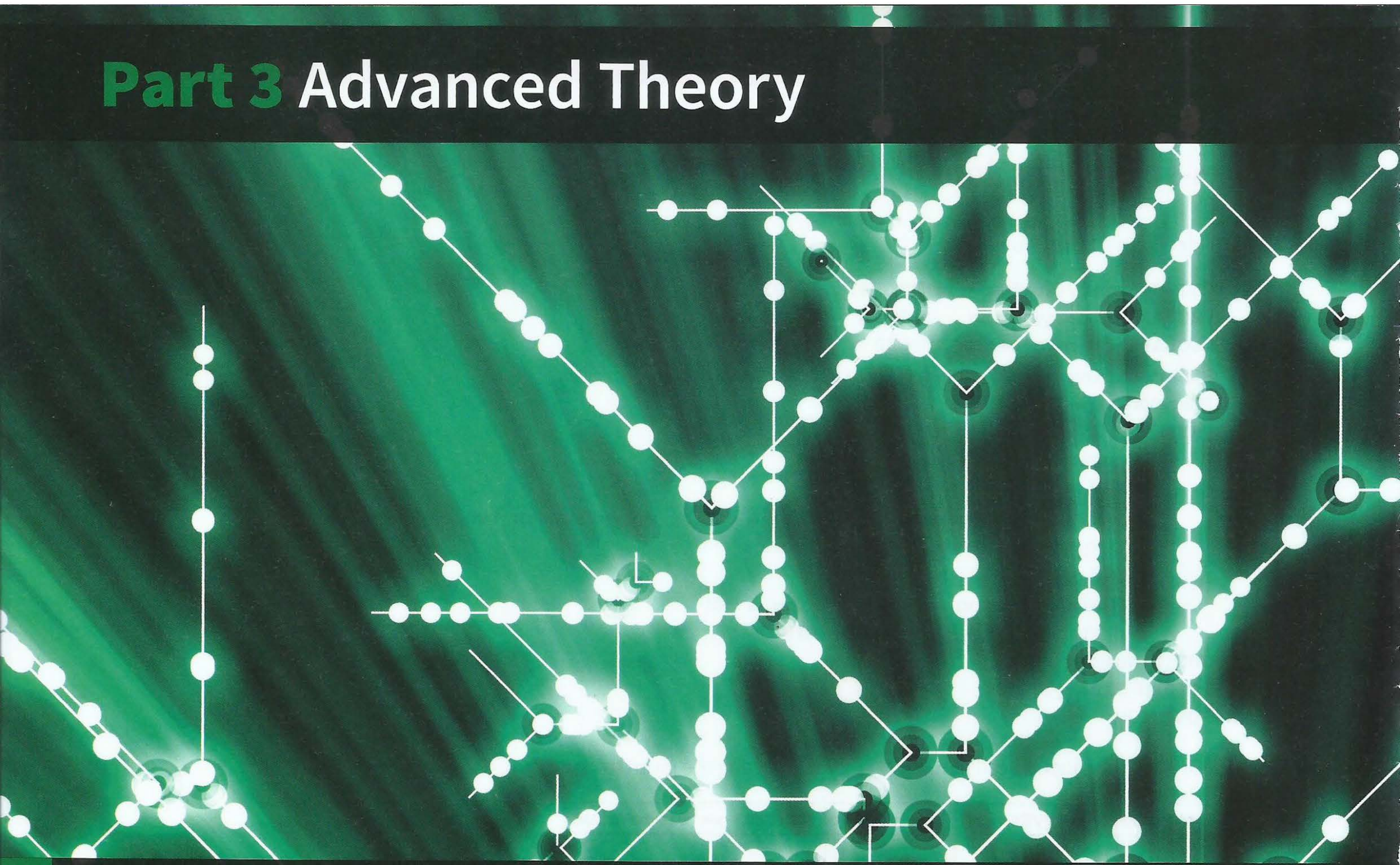
Part 3 Advanced Theory



Chapter 16 Data Representation

Learning objectives

By the end of this chapter you should be able to:

- show understanding of why user-defined types are necessary
- define composite and non-composite types
- show understanding of methods of file organisation and of file access
- select an appropriate method of file organisation and file access for a given problem
- describe the format of binary floating-point real numbers
- normalise floating-point numbers and show understanding of the reasons for normalisation
- show understanding of the effects of changing the allocation of bits to mantissa and exponent in a floating-point representation
- convert binary floating-point real numbers into denary and vice versa
- show understanding of the consequences of a binary representation only being an approximation to the real number it represents and that binary representations can give rise to rounding errors
- show understanding of how underflow and overflow can occur.

