way.

```
     ----       ----
    p|&i|----> i| 1|
     ----       ----
```

The operator **&** is called the "address of" operator. It returns the address of whichever variable it is applied to. Now the value of i is 1 and the value of p is &i (address of i).

The statement *cout << "contents of i are: " << *p << "\n"* prints out the current value of i. It does this by following the pointer p. The code **p* means "get the value of whatever p is pointing at". We call this "dereferencing a pointer" and say that ***** is the *dereferencing operator*.

The statement *p = &j* resets the pointer p to point at the value of the variable j.

```
     ----       ----
    p|&i|----> j| 2|
     ----       ----
```

We can now display this value by *cout << *p*. Notice that at this moment in time *j* and **p* are synonymous.

To summarise:

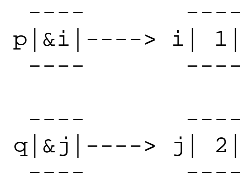
- a pointer can point at values by storing their addresses.
- A pointer is made to point at something by the "address of" operator **&**.
- A value can be accessed through a pointer via the "dereferencing operator" *****.

Now let us look at a second, more complex, example.

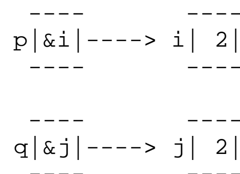
```
    // declare and use pointers - example 2
    #include <stream.h>

    main() {

        int i, j, *p, *q;


        p = &i;
        q = &j;
```

```
    *p = 1;
    *q = 2;

    cout << *p << *q << "\n";

    *p = *q;

    cout << *p << *q << "\n";

    p = q;

    *q = 3;

    cout << *p << *q << "\n";

    cout << "address: " << p << "   contains value: " << *p << "\n";

}
```

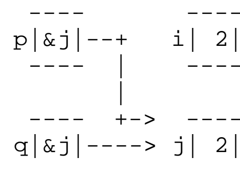
This example uses two pointers, *p* and *q*, and demonstrates the difference between assigning the values of pointers and the values of the things to which they point. We can draw the situation at the end of the statement *q = 2 as the following.



The statement *p = *q copies "the value of the thing q points at" (the value of j) to "the thing that p points at" (i). The situation is now:

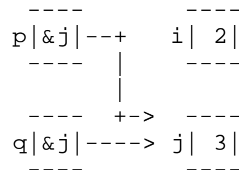


Contrast this with the statement *p = q*. This copies the value of the pointer q - i.e. the address of j, into pointer p. In other words, we have reset p to point at j. Both pointers now point at the same value. The situation is now:



It is vital to understand the difference between the statements $*p = *q$ and $p = q$. The first copies the values being pointed at. The second actually resets the pointers.

The statement $*q = 3$ now makes the following change:



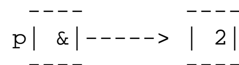
The final statement of the program prints out this value by accessing it through the pointer *p*.

5.3. Allocating and freeing memory with new and delete

So far we have used pointers to point at existing variables which had their memory automatically allocated. In this section we see how to allocate and free our own memory through the operations **new** and **delete**. Consider the following program:

```
// declare and use pointers - example 3
#include <stream.h>

main() {

     int *p = new int;

     *p = 3;

     *p = *p + 1;

     cout << *p << "\n";

     delete(p);
}
```

The statement *new int* allocates enough memory to store an integer value and returns its address as a result. The statement *int *p = new(int);* declares a pointer and makes it point at this new memory.

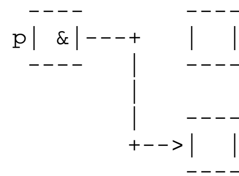


Notice that we are not pointing at an existing variable. The program then stores the value 3 in this memory and then adds 1 to its value before printing it out. The final statement *delete(p)* tidies up by freeing the memory currently pointed at by p.

In general, **new** allocates enough memory to store a value of the specified type and returns its address. Its argument can be any C++ type, including user defined classes. **Delete** frees the memory pointed at by the given pointer. For those interested in details, the available memory is actually stored in an area called the *heap*. New grabs some memory from the heap and delete returns it back to the heap. It should be stressed that all memory is automatically freed at the end of the program - it is not allocated for all time. However, it is a very good idea to make sure that there is an explicit call to delete for each call to new. This style of tidy programming reduces the chances of bugs occurring. In particular watch out for things like:

```
int *p;
p = new(int);
p = new(int);
```

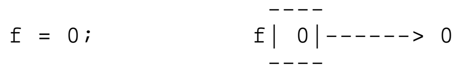
The first call to new allocates some memory and sets p to point at it. The second allocates some more and sets p to point at this. This is shown below:



The first piece of memory is no longer pointed at by anything. If there was a value stored there, it would no longer be accessible. We call this a *hanging memory* problem and it is to be avoided.

In order to avoid this problem, we need to be able to test whether a pointer already points at some memory before we allocate any new memory. When we first declare a pointer, in points no-where.

We can explicitly set it to point at the **null address** by a statement such as the following.



In this case, 0 represents the null address. This is often drawn as the electrical earth symbol.

We can explicitly test for the null address. The following fragment of code allocates new memory for the pointer f. However, it checks to see whether it first needs to delete any old memory.

```
if(f == 0)
        f = new float;
else {
        delete f;
     f = new float;
}
```

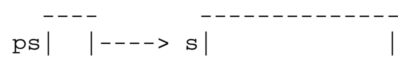
This is a particularly important piece of code to understand and will come up in many examples.

5.4. Pointers and objects

Pointers can point at user defined classes.

```
Charset s, *ps;
ps = &s;

*ps.print();
```

This fragment of code constructs a new Charset and sets the pointer *ps* to point at it.



The statement **ps.print()* invokes the *print* method on the object that ps points at (i.e. on s).

We can also use *new* to initialise an object. In this case **new automatically calls the constructor for the object** as well as allocating memory. We need to give *new* any arguments needed by the constructor. The general syntax is *pointer* = **new** *type*(*arguments*). For example,

```
Money *m;
m = new Money(3,20);
```

The operator delete frees the memory and calls a special function called the **destructor** to tidy up. Destructors are optional in a class and we shall not come across them until a bit later. Note that the brackets around new and delete arguments are optional.

```
delete m;
```

The cumbersome notation **a.method()* can be written as *a->method()*. Thus we can write:

```
    Money *m = new Money(3,20);
m->print();
```

This tends to be a bit easier to read. We can also access data members this way - thus **a.X = 3* becomes *a->X = 3*.

In general, the operator **->** is extremely useful for accessing the members of objects through pointers and we will use it many times throughout the rest of the course.

5.5. Pointers as function arguments

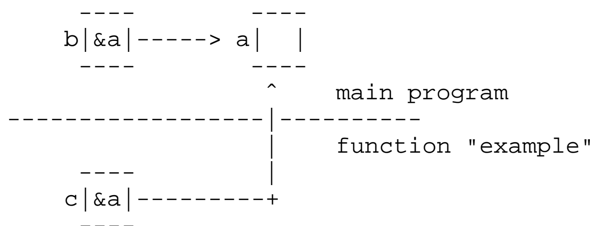
Pointers can be passed as function arguments and returned as function results. Consider the following function which takes a pointer to a character as its argument.

```
// print out a character
void example(char *c) {
        cout << *c;
}
```

This might be called in the following ways:

```
char a, *b;
b = &a;

example(b);
example(&a);
```

Once inside the function call *example(b)* we have the following situation.



The formal parameter *c* takes the value *&a* and therefore points at the variable a. Any change to **c* in the function would change the value of *a*. In other words, the function can cause "side effects" by changing the value of a variable in the main program. This is the same effect as "pass by reference". In fact, the computer implements pass by reference by passing addresses behind the scenes.

We call passing pointers as function arguments "pass by address".

