3.1.1 User-defined data types

You have already studied a variety of built-in datatypes such as: integers, strings, chars and more. But often these limited datatypes fail to meet the programmer's demand which forces him/her to build their own datatypes. Just as an integer is restricted to "a whole number from -2,147,483,648 through 2,147,483,647", user-defined datatypes have boundaries places according to the programmer's need.

There are 2 categories of user defined data types:

- **Composite**
  - Set
  - Record
  - Class/Object
- **Non-composite**
  - Enumerated (enum)
  - Pointers

Non-composite data types

Enumeration

If you are using lots of constants in your program that are all related to each other, then it is a good idea to keep them together using a structure called an "Enum". For example, you might want to store the names of each set of cards in a deck.

Here's one way of writing that in code:

```
Const heart as integer = 1
Const club as integer = 2
Const spade as integer = 3
Const diamond as integer = 4

dim cardset as string
cardset = spade
```

We can bring them all together in a nice organized enum:

3.1.1 User-defined data types

```
Enum suits
   HEARTS = 1
   CLUBS = 2
   SPADES = 3
   DIAMONDS = 4
End Enum
dim cardset as suits
cardset = suits.HEARTS
```

This allows you to set meaningful names to the enum and its members, making it much easier to remember as well as making the code easier to read.

We may also create separate constants to store the points of a football match

```
Const Win as Integer = 3
Const Draw as Integer = 1
Const Lose as Integer = 0
```

With enums, we could create a datatype called “Result” and store the points within it, under an easy to remember name.

```
Enum Result
    Win = 3
    Lose = 1
    Draw = 0
End Enum

dim ManUvChelsea as Result
ManUvChelsea = Result.Win
Console.Writeline("ManU scored " & ManUvChelsea & " points"
```

3.1.1 User-defined data types

Pointers