16.01 User-defined data types

This chapter must start with a clarification. It is generally accepted that a programmer writes a program which is to be used by a 'user' in the same way that an operating system provides a 'user' interface. However, in the activity of programming the programmer now becomes the 'user' of the programming language. The term 'user-defined data type' applies to this latter type of user.

Non-composite user-defined data types

A non-composite data type has a definition which does not involve a reference to another type. The simple built-in types such as integer or real are obvious examples. When a programmer uses a simple built-in type the only requirement is for an identifier to be named with a defined type. A user-defined type has to be explicitly defined before an identifier can be created. Two examples are discussed here.

Enumerated data type

An **enumerated data type** defines a list of possible values. The following pseudocode shows two examples of type definitions:

```
TYPE
TDirections = (North, East, South, West)
TDays = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)
```

Variables can then be declared and assigned values, for example:

```
DECLARE Direction1 : TDirections
DECLARE StartDay : TDays
Direction1 ← North
StartDay ← Wednesday
```

It is important to note that the values of the enumerated type look like string values but they are not. They must not be enclosed in quote marks.

The values defined in an enumerated data type are ordinal. This means that they have an implied order of values. This makes the second example much more useful because the ordering can be put to many uses in a program. For example, a comparison statement can be used with the values and variables of the enumerated data type:

```
DECLARE Weekend : Boolean
DECLARE Day : TDays
Weekend = TRUE IF Day > Friday
```

KEY TERMS

Enumerated data type: a list of possible data values

Pointer data type

A pointer data type is used to reference a memory location. It may be used to construct dynamically varying data structures.

The pointer definition has to relate to the type of the variable that is being pointed to. The pseudocode for the definition of a pointer is illustrated by:

```
TYPE
TMyPointer = ^<Type name>
```

Declaration of a variable of pointer type does not require the caret symbol (^) to be used:

```
DECLARE MyPointer : TMyPointer
```

A special use of a pointer variable is to access the value stored at the address pointed to. The pointer variable is said to be 'dereferenced':

```
ValuePointedTo ← MyPointer^
```

Composite user-defined data types

A composite user-defined data type has a definition with reference to at least one other type. Three examples are considered here.

Record data type

A **record data type** is the most useful and therefore most widely used. It allows the programmer to collect together values with different data types when these form a coherent whole.

KEY TERMS

Record data type: a data type that contains a fixed number of components, which can be of different types

As an example, a record could be used for a program using employee data. Pseudocode for defining the type could be:

```
TYPE
TEmployeeRecord
    DECLARE EmployeeFirstName : STRING
    DECLARE EmployeeFamilyName : STRING
    DECLARE DateEmployed : DATE
    DECLARE Salary : CURRENCY
ENDTYPE
```

An individual data item can then be accessed using a dot notation:

```
Employee1.DateEmployed ← #16/05/2017#
```

A particular use of a record is for the implementation of a data structure where one or possibly two of the variables defined are pointer variables.

Set data type

A set data type allows a program to create sets and to apply the mathematical operations defined in set theory. The following is a representative list of the operations to be expected:

- union
- difference
- intersection

- include an element in the set
- exclude an element from the set
- check whether an element is in a set.

Objects and classes

In object-oriented programming, a program defines the classes to be used – they are all user-defined data types. Then for each class the objects must be defined. Chapter 27 (Section 27.03) has a full discussion of this subject.

Why are user-defined data types necessary?

When object-oriented programming is not being used a programmer may choose not to use any user-defined data types. However, for any reasonably large program it is likely that their use will make a program more understandable and less error-prone. Once the programmer has decided because of this advantage to use a data type that is not one of the built-in types then user-definition is inevitable. The use of, for instance, an integer variable is the same for any program. However, there cannot be a built-in record type because each different problem will need an individual definition of a record.

16.02 File organisation

In everyday computer usage, a wide variety of file types is encountered. Examples are graphic files, word-processing files, spreadsheet files and so on. Whatever the file type, the content is stored using a defined binary code that allows the file to be used in the way intended.

