Syllabus Content:

13. Data representation

13.1- User-defined data types

- 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 (Non-Composite)
- Including set, record and class / object (composite)

13.2 File organisation and access

- 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

Notes and guidance

- Including serial, sequential (using a key field), random (using a record key)
- Sequential access for serial and sequential files
- Direct access for sequential and random files

User defined Data Type

You have already met a variety of built-in data types with integers, strings, chars and more. But often these limited data types aren't enough and a programmer wants to build their own data types. Just as an integer is restricted to "a whole number from -2,147,483,648 through 2,147,483,647", user-defined data types have limits placed on their use by the programmer.

A **user defined data type** is a feature in most high level programming languages which allows a user (programmer) to define data type according to his/her own requirements

There are two categories of user defined data types.:

1. **Non-composite data type**
   a. Enumerated data type
   b. Pointer data type
2. **Composite**
   a. Record data type
   b. Set data type
   c. Objects and classes

Non-composite user-defined data type:

A non-composite data type is defined without referencing another data type. They don't combine different built-in data types in one data type. Non-Composite data types are:

- Enumerated data type
- Pointers

Enumerated data type

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

```
TYPE <identifier> = (value1, value2, value3, …)
```

Pseudocode of data Type for **Working Days** can be declared as **TWdays:**

Types names are normally start with **T** to facilitate the programmer

```
TYPE TWdays = (Monday, Tuesday, Wednesday, Thursday, Friday)
```

These values are ordinal and you can use mathematical and comparison operators on them.

These values are not strings so not enclosed in **"quotation marks"**

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

```
DECLARE Workdays : TWdays
DECLARE  thisday  : TWdays
DECLARE  nextday  : TWdays
today   ←   Monday
nextday ←   Monday + 1
```

nextday is now set to Tuesday by applying mathematical operator +

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.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

Enumerated data type in vb.net

Enum Example

When you are in a situation to have a number of constants that are logically related to each other, you can define them together these constants in an enumerator list. An enumerated type is declared using the enum keyword.

Syntax:

```
Enum enumerationname [ As datatype ]
  memberlist
End Enum
```

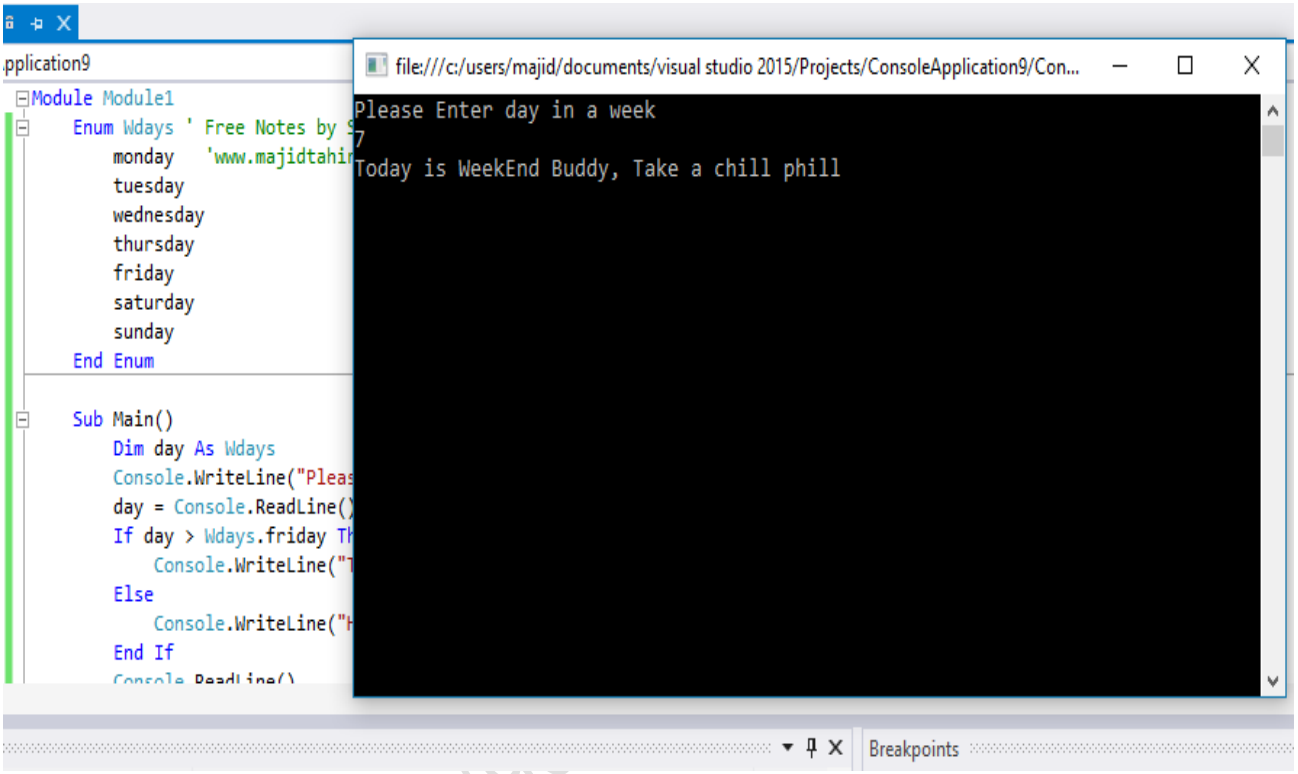
Enumerated declaration in VB (Sample Program)

