Introduction to C++ Programming II

Ian Aitchison and Peter King

November 1997
## Contents

1 More on Data Types ........................................ 5
   1.1 Integer types ........................................ 5
   1.2 Floating point types ................................ 8
   1.3 Enumerated types .................................. 8
   1.4 Summary .............................................. 11
   1.5 Review Questions .................................. 11
   1.6 Exercises ............................................ 11

2 More on Characters and Strings .......................... 13
   2.1 Character Input and Output .......................... 13
      2.1.1 Example Program: Counting characters .......... 15
      2.1.2 Example Program: Counting words ............... 16
      2.1.3 Character Functions ............................ 18
   2.2 The String Library .................................. 19
      2.2.1 Setting up a String type using `typedef` ....... 20
      2.2.2 Example: Printing an enumerated variable ....... 21
      2.2.3 Example: Entering an enumerated value ......... 22
      2.2.4 Putting it all together .......................... 24
   2.3 Summary .............................................. 26
   2.4 Review Questions .................................. 26
   2.5 Exercises ............................................ 27

3 Multiple Source File Programs ........................... 29
   3.1 Multiple Source Files ............................... 29
   3.2 Compiling Multiple File Programs ................... 32
   3.3 Practical Considerations ............................. 33
   3.4 Scope ................................................. 34
   3.5 External Variables .................................. 36
   3.6 `static` Variables ................................. 37
   3.7 Summary .............................................. 37
   3.8 Review Questions .................................. 38
   3.9 Exercises ............................................ 38
### 4 Two-dimensional Arrays

- 4.1 Examples
- 4.2 Initialisation of two-dimensional arrays
- 4.3 Two-dimensional arrays as function parameters
- 4.4 Example: Mark Processing
- 4.5 Summary
- 4.6 Review Questions
- 4.7 Exercises

### 5 The `struct` data type

- 5.1 Examples
- 5.2 Initialising a `struct`
- 5.3 A `struct` as a function parameter
- 5.4 Example: Mark Processing re-visited
- 5.5 A data type for financial calculations
- 5.6 Summary
- 5.7 Review Questions
- 5.8 Exercises

### 6 Tables and Table Searching

- 6.1 The Search Process
  - 6.1.1 Linear Search
- 6.2 Improving the performance - Binary Search
- 6.3 Insertion and Deletion
  - 6.3.1 Insertion
  - 6.3.2 Deletion
- 6.4 Conclusions
- 6.5 Summary
- 6.6 Review Questions
- 6.7 Exercises

### 7 Sorting tables

- 7.1 Simple Algorithms
  - 7.1.1 Selection Sort
  - 7.1.2 Exchange Sort
  - 7.1.3 Insertion Sort
- 7.2 An Advanced algorithm - Quicksort
- 7.3 Conclusions
- 7.4 Summary
- 7.5 Review Questions
- 7.6 Exercises
13 Object-oriented Design
13.1 An example ........................................... 139
13.2 Conclusion ........................................... 149
13.3 Summary ........................................... 150
13.4 Review Questions ..................................... 150
13.5 Exercises ........................................... 151

14 Debugging ........................................... 152
14.1 Compile Time Errors ................................ 152
14.2 Run Time Errors ...................................... 153
14.3 Logical Errors ...................................... 154
14.4 Example: Calculating interest ....................... 155
14.5 Using const as an aid in Debugging ............... 157
14.6 Defensive Programming .............................. 157
14.7 Testing ............................................ 158
About the course

Welcome to Heriot-Watt University *Introduction to Computing II* course. This course is being provided for students taking the Certificate in Science of Aberdeen and Heriot-Watt Universities. It is being delivered by way of the World Wide Web, however all students will also receive the entire course in a printed format.

The course is arranged in a series of Lessons. As well as new material to learn, each Lesson contains review questions, multiple choice questions, and programming exercises. The review questions are designed to get you to think about the contents of the Lesson. These review questions should be straightforward; if the answer is not obvious, reread the Lesson. The programming exercises are an essential part of the course. Computer programming is a practical subject, without practice no progress will be made.

There will be some assessed coursework, this may be submitted by electronic mail. Electronic mail will also be used to provide help with problems.

Course contents

This course is intended as a continuation in C++ programming to follow on from the course Introduction to Computing I. It is assumed that the student has already done that course. It covers more advance features of C++, in particular it provides an introduction to the object-oriented features of C++.

The lessons constituting the course cover topics as follows:

1. Lesson 1 - Integer types. Floating point types. Enumerated types.
4. Lesson 4 - Two-dimensional arrays. Initialisation of two-dimensional arrays. Two-dimensional arrays as parameters of functions.


9. Lesson 9 - Procedural and Data Abstraction. Definition and implementation of a simple class in C++.

10. Lesson 10 - Dynamic allocation of Arrays and Objects.

11. Lesson 11 - Further classes. The implementation of a string class. Destructors, inspection methods and mutators.

12. Lesson 12 - Operator overloading. Overloading relational operators in the string class. Setting up safe array access operators.


14. Lesson 14 - Debugging techniques.

Assessment

There will be two class assignments to carry out. One will be given out which requires that you know the material up to about Lesson 4 or 5. The other will require that you know the material up to and including near the end of the course. The first assignment will be worth 10% in the final assessment and the second assignment will be worth 15% in the final assessment. The remaining 75% will come from the class examination.

Further information will be available on the World Wide Web version of the course.

Accessing the course on the World Wide Web

The course, and its successor Introduction to Computing II, are available on the World Wide Web (WWW). You are assumed to know how to connect to WWW. The course has been developed using Netscape as the browser, but it should be possible to access it equally well with Microsoft’s Internet Navigator, or with Mosaic. To access the course, use your browser to open the URL
http://www.cee.hw.ac.uk/

This will present you with a page about the Department of Computing and Electrical Engineering at Heriot-Watt University. Follow the link labelled Distance Learning Courses

The page loaded will give general information about the courses. There will also be reminders about approaching deadlines for coursework, and notification of any corrections to parts of the course. (Any corrections will have been made in the WWW version of the course, but this will enable you to annotate your printed version of the notes.) You are advised to always enter the course in this fashion. Saving bookmarks with your browser may cause problems if any corrections cause extra sections or questions to be added to the course.

Coursework should be submitted using electronic mail. Send any files requested to pjbk@cee.hw.ac.uk, with clear indications of which assignment is being submitted within the file.

Getting Help

Although the notes are intended to be comprehensive, there will undoubtedly be times when you need help. There are two mechanisms for this. An electronic mailing list has been set up at Heriot-Watt University which copies messages submitted to it to all members of the course. The course organisers and teachers receive all messages sent to this list. All students should join this mailing list, and use it to replace the discussion that would normally take place in a classroom. In order to join the mailing list, send an electronic mail message to majordomo@cee.hw.ac.uk with the following contents:

subscribe pathways

Once you have joined the list, messages to the list are sent as ordinary electronic mail addressed to pathways@cee.hw.ac.uk

If your problems are of a more individual nature, you may prefer to send them to the course teacher only, in which case an email message should be addressed to pjbk@cee.hw.ac.uk
Lesson 1

More on Data Types

In the course Introduction to Computing I only the types int, float and char were considered. C++ has various other data types which are now considered.

1.1 Integer types

Integer variables are numeric values which can be represented exactly on the computer. Operations between integer types also give exact results. The only integer type that has been encountered so far is int. The C++ standard does not actually define what range of values the type int should be capable of representing. This decision is left to the C++ implementation. In Borland C++ the type int is stored in two bytes and hence has a range of -32768 to 32767. If the values are expected to be outside this range, for example if you are calculating the number of seconds in a year, then the variable will need to be declared as a long int. This allocates four bytes to store the variable and extends the allowable range of values to -2147483648 to 2147483647.

If it is known that an integer variable is always going to take positive values then the prefix unsigned can be used in front of int. This allows a larger range of positive integers to be represented. The prefix short can also be used, however in Borland C++ short int is the same as int. In some implementations int and long int are four bytes and short int is two bytes and it may be worthwhile to use short int rather than int to save space if the larger range of values is not required.

These types are summarised in the following table which applies to Borland C++:
<table>
<thead>
<tr>
<th>Type</th>
<th>Storage Space</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>2 bytes</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>short int</td>
<td>2 bytes</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>long int</td>
<td>4 bytes</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>2 bytes</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>2 bytes</td>
<td>0 to 65535</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>4 bytes</td>
<td>0 to 4294967295</td>
</tr>
</tbody>
</table>

When int has a prefix then the word int may be missed out. For example the type long is the same type as long int.

While C++ does not specify the storage space to be allocated to these types it does require that short int should be at least two bytes and that long int should be at least 4 bytes. In addition short int must take no more space than int which in turn must take no more space than long int.

The file limits.h associated with a C++ compiler defines constants which give the ranges for each type of integer in that implementation. For example the number of bits used to represent the type int is given by a constant WORD_BIT, which is set to 16 in the Borland C++ compiler. This value could be used to allow a program to avoid producing erroneous results caused by integer overflow. For example consider the following function which evaluates $2^n$ and tests that the value of $n$ cannot produce an integer overflow. The function return value is used to indicate success or failure of the operation and in the case of success the value is returned via a reference parameter.

```c++
#include <limits.h>
int power2(int n, int& power)
// Returns True if n is such that two
// to the power n is within the range
// of the type int in the implementation
// then the function returns true and
// returns the result in power. Else
// the function returns false.
// The largest power of two that can
// be represented in the type int is
// two to the power WORD_BIT-2.
{
    int prod = 1, i;
    if (n < WORD_BIT-1 && n >= 0)
    {
        for (i=0; i<n; i++) prod *= 2;
        power = prod;
        return 1;
    }
```
else
    return 0;
}

power.cpp

Integer constants are normally held as being of type int. However if an integer constant is outside the range of int then it will be held as a long integer. Long integer constants can also be written with an L appended. For example 1234567L represents the number 1,234,567.

It is worth noting that unsigned int quantities obey the rules of arithmetic to the modulo $2^n$ where $n$ is the number of bits used in the representation. This means that if 16 bits are used in the representation then an unsigned int has the range $0–2^{16} – 1$ (65535). Thus all positive values are represented by the remainder when the value is divided by 65536 and to represent a negative value multiples of 65536 are added until a value between 0 and 65535 is obtained. Thus 65530 + 10 added as unsigned values gives 4 as a result. This means that great care must be taken when working with unsigned values.

The following piece of C++ when run on a compiler that uses 2 bytes to represent an int

```cpp
int u = 10;
int v = -17;
int sum;
unsigned int usum;
sum = u + v;
usum = u + v;
cout << "the int sum of 10 and -17 is "
   << sum << endl;
cout << "the unsigned int sum of 10 and -17 is "
   << usum << endl
```

would output:

```
the int sum of 10 and -17 is -7
the unsigned int sum of 10 and -17 is 65529
```

When -7 is assigned to an unsigned int then 65536 is added to it to bring it into the range 0–65535, giving the result 65529.

Using unsigned quantities can very easily lead to errors in programs unless they are never subtracted from one another, and one can guarantee that their magnitude will stay in range. Their use is not recommended.

Integer overflow occurs when an integer takes a value outside the appropriate limits. A compiler will usually detect any integer overflow at compile
time when using integer constants. However at run-time integer overflow will not be reported and the consequences are unpredictable. Processing may continue with the integer truncated to fit or the program may exit. Portability problems may also occur with programs which are written for a compiler that uses 4 bytes for the type int when transferred to a compiler that only uses 2 bytes for an int. These problems will only occur if values larger than 32767 are used. If this is so and portability is a concern then it is probably best to use the type long int instead.

1.2 Floating point types

Floating point numbers are held in the computer in a form which allows a much larger range of values to be expressed, but not all of these values can be represented exactly. As well as the type float the type double can be used. double uses more storage space, but it also allows more accuracy. The approximate range of values and precision are noted in the table below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Storage Space</th>
<th>Range</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>-1E+38 to 1E+38</td>
<td>7 decimal digits</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>-1E+308 to 1E+308</td>
<td>17 decimal digits</td>
</tr>
</tbody>
</table>

Various parameters that define properties of floating point arithmetic in the C++ implementation can be found in the include file float.h.

1.3 Enumerated types

It has already been shown how the declaration of constants can make programs more readable. For example in a program that dealt with the days of the week constants could be declared as follows:

    const int SUN = 0,
                MON = 1,
                ........
                SAT = 6;

The identifiers SUN, ...., SAT can now represent the days of the week instead of using integers 0 to 6. An alternative way of introducing named constants such as the above is to use an enumerated type. An enumerated type allows the association of named constants with the integers 0, 1, 2, 3.... Enumerated types are declared by the keyword enum, followed by the constant identifiers in order. Thus the set of constants above could be declared as follows:

    enum {SUN, MON, TUE, WED, THU, FRI, SAT};
This declaration declares a set of seven enumerated constants and associates them with the integers 0 to 6. Zero is associated with the first of the enumerated constants in the list and the succeeding constants are associated in turn with the integers 1, 2, 3, ... etc.

This gives a slightly improved way of defining a set of constants, however the real power of the enum type comes when an enumeration tag name is associated with the list of enumerated constants. This allows the creation of a new data type which is identified by the enumeration tag name. For example a new type Day is created as follows:

```c
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
```

The type Day can now be used in declarations of variables in the same manner as the pre-defined data types such as int and float. For example variables d1 and d2 of type Day could be declared as follows:

```c
Day d1, d2;
```

Just as a variable declared as being of type int can take any integer value so a variable declared as being of type Day can take as its value any of the enumerated constants which belong to the enumerated list associated with the type Day, i.e. SUN, MON, TUE etc. Values of the enumerated type can then be assigned to the identifiers d1 or d2, they can be tested for equality with each other or with the enumerated constants etc. For example:

```c
Day d1, d2;

d1 = MON;

if (d1 == d2) ..........

if (d2 == SAT || d2 == SUN) ........
```

The values assigned to the enumerated constants are, by default, the integers 0, 1, 2, 3, ... . These default values can be changed by assigning a constant (integer) value to each element in the list. For example:

```c
enum Direction {NORTH, WEST = 90, SOUTH = 180, EAST = 270};
```

where the enumerated constant NORTH has the default value 0, while the other enumerated constants have the values as assigned to them.

When a value from an enumerated type is used the compiler converts it to type int. This means that arithmetic may be carried out on enumerated types. For example to find the day of the week that is n days later than the current day we could use:

```c
12
```
Day dn, today;
int n;

dn = Day((int(today)+n) % 7);

Note that we have cast the variable today to type int and then cast the integer result of the expression back to the type Day. The cast of today to int is required since the types Day and int are distinct types. Most C++ compilers would give a warning message if this cast was missed out. In general it is good practice when assigning a value of one type to another to cast the assigned value to the type of the variable to which it is assigned, as was done above by casting the result of the computation above to the type Day.

In this example the method of calculation ensures that the value assigned is in the required range and can be cast without loss of information. If the value was not a valid value of the enumerated type then casting does not make it a valid value and the effect of the assignment is implementation dependent. In most cases it is not good practice to perform arithmetic on enumerated types.

The input and output of enumerated types is not handled automatically by C++. Input of enumerated type values is difficult and consideration of its problems is postponed until section 2.2.3 A variable of type Day as defined above could be output by the following function:

```c++
void printday(Day d)
    // Prints the day of the week d in English form
    {
        switch (d)
        {
            case SUN : cout << "SUN";
                    break;
            case MON : cout << "MON";
                    break;
            case TUE : cout << "TUE";
                    break;
            case WED : cout << "WED";
                    break;
            case THU : cout << "THU";
                    break;
            case FRI : cout << "FRI";
                    break;
            case SAT : cout << "SAT";
                    break;
            default : cout << "Illegal value";
        }
    }
```
1.4 Summary

- Whole number valued variables can be specified as `short`, `int`, or `long`. The specification affects the amount of storage reserved for the variable and the range of values it can represent.
- Real valued variables can be specified as `float` or `double`, modifying the accuracy to which they are held, the range of values they can represent and the amount of storage required.
- Variable which are only to take on a limited number of values should be declared as enumerated types. If an enumeration tag is given, it can be used as a type for declaring new variables.

1.5 Review Questions

1. Choose a variable name to represent the following quantities. In all cases, consider which is the most appropriate type for the variable.

   (a) military rank of a soldier
   (b) state of a traffic light
   (c) distance to the moon
   (d) population of the earth in 1900 (800,000,000)
   (e) population of the earth as it evolves from now until 3000 (current population c. 4,500,000,000)

2. What is wrong with the following declarations?

   ```
   enum day {sun,mon,tue,wed,thu,fri,sat};
   enum week_day {mon,tue,wed,thu,fri};
   enum week_end {sat,sun};
   ```

1.6 Exercises

1. Investigate how your computer deals with overflow in programs. Using integer types and the values found in `limits.h` investigate the effects of generating a number which is outside the valid range by both adding numbers to create the overflow, and by multiplying numbers. Similar experiments can be performed with floating point numbers using the values in the `float.h` file.
2. Write a program to evaluate Fibonacci numbers $F_n$. They are defined as $F_n = F_{n-1} + F_{n-2}$ with $F_0 = F_1 = 1$. The first few numbers of the series are

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

How many terms of the series are needed before using `int` variables gives insufficient range?

3. In section 16.6 of Introduction to Computing 1 a program was written to return the square root of a `float` variable. Using this program as a guide write your own square root function which takes a `double` parameter and returns a `double` result. Test it against the supplied square root function by producing a table of the square roots of the numbers 1 to 10, printing the results to 16 decimal places.
Lesson 2

More on Characters and Strings

Character variables and Strings were introduced in Introduction to Computing I. In this Lesson further features of these data types are covered.

2.1 Character Input and Output

Most of the input and output so far considered has been of numbers, both integers and floating-point. Of course numbers are input or output in character form and are converted from a string of characters to the internal binary form, and vice versa, automatically. There are situations in which this is not desirable and it is necessary to handle characters directly, for example in text-processing applications.

It has already been noted that the stream `cin` can be used to input single characters. However when entering a character all space characters (and newline characters) are skipped until a non-space character is encountered. For example:

```cpp
char ch;
cout << "Enter a single character: ";
cin >> ch;
cout << "The input character was " << ch
    << endl;
```

would output

```
The input character was a
```

if the user typed in ‘a’ or ‘ a’. Sometimes spaces are important in an application and therefore the stream `cin` cannot be used directly for the input of characters in those situations.
The `get` function can be used to return the next character from the input, regardless of whether it is a space or not. `get` is a **member function** of an input stream. It takes a single `char` variable as its parameter, and returns the character read in that variable.

For example the following statement would obtain the next character from the input stream `cin` and store it in the character variable `ch`.

```cpp
cin.get(ch);
```

The general form of a call of the `get` member function of a stream `in_stream` is:

```cpp
in_stream.get(char_variable);
```

which would obtain the next character from the input stream `in_stream` and assign it to the character variable `char_variable`. If `in_stream` is not `cin` but another `ifstream` variable, then a file must have been associated with the stream using the `open` member function of the stream.

As well as reading the normal input characters such as letters, digits, punctuation symbols etc. the `get` function will also read space characters, newline characters etc. Character constants are written enclosed in single quotes in C++. For example the character `a` would be written as `'a'`. Special characters such as `newline`, `horizontal tab` etc. are represented using escape sequences. An escape sequence consists of the backslash character \ followed by a single character. Even though two symbols are used the two characters together represent a single character. The special characters allowed in C++ are:

```
newline   \n
horizontal tab \t
vertical tab \v
carriage return \r
formfeed \f
alarm bell \a
single quote \'
double quote \"
backslash \\
question mark \?
```

These characters can be used in strings. For example `\n` can be used as part of a string as follows:

```cpp
cout << "Now take a new line\n";
```

which has the same effect as:

```cpp
cout << "Now take a new line" << endl;
```
The following example outputs a string surrounded by double quotes:

```
cout << "Output a "quoted string"
"\n";
```

which outputs:

```
Output a "quoted string"
```

### 2.1.1 Example Program: Counting characters

This program illustrates that the `get` function for input returns every character entered, including spaces and newline characters etc. A line of input is read and each character is placed into a character array. This character array is converted into a string by appending the null character after the last character. The end of the input line is recognised by testing for the newline symbol `\n`. Whatever their internal representation for the end of line, most C++ systems convert it to `\n` for input and output.

```
// IEA November 96
// Program enters characters from keyboard and
// places them in a string, then outputs the
// string together with a count of the number of
// characters in the string

#include <iostream.h>

void main()
{
    char c;
    int count = 0;        // character count
    char string[81];     // string variable
    cout << "Enter a string terminated by return: "
        << endl;
    cin.get(c);           // get character from input
    while (c != '\n')    // test for return
    {
        string[count] = c;  // save character in string
        count++;            // increment index
        cin.get(c);
    }
    string[count] = '\0'; // terminate string
    cout << "The input string was "
        << endl << string << endl
        << "and it had " << count
        << " characters"
        << endl;
}
```
charcnt.cpp

2.1.2 Example Program: Counting words

The following program reads a piece of text from a file and prints the individual words in the file, one to a line, together with a count of the number of words and the number of lines found in the file. A word is assumed to only consist of letters of the alphabet, upper or lower case. There are several points to note about this program:

1. The declaration of an input file stream and its association with a file whose name is entered by the user using the `open` member function. This was covered in Lesson 19 of Introduction to Computing I.

2. The use of the `eof()` member function to terminate data entry. This function returns `true` whenever the program has read past the end of the input file.

3. The program uses a function from the C++ Character Function Library, namely the function:

   ```
   int isalpha(char c)
   ```

   which returns `true` if `c` is an alphabetic character, upper or lower case. See Section 2.1.3 for a list of some additional functions in this library.

4. The basic algorithm is

   ```
   while not at end of file do
   {
     skip characters until a letter found or end of file is reached.
     if not end of file reached then
     {
       while character is letter do
       {
         get next character.
         echo character.
       }
       increment word count.
     }
   }
   ```
Note that end of file has to be tested again inside the inner while loop that skips characters to allow for a file which has characters after the last word. Processing only continues to scan the characters of a word after skipping characters between words if end of file has not meanwhile become true.

// IEA November 96
// Program reads a filename from cin and counts the
// number of lines in that file and the number of words.
// A word is considered to only consist of letters of
// the alphabet, any other character is treated as a
// separator or terminator (including newline)
// The words are printed as they are input.

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>

int main()
{
    char c;
    char filename[20];
    int wordcount = 0; // word count
    int linecount = 0; // line count
    ifstream ins;
    cout << "Enter name of input file: ";
    cin >> filename;
    ins.open(filename);
    if (ins.fail())
    {
        cout << "Failure on opening " << filename
             << endl;
        return 1;
    }
    ins.get(c);
    while (!ins.eof())
    {
        while (!isalpha(c) && !ins.eof())
        {
            if (c == '\n') linecount++;
            ins.get(c);
        }
        if (!ins.eof())
        {
            if (c == ' ')
            {
                wordcount++;
                ins.get(c);
            }
        }
    }
}
while (isalpha(c) && !ins.eof())
    {
        cout << c;
        ins.get(c);
    }
    wordcount++;
    cout << endl;
}

cout << endl << "The word count was ")
    << wordcount
    << endl;

cout << endl << "The line count was ")
    << linecount
    << endl;
ins.close();
return 0;
}

wordcnt.cpp

2.1.3 Character Functions
The standard include file <ctype.h> contains the declarations of a set of functions which are useful in carrying out character handling operations. They include:

int isalnum(int c) Returns true if c is an alphanumeric character, i.e. it is a letter of the alphabet (upper or lower case) or one of the digits 0—9. Otherwise returns false.

int isalpha(int c) Returns true if c is an alphabetic character (lower or upper case). Otherwise returns false.

int isdigit(int c) Returns true if c is a decimal digit character (0—9). Otherwise returns false.

int islower(int c) Returns true if c is a lower case alphabetic character. Otherwise returns false.

int isupper(int c) Returns true if c is an upper case alphabetic character. Otherwise returns false.

int isspace(int c) Returns true if c is a whitespace character. Otherwise returns false. The whitespace characters are horizontal tab, newline, space, formfeed, carriage return and vertical tab.
int tolower(int c)  If c is an upper case alphabetic character then tolower returns the corresponding lower case character, otherwise c is returned unchanged.

int toupper(int c)  If c is a lower case alphabetic character then toupper returns the corresponding upper case character, otherwise c is returned unchanged.