For the very specific task of storing data to be used by a computer program, there are only two defined file types. A file is either a text file or a **binary file**. A text file, as discussed in Chapter 13 (Section 13.09), contains data stored according to a defined character code as defined in Chapter 1 (Section 1.03). It is possible, by using a text editor, to create a text file. A binary file stores data in its internal representation, for example an integer value might be stored in two bytes in two's complement representation. This type of file will be created using a specific program.

The organisation of a binary file is based on the concept of a **record**. A file contains records and each record contains fields. Each field consists of a value.

KEY TERMS

Binary file: a file designed for storing data to be used by a computer program

Record: a collection of fields containing data values

Discussion Point:

A record is a user-defined data type. It is also a component of a file. Can there be or should there be any relationship between these two concepts?

Serial files

A serial file contains records which have no defined order. A typical use of a serial file would be for a bank to record transactions involving customer accounts. A program would be running. Each time there was a withdrawal or a deposit the program would receive the details as data input and would record these in a transaction file. The records would enter the file in chronological order but otherwise the file would have no ordering of the records.

A text file can be considered to be a type of serial file but it is different because the file has repeating lines which are defined by an end-of-line character or characters. There is no end-of-record character. A record in a serial file must have a defined format to allow data to be input and output correctly.

Sequential files

A sequential file has records that are ordered. It is the type of file suited to long-term storage of data. As such it should be the type of file that is considered as an alternative to a database. The discussion in Chapter 10 (Section 10.01) compared a text file with a database but the arguments for using a database remain the same if a sequential file is used for the comparison. In the banking scenario, a sequential file could be used as a master file for an individual customer account. Periodically, the transaction file would be read and all affected customer account master files would be updated.

In order to allow the sequential file to be ordered there has to be a key field for which the values are unique and sequential but not necessarily consecutive. It is worth emphasising the difference between key fields and primary keys in a database table, where the values are required to be unique but not to be sequential. In a sequential file, a particular record is found by sequentially reading the value of the key field until the required value is found.

Direct-access files

Direct-access files are sometimes referred to as 'random-access' files but, as with random-access memory, the randomness is only that the access is not defined by a sequential reading of the file. For large files, direct access is attractive because of the time that would be taken to search through a sequential file. In an ideal scenario, data in a direct-access file would be stored in an identifiable record which could be located immediately when required. Unfortunately, this is not possible. Instead, data is stored in an identifiable record but finding it may involve an initial direct access to a nearby record followed by a limited serial search.

The choice of the position chosen for a record must be calculated using data in the record so that the same calculation can be carried out when subsequently there is a search for the data. The normal method is to use a hashing algorithm. This takes as input the value for the key field and outputs a value for the position of the record relative to the start of the file. The hashing algorithm must take into account the potential maximum length of the file, that is, the number of records the file will store. A simple example of a hashing algorithm, if the key field has a numeric value, is to divide the value by a suitably large number and use the remainder from the division to define the position. This method will not create unique positions. If a hash position is calculated that duplicates one already calculated by a different key, the next position in the file is used. This is why a search will involve a direct access possibly followed by a limited serial search.

File access

Once a file organisation has been chosen and the data has been entered into a file, the question now to be considered is how this data is to be used. If an individual data item is to be read then the access method for a serial file is to successively read record by record until the required data is found. If the data is stored in a sequential file the process is similar but only the value in the key field has to be read. For a direct-access file, the value in the key field is submitted to the hashing algorithm which then provides the same value for the position in the file that was provided when the algorithm was used at the time of data input.

File access might also be needed to delete or edit data. The normal approach with a sequential file is to create a new version of the file. Data is copied from the old file to the new file until the record is reached which needs deleting or editing. If deletion is needed, reading and copying of the old file continues from the next record. If a record has changed, an edited version of the record is written to the new file and then the remaining records are copied to the new file. For a direct-access file there is no need to create a new file (unless the file has become full). A deleted record can have a flag set so that in a subsequent reading process the record is skipped over.

Serial file organisation is well suited to batch processing or for backing up data on magnetic tape. However, if a program needs a file in which individual data items might be read, updated or deleted then direct-access file organisation is the most suitable and serial file organisation the least suitable.

