

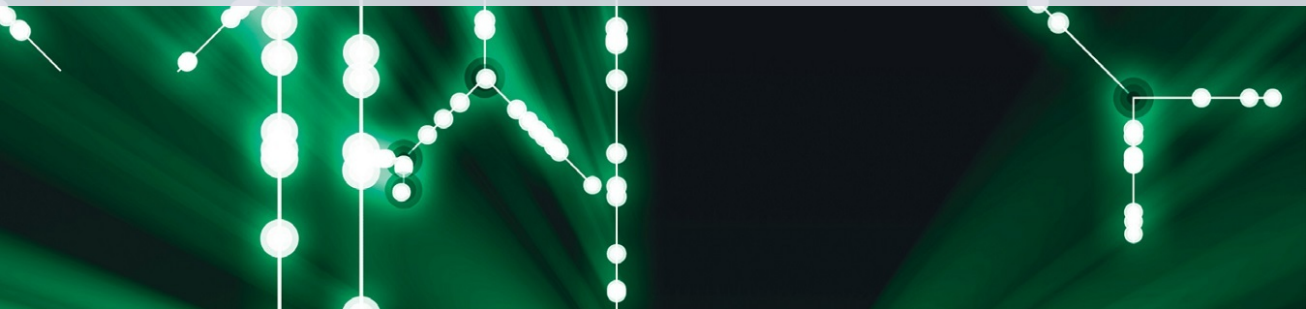


Chapter 16:

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 and use non-composite data types
 - define and use composite data types
 - choose and design an appropriate user-defined data type for a given problem
 - show understanding of the methods of file organisation and select an appropriate method of file organisation and file access for a given problem
 - show understanding of methods of file access
 - show understanding of hashing algorithms
 - describe the format of binary floating-point real numbers
 - 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
 - normalise floating-point numbers
 - show understanding of the consequences of a binary representation only being an approximation to the real number it represents (in certain cases)
 - show understanding that binary representations can give rise to rounding errors.
- 

16.01 Data types

Sections 13.01 to 13.05 of Chapter 13 introduced the concept of a variable being associated with a data type. Before a variable can be used in a program, the variable's data type has to be identified. Chapter 13 introduced the most frequently used data types that are available for association with a variable in a program. This chapter introduces some additional data types that might be used.

Built-in data types

Remember, for each built-in data type:

- the programming language defines the range of possible values that can be assigned to a variable when its type has been chosen.
- the programming language defines the operations that are available for manipulating values assigned to the variable.

User-defined data types

The term 'user' is regularly applied to someone who is provided with a 'user interface' by an operating system – the 'user' is the person supplying input to a running program and receiving output from it.

However, when writing a program, a programmer becomes a 'user' of a programming language. The term **user-defined data type** applies to this latter type of user.

A user-defined data type is a data type for which the programmer has included the definition in the program. Once the data type has been defined, variables can be created and associated with the user-defined data type. Note that, although the user-defined data type is not a built-in data type, using the user-defined data type is only possible if a programming language offers support for the construct.



TIP

Make sure that you do not confuse user-defined data types and abstract data types (defined in Section 13.07 of Chapter 13).

Non-composite data types

A **non-composite data type** is one which has a definition which does not involve a reference to another data type. The simple built-in data types, such as integer or real, are examples of built-in non-composite data types. It is also possible for a user-defined data type to be non-composite.

Enumerated data type

An enumerated data type is an example of a user-defined non-composite data type. When a specific **enumerated data type** is defined, every single possible value for it is identified. The following pseudocode shows two examples of enumerated data type definitions:

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

Following these definitions, variables can be declared and assigned values, for example:

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

It is important to note the following points.

- The values of the enumerated type look like string values but they are not. The values must not be enclosed in quotes.
- The values defined in an enumerated data type are ordinal. This means that enumerated data types

have an implied order of values.

The ordering can be put to many uses in a program. For example, a comparison statement can be used with the values of the variables of an enumerated data type:

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

The enumerated data type is one reason why user-defined data types are sometimes needed. There could not be a built-in generic definition of an enumerated data type because the possible values would not be known. The values can only be known when the programmer has identified them in the type definition.