2.2 The String Library

Although strings are single data entities no operators are defined on strings. To carry out operations such as assigning a value to a string, comparing two strings for equality, or concatenating two strings together, functions from the string library string.h have to be used.

Some of the functions in this library are

strcpy(dest,src) copies the source string src to the destination string dest. No check is made that dest is actually large enough to accommodate src. All characters up to, and including, the null character are copied from source. The source string is not changed by this function.

strncpy(dest,src,n) copies the first n characters in the source string src to the destination string dest. If a null character is encountered in src before n characters have been copied then sufficient null characters are placed into dest to fill the first n places. If n is larger than the length of the destination then no null character will be copied to dest and possibly other data may be overwritten.

strcmp(s1,s2) compares the two strings s1 and s2 for equality. If s1 < s2 in lexicographic ordering (based on the ASCII character set) then strcmp returns a value less than zero, if s1 > s2 in lexicographic ordering then strcmp returns a value greater than zero. If the strings s1 and s2 are equal then strcmp returns a value of zero.

strlen(s) returns the length of the string s, that is the number of characters in the string not counting the terminating null character.

Some simple examples of these functions follow:

```c
char s1[10], s2[20];
strcpy(s2,"A long string");

strncpy(s1,s2,6);
s1[6] = '\0';
```
which sets s1 to

A l o n g \0yyy

The following function illustrates a simple use of these functions to do safe string copying.

```c
int safecopy(char dest[], int n, const char src[]) {
    // If the string in src can fit into the string dest
    // then it is copied to the string dest which has
    // space for n characters and the value TRUE is
    // returned. If dest is not sufficiently long then
    // the value FALSE is returned and the first n-1
    // characters of the source only are copied to dest.
    {
        if (strlen(src) > n-1)
            {
                strncpy(dest,src,n-1);
                dest[n-1] = '\0';
                return FALSE;
            }
        else
            {
                strcpy(dest,src);
                return TRUE;
            }
    }
}
```

This could be called as follows:

```c
if (safecopy(s1,10,s2))
    cout << "Copied string is " << s1;
else
    cout << "Partially copied string is " << s1;
```

### 2.2.1 Setting up a String type using typedef

Strings have been defined as arrays of characters, for example the following declarations define a string, with the name s, that may contain up to STRING_SIZE-1 characters:

```c
const int STRING_SIZE = 10;
char s[STRING_SIZE];
```
A particular program may use several strings all of the same size. Each of these strings must be declared as above, though from the declarations it is not obvious that each of these variables is used to hold a string. For example an array of characters could be used to access each character individually and never be used as a means of treating the array of characters as a complete entity representing a string. It is preferable from the point of view of readability that the declaration of a variable should reflect its use. This could be achieved if there was a data type for strings. C++ provides the `typedef` declaration which allows the setting up of programmer-defined types.

Thus the following declarations define a new data type with the name `string`:

```cpp
const int STRING_SIZE = 10;
typedef char string[STRING_SIZE];
```

Once the type `string` has been defined, variables of type `string` can be declared. For example to declare three variables `s1`, `s2` and `s3` of type `string`:

```cpp
string s1, s2, s3;
```

It is instructive to consider how the type definition was produced from the original declaration of a `char` array. First the variable name `s` in the original variable declaration was replaced by the name of the new type, namely, `string`. The declaration was then prefixed by the word `typedef`.

### 2.2.2 Example: Printing an enumerated variable

In Lesson 1 a function was written that used a `switch` statement to output a value for an enumerated type. This function can now be rewritten by using the type `string` that was defined above.

Thus given the enumerated type `Day` defined as follows:

```cpp
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
```

a function is required that will print out the appropriate external representation corresponding to a given value of the enumerated type. For example when the value `MON` is passed to the function then the string “Monday”, say, should be printed.

The first step is to declare an array whose elements are of the type `string` defined above and to then initialise the array to the strings which representing the enumerated values. Hence:

```cpp
const int STRING_SIZE = 10,
    DAYS_IN_WEEK = 7;
```
typedef char string[STRING_SIZE];

string days[DAYS_IN_WEEK] = {
    "Sunday", "Monday",
    "Tuesday", "Wednesday",
    "Thursday", "Friday",
    "Saturday"
};

To print out the string corresponding to the value MON then only requires:

    cout << days[int(MON)];

Note that the value for STRING_SIZE was chosen to allow space for the longest string required plus the terminating '\0'. Also note the cast of the enumerated type value to an integer. The enumerated type as defined takes integer values from 0 to 6 and the appropriate output strings also have indices 0 to 6 in the array days.

The function printday could now be written in the much simpler form:

    void printday(Day d)
    // Prints the day of the week d in English form.
    // Assumes that d takes a valid value.
    {
       cout << days[int(d)];
    }

2.2.3 Example: Entering an enumerated value

The inverse of the problem considered in the above section is now considered using the same enumerated type as an example. The problem of entering a value for an enumerated variable can be split into two steps, namely:

1. Enter the character string for the value of the enumerated variable into a variable of type string.

2. Search the array days to find if the entered string matches any of the stored external representations. If it does then the appropriate enumerated value can be returned. If it does not then an error situation must be signalled.

Each of the above steps can be implemented by a function. Assumptions have to be made as to which characters an external representation might contain. In this case it is assumed that each input string will consist of letters of the alphabet only and that any other characters will act as separators or terminators. With these assumptions the following basic algorithm is used to implement step 1:
Enter a character.
while the current character is not a letter do
    get the next character.
while the current character is a letter
{
    place the character in the string.
    get the next character.
}
Terminate the string properly.

This algorithm can be written as a C++ function as follows:

```cpp
void readday(string s)
    // Enters a string from the input stream cin.
    // Assumes that the string consists of letters
    // only so skips over any non-letters initially.
    // Then enters letters into string as long as
    // there is room for them and terminates input
    // on the first non-letter character found.
{
    char ch;
    int i = 0;
    cin.get(ch);
    while (!isalpha(ch)) cin.get(ch);
    while (isalpha(ch))
    {
        if (i < STRING_SIZE-1)
        {
            s[i] = ch;
            i++;
        }
        cin.get(ch);
    }
    s[i] = '\0';
}
```

testdays.cpp

Note that the function is extended from the algorithm description so that
if the input string is too long then no overflow of the string variable takes
place. Since the type string is defined as an array of char then even though
the parameter s is an output parameter it is not declared as a reference
parameter (compare using arrays as output parameters).

To implement step 2 a simple search is carried out which compares the
input string with the external representations that are contained in the
array `days`. This comparison takes place with each array element in turn until either a match is found or the end of the array is reached without a match having been found. A suitable C++ function is:

```cpp
int isday(string d, Day& day)
// If the string d is a member of the days
// array the function returns TRUE and
// the internal value of the corresponding
// enumerated type Day is returned in day.
// Otherwise FALSE is returned.
{
    int i = 0;
    int found = FALSE;
    while (i < DAYS_IN_WEEK && !found)
        if (!strcmp(d,days[i]))
            {found = TRUE;
             day = Day(i);
            }
        else i++;
    return found;
}
```

Observe that the reference parameter `day` is not changed if the string `d` does not appear in the array `Days`. Remember that the function `strcmp` returns zero when the two parameter strings are equal. Zero is interpreted as `false` by C++ hence `!strcmp(d,days[i])` is `true` when the two strings `d` and `days[i]` are the same.

### 2.2.4 Putting it all together

As already noted nothing in C++ can be used before it has been defined. If the functions of the last two sections are considered then it is obvious that they use various constants, variables and a data type that are not part of their own definition. The definition of these objects cannot be included in the `main()` function since they would only then be `local` to that function and would hence have no meaning outside it. The solution to this problem is to make them `global` to the main program and to the functions that require them. This is done by declaring them outside the `main()` program and before it in the file. Hence the following simple driver program to test these functions.

```
testdays.cpp
```
```cpp
#include <iostream.h>
#include <ctype.h>
#include <string.h>

// Global constants
const int STRING_SIZE = 10,
    DAYS_IN_WEEK = 7,
    FALSE = 0,
    TRUE = 1;

// Global Types
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
typedef char string[STRING_SIZE];

// Global Table of string values
string days[DAYS_IN_WEEK] = {
    "Sunday", "Monday",
    "Tuesday", "Wednesday",
    "Thursday", "Friday",
    "Saturday"
};

// Function Prototypes
void printday(Day);
void readday(string);
int isday(string, Day&);

void main()
{
    // Enters strings until user types "done". Checks
    // if strings are valid values for a variable of the
    // enumerated type Day.

    string s;
    Day d;
    cout << "Enter some days of the week: ";
    readday(s);
    while (strcmp(s,"done"))
    {
        cout << "String was " << s;
        if (isday(s,d))
        {
            cout << " day is ";
            printday(d);
```
} else cout << " Not a day";
  cout << endl;
  readday(s);
}
}

// Definitions of functions go in here.

testdays.cpp

2.3 Summary

- character strings are represented by arrays of characters with a terminating null character '\0'.
- a character string may be initialised at declaration.
- in a declaration the specified string length, if given, must be at least one greater than the length of the character string to which it is initialised.
- If the size of a string is not given in the declaration of a string then the size will be set to be one greater than the string to which it is initialised.
- operations on strings use library functions
- the string library functions will overwrite memory beyond that allocated to a string variable if the execution of a function would lead to placing more characters into a string than it can hold.
- the typedef statement allows string types to be defined.

2.4 Review Questions

1. Consider the following portion of C++:

```cpp
char ch1,ch2,ch3;
cout << "Enter a line of input: " << endl;
cin >> ch1 >> ch2 >> ch3;
cout << ch1 << ch2 << ch3 << "End output" << endl;
```

If the input from the user was:
what would be output after the input has taken place? There is a space between each letter that is input.

2. If the statement
   
   ```
   cin >> ch1 >> ch2 >> ch3;
   ```
   
in the previous question was replaced by
   
   ```
   cin.get(ch1); cin.get(ch2); cin.get(ch3);
   ```
   
   what would be output after input has taken place?

3. The string `s` contains a surname but with all letters lower case. What code would you write to capitalise the first letter?

4. The string `s` contains a word with the letters a mixture of lower and upper case. Write code to convert all the letters to upper case.

5. What is wrong with the following portion of C++?
   
   ```
   char s1[] = "Hi";
   strcat(s1, " there!");
   ```
   
6. The functions in the string library do not test that there is room in a destination string variable when placing or appending characters to it. If there is not enough space then the following locations in memory are overwritten. If you were writing your own safer functions for string copying and string concatenation what would your function prototypes be for these functions? Give a comment for each function that explains its operation.

7. Write code to find the longest string in an array of strings.

### 2.5 Exercises

1. Write a program to simulate a set of traffic lights at each end of a narrow bridge. Use an enumerated type to represent the state of the light at each end. The program should cycle through the possible states of each light.

2. Write a program which updates a football league on the basis of results. The league is read from a file, and consists of team name, followed by a number of integers representing games played, games won, games drawn, games lost, goals for, goals against, and league points.
The current week’s results are then read in, one to a line, consisting of home team name, goals scored, and away team name and the goals it scored. Points are awarded as 3 for a win, 1 for a draw, and 0 for a loss.

Output the resulting league.

3. Modify your solution to exercise 2 so that the points awarded for different results are changed to 7 for a win, 3 for a draw, and 1 for a loss.
Lesson 3

Multiple Source File Programs

So far, all the programs that have been written are comparatively small, and it causes no difficulty to keep the whole source code in a single file. However, as larger programs are developed, it is convenient to structure them in a modular style. Functions which are related to one another can be kept in the same source code file, which need only be recompiled when the code in that file changes. The main program, or other functions which use those functions only need the declaration of the functions, not their definitions.

In addition, access to variables can be restricted in several ways. At present, variables are either local to a particular function, or are global, and are accessible from any other function. This is not necessarily good practice.

3.1 Multiple Source Files

In Lesson 2 we developed a number of functions for dealing with the enumerated data type Day. In particular, some functions and an array of the strings representing the data values to be output corresponding to the values of the enumerated type. If we are to use these functions in other programs, then the source code could be copied into each program that wants to use it. This is wasteful, since we can compile these functions without knowing which programs or other functions are going to call them.

We create a file day.cpp which will have the definitions of the functions, and the declarations of the variables to be used.

```c++
#include <iostream.h>
#include <ctype.h>
#include <string.h>
```
// Global constants
cost int STRING_SIZE = 10,
        DAYS_IN_WEEK = 7,
        FALSE = 0,
        TRUE = 1;

// Global Types
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
typedef char string[STRING_SIZE];

// Global Table of string values
string days[DAYS_IN_WEEK] = {
    "Sunday", "Monday",
    "Tuesday", "Wednesday",
    "Thursday", "Friday",
    "Saturday"
};

// Function Definitions
void printday(Day)
{
    // ...
}

void readday(string)
{
    // ...
}

int isday(string, Day&)
{
    // ...
}

day.cpp

This can be separately compiled, but will not execute, because there is no main function. If we are to use Day variables in another program, then the file containing the source code will need the definition of the enumerated type Day and also the declarations of the functions readday, isday and printday. For example, we could set up another file, trial.cpp which has our program for testing the functions.
#include <iostream.h>
#include <ctype.h>
#include <string.h>

// Global constants
const int STRING_SIZE = 10;

// Global Types
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};

typedef char string[STRING_SIZE];

// Function Prototypes
void printday(Day);
void readday(string);
int isday(string, Day&);

void main()
{
    // Enters strings until user types "done". Checks
    // if strings are valid values for a variable of the
    // enumerated type Day.

    string s;
    Day d;
    cout << "Enter some days of the week: ";
    readday(s);
    while (strcmp(s,"done"))
    {
        cout << "String was " << s;
        if (isday(s,d))
        {
            cout << " day is ";
            printday(d);
        }
        else cout << " Not a day";
        cout << endl;
        readday(s);
    }
}

trial.cpp

Notice that both the files day.cpp and trial.cpp contain definitions of
string, Day, and STRING_SIZE, and declarations of the functions. This is bad practice, because it makes it possible to change the definition of Day (say) in one file, but not in the other, leading to inconsistent results when the functions are run on the same data. This problem can be avoided by using a header file. This file contains all the definitions of data types, declarations of functions, and constants that are needed by programs that are going to use the functions. Conventionally, it has a .h suffix. The file iostream.h contains the declarations of various functions and datatypes that are provided in the standard libraries. Similarly, math.h contains the declarations of the functions in the mathematics library.

For the functions and datatypes in the day.cpp file, define the file day.h, and give it the following contents:

```cpp
// Global constants
const int STRING_SIZE = 10;

// Global Types
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};

typedef char string[STRING_SIZE];

// Function Prototypes
void printday(Day);
void readday(string);
int isday(string, Day&);
```

day.h

and ensure that its contents are included in every file that uses the functions. To include the contents of a header in a file specify the header file name in a #include line, using " to surround the name.

```cpp
#include "day.h"
```

The < > used to delimit the file name previously signify that a system header file is to be included, using " signifies a user supplied header file.

### 3.2 Compiling Multiple File Programs

This section relates to compiling multiple file programs under Borland C++ for Windows, version 4.5. Other versions, such as Turbo C++, may be similar, or significantly different.
From the menubar, select the Project menu, then New Project. The name you give the project in “Project Path and Name” will by default be the name that the executable file will have. If you want a different name for the executable file, enter that name in the “Target” field. By default the .cpp file with the same name will be added to the list of source files. In the example in the last section, if the project is given the name trial, then the executable file will be trial.exe and the source code file trial.cpp will automatically be included in the list of files to be compiled. Before pressing the OK button, you should select the “Easywin” target option, and also press the “Application Expert” button, and clear the check marks against the .def and .rc files. Once the project is set up, you can add other source files to the list of those that need to be compiled by either using the “Add node” button (which has an icon, but no text), or by dragging the source file with the mouse from the ordinary Windows File Manager and releasing the mouse button over the representation of the target (in this case trial.exe) in the project.

3.3 Practical Considerations

In practice, it would be wise to separate the notion of a string into a header file of its own, mystring.h, which can be included in any file that uses the type string that is defined above. This will enable the type string to be used independently of the type Day. day.h needs to include mystring.h because it would not be possible to use the functions (or even to declare them) without knowing how the type string was represented.

Thus mystring.h would contain:

```c
// Global constants
const int STRING_SIZE = 10;

// Global Types
typedef char string[STRING_SIZE];
```

and day.h would contain

```c
#include "mystring.h"

// Global Types
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};

// Function Prototypes
void printday(Day);
void readday(string);
int isday(string, Day&);
```

This works well, so long as no file includes both mystring.h and day.h. (Although this is unlikely at the moment when only small program are being
written, it is almost impossible to prevent this in large software systems.)

If the file `mystring.h` is included twice, then there will be two declarations of the `const STRING_SIZE` and two `typedef` statements. This is a compile time error. In order to avoid this problem automatically, it is necessary to use the C++ preprocessor language. The preprocessor is already involved in processing the `#include` lines. It has facilities for limited programming too. We modify `mystring.h` to read:

```c
#ifndef MYSTRING_TYPE
// Global constants
const int STRING_SIZE = 10;

// Global Types
typedef char string[STRING_SIZE];
#define MYSTRING_TYPE
#endif
```

The line `#ifndef` means that the lines down to the following `#endif` line should be included in the code to be compiled only if the preprocessor symbol `MYSTRING_TYPE` is not currently defined. The line starting `#define` defines that symbol (it could give it a value too if needed). The first time through, the symbol will not be defined, so the definition of the type, and the `const` declaration will be included. With any subsequent attempts to include the same file, the symbol `MYSTRING_TYPE` will be already defined, so the conditional statement causes the lines to be skipped until the matching `#endif`.

### 3.4 Scope

In the previous Lesson, in section 2.2.2, the `string` array `days` was declared outside the function `printday` where it was used. This form of declaration, outside the body of any function, is known as global declaration. Any variable declared globally is available to all the functions defined in that file which appear after the global variable’s declaration. Thus in the following file:

```c
...
int funct_a(...) {
  ...
}

int global_var1;

int funct_b(...)```
the variable global_var1 is available in functions funct_b and funct_c, but not in the function funct_a. In general, it is not available to functions defined in other files either.

Other variables, which are declared within the body of a function, are local variables. They are only accessible within the function within which they are declared. If the identifier is the same as that of a global variable, then the global variable will be inaccessible from the body of the function. When that function exits, the variable is destroyed, and a new variable is created each time the function is called. The new variable will be reinitialised if it is declared with an initial value.

Variables can be declared inside any compound statement, that is within any statement group delimited by braces { and }. The variable exists until control passes outside the brace pair. If it has the same identifier as a variable declared globally, or declared within the same function, but in an enclosing pair of braces { and } then the global variable will be inaccessible. For example, consider the following program:

```c
float x;
int main()
{
 int x, y;
 ... // A
}

char y;

int funct_b(...)
{
 double y;
 ... // B
}

int funct_c(...)
{
 double x;
 ... // C
```
for ( i = 0; i < 10; i++ )
{
    int x = i+1;
    ... // D
}
... // E

At point A in the program, the identifiers x and y both access the local variables declared as type int in main; the global x, declared as type float is inaccessible. At point B, y gives access to the double variable, and x to the global variable declared float. At both points C and E, the variable accessed by x is the local one, declared to be of type double. y accesses the global variable of type char. At point D, x is a local int and the double x declared at the head of the function is not accessible directly. At point D, x initially has the value i+1 and is reset to that value each time that the for loop is executed.

3.5 External Variables

It was stated above that global variables were not usually accessible to functions defined in other files. In order to make a global variable accessible in other files than the one in which it is defined, it must also be declared in the files with functions which need to access the variable. If it is defined in the normal manner, then an error would occur when the object code was linked together to form an executable program because the variable would be multiply defined. Some linker programs will allow this if the definitions are identical, but it is not correct C++. If the variable is to be made available in other files, it should be declared extern in those files. For example, if the array of Day names was to be made available to functions outside the file day.cpp, it would be declared

    extern string days[];

This declaration tells the compiler the type of the variable, and its identifier. Notice that the size of the array is not specified, nor is any initialisation specified. No restrictions can be placed on the type of access that another function has to a global variable.

It is common practice to place extern declarations in a header file, and #include the file in whichever other files need the declarations. This ensures that each function is using a consistent declaration of the variables. Including the header file in the file in which the variables are defined ensures that the extern declaration is consistent with the definition.
3.6 static Variables

It is sometimes convenient to have some variables which are accessible to several related functions, but not accessible in any way to other functions. If the variables and the functions which need to access them are stored in a single source file, then the functions will have access to the variables as global variables. However, functions declared in other source files will be able to access the variables too, merely by making an extern declaration of the same identifiers. Declaring a global variable to be static prevents it from being accessed directly by functions other than those functions which are defined in the same source file. If need be, functions can be declared to be static too, making them inaccessible to functions from outside the same source file.

static local variables are similar to ordinary local variables, except that they keep their values from one invocation of the function within which they are declared to another. For example,

```c
int dividend(int i, int j)
{
    static int no_zero = 0;
    if ( j == 0 ) {
        no_zero++;
        cerr << "Zero divisor " << no_zero << " times" << endl;
        return 0;
    }
    return i / j;
}
```

will output a message each time that division by 0 is attempted, and will record how many times it has been attempted.

3.7 Summary

- Program sources can be split across multiple files. External variables, constants, and function declarations can be declared in header files (with suffix .h). This file is then #included in any file that uses the variables.

- Multiple definitions can be prevented by using the #ifndef, #endif, and #define compiler directives.

- static declarations can be used to make global variables private to the file within which they are declared.
3.8 Review Questions

There are no review questions for this Lesson.

3.9 Exercises

1. Implement the files day.cpp, day.h, trial.cpp etc. as described in this Lesson. Compile and execute the resultant multi-file program.
Lesson 4

Two-dimensional Arrays

Frequently in computing data has to be stored in the form of a table. For example a table which records the rainfall in each month for this century. This could be laid out as a two-dimensional array of numbers, each row representing a year and each column representing a particular month somewhat as follows:

\[
\begin{array}{cccccccccccc}
\text{j an} & \text{f eb} & \text{m ar} & \text{a pr} & \text{m ay} & \text{jun} & \text{j ul} & \text{a ug} & \text{sep} & \text{oct} & \text{nov} & \text{d ec} \\
1900 & 5 & 3 & 6 & 7 & 4 & 1 & 1 & 3 & 5 & 7 & 5 & 4 \\
1901 & 6 & 6 & 7 & 2 & 5 & 3 & 4 & 2 & 4 & 7 & 3 & 4 \\
1902 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
1997 & 7 & 5 & 6 & 3 & 4 & 5 & 6 & 6 & 7 & 8 & 7 & 8 \\
\end{array}
\]

A particular element in this table would be accessed by first using the year to find the correct row and then the month to find the correct column within the row. Thus a particular value in the table is found by using two subscripts, the year and the month.

C++ allows such tabular data structures to be set up by extending the concept of a one-dimensional array to allow two-dimensional (and higher order) arrays. The rainfall data above can be stored in a two-dimensional array and individual elements are accessed by using two subscripts. A two-dimensional array is declared as follows:

\[
\text{type arrayname[ rows ][ columns ]}
\]

where \text{type} indicates the type of the elements (float, int etc.), the \text{arrayname} follows the usual rules for identifiers in C++ and \text{rows} indicates the number of rows in the array and \text{columns} indicates the number of columns.

For example an array of floats with 4 rows and 3 columns is declared as follows:
float A[4][3];
to represent the 12 real values:

\[
\begin{array}{ccc}
  a_{00} & a_{01} & a_{02} \\
  a_{10} & a_{11} & a_{12} \\
  a_{20} & a_{21} & a_{22} \\
  a_{30} & a_{31} & a_{32}
\end{array}
\]

where the \((i,j)\)th element is accessed by \(A[i][j]\).

In terms of the layout above the first subscript, \(i\), represents the row number and the second subscript, \(j\), represents the column number. As in one-dimensional arrays the subscripts start at zero.

### 4.1 Examples

Given the declarations:

```c
const int m = 4,
       n = 5,
       p = 6;
float A[m][n], B[m][n], C[m][n], D[n][p], E[m][p];
float sum;
int i, j, k;
```

the following fragments of code would carry out the operations detailed:

1. Calculate the sum of the elements in the \(j\)th column of \(A\)
   ```c
   sum = 0.0;
   for (i=0; i<m; i++)
       sum += A[i][j];
   ```

2. Set array \(C\) to the sum of the arrays \(A\) and \(B\)
   ```c
   for (i=0; i<m; i++)
       for (j=0; j<n; j++)
           C[i][j] = A[i][j] + B[i][j];
   ```

3. Find the smallest element in array \(A\) and store its value in \(min\)
   ```c
   min = A[0][0];
   for (i=0; i<m; i++)
       for (j=0; j<n; j++)
           if (A[i][j] < min)
               min = A[i][j];
   ```

4. Form the matrix product \(A*D\) and store the result in array \(E\)
for (i=0; i<m; i++)
    for (j=0; j<p; j++)
    {
        sum = 0.0;
        for (k=0; k<n; k++)
            sum += A[i][k]*D[k][j];
        E[i][j] = sum;
    }

5. The elements of a two-dimensional float array are to be input from a data file called arraydata and placed in a two-dimensional array which has RANGE rows and columns. The file holds, in order, values for m and n, the number of rows and columns respectively, followed by the array elements in row order. It is assumed that both m and n are less than or equal to RANGE. The following program would carry out this task:

```cpp
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
char infile[10] = "arraydata";

int main()
{
    const int RANGE = 100;
    float A[RANGE][RANGE];
    ifstream ins; // input stream
    int i,j;
    int m,n;

    // Associate file name with input stream

    ins.open(infile);
    if (ins.fail())
    {
        cout << "could not open file " << infile
             << " for input" << endl;
        return 1;
    }

    // Get values for m and n from file

    ins >> m;
```
ins >> n;

    // Read values of array data into array

    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            ins >> A[i][j];

    etc.

4.2 Initialisation of two-dimensional arrays

The initialisation of one-dimensional arrays at declaration time has already been covered. To accomplish this a list of constant expressions enclosed in braces is assigned to the array. For example:

    int A[6] = {1, 2, 3, 4, 5, 6};

    Since C++ really treats a two-dimensional array with m rows and n columns as a one-dimensional array with m rows whose elements are one-dimensional arrays with n elements then a two-dimensional array is initialised as follows:

    int A[4][3] = { row1 , row2 , row3 , row4 };

using the rule for initialising a one-dimensional array. The elements row1, row2 etc are of course one-dimensional arrays so can be expanded similarly, giving:

    int A[4][3] = { {1, 2, 3} ,
                    {4, 5, 6} ,
                    {7, 8, 9} ,
                    {10, 11, 12} }

which sets up the array as:

    1 2 3
    4 5 6
    7 8 9
    10 11 12

4.3 Two-dimensional arrays as function parameters

When declaring a two-dimensional array as a formal parameter to a function the second dimension of the array must be specified. However the first
dimension of the array need not be specified. Thus if a function was to take a two-dimensional array of \texttt{ROWS} rows and \texttt{COLS} columns as a parameter then the function declaration should declare the array as follows:

\begin{verbatim}
int example(float array[][COLS], other parameters)
\end{verbatim}

This function could then be used with any two-dimensional float array that had \texttt{COLS} columns no matter how many rows it had.

\textbf{Example}

The following function returns the sum of the elements in the \texttt{j}th column of the first \texttt{m} rows of a float array with \texttt{COLS} columns, \texttt{j} being less than \texttt{COLS}. Obviously \texttt{m} would have to be less than \texttt{ROWS}. \texttt{COLS} is a constant.

\begin{verbatim}
const int COLS = 10,
   int ROWS = 20;

float sumcol(const float A[][COLS],
              int m,
              int j)

   // returns the sum of the first \texttt{m} elements in the \texttt{j}th
   // column of the array \texttt{A} which has \texttt{COLS} columns and
   // greater than or equal to \texttt{m} rows.
{
   int i;
   float sum = 0.0;
   for (i = 0; i < m; i++)
   {
      sum += A[i][j];
   }
   return sum;
}
\end{verbatim}

It is worth noting that because the formal parameter \texttt{A} is an input parameter of the function it should not be changed by the function. Since arrays are passed as reference parameters there is nothing to stop the function from changing some or all of the array elements. Hence the array parameter has in this case been declared as being \texttt{const} which prevents any value being changed.

The function might be called in a program as follows:

\begin{verbatim}
void main()
{
\end{verbatim}
Thus space would be reserved for 20 rows and 10 columns, however the call of `sumcol` specifies that the sum of the elements in the 3rd column in the first m (=5) rows only should be returned.

### 4.4 Example: Mark Processing

Consider the following specification for a program that is to carry out the processing of Examination marks:

Information on the performance of a class of students is available on a file. The file contains firstly the number of students in the class followed by, for each student, the student’s name and the students percentage marks in each of four subjects. The student’s name always begins with a letter and can contain any character, it is terminated by a dollar sign. A typical file might be as follows:

25
Robert Brown$ 43 56 28 61
W.S.McLean$ 53 57 80 76
William S. Smith$ 56 34 76 45

The requirement for the program is to print out the data for each student together with the student’s average mark. The output should firstly be in the order of entry from the file and then in descending order of the student’s average mark.

The input data must obviously be stored when it is input so that it can be sorted later before outputting it in sorted order. The stored data can be thought of as a table of entries, each entry holding the complete information on one student. Using this structure the following algorithm is constructed:

initialise.
Open file.
if successful then
{
    get number of students.
    for each student do
    {
        Enter name and store it in table.
        Enter marks and store in table.
        Calculate average and store in table.
    }
    Output table.
    Sort entries in table on average mark.
    Output table.
}
else
    output error message.

The obvious way to store a table is as an array of entries. However each entry must contain a name (a character string), four marks (integers) and an average mark (a real). With the data-structuring facilities so far considered it is not possible to define a single entity which holds such disparate information. However an array of strings has already been used in Lesson 2, this suggests that the student names could be stored in this way (say in an array called names) with an index being used to retrieve a particular student name. Similarly an array of floats could be set up to store the average (say in a structure called average). That only leaves the marks to be considered, for each student four marks have to be stored, this obviously is a two-dimensional structure of integers with a row for each student and each row containing four marks. Hence if this structure is called marks then the entry for a particular student is held in names[i], marks[i][0], marks[i][1], marks[i][2], marks[i][3] and average[i].

The above method of storing related information of different types in different arrays so that by using the same index to access each array one retrieves all the information relating to a particular case is often referred to as using parallel arrays.

The above algorithm is now rewritten in terms of arrays names, marks and average and functions.

    initialise.
    open_file(markfile).
    if successful then
    {
        enter number of students -> ns.
        for each student do
        {

48
set student index i to zero.
read_name(markfile,name[i]).
get_marks(markfile,marks,i).
set average[i] to calc_average(marks,i);
increment i.
}
output_marks(name,marks,average,ns).
sort(name,marks,average,ns).
output_marks(name,marks,average,ns).
}

where the functions are defined as follows:

Function name: open_file
Operation: Open a file for input.
Description: Opens a file whose name is entered by the user and connects it to an input stream passed as a parameter. If successful returns true else false.
Parameters: Input parameter - ifstream& ins
Returns: int

Function name: read_name
Operation: Enters a character string from an input stream.
Description: Enters a character string from an input stream. Starts the string with the first letter found and terminates when STRING_SIZE characters have been entered or when a dollar character is entered. Any extra characters are ignored. Returns the entered string by a parameter.
Parameters: Input parameter - ifstream& ins
Output parameter - string name

Function name: get_marks
Operation: Enters marks from an input file stream.
Description: Enters SUBJECTS percentage marks from a specified input stream and stores them in the ith row of the marks array.
Parameters: Input parameter - ifstream& ins
Input parameter - int i
Output parameter - array marks

Function name: calc_average
Operation: Calculates a student’s average mark.
Description: Evaluates the average mark of the marks held in the ith row of the marks array.
Parameters: Input parameter - array of marks
Input parameter - int i
Returns: float

Function name: output_marks
Operation: Outputs the tabulated marks for a class.
Description: Outputs a table with headings. Each row holds the name of a student, SUBJECTS percentage marks and an average mark.
Parameters: Input parameter - an array of names
Input parameter - an array of marks
Input parameter - an array of average marks
Input parameter - number of students

Function name: sort
Operation: Sorts the arrays holding student details into descending order of average marks.
Description: Sorts the arrays holding student names, student marks and student average marks into descending order on average mark.
Parameters: Input parameter - an array of names
Input parameter - an array of marks
Input parameter - an array of average marks
Input parameter - number of students

It is assumed in the above that constants SUBJECTS, STUDENTS and STRING_SIZE have been defined global to the main program and the functions. Hence the main program together with constant declarations and function prototypes is:

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <string.h>
#include <iomanip.h>

const int STRING_SIZE = 30,
        STUDENTS = 100,
        SUBJECTS = 4,
        FALSE = 0,
        TRUE = 1;
typedef char string[STRING_SIZE];

int open_file(ifstream&);  
void read_name(ifstream& ,string);  
void get_marks(ifstream&, int[][][SUBJECTS], int);  
float calc_average(int[][SUBJECTS], int);  
void sort(string[], int[][][SUBJECTS], float[], int);  
void output_marks(string[], int[][][SUBJECTS], float[], int);

void main()
{
  int i;
  int ns;  // number of students
  ifstream results;
  string name[STUDENTS];  // student names
  float average[STUDENTS];  // student average marks
  int marks[STUDENTS][SUBJECTS];  // student marks
  if (open_file(results))
  {
    results >> ns;
    for (i=0; i<ns; i++)
    {
      read_name(results,name[i]);
      get_marks(results,marks,i);
      average[i] = calc_average(marks,i);
    }
    output_marks(name,marks,average,ns);
    sort(name,marks,average,ns);
    output_marks(name,marks,average,ns);
  }
}

marks.cpp

Note that the use of constants for the number of characters in the names, the maximum number of students and the number of subjects taken by a student allows the simple changing of the program to allow longer names, more students or a different number of subjects.

Each of the functions used can now be written, a few of them are detailed below. The rest are left as an exercise.

void read_name(ifstream& ins,string s)

  // Enters a string from the input stream ins.
// Assumes that the string is terminated by
// a $ character and commences with an
// alphabetic character.
// Only the first STRING_SIZE characters are
// entered, any more are skipped. The string
// entered is stored in s.

{}
char ch;
int i = 0;
ins.get(ch);
    // skip to an alphabetic character
while (!isalpha(ch)) ins.get(ch);
    // enter characters until a '$' terminates
    // string, stop storing characters after
    // string is full.
while (ch != '$')
{
    if (i < STRING_SIZE - 1)
    {
        s[i] = ch;
        i++;
    }
    ins.get(ch);
}
    // terminate string
s[i] = '\0';

void get_marks(ifstream& ins, int m[][SUBJECTS], int sno)

    // Enters SUBJECTS marks from the file ins
    // and places them in the row with index
    // sno in the array m

{}
int i;
for (i = 0; i < SUBJECTS; i++)
    ins >> m[sno][i];

float calc_average(int m[][SUBJECTS], int i)

    // Returns the average of the marks in the
// row with index i of the array m.
{
    int j;
    float sum = 0.0;
    for (j=0; j<SUBJECTS; j++)
        sum += m[i][j];
    return sum/SUBJECTS;
}

marks.cpp

The constant definitions and function prototypes could be placed in an appropriate .h file and included in the main program file. The function implementations could be incorporated in a separate file and pre-compiled before being linked with the main program as covered in Lesson 3.

4.5 Summary

- A two-dimensional array element is accessed by two subscripts. The first subscript is usually thought of as a row number and the second subscript as a column number in a tabular layout.
- A two-dimensional array is treated as a one-dimensional array whose elements are themselves one-dimensional arrays.
- A two-dimensional array can be initialised when it is declared.
- When a two-dimensional array is a formal parameter to a function the second dimension must be given in the function declaration. This formal parameter can only be replaced in a call of the function by an actual parameter which has been declared with the same value for the second dimension.

4.6 Review Questions

1. An array is declared and initialised in a C++ program as follows:

   ```cpp
   int A[4][4] = {{1 2 3 4},{5 6 7 8},{9 8 7 6},{5 4 3 2}};
   ```

   what are the values of A[1][1], A[2][3] and A[4][3]?

2. To what would the variable `sum` be set by the following assuming that the array A is as in question 1:

   ```cpp
   marks.cpp
   ```
int sum = 0;
for (i=0; i<4; i++)
    sum += A[i][2];

3. What is wrong with the following function heading?

    float example(float A[], int m, int n)

What would have to be done to correct it? If the function is supposed
to return the sum of the elements in the first m rows and first n columns
of the array A what restrictions would have to be placed on the values
of m and n used in a particular call?

4. Consider the following definitions:

    const int COLS = 10,
    ROWS = 10;

    // function prototype
    float example(float A[][COLS], int m, int n); 

    float A[ROWS][COLS], B[15][COLS], C[10][8];

Which of the following function calls would produce the correct re-
sults?

    sum = example(A,8,9);
    sum = example(A,12,4);
    sum = example(B,15,COLS);
    sum = example(C,10,COLS);

4.7 Exercises

In the following exercises it is suggested that you write various functions re-
lated to operations on two-dimensional arrays. As you write these functions
place them in a separate implementation file and add a suitable prototype
declaration to an associated .h include file. Also place in the include file any
constant definitions that are required. You will also require a separate driver
program to test the functions, as you add each function add appropriate test
instructions to the driver program.

For details of how to compile multi-file programs see Lesson 3.

1. Write a function to input a filename from a user and to return the file-
name as an argument of the function. Assume that a file name cannot
be longer than 20 characters. Once you have written the function incorporate it in a driver program that inputs a filename from the user and then prints the filename so that the correctness of the function can be checked.

2. Make up a file that contains data as follows:

   (a) two integers representing the number of rows and columns in a two-dimensional array (m and n say).

   (b) m*n floating point numbers which represent the elements of an m*n array in row-by-row order.

   Make the dimension fairly small and the numbers simple. A 5*3 array would be fine.

3. Decide on a maximum size of array that you will require, say 10*10 and set up appropriate constants in the include file. Now write a function which takes a file name as a parameter and inputs data from the file into a two-dimensional array parameter and also returns the actual dimensions of the array (m and n). This function should return false if the file cannot be accessed, otherwise true. Place a test for this function in your driver program. Check the contents of the array by printing them.

4. Write a function to output the contents of the first m rows and first n columns of a two-dimensional array. Assume m and n will not exceed your assumed maximum size and that the maximum size is small enough to allow the array to be printed in tabular row and column form. A call of this function can now be incorporated in the driver program.

5. Now that you can input an array from file and output its contents write more functions to operate on two-dimensional arrays, and test them as you proceed. Some suggestions are:

   (a) A function to output the sum of the elements in a 2D array.

   (b) A function to find the largest element in the ith row of an array.

   (c) A function to find the largest element in a 2D array that uses the previous function.

   (d) A function which subtracts two arrays of the same dimensions from each other.

   (e) A function which returns the sum of the elements in the ith row of a two-dimensional array.

   (f) A function to return the subscript of the row which has the largest sum. This should use the function defined in question 5e.
Lesson 5

The struct data type

The representation of a collection of related items, all of the same type, in an array has already been covered. Also in Lesson 4 the representation of related data of different types was covered by using parallel arrays. Another, and better, method of representing a collection of items which are related to each other but have different types is to use the idea of a structure, or, as it is called in C++, a struct.

For example each item in a store may have an associated record which contains:

- A reference number
- A description
- Who the supplier is
- The number in stock
- The price
- The re-order level
- etc.

To allow the storage of such varied information in a single structure C++ allows related data of different types to be collected together in a struct. A struct in C++ is a type which allows the definition of a data structure which contains a fixed number of components. Each component is called a member and is identified by a member name. Each member name is given its own type.

For example a struct holding a date could be defined as follows:

```cpp
const int string_length = 10;
struct date
{
    int day;
    char month[string_length];
    int year;
}; // Note the semi-colon!
```
In this struct the member names are day, month and year. It is important to note that the above declaration of a struct only defines a new data type and does not allocate any storage space. Space is only allocated when a variable of the type is declared. For example two dates date1 and date2 could be declared as follows:

```c
date date1, date2;
```

The variables date1 and date2 now have the structure defined above for a date, that is they have members day, month and year. To access a member of a struct the following construction is used:

```c
struct variable. member_name
```

For example the year member of the variable date1 of type date is accessed by:

```c
date1.year
```

Since the member year was defined as an int it can be used in any situation where an int variable can be used. For example values could be assigned to a variable of type date as follows:

```c
date1.day = 24;
strcpy(date1.month,"January");
date1.year = 1995;
```

or arithmetic could be carried out on the members of a struct type as in:

```c
date1.year++;  
if (date1.year % 4 == 0) leap_year = TRUE;
```

Structures of the same type can be assigned to each other, for example:

```c
date2 = date1;
```

The general form of a struct type declaration is:

```c
struct struct_type  
{  
type1 id_list1;  
type2 id_list2;  
.  
.  
typen id_listn;  
};
```

Where the typei can be any basic type or user-defined type such as another structure. Note that, unusually, the final closing brace is followed by a semi-colon.
5.1 Examples

A type definition for a structure to hold a point in two-dimensional space could be defined as follows:

```c
struct point
{
    float x, y;
};
```

Then given definitions:

```c
point p1, p2;
float dist, rad;
```

the distance of a point from the origin or the distance between two points could be computed as follows:

```c
// distance of p1 from origin
rad = sqrt(p1.x * p1.x + p1.y * p1.y);
// distance between p1 and p2
dist = sqrt((p2.x - p1.x)*(p2.x - p1.x) + (p2.y - p1.y)*(p2.y - p1.y));
```

Using the above definition of a point a `struct` to hold a line as its two end points could now be defined:

```c
struct line
{
    point p1, p2;
};
```

The distance between the x co-ordinates of the end points of a line would then be given as follows:

```c
float xdist;
line l;

xdist = l.p1.x - l.p2.x;
```

Note how a member of a member of a function is accessed. Since `l.p1` is of type `point` it is possible to access the x co-ordinate of that point by appending `.x` to it.

The following piece of code evaluates the centroid of n points that are held in an array of `point` structures. The centroid is at the point whose x and y values are given by the average x-value and average y-value of the points:

```c
58```
const int n = 100;
int i;
float sumx = 0.0, sumy = 0.0;
point centroid, p[n];

for (i=0; i<n; i++)
{
    sumx += p[i].x;
    sumy += p[i].y;
}
centroid.x = sumx/n;
centroid.y = sumy/n;

5.2 Initialising a struct

Struct variables can be initialised at declaration time. The initialising values must correspond in type and order to the member variables in the struct definition.

For example given the struct date defined earlier as:

```c
const int string_length = 10;
struct date
{
    int day;
    char month[string_length];
    int year;
}; // Note the semi-colon!
```
a variable of type date could be initialised as follows:

date date1 = {21, "August", 1934};

5.3 A struct as a function parameter

A struct can be passed as a parameter to a function. As usual the actual parameter in the function call must agree with the type of the formal parameter in the function declaration.

For example the following function returns the distance between two variables of the type point defined above:

```c
float dist(point p1, point p2)
{
```

// Returns the distance between two points p1 and p2
float d;
d = sqrt((p1.x-p2.x)*(p1.x-p2.x)
    + (p1.y-p2.y)*(p1.y-p2.y));
return d;
}

This function could be used as follows:

point p,q;
line l1;
float dist1, length;

dist1 = dist(p, q); // distance from p to q
length = dist(l1.p1, l1.p2); // length of line l1

Structures can also be passed as reference parameters, for example the following function will enter two real values into the member variables of a point variable.

void inpoint(point& p)
{
    cout << "Enter the x and y co-ordinates of a point: ";
    cin >> p.x >> p.y;
}

The range of a set of numbers could be represented as the minimum and maximum values of the numbers, thus a struct for the range could be defined as follows:

struct range
{
    float min, max;
};

The following function will return the range of x values and y values in an array of n points:

void rangefunc(point p[], // an array of points
    int n,       // number of points
    range& rx,  // range of points in x
    range& ry   // range of points in y
)

    // Finds the range of x values and y values in
// the array of n points p and places results in
// rx and ry
{
    int i;

// initialise max and min values
    rx.min = rx.max = p[0].x;
    ry.min = ry.max = p[0].y;

// Now loop over remaining values
    for (i=1; i<n; i++)
    {
        if (p[i].x < rx.min)
            rx.min = p[i].x;
        else if (p[i].x > rx.max)
            rx.max = p[i].x;
        if (p[i].y < ry.min)
            ry.min = p[i].y;
        else if (p[i].y > ry.max)
            ry.max = p[i].y;
    }

5.4 Example: Mark Processing re-visited

In section 4.4 parallel arrays were used to hold the details about a class of
students. It is much better to represent such information in a struct. A
suitable definition for a struct to hold the information on a student could be:

const int STRING_SIZE = 30,
    STUDENTS = 100,
    SUBJECTS = 4;

typedef char string[STRING_SIZE];

struct studdata
{
    string name;
    int marks[SUBJECTS];
    float average;
};
The student marks together with the student names and average marks can now all be held in a single array. This array could be declared as:

```cpp
studdata data[STUDENTS];
```

The third mark of the student with index i would be accessed by:

```cpp
data[i].marks[2]
```

The function prototypes are now changed as follows:

```cpp
void read_name(ifstream&, string);
int open_file(ifstream&);
void get_marks(ifstream&, studdata[], int);
float calc_average(studdata[], int);
void sort(studdata[], int);
void output_marks(studdata[], int);
```

The function `get_marks`, for example, could now be written as follows:

```cpp
void get_marks(ifstream& ins, studdata m[], int sno)

// Enters SUBJECTS marks from the file ins
// and places them in the marks member of
// the struct m[sno].
{
    int i;
    for (i=0; i<SUBJECTS; i++)
        ins >> m[sno].marks[i];
}
```

The other functions are left as an exercise.

### 5.5 A data type for financial calculations

C++ supplies various data types together with operations on them. For example integer types with operations of add, subtract, multiply, divide and modulus. This section shows one way in which a user-defined data type can
be added to C++ together with operations that operate on variables of the type. This can be done by the use of structs to define the data type and by the use of functions to implement the functions. By using the ideas of separate compilation much of the detail can be hidden from the user of the new type.

The illustration used is that of the manipulation of sums of money expressed in pounds and pence with operations such as add two sums of money together, multiply a sum of money by an integer, find a given percentage of a sum of money etc.