16.03 Real numbers

A real number is one with a fractional part. When we write down a value for a real number in the denary system we have a choice. We can use a simple representation or we can use an exponential notation (sometimes referred to as scientific notation). In this latter case we have options. For example, the number 25.3 might alternatively be written as:

$$.253 \times 10^{2} \quad \text{or} \quad 2.53 \times 10^{1} \quad \text{or} \quad 25.3 \times 10^{0} \quad \text{or} \quad 253 \times 10^{-1}$$

For this number, the simple expression is best but if a number is very large or very small the exponential notation is the only sensible choice.

Floating-point and fixed-point representations

A binary code must be used for storing a real number in a computer system. One possibility is to use a fixed-point representation. In this option, an overall number of bits is chosen with a defined number of bits for the whole number part and the remainder for the fractional part. The alternative is a **floating-point representation**. The format for a floating-point number can be generalised as:

$$\pm M \times R^{E}$$

In this option a defined number of bits are used for what is called the significand or mantissa, ±M. The remaining bits are used for the exponent or exrad, E. The radix, R is not stored in the representation; it has an implied value of 2.

KEY TERMS

Floating-point representation: a representation of real numbers that stores a value for the mantissa and a value for the exponent

To illustrate the differences between the two representations a very simple example can be used. Let's consider that a real number is to be stored in eight bits.

For the fixed-point option, a possible choice would be to use the most significant bit as a sign bit and the next five bits for the whole number part leaving two bits for the fractional part.

Some important non-zero values in this representation are shown in Table 16.01. (The bits are shown with a gap to indicate the implied position of the binary point.)

Description	Binary code	Denary equivalent
Largest positive value	011111 11	31.75
Smallest positive value	000000 01	0.25
Smallest magnitude negative value	100000 01	–0.25
Largest magnitude negative value	111111 11	–31.75

Table 16.01 Example fixed-point representations (using sign and magnitude)

For a floating-point representation, a possible choice would be four bits for the mantissa and four bits for the exponent with each using two's complement representation. The exponent is stored as a signed integer. The mantissa has to be stored as a fixed-point real value. The question now is where the binary point should be.

Two of the options for the mantissa being expressed in four bits are shown in Table 16.02(a) and Table 16.02(b). In each case, the denary equivalent is shown and the position of the implied binary point is shown by a gap. Table 16.02(c) shows the three largest magnitude positive and negative values for integer coding that will be used for the exponent.

a)

First bit pattern for a real value	Real value in denary
011 1	3.5
011 0	3.0
010 1	2.5
101 0	–3.0
100 1	–3.5
100 0	–4.0

b)

Second bit pattern for a real value	Real value in denary
0 111	.875
0 110	.75
0 101	.625
1 010	–.75
1 001	–.875
1 000	–1.0

c)

Integer bit pattern	Integer value in denary
0111	7
0110	6
0101	5
1010	–6
1001	–7
1000	–8

Table 16.02 Coding a fixed-point real value in eight bits (four for the mantissa and four for the exponent)

It can be seen that having the mantissa with the implied binary point immediately following the sign bit produces smaller spacing between the values that can be represented. This is the preferred option for a floating-point representation. Using this option, the most important non-zero values for the floating-point representation are shown in Table 16.03. (The implied binary point and the mantissa exponent separation are shown by a gap.)

Description	Binary code	Denary equivalent
Largest positive value	0 111 0111	$.875 \times 2^{7} = 112$
Smallest positive value	0 001 1000	$.125 \times 2^{-8} = 1/2048$
Smallest magnitude negative value	1 111 1000	$-.125 \times 2^{-8} = -1/2048$
Largest magnitude negative value	1 000 0111	$-1 \times 2^{7} = -128$

Table 16.03 Example floating-point representations

The comparison between the values in Tables 16.01 and 16.03 illustrate the greater range of positive and negative values available if floating-point representation is used.

Extension question 16.01

1 Using the methods suggested in Chapter 1 (Section 1.01) can you confirm for yourself that the denary equivalents of the binary codes shown in Tables 16.02 and Table 16.03 are as indicated?

2 Can you also confirm that conversion from positive to negative or vice versa for a fixed-format real value still follows the rules defined in Chapter 1 (Section 1.02) for two's complement representation.