We can also return results this way. For example:

```
int *example2(int a, int b) {
        int *res = new int;
        *res = a + b;
        return res;
}
```

However, **NEVER RETURN THE ADDRESS OF A LOCAL VARIABLE THAT HAS BEEN AUTOMATICALLY ALLOCATED INSIDE A FUNCTION!**. This is because the variable will be automatically freed at the end of the function and the resulting address will no longer be valid. Thus, although the above function is valid, the following is incorrect:

```
// THIS CODE IS INCORRECT AND VERY DANGEROUS
// BECAUSE WE RETURN THE ADDRESS OF AN AUTOMATICALLY ALLOCATED
// LOCAL OBJECT
int *example3(int a, int b) {
 int res;
        res = a + b;
        return &res;
}
```

You can only safely return the address of memory that you have allocated yourself.

We have now seen three ways of passing arguments to functions:

- by value - a formal argument is created and the value copied in. This does not cause side effects in the calling program.
- by reference - the formal parameter and calling program refer to the same memory so side-effects may occur. However, the user program looks the same as for call by value.
- by address - addresses are passed and so side-effects can occur. The calling program needs to be aware that addresses are being used and the syntax is more difficult to follow.

We can see that pass by reference and pass by address are similar in effect. The main difference is in the syntax - pass by reference appears more "natural" for the user of a function. In effect, the computer is adding a layer of gloss to addresses to make them appear as references.

5.6. Pointers and Arrays

There is a very close relationship between pointers and arrays in C++. In fact, we shall see that they can almost be used interchangeably. Before we embark on this discussion, I should point out that the syntax you are about to see often becomes very terse and confusing. Furthermore, these aspects of C++ are not really found in other languages to the same degree (except for C). However, even if you don't use them yourself, you will need to be able to understand them in other peoples programs.

Pointers can point at array elements. For example, the following code uses a pointer to read values into an array.

```
int a[10], *p;

for(int i = 0; i < 10; i++) {
     p = &a[i];             // point to ith array element
     cin >> *p;             // read a value into this element
}
```

Pointers are C++ types and some of the standard operators have been overloaded to apply to them. These include:

```
+ - ++ --
< > <= >= == != !
>> <<
```

The idea of "pointer arithmetic" makes most sense when pointers are combined with arrays. If the pointer *p* currently points at array element *a[i]*, then:

- *p+1* is a pointer to *a[i+1]*.
- *p-3* is a pointer to *a[i-3]*.
- *p++* moves the pointer to point at the next element.
- *p = p + 3* moves the pointer to point 3 places on down the array.

Also, if *p* points to *a[i]* and *q* points to *a[i+1]*, then we can assert that *p < q* evaluates to true.

We have already seen that if *p* points to the null address then *p == 0* evaluates to true. Of course *!p* evaluates to false (this could be written *p != 0*).

The following for loops are equivalent

```
for(i = 0; i < 10; i++)
        cout << a[i];

p = &a[0];
for(i = 0; i < 10; i++)
     cout << *(p + i);
```

Notice the use of parentheses in **(p + i)*. This is because operator * (pointer dereference) has higher priority than operator + (addition). The statement **p + i* means "get the value pointed at and add i to it". In other words, increase the value of an element of the array. The statement **(p + i)* means "increment the pointer and get what it now points at". In other words, get the value of the next array element.

Now we can go one step further. **The name of an array is the same as a pointer to its first element!**

Thus, the statement *p = &a[0]* could more simply be written as *p = a*. Also, *a[10]* can be written as **(a + 10)*. The following for loop is a valid way to print out the values of the array *a*.

```
for(p = a; p < a + 10; p++)
        cout << *p;
```

In essence, subscript and pointer array access are interchangeable - **a[i] <=> *(a + i)**.

All of this may seem confusing and pointless because it introduces no new functionality into C++. However, we will see later examples where it is more convenient to process arrays through pointers than through subscripts. The following program defines a function to search an array for a specified value. It uses pointers, not subscripts, to access the array elements.

```
// demonstrate pointers and arrays
#include <stream.h>

const int MAX = 5;

// function to determine if value  n occurs in the given integer array
int find(int n, int a[MAX]) {
     int *p;

     // compare with each element in the array
     for (p = a; p < a + MAX; p++)
          if(*p == n)
               return 1;
```

```
     // if we got here then the element isn't in the array
     return 0;
}

main() {
     int fred[MAX], *q, j;

     for(q = fred; q < fred + MAX; q++) {
          // note the pointer subtraction returns an integer
          cout << "please enter array element " << q - fred;
          cin >> *q;
     }


     cout << "enter target value: ";
     cin >> j;

     // note that arrays are passed by address!
     if ( find(j, fred) )
          cout << "element was found in array\n";
     else
          cout << "element was not found in array\n";
}
```

Notice also that because the name of an array is a pointer to its first value, arrays are always passed by address and never by value.

5.7. Character arrays

We finish this chapter with a brief discussion of character arrays, or "character strings" as they are often called. Character arrays have a special representation in programs as literal strings. For example,

```
"Hello World"
```

Of course, character strings may be of different lengths. As a result, the computer appends an additional null character to them in order to show where they end. This can be quoted as the literal character '\0'. Thus the statement:

```
char *message = "Hi Fred";
```

Defines a new character array (pointer) with the following initial value.

```
        ----      ------------------
message|    |----> |H|i| |F|r|e|d|\0|
        ----      ------------------
```

The length of this array is 8 characters because the computer has automatically appended the additional \0. We can display this array with the statement *cout << message*.

In general, the type *char ** should be interpreted as a null terminated character array.

As a final example, the following function returns the length (number of characters) in an arbitrary sized character array. Note that it doesn't count the null character at the end.

```
int strlen(char *c) {
        int i = 0;
        char *p = c;
```

```
        while(*p != '\0') {
             i++;
                p++;
        }

        return i;
}
```

Chapter 6: Expandable ADTs

6.1. Introduction

This chapter shows how to use pointers to overcome the fixed size limitations inherent in data types such as arrays. The chapter is based around two example C++ classes. The first, *Intarr*, is a dynamically expandable array class which implements bounds checking on subscripts. The second is a much more complex *String* class for manipulating text. Its functionality includes comparison of strings, concatenation, substring matching and input and output.

In order to build these classes some new C++ concepts are introduced. These include:

- overloading the array subscript operator ([]);
- providing destructors for a class;
- The need to provide a copy constructor for a class.
- The problem of *memberwise assignment* and the need to overload the assignment operator (=);

These techniques turn out to be generally applicable to a range of new C++ classes.

6.2. Intarr - a dynamic array

The goal of the *Intarr* class is to provide a resizable array of integers which supports bounds checking.

First, it is necessary to remember the general properties of the in-built C++ array data type. An array is an ordered collection of elements all of which are of the same base type. Elements are accessed by their position in the array through a subscript using the *[]* operator. There are two major drawbacks to C++ arrays.

- Once defined, they have a fixed size. This makes them unsuitable for storing highly dynamic kinds of information such as general text (the length of a sentence can vary greatly).
- The [] operator doesn't check whether the given subscript falls within the array bounds. This makes arrays dangerous to use.

6.2.1. Definition of the Intarr class

The new *Intarr* class will provide an array of integers with all of the usual functionality of arrays and with the above drawbacks removed. The interface to Intarr supports the following functions:

```
Constructor: set up an array with an initial specified
             length and all elements initialised to zero.

Access: return a specified element from the array.
        return the current size of the array
        reset the size of the array (either increase or decrease)

I/O: print out the elements in the array
```

This leads to the following class definition.

```
// C++ class definition for a dynamically resizable array of
// integers with bounds checking

class Intarr {
    private:
        int *a, len;

    public:
        //constructor
        Intarr(int length);

        // destructor
        ~Intarr();
```

```
        // return nth element
        int& operator[](int n);

        // return size
        int size();

        // reset the size
        void resize(int length);

        // print
        void print();
};
```

Notice the inclusion of the method *~Intarr*. This is called the **destructor** for the class and is automatically called when an object of the class is destroyed (i.e. "goes out of scope"). This might occur at the end of the program, at the end of a function (for a local object) or when the user explicitly deletes an object using the *delete* operator. Any class may have a destructor and the destructor may take any action the user deems necessary or useful. One common use of destructors is to print out a message saying that the object has been destroyed in order to aid the debugging process. For classes which allocate their own memory for data members, a destructor is definitely required. The name of the destructor is always the name of the class with the ~ character in front of it.