A suitable definition for a struct defining a sum of money is:

```c
struct money
{
    int pounds;
    int pence;
};
```

Functions could then be written to implement operations upon variables of type money. Among useful functions would be:

```c
void addmoney(money m1, money m2, money& m3);
void submoney(money m1, money m2, money& m3);
void mulmoney(money m, int n, money& m1);
void percent(money m, float percentage, money& m1);
void initmoney(money& m, int pounds, int pence);
void printmoney(money m);
void readmoney(money& m);
```

These functions are easy to write, for example addmoney could be written as:

```c
void addmoney(money m1, money m2, money& m3)
{
    m3.pounds = m1.pounds + m2.pounds;
    m3.pence = m1.pence + m2.pence;
    if (m3.pence > 99)
    {
        m3.pounds++;
        m3.pence -= 100;
    }
}
```
Obviously the struct definition and function definitions could be passed on to another programmer who required to use calculations on money in a program. The best way to do this is to follow the method described in Lesson 3:

1. Make up a file money.h which holds the structure definition for the data type money and prototype definitions of the functions. Thus the file money.h might look like the following:

```c
// money.h Structure and Function Prototypes for money data type.

#ifndef MONEY_H
#define MONEY_H

// Structure for a sum of money

struct money
{
    int pounds;
    int pence;
};

//Operations on sums of money

void addmoney(money, money, money&);
void submoney(money, money, money&);
void mulmoney(money, int, money&);
void percent(money, float, money&);
int initmoney(money&, int, int);
void printmoney(money);
void readmoney(money&);
#endif
```

money.h

2. If this file is now stored with all the other include files in the system then it can be included by the user in the usual way by:

```c
#include <money.h>
```

In most cases however, it will be stored in the users own files and will be specified as

```c
#include "money.h"
```
3. A file `money.cpp` which includes the full function definitions is now produced. For example:

```cpp
#include <iostream.h>
#include <iomanip.h>
#include <money.h>

// function definitions

void addmoney(money m1, money m2, money& m3)
{
    m3.pounds = m1.pounds + m2.pounds;
    m3.pence = m1.pence + m2.pence;
    if (m3.pence > 99)
    {
        m3.pounds++;
        m3.pence -= 100;
    }
}

// and the implementations of the other functions

money.cpp
```

This file is now compiled on its own to produce an object file which can be stored with the other library files. If the user's program is now compiled and linked with this library file then an executable program is produced.

What has been done here is to supply the user with a new data type together with a set of operators and functions which now appear to the user as part of the language. Such types are often called Abstract Data Types (ADT's).

The use of Abstract Data Types has many advantages, e.g.

1. The `money` library can be written by an expert and comprehensively tested. This obviously saves the user work and cuts down on possible sources of error in a program and makes debugging easier. If an ADT is defined in a sufficiently general way then it may be re-used in many other programs.
2. The internal details of the implementation are hidden from the user, the only way a user can operate upon a sum of money is by using the supplied functions. This process is called encapsulation. This prevents the user from tampering with the ADT, possibly introducing errors.

3. Because the money functions have already been compiled and are stored in object form compile time is saved.

4. If a better way of writing one of the functions or representing the type becomes available then the implementation file for the library can be changed and re-compiled for the library. The user does not have to change the program in any way, but gains the advantage of the new, more efficient routines. Of course the program will have to be re-linked with the new library file.

For example say a programmer had to write a program to enter

1. The number of items on an invoice.

2. For each item, a reference number, the quantity ordered and the unit price.

and was to output an invoice detailing the total cost for each item and the total cost of the invoice. This could be easily done using the abstract data type developed above for handling money. A very simple program with elementary formatting of output etc. could quickly be written using the money data type.

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

void main()
{
    money unitp, itemsum, totalsum;
    int no_items; // no of items
    int i;
    int refno, quantity;
    cout << "Enter no of items: ";
    cin >> no_items;
    i = initmoney(totalsum,0,0);
    for (i=0; i<no_items; i++)
    {
        cout << "Enter reference number and quantity ordered: ";
        cin >> refno >> quantity;
        cout << "Enter unit price: ";
```
```cpp
readmoney(unitp);
mulmoney(unitp,quantity,itemsum);
addmoney(itemsum,totalsum,totalsum);
cout << refno << " " << quantity << " ";
printmoney(unitp);
cout << " ";
printmoney(itemsum);
cout << endl;
}
cout << "Total is ";
printmoney(totalsum);
cout << endl;
}
```

trymoney.cpp

The important point to be made is that the user of the money data type did not require to know anything about how the type is implemented or how to carry out operations on variables of the data type. The above program could easily be used for any currency without change if the library functions were changed. For example the currency could be dollars and cents and dollar signs could be printed instead of pound signs, similarly for any other currency.

5.6 Summary

- A **struct** is an aggregate type which allows a collection of components, of possibly differing types, to be used as a single entity.

- Each component of a **struct** is called a **member** and is identified by a **member name**.

- A **struct** can be initialised at declaration.

- A **struct** can be passed as a function parameter, either as a value or reference parameter.

- An **Abstract Data Type** is a data structure definition plus a set of operations (functions) that operate on the data structure.

- An Abstract Data Type can be implemented by a **struct** and functions that operate on the **struct**. If the **struct** and the functions are placed in a separate file and separately compiled then all details of the Abstract Data type can be hidden from the user of the Abstract Data Type.
5.7 Review Questions

1. What is the main difference between a struct and an array? Which would you use to store a record for a student (Name, address, course, marks etc.)? Which would you use to store a list of the students in a class?

2. A struct is defined as follows:

   ```
   const int STRING_LENGTH = 20;

   struct employee_record
   {
       char first_name[STRING_LENGTH];
       char second_name[STRING_LENGTH];
       int salary;
       char department[STRING_LENGTH];
   };
   ```

   If two variables `emp1` and `emp2` are declared of type `employee_record` then write C++ statements to
   
   (a) Assign the contents of `emp1` to `emp2`.
   (b) Compare `emp1` and `emp2` for equality.
   (c) Write out the complete name of `emp1`.
   (d) Write out the salary of `emp2`.
   (e) Write out the initials of `emp1`.

3. Declare a variable of type `employee_record` as defined in question 2 and initialise it to the name Joe Bloggs who works in the Workshop and earns 10,000 pounds a year.

4. Define a struct to hold a student record. It should hold the student christian name and surname, date of birth (define a separate struct for this), current year of course and marks for each year of the course. Define suitable constants for string lengths, number of subjects per year etc. Then write statements to output a student’s name and date of birth and to assign a mark of 72 to the second subject in the third year.

5.8 Exercises

1. In 5.5 a struct for money together with operations that operated upon such types was developed. Implement the functions which were
not implemented in that treatment. Alter the program given for invoice processing so that VAT is included in the invoice. The function percent can be used to calculate the VAT.
Lesson 6

Tables and Table Searching

A table is a data structure which can represent a list of entries of some kind. Typical operations on a table are to insert an entry, to delete an entry and to search for a particular entry. The entries may be simple objects like an integer or a floating-point number. However it is more common that each entry should consist of several items of information, e.g. it may be a C++ structure. Typically one of the items constituting an entry will be used as the prime means of identifying the entry, this item is often called the key.

For example a student data-base might maintain a table in which each entry holds the students name, Department, Home address, Term address, year in course, examination results, etc. The obvious key on which such a table might be searched would be the student name.

It is assumed for the moment that a table will be represented as a one-dimensional array of structs set up as follows:

```c
const int max = ?, // maximum number of elements
          length = 20; // maximum length of a key

struct entrytype
{
    char key[length]; // key on which table is accessed
    infotype info;   // holds information associated
                     // with the key
};

entrytype table[max]; // An array of table entries
```

For the moment while considering algorithms that operate upon tables it does not matter what precise information is held in the struct infotype.
6.1 The Search Process

One of the commonest operations that takes place upon a table is that of
searching a table for the entry defined by a particular key and returning the
information associated with the key. Thus a function such as the following
is required:

```c
int search(char word[], // search for this word
tentrytype table[], // in this table
int n, // number of entries
int& index // index of entry if found )

// Searches the table table for an entry with word
// as its key. table has n entries and if found
// the index of the entry is returned in index.
// If found then TRUE is returned as the value of
// the function, otherwise FALSE is returned.
```

Different methods for implementing a function such as the above exist.
Two methods are considered along with their advantages and disadvantages.

6.1.1 Linear Search

The simplest and most obvious way to search a table is to use Linear search
i.e. examine the 1st, 2nd, 3rd,... entries until the entry with the required
key is found or the end of the table is reached without the entry being found.
The body of the search function in section 6.1 could be written as follows if
Linear search is used:

```c
int found;
int i;
found = FALSE;
i = 0;
while (!found && i<n) {
    if (!strcmp(word, table[i].key)) // strcmp returns zero
        // if strings are same
        { // if strings are same
        found = TRUE;
        index = i;
        }
    else i++;
}
return found;
```
The efficiency of Linear search is now considered. In assessing the efficiency of an algorithm it is usual to count the number of occurrences of some typical operation as a function of the size of the problem.

In searching the major operation is the number of comparisons of the searched-for key against the table entries and the problem size is taken to be the number of entries in the table. Hence in Linear search of a table with \( n \) entries we have:

**Best Case** Find at first place - **one comparison**

**Worst Case** Find at \( n \)th place or not at all - **\( n \) comparisons**

**Average Case** It is shown below that this case takes - \((n+1)/2\) comparisons

In considering the average case there are \( n \) cases that can occur, i.e. find at the first place, the second place, the third place and so on up to the \( n \)th place. If found at the \( i \)th place then \( i \) comparisons are required. Hence the average number of comparisons over these \( n \) cases is:

\[
\text{average} = \frac{(1+2+3+\ldots+n)}{n} = \frac{(n+1)/2}{2}
\]

where the result was used that \( 1+2+3\ldots+n \) is equal to \( n(n+1)/2 \).

Hence Linear Search is an **order(\( n \))** process - i.e. it takes a time proportional to the number of entries. For example if \( n \) was doubled then, on average, the search would take twice as long.

### 6.2 Improving the performance - Binary Search

It is obvious that Linear search would be impractical in searching something as large as a telephone directory. Imagine how long it might take to find someone’s number if the names were in a random order! Of course the key to searching a telephone directory efficiently is that it is **ordered** in alphabetic order. Thus the directory is first opened at roughly the correct place, then by a single comparison it is easily decided if the required entry is before or after that place. A smaller distance is then moved in the requisite direction and the process is repeated with smaller and smaller steps until the required entry is found. This method at each step eliminates a large part of the directory from the search - hence the efficiency.

To apply this idea to searching a table in computing the table must be **ordered** on the search key. In what follows it is assumed that the ordering is in increasing order. The Binary search algorithm starts by comparing the
required key against the mid-point of the table, if it is equal to that entry then the key has been found and the algorithm exits. If it is less than that entry it is in the first half of the table otherwise it is in the second half of the table. The algorithm is then repeated on the appropriate half of the table. By continually repeating this halving process the entry containing the key is found or the search area becomes zero size - in which case the entry is not in the table. This algorithm is known as **Binary Search**.

The Binary Search Algorithm can be described as follows:

```plaintext
set partition to be searched to whole table.
set found to false.
while not found and partition size > 0
{
    find the midpoint of partition.
    if required key is before entry at midpoint
        then partition becomes first half of partition.
    else if required key is after entry at mid point
        then partition becomes second half of partition.
    else
        found is set to true.
}
if found then return index of midpoint.
```

A partition within the table can be represented by the index of its first element and the index of its last element, in the following algorithm the variables `low` and `high`. The body of the search function in section 6.1 is then written as follows if Binary search is used:

```plaintext
low = 0;
high = n-1; // initial partition is whole table
found = FALSE;
while (!found and low <= high)
{
    mid = (low + high) / 2;
    if (strcmp(word,table[mid].key) > 0)
        low = mid + 1;
    else
        if (strcmp(word,table[mid].key) < 0)
            high = mid - 1;
        else
            found = true;
}
if (found)
    index = mid;
else
    index = low;
```
Remember that `strcmp(a,b)` returns -1 if $a < b$ and +1 if $a > b$. Note that if a matching entry is not found then `index` indicates the location at which a new entry should be inserted to maintain the sorted order.

The efficiency of Binary search is not so easy to derive as was the efficiency of Linear search. It can be shown that the number of comparisons required to find an entry is at worst (and on average) approximately $\log_2(n)$, where $n$ is the number of entries. The log to the base 2 of $n$ is the power to which 2 must be raised to obtain $n$. Thus the log to the base 2 of 1024 is 10 since 2 to the power of 10 is 1024. Thus Binary search is an $O(\log_2(n))$ process.

**Compare** the number of comparisons taken on average for Binary search and Linear search on a table of size 1024.

```
n = 1024
Linear search = (n+1)/2 -> 512 comparisons
Binary search = log2(n) -> 10 comparisons
```

The above example shows how superior Binary search is to Linear search for large tables, in this case a factor of fifty times better! However remember that before Binary search can be used the table must be in sorted order.

### 6.3 Insertion and Deletion

The above shows that Binary search is much more efficient at searching tables than Linear search. Thus when a table has already been set up and placed in sorted order then Binary search is preferable. However there is usually a cost involved in sorting the table in the first place. If only a few accesses are going to be made to the table then the extra efficiency of Binary search may not give as much saving as the time spent in sorting the table. So Binary search is best (in this form) for applications where the table may be set up once in sorted order and is very rarely changed thereafter.

For example in writing a compiler one of the first tasks is to carry out lexical analysis. The task of lexical analysis is to enter the program and to output to the next stage of the compiler (the Parser - which checks the syntax) a string of tokens which indicate, in a shorthand form, the symbols which are being input. Thus the token for a reserved word would indicate that this was a reserved word and which reserved word it was. Thus the reserved word `while` might be passed on to the Parser as a code indicating it is a reserved word together with, say, a numeric code which identifies which reserved word it is. The method the lexical analyser uses to do this is to
maintain a table of the reserved words and as each word is identified in the
input stream it is searched for in this table, if found in the table then it is
a reserved word and the index in the table can be used as its code. This
table only has to be set up in sorted order once and will never change so
Binary search would be a suitable method to use in searching the reserved
word table.

However many applications require search operations to take place on
tables which are continually changing due to the insertion and deletion of
entries. Many database applications fall into this category.

6.3.1 Insertion

In Linear search the table is unsorted so it does not matter where in the
table the new entry is placed. So it is placed at the $n+1$ position if there
are $n$ entries already in the table. This only takes one operation so it is an
order(1) operation, that is it takes a constant time whatever the size of the
table.

In Binary search the new element has to be placed into its correct posi-
tion in the table to maintain the sorted order. Using Binary search provides
the index of where the new entry should go. This takes time proportional
to $\log_2(n)$. However once the correct place has been found all the elements
from that place up must be moved to generate a spare space to put the new
entry in. This can be, at worst, $n$ moves when the correct place is the first
place. Hence insertion is an order($n$) operation. Thus if many insertions
take place the time saved in using Binary search is negated by the much
longer time required to carry out the insertions.

6.3.2 Deletion

In both Linear and Binary search deletion will entail moving elements to
fill up the empty slot. In the worst case this could mean moving $n$ entries.
Hence in both methods deletion is an order($n$) operation.

The above shows that the greater efficiency of Binary search may be lost
in applications where regular updates (insert or delete) are required to the
table. If however the number of accesses is much greater than the number
of insertions and deletions then Binary search is much preferable, especially
when $n$ is large.

There are many other algorithms for carrying out table searching but
they are all more advanced and are not considered in this course.

6.4 Conclusions

Binary search is much more efficient than Linear Search when searching
a large table. However the table must be sorted. If many insertions and
deletions take place then the need to move elements in the table to maintain
the sorted order negates the efficiency of the search process. Hence Binary
search is only used on tables in which the entries in the table are rarely
changed.

In situations where insertions are frequent then Linear search may be
used if the table is never very large.

Techniques for handling large tables with frequent insertion and deletion
are not covered in this elementary treatment of searching.

6.5 Summary

- Linear search, starting at the head of a list and examining each entry
  is inefficient for long lists.

- If the list is ordered, binary search is much more efficient.

- Insertion and deletion from sorted lists are expensive operations in-
  volving moving other entries.

6.6 Review Questions

1. A table has 128 entries and it is searched for a particular entry. The
   method of search used is Linear search. What is the least number of
   comparisons that would be made in this search? What is the most
   that could be made? What would happen if the entry was not in the
   table?

2. If the table in the above question was searched using Binary search
   what would be the average number of comparisons made? \(\log_2(128)\)
   is seven).

3. Assume that moving an entry in a table takes twice as long as a com-
   parison of a key with the key field of an entry. An entry is searched
   for in the previous table using linear search and if not found is placed
   at the end of the table (count as a movement). What is the total cost
   of adding this entry to the table?

4. Binary search is used to search the above table (now sorted) and if
   the key is not found the new entry is placed in the table so that the
   sorted order is maintained. Assuming that the correct place for the
   new entry is in the middle of the table what is the total cost of adding
   this new entry to the table? How does this compare with using linear
   search?
6.7 Exercises

1. A piece of text is available in a file. Words in the text contain only letters of the alphabet. Write a function which given an input file stream as a parameter will return the next word in the text as an output parameter. If there is no next word then false should be returned by the function, otherwise true is returned. You will have to provide a data type for a word. See the example program in Section 2.1.2. Write a driver program to test this function on a piece of text, printing each word as it is read from the file.

2. Make up a struct definition to hold a word (as you defined it above) and an integer count. Write a search function as defined in Section 6.1 to search for a word in an array of such structs. Test it by expanding the driver you wrote for the previous exercise to place each word as it is entered into an array of such structs. The search function can then be tested by entering words from the keyboard and searching the array of structs.

3. A concordance of a piece of text usually contains (among other things) a list of the distinct words in a text together with a count of how many times each word occurs. Basically as each word of the text is entered it has to be decided whether this word has been seen before. If it has then its count of occurrences is incremented otherwise it has to be noted. Write an algorithm for this process that uses functions to obtain the next word in the text, to search for a word in a list, to amend an entry in a list and to place a new entry in the list. Now implement the algorithm as a program - you have already done most of the implementation above. Test your program on a suitable text file. You should remember that the word ‘the’ is the same as the word ‘The’ so store all words as all lower case or all upper case.
Lesson 7

Sorting tables

As seen in Lesson 6 Binary search is much more efficient than Linear search. However it requires that the table be in some sorted order. So sorting algorithms are now considered briefly.

Sorting arranges items in a table according to some predefined ordering. The ordering is usually alphabetic or numeric. For simplicity in describing the algorithms it is assumed that the tables are only arrays of integers and that the requirement is to sort into ascending numeric order.

Sorting algorithms can be split into two categories:

**Simple** Are easily understood and are characterised by taking time proportional to \( n^2 \), where \( n \) is the number of items to be sorted.

**Advanced** More complex and harder to understand and are characterised by taking time proportional to \( n \log_2(n) \), where \( n \) is the number of items to be sorted.

The difference in these two efficiency ratings is considerable if \( n \) is large. For example if \( n \) is 8000 then \( n^2 \) is about 64 million while \( n \log_2(n) \) is about 104 thousand. That is a factor of about 615, the difference between a second and 10 minutes!

7.1 Simple Algorithms

7.1.1 Selection Sort

Selection sort carries out a sequence of passes over the table. At the first pass an entry is selected on some criteria and placed in the correct position in the table. The possible criteria for selecting an element are to pick the smallest or pick the largest. If the smallest is chosen then, for sorting in ascending order, the correct position to put it is at the beginning of the table. Now that the correct entry is in the first place in the table the process is repeated on the remaining entries. Once this has been repeated \( n-1 \) times the \( n-1 \)
The smallest entries are in the first $n-1$ places which leaves the largest element in the last place. Thus only $n-1$ passes are required. The algorithm can be described as follows:

```c
for i from 1 to n-1
{
    Find smallest element in ith to nth entries.
    Exchange this element with ith entry.
}
```

For example consider the following example of selection sort being carried out on a sample set of data:

9 2 5 7 4 8 on pass 1 look for smallest in 1st to 6th
swap 2nd with first giving
2 9 5 7 4 8 on pass 2 look for smallest in 2nd to 6th
swap 5th with second giving
2 4 5 7 9 8 on pass 3 look for smallest in 3rd to 6th
swap 3rd with third giving
2 4 5 7 9 8 on pass 4 look for smallest in 4th to 6th
swap 4th with fourth giving
2 4 5 7 9 8 on pass 5 look for smallest in 5th to 6th
swap 5th with 6th giving
2 4 5 7 8 9 sorted.

The algorithm is now written in C++. Of course the limits 1 to $n-1$ in the above now become 0 to $n-2$ because array subscripts start at 0.

```c
for (i = 0; i < n-1; i++)
{
    // find smallest entry in ith to n-1 th place
    // p is subscript of smallest entry yet found
    p = i;
    for (j = i+1; j < n; j++)
        if (a[j]<a[p])
            p = j;
    // exchange pth element with ith element
    t = a[p];
    a[p] = a[i];
    a[i] = t;
}
```

Note that when finding the smallest element the index $p$ is the index of the smallest value found so far. Only after the position of the minimum value has been found does the actual interchange take place.
In assessing the efficiency of sorting routines it is usual to count the number of comparisons made between keys and the number of exchanges of table entries.

To count the number of comparisons in the above algorithm note that the outer loop is traversed \( n-1 \) times. At the \( i \)th execution of the outer loop the inner loop is executed as \( j \) takes values from \( i+1 \) to \( n-1 \), that is \( n-i-1 \) times. Hence the total number of comparisons made is

\[
(n-1)+(n-2)+\ldots+3+2+1 = \frac{n(n-1)}{2}
\]

So on comparisons selection sort is order\((n^2)\). Thus doubling the number of entries to be sorted will mean that four times as many comparisons will be made.

Inside the outer loop entries are exchanged three times. Thus the total number of exchanges is \(3(n-1)\), i.e. of order\((n)\).

### 7.1.2 Exchange Sort

Exchange sorts attempt to improve the ordering by comparing elements in pairs and interchanging them if they are not in sorted order. This operation is repeated until the table is sorted. Algorithms differ in how they systematically choose the two elements to be compared. The following method compares adjacent elements and is known as **Bubble Sort**.

The first and second elements are compared and interchanged if necessary, this places them in the correct positions relative to each other. However, is the second element in the correct position relative to the third element? This suggests a loop in which first entry is compared with second, second with third, third with fourth etc. This could be written in C++ as follows:

```cpp
for (j = 0; j < n-1; j++)
    if (a[j] > a[j+1])
        interchange(a[j],a[j+1])
```

What does this achieve? It ‘bubbles’ the largest element to the end of the table. Consider

\[
\begin{align*}
9 & 2 5 4 7 8 \text{ compare 1st & 2nd } \\
2 & 9 5 4 7 8 \text{ compare 2nd & 3rd } \\
2 & 5 9 4 7 8 \text{ compare 3rd & 4th } \\
2 & 5 4 9 7 8 \text{ compare 4th & 5th } \\
2 & 5 4 7 9 8 \text{ compare 5th & 6th } \\
2 & 5 4 7 8 9
\end{align*}
\]

This has now bubbled the largest value up to the last place. This process is now repeated with the first \( n-1 \) elements to place the second-largest element in the second last place, and so on until the table is sorted. Again
only \( n-1 \) passes are required since once the \( n-1 \) largest values are in the last \( n-1 \) places the first value must be the smallest. The sorting process can now be written in C++:

```cpp
for (i = 0; i < n-1; i++)
    for (j = 0; j < n-i-1; j++)
        if (a[j] > a[j+1])
            {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
```

Applying the algorithm to the example above the following intermediate configurations are obtained:

- Initially: 9 2 5 4 7 8
  - \( i = 0 \): 2 5 4 7 8 9
  - \( i = 1 \): 2 4 5 7 8 9
  - \( i = 2 \): 2 4 5 7 8 9  **on this pass no exchanges made**
  - \( i = 3 \): 2 4 5 7 8 9
  - \( i = 4 \): 2 4 5 7 8 9

In the above set of data the table was already sorted before the end of the algorithm. The following version of the algorithm is much better in that it recognises when the table is sorted and does not carry out any superfluous passes.