Precision and normalisation

In principle a decision has to be made about the format of a floating-point representation both with regard to the total number of bits to be used and the split between those representing the mantissa and those representing the exponent. In practice, a choice for the total number of bits to be used will be available as an option when the program is written. However, the split between the two parts of the representation will have been determined by the floating-point processor. If you did have a choice you would base a decision on the fact that increasing the number of bits for the mantissa would give better precision for a value stored but would leave fewer bits for the exponent so reducing the range of possible values.

In order to achieve maximum precision, it is necessary to normalise a floating-point number. (This normalisation is totally unrelated to the process associated with designing a database.) Since precision increases with an increasing number of bits for the mantissa it follows that optimum precision will only be achieved if full use is made of these bits. In practice, that means using the largest possible magnitude for the value represented by the mantissa.

To illustrate this we can consider the eight-bit representation used in Table 16.03. Table 16.04 shows possible representations for denary 2 using this representation.

Denary representation	Floating-point binary representation
0.125×2^4	0 001 0100
0.25×2^3	0 010 0011
0.5×2^2	0 100 0010

Table 16.04 Alternative representations of denary 2 using four bits each for mantissa and exponent.

For a negative number we can consider representations for −4 as shown in Table 16.05.

Denary representation	Floating-point binary representation
-0.25×2^4	1 110 0100
-0.5×2^3	1 100 0011
-1.0×2^2	1 000 0010

Table 16.05 Alternative representations of denary −4 using four bits each for mantissa and exponent.

It can be seen that when the number is represented with the highest magnitude for the mantissa, the two most significant bits are different. This fact can be used to recognise that a number is in a normalised representation. The values in these tables also show how a number could be normalised. For a positive number, the bits in the mantissa are shifted left until the most significant bits are 0 followed by 1. For each shift left the value of the exponent is reduced by 1.

The same process of shifting is used for a negative number until the most significant bits are 1 followed by 0. In this case, no attention is paid to the fact that bits are falling off the most significant end of the mantissa.

Conversion of representations

In Chapter 1 (Section 1.01), a number of methods for converting numbers into different representations were discussed. The ideas presented there now need a little expansion.

Let's start by considering the conversion of a simple real number, such as 4.75, into a simple fixed-point binary representation. This looks easy because 4 converts to 100 in binary and .75 converts to .11 in binary so the binary version of 4.75 should be:

100.11

However, we now remember that a positive number should start with 0. Can we just add a sign bit? For a positive number we can. Denary 4.75 can be represented as 0100.11 in binary.

For negative numbers we still want to use two's complement form. So, to find the representation of −4.75 we can start with the representation for 4.75 then convert it to two's complement as follows:

0100.11 converts to 1011.00 in one's complement

then to 1011.01 in two's complement

To check the result, we can apply Method 2 from Worked Example 1.01 in Chapter 1. 1011 is the code for −8 + 3 and .01 is the code for .25; −8 + 3 + .25 = −4.75.

We can now consider the conversion of a denary value expressed as a real number into a floating-point binary representation. The first thing to realise is that most fractional parts do not convert to a precise representation. This is because the binary fractional parts represent a half, a quarter, an eighth, a sixteenth and so on. Unless a denary fraction is a sum of a collection of these values, there cannot be an accurate conversion. In particular, of the values from .1 through to .9 only .5 converts accurately. This was mentioned in Chapter 1 (Section 1.02) in the discussion about storing currency values.

The method for conversion of a positive value is as follows:

1 Convert the whole-number part using the method described in Chapter 1 (Section 1.01).
2 Add the 0 sign bit.
3 Convert the fractional part using the method described in Worked Example 16.01.
4 Combine the two, with the exponent expressed as zero.
5 Adjust the position of the binary point and change the exponent accordingly to achieve a normalised form.

WORKED EXAMPLE 16.01

Converting a denary value to a floating-point representation

Example 1

Let's consider the conversion of 8.75:

1 The 8 converts to 1000, adding the sign bit gives 01000.
2 The .75 can be recognised as being .11 in binary.
3 The combination gives 01000.11 which has exponent value zero.

4 Shifting the binary point gives 0.100011 which has exponent value denary 4.

5 The next stage depends on the number of bits defined for the mantissa and the exponent; if ten bits are allocated for the mantissa and four bits are allocated for the exponent the final representation becomes 0100011000 for the mantissa and 0100 for the exponent.