This example shows that it is also possible to overload the array subscript operator *[]*. This allows users of the class to write code such as the following:

```
    Intarr a(6);
    a[1] = 2;
    cout << a[1];
```

The argument to this operator is the array subscript (an integer number). The result is an element of the array. This result MUST be returned using the reference mechanism! This is because we wish to include elements on both the right and left hand sides of assignment statements. Appearing on the right hand side means that we wish to change the original element (e.g. *a[1] = 2*). In other words we want the function to deliberately cause a side effect on the Intarr object and so we must have a reference result type.

6.2.2. Implementation of the Intarr class

Our implementation of this class requires two data methods - an array of integers to store its elements and a count of how big the array currently is. The latter is to be used in bounds checking and resizing. The array is declared as an integer pointer (remember that pointers and arrays are inter-changeable). Thus we have:

```
      int *a, len;
```

The constructor function needs to allocate enough memory to store the requested number of elements and set the pointer *a* to point at this memory. It also needs to record the initial length of the array and to initialise all the elements to the value zero. This is achieved by the following code:

```
Intarr::Intarr(int length) {

        // allocate memory for the array
        a = new int[length];

        // note the length
        len = length;

        // initialise all elements to 0
        int i;
        for(i = 0; i < len; i++)
                a[i] = 0;
```

```
}
```

Notice that we can use the operator *new* to allocate memory for an array of objects in one go by telling it how many elements there are between square brackets.

The destructor is responsible for freeing the memory allocated by the class whenever an Intarr object is destroyed. It is necessary to do this because we explicitly allocated the memory ourselves in the first place.

```
// destructor frees the memory
Intarr::~Intarr() {

        // free memory for len elements
        delete [len]a;
}
```

Notice that in order to free memory for an array of objects, we have to tell *delete* how many objects there are.

Operator[] first checks that the given subscript is in bounds and, if it is, simply returns the specified element of the array *a*.

```
// return reference to nth element
int& Intarr::operator[](int n) {

        // check bounds (remember arrays start at 0)
        if( 0 <= n && n < len)
                // if in bounds return reference
                return a[n];
        else {
                // indicate error and stop the program
                cout << "Error in Intarray operator[]: subscript "
                     << n << " is out of bounds\n";
                exit(-1);
        }
}
```

The *size()* function merely returns the value of the data member *len*.

```
// return size of array
int Intarr::size() {
        return len;
}
```

The *resize* function is a little trickier. Its first action is to set up a temporary new array with the correct new length. Next it copies across as many elements of the old array as it can into the new one. If the new array is longer, it sets the additional elements to zero. If it is shorter, it truncates the old array. Then it frees the memory for the old array using *delete*. Finally, is sets the data member *a* to point at the new array and resets *len* to the new length.

This technique of setting up new memory and copying values across will also be used in several later classes.

```
// resize array to 'length' elements
void Intarr::resize(int length) {

     // first check that the new length is not the same as the old
     if ( length == len )
          return;

        // allocate new memory and new length
        int *newa = new int[length];
```

```
		int newlen = length;

		// note the shorter of the new and old lengths
		int min;
		if (newlen < len)
			min = newlen;
		else
			min = len;

		// copy across old elements
		// (truncate if new length is shorter)
		for(int i = 0; i < min; i++)
			newa[i] = a[i];

		// set any remaining elements to 0
		for(i = min; i < newlen; i++)
			newa[i] = 0;

		// free old memory
		delete [len]a;

		// set new memory and new length
		a = newa;
		len = newlen;
	}
```

Finally, the *print* method prints out the array in the form *a,b,c ...z*.

```
	// print out array
	void Intarr::print() {
		int i;
		for (i = 0; i < len; i++)
			cout << a[i] << ",";
		cout << "\n";
	}
```

6.2.3. Testing the Intarr class

The following simple program demonstrates the syntax for using the Intarr class. Notice how the overloaded [] operator makes the class appear much more like in-built arrays. You might like to think about other possible overloaded operators for this class (e.g. overloading >> and <<).

```
	// test the Intarr class
	#include <stream.h>
	#include "Intarr.h"

	main() {
		Intarr a(3);

		a[0] = 0;
		a[1] = 1;
		a[2] = 2;

		a.print();

		cout << "size of a is" << a.size() << "\n";

		a.resize(5);
		a.print();
		a[3] = 3;
		a[4] = 4;
```

```
        // this next one would cause an error
        // a[5] = 5;

        a.print();

        a.resize(2);
        a.print();
    }
```

The following briefly summarises the key new points introduced through the Intarr class:

- The constructor needs to allocate memory using the *new* operator.
- The destructor needs to free this memory using the *delete* operator.
- Both *new* and *delete* can deal with arrays of objects provided they are told how big the arrays are.
- The overloaded *[]* operator must return a reference result so that it can be used on the left hand as well as the right hand side of assignment statements.
- The resize method works by setting up new memory, copying across data from old memory and then deleting the old memory.

6.3. The String class

Manipulating text is an important part of many programs. Text might come in the form of words, sentences, names, paragraphs or entire documents. C++ provides the basic *char** type for manipulating text. However, this is extremely limited. In effect, we can only:

- Point to a string (e.g. *char *s = "Hello"*).
- Index individual letters of a string (e.g. *if (s[0] == 'H') ....*
- Display a string (e.g. *cout << s*).

There are many additional facilities that might be included in a general purpose string class.

- The ability to copy strings to each other.
- The ability to concatenate (i.e. join-together) strings to form new strings.
- The ability to compare strings based on dictionary ordering (e.g. <, >, == etc).
- The ability to read strings using the >> operator and print them using <<.

A string class that supported such functionality would be extremely useful for many applications. Our goal is to design and implement such a class. There are many additional functions that might be useful that we won't consider in our class. These include case conversion, encryption and reversing strings.

6.3.1. Definition of the String class

My analysis of the *String* class led to the following interface design.

```
Constructors:  build an empty string; a blank string of a given length;
               a string from an existing char*; a string from another
               string.

Combination:   concatenate two strings to form a new one (overload
               operator +).

Tests:         Test for empty string (!) as well as compare strings
               ( <, >, <=, >=, ==, !=). Also does one string contain
               another?

Access:        Return the length of the string.
               Access individual letters (overloaded []).

Input/output:  Overload >> and << to read and write strings
```

The analysis above leads to the following class definition.

```
// definition of a character string class
// Steve Benford Nov 1990, Revised Nov 1991
#include <stream.h>

class String {

    private:
        char* str;                      // pointer to character array
        int len;                        // length of the string;

    public:
        // constructors
        String(char *s);                // allocate and initialise
        String(int i);                  // allocate length and set to blanks
        String();                       // allocate to zero length
        String(const String& s);        // copy constructor

        // destructor
        ~String();                      // destructor;

        // combination
        void operator=(String& s);      // assignment between two strings
        String operator+(String& s);    // concatenation

        // tests
        int operator!();                // test whether 0
        int operator==(String& s);      // test for equality
        int operator!=(String& s);      // test for inequality
        int operator<(String& s);       // test alphabetically less than
        int operator>(String& s);       // test alphabetically greater than
        int operator<=(String& s);      // test alphabetically less than/equals
        int operator>=(String& s);      // test alphabetically greater/equals
        int contains(String& s);        // does one string contain another?

        // access
        char& operator[](int n);        // return char in position n
        int length();                   // return length of the string

        // I/O
        // overloaded output operator
        friend ostream& operator<<(ostream& os, String &s);

        // overloaded input operator
        // read a sequence of characters ending in a newline into parameter s
        // s will be set to NULL if the line length is greater than
        // a specific length (BUF_LEN = 200)
        friend istream& operator>>(istream& is, String &s);

};
```

The class provides four constructors including the copy constructor *String(const String& s)* which builds one string from another. Note the word "const" in front of the argument declaration. Remember that we pass the String argument by reference for reasons of efficiency, but that this can cause side-effects (i.e. our constructor could change the string object that is its argument). The use of const prohibits these side-effects. In general, you can use const arguments when using references for increased efficiency both where you want to stop possible side-effects.