Module1.vb*

ConsoleApplication9 | Module1 | Main

```
Module Module1
    Enum Wdays ' Free Notes by Sir Majid Tahir
        monday   'www.majidtahir.com
        tuesday
        wednesday
        thursday
        friday
        saturday
        sunday
    End Enum

    Sub Main()
        Dim day As Wdays
        Console.WriteLine("Please Enter day in a week")
        day = Console.ReadLine()
        If day > Wdays.friday Then
            Console.WriteLine("Today is WeekEnd Buddy, Take a chill phill ")
        Else
            Console.WriteLine("Hurry up buddy, You are getting late for work, Today is " & day)
        End If
        Console.ReadLine()
    End Sub

End Module
```

When the code is executed, following output will be produced:



An enumeration data type has a name, an underlying data type, and a set of members.

Each member represents a constant.

It is useful when you have a set of values that are functionally significant and fixed.

Another example of Enumerated Data Type in Visual Basic is given below:

Example

The following example demonstrates declaration and use of the Enum variable *Colors*:

```
Module constantsNenum
   Enum Colors
      red = 1
      orange = 2
      yellow = 3
      green = 4
      azure = 5
      blue = 6
      violet = 7
   End Enum
   Sub Main()
      Console.WriteLine("The Color Red is : " & Colors.red)
      Console.WriteLine("The Color Yellow is : " & Colors.yellow)
      Console.WriteLine("The Color Blue is : " & Colors.blue)
      Console.WriteLine("The Color Green is : " & Colors.green)
      Console.ReadKey()
   End Sub
End Module
```

When the above code is compiled and executed, it produces the following result:

```
The Color Red is: 1
The Color Yellow is: 3
The Color Blue is: 6
The Color Green is: 4
```

Pointer Data Type:

A pointer data type is used to reference a memory location.

This data type needs to have information about the type of data that will be stored in the memory location. In pseudocode the type definition has the following structure, in which **^** shows that the type being declared is a pointer and **<TYPENAME>** is the type of data to be found in the memory location: for example **INTEGER** or **REAL**: or any user-defined data type.

```
TYPE <pointer> = ^<Typename>
```

For example: a pointer for months of the year could be defined as follows:

```
TYPE TmonthPointer = ^Tmonth
DECLARE monthPointer : TmonthPointer
```

Tmonth is the data type in the memory location that this pointer can be used to point to

It could then be used as follows:

```
monthPointer ← ^thisMonth
```

If the contents of the memory location are required rather than the address of the memory location: then the pointer can be **dereferenced**. For example: **myMonth** can be set to the value stored at the address **monthpointer** is pointing to:

```
DECLARE myMonth : Tmonth
myMonth ← monthPointer^
```

monthPointer has been dereferenced

DECLARE myMonth : month myMonth e-monthPointer
monthPointer has been dereferenced

- The pointer data type is unique among the FreeBasic numeric data types.
- Instead of containing data, like the other numeric types, a pointer contains the memory address of data.

The main non-composite, derived type is the pointer, a data type whose value refers directly to (or "points to") another value stored elsewhere in the computer memory using its address.