```cpp
sorted = FALSE;
i = 0;
while (i < n-1 && !sorted)
    {
        sorted = TRUE;
i++;
        for (j = 0; j < n-i; j++)
            if (a[j] > a[j+1])
                {
                    t = a[j];
                    a[j] = a[j+1];
                    a[j+1] = t;
                    sorted = FALSE;
                }
    }
```

Here the marker `sorted` is set to `true` at the beginning of each pass, if no interchanges take place at that pass it means that the list is already in
sorted order. Also if no interchanges take place then sorted is never set to false hence the outer loop is terminated because !sorted is false. If an interchange takes place then sorted is set to false and the next pass is carried out.

In its original form the loop within a loop structure again gives order($n^2$) performance on comparisons. However the exchanges are now within the innermost loop and are also order($n^2$) in the worst case.

Hence in general it is better to use Selection sort because Bubble Sort carries out many more exchanges. However if it is likely that only a few elements are out of sorted order in a table then it is worth while using the improved version of Bubble Sort since it is likely to terminate after a few passes.

### 7.1.3 Insertion Sort

Assume that at some stage of the sorting process the first $i$ entries in a table are in sorted order. The basic idea of insertion sort is to insert the $i+1$ th entry into its correct place in the first $i$ entries. This increases the length of the sorted section to $i+1$. Initially $i$ is set to 1, obviously the first number on its own constitutes a sorted list. After $n-1$ iterations of this process the table is sorted. Hence the algorithmic description:

```plaintext
for i from 1 to n-1
    { insert i+1 th entry in correct place in first to i+1 th positions.
    }
```

This is illustrated below, where the sorted initial section at each stage is underlined:

```
9 2 5 4 7 8       i = 1
2 9 5 4 7 8       i = 2
2 5 9 4 7 8       i = 3
2 4 5 9 7 8       i = 4
2 4 5 7 9 8       i = 5
2 4 5 7 8 9
```

The insertion process is carried out by taking a copy of the $i+1$th element and repeatedly comparing it with the $i$th, $i-1$th, ... entries until an entry is found which is smaller, at each stage the larger entry is moved one place to the right. When the correct place is found then the copied entry is inserted in the free space generated by the previous movements of entries to the right. This insertion process can be written in C++ as follows:

```plaintext
x = a[i+1];       // take copy
```
```plaintext
j = i;      // start scan at ith position  
while (x < a[j])  
  {  
    a[j+1] = a[j];  // move an entry right  
    j--;         // scan backward  
  }  
  a[j+1] = x;  // place copied entry  

and is illustrated below:

2 5 9 4       4 to be inserted
2 5 9         4 < 9 move 9 to right
2 4 9 5       4 < 5 move 5 to right
2 4 5 9       4 > 2 so insert 4

There is a difficulty with the above however, if a[i+1] is less than all
a[0..i] then the loop will never terminate. One way to get round this
case is to treat j=0 as a special case in the loop. Hence the algorithm becomes:

for (i = 0; i < n-1; i++)  
{  
  x = a[i+1];  // take copy  
  j = i;      // start scan at ith position  
  while (j > 0 && x < a[j])
    {  
      a[j+1] = a[j];  // move an entry right  
      j--;         // scan backward  
    }  
  if (j == 0)       // deal with special case  
    {if (x < a[0])
      {  
        j = -1;  
        a[1] = a[0];  // move entry right  
      }  
    }  
  a[j+1] = x;  // place copy  
}

The algorithm is illustrated below at an intermediate step in the process:

4 5 9 2 7 8   2 to be inserted
4 5 9 9 7 8   2<9 and j=2, move 9 right
4 5 9 9 7 8   2<5 and j=1, move 5 right
2 4 5 9 7 8   j=0, 2 < 4, move 4 right and
              put 2 into a[0].
```

83
In assessing the efficiency of Insertion sort it is noted that again the outer loop is executed \( n-1 \) times while executing the inner loop 1, 2, 3, \( \ldots \) \( n-1 \) times. The exchanges all take place inside the inner loop as well so the algorithm is order\( (n^2) \) on comparisons and exchanges.

### 7.2 An Advanced algorithm - Quicksort

This algorithm uses a ‘Divide and Conquer’ approach similar to that used in Binary search. Initially it chooses an element of the array, say \( a[i] \), and then partitions the array as follows:

\[
\begin{array}{c|c|c}
\text{elements < } a[i] & a[i] & \text{elements > } a[i] \\
\end{array}
\]

where a partition process has been used to place all the elements less than \( a[i] \) at the beginning of the table and all the elements greater than \( a[i] \) at the end of the table. This process is now repeated on the left and right partitions. In turn the process is repeated on the partitions generated at this step etc. until all partitions are reduced to size 1 and are thus sorted. A possible rule for the selection of an element is to merely choose the first one. This rule is used in the following example:

6 3 12 9 7 14 15 4 8 2 5 1 10 13 11 initially

selecting 6 as pivot gives left and right partitions

(3 4 2 5 1) (6) (12 9 7 14 15 8 10 13 11)

selecting 3 and 12 as pivots gives:

(2 1) (3) (4 5) (6) (9 7 8 10 11) (12) (14 15 13)

selecting first elements again as pivots gives:

(1) (2) () (3) () (4) (5) (6) (7 8) (9) (10 11) (12) (13) (14) (15)

and repeating

(1) (2) () (3) () (4) (5) () (7) (8) (9) () (10) (11) (12) (13) (14) (15)

where () indicates an empty partition.

To implement the partition process a scan is carried out from the left until an element greater than or equal to the chosen element is found, then a scan from the right until an element less than the chosen element is found. These two elements are then interchanged. The left/right scans are then continued from the points they halted at looking for more elements to interchange. The process stops when the two scans cross. Hence the function:
void partition(int a[],
    // Array to be sorted
    int first,    // first index of partition
    int last,     // last index of partition
    int& pivindex // final position of pivot
)

// partitions the elements from first to last in
// the array t so that all the elements less than
// the first element are placed at the beginning
// of the array and all the elements greater than
// the first element are placed at the end of the
// array. The original element appears between the
// two partitions at the position indicated by
// pivindex.
{
    int pivot;
    int up,down;
    pivot = a[first];
    up = first;
    down = last;
    do
    {
        // scan for element > pivot from left
        while (a[up]<=pivot && up<last) up++;
        // scan for element <= pivot from right
        while (a[down]>pivot) down--;
        // exchange
        if (up<down) exchange(a[up],a[down]);
    }
    while (up<down);
    // put pivot in between partitions
    exchange(a[first],a[down]);
    pivindex = down;
}

The function exchange is easily written as:

void exchange(int& a, int& b)
    // Exchanges the values of the variables a and b.
{
    int t;
    t = a;
    a = b;
    b = t;
The function quicksort can now be written very simply if quicksort is allowed to call itself. This technique is called recursion and will be covered in a later lesson.

```c
void quicksort(int t[],
    int first,
    int last)
    // Sorts the first to last elements of the
    // int array t using the quicksort method.
    // If there is only one element in the array
    // then nothing is done. After partitioning
    // the elements into two sets, those less than
    // a pivotal element and these greater than the
    // pivotal element, quicksort is called
    // recursively to sort each set in turn.
{
    int pivindex;
    if (first<last)
    {
        partition(t,first,last,pivindex);
        quicksort(t,first,pivindex-1);
        quicksort(t,pivindex+1,last);
    }
}
```

To sort the first \( n \) entries in the array `table` the function `quicksort` would be called by:

```c
quicksort(table,0,n-1);
```

This algorithm is usually considered to be the fastest sorting method available. Its performance does depend however upon a choice of pivotal element which leads to two roughly equal in size partitions. The choice of the first element may not always accomplish this (consider a table that is already sorted). Various other methods are used, for example choosing the middle element of a small sorted subset of the partition.

On average the performance is order(\( n \log_2(n) \)). As \( n \) increases the difference between this order of performance and that of an order(\( n^2 \)) algorithm is startling. For example in a test the times taken to sort 10,000 real numbers were as follows:

<table>
<thead>
<tr>
<th>Method</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection sort</td>
<td>187.18</td>
</tr>
<tr>
<td>Quicksort</td>
<td>0.83</td>
</tr>
</tbody>
</table>
7.3 Conclusions

If the number of entries to be sorted is small (say < 50) it probably makes very little difference whether a simple or advanced sorting method is used. However as the number of entries increases the more advanced methods rapidly outperform the simple methods and consequently it is essential they are used rather than the simple methods.

If a simple method is used it is usually best to use Selection sort because of its order(n) performance on exchanges. If the table entries are large structures then this reduction in the number of exchanges compared with the other simple methods is crucial. If however it is likely that a few entries are out of place by a small amount then the use of Bubble sort may be worthwhile. It must be the version of Bubble sort that includes tests for a sorted table occurring at an intermediate step.

The algorithms in this Lesson are easily re-written in terms of the table data structure defined in Lesson 6. The main change is to replace the numeric comparisons by calls of strcmp on the keys. Only minor changes are required if sorting into descending order is required.

7.4 Summary

- Sorting data into order can dramatically improve the performance of various searching algorithms.

- Choice of sorting algorithm should depend on the quantity of data to be sorted, and knowledge about whether the data is currently sorted or not.

- If the items to be sorted are large, for example are structs, then it may be advantageous to use a method that minimises the number of exchanges of items.

- Exchange sorts interchange items until the data is sorted

- Insertion sorts gradually increase the size of the sorted list, by inserting new data items in the correct place

- Simple sorting methods take a time proportional to \( n^2 \) where \( n \) is the number of items to be sorted. Advanced sorting methods take a time proportional to \( n \log_2(n) \).

7.5 Review Questions

1. Trace the execution of Selection sort on the following list:

   8 17 12 18 23 5 7
2. Trace the execution of Bubble and Insertion sorts on the list in the previous question.

3. Change the selection sort algorithm so that it sorts into descending order rather than ascending order.

4. How would the selection sort written in Section 7.1.1 be changed to sort an array of structs as defined in Lesson 6.

7.6 Exercises

For all the following exercises, you can generate some test data using the random function used in Introduction to Computing I, Lesson 16, to generate a large array of randomly distributed real numbers between 0 and 1.

It is probably worth saving the output to a file, and generating a sorted copy of the same file, so that you can experiment with testing the algorithms on sorted and unsorted data.

1. Funnel sort is a version of bubble sort which alternately bubbles the maximum element towards one end of the array and the minimum element towards the other end of the array. Implement funnel sort, and compare it with bubble sort.

2. Modify the version of quicksort given in this Lesson so that it chooses the element to pivot on by taking the first, last and middle element of the section of array to partition, and then uses the median value of these three as its pivot.

3. In the Exercises 6.7 you should have written a concordance program. Now write a function which will sort the array of structs holding word and its count of occurrences into alphabetic order on the word. Use any sorting method you like.

4. In producing a concordance it is usual to ignore common words like ‘a’, ‘the’, ‘and’, ‘but’ etc. Extend your concordance program so that it does not count such words. Commence by initialising an array of strings to, say, about 20 to 30 common articles, conjunctions, prepositions etc. arranged in alphabetic order. Now write a binary search function to search for a word in this array. Amend your concordance program to only count words which do not occur in this list of trivial words.
Lesson 8

More on Functions

Before proceeding to consider the Object-Oriented aspects of C++ a few more features of functions are covered. Firstly a brief review of functions is given.

In Introduction to Computing I the process of Top-down Design was covered. In top-down design a problem was split into sub-tasks, which in turn were further split into still lower level tasks. Functions were then specified for each sub-task. This specification indicated what input information was required by the sub-task and what output information it returned. Also specified was the relationship required between the input and output. Once the function was fully specified then it could be implemented as a C++ function. Once a function has been implemented then the programmer using it should not have to know anything about how the function has been implemented internally. All the information the programmer requires should be available in the function prototype and a header comment that describes the effect of the function. Thus the function should be written as if it is a black box into which information is passed and out of which information is returned without it being necessary or possible to see what is happening inside. Writing and using functions in this fashion is often called the principle of Procedural Abstraction.

All information required by the function is passed to the function via input parameters, these are called value parameters. Information can be passed back from the function by reference parameters or by using the return statement to return a value. When defining the function formal parameters are used in writing the function code. When the function is called these formal parameters are replaced by the actual parameters or arguments of the call. These actual parameters must agree in number, type and order with that defined in the function prototype which must precede the use of the function. If the function returns a value then the function must have the same type as the value returned, if no value is returned then the function should have type void.
Many simple functions return a single value which is a function of the parameters of the function. In this case the formal parameters will be value parameters and the function will have the type of the value returned. If several values must be returned then they will be returned by the use of reference parameters while input to the function will be by the use of value parameters. In some cases it is necessary to return an error indication when input parameter values are invalid, in this situation it is usual to return the error indicator as the value of the function and any other information by reference parameters.

8.1 Function Overloading

Sometimes it would be natural to use the same name for functions which carry out similar tasks but on different input information. For example there might be a need for functions that carry out such tasks as

- Find the maximum of two doubles
- Find the maximum of two ints
- Find the maximum of three doubles
- Find the maximum of three ints
- Find the maximum value in an array of n doubles
- Find the maximum value in an array of n ints

A natural name for each of these function might be max. C++ allows the use of the same function name for several functions which have different input parameters. This is called overloading the function name. For example some of the functions above could be implemented as follows:

```cpp
double max( double x1, double x2)
{
    if (x1 > x2)
        return x1;
    else
        return x2;
}

int max(int x1,int x2)
{
    if (x1 > x2)
        return x1;
    else
        return x2;
}
```
return x2;
}

double max( double x1, double x2, double x3)
{
    if (x1 > x2)
    {
        if (x1 > x3)
            return x1;
        else
            return x3;
    }
    else
    {
        if (x2 > x3)
            return x2;
        else
            return x3;
    }
}

int max( int a[], int n)
{
    int i, big;
    big = a[0];
    for (i=1; i<n; i++)
        if (a[i] > big)
            big = a[i];
    return big;
}

maxfunc.cpp

How does the compiler know which function to use when one of the above functions is called in a program? If you study the above definitions you will note that they all differ in their parameter lists, either in the number of parameters and/or the types of the parameters. The compiler checks the number and types of the parameters in the function call and then uses the function whose parameters match in number and type. This obviously means that you cannot overload functions which take exactly the same number and types of parameters. The type returned by the function is not sufficient on its own to distinguish between two functions which have identical parameters.

Thus if you wish to overload a function name then the function definitions must have a different number of formal parameters and/or
formal parameters of different types.

8.2 Recursion

In Lesson 7 the sorting method quicksort was covered. The final version of this algorithm was:

```c
void quicksort(int t[],
               int first,
               int last)
// Sorts the first to last elements of the
// int array t using the quicksort method.
// If there is only one element in the array
// then nothing is done. After partitioning
// the elements into two sets, those less than
// a pivotal element and these greater than the
// pivotal element, quicksort is called
// recursively to sort each set in turn.
{
  int pivindex;
  if (first<last)
  {
    partition(t,first,last,pivindex);
    quicksort(t,first,pivindex-1);
    quicksort(t,pivindex+1,last);
  }
}
```

The basic method used in the quicksort algorithm is to initially partition the array to be sorted into two sections, one partition holding all the entries less than some pivotal value and the other holding all the entries greater than the pivotal value. The first partition is placed before the pivotal value and the second is placed after the pivotal value. Thus the arrangement of the array after the partition stage is:

```
elements < pivot  | pivot  | elements > pivot
```

This obviously carries out a rough sorting of the array, all that now remains is to sort the two partitions. Sorting a partition is of course identical to the original problem but is carried out on a smaller partition. Hence the original algorithm can be used at this lower level thus making the algorithm recursive. If when designing an algorithm it is split into sub-tasks which are ‘smaller’ versions of the original task then the problem can be solved using recursive functions.

In C++ a function is recursive if its definition contains a call of itself.
Some of the algorithms that have already been studied can be decomposed into sub-tasks that would allow recursion to be used in their solution. For example:

1. Linear Search. In searching an array consider the first element, if it is the required key then exit else carry out linear search on the remainder of the array.

2. Binary Search. Look at the middle element, if it is the required key then exit else carry out binary search on the appropriate half of the array.

In neither of the above algorithms is there any efficiency advantage in using a recursive solution compared with the iterative solutions already considered. However in designing algorithms it is worth keeping the possibility of a recursive solution in mind since it often leads to much quicker algorithm development.

Another feature of the quicksort algorithm that must always be present in any recursive solution is the existence of a base case. In any recursive algorithm there must be a way of halting the recursion process, otherwise the algorithm would recurse for ever! In quicksort the base case that terminates the algorithm is the recognition that the partition to be sorted has only one element, hence nothing further need be done. This case is recognised by the partition boundaries first and last being equal, hence only when first is less than last is the recursive call of quicksort carried out.

Hence the requirements for the development of a recursive algorithm are:

- In splitting the problem down into sub-problems the sub-problems must be the same as the original problem only they are ‘smaller’ in some sense.
- There must exist a base case which is simply solved and which is used to terminate the recursion.

### 8.2.1 Example : Binary search

Binary search is easily recast as a recursive algorithm as follows:

```c
// Search for key in array table from table[low] // to table[high]. If found return true and index // of key otherwise return false.
if low > high // zero length section to search
{
    set index to low.
    return false.
}
```
else
{
    set mid to (low+high)/2.
    if key is equal to table[mid]
    then
    {
        set index to mid.
        return true.
    }
    else
    if key is < table[mid]
        search from table[low] to table[mid-1].
    else
        search from table[mid+1] to table[high].
}

Termination is guaranteed in this algorithm. If the key exists in the table then it will be found at one of the intermediate mid-points or at the final mid-point when low is equal to high and the algorithm will terminate. If it is not in the table then the situation low equal to high will be reached, the comparison will fail and depending upon the supplied key either low will become high+1 or high will become low-1, in either case low will become greater than high and the base case of no entries left to search will have been attained and the algorithm terminates.

The corresponding C++ function is:

```c++
int binsearch(char word[],
    entrytype table[],
    int low,
    int high,
    int& index
)
// Search for word in array table from table[low] // to table[high]. If found return true and index // of key. If not found then index returns the // position in which the entry should be placed to // maintain sorted order in the table.
{
    int mid;
    if (low > high) // zero length section to search
    {
        index = low;
        return FALSE;
    }
    else
```
{  
    mid = (low+high)/2;
    if (!strcmp(word, table[mid].key))  
    {   
        index = mid;
        return TRUE;
    }
    else  
    if (strcmp(word,table[mid].key) < 0)  
        return binsearch(word,table,low,mid-1,index);  
    else  
        return binsearch(word,table,mid+1,high,index);
}

8.2.2 Example: Towers of Hanoi

A game called the Tower of Hanoi is often used as an example of a recursive solution to a problem. The specification of the problem is:

A platform carries three posts. On the left-hand post $n$ discs with a hole in the centre are placed. The discs are all different diameters and are arranged so that they are in descending order of diameter with the largest diameter disc at the base. The object of the game is to move the tower of discs from the left-hand post to the right-hand post moving only one disc at each step and never placing a larger disc on a smaller disc. The middle post is used as an intermediate position. Develop a program which outputs the sequence of moves required to solve the $n$ disc problem.

For example if $n$ is 3 the solution is:

move disc on top of left post to right post
move disc on top of left post to middle post
move disc on top of right post to middle post
move disc on top of left post to right post
move disc on top of middle post to left post
move disc on top of middle post to right post
move disc on top of left post to right post

To obtain a recursive solution to this problem it is necessary to find a solution which solves the problem for $n$ discs in terms of a solution that solves the problem for some smaller number of discs. The solution is obvious if the number of discs is 1, thus this will be the base case for the solution. Because the largest disc is at the bottom of the tower on the left-hand disc
and it has to end up at the bottom of the tower on the right hand disc there must be a stage when all the discs have been removed from above it and the right-hand post is empty so that the largest disc can be moved to the right-hand post. At that stage the remaining $n-1$ discs must be on the middle post in their correct order and can now be moved on to the right-hand post to complete the solution. This suggests the recursive solution:

Move $n-1$ discs from left post to middle post.
move the largest disc from left post to right post.
move the $n-1$ discs from middle post to right post.

Thus if a function `hanoi` was written which specified as parameters

1. The number of discs to be moved — $n$
2. The post from which they have to be moved — `from`
3. The post to which they must be moved — `to`
4. The post which can be used as an intermediate post — `using`

then the solution could be written in C++ as:

```cpp
hanoi(n-1,left,middle,right);
hanoi(1,left,right,middle);
hanoi(n-1,middle,right,left);
```

Hence the solution written as a recursive function is:

```cpp
void hanoi(int n, post left, post right, post middle)
{
    if (n == 1)
    {
        cout << "Move disc from ";
        outpost(left);
        cout << " post to ";
        outpost(right);
        cout << " post " << endl;
    }
    else
    {
        hanoi(n-1,left,middle,right);
        hanoi(1,left,right,middle);
        hanoi(n-1,middle,right,left);
    }
}
```

`hanoi.cpp`

96
This assumes that an enumerated type called \texttt{post} has been declared by:

\begin{verbatim}
enum post = (left,middle,right);
\end{verbatim}

and that a function \texttt{outpost} has been written to output a value of type \texttt{post}.

8.2.3 Example: Cells in a connected group

This example considers a problem which is difficult to solve without recursion. The problem is as follows:

A two-dimensional array represents a grid of cells, each cell may be empty or full. Full cells that are connected to each other constitute a connected group. Two cells are connected if they are adjacent to each other horizontally, vertically or diagonally. Given the co-ordinates \((i,j)\) of a cell within the grid the problem is to find the number of cells in the connected group which starts on \((i,j)\).

The grid below shows an example, the cells marked with an asterisk are full.

\[
\begin{array}{cccccccc}
0 & * & * & * & 4 & 5 & 6 & 7 \\
1 & * & * & * & * & * & * & * \\
2 & * & * & * & * & * & * & * \\
3 & * & * & * & * & * & * & * \\
4 & * & * & * & * & * & * & * \\
5 & * & * & * & * & * & * & * \\
6 & * & * & * & * & * & * & * \\
7 & * & * & * & * & * & * & * \\
\end{array}
\]

If the number of cells in a connected group is given by \texttt{cell\_count(i,j)} then it is easily seen that in the above example:

\[
\begin{align*}
\text{cell\_count}(0,0) &= 8 \\
\text{cell\_count}(1,6) &= 1 \\
\text{cell\_count}(3,3) &= 0 \\
\text{cell\_count}(5,6) &= 11 \\
\text{cell\_count}(7,0) &= 4 \\
\end{align*}
\]

If the \((i,j)\) cell is full then \texttt{cell\_count}(i,j) is obviously one plus the number of full cells connected to it. To find the number of full cells connected to it requires the evaluation of \texttt{cell\_count()} for each of the surrounding eight cells, hence the recursive rule that:
cell_count(i,j) = 1 + cell_count(i,j-1) 
    + cell_count(i,j+1) 
    + cell_count(i+1,j+1) 
    + cell_count(i+1,j) 
    + cell_count(i+1,j-1) 
    + cell_count(i-1,j+1) 
    + cell_count(i-1,j) 
    + cell_count(i-1,j-1);

However this is not correct. When evaluating cell_count(i,j-1) the original (i,j) cell will be counted again since it is connected to the (i,j-1) cell. Hence as each cell is visited it is marked as having been ‘visited’ and will not be counted again. There are two obvious base case for this recursion, namely:

1. If a cell is empty or has been visited previously then return cell_count(i,j) as zero.
2. If (i,j) does not give an address within the array then return cell_count(i,j) as zero.

Hence the complete algorithm:

if i or j out of range
    then return zero
else
    if cell(i,j) is not full
        then return zero
    else
        set cell(i,j) to ‘visited’.
        cell_count(i,j) = 1 + cell_count(i,j-1) 
                        + cell_count(i,j+1) 
                        + cell_count(i+1,j+1) 
                        + cell_count(i+1,j) 
                        + cell_count(i+1,j-1) 
                        + cell_count(i-1,j+1) 
                        + cell_count(i-1,j) 
                        + cell_count(i-1,j-1).
}

The above algorithm changes the contents of the original cell array - this is bad practice. Hence after evaluating the count all cells marked ‘visited’ are reset to ‘full’. This cannot be done inside the recursive function cell_count(i,j) since cell(i,j) has to remain marked as ‘visited’ while each of the neighbours is examined, otherwise it will be counted in each of the eight calls of cell_count(). Hence the following C++ version of the above algorithm which uses two functions:
enum marker {empty, full, visited}; // state of cell
const int max = 20; // max size of cell array
typedef marker state[max][max]; // type definition for an // array of cells

int cell_count(state s, int i, int j, int dim)

    // Returns the number of full cells connected to // cell(i,j) in the dim*dim section of the state // array s. Calls the recursive function // count_cellij to compute the result and resets // the elements marked as visited by that function // back to full.
    {
        int count;
        int k, l;
        // get count of connected cells
        count = count_cellij(s, i, j, dim);
        // restore state array s to original state
        for (k=0; k<dim; k++)
            for (l=0; l<dim; l++)
                if (s[k][l] == visited)
                    s[k][l] = full;
        return count;
    }

int count_cellij(state s, int i, int j, int dim)

    // Returns the number of full cells connected to // cell(i,j) in the dim*dim section of the state // array s. Is implemented recursively.
    {
        int count;
        int k, l;
        if ((i<0 || i>=dim || j<0 || j>=dim))
            count = 0; // off edge of cell array
        else
            if (s[i][j] != full)
                count = 0; // cell empty or visited
            else
                { // cell full
                    s[i][j] = visited;
                    count = 1 + count_cellij(s, i, j+1, dim)
                }
Note how the state array s and the size of the section of it that is used (dim) have been passed as parameters. dim must be less than or equal to the constant max.

8.3 Summary

- A programmer using a function should not require to know any details of how the function is programmed. It should only be necessary to know the function prototype and to know what operation the function carries out. This is the Principle of Procedural Abstraction.

- It is allowable to have more than one function definition associated with a function name. This is called Function Overloading.

- When a function name is overloaded then the different function definitions must differ in number of parameters or in the types of the parameters.

- A C++ function may contain a call to itself. Such a function is called a Recursive function.

- For a problem to be solved by a recursive method it must be possible to specify its solution in terms of the solution of smaller versions of the same problem.

- In a recursive solution to a problem there must exist a base case which is simply solved and which will eventually be reached to allow termination of the solution.
8.4 Review Questions

1. You have written a function. If another programmer is to use this function what information would you have to pass to this programmer?

2. If a function name is to be overloaded what must be true of the parameters in the various definitions?

3. The following are prototypes for two functions:
   
   float example(float);
   float example(float, float);

   Which definition would be used in the following function call? Why?
   
   a = example(3.0, 5.0);

4. Write definitions for functions that have the name area, one should return the area of a circle and the other the area of a rectangle. Could you similarly write a function to return the area of a square? If not why not?

5. A function is defined as follows:

   void test(int i)
   {
     if (i<10)
       cout << i;
     else
     {
       cout << i % 10;
       test(i/10);
     }
   }

   It is assumed that i is positive. What would be output by the call test(12345)?

6. Write an iterative version of the function test defined in the previous question.

7. Write a recursive function to return the value of $x^n$ for positive n. Hint $x^n$ is $x^{n-1} \times x$ and $x^0$ is 1.

8. Amend your solution to the previous question to allow n to be positive or negative.
8.5 Exercises

1. \( C(n,r) \) is the number of combinations of \( r \) objects chosen from \( n \) objects. It can be shown that:

\[
C(n,n) = C(n,0) = 1 \\
C(n,r) = C(n-1,r-1) + C(n-1,r)
\]

Write a recursive function for \( C(n,r) \) and test it by producing Pascal’s triangle which is:

\[
\begin{array}{ccccccc}
C(0,0) & & & & & & \\
C(1,0) & C(1,1) & & & & & \\
C(2,0) & C(2,1) & C(2,2) & & & & \\
C(3,0) & C(3,1) & C(3,2) & C(3,3) & & & \\
& & & & & & \\
\end{array}
\]

2. The Greatest Common Divisor (GCD) of two integers \( u \) and \( v \) is the largest integer which divides exactly into \( u \) and \( v \). For example the two integers 36 and 27 have a GCD of 9. Euclid’s algorithm for calculating the GCD of two positive integers can be written in the following recursive form:

\[
GCD(u,0) = u \\
GCD(u,v) = GCD(v, u \mod v)
\]

where \( u \mod v \) is the remainder when \( u \) is divided by \( v \). Write a recursive function to return the GCD of two long int as a long int and test it. The GCD of 3846313 and 6856471 is 167231 which makes a non-trivial test.

3. Write a program to find a way through a maze. The specification of the maze should be read in from a file. The first quantity in the file is an integer giving the size of the maze, \( n \). This is followed by \( n \) lines of \( n \) characters representing the maze. The maze should be defined as an \( n\times n \) array of characters, each character being an ‘X’ or a ‘0’. You are not allowed to enter the ‘X’ squares and the entry point is the \([0][0]\) square (which is a ‘0’) and the desired exit point is the \([n-1][n-1]\) point (also a ‘0’). Only horizontal and vertical moves are allowed. If no path can be found then a suitable message should be output, if a path is found then a list of the squares on the path should be output. A recursive function should be written for this program. Make up a file to define a suitable maze for testing.
Lesson 9

Introduction to Classes

In Lesson 8 the concept of Procedural abstraction was introduced. This concept requires that functions should be written so that they can be used by a programmer who only has access to a function prototype and an explanatory comment detailing what the function does. What the programmer does not have to know are any details of the implementation of the function. Thus the programmer has an abstract view of the function, hence the name Procedural Abstraction. Procedural Abstraction allows well-tested and debugged code to be re-used with all the attendant benefits in increased productivity and correctness that this brings.

In Lesson 5 the concept of a struct that could collect several different related entities into a single object was covered. This allows a programmer to set up his or her own data types and by supplying functions that operate upon these data types allows new Abstract Data Types (ADT) to be implemented. Again the programmer using these types need not know any details of the implementation. In this case an abstract idea of the data type is all the programmer need know. In this case the concept of Data Abstraction is being used.

In Lesson 5 the ADT money was set up using structs together with appropriate functions. In practice this is not the best way to implement an ADT in C++. Any user who knew the implementation of the struct could directly access the members of the struct and bypass the functions that operate upon the struct. This is undesirable since a user might introduce errors by doing this, hence the advantage of well-tested and debugged functions is lost.

What is required is to apply both Procedural and Data Abstraction together and bind data and the functions that operate upon the data together in a self-contained entity whose internal details are unavailable to the user and with which the user can only interact by using a well-defined functional interface. An instance of such an entity that contains both data and functions to operate upon the data is called an object.
C++ allows a programmer to define a new data type which includes definitions of what type of data it holds and what functions operate on this data. Such a data type is called a class. Objects are then variables of this class.

9.1 Classes in C++

Classes have already been encountered in C++. For example I/O streams such as cin and cout are defined as classes. No knowledge of the details of the implementation of these classes is needed to be able to use them. All use of streams is via a defined functional interface, for example the functions get(), put() etc. have been covered.

A class is a data type similar to a struct in that it is a collection of related members, however the members can be member variables or member functions. Thus a class definition looks very like a struct definition. The member functions define the interface of the class to the outside world.

Several steps are required in designing and building a class. These are:

1. Definition - decide what member variables should be held by the class and design the functional interface to the class, i.e. decide on the member functions.
2. Implementation - implement the member functions.
3. Use - use the class in programs.

The above steps are now carried out using a simple example for illustration. The example used is that of building a class Date to represent a calendar date.

9.1.1 Definition of the class Date

For simplicity it is assumed that a date will be held in the form of a day number, a month number and a year number as in 23/7/1986. Such a simple form for a date could be defined by a struct as follows:

```cpp
struct Date
{
    int day;
    int month;
    int year;
};
```

To turn this struct definition into a class definition requires that some member functions should be added. For the moment member functions to print out a variable of type Date and to update a variable of type Date to
the next day are the only operations considered. The class definition for
`Date` is then:

```cpp
class Date
{
    public:
    // member functions
    void print();  // prints date
    void next_day();  // sets date to next day
    // member variables
    int day;
    int month;
    int year;
};
```

The keyword `public` indicates that all subsequently declared member
variables and functions are accessible from outside the class. This is not
necessarily desirable and will be considered later. For the moment the mem-
ber variables `day`, `month` and `year` are accessed just as if they were member
variables of a struct by use of the dot operator. The prototypes only are
given for the member functions. A `Date` object can now be declared and
given a value as follows:

```cpp
Date today;

today.day = 19;
today.month = 2;
today.year = 1997;
```

A member function is called in the same manner as a member variable
is accessed, it is appended, using the dot operator, to the object (variable)
to which it is to be applied.

For example the functions `print()` and `next_day` could be applied to
the variable `today` defined above as illustrated below:

```cpp
    // Print the date
    cout << "The date today is ";
today.print();
    // step forward to next day and print date
    today.next_day();
cout << "The date tomorrow is ";
today.print();
```

It is important to note that at the moment the class has only been
defined, the implementation of the class has still to be done. However the
definition is all that the user has to know.
A supplied type such as `double` might be implemented differently on different computers, but one would expect that a program should be able to be moved from one computer to another and still work correctly. Similarly any data types that you define should behave as well as the pre-defined types. Any change in the implementation of the type should not affect the working of the program that uses the type. Thus in the definition of the type `Date` above it is bad practice to allow the user to directly access the member variables `day`, `month` and `year`. For example the implementation may be changed at a later time to store a date internally as the number of days elapsed since some standard date. This can make some functions on dates much easier to write. If this was done then any program which directly accessed the member variables would no longer work. Hence all details of the implementation of a class should be hidden from the user and all access to the member variables of the class should be by the functional interface.

To make any member variable or member function inaccessible to the user it is listed after the keyword `private`. All members that appear after the label `private` will be private members. However a user might still have a requirement to use the actual `day`, `month` and `year` values of a date. These can be supplied by adding public accessor functions to the class definition that return these values no matter what the internal representation. Hence the amended definition below for the class `Date`.

```cpp
class Date
{
    public:
        // member functions
    void print();    // prints date
    void next_day(); // sets date to next day
        // accessor functions
    int get_day();   // returns day
    int get_month(); // returns month
    int get_year();  // returns year

    private:
        // member variables
    int day;
    int month;
    int year;
};
```

A program can now access the private member variables by the use of the accessor functions as follows:

```cpp
if (today.get_day() == 31
    && today.get_month() == 12
```
```cpp
kk today.get_year() == 1999)
cout << "Its the start of a new millenium tomorrow!";
```

Now even if the implementation of the class is changed get_day etc. will still return the correct values to the program.

At present there is no way the user can set a variable of type Date to a particular value. The method used above of assigning values directly to the member variables cannot be used now that the member variables are private. One solution to this problem would be to include a member function in the class definition somewhat as follows:

```cpp
void set_date(int d, int m, int y);
```

This is quite feasible, however it is better to allow a variable to be initialised when it is declared just as other variables are. This can be done by the use of Constructors. A constructor is called automatically whenever a variable of the class is declared. It is defined in the same manner as other member functions except that

1. The name of the constructor is the same as the name of the class. Thus in the Date class any constructor for the class would be called Date.
2. No type is given for a constructor (not even void) and no value is returned by a constructor.

Constructors are normally declared as public member functions. Also it is possible to overload constructors to allow more than one form of initialisation. The definition of the class Date can now be extended by the addition of constructors as follows:

```cpp
class Date
{
    public:
        // Constructors
        // initialise to 1/1/1900
        Date();
        // initialise to day/month/year
        Date(int dd, int mm, int yy);
        // member functions
        void print(); // prints date
        void next_day(); // sets date to next day
        // accessor functions
        int get_day(); // returns day
        int get_month(); // returns month
        int get_year(); // returns year
```
private:
    int day;
    int month;
    int year;
    int leap_year(); // returns true if year
        // is a leap year
};

The constructor Date() without any parameters initialises the declared Date variable to 1/1/1900. This constructor without any arguments is called the Default Constructor and need not carry out any initialisation at all. The constructor Date(int dd, int mm, int yy) initialises the declared Date variable to the specified date. Hence the declaration:

    Date date, today(20,2,1997),birthday(4,1,1896);

would initialise the variable date to 1/1/1900, the variable today to 20/2/1997 and the variable birthday to 4/1/1896.

A constructor is called whenever an object is declared. After an object has been declared it can be called again to give a new value to the object. This is done using an assignment statement which assigns the constructor to the object. For example the object today defined above could be re-assigned the value 24/3/1998 by

    today = Date(24,3,1998);

When the default constructor is used in an assignment the null argument list must be specified. Thus today could be reset to 1/1/90 by using the default constructor as follows:

    today = Date();

However when the default constructor is used in a declaration the brackets and null list are not specified. It should be noted that an object can be assigned to another object of the same class. The assigned object must have a value at that point. Thus with the definitions as used in the example above it is allowable to write:

    date = today;

It is normal to place the definition of a class in an include file for inclusion in any program that requires the class. Usually the file takes the name of the class. Thus a suitable include file Date.h for the class Date would be:

    // File Date.h
    // Definition file for the class Date


```cpp
// which represents a simple date class
// IEA February 1997

#ifndef DATE_H
#define DATE_H

class Date
{
    public:
    // Constructors
    // initialise to 1/1/1900
    Date();
    // initialise to dd/mm/yy
    Date(int dd, int mm, int yy);
    // member functions
    void print(); // prints date
    void next_day(); // sets date to next day
    // Accessor functions
    int get_day(); // returns day
    int get_month(); // returns month
    int get_year(); // returns year

    private:
    int day;
    int month;
    int year;
    int leap_year(); // returns true if year is a leap
    // year. Required in next_day()
};

#endif

date.h

9.1.2 Implementation of the class Date

Once the class has been defined as in the previous section it must be implemented. Implementing a class requires writing the member functions. The definition of the type Date only provides prototypes for the member functions. These functions must now be written. Writing member functions is exactly the same as writing any other function with one small addition. There may be many classes which have member functions with the same name. For example many classes may have a print() member function. When defining a member function it is therefore necessary to indicate to
```
which class it belongs, this is done using the **Scope Resolution operator ::**. Thus the heading for the member function `int get_day()` is:

```cpp
int Date::get_day()
```

which indicates that the function `get_day()` belongs to the class `Date`. The general form of a Member function Definition is:

```
Return_type Class_name::Function_name(Parameter_list)
{
   Function_body
}
```

For example the implementation of the function `print()` is:

```cpp
void Date::print()
{
   cout << day << '/'
       << month << '/'
       << year;
}
```

For example the implementation of the function `print()` is:

```cpp
void Date::print()
{
   cout << day << '/'
       << month << '/'
       << year;
}
```

**date.cpp**

When writing the member function `next_day()` it is necessary to test whether a year is a leap year or not. Hence the function `leap_year()` has been added. It has been made a private function and hence can only be used by the member functions of the class `Date`.

The implementation of the member functions is now placed in a separate file, say `Date.cpp`, for separate compilation. A suitable implementation file is given below:

```cpp
// File Date.cpp
// Implementation file for the Date class
// IEA February 1997

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

// Constructor functions
Date::Date(int dd, int mm, int yy)
{
   day = dd;
   month = mm;
```
year = yy;
}

Date::Date()
{
    day = 1;
    month = 1;
    year = 1900;
}

// Accessor functions

int Date::get_day()
{
    return day;
}

int Date::get_month()
{
    return month;
}

int Date::get_year()
{
    return year;
}

// Other functions

void Date::print()
{
    cout << day << '/'
         << month << '/'
         << year;
}

void Date::next_day()
{
    int days_in_month;
    day++;
    switch (month)
    {
        case 1 :
        case 3 :
case 5:
case 7:
case 8:
case 10:
case 12: days_in_month = 31;
    break;
case 4:
case 6:
case 9:
case 11: days_in_month = 30;
    break;
case 2: if (leap_year())
    days_in_month = 29;
    else days_in_month = 28;
    break;
}
if (day > days_in_month)
{
    day = 1;
    month++;
    if (month > 12)
    {
        month = 1;
        year++;
    }
}

// Private function

int Date::leap_year()
{
    if (year % 400 == 0)
        return 1;
    else if (year % 100 == 0)
        return 0;
    else if (year % 4 == 0)
        return 1;
    else return 0;
}

date.cpp
9.1.3 Use of the class Date

Once the file Date.cpp has been compiled then it can be linked to a user program that uses the class Date. The user program must include the file Date.h that defines the class. To access the member of the class the dot operator is used as in structs. This applies to member variables and member functions. This is illustrated in the following simple test program to test the class Date.

// A test program for the class Date
// IEA February 1997

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

void main()
{
    Date d1, d2(21,8,34);
    int d,m,y;
    // check initialisation is correct
    cout << "First date is ";
    d1.print();
    cout << endl;
    cout << "Second date is ";
    d2.print();
    cout << endl;
    // check accessor functions
    d = d2.get_day();
    m = d2.get_month();
    y = d2.get_year();
    cout << endl << d << " "
    << m << " 
    << y << endl;
    // check next day is correct
    d2.next_day();
    cout << "Next day is ";
    d2.print();
    cout << endl;
    // check setting date by assignment
    // and another check on next day.
    d1 = Date(31,12,1999);
    d1.next_day();
    cout << "Day after 31/12/1999 is ";
    d1.print();
    cout << endl;
}
9.1.4 Parameters and return values of member functions

The formal parameters of a function may have a class type. In particular
the type of a formal parameter of a member function may be that of the
class within which the member function is defined.

Thus in the Date example the public member function next_day() could
have been overloaded to a version which takes a Date object as a parameter.
The prototype for this function would be:

```c++
void next_day(Date date);  // Returns a Date which is
// the day after date.
```

This function could then be implemented as follows:

```c++
void Date::next_day(Date date)
{
    int days_in_month;
    day = date.day;
    month = date.month;
    year = date.year;
    day++;
    switch (month)
    {
    case 1 :
    case 3 :
    case 5 :
    case 7 :
    case 8 :
    case 10:
    case 12: days_in_month = 31;
              break;
    case 4 :
    case 6 :
    case 9 : case 11: days_in_month = 30;
              break;
    case 2 : if (leap_year())
              days_in_month = 29;
              else days_in_month = 28;
              break;
    }
}
```
if (day > days_in_month)
{
    day = 1;
    month++;
    if (month > 12)
    {
        month = 1;
        year++;
    }
}

In a program where d1 and d2 were declared as objects of type `Date`, d2 could be set to the day after d1 by:

```cpp
d2.next_day(d1);
```

It is also possible for a function to return an object as its value, that is the type of a function may be a class. For example a user of the `Date` class may wish to produce a `Date` object which is one week after a given date. A function could be written as follows to carry out this task:

```cpp
Date week_later(Date date)
{
    // This function returns the date a week
    // later than date.
    Date temp;
    int days_in_month;
    int day,month,year;
    day = date.get_day();
    month = date.get_month();
    year = date.get_year();
    day+= 7;
    switch (month)
    {
        case 1 :
        case 3 :
        case 5 :
        case 7 :
        case 8 :
        case 10:
        case 12: days_in_month = 31;
            break;
        default: days_in_month = 30;
            break;
    }
    day += days_in_month;
    month = (month + 1) % 12;
    year = (year + 1) % 100;
    return Date(day, month, year);  
}
```
case 4 :
case 6 :
case 9 :
case 11: days_in_month = 30;
    break;
case 2 : if (leap_year(year))
    days_in_month = 29;
    else days_in_month = 28;
    break;
}
if (day > days_in_month)
{
    day = day-days_in_month;
    month++;
    if (month > 12)
    {
        month = 1;
        year++;
    }
}
temp = Date(day,month,year);
return temp;
}

Note that the user would now have to provide a function leap_year(int) since the leap_year() function in the class Date is private. This function could then be called in the user’s program as follows:

d2 = week_later(d1);

If such a function was required frequently then it would be much better if it was included as a member function of the class Date. The addition of this new member function would not affect the operation of any existing programs that used the Date type, they would merely have to be re-compiled with the new implementation file for the class Date.

9.2 Summary

• An object is a variable that has both functions and data associated with it.

• A class is a collection of members which may be variables or functions.

• Objects are variables whose type is a class.
• **Member variables** and **Member functions** of a class may be **public** or **private**.

• **Public members** of a class may be accessed from outside the class.

• **Private members** of a class cannot be accessed from outside the class. They can only be used inside the definition of member functions.

• Member variables are normally labelled as **private** and member functions are labelled as **public**.

• Functions may have parameters of a class type and may return a value of a class type.

• A **Constructor** member function of a class is called automatically when an object of the class is declared. The constructor member function initialises the object.

• The constructor member function must have the same name as the class. The constructor member function name may be overloaded to give various forms of initialisation.

• The constructor member function without any arguments is called the **Default Constructor**. A Default Constructor should always be included in a class definition.

• The **dot operator** is used to access all public members of a class, both variables and functions.

### 9.3 Review Questions

Assume the following class definition:

```cpp
class point
{
    public:
        point();
        point(float x, float y);
        float get_x();
        float get_y();
        int dist(float x, float y);
    private:
        float x;
        float y;
};
```

and the following definitions in a main program:
float d;
point p1,p2;

1. Identify in the class definition above the constructor function(s), member functions and member variables of the class point.

2. Which function is the Default Constructor in the above class definition?

3. Which of the following statements are allowed in the main program?