The String class provides a destructor and also makes extensive use of operator overloading, including overloaded << and >> operators. This also includes overloading the assignment operator, *void operator=(String& s)*, something that we haven't seen before. This will be fully explained below.

6.3.2. Implementation of the String class

Internally, the String class resembles the Intarr class in that there are two data members - a char* pointer to store the string and an integer to record its current length.

The first step of the implementation is to define two generally useful functions *strlength* and *strcopy* which calculate the length of a char* and copy one char* to another respectively. In fact, there are standard versions of these functions available in the string library called *strcpy* and *strlen*. However, it is worth understanding them for yourselves. It is important to note that *strcopy* assumes that the right amount memory has already been allocated for the new string before the function is called. If it hasn't, the function will fail and will probably cause the program to crash.

```
// generally useful functions
// calculate length of char*
int strlength(char* c) {

        int i = 0;
        // set up a new pointer to the front of the string

        char *p = c;
        // keep going till you reach the end of the string (i.e. null)
        while(*p++ != '\0')
                i++;
        return i;
}

// copy one char* to another char* (assume space allocated)
void strcopy(char *s, char *t) {
        // while not at end of string
        while(*t != '\0')  {
                // copy the current characters
                *s = *t;
                // move the pointers one notch along the strings
                s++; t++;
        }
        // put the null character at the end
        *s = '\0';
}
```

The first two constructors set up an empty string and a string of n blanks (space characters).

```
// construct an empty String
String::String() {
        // length is currently zero
        len = 0;
        // set up array with end of string in it
        str = new char[1];
        str[0] = '\0';
}

// construct a String initialised to n blanks
String::String(int n) {
        len = n;
        // allocate enough space for the string (+ null)
        str = new char[len+1];
        // copy blanks into the memory
        for(int i=0; i<n; i++)
                str[i] = ' ';
        // set end of string marker
        str[n] = '\0';
}
```

Constructing a string from a char* involves recording the length of the string, allocating memory for the internal *str* pointer and then copying the char* pointed at by the argument *s* to this memory. Note that we must allocate anough memory for the null character ('\0') as well (i.e. len+1 chars).

```
// construct a String from a char*
String::String(char *s) {
        len = strlength(s);
        // allocate enough space for the string (+ null)
        str = new char[len+1];
        // copy the string into this memory
        strcopy(str,s);
}
```

The copy constructor is similar, except that we copy the internal char* given by *s.str*.

```
// copy constructor' one string from another
String::String(const String& s) {
        // set the length
        len = s.len;
        // allocate space
        str = new char[len + 1];
        // fill it up
        strcopy(str, s.str);
}
```

The destructor is responsible for freeing the memory allocated to the string.

```
// destructor
String::~String() {
        // delete the memory associated with the string
        delete str;
}
```

Concatenation involves setting up a new temporary string whose length is the combined length of the strings to be joined. The contents of both strings are then copied into this new memory.

```
// concatenation
String String::operator+(String& s) {

        // initialise result string of correct length
        String res(len + s.len);

        // copy in first part
        strcopy(res.str, str);

        // copy in second part
        for (int i = 0; i< s.len; i++)
                res.str[len + i] = s.str[i];

        // set end of string marker
        res.str[res.len] = '\0';

        // return result
        return res;
}
```

The following code implements the various test methods in the string class. Notice that when testing equality we first compare the lengths of the strings before each character in turn.

```
// test for empty string
int String::operator!() {
        return !len;
}

// test for inequality
int String::operator!=(String& s) {

        // first, are strings the same length?
        if(len != s.len)
                return 1;

        // if they are, test individual characters
        for(int i=0; i < len; i++)
                if (str[i] != s.str[i])
                        return 1;

        // if we get here they are the same
        return 0;
}

// test for equality
int String::operator==(String& s) {
        // are strings the same length?
        if(len != s.len)
                return 0;

        // test individual characters
        for(int i=0; i < len; i++)
                if (str[i] != s.str[i])
                        return 0;

        // if we get here they are the same
        return 1;
}

// test whether alphabetically less than
int String::operator<(String& s) {
        // set two pointers to the start of each string
        char *p = str, *q = s.str;

        // compare each character in turn while not at end of either string
        while((*p != '\0') && (*q != '\0')) {
                if (*p < *q)
                        return 1;
                if (*p > *q)
                        return 0;
                // move pointers along the string
                p++;
                q++;
        }

        // is q longer than p
        if ((*p == '\0') && (*q != '\0'))
                return 1;
        else
                return 0;
}

// test alphabetically greater than
int String::operator>(String& s) {
        // set two pointers to the start of each string
        char *p = str, *q = s.str;

        // compare each character in turn while not at end of either string
```

```
        while((*p != '\0') && (*q != '\0')) {
                if (*p > *q)
                        return 1;
                if (*p < *q)
                        return 0;
                // move pointers along the string
                p++;
                q++;
        }

        // is p longer than q?
        if ((*p != '\0') && ( *q == '\0') )
                return 1;
        else
                return 0;
}

// test alphabetically less than/equals
int String::operator<=(String& s) {
        // set two pointers to the start of each string
        char *p = str, *q = s.str;

        // compare each character in turn while not at end of either string
        while((*p != '\0') && (*q != '\0')) {
                if (*p < *q)
                        return 1;
                if (*p > *q)
                        return 0;
                // move pointers along the string
                p++;
                q++;
        }

        // we are at the end of string p?
        if ((*p == '\0'))
                return 1;
        else
                return 0;
}

// test alphabetically greater/equals
int String::operator>=(String& s) {
        // set two pointers to the start of each string
        char *p = str, *q = s.str;

        // compare each character in turn while not at end of either string
        while((*p != '\0') && (*q != '\0')) {
                if (*p > *q)
                        return 1;
                if (*p < *q)
                        return 0;
                        // move pointers along the string
                p++;
                q++;
        }

        // are we at the end of string q?
        if ((*q == '\0'))
                return 1;
        else
                return 0;
}

// does one string contain another
int String::contains(String& s) {
```

```
        int i,j,match;

        // test lengths first to see if contained is longer than container
        if(s.len > len)
                return 0;

        // go down string looking to match first character
        // note: we don't need to go to the end of the array
        for(i = 0; i < len - s.len; i++)
                // do characters match?
                if(str[i] == s.str[0]) {
                        // if so ASSUME a positive match for the whole word
                        match = 1;
                        // now check a character at a time to see
                        // if the assumption was true
                        for(j = 0; j < s.len; j++)
                                // oops, it wasn't
                                if(str[i+j] != s.str[j])
                                        match = 0;
                        // do we still have a positive match?
                        if(match)
                                return 1;
                }

        // if we got here there was no match
        return 0;
}
```

Like in the *Intarr* class, the overloaded [] operator allows access to individual elements of an array object.

```
// index an element of the String using the array [] operator
char& String::operator[](int n) {
        // check bounds
        if(n > len) {
                cout << "String error: reference out of bounds\n";
                exit(-1);
        }
        else
                return str[n];
}

// return the length of the string
int String::length() {
        return len;
}
```

Here are the overloaded input and output operators.

```
// overloaded output operator (friend function)
ostream& operator<<(ostream& os, String &s) {
        if(s.len > 0)
                os << s.str;
        return os;
}


// overloaded input operator (friend function)
// the strategy is to read into an array one character at a time
// (watching for overflow of the array bounds)
// and then to use the array to initialise a string object
istream& operator>>(istream& is, String &s) {

        // temporary array to hold input
```

```
        const int BUF_LEN = 200;
        char buffer[BUF_LEN];

        // read a character at a time until end of line or buffer overflow
        int i = 0;

        // keep going till overflow
        while(i < BUF_LEN) {

                // read next character (use get function)
                is.get(buffer[i]);

                // check for end of input (i.e. newline)
                if(buffer[i] == '\n') {
                        // put in end of string character
                        buffer[i] = '\0';
                        // initialise the string object s
                        s = String(buffer);
                        // return
                        return is;
                }
                // increment i
                i++;
        }

        // the buffer has overflowed if we get here
        // so set s to be the empty String
    String temp;
        s = temp;
        return is;
}
```

6.4. Overloading the assignment operator

In this section we discuss overloading the assignment operator (=). First we find out the motivation for doing this. Then we look at how it is achieved.