It is a primitive kind of reference. (In everyday terms, a page number in a book could be considered a piece of data that refers to another one). Pointers are often stored in a format similar to an integer; however, attempting to dereference or "look up" a pointer whose value was never a valid memory address would cause a program to crash. To ameliorate this potential problem, pointers are considered a separate type to the type of data they point to, even if the underlying representation is the same.

As you can see there are three basic steps to using pointers.

1. Declare a pointer variable.
2. Initialize the pointer to a memory address.
3. Dereference the pointer to manipulate the data at the pointed-to memory location.

This isn't really any different than using a standard variable, and you use pointers in much the same way as standard variables. The only real difference between the two is that in a standard variable, you can access the data directly, and with a pointer you must dereference the pointer to interact with the data.

A pointer *references* a location in memory, and obtaining the value stored at that location is known as *dereferencing* the pointer. As an analogy, a page number in a book's index could be considered a pointer to the corresponding page; dereferencing such a pointer would be done by flipping to the page with the given page number and reading the text found on the indexed page.

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 th en 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.

WORKED EXAMPLE 26.01

Using records

A car manufacturer and seller wants to store details about cars. These details can be stored in a record structure:

```
TYPE CarRecord
   DECLARE VehicleID            : STRING  // unique identifier and record key
   DECLARE Registration         : STRING
   DECLARE DateOfRegistration   : DATE
   DECLARE EngineSize           : INTEGER
   DECLARE PurchasePrice        : CURRENCY
ENDTYPE
```

To declare a variable of that type we write:

```
DECLARE ThisCar : CarRecord
```

Note that we can declare arrays of records. If we want to store the details of 100 cars, we declare an array of type `CarRecord`

```
DECLARE Car[1:100] OF CarRecord
```

Records in VB:

VB.NET

```
Structure CarRecord
    Dim VehicleID As String
    Dim Registration As String
    Dim DateOfRegistration As Date
    Dim EngineSize As Integer
    Dim PurchasePrice As Decimal
End Structure
Dim ThisCar As CarRecord   ' declare a variable of CarRecord type
Dim Car(100) As CarRecord  ' declare an array of CarRecord type

ThisCar.EngineSize = 2500  ' assign value to a field
Car(2).EngineSize = 2500   ' assign value to a field of 2nd car in array
```

Sample code of Record Data Type in VB and its Output below:

Module1.vb*

ConsoleApplication10 | Module1 | Main

```
Module Module1
    Structure CarRecord ' Declaration Of RECORD DATA TYPE IN VB (Notes by Sir Majid Tahir)
        Dim carID As String '                                   www.majidtahir.com
        Dim registration As String
        Dim dateofregistraion As Date
        Dim Price As Double
        Dim enginesize As Integer
    End Structure

    Sub Main()
        Dim newcar As CarRecord
        Console.WriteLine("Please Enter CarID")
        newcar.carID = Console.ReadLine()
        Console.WriteLine("Please Enter Date Of Registration")
        newcar.dateofregistraion = Console.ReadLine()
        Console.WriteLine("Please Enter Car Engine Size")
        newcar.enginesize = Console.ReadLine()
        Console.WriteLine("Please Enter CarRegistration number")
        newcar.registration = Console.ReadLine()
        Console.WriteLine("Please Enter Car Price")
        newcar.Price = Console.ReadLine()

        Console.WriteLine("Please Verify CarID = " & newcar.carID)
        Console.WriteLine("Registration Date =  " & newcar.dateofregistraion)
        Console.WriteLine("Engine size = " & newcar.enginesize)
        Console.WriteLine("Registration number =  " & newcar.registration)
        Console.WriteLine("Price = " & newcar.Price)
        Console.ReadLine()
    End Sub
End Module
```

100 %

```
Application10                    Module1                    Main
Module Module1
    Structure CarRecord ' Declaration Of RECORD DATA TYPE IN VB (Notes by Sir Majid Tahir)
        Dim carID As String '                                   www.majidtahir.com
        Dim registration As String
        Dim dateofregistraion As Date
        Dim Price As Double
        Dim enginesize As Integer
    End Structure

    Sub Main()
        Dim newcar As CarRecord
        Console.WriteLine("Please Enter C
        newcar.carID = Console.ReadLine()
        Console.WriteLine("Please Enter D
        newcar.dateofregistraion = Consol
        Console.WriteLine("Please Enter C
        newcar.enginesize = Console.Readl
        Console.WriteLine("Please Enter C
        newcar.registration = Console.Rea
        Console.WriteLine("Please Enter C
        newcar.Price = Console.ReadLine()
```

```
file:///c:/users/majid/documents/visual studio 201...
Please Enter CarID
CA1234
Please Enter Date Of Registration
28/09/2018
Please Enter Car Engine Size
2500
Please Enter CarRegistration number
LZD 21
Please Enter Car Price
500000
Please Verify CarID = CA1234
Registration Date =  28/09/2018
Engine size = 2500
Registration number =  LZD 21
Price = 500000
```