Composite user-defined data types

A composite user-defined data type has a definition with reference to at least one other type. There are two very important examples of composite user-defined data type.

- 1 The **record data type** (introduced in [Chapter 13](#)). Although there could be built-in record data types the expectation is for a record data type to be user-defined. This allows the programmer to create record data types with components that precisely match the data requirements of the particular program. Note that Python is a language that does not support the use of a record data type.
- 2 The **class**. A class is a data type which is used for an object in object-oriented programming. For a given object-oriented programming language there are likely to be a number of built-in classes. However, if a programmer intends to utilise the benefits of the object-oriented approach, then the programmer will have to create a number of user-defined classes.

Pointer data type

A **pointer variable** is one for which the value is a reference to a memory location. The following is a commentary on some examples of pseudocode involving the use of the pointer data type.

- 1 An example of the definition of a pointer type which requires only the identification of a data type for which the pointer is to be used.

```
TYPE
TIntegerPointer ← ^Integer
```

- 2 An example of the declaration of a variable of the pointer data type which does not require the use of the caret (^) symbol.

```
DECLARE MyIntegerPointer : TIntegerPointer
```

- 3 An example of the declaration of two ordinary variables of type integer and the assignment of a value for one of them.

```
DECLARE Number1, Number2 : INTEGER
Number1 ← 100
```

- 4 An example of an assignment to a pointer variable of a value which is the address of a different variable.

```
MyIntegerPointer ← @Number1
```

- 5 An example of an assignment which uses the 'dereferenced' value which has been stored at the address defined by the pointer variable. This assigns the value 200 to Number2.

```
Number2 ← MyIntegerPointer^ * 2
```

Not all programming languages offer support for the use of a pointer data type. Those languages that do so will have their own version of the symbolism illustrated above with ^ and @.

Because arithmetic can be performed on pointer variables, it is possible to use pointer variables to construct dynamically varying data structures. For some programming languages it is necessary to declare an array with a large upper bound to ensure that the array is unlikely to be fully populated with values. If the language supports the use of a pointer variable, the size of an array can expand while a

program is running. The details of how this can be done are beyond the scope of this discussion.

Set data type

A **set** data type allows a program to create sets and to apply the mathematical operations defined in set theory. A set is a mathematical concept with important properties.

- It contains a collection of data values.
- There is no organisation of the data values within the set.
- Duplicate values are not allowed.
- Operations that can be performed on a set include:
 - checking if a value exists in a set
 - adding a new data value
 - removing an existing data value
 - adding one set to another set.

A set variable can be created if a programming language supports the set data type. It is difficult to classify the set data type. Because the set contains multiple components it is tempting to say that the set is a structured data type. However, this contradicts the fact that the set has no structure and therefore no indexing can be associated with the members of the set.

The most useful property of a set is the fact that duplicate values are not allowed. A list or a one-dimensional array might be created but has to be checked to remove duplicate values. A simple way of removing duplicate values would be to convert the structure to a set and then convert the set back to the original structure.

A slightly different example would be if students were allocated to groups for studying a particular subject. For each subject, the students' names would be entered into a data structure defined for that subject. Set data types could then, for example, find out which students were studying both computer science and physics. The students studying both subjects would be found by applying the 'intersection' operation on the two individual sets.

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 specific binary code that allows the file to be used as intended.

For the specific task of storing data to be used for input to a computer program or for output from 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 \(Section 13.06\)](#) contains data stored according to a character code of the type described in [Chapter 1 \(Section 1.04\)](#). It is possible, by using a text editor, to create a text file to be used as input to a program.

A program may create a binary file as output with the intention of subsequently using it for input. 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.

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. For a text file the number of data items per line must be known and the number of characters per item must be known. If these are not known then item separator characters must be used. The file has repeating lines which are defined by an end-of-line character or characters.

For a binary file the number of fields per record must be known. If any of the fields represent a string, the length of the string must be known. For any other field the internal representation will define the number of bytes required to store the field value. There is no need for field separator characters or for an end-of-record character.