C++ allows us to assign objects of user-defined classes directly to each other. Thus, the following is valid.

```
Point a(1,0), b(2,1);
b = a;
```

When the assignment takes place, the computer simply copies the values of the data members of b into those of a. In other words,

```
b.X = a.X;
b.Y = a.Y
```

This simple approach is called **memberwise copying** and is the default technique for copying objects. It works fine for simple classes with automatically allocated memory (e.g. Point, Counter and Money). However, it fails for classes which allocate their own memory due to the effect of copying pointers. Consider the following.

```
String a("Hi"), b("Yo");
b = a;
```

This has the effect of:

```
b.len = a.len;
b.str = a.str;
```

This produces the following effect:

There are now two major problems:

- *a.str* points at the wrong place. What happens if *b* changes or is deleted?
- The memory containing "Yo" is now hanging.

The solution to this problem is that we need to replace the memberwise copying assignment mechanism with something more sophisticated. This is achieved by overloading the assignment operator (*operator=*). This operator is called when ever two objects are copied - THIS INCLUDES PASSING FUNCTION ARGUMENTS AND RESULTS BY VALUE!

We want our version to do the following:

- Free the old memory pointed at by *b.str*.
- Allocate sufficient new memory for *b.str*.
- Copy across the contents of *a.str* into *b.str*.

This is achieved by the following code.

```
// assignment
void String::operator=(String& s) {
        // first delete the old memory
        delete str;

        // copy across new string
        len = s.len;
        str = new char[len + 1];
        strcopy(str, s.str);
}
```

It is crucial to note that this operator will be called for argument and result passing in functions as well as for direct assignment.

It will be necessary to overload the assignment operator for any class which allocates and frees its own memory.

In summary, the String class has had to consider memory allocation and deletion in the following places:

- Constructors
- Destructors
- Concatenation
- Assignment

It is possible to think of many interesting extensions to the String class:

- Reformating - case conversion, encryption, reversal, compression
- Matching - wildcards, case-ignore
- Combination - splitting, selecting parts

Chapter 7: Linked ADTs

7.1. Introduction

In this chapter, we look at the use of pointers to build "linked" data types. A linked data type allows separate objects to be linked together into more complex information structures. There are many examples of such linked structures including *lists, queues, trees* and *graphs.* It is not the goal of this course to explore these in detail - this is left to a later course on *Data Structures*. Instead, we focus on the relatively simple example if a *list* in order to demonstrate the overall approach.

This chapter introduces the following key concepts:

- Linking objects through pointers.
- Making one class a "friend" to another.

7.2. A Linked List ADT

Lists are a familiar feature of our lives and examples include shopping lists and address books containing lists of names. For our purposes we shall consider a list to have the following basic properties:

- The elements of the list are ordered.
- The elements of the list are of the same type.
- The list may be "empty" - in other words have no elements.
- The list may grow to an arbitrary size.
- It is possible to insert an element anywhere into the list.
- It is possible to delete an element from anywhere in the list.

The properties of insertion and deletion are shown below and make lists quite different from arrays where these operations are not possible.

```
Carrots --> Peas --> Cabbage --> Parsnips
```

{ insert "Sweetcorn" into position 3 }

```
Carrots --> Peas --> Sweetcorn --> Cabbage --> Parsnips
```

{ delete "Cabbage" from position 4 }

```
Carrots --> Peas --> Sweetcorn --> Parsnips
```

These basic properties led me to the following design of a list Abstract Data Type (based on our five categories of interface function).

```
Constructors: build an empty list (no elements)
              build a list from another list (copy constructor)

Access:       insert a new element into position n
              delete the element at position n
              retrieve the value at position n (overload [])
              return the length of the list (number of elements)

Combination:  join two lists together to get a new one (operator +)

Tests:        is the list empty (operator !)
              are two lists equal (operator ==)
              are two lists not equal ( != )

I/O:          print a list in the form [a] -> [b] -> [c] -> oo
              read a list from the keyboard
```

7.3. Definition of a list class

We will limit our first example to be a list of single characters. However, it should be easy to see how to extend this to other types, including your own classes. The design above yields the following class definition for a new *List* class. Notice that in this definition I have left out the data members - these will be introduced in the implementation section below.

```
// class to represent a linked list of characters
class List {
    private:
        // to be completed
    public:
        // constructors
        List();
        List(List& l);

        // combination
        List operator+(List& l);

        // access
        int insertN(char c, int n);
        int deleteN(int n);
        int retrieveN(char &c, int n);

        // tests
        int operator!();
        int operator==(List& l);
        int operator!=(List& l);
        int length();

        // I/O operators
        friend ostream& operator<<(ostream& os, List& l);
        friend istream& operator>>(istream& is, List& l);
};
```

Notice that the *retrieveN* method places the retrieved value into the argument *char &c* - a deliberate side effect. As a result, this argument must be passed by reference. List arguments are also passed by reference, but in this case for reasons of increased efficiency.

7.4. Implementing the List Class

The implementation of the List class uses the fact the pointers can be used to link objects together. First, consider the following definition of a simple *Element* class.

```
class Element {
    private:
        // value stored in this element
        char val;
        // pointer to next element in list
        Element *next;
    public:
        Element(char);
};
```

An element object may contain a single character (*val*) and also a pointer (*next*) which can be set to point at some other element. The following constructor function is provided to initialise an Element object.

```
// construct a single list element
Element::Element(char c) {
        val = c;
        next = 0;
}
```

Now, **assuming that our program has permission to manipulate the data-members of the class Element**, we can build chains of elements in the following way.

```
Element a('a'), b('b');
```

```
a.next = &b;
```

```
cout << a.val << b.val << a.next->val;
```

Or better ...

```
Element *first;
first = new Element('a');
```

```
first->next = new Element('b');
```

```
first->next->next = new Element('c');
```

```
first->next->next->next = new Element('d');
first->next->next->next->next = new Element('e');
first->next->next->next->next->next = new Element('f');
```

We have now managed to build up a chain of linked objects - in other words a simple list. The following code prints out the values in this list, in sequence.

```
Element *p = first;
```

```
while ( p != 0 ) {
        cout << p->val;
        p = p->next;
}
```

These ideas form the basis of our list implementation. However, instead of adding each element to the chain explicitly we need a much more flexible approach. This is given by implementing the methods *insertN* and *deleteN* in a general way so that we can access any part of an arbitrary long list.

Overall, our List implementation will require two classes - the basic Element class described above and the more general List class. Thus, once again we see an example of a using relationship between different classes.

The following is the completed header file which defines both classes. Note that we have also added some extra methods to the List class. These include a destructor and an overloaded assignment operator. These are required because we will we be doing our own memory allocation.