Record Data type with Arrays in VB.net:

```
odule1.vb*
Module1                                  Main
Module Module1
    'Notes of Sir Majid Tahir at www.majidtahir.com
    Structure CarRecord 'Declaration Of Record Data Type
        Dim CarID As String
        Dim CarRegistration As String
        Dim DateOfRegistration As Date
        Dim CarManufacturer As String
        Dim EngineSize As Integer
        Dim SellingPrice As Double
    End Structure
    Sub Main()
        Dim newcar(5) As CarRecord '1 Dimension ARRAY of Record Data type declared
        For count = 1 To 5
            Console.WriteLine("Please Enter Car ID =" & count)
            newcar(count).CarID = Console.ReadLine()
            Console.WriteLine("Please Enter Car Registration = " & count)
            newcar(count).CarRegistration = Console.ReadLine()
            Console.WriteLine("Please Enter Car Manufacturer = " & count)
            newcar(count).CarManufacturer = Console.ReadLine()
            Console.WriteLine("Please Enter Car EngineSize = " & count)
            newcar(count).EngineSize = Console.ReadLine()
            Console.WriteLine("Please Enter Car Sale Price =" & count)
            newcar(count).SellingPrice = Console.ReadLine()
        Next
        For count = 1 To 5
            Console.WriteLine(count & " Car ID = " & newcar(count).CarID)
            Console.WriteLine(count & " Car Registration = " & newcar(count).CarRegistration)
            Console.WriteLine(count & " Date Of Registration = " & newcar(count).DateOfRegistration)
            Console.WriteLine(count & " Car Manufacturer = " & newcar(count).CarManufacturer)
            Console.WriteLine(count & " Car Engine size = " & newcar(count).EngineSize)
            Console.WriteLine(count & " Car Sale Price = " & newcar(count).SellingPrice)
        Next
        Console.ReadKey()
    End Sub
```

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.

A set is an unordered collection of items. Every element is unique (no duplicates) and must be immutable (which cannot be changed).However, the set itself is mutable. We can add or remove items from it. The number of elements in a SET data type can vary, but no nulls are allowed.

Set can have any number of items and they may be of different types (integer, float, string etc.).

Assume we wish to create a **Set.** In PSEUDOCODE the type definition has this structure.

```
TYPE <set-identifier> = SET OF <Basetype>
```

The variable defination of set includes the elements of the set.

```
DECLARE <identifier> (value1, value 2, value3, …):
```

A set of vowels can be declared as follows:

```
TYPE Sletter = SET OF CHAR
DECLARE vowel ('a' , 'e' , 'i' , 'o' , 'u') : Sletter
```

Or we can declare sets in Pseudocodes this way
Here is another example

```
TYPE Sizes = SET OF String
DECLARE SweaterSizes : (XXLarge, XLarge, Large, Medium, Small, XSmall)

DECLARE Title: SET OF  (Mr, Mrs, Ms, Miss, Dr)
```

NOTE: Of the three programming languages, **Pascal** and **Python** support the set data type.

The following Python code illustrates that the set data type has operators – union and intersection- which can be used in two sets:

How to create a set in PYTHON?