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 that have not been organised in any 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 the data in a transaction file. In a serial file each new record is simply appended to the file so that the only ordering in the file is the time order of data entry.

Sequential files

A sequential file has records that are ordered. In the bank example, a sequential file could be used as a master file for an individual customer account. At regular periods of time, the transaction file would be read, and all affected customer account master files would be updated. In order to allow a sequential file to be ordered, there has to be a key field for which the values are unique and sequential but not necessarily consecutive. When a new record is to be added to a sequential file it would be possible to simply append the record, with the intention of sorting the file later. A more likely approach is for the file to be read sequentially and each record written to a new file. This is continued until the appropriate position for the new record is reached. The new record is then written to the new file before the remaining records in the old file are copied in.

Extension Question 16.01

Can you think of reasons why you might want to use binary files with variable-length records? How would you make sure a binary file with variable-length records would be read correctly?

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 can be to any record in the file without sequential reading of the file. Direct access can be achieved with a sequential file. A separate index file is created

which has two fields per record. The first field has the key field value and the second field has a value for the position of this key field value in the main file.

The alternative is to use a hashing algorithm when a record is entered into the direct-access file.

One simple hashing algorithm is applicable if there is a numeric key field in each record. The algorithm chooses a suitable number and divides this number by the value in the key field. The remainder from this division then identifies the address in the file for storage of that record. The suitable number works best if it is a prime number of a similar size to the expected size of the file.

For simplicity this can be illustrated for 4-digit values in the key field where 1000 is used for the dividing number. The following represent three calculations:

0045/1000 gives remainder 45 for the address in the file

2005/1000 gives remainder 5 for the address in the file

3005/1000 gives remainder 5 for the address in the file

There are two facts apparent from these calculations. The first fact is that the addresses calculated do not have any order depending on the value in the key field. The second fact is that different key field values can produce the same remainder and therefore the same address in the file.

If the records do not have a suitable field with numeric digits, an alternative is to choose a field with some alphabetic characters. The ASCII code for each character can be looked up and the values then added. The sum is then used in the same way as described above, to calculate an address as the remainder from an integer division.

When the same address is calculated for different field values, it is usually referred to as a collision (the addresses are sometimes called synonyms). The best choice for a hashing algorithm is one that spreads the addresses most evenly and minimises the number of collisions. However, collisions cannot be avoided altogether so there has to be a defined method for dealing with collisions. There are a number of options, including the following:

- use a sequential search to look for a vacant address following the calculated one
- keep a number of overflow addresses at the end of the file
- have a linked list accessible from each address.

Question 16.01

Imagine the possible numeric values for a key field in a direct-access file are in the range of 1 to 30 but you want the file to have fewer than 30 file addresses.

You decide to test two examples of a modular division hashing algorithm. The first test uses 10 as the number for division, the second test uses 11.

- a** What are the two sets of addresses generated as remainders from the division for the key values 0 to 39 using 10 and 11?
- b** State one difference between the two sets of addresses.
- c** Is there any significant difference between the two sets of addresses?
- d** 11 is a prime number. Prime numbers are stated to give a better spread of use of the addresses in a file. Do you know when this is more likely to be true?

File access

Once a file organisation has been chosen and the data has been entered into a file, you need to consider how this data is to be accessed. For a serial file, the normal usage is to read the whole file record by record. If there was a need to search for a particular value in one of the fields, the only option would be to read the records from the beginning until the target record was found. If the data is stored in a sequential file and a particular value is needed, searching may have to be done in the same way. However, if the key field value is known for the record containing the wanted data, the process is faster because only key field values need to be read. For a direct-access file, the value in the key field is

submitted to the hashing algorithm. The value is the same value that was used when entering the data originally and will provide the same value for the position in the file that was provided when the algorithm was used at the time of data input. This eliminates the need to read records from the beginning of the file. However, because of the collision problem some serial searching might be needed after the initial jump to the hashed position.

File access might also be needed to delete or edit data. For a sequential file the same method is used as when a new record was added. Records are copied from the old file to a new file until the record that needs to be deleted or edited is reached. Following deletion or editing all remaining records are copied to the new file.