```cpp
point pnt(1.6,7.3);
point p3(1,5);
point p();
p3 = point(6.1,8.4);
p1 = point();
d = p1.x - p2.x;
p1 = p2;
if (p1 == p2)
    cout << "points are same";
    cout << "x co-ord is " << p1.get_x();
```

For those that are not allowable write down a correct statement that has the desired effect.

4. Write the implementation of the member functions get_x and dist.

5. When using a class to implement an ADT should the member variables be public or private? How about the member functions?

6. Why is the scope resolution operator :: required in the implementation of a member function?

### 9.4 Exercises

1. A rational number is any number that can be written as the ratio of two integers. For example 1/3, 22/7, -4/5 etc. Write a class definition for a class called `rational_number` which represents a rational number as an ordered pair of integers. Include a default constructor which initialises a `rational_number` to 0/1 and a constructor which initialises a `rational_number` to a/b when the supplied arguments are a and b. Adopt the convention that negative `rational_numbers` will be represented by the ordered pair a/b with the numerator a being negative and the denominator b positive. Positive `rational_numbers` will always have both numerator and denominator positive. Define the numerator and denominator as `long ints`.

The following member functions should be provided:
(a) Entry of a `rational_number` as two integers.

(b) Output of a `rational_number` in the form `a/b`, where `a` and `b` are integers (a possibly negative).

(c) The arithmetic operations of add, subtract, multiply, divide which are defined as follows:

\[
\begin{align*}
\frac{a}{b} + \frac{c}{d} &= \frac{ad + bc}{bd} \\
\frac{a}{b} - \frac{c}{d} &= \frac{ad - bc}{bd} \\
\frac{a}{b} \times \frac{c}{d} &= \frac{ac}{bd} \\
\frac{a}{b} \div \frac{c}{d} &= \frac{ad}{bc}
\end{align*}
\]

Write a suitable definition file for the `rational_number` class and an implementation file. Compile your implementation file. Now write a simple driver program that tests each of the member functions with simple numbers.

2. Write a program which uses the class `rational_number` to produce `n` (input by user) successive rational approximations to the constant `e` which is defined as the sum of the infinite series

\[
e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!} + \cdots
\]

Note that `n!` is `1 \times 2 \times 3 \times 4 \cdots \times (n-1) \times n`. `0!` is 1. Produce the successive approximations by summing the first 1, 2, 3, 4 \ldots terms of the series in turn using the `rational_number` type.

3. Your results from the previous question show that the numbers involved rapidly become very large. This frequently happens when using rational arithmetic. However the numerator and the denominator often have a common factor which if removed decreases this growth. Extend your member functions to remove any common factors in the results they produce. Use the Greatest Common Divisor function that you wrote in Exercise 2 to find the factor. Make this function a private member of the class `rational_number`. Remember that this function only takes positive parameters.

Now re-run your program of the previous exercise. Note the difference in the size of the numbers produced. Also note that this change in the implementation of the class did not require any change in the application using the class.
Lesson 10

Dynamic Allocation of Arrays and Objects

So far in the course, the size of an array has been known at the time that the program was compiled. This is inconvenient in many problems. In this Lesson, the facilities for allocating arrays with sizes which depend on values calculated by the program, or read in to the program as part of the data will be outlined.

10.1 Dynamic Allocation of Arrays

Consider a program to read the number of elements in a vector, read two vectors, each with that number of components, and print their sum. A program to do this in a simple manner is:

```c
const int maximum_vector_size = 1000000;

do double vec1[maximum_vector_size];

do double vec2[maximum_vector_size];

do double sum[maximum_vector_size];

void read_vector( double vec[], int size);

void write_vector( double vec[], int size);

void add_vector( double vec[], double vec2[],

do double vec_sum[], int size);

main()
{
    int size;
    cout << "Vector Addition: Enter size of vector: ";
    cin >> size;
```
if ( size > maximum_vector_size )
{
    cout << "Maximum size (" << maximum_vector_size
    << ") exceeded" << endl;
    exit(1);
}

read_vector( vector1, size);
read_vector( vector2, size);
add_vector( vector1, vector2, sum, size);
write_vector( vector2, size);
}

dynamic1.cpp

The definitions of the functions, read_vector, write_vector, and add_vector are left as an exercise for the interested reader. Their exact definition is not important.

Although this program is correct, it suffers from a major drawback. Three very large arrays have been declared. The size of the executable file will be very large also, and it will take a long time to load the program into memory. On the other hand, if we make the arrays smaller, the program may fail because the data specifies a larger array than the one specified when the program was compiled. Notice that we have been careful to check that the size read in is less than the size of the arrays.

This problem can be overcome by only allocating the arrays once the size is known. This dynamic allocation means that if the program only needs a small array, it will not have to reserve a large area of memory that it does not need. Also the executable file will be much smaller. Dynamic allocation of arrays is provided by the new operator. This operator returns a pointer to a variables of the appropriate type, which can be used in the same way as an array. For example:

double *vector1;
double *vector2;
double *sum;

void read_vector( double vec[], int size);
void write_vector( double vec[], int size);
void add_vector( double vec[], double vec2[],
                double vec_sum[], int size);

main()
{

int size;
cout << "Vector Addition: Enter size of vector: ";
cin >> size;
vector1 = new double[size];
vector2 = new double[size];
sum = new double[size];
if ( vector1 == 0 || vector2 == 0 || sum == 0 )
{
    cout << "Error: Insufficient memory for (" << size
        " ) vector" << endl;
    exit(1);
}
read_vector( vector1, size);
read_vector( vector2, size);
add_vector( vector1, vector2, sum, size);
write_vector( vector2, size);
}

**dynamic2.cpp**

There are a number of differences between this version of the program and the previous version.

1. The arrays are now declared as **pointers**. The [] with the size of the array which came after the identifier are replaced with an * before the identifier. Pointer variables store the location of a variable of the appropriate type. A pointer variable can also have the value 0, indicating that it is pointing to no variable. The variable pointed to by the pointer can be accessed using a prefixed *, or by treating it as an array, and addressing it as element 0 of the array. Use of the arrays is the same as before.

2. The space for the arrays is allocated using the **new** operator. After the operator, the type of the array elements is given, (in this case double) and the number of elements required is specified between the [].

3. If there is insufficient memory then the **new** operator will return the value 0. This result should be tested, otherwise some very obscure errors can result.

### 10.1.1 Releasing Memory

In some programs, there is a need to allocate an array which is needed only for part of the program, and after there is no further need for it, to allocate
a different array. If both arrays are large, there is a temptation, particularly if they are of the same type, to use one array for both purposes. Apart from making the program harder to understand, it is not possible to do this if one of the arrays is of type int (say) and the other is double.

The solution is to use the delete operator. If the array vector2 was no longer needed in the program above, one could add a line

    delete[] vector2;

and the memory would be released, allowing another new operation to use it. Observe the [] after the operator; no size is specified, but the brackets must still be specified. It is important to realise that delete does not change the value of the pointer or array vector2; it will still be possible to access the memory that was associated with it. It is an error on the part of the programmer to access memory which has been freed by the use of delete[]. The memory formerly associated with vector2 may have been reallocated to another array by the new operator. Using a pointer which has previously been deleted, will lead to obscure run time errors. In general, it is a good idea to also assign 0 to the deleted variable, so that attempts to use it will be trapped.

10.1.2 Strings

Character strings have already been encountered as arrays of characters. It is common to allocate space for strings dynamically, rather than having a statically allocated character array which will usually be far too large for the string concerned. Recall that a string is specified by giving an array of characters, the last of which is the NULL character, ‘\0’. When reading strings from cin, a white space delimited set of characters will form a string. It is not known in advance how long a string will be, so it is usual to use a fixed (large) char array to store the string at first, and then copy it to the variable required. Thus:

    char long_string_for_input[1000];
    char *word;
    int length;
    ...
    cin >> long_string_for_input;
    length = strlen(long_string_for_word);
    word = new char[length+1]; // leave space for ’\0’ character
    strcpy(word, long_string_for_word);

When manipulating a large number of strings, it is important to remember to use delete so that the allocated memory does not become inaccessible. For example:
char *word, *phrase;
int length;
...
// want to do phrase = word
length = strlen(word);
delete[] phrase;
phrase = new char[length+1];
strcpy(phrase, word);

Without the delete, the memory previously allocated for phrase would remain allocated, but all means of referencing it would have been lost. Memory leaks like this are a major problem in large software systems.

10.2 Summary

- Arrays can be allocated with sizes that are calculated at run time using the new operator. The arguments to new are the type of the array entries, and the number of entries.
- Dynamically allocated arrays are accessed using pointer variables.
- When the memory is no longer required, the delete operator releases it.
- Objects can be allocated dynamically as well. If they are allocated dynamically, then they should be freed using delete as well.

10.3 Review Questions

1. Why is it sometimes better to use a dynamically allocated array than to allocate a very large static array, and check that the actual size needed is less than the size statically allocated?

2. What is wrong with the following segment of code?

```c
double *a;

a = new double[100];
...
for ( i = 0; i < 10; i++) {
    a = new double[100];
    ...
}
```

How about the following?
double *a;

a = new double[100];
...
for ( i = 0; i < 10; i++) {
    ...
    delete[] a;
}
Lesson 11

Further Classes

Classes can be used to encapsulate complex data structures and to ensure that any restrictions on access to the data are enforced. As a simple example, we shall use character strings. In C++ strings are represented by arrays of characters terminated with a null character. All operations on strings are provided by library functions. In general, we do not want to specify a maximum length for the string in advance, so dynamic allocation of the array will be used.

11.1 Data Representation and Operations

For dynamic allocation of arrays, a pointer variable is required to store the allocated array. It should be possible to find out the length of the string, and also to make copies of the string. Other operations that should be possible are extracting a single character and extracting a substring. There should be constructors for the default case when there is no argument. Other constructors will be needed to initialise the string from a string constant, and from another string.

With those considerations in mind, we can give the declaration of the class:

```c++
class string {
    private:
        char *str;
    public:
        // constructors
        string(void);
        string(char *);
        string(string &);
        // operations
        int length(void);
        char character(int);
```
string sub(int, int);
// returns substring starting at first argument
// length second argument
void append(string &);
// append argument to current object
};

mystring.h

This declaration could be written into a header file mystring.h which could be included in every file that needed to use string objects.

11.2 Implementation

The implementation of the string class is of interest in itself. The constructors are designed to ensure that the str member variable is always valid.

    string::string(void) // default constructor
    {
        str = new char[1]; // use array allocation consistently
        str[0] = '\0'; // ensure str is valid string
    }

mystring.cpp

The constructor which initialises from a character string is slightly more complex:

    string::string(char initial_value[]) //
    {
        int length;
        length = strlen(initial_value); // library function
        str = new char[length+1]; // use array allocation consistently
        strcpy(str, initial_value); // ensure str is valid string
    }

mystring.cpp

When another string is to be used to initialise the string, the new object must copy the str member of the argument:
string::string(string &orig_str) // default constructor
{
    int length;
    length = strlen(orig_str.str); // library function
    str = new char[length+1]; // use array allocation consistently
    strcpy(str, orig_str.str); // ensure str is valid string
}

mystring.cpp

Notice how all the constructors ensure that str always refers to a valid char array. A fully robust implementation of a string would check that the value returned by new was not 0 and give an error message if it was.
The length member is straightforward.

int string::length(void)
{
    int len;
    len = strlen(str); // library function
    return len;
}

mystring.cpp

The character member returns the character which is where in the string.

char string::character(int where)
{
    char c;
    if ( where >= 0 && where < length() )
        c = str[where];
    else
        c = '\0';
    return c;
}

mystring.cpp

sub gives a new string object, initialised as a substring of length len
starting at the start character.
string string::sub(int start, int len)
{
    char *newstr;
    newstr = new char[len+1];
    strncpy(newstr, str+start, len);
    newstr[len] = '\0';
    return string(newstr);
}

void string::append(string &to_add)
{
    char *newstr;
    int len;
    len = length() + to_add.length();
    newstr = new char[len+1];
    strcpy(newstr, str);
    strcat(newstr, to_add.str);
    delete [] str;
    str = newstr;
}

mystring.cpp

This implementation does not check that there are indeed len characters after the start character, i.e. that the length of the original string is at least start+len. strncpy is a library function which copies at most the number of characters given by its third parameter. If the end of the second character array is met before that number of characters has been copied, then a null character is appended. If the end of the second array is not met, no null character is copied. The null character is placed there in any case to maintain the integrity of the string operations.

The append operation is slightly different from the others in that it modifies the contents of the string, rather than returning a copy of part of the string.

strcat is a library function which appends its second argument to the end of its first. The append member function ensures that there is sufficient memory allocated in the array, by allocating a new one, and the releases the old one to the system using delete[]. If the memory is not released, programs which run for a long time and make use of a large number of strings will be unable to allocate any new strings when there is no longer any free memory. This loss of free memory because of not returning memory which is no longer accessible is sometimes known as a memory leak.
11.3 Destructors

If an object has dynamically allocated storage associated with it, then this storage will not be automatically recovered when the object becomes unreachable. For example, if the constructor has used `new` to allocate an array as `string` has, when the variable becomes unreachable, the array is still allocated. This would happen if a `string` variable was declared locally to some function.

To allow class implementors to prevent such memory leaks, classes can have a `destructor` which is called whenever the variable goes out of scope, or if the `delete` operator is used with a pointer to the object. The destructor is similar to a constructor, except that its name is the class identifier prefixed with `~` and that it takes no parameters. Thus for the `string` class, a destructor could be

```cpp
string::~string(void)
{
    delete [] str;
}
```

`mystring.cpp`

In general, it is only necessary to provide destructors for objects which use dynamic allocation of memory.

11.4 Inspection Methods

Methods such as `length` do not modify the object: they are known as `inspection` methods. To emphasise the fact that a method does not modify the value of an object, it is good practice to specify `const` between the parameter list of the method and its body (in the definition) or the semicolon (in a declaration). Thus the declaration of `string` might be rewritten

```cpp
class string {
    private:
        char *str;
    public:
        // constructors
        string(void);
        string(const char *);  // const modifier
        string(const string &);
        // destructor
        ~string(void);
        // operations
```
int length(void) const;
char character(const int) const;
string sub(const int, const int)const;
// returns substring starting at first argument
// length second argument
void append(const string &);
// append argument to current object
};

The parameter lists of the methods have also been modified to take full account of any parameters that might be modified. For most simple parameters, such as those of type int or type double, specifying nothing means that no changes can be made to the actual parameter in any case. The power of const is that it can be applied to reference parameters too. If a user defined object, such as string is used as an actual parameter, then whenever it is passed in the default style, a new temporary object will be created as a copy of the actual object. If the object is large, this may cause a significant overhead. This overhead is not incurred if reference parameters are used. The use of const with a reference parameter prevents any changes being made to the object which is being passed as a parameter.

11.5 Mutators

Mutator methods are those which modify the state of the object in some way. Typically, they do not return a value, although they might return an error indication. append is an example of a mutator.

11.6 Example use of String class

Textual analysis was one of the earliest non scientific uses of computers. Linguists (and others) often wish to analyse the structure of literary works by counting word frequencies, word lengths, sentence lengths, and similar statistics about the text.

The general structure of language is far too difficult to treat with simple tools such as we will build, but it provides an easy to understand example. The program to be built will analyse the text from a file, calculating the frequency of words and sentences of different lengths, and printing out the sentences starting on a new line. For simplicity, it will be assumed that a sentence always ends with a period as the last character of the last word in the sentence. The data structures required for such a program are straightforward: arrays of counts corresponding to the number of times that a particular word length or sentence length has been encountered, and strings to hold the current word and sentence.
const int maximum_sentence_length = 30;
const int maximum_word_length = 20;

string current_sentence, current_word;

int word_length_frequencies[maximum_word_length+1];
int sentence_length_frequencies[maximum_sentence_length+1];

The main structure of the program will be a loop, reading sentences from the input file until the end of the data is reached. Reading a sentence consists of reading words and adding them to the current sentence until a word ending in a full stop is encountered.

void analyse( istream &instream)
{
    int sentence_length, sentence_ended, word_length;
    char new_word[5*maximum_word_length];
    while ( ! instream.eof() )
    {
        sentence_ended = FALSE;
        sentence_length = 0;
        while ( ! sentence_ended )
        {
            instream >> new_word;
            current_word = string(new_word);
            process_word(current_word);
            sentence_length++;
            word_length = current_word.length();
            sentence_ended = current_word.character(word_length-1) == '.';
            current_sentence.append(string(" "));
            current_sentence.append(current_word);
        }
        // output sentence
        cout << "Sentence length " << sentence_length << endl;
        // record length of sentence
        if ( sentence_length > maximum_sentence_length )
            sentence_length = maximum_sentence_length;
        // prevent array subscripting error
        sentence_length_frequencies[sentence_length]++;
    }
    // now print out sentence and word length frequencies
}

testmyst.cpp
Notice that the word is temporarily stored in an array much large than any likely word to be read. Completion of the program is left as an exercise to the reader.

Processing a word is similar to processing a sentence, except that one has to remove any trailing full stop before recording the length of the word.

```cpp
void process_word( const string & w)
{
    int len = w.length();
    if ( w.character(len-1) == '.' )
        len --; // remove . from length calculation
    if ( len > maximum_word_length )
        len = maximum_word_length;
    word_length_frequencies[len]++;
}
```

Notice how the arrays for storing the length frequencies have been declared with +1 in their sizes, so that lengths equal to the maximum can be recorded.

### 11.7 Summary

- Constructors of classes should ensure that any constraints expected of the data represented by the object are satisfied. For example, if an object represents an $X$ then even an otherwise uninitialised $X$ object will have reasonable default properties.

- If dynamic allocation of memory is used in the constructor of an object, then a destructor method to release the memory to the system when the object becomes unreachable. This will be called automatically when the object goes out of scope.

### 11.8 Review Questions

### 11.9 Exercises

1. Complete the program described in section 11.6. You will need to provide a main program which appropriately initialises the input file so that `analyse` can be called with its `instream` parameter set to the input file. `analyse` may need extension to print out the statistics gathered from the text which has been read.
2. Modify the class `string` so that the length of the string is maintained by the object as a separate variable, rather than having to call the library function `strlen` every time.

3. Modify class `string` so that the actual arrays of characters are only allocated in multiples of 10 characters. If an assignment of a shorter string to a longer one is made, then there is not any need to `delete` the array and allocate a new one. Only if the string which is to be copied is larger than the currently allocated length should the `delete` and `new` operations be needed.
Lesson 12

Operator Overloading

Earlier Lessons have demonstrated that functions in C++ can be overloaded. A single identifier can be used for several different functions. Depending on the number and type of parameters with which the function is called, a different implementation is used. It is also possible to overload operators such as +.

Consider the class `rational_number` which was produced as the answer to Exercise 1 in Lesson 9. Its declaration will be something similar to this:

```cpp
class rational_number {
    private:
        long int a, b;
    public:
        // constructors
        rational_number(void);
        rational_number(long int, long int);
        // operations
        rational_number add( rational_number);
        rational_number sub( rational_number);
        rational_number mult( rational_number);
        rational_number div( rational_number);
        rational_number input( istream &);
        rational_number output( ostream &);
        double double_value(void);
        long int numerator(void);
        long int denominator(void);
};
```

`rational.h`

where the operations are the obvious arithmetic operations, and `double_value` returns the best floating point approximation to the value of the `rational_number`. 
The harmonic series in mathematics are is the sum of numbers of the form $1/i$. It is easily shown that the sum of this series increases without limit. The program below calculates the sum of the harmonic series using both double variables, and also rational_numbers. The value of the relevant variables is printed out each time an integral value is passed.

```c++
main()
{
    double harmonic_total, current_fraction;
    rational_number harmonic_sum, current_rational;
    long int i = 0;
    int j;

    cout << "Harmonic series lengths" << endl;
    for ( j = 1; j <= 5; j++ )
    {
        while ( harmonic_sum.double_value() <= j )
        {
            i++;
            current_rational = rational_number( 1, i );
            harmonic_sum = harmonic_sum.add(current_rational);

            current_fraction = 1 / double(i);
            harmonic_total = harmonic_total + current_fraction;
        }
        cout << " Sum to " << i << " terms is " << harmonic_total
             << " (double) " << harmonic_sum.double_value()
             << " (rational = ";
        harmonic_sum.output(cout);
        cout << ")" << endl;
    }
}
```

harmonic.cpp

Although it is interesting to note how early the rational_number objects become unusable because of the overflow of the long int representing the numerator and denominator of the fraction, it is also interesting to observe that the program itself is not too easy to understand. Expressing an addition as .add(y) is not as easy to understand as +y for most people. C++ allows the overloading of operators as well as the overloading of functions. Basically, one provides functions with names operator+, operator-, etc. and the compiler will use those functions for + and - operations involving the appropriate type of object. It is usual also to overload the operator =.
Taking the `rational_number` class declaration given above, we can include the ordinary arithmetic operations

```cpp
class rational_number {
private:
    long int a, b;
public:
    // constructors
    rational_number(void);
    rational_number(long int, long int);
    // operations
    rational_number add( rational_number);
    rational_number sub( rational_number);
    rational_number mult( rational_number);
    rational_number div( rational_number);
    rational_number input( istream &);
    rational_number output( ostream &);
    double double_value(void);
    long int numerator(void);
    long int denominator(void);
    // operator overloading
    rational_number operator +(rational_number);
    rational_number operator -(rational_number);
    rational_number operator *(rational_number);
    rational_number operator /(rational_number);
};
```

The actual definitions of the `operator` functions can call the member functions that have already been written. For example,

```cpp
rational_number rational_number::operator +(rational_number b) {
    return *this.add(b);
}
```

*`this` is always a way of referring to the current object.

The restrictions on the overloading of operators are that the relative priority of the different operators cannot be changed, and also that the operations on the predefined types such as `double`, `int`, etc. cannot be overloaded.

There is no restriction in the language which prevents the implementor from overloading the `-` operator with a function that actually `adds`. Needless to say, this is very bad practice, and leads to programs which are very difficult to maintain.

Unary operators
When unary operators are overloaded as member functions, they match a function with no parameters. That is, an expression of the form \( !X \) is equivalent to the call \( X.\text{operator} \, !() \). The only exceptions to this are the increment and decrement operators \( ++ \) and \( -- \). \( ++X \) is equivalent to \( X.\text{operator}++() \) and \( X++ \) is equivalent to \( X.\text{operator}++(0) \). This allows the prefix and postfix uses of these operators to be separated.

Binary operators

```
new delete () [] * / % + - << >>
< > <= >= != & ~ | & & || ,
+= -= *= /= %= <<= >>= &= |= ^=
```

When binary operators are overloaded as member functions, they match a function with one parameter. That is, an expression of the form \( X \^ Y \) is equivalent to the call \( X.\text{operator} ^{(Y)} \).

Clearly one can create many interesting effects with operator overloading, and make programs easier to understand. However it behooves the programmer to maintain the semantics of the operations so that \( X+Y \) does not actually subtract \( Y \) from \( X \), since that would only add confusion!

One consequence of this is that all operator overloading should return values. In C++’s built in types, even assignment is an expression, returning the value which has been assigned. This allows one to write

```
a = b = c; // equivalent to b = c; a = b;
```

It is important however that any operator of the form \( &= \), or \( @= \) where \( @ \) is one of the binary operations, returns a reference to the invoking class object. For example,

```
rational_number rational_number::operator =(const rational_number &y)
{
    rational_number x;
    x.a = y.a;
    x.b = y.b;
    return x;
}
```

Although syntactically correct leads to code which is not semantically correct, because it actually creates a new \( \text{rational_number} \), and does not modify the left hand operand of the \( = \) operator. The correct version is:

```
rational_number& rational_number::operator =(const rational_number &y)
{
    a = y.a;
    b = y.b;
    return *this;
}
```
12.1 Overloading of Relational Operators

Given the (non-enforceable) restriction that the semantics of the overloaded operators should mimic their non overloaded versions, the relational operators, \(==, \leq, <, \geq, >, \) and \(!=\) should be overloaded to return \(\text{int}\). Using the \texttt{string} class as an example:

```cpp
int string::operator == (const string &s2) const
{
    if (length() != s2.length())
        return 0; // false
    if (strcmp(s, s2.s) == 0)
        return 1; // true
    return 0; // false
}
```

```cpp
int string::operator <= (const string &s2) const
{
    if (strcmp(s, s2.s) <= 0)
        return 1; // true
    return 0; // false
}
```

12.2 Overloading \([\ ]\) subscripting

Overloading the \([\ ]\) operator allows one to provide a “safe” array index. For example, using the \texttt{string} class, access to the \(i\)th character can be granted:

```cpp
char string::operator [](int i) const
{
    if (i < 0 || i >= length())
    {
        cerr << "String indexing" << endl;
        exit(1);
    }
    return s[i];
}
```

This allows us to access the \(i\)th character in a string so that one can write:

```cpp
string name("Wilberforce");
char c = name[3];
```

and have \(c\) take the value \(b\). This version of \([\ ]\) overloading does not allow us to do
name[3] = 'm';

In order to do that, the overloading must return a reference

    char& string::operator [](int i)
    {
      if ( i < 0 || i >= length() )
      {
        cerr << "String indexing" << endl;
        exit(1);
      }
      return s[i];
    }

This introduces another problem, since we can now subvert the type checking of C++

    const string jerrard( "Gerrard");
    jerrard[0] = 'J';

would allow one to change a string which had been declared as a const. The solution is to include both definitions, and the compiler will select the appropriate one for the circumstances.

12.3 Overloading function call ()

The function call () operator can be overridden to provide a natural interpretation of some operations. A simple example is to provide a substring operation.

    string string::operator() (int pos, int length)
    { // checks omitted
      char *sstr = new char[length+1];
      strncpy( sstr, s+pos, length);
      sstr[length] = '\0';
      return string(sstr);
    }

Now the extraction of a substring is straightforward:

    string x, name("Smythe");
    ...
    x = name(4,2);

will set the variable x to the string he. If desired, a number of different overloads can be defined for different parameter signatures.
12.4 Overloading casts

The final type of operator that can be overridden is the cast. Thus one can define an