A set is created by placing all the items (elements) inside curly braces { }, separated by comma or by using the built-in function `set()`.
It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have a mutable element, like list, set or dictionary, as its element.

script.py IPython Shell

```
# set of integers
my_set = {1, 2, 3}
print(my_set)

# set of mixed datatypes
my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)
```

You can try creating SET and its operations online on below link:
https://www.programiz.com/python-programming/set#operations

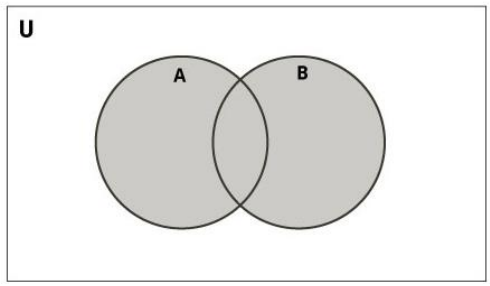
Python Set Operations

Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Let us consider the following two sets for the following operations.

```
>>> A = {1, 2, 3, 4, 5}
>>> B = {4, 5, 6, 7, 8}
```

Set Union

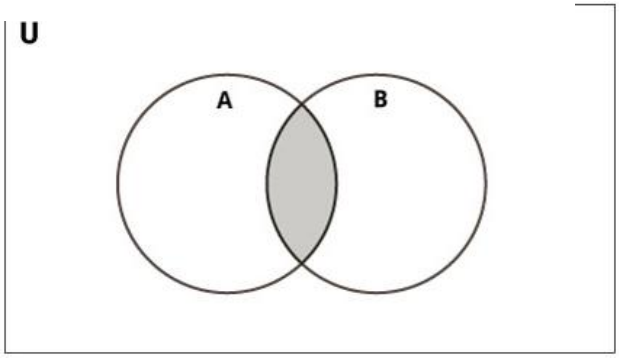


```python
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use | operator
# Output: {1, 2, 3, 4, 5, 6, 7, 8}
print(A | B)
```

Union of **A** and **B** is a set of all elements from both sets.
Union is performed using **|** operator. Same can be accomplished using the method **union()**.

Set Intersection

```python
# initialize A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

# use & operator
# Output: {4, 5}
print(A & B)
```



Intersection of `A` and `B` is a set of elements that are common in both sets.
Intersection is performed using `&` operator. Same can be accomplished using the method `intersection()`.

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.

Syllabus Content:

13.1 Data representation

13.2 File organisation and access

- file organisation: serial, sequential (using a key field) and random (using record key)
- file access: sequential access (serial & sequential files) – direct access (sequential & random files)
- select an appropriate method of file organisation and file access for a given problem

(Sec 13.2)

File Organisation and Access:

Serial Files:

Serial file organisation is the simplest file organisation method. In serial files, records are entered in the order of their creation. As such, the file is unordered, and is at best in chronological order. Serial files are primarily used as transaction files in which the transactions are recorded in the order that they occur.

Sequential Files:

The key difference between a sequential file and a serial file is that it is ordered in a logical sequence based on a key field. This key is usually the primary key, though secondary keys may be used as well. Sequential files are therefore files that are sorted based on some key values.

Sequential files are primarily used in applications where there is a high file hit rate. Hit rate is a measure of the proportion of the records that is accessed in a single run of the application. Therefore, sequential files are ideal for master files and batch processing applications such as payroll systems in which almost all records are processed in a single run of the application.

Random Files:

In random file organisation, records are stored in random order within the file. Though there is no sequencing to the placement of the records, there is however, a pre-defined relationship between the key of the record and its location within the file. In other

words, the value of the record key is mapped by an established function to the address within the file where it resides. Therefore, any record within the file can be directly accessed through the mapping function in roughly the same amount of time. The location of the record within the file therefore is not a factor in the access time of the record. As such, random files are also known in some literature as direct access files.

To create and maintain a random file, a mapping function must be established between the record key and the address where the record is held. If M is the mapping function, then

M(value of record key) => address of record

Hashing

There are various mapping techniques. Some involve using the key field to directly map to the location with the file, while others refer to some lookup table for the location. However, the more common method is to employ a hash function to derive the address.

Hashing is the process of transforming the key value of a record to yield an address location where the record is stored. In some literatures, it is also known as Key-to-Address Transformation, Address-Calculation, Scatter Storage, or Randomization.

A hash function generates the record address by performing some simple operations on the key or parts of the key. A good hashing function should be

- quick to calculate
- cover the complete range of the address space.
- give an even distribution
- not generate addresses that tend to cluster within a few locations, thus resulting in frequent collisions.

Refrences

- Book: AS and A-level Computer Science by
- http://www.bbc.co.uk/education/guides/zjfgjxs/revision/1
- https://www.tutorialspoint.com/vb.net/vb.net_constants.htm
- https://www.cs.drexel.edu/~introcs/F2K/lectures/5_Scientific/overflow.html
- https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/underflow.html