For a direct-access file there is no need to create a new file. If a record needs editing it can be accessed directly and edited without disturbing any other content. However, if a record is to be deleted it is necessary to have a flag set in the record. Then, in a subsequent reading process, that record is skipped over.

Choice of file organisation

Serial file organisation is well suited to batch processing or for backing up data on magnetic tape. A direct access file is used if rapid access to an individual record in a large file is required. An example would be on a system with many users. In this case, the file that is used to check passwords when users log in should be direct-access. A sequential file is suitable for applications when multiple records are required from one search of the file. An example could be a family history file where a search could be used for all records with a particular family name.

At this point it is worth mentioning the difference between key fields in a file and primary keys in a database table. In the database table the primary key values must all be unique. This is not a requirement for key fields in any type of file. It may be sensible in certain applications to ensure key fields have unique values, but it is not mandatory.

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). For example, the number 25.3 might be written as:

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

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 fixed-point representation, 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 floating-point representation a defined number of bits are used for what is called the significand or mantissa, $\pm M$. The remaining bits are used for the exponent or exrad, E . The radix, R is not stored in the representation; R has an implied value of 2.

A simple example can be used to illustrate the differences between the two representations. 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. This would leave 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)

A possible choice for a floating-point representation 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

b

Second bit pattern for a real value	Real value in denary
0 111	0.875

c

Integer bit pattern	Integer value in denary
0111	7
0110	6

011 0	3.0	0 110	0.75	0101	5
010 1	2.5	0 101	0.625	1010	-6
101 0	-3.0	1 010	-0.75	1001	-7
100 1	-3.5	1 001	-0.875	1000	-8
100 0	-4.0	1 000	-1.0		

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

When the mantissa has the implied binary point immediately following the sign bit, a smaller spacing is produced 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	$0.875 \times 2^7 = 112$
Smallest positive value	0 001 1000	$0.125 \times 2^{-8} = 1/2048$
Smallest magnitude negative value	1 111 1000	$-0.125 \times 2^{-8} = -1/2048$
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 16.02

- 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?
- Can you also confirm that conversion from positive to negative (or the conversion from negative to positive) 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

You have to decide about the format of a floating-point representation in two respects. You have to decide the total number of bits to be used and decide on 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 your 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, which reduces the range of possible values.

To achieve maximum precision you have to normalise a floating-point number. (This normalisation is unrelated to the process associated with designing a database.) Precision increases with an increasing number of bits for the mantissa, so 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

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 Tables 16.03 and 16.04 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.

Extension Question 16.03

Look at the data in Table 16.03. Do you see any conflict with the above discussion? What is likely to be the approach in a typical floating-point system?

Conversion of representations

In Chapter 1 (Section 1.01), a number of methods for converting numbers into different representations were discussed. These only considered integer values. We now need to consider the conversion of real numbers.

We can 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, 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. Before considering the conversion method it should be remembered 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 1.03) 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 choosing a method from one of the examples in Worked Example 16.01.
- 4 Combine the whole number and fractional parts and enter these into the most significant of the bits allocated for the representation of the mantissa.
- 5 Fill the remaining bits for the mantissa and the bits for the exponent with zeros.
- 6 Adjust the position of the binary point by changing the exponent value to achieve a normalised representation.

To convert a negative value the number is treated initially as positive and the same first five steps are followed. At this stage a two's complement conversion of the mantissa code is used to convert this to a negative value before step 6 is carried out.

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:

$.63 \times 2 = 1.26$ so 1 is stored to give the fraction .1

$.26 \times 2 = .52$ so 0 is stored to give the fraction .10

$.52 \times 2 = 1.04$ so 1 is stored to give the fraction .101

$.04 \times 2 = .08$ 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. .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. There is also a restriction of the number of bits used to store the mantissa.

Floating-point numbers are used in extended mathematical procedures involving repeated calculations. For example, 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 the errors 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 1 \(Section 1.02\)](#) 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 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 the best precision for the value stored.
- Stored floating-point values rarely give an accurate representation of the denary equivalent.

