CHAPTER 1: DATA REPRESENTATION

1.1 USER-DEFINED DATA TYPES

1.1.1 Introduction to User-Defined Data Types

Definition: User-defined data types are data types designed by the programmer. When object-oriented programming is not being used, a programmer may choose to utilize user-defined data types for a large program as their use can reduce errors and make the program more understandable.

Why Use User-Defined Data Types:

- Reduce errors in the program
- Make code more understandable
- Less restriction than built-in types
- Allows for inevitable user definition

1.1.2 Non-Composite User-Defined Data Types

Enumerated Data Type: A non-composite user-defined data type that is a list of possible data values. The values defined have an implied order of values to allow comparisons.

<PSEUDOCODE>

```
TYPE Season = (Summer, Winter, Autumn, Spring)

DECLARE ThisSeason : Season
DECLARE NextSeason : Season
ThisSeason ← Autumn
NextSeason ← ThisSeason + 1 // NextSeason is set to Spring
```

Characteristics:

- Values are countable and finite

- Allow comparisons (value2 > value1)
- NOT string values (don't use quotes)

Pointer Data Type: 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 being pointed to.

<PSEUDOCODE>

```
TYPE IntegerPointer = ^INTEGER

DECLARE IPointer : IntegerPointer
DECLARE MyInt1 : INTEGER
DECLARE MyInt2 : INTEGER

// Store address of MyInt2 in IPointer
IPointer ← @MyInt2
```

Pointer Notation:

- `@identifier` - returns the address of identifier
- `Pointer^` - accesses the data stored at the address (dereferencing)

```
// Access data at address pointed to by IPointer
IPointer^ ← 33
// Store address of MyInt1 in SecondPointer
```

1.1.3 Composite User-Defined Data Types

Record Data Type: A data type that contains a fixed number of components that can be of different types. Allows the programmer to collect values with other data types together when these form a coherent whole.

<PSEUDOCODE>

```
TYPE TEmployeeRecord
 DECLARE FirstName : STRING
 DECLARE LastName : STRING
 DECLARE Salary : REAL
 DECLARE Position : STRING
ENDTYPE

DECLARE Employee1 : TEmployeeRecord
```

Set Data Type: Allows a program to create sets and to apply mathematical operations defined in set theory. All elements in the set should be unique.

<PSEUDOCODE>

```
TYPE Days = SET OF STRING
DEFINE Today(Monday, Tuesday, Wednesday, Thursday, Friday) : Days
```

Operations on Sets:

- Union
- Difference
- Intersection
- Include an element
- Exclude an element
- Check whether an element is in a set

Class (in OOP): In object-oriented programming, a program defines the classes to be used. Then, for each class, the objects must be defined. A Class includes variables and methods (functions or procedures that an object can run).

1.2 FILE ORGANISATION AND ACCESS

1.2.1 Types of Files

Text Files:

- Contains data stored according to a defined character code (ASCII or Unicode)
- Can be created using a text editor
- Data appears as readable characters

Binary Files:

- Designed for storing data to be used by a computer program
- Stores data in its internal representation
- Created using specific programs
- Structure: File → Records → Fields → Values

1.2.2 Methods of File Organisation

Serial Files:

- Records have no defined order
- Stored one after another in the order they were added
- New records added at the end of the file
- No end of record character (must have defined format)

Advantages:

- Simple task
- Low cost

Disadvantages:

- Difficult to access specific records
- Must read all preceding records
- Cannot support modern high-speed requirements

File Access: Sequential access only

Uses:

- Batch processing
- Backing up data on magnetic tape
- Bank transactions

Sequential Files:

- Records ordered using a key field
- Key field values must be unique and ordered
- More efficient than serial files due to data integrity
- New records must be added in correct position

File Access Methods:

1. **Sequential Access:** Read key field values until required value found
2. **Direct Access:** Use index to look up address of record location

Random Files:

- Records stored randomly
- Accessed directly using hashing algorithm
- Uses hashing on record's key field to calculate address
- Well suited for magnetic and optical disks

Advantages:

- Quick retrieval of records
- Records may vary in size

1.2.3 Hashing Algorithms

Definition: Takes the key field as input and outputs a value for the record's position relative to the file's start.

Example:

```
<TEXT>
If key field is numeric: Divide by suitable large number and use remainder
Position = Key MOD FileSize
```

Handling Collisions: When two different keys produce the same position, use the next available position in the file.

File Access:

- Value in key field submitted to hashing algorithm
- Algorithm provides position in file
- May require short linear search due to collisions

1.3 FLOATING-POINT NUMBERS

1.3.1 Floating-Point Representation

Definition: The approximate representation of a real number using binary digits.

Format:

<TEXT>

```
Number = ±Mantissa × Base^Exponent
```

- **Mantissa:** The non-zero part of the number
- **Exponent:** The power to which the base is raised
- **Base:** 2 (in binary floating-point)

1.3.2 Converting Denary to Floating-Point Binary

Steps:

1. Convert whole number part to binary
2. Add sign bit (0 for positive, 1 for negative)
3. Convert fractional part by multiplying by 2 and recording whole parts
4. Combine parts and adjust exponent
5. Normalise the number

Example: Converting 8.75 to floating-point (8-bit: 4 for mantissa, 4 for exponent)

<TEXT>

```
Step 1: Convert whole number
8 = 1000

Step 2: Convert fractional part
0.75 × 2 = 1.5 → 1
0.5 × 2 = 1.0 → 1
0.0 × 2 = 0 → 0
0.75 = 0.11 (binary)

Step 3: Combine
8.75 = 1000.11

Step 4: Normalise
1000.11 = 0.100011 × 2^4

Step 5: Represent
Mantissa: 10001100 (0.100011 with padding)
Exponent: 0100 (4 in 4-bit two's complement)
```

1.3.3 Normalisation

Purpose:

- Maximise precision using all available bits in mantissa
- Ensure most significant bits are different (0 1 for positive, 1 0 for negative)
- Avoid multiple representations of the same number

Process:

- For positive numbers: Shift left until first bit after binary point is 1
- For negative numbers: Shift left until first two bits are different (10)
- Each shift left decreases exponent by 1

1.3.4 Problems with Floating-Point Numbers

1. **Rounding Errors:**
   - Binary representation of fractions is often approximate
   - Can become significant after multiple calculations
2. **Overflow:**
   - Result too large to represent
   - Occurs when exponent exceeds maximum
3. **Underflow:**
   - Result too small to represent
   - May be rounded to zero
4. **Inability to Store Zero:**
   - Normalised form requires mantissa to be 0.1 or 1.0
   - Zero must be handled as special case

1.3.5 Precision vs Range

Increase	Effect
Mantissa bits	Better precision
Exponent bits	Larger range

Trade-off: Must balance precision and range based on application needs

---

Revision #1
Created 2026-03-16 12:14:29 UTC by Samuel Lee

Updated 2026-03-16 12:15:27 UTC by Samuel Lee