Example 2

Let's consider the conversion of 8.63. The first step is the same but now the .63 has to be converted by the 'multiply by two and record whole number parts' method. This works as follows:

.63 x 2 = 1.26 so 1 is stored to give the fraction .1

.26 x 2 = .52 so 0 is stored to give the fraction .10

.52 x 2 = 1.04 so 1 is stored to give the fraction .101

.04 x 2 = .08 so 0 is stored to give the fraction .1010

At this stage it can be seen that multiplying .08 by 2 successively is going to give a lot of zeros in the binary fraction before another 1 is added so the process can be stopped. What has happened is that .63 has been approximated as .625. So, following Steps 3–5 in Example 1, the final representation becomes 0100010100 for the mantissa and 0100 for the exponent.

TASK 16.01

Convert the denary value −7.75 to a floating-point binary representation with ten bits for the mantissa and four bits for the exponent. Start by converting 7.75 to binary (make sure you add the sign bit!). Then convert to two's complement form. Finally, choose the correct value for the exponent to leave the implied position of the binary point after the sign bit. Convert back to denary to check the result.

Problems with using floating-point numbers

As illustrated above, the conversion of a real value in denary to a binary representation almost guarantees a degree of approximation. This is then added to by the restriction of the number of bits used to store the mantissa.

Many uses of floating-point numbers are in extended mathematical procedures involving repeated calculations. Examples of such use would be in weather forecasting using a mathematical model of the atmosphere or in economic forecasting. In such programming there is a slight approximation in recording the result of each calculation. These so-called rounding errors can become significant if calculations are repeated enough times. The only way of preventing this becoming a serious problem is to increase the precision of the floating-point representation by using more bits for the mantissa. Programming languages therefore offer options to work in 'double precision' or 'quadruple precision'.

The other potential problem relates to the range of numbers that can be stored. Referring back to the simple eight-bit representation illustrated in Table 16.03, the highest value represented is denary 112. A calculation can easily produce a value higher than this. As Chapter 5 (Section 5.02) illustrated, this produces an overflow error condition. However, for floating-point values there is also a possibility that if a very small number is divided by a number greater than 1 the result is a value smaller than the smallest that can be stored. This is an underflow error condition. Depending on the circumstances, it may be possible for a program to continue running by converting this very small number to zero but clearly this must involve risk.

Summary

- Examples of non-composite user-defined data types include enumerated and pointer data types.
- Record, set and class are examples of composite user-defined data types.
- File organisation allows for serial, sequential or direct access.
- Floating-point representation for a real number allows a wider range of values to be represented.
- A normalised floating-point representation achieves optimum precision for the value stored.
- Stored floating-point values rarely give an accurate representation of the denary equivalent.

Exam-style Questions

1 A programmer may choose to use a user-defined data type when writing a program.

a Give an example of a non-composite user-defined data type and explain why its use by a programmer is different to the use of an in-built data type. [3]

b A program is to be written to handle data relating to the animals kept in a zoo. The programmer chooses to use a record user-defined data type.

i Explain what a record user-defined data type is. [2]

ii Explain the advantage of using a record user-defined data type. [2]

iii Write pseudocode for the definition of a record type which is to be used to store: animal name, animal age, number in zoo and location in the zoo. [5]

2 a A binary file is to be used to store data for a program.

i What are the terms used to describe the components of such a file. [2]

ii Explain the difference between a binary file and a text file. [3]

b A binary file might be organised for serial, sequential or direct access.

i Explain the difference between the three types of file organisation. [4]

ii Give an example of file use for which a serial file organisation would be suitable. Justify your choice. [3]

iii Give an example of file use when direct access would be advantageous. Justify your choice. [3]

3 A file contains binary coding. The following are four successive bytes in the file:

10010101	00110011	11001000	00010001

a The four bytes represent two numbers in floating-point representation. The first byte in each case represents the mantissa. Each byte is stored in two's complement representation.

i Give the name for what the second byte represents in each case. [1]

ii State whether the representations are for two positive numbers or two negative numbers and explain why. [2]

iii One of the numbers is in a normalised representation. State which one it is and give the reason why. [2]

iv State where the implied binary point is in a normalised representation and explain why a normalised representation gives better precision for the value represented. [3]

v If two bytes were still to be used but the number of bits for each component was going to be changed by allocating more to the mantissa, what effect would this have on the numbers that could be represented? Explain your answer. [2]

b Using the representation described in part (a), Show the representation of denary 12.43 as a floating-point binary number. [5]