Reflection Point:

Whenever you are asked to create a binary representation from a denary value or vice-versa are you always checking your answer by converting it back to the original value?

- 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** State 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

- [illegible]

```

DECLARE ThisMonth      :   (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
                             Sep, Oct, Nov, Dec)
DECLARE ThisYear       :   INTEGER
ENDTYPE

```

A variable of this new type is declared as follows:

```

DECLARE DateOfBirth   :   ThisDate

```

- i** Name the non-composite data type used in the `ThisDay` and `ThisMonth` declarations. [1]
- ii** Name the data type of `ThisDate`. [1]
- iii** The month value of `DateOfBirth` needs to be assigned to the variable `MyMonthOfBirth`.
Write the required statement. [1]

b Annual rainfall data from a number of locations are to be processed in a program.

The following data are to be stored:

- location name
- height above sea level (to the nearest metre)
- total rainfall for each month of the year (centimetres to 1 decimal place).

A user-defined, composite data type is needed. The programmer chooses `LocationRainfall` as the name of this data type.

A variable of this type can be used to store all the data for one particular location.

- i** Write the definition for the data type `LocationRainfall`. [5]
- ii** The programmer decides to store all the data in a file. Initially, data from 27 locations will be stored. More rainfall locations will be added over time and will never exceed 100.

The programmer has to choose between two types of file organisation. The two types are serial and sequential.

Give **two** reasons for choosing serial file organisation. [2]

Cambridge International AS & A level Computer Science 9608 paper 31 Q3 June 2015

5 In a particular computer system, real numbers are stored using floating-point representation with:

- 8 bits for the mantissa
- 8 bits for the exponent
- two's complement form for both mantissa and exponent.

a Calculate the floating point representation of +3.5 in this system. Show your working.

Mantissa	Exponent
<div style="display: flex; align-items: center; justify-content: space-between;"> <div style="width: 10px; height: 10px; background-color: black; border-radius: 50%;"></div> <div style="flex-grow: 1; border-bottom: 1px solid black; position: relative;"> <div style="position: absolute; left: 0; top: -1px; width: 100%; height: 1px; background-color: black;"></div> </div> </div>	<div style="display: flex; justify-content: space-around; height: 20px;"> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> </div>

[3]

b Calculate the floating point representation of -3.5 in this system. Show your working.

Mantissa	Exponent
<div style="display: flex; align-items: center; justify-content: space-between;"> <div style="width: 10px; height: 10px; background-color: black; border-radius: 50%;"></div> <div style="flex-grow: 1; border-bottom: 1px solid black; position: relative;"> <div style="position: absolute; left: 0; top: -1px; width: 100%; height: 1px; background-color: black;"></div> </div> </div>	<div style="display: flex; justify-content: space-around; height: 20px;"> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> </div>

c Find the denary value for the following binary floating-point number. Show your working.

Mantissa	Exponent
<div style="display: flex; align-items: center; justify-content: space-between;"> <div style="width: 10px; height: 10px; background-color: black; border-radius: 50%;"></div> <div style="flex-grow: 1; border-bottom: 1px solid black; position: relative;"> <div style="position: absolute; left: 0; top: -1px; width: 100%; height: 1px; background-color: black;"></div> </div> </div>	<div style="display: flex; justify-content: space-around; height: 20px;"> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> <div style="border: 1px solid black; width: 20px; height: 20px;"></div> </div>

[3]

- d i** State whether the floating-point number given in **part (c)** is normalised or not normalised. [1]
- ii** Justify your answer given in **part (d)(i)**. [1]
- e** Give the binary two's complement pattern for the negative number with the largest magnitude.

Mantissa							Exponent						
●													

[2]

Cambridge international AS & A Level Computer Science 9608 paper 32 Q1 November 2016

SYALLLBUS REQUIREMENTS:

13 Data Representation

13.1 User-defined data types

Candidates should be able to:

Show understanding of why user-defined types are necessary

Define and use non-composite types

Define and use composite data types

Choose and design an appropriate user-defined data type for a given problem

Notes and guidance

Including enumerated, pointer

Including set, record and class/object