```cpp
double rational_number::operator double() const
{
    return double_value();
}
```

double_value is a member function returning the best approximation to the rational number. Overenthusiastic use of this facility can lead to problems with ambiguity in function overloading.

12.5 Summary

- Operators can be overloaded to provide operations between objects. The programmer must ensure that the operations defined by the overloaded operators make sense.
- The assignment operators =, += etc. should all return *this to allow the expected effects in multiple assignment statements.

12.6 Exercises

In all cases write a program to test your operators.

1. Overload the relational operators >, <, etc. to operate with rational_number objects.

2. Provide overloaded operators for adding an int to a Date object to give a Date, and also for subtracting two dates to find the number of days between them.
Lesson 13

Object-oriented Design

Previously in this course and its predecessor structured programming methods have been advocated for the design of programs. In this lesson a very elementary and brief introduction to the design of object-oriented programs is given. The approach used in object-oriented design is to first decide upon what objects are required in the application and what operations have to be carried out upon the objects. This does not mean that all the work done on the design of structured algorithms is wasted. When the member functions have to be implemented structured programming methods are once again used.

The object-oriented approach is illustrated by using an object-oriented approach to rewriting an application previously considered using structured programming methods.

13.1 An example

In Lesson 5 a mark processing program was considered in which the marks achieved by a class of students were held in an array of structs. An implementation of this problem is now considered from an Object-Oriented point of view. The Requirement Specification for that mark-processing application is slightly altered to make the application slightly more general. The new Requirement Specification is:

Information on the performance of a class of students is available on a file. The file contains the title of the class in the first line and then in the next line the number of students in the class. These lines are followed by, for each student, the student’s name and the student’s percentage marks in each of some standard number of subjects. The student’s name is given in the order, surname followed by first name then any initials. A name always begins with a letter and can contain any character. The complete name is terminated by a dollar sign. A typical file might be as follows:
The requirement for the program is to print out a class list which lists the students together with their subject marks and their average mark. Any failure should have an asterisk printed alongside the mark. The average marks in each subject should also be produced. It should be possible to produce the output listing in alphabetic order of student name or in descending order of student average mark.

In designing an object-oriented solution to this problem we commence by thinking of the objects and the operations upon the objects that are required. This is a completely different approach to that adopted in the structured programming approach previously used. In the structured programming approach the emphasis was on producing an algorithm first and then deciding upon the data structures to be used.

In this application there are two obvious objects that are required, namely, an object representing an individual student record and an object representing a collection of student objects to represent a class of students. By studying the specification a first attempt may be made at deciding what operations are required upon these objects. This gives the following list:

**Student Object**
- Enter the results for a student
- Evaluate the average mark
- Print the results for a student

**Class Object**
- Enter the results for a class
- Print the results for a class
- Sort the class in alphabetic order
- Sort the class on average mark order
- Evaluate the subject averages

When studying the operations required on the Class Object it becomes clear that to carry out the sorting operations it will be necessary to access some of the private variables of the Student objects, namely the student name and average mark member variables. Similarly to evaluate the average
marks over each subject it will be necessary to have access to the individual marks. Hence access functions must be included for these members. Also it is not strictly necessary to have a separate ‘Evaluate the subject averages’ member function as this can be calculated during input of the class data file. Hence the expanded version of the list of operations:

Student Object Enter the results for a student
Evaluate the average mark
Print the results for a student
Get average mark
Get student name
Get an exam mark

Class Object Enter the results for a class
Print the results for a class
Sort the class in alphabetic order
Sort the class on average mark order

Hence the following file which gives a class definition for a Student_Record:

```c++
// File Student_Record.h
// Definition of the Student_Record class.

#ifndef STUDENT_RECORD_H
#define STUDENT_RECORD_H

#include <iostream.h>
#include <fstream.h>

const int MAXNAMECHARS = 30, // Max chars in student name
    PASS_MARK = 40, // Pass mark for all exams
    SUBJECTS = 4; // Number of subjects taken in
                  // an examination diet.

typedef char nametype[MAXNAMECHARS];

class Student_Record
{

public:
    // Default Constructor
    Student_Record(); // Sets no of subjects to SUBJECTS
    // Member functions
    void enter_record(ifstream&); // enter record from given
                                   // file stream
    void print_record(); // Print record

...
void eval_average(); // evaluate average
int get_mark(int);  // Return a student mark
    // indicated by parameter
int get_av_mark();  // Return average mark
void get_name(nametype); // Returns student name
    // via the parameter

private:
int ns;             // number of subjects
nametype name;     // students name
int marks[SUBJECTS]; // array of students marks
int average;       // student's average mark

    // Private function
void get_name(ifstream&); // enters name from stream
};

#endif

stud_rec.h

Note in the above definition that the number of subjects taken by a
student is set as a constant. Also set as constants are an examination Pass
Mark and the number of characters to be used in the storage of a student
name. If a change to any of these parameters is required in the future then
they may be re-defined in this file.

Before looking at the implementation of the Student_Record class the
definition of a class to define a class of students is considered. This class
will basically hold an array of Student_Record objects. Since the number
of students is not known until run-time this array is allocated storage space
dynamically. The other information to be stored is also not available un-
til run-time, hence no constructor is required. This leads to the following
definition of the class Class_Record:

    // File Class_Record.h
    // Definition of the Class_Record class

#ifndef CLASS_RECORD_H
#define CLASS_RECORD_H

class Class_Record

Since dynamic allocation is used for the array of Student_Records a Destructor function has been included.

These two classes can now be implemented. For example a partial implementation file for the class Student_Record is:

```
// File Student_Record.cpp
// Implementation file for Student_Record class

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <string.h>
#include <ctype.h>
#include "Student_Record.h"

// Constructor function

Student_Record::Student_Record()
{
    // default - Sets up for SUBJECTS subjects.
```
ns = SUBJECTS;
}

// member functions - public

void Student_Record::enter_record(ifstream& ins)
{
    // Enters a student record from file stream ins
    int i;
    get_name(ins); // get student name
    for (i=0; i<ns; i++)
        ins >> marks[i]; // read marks
}

etc.

stud_rec.cpp

The remaining functions are left as an exercise. The class Student_Record

can now be tested by making up a file with a few typical student entries on

it and writing a driver program somewhat as follows:

#include <iostream.h>
#include <fstream.h>
#include "Student_Record.h"

void main()
{
    int i;
    ifstream ins;
    Student_Record student;
    ins.open("examdata");
    if (ins.fail())
    {
        cout << "Failed to open file examdata"
            << endl;
        exit(1);
    }
    for (i=0; i<3; i++)
    {
        student.enter_record(ins);
        student.eval_average();
        student.print_record();
    }
}

teststud.cpp
which assumes that there are three sample student records on the input file. Once the class `Student_Record` has been tested and found to be working the class `Class_Record` can be implemented. Again a portion of the implementation file is given below:

```cpp
// File Class_Record.cpp
// Implementation file for Class_Record class

#include <iostream.h>
#include <fstream.h>
#include <ctype.h>
#include <iomanip.h>
#include <string.h>
#include "Student_Record.h"
#include "Class_Record.h"

// Destructor

Class_Record::~Class_Record()
{
    delete [] records;
}

// Member functions - public

void Class_Record::enter_class(ifstream& ins)
{
    // Enters the data for a class from the stream ins.
    // Evaluates individual student averages and
    // subject averages.
    int i,j;
    char ch;
    int sum[SUBJECTS];
    for (i=0; i<SUBJECTS; i++) sum[i] = 0;
    // get class name - terminated by end of line
    i = 0;
    // Look for first alpha char
    ins.get(ch);
    while (!isalpha(ch)) ins.get(ch);
    // input name until end of line and
    // properly terminate it.
    while (ch != '\n')
    {
        classname[i] = ch;
    }
```
i++;
ins.get(ch);
}
classname[i] = '\0';
// get number of students and allocate space for
// student records.
ins >> nstud;
records = new Student_Record[nstud];
// Enter student records, evaluate student average
// and accumulate sums of subject marks.
for (i=0; i<nstud; i++)
{
    records[i].enter_record(ins);
    records[i].eval_average();
    for (j=0; j<SUBJECTS; j++)
        sum[j] += records[i].get_mark(j);
}
// evaluate subject averages
for (i=0; i<SUBJECTS; i++)
    av_marks[i] = int(float(sum[i])/nstud+0.5);
}

void Class_Record::print_class()
{
    // Outputs the results for a class in tabular form
    int i;
    cout << endl
        << "Results for the class "
        << classname
        << endl << endl;
    for (i=0; i<nstud; i++)
        records[i].print_record();
    cout << endl << endl;
    for (i=0; i<MAXNAMECHARS+2; i++)
        cout << ' ';
    for (i=0; i<SUBJECTS; i++)
        cout << setw(3) << av_marks[i] << " ";
    cout << endl;
}

etc.

clas_rec.cpp

It is worth noting that to implement the member functions it is necessary to go back to structured programming methods in developing the algorithms. For example the comments in the enter_class() function are
a basic algorithmic description that might have been produced before writing the function.

Now that the classes have been defined the main program is trivial. For example to enter the data file and print out the results in alphabetic and in average mark order would require:

```cpp
#include "Class_Record.h"

void main()
{
    Class_Record Form;
    ifstream ins;
    ins.open("examdata");
    if (ins.fail())
    {
        cout << "Couldn’t open file examdata" << endl;
        exit(1);
    }
    Form.enter_class(ins);
    Form.sort_av_mark();
    Form.print_class();
    Form.sort_alpha();
    Form.print_class();
}
```

testclas.cpp

### 13.2 Conclusion

If an extensive package was to be produced for handling of student records the approach detailed in the previous section has many advantages. Keeping all the details about a student in a `Student_Record` object as private members of the object means that any change to the details kept about a student is easily implemented by only changing the definition and the implementation of the `Student_Record` class. No change need be made to existing programs that use the class. For example:

- A Departmental Code Number for each student could be added to the definition of a `Student_Record`. The form of the input file could then be changed to include this information. The functions `print_record()` and `enter_record` could then be suitably amended. No consequent change would be required in any program that used the `Student_Record` class.

152
A final mark is often computed from a combination of an examination mark and a coursework mark, these marks could be entered for each subject in the input file and a final mark computed and stored in the marks member as the result file is processed. Alternately they could be kept separately in the Student_Record and the function get_mark could be amended to calculate the final mark and return it so that consistency is maintained. This ensures compatibility with older programs.

The above examples show the advantages of Object-Oriented programming for improving the Maintainability of software, that is new features can be added to a particular class without all programs that use the class having to be re-written. Also the class can be thoroughly tested and debugged on its own thus enhancing the Reliability of any programs using the class. In writing a general student-record processing program the class Student_Record could be of use in many parts of the application, especially if additional information such as addresses etc. was included in the record. This Reusability of software is a major advantage of Object-Oriented programming.

13.3 Summary

- The starting point for Object-Oriented design is the determination of the objects that are required by the application and the operations that require to be carried out on them.

- Among the advantages of Object-Oriented programming are the enhancement of Maintainability, Reliability and Reusability of Software.

13.4 Review Questions

- If a line was added to the student result file containing the course codes for the classes taken by the class how easy would it be to amend the program to incorporate this extra information and amend the print out so that the course codes are printed as column headings? Identify which classes, member functions etc. would have to be changed.

- If the number of subjects taken by a student was changed what change would have to be made to the class definitions and the program? Would it be possible to incorporate the number of subjects taken as part of the input file and allocate the marks array dynamically?
13.5 Exercises

1. Implement the functions whose implementation has not been given for the Student_Record class and test them.

2. Similarly implement the remaining member functions of the Class_Record class. Now test the whole program.

3. Change the format of the input file so that each single examination mark is replaced by an examination mark and a coursework mark. Also given in the file are two fractions $a$ and $b$. The final mark is $a$ times the examination mark plus $b$ times the coursework mark. Make any required changes to the Student_Record class and re-test the complete program. Note that this should require no changes to the Class_Record class or to the main program.
Lesson 14

Debugging

Debugging is the name given to the process of removing the errors from a program. It is probably because we do not like to admit our mistakes that we call errors in programs bugs — it would be offensive for human beings to confess to making so many mistakes! Debugging is as much a black art as a science, but following some general guidelines can certainly help.

Errors can be split into three broad categories:

- Compile-time errors: indicated by the compiler, with a message indicating that the program is not understood, and cannot be translated. These can be the result of typing errors, such as typing Float instead of float.

- Run-time errors: detected by the program when it is running, and cause the program to terminate, probably with a more or less cryptic message. A common cause of such errors is division by 0.

- Logical errors: cause no messages or obvious problems, but the program does not calculate the required result. A simple example is a program to calculate the number of tins of paint needed to paint a room which returns a fractional number of tins, or worse still, doesn’t print the fractional value, but uses the fractional value in its calculations of cost.

The three categories are now considered in turn, with advice and hints on how to detect them and how to overcome them.

14.1 Compile Time Errors

Compile time errors are detected at the time that the program is translated into machine language. Some programming construct that the programmer has used is not recognised by the compiler. This is usually because the programmer has either made a simple typing slip, or because he or she has
not completely understood the rules of the language as they apply to the programming construct. In some sense, compile time errors are the easiest to correct, because they always result in an error message. Unless the error is so severe that the compiler cannot continue, the translation process will continue with the possibility of detecting further errors. It is possible that the errors detected are not “genuine” errors, but consequences of earlier errors. A simple example of this is the problems caused if there is an error in the declaration of a variable. The compiler will not recognise the variable as such when it is used, and it will complain about “Undeclared variable”. In order to recover from this, many compilers make some assumption about the variable and continue, so that other errors can be found. Typically this will involve assuming that the variable has been declared of type int, and continuing. This assumption may itself cause further compilation errors if an int variable is not acceptable.

A common cause of apparently correct programs causing compilation errors is a comment which is not terminated correctly. Apart from the // comment which terminates at the end of the line, C++ supports a comment that starts with /* and extends until a matching */ character pair. Omitting the closing */ causes havoc! Similar problems occur with missing quotation marks on strings " or characters '.

Typing errors in variable identifiers and keywords can be difficult to spot: watch out for confusion between the letter l (el) and the digit 1 (one). Other characters that often cause errors, particularly when transcribed from handwritten notes are z and 2, s and 5, and o and 0. The compiler also distinguishes between upper and lower case letters. It is not good practice to have identifiers which only differ in the case of some letters.

A good strategy for dealing with compile time errors is to identify the first error reported. Once the first error is identified, usually either in the statement in which it is reported, or possibly the previous statement, one can attempt to decide which errors are consequences of that error. These can safely be ignored. Errors which cannot be explained in this way should be investigated too. Once as many as possible of the errors have been corrected, the program should be recompiled, and the cycle of error correction should be repeated.

Many compilers give warnings as well as error messages. These relate to unusual, but legal, constructs in C++ programs.

14.2 Run Time Errors

Run-time errors are a rather more difficult case. They are simple to detect, in the sense that a message is produced, and the program usually halts. Classic examples of this type of error are attempting to divide by zero, or causing an overflow by multiplying two large numbers together. More sub-
tle examples come from accessing array elements that are beyond the range of subscripts, e.g. accessing element \texttt{a[1000]} when the declaration was for \texttt{a[500]}. Depending on machine architecture, and the implementation of the compiler, this may produce a run time error or not. If the “overstepping” of subscript bounds is small, no message may be produced. If it is large, a message almost certainly will be produced and the program will terminate. Most of these errors are the result of shortcomings in the design or implementation of the program that is running.

In C++ the detection of run-time errors is very much a hit-or-miss affair, and is closely related to the detection and correction of logical errors, so techniques for their detection and prevention will be dealt with below.

### 14.3 Logical Errors

Logical errors are not programming errors; the program runs to completion. The error is that the results of the program do not match the specification. The programmer has to extract information from the program to find out where exactly things go wrong and has to deduce what the error in the design is, so that it can be corrected.

Common logical errors are to exceed the bounds of an array — but to overwrite other data rather than cause a run time error. Assuming that there is a subscript equal to the size of the declared array is the commonest cause of this. Variables which are not initialised can cause problems which depend on the programs that have been executed before, since the memory may have values remaining from previous executions of this or another program. Good practice suggests that initialising all variables with an apparently random value will cause problems such as these to become evident. Variables that have similar identifiers can cause problems too, with the wrong variable being used in some circumstances. It is also common to generate loops that do not terminate correctly.

The debugging process can be split into four main stages:

1. Gather available information.

2. Reproduce the problem, looking for patterns in the program’s misbehaviour and reduce the data to the minimum set required to cause the problem.

3. Form a conjecture about the reason for the misbehaviour, based on an analysis of the patterns.

4. Make necessary changes and verify that these changes have had the desired effect.
14.4 Example: Calculating interest

A simple program is to be written to calculate the interest added to a savings account each year. The interest is added to the balance at the start of the year to give a new starting balance for the next year. The program is to calculate the amount to be added each year, and the final balance. Here is the first attempt:

```cpp
// program to calculate the accumulated funds in a savings account
// PJBK December 1996
#include <iostream.h>
#include <iomanip.h>
main() {
    int i, years;
    double capital, interest;
    int rate;

    cout << "Enter opening capital in account";
    cin >> capital;
    cout << "Enter interest rate";
    cin >> rate;

    cout << setprecision(2) << setiosflags( ios::fixed | ios::showpoint );
    cout << "Initial capital is " << setw(10) << capital << endl;
    for i = 1; i <= years; i++ ) {
        interest = capital * (rate/100);
        capital += interest;
        cout << "Interest added is " << setw(10) << interest << " New principal: " << setw(10) << capital << endl;
    }
}
```

There are a number of errors in this program, which will now be investigated. When the program is compiled, the following messages are output:

```
interest.cpp:24: unterminated string or character constant
interest.cpp:15: possible real start of unterminated constant
```

This is fairly unequivocal, and can be simply corrected by inserting the correct quotation marks " in front of the Enter interest. Can you spot the other error?

When the quotation mark has been replaced, the program is compiled again, and the following messages result
This is a little bit more confusing, but with a little inspection it appears that the for statement lacks a `. When this is added, the program will compile successfully.

When the program runs, no errors occur, but the only output that appears is

Enter opening capital in account 100
Enter interest rate 3
Initial capital is 100.00

and we can see that the variable years is not initialised.

Adding statements

```
cout << "Enter term in years?";
cin >> years;
cout << "Interest at " << rate << "% add for " << years << " years" << endl;
```

interest2.cpp

will correct this problem, but then the output needs to be examined.

It is clearly wrong, as can be seen below

Enter opening capital in account 100
Enter interest rate 3
Enter term in years? 20
Interest 3% add for 20 years
Initial capital is 100.00
Interest added is 0.00 New principal: 100.00
Interest added is 0.00 New principal: 100.00
Interest added is 0.00 New principal: 100.00
Interest added is 0.00 New principal: 100.00
Interest added is 0.00 New principal: 100.00
... and it is clear that something is amiss. Why is interest equal to 0.0 for each year? It is a consequence of the declaration of rate as int, so rate/100 is also int and for all sensible values of rate will evaluate to 0!

Once this problem is corrected the program runs as expected.
14.5 Using const as an aid in Debugging

Often when writing a system you will find that you will start getting errors in your code, will add a list of cout statements to make sure your data is what you think it is, find the error and then delete all the cout statements you put in to find the error. You will be feeling quite smug that you found the error. Half an hour later you will have a different error and will wish that you had never deleted those statements. Within the program code one can put in blocks of code similar to that shown below:

```c
if( DEBUG)
    cout << "The value of variable val at point 10 is: "
    << val << endl;
```

The if statement allows the cout statement following it to be executed depending only on the value of DEBUG. If this is declared const at the head of the file in which we are interested:

```c
const int DEBUG = 1;
```

When the problem has been solved, instead of deleting all the debugging output, redefine the value of DEBUG to be 0, and when the code is compiled none of the cout statements will be included.

14.6 Defensive Programming

The watchword here is “defensive”. The idea is that your program should be written in such a way that it operates safely, that is, it will stop with an error message, rather than continue and later write erroneous data in files etc. The aim is to detect errors before their consequences have a chance to do damage.

Suppose you are writing a program to read student exam marks from a file. Each record in the file consists of a student identifier, followed by a number of class identifiers and the corresponding marks. The program prints the average mark for each class, and also writes the marks into a separate file for each class. When the program is run, the average mark for one class comes out as 150, when the maximum mark possible is 100. Tracking down the cause of this problem is a matter of detective work now. The marks contributing to the average have to be examined. Probably one will find a very large mark somewhere. This mark then has to be cross referenced to the student so that the student record can be checked. Then the student marks need to be corrected, and the program rerun. A defensive attitude to programming would have checked that all the marks associated with a particular student were in range, and would have printed an error message when this was not the case. No update of the files representing the separate classes would be done unless all the input data passed its range test.
Notice that we assumed that this problem was one of data being incorrectly entered. If a program is for interactive use, incorrect data can be queried and reentered immediately. If it is to take most of its input from large data files, then the integrity check can only halt the program, or issue a warning message and ignore the failing record. It might be that the incorrect data entry was a symptom of a program error in the record reading function. Integrity checks as soon as the data is entered allow this error to be isolated.

The inclusion of integrity checks in a program can be assisted by the use of assertions, logical conditions that one expects to be true. If the file assert.h is included in your program, then the function assert(logical-expression) can be used in your program. If the logical expression is true, the function call has no effect, but if the expression is false, then the program terminates with an error message. Consider

\[
\text{assert ( } x < i \text{ );}
\]

If that code is included in a program, then provided that \(x<i\) there is no effect. If \(x\geq i\) then the program exits with an error message:

\text{assert.cpp:13: int main(): Assertion ‘x < i’ failed.}

Judicious use of assertions can prevent runaway programs from damaging or destroying your valuable data. From the point of view of debugging, though, they are of limited use, because they give no information about the variables involved—only the fact that the assertion failed.

14.7 Testing

"Testing can prove the presence of bugs, never their absence."

Although one cannot prove a program is correct by testing it, testing can increase confidence that a program is indeed working correctly, or it can demonstrate the presence of some logical errors in the program. Tests can be organised in such a way that no knowledge of the internal working of the program is assumed, and have to rely only on the specification of the program to choose suitable sets of test data. This is known as black box testing. If the structure of the program is used to guide the choice of tests, then white box testing is being used.

Consider a program to find the roots of a quadratic equation. A black box test will have to provide data that represents valid quadratic equations that have no real roots, one repeated real root, and two distinct real roots. In addition, the behaviour of the program when presented with data that does
not represent a quadratic equation would have to be tested with non numeric input, and numeric input representing a constant, or a linear equation.

White box testing would generate the same tests, although it might be able to dispense with some of the non numeric input tests if the structure of the program allowed it. However, the construction of the program might well lead to more than three different sets of valid test data representing the three possibilities for roots. Some such programs calculate the smaller root using the larger root and relationship between the roots of a quadratic equation. Such a program would have extra paths to be tested, that could not be deduced form the external specification.

When white box testing, tests can be constructed to exercise the program to its limits. Data can be found that will cause loops not to be executed, to be executed once only, or to be executed many times. If there are internal limits on variable values, then the tests should exceed these limits and check that the program responds sensibly.