A new *searchN* method has been introduced into the private section of the class. This method returns the address of the nth element in the list (and the 0 pointer if there is no such element). This is generally useful for implementing *insertN, retrieveN* and *deleteN*. However, a user of the class should not see such addresses and so *searchN* is defined as a private method, not a public one. In other words, it can only be used by other methods of the List class.

```
// specification for an ADT representing a list of chars
// implementation using pointers to link elements together
#ifndef LIST_H
#define LIST_H

#include <stream.h>

// status codes returned by public functions
const int OK = 1;
const int OUT_OF_BOUNDS = -1;

// class to represent a single list element
class Element {
    private:
        char val;
        Element *next;
    public:
        Element(char);
        friend class List;
        friend ostream& operator<<(ostream& os, List& l);
};

// class to represent a linked list of characters
        class List {
    private:
        Element *first;
        int len;
        Element *searchN(int n);

    public:
        // constructors
        List();
        List(List& l);

        // destructor
        ~List();

        // combination
        void operator=(List& l);
        List operator+(List& l);
```

```
        // access
        int insertN(char c, int n);
        int deleteN(int n);
        int retrieveN(char &c, int n);

        // tests
        int operator!();
        int operator==(List& l);
        int operator!=(List& l);
        int length();

        // I/O operators
        friend ostream& operator<<(ostream& os, List& l);
        friend istream& operator>>(istream& is, List& l);
};
```

Notice that we define two error codes to be returned by public functions. One represents the fact that a requested element is not in the list (e.g. trying to delete the fourth element of a list that is three elements long). The other indicates that the operation completed successfully.

Note also that the class List is declared as a friend to the class Element. This enables ALL methods of List to manipulate the internal data-members of Element objects (e.g. making and breaking links via the *next* pointers). We also have to declare the overloaded output operator as a friend to both List and Element.

The List class defines two private data members, *len* which indicates how many elements are in the list at a given time and *first* which is a pointer to the first element in a chain. Thus the overall approach is to have a List object point to an Elment object which, in turn, is linked to a chain of further Element objects. The following figures show two example lists, one empty (i.e. no elements) and the other containing several elements.

Each of the methods defined for List will need to access and manipulate this structure in some way. We will now look at each of these methods in turn.

The constructor to build an empty list merely sets the pointer *first* to be null and notes that there are zero elements in the list.

```
// construct an empty list
List::List() {
        first = 0;
        len = 0;
}
```

Next, we need to look at *searchN*. The goal of this private method is to return a pointer to the nth element in the list (where *n* is its argument). To do this, a while loop starts at the first element and moves down the list, following the *next* pointers, until either the end of the list is reached (the 0

pointer) or it has seen n elements. Of course, it needs to keep a record of how many elements have already been seen. This is achieved through the local variable *int seen*. Finally, it returns a pointer to the element at which it stopped (this will be the 0 address if it ran over the end of the list).

```
// general function to return a pointer to the nth element
// return null pointer for error
Element* List::searchN(int n) {
        Element *pos = first;
        int seen = 1;

        // check bounds
        if((n < 1) || (n > len))
                return 0;

        // move down list one element at a time
        while( (pos != 0) && (seen != n)) {
                pos = pos->next;
                seen++;
        }

        // return pointer to element
        return pos;
}
```

Next we look at the insert function. There are two distinct cases to be considered - inserting at the beginning of the list and inserting somewhere in the middle or at the end of the list.

First, we check whether the subscript is in bounds. Then we construct a new Element with the correct value. If we are inserting at the first position, we link this new element to the one that *first* is currently pointing at and then link *first* to our new element. This is shown below.

If we are inserting in the middle of the list we get a pointer to the preceding element and then follow these steps:

- Link our new element to the ones following the preceding element.
- Line the preceding element to our new element.

In both cases we increase the length of the list.

```
// insert character c in position n
```

```
int List::insertN(char c, int n) {

        // check bounds (note we can insert in position n+1)
        if ((n > len+1) || (n < 1))
                return OUT_OF_BOUNDS;

        // build a new element to be added to the list
         Element *new_elem = new Element(c);

        // first element is a special case
        if(n == 1) {
                new_elem->next = first;
                first = new_elem;
        }
        else {
                // otherwise find element n-1
                Element *prev = searchN(n-1);

                // link new element to the following elements
                new_elem->next = prev->next;

                // ling preceding element to the new element
                prev->next = new_elem;
        }

        // increase the length of the list
        len++;

        return OK;
}
```

Deletion also starts with bounds checking. Beyond this, the procedure is the reverse to insertion. Notice that as well as making and breaking pointers, we also free the memory associated with the element being deleted.

```
// delete the element at position N in the list
int List::deleteN(int n) {

        // pointer to the preceding element and the element being deleted
        Element *prev, *pos;

        // check bounds
        if ((n > len) || (n < 1))
                return OUT_OF_BOUNDS;

        // first element is a special case
```

```
        if (n == 1) {
                // remember the old first element
                pos = first;
                // reset the start of the list
                first = first->next;
        }
        else {
                // find previous element
                prev = searchN(n-1);
                // remember the element to be deleted
                pos = prev->next;
                // chop element out of list
                prev->next = pos->next;
        }

        // delete the element and free its memory
        delete pos;

        // decrease the length of the list
        len--;

        return OK;
}
```

Retrieving a value involves getting a pointer to the element in question using *searchN* and then extracting its value.

```
// retrieve the value of element N
int List::retrieveN(char &c, int n) {
        // check bounds
        if((n < 1) || (n > len))
                return OUT_OF_BOUNDS;

        // get a pointer to the element
        Element *pos = searchN(n);

        // assign its value and return
        c = pos->val;
        return OK;
}
```

The destructor needs to free all of the memory that has been allocated for the list. This means going down the list and element at a time freeing each element. Note that it is important to get the address of the next element in the list BEFORE deleting the current one.

```
// destructor - free all memory
List::~List() {
        Element *p = first;        // current element
        Element *temp;                // temporary

        // examine elements one at a time
        while(p != 0) {
                // remember old element
                temp = p;
                // find next element
                p = p->next;
                // delete old element
                delete temp;
        }
}
```

As with the *String* class in the preceding chapter, we need to overload the assignment operator (otherwise *memberwise copying* would result in different lists sharing the same memory). First, we go

down the list deleting each of the old elements. Then we reset the list to be empty. Finally we go down the list to be copied, an element at a time, inserting each value. Notice how easy this is once we have defined the *insertN* method.

```
// overloaded assignment operator
void List::operator=(List& l) {
        Element *p = first, *temp;

        // first destroy the current contents of this list
        while( p != 0) {
                temp = p;
                p = p->next;
                delete temp;
        }

        // now reset the list to be empty
        len = 0;
        first = 0;

        // now insert the elements of the given list one at a time
        for(p =  l.first; p != 0; p = p->next)
                insertN(p->val, len+1);
}
```

The copy constructor is very like the assignment operator except that we don't need to delete any existing elements first.

```
// copy constructor
List::List(List& l) {
        Element *p;

        // initialise the list to be empty
        len = 0;
        first = 0;

        // now insert the elements of the given list one at a time
        for(p =  l.first; p != 0; p = p->next)
                insertN(p->val, len+1);
}
```

Concatenation involves the following:

- Build a result list, initially empty;
- Go down the first list inserting the elements one at a time;
- Go down the second list, inserting the elements one at a time.

```
// concatenate two lists and return result
List List::operator+(List& l) {
        // initialise results variable
        List res;

        // insert the elements of the current list into results
        Element *pos = first;
        while(pos != 0) {
                res.insertN(pos->val, res.len + 1);
                pos = pos->next;
        }

        // insert all the elements of the given list 'l'
        pos = l.first;
        while(pos != 0) {
                res.insertN(pos->val, res.len + 1);
```

```
                        pos = pos->next;
                }

                // return results (calls overloaded assignment)
                return res;
        }
```

The empty list test is trivial.

```
// tests for an empty list
int List::operator!() {
        return !len;
}
```

In order to test equality we first compare the lengths of the lists and, if they are equal, then compare each element in turn.

```
// test equality of lists
int List::operator==(List& l) {
        // cursors to move down each list
        Element *pos1, *pos2;

        // are the lists the same length?
        if (len != l.len)
                return 0;

        // move down both lists an element at a time, comparing values
        pos1 = first;
        pos2 = l.first;
        while(pos1 != 0) {
                // compare elements
                if (pos1->val != pos2->val)
                        return 0;
                // increment position in both lists
                pos1 = pos1->next;
                pos2 = pos2->next;
        }

        // if we get here they must be the same
        return 1;
}

// test inequality of lists
int List::operator!=(List& l) {
        // cursors to move down each list
        Element *pos1, *pos2;

        // are the lists the same length?
        if (len != l.len)
                return 1;

        // move down both lists an element at a time, comparing values
        pos1 = first;
        pos2 = l.first;
        while(pos1 != 0) {
                // compare elements
                if (pos1->val != pos2->val)
                        return 1;
                // increment position in both lists
                pos1 = pos1->next;
                pos2 = pos2->next;
        }
```

```
        // if we get here they must be the same
        return 0;
}
```

The following implements the remaining List methods.

```
// return the length of the list
int List::length() {
        return len;
}

// print a list on the standard output
ostream& operator<<(ostream& os, List& l) {

        // go down the list one element at a time
        Element *pos = l.first;
        while (pos != 0) {
                cout << " [" << pos->val << "] ->";
                pos = pos->next;
        }

        cout << "oo\n";;
        return os;

}

// read in a list from the keyboard and append to current list.
// stop at end of line
istream& operator>>(istream& is, List& l) {
        char data;

        // read and insert chars until end of line or no more space
        while(1) {
                // read a single character into data
                cin.get(data);

                // check for end of line
                if(data == '\n')
                        return is;

                // append to the end of the list
                l.insertN(data, l.len+1);
        }
}
```

To finish this chapter, the following user program demonstrates the syntax for using the List class.

```
// program to test the list implementation
#include "List.h"

main() {
        List l, m ,n;
        char c, data;
        int pos;

        cout << "List testing program\n";

        // input initial list
        cout << "Please enter a list\n";
        cin >> l;
        cout << l;

        // test not operator
        if(!l)
```

```
                cout << "l is empty\n";
        else
                cout << "l is not empty\n";

        // test length operator
        cout << "length of l is " << l.length() << "\n";

        // test insert
        c = 'y';
        while(c == 'y') {
                cout << "enter value to be inserted\n";
                cin >> data;
                cout << "enter position\n";
                cin >> pos;
                if(l.insertN(data, pos) < 0)
                        cout << "error\n";
                else
                        cout << l << "\n";
                cout << "another? (y/n): ";
                cin >> c;
        }

        // test delete
        c = 'y';
        while(c == 'y') {
                cout << "enter position to be deleted\n";
                cin >> pos;
                if(l.deleteN(pos) < 0)
                        cout << "error\n";
                else
                        cout << l << "\n";
                cout << "another? (y/n): ";;
                cin >> c;
        }

        // flush line
        cin.get(c);

        // read a second list
        cout << "Please enter second list\n";
        cin >> m;
        cout << m;

        // test comparison operators
        if(l == m)
                cout << "lists are equal\n";
        else
                cout << "lists are not equal\n";

        if(l != m)
                cout << "lists are not equal\n";
        else
                cout << "lists are equal\n";

        // test concatenate operator
        n = l + m;
        cout << "first + second is\n" << n << "\n";
}
```

Chapter 8: The Line Editor - A Final Example

8.1. Introduction

We finish the course with a final example which pulls together many of the concepts we have seen in recent chapters. The example is an implementation of a *Line Editor* which can be used to edit a specific file a line at a time. At first sight this might seem a challenging prospect. The chapter aims to show that, given suitable general purpose classes, the task is in fact relatively easy. This should serve as a demonstration of the power of *Abstract Data Typing* as an approach to programming and should highlight the issues of encapsulation and re-use of code.

However, in to order to build our Line editor we do need to briefly digress for a moment to consider a final new topic - the issue of *command line arguments* to C++ programs.

8.2. Command line arguments

You should be aware that most operating system commands may be given "command line arguments" when they are called. For example, the UNIX *cat* command may take the names of files to be displayed as its arguments:

```
cat myprog.C notes.txt
```

Two common uses of command line arguments are to specify filenames which commands will then access or to give flags to commands in order to modify their behaviour in some way. For example, the *ls* command can be instructed to give extra information about files by using the *-l* flag:

```
ls -l
```

On this course we have learned how to write, compile and then run our own C++ programs from within UNIX. Running a program simply involves entering the name of the file which contains its executable binary code. It would be very useful if our own programs could be given command line arguments whenever they were run in just the same way as normal UNIX commands and if these arguments could then be used inside the program.

This is possible in C++ by declaring special arguments for the function called *main()*. If you remember, main is the function where execution of a C++ program begins. It turns out that, like any other function, main can be declared with arguments. These will contain the particular command line arguments that were given to UNIX when the program was first invoked. If you want to use these arguments, you must declare main in the following way.

```
main(int argc, char **argv) {
        // program code
}
```

We are using two arguments to main, *argc* and *argv* (in fact, there can be a third which we don't need to consider here). Argc is an integer which will contain the number of command line arguments that were given to the program. You need to be a little careful here because the name of the program is ALWAYS the first argument and so argc is always >= 1. For example, if our executable is in the file *a.out* and the user enters the command *a.out -q fred jim* at the UNIX shell then argc would have the initial value 4 at the start of function main.

The second argument, *argv*, is a pointer to a pointer to a char.

```
char **argv;
```

This double pointer is no mistake. Remembering that a pointer can represent an array, you should think of argv as pointing to the beginning of an array, each of whose elements itself points at an array of characters. In other words we have an array of character strings!

This array will contain each of the arguments given to the program in the order the were given. The first string will always be the name of the program itself. Thus, for the above example, argv would point to the following.

Remembering that we can use the [] operator with pointers, we arrive at the following.

- *argv[0]* is the first string (i.e. the name of the program).
- *argv[1]* is the first command line argument.
- *argv[2]* is the second command line argument.

And so on.

Extending this idea, *argv[3][1]* is the second character in the third argument (i.e. the single letter 'i').

The following program uses argc and argv to implement the UNIX *echo* command in C++.

```
// echo command line arguments
// Steve Benford - November 1991
#include <stream.h>

main(int argc, char **argv) {


        // echo arguments that were given
        for(int i = 1; i < argc; i++)
                cout << argv[i] << " ";
        cout << "\n";
}
```

8.3. Designing the line editor

The goal of our line editor is to allow us to edit a UNIX file. However, unlike *emacs* which is a "screen editor", our line editor will only let us change specific lines at one time by explicitly giving their line numbers (e.g. *delete line 6*). This kind of limited editor can still be useful in some circumstances and UNIX contains a well known example called *ed*.

Our simple line editor will support the following:

- edit a named file;
- Insert new text after the specified line number;
- Display all lines between two line numbers;
- Delete all lines between two line numbers;
- Copy lines between two line numbers to a different position.
- Save the file at any time;
- Quit the editor;

Our solution will involve two steps. First, design an *Editor* Abstract Data Type which supports these functions in an abstract way and then implement this as a C++ class. Second, implement an interface built on top of this ADT. The advantage of separating the interface from the rest of the implementation is that we can easily alter the interface at a later time - perhaps even replacing it with a screen interface!

The following is the C++ class definition for the *Editor* class.

```
class Editor {
    private:
        int changed;
        List buffer;
        char* filename;

    public:
        Editor(char* filename);
        ~Editor();
        int display(int m, int n);
        int insert(int n);


        int remove(int m, int n);
        int copy(int m, int n, int o);
        int save();
        int quit();
};
```

This class uses the new class *List* which implements a linked list of Strings. Although we haven't seen the code for this it is the same at the linked list of characters from chapter 7 except that the value stored in each list element is a *String* not a *char*.

The constructor function takes the name of the file to be edited (a char*). Its job is to open the file and to read its contents into an edit buffer. *Display* shows the text between lines *n* and *m*. *Remove* deletes the text between lines *n* and *m*. *Insert* starts inserting new text after line *n*. It stops when a full stop is entered at the beginning of a line, on its own. *Copy* copies the lines between *m* and *n* and inserts them after line *o*. *Save* saves the current buffer back to the file. *Quit* quits the editor. If the changes have been made since the last save, it asks the user if they want to save to the file.

8.4. Implementing the Line Editor

The Editor uses three internal data members:

> **List buffer** - a linked list of Strings containing the text being edited (each String represents a separate line).
>
> **char* filename** - the name of the file being edited.
>
> **int changed** - an indication of whether the buffer had been changed since the file was last saved (the value 1 indicates that it has and 0 that it has not).

The overall approach towards implementation is the following.

1. The constructor opens the filename given as its argument, reads it a line at a time and builds up the linked list of strings in *buffer*.
2. Operations to edit the text alter the contents of *buffer*. This is easy given the available linked list methods.
3. Saving a file writes the contents of the buffer back to the file a line at a time, overwriting the old contents.

We now look at the code for each Editor method in turn.

Notice how the constructor has to allocate new memory to make a permanent copy of the filename. In addition, it has to check that the file exists and is readable. Finally, it has to note that the file doesn't need saving.

```
// initialise editor, open and load file
Editor::Editor(char* infile) {

        // make a permanent copy of the filename
```

```
        int len = strlen(infile);
        filename = new char[len+1];
        strcpy(filename, infile);


        istream ifile(filename, "r");

        if(!ifile.is_open()) {
                cerr << "couldn't open file: " << filename << "\n";
                exit(-1);
        }

        // String object for reading from file
        String line;
        int n = 1;

        // load file a line at a time
        while(ifile >> line)
                if(!buffer.insertN(line, n++)) {
                        cerr << "error building buffer" << "\n";
                        exit(-1);
                }

        // set indication that file hasn't been updated
        changed = 0;

        // close the file and delete the file object
        ifile.close();
}
```

The destructor needs to free the memory that was allocated for the permanent copy of the filename.

```
// tidy up at finish
Editor::~Editor() {

        // delete the space allocated for the filename
        delete filename;
}
```

Display simply retrieves the specified range of elements from the *buffer* linked list. It displays all lines found and returns a status code indicating if the specified range was out of bounds.

```
// display lines start to finish
int Editor::display(int start, int finish) {
        String line;
        for(int i = start; i <= finish; i++)
                if(buffer.retrieveN(line, i))
                        cout << i << '\t' << line << "\n";
                else
                        return 0;
        return 1;
}
```

Insert reads new lines from the input and inserts them into the buffer. It also has to note that the buffer now differs from the file.

```
// insert text at line start with "." on a line of its own
int Editor::insert(int start) {
        String line;
        String end(".");

        while(1) {
```

```
                cout << "> ";
                cin >> line;
                if(line == end) {
                        changed = 1;
                        return 1;
                }
                if(!buffer.insertN(line, start++))
                        return 0;
        }
}
```

Delete simply calls the *deleteN* function on the list.

```
// delete lines start to finish
int Editor::remove(int start, int finish) {
        for (int i = start; i <= finish; i++)
                if(!buffer.deleteN(start))
                        return 0;
        // note that file is changed
        changed = 1;
        return 1;
}
```

Copy is also straight forward.

```
int Editor::copy(int start, int finish, int destination) {
        String line;

     // first, check that the destination is not within the range
     if ( ( start <= destination ) && ( destination <= finish ) )
          return 0;

     // OK, do the copying
        for (int i = start; i <= finish; i++)
                if(!buffer.retrieveN(line,i))
                        return 0;
                else if(!buffer.insertN(line, destination++))
                        return 0;
        // note that file is changed
        changed = 1;
        return 1;
}
```

The *save* method indicates that the buffer is being written back to the file and then opens the file in write mode. Each line of *buffer* is then written to the file and the data member *changed* set to the value 0.

```
// save the current buffer contents back to the file
int Editor::save() {
        cout << "saving file: " << filename << "\n";

        // open file in write mode
        ostream ofile(filename, "w");

        // check if file was opened OK
        if(!ofile.is_open() )  {
                cerr << "save: file " << filename << " not opened\n";
                return 0;
        }

        // write contents of file
        int len = buffer.length();
        String s;
```

```
        // retrieve each line at a time and save it
        for(int i = 1; i <= len; i++)
                if(!buffer.retrieveN(s, i)) {
                        cerr << "save: error in writing line " << i << "\n";
                        return 0;
                }
                else
                        ofile << s << "\n";

        // close file and delete associated file object
        ofile.close();

        // note that file no longer needs saving
        changed  = 0;

        return 1;
}
```

Finally, *quit* only needs to check whether the file needs saving.

```
// quit editor - check if file needs saving
int Editor::quit() {
        char c;
        if (changed) {
                cout << "File has been altered - do you want to save? (y/n)\n";
                cin >> c;
                if(c == 'y')
                        save();
        }
        return 1;
}
```

8.5. The Line Editor Interface

This section outlines one possible interface to the Editor class. The syntax of its commands is loosely based on the UNIX *ed* line editor.

Assuming that the executable program is in the file *edit*, the user starts the editor by entering the UNIX command:

```
edit <filename>
```

They then see a *$* prompt indicating that the editor is ready to receive commands. Each command is represented by a single letter followed by appropriate line numbers. A summary of the syntax is:

```
d m n    (display lines numbered m to n)
r m n    (remove lines)
c m n o  (copy lines)
s        (save)
q        (quit)
?        (help - display the menu of commands)
i m      (insert - then type text ending with a '.' on
          a line of its own).
```

The first job of the interface program is to process the command line arguments to get the filename to be edited. Notice that an error message is generated if no filename is supplied. Beyond this, the program enters a continuous while loop, processing commands through a switch statement until the user selects the *quit* option.

```
// line editor interface
#include <stdlib.h>
```

```
#include "Editor.h"

main(int argc, char **argv) {
        char command, ch;
        int start, finish, dest;

        // process command arguments
        if(argc != 2) {
                cerr << "usage: edit filename\n";
                exit(-1);
        }

        // initialise editor object
        Editor ed(argv[1]);
        cout << "file successfully loaded\n";

        while(1) {
                cout << "$ ";
                cin >> command;

                if(command == 'd') {
                        cin >> start;
                        cin >> finish;

                        if(!ed.display(start, finish))
                                cout << "bad line number\n";
                }

                else if (command == 'i') {
                        cin >> dest;
                        cin.get(ch);
                        if(!ed.insert(dest))
                                cout << "bad line number\n";
                }

                else if (command == 'r') {
                        cin >> start;
                        cin >> finish;
                        if(!ed.remove(start, finish))
                                cout << "bad line number\n";
                }

                else if (command == 'c') {
                        cin >> start;
                        cin >> finish;
                        cin >> dest;
                        if(!ed.copy(start, finish, dest))
                                cout << "bad line number\n";
                }

                else if (command == 's') {
                        if(!ed.save())
                                cout << "error saving file\n";
                }

                else if (command == '?') {
                        cout << "d m n    display lines m through n\n";
                        cout << "i n      insert text at line n\n";
                        cout << "r m n    remove lines m through n\n";
                        cout << "c m n o  copy lines m through n to o\n";
                        cout << "s        save file\n";
                        cout << "?        display this message\n";
                        cout << "q        quit\n";
                }
```

```
                else if (command == 'q') {
                        if(!ed.quit())
                                cout << "error closing down editor\n";
                        exit(1);
                }

                else
                        cout << "syntax error\n";

        } // while

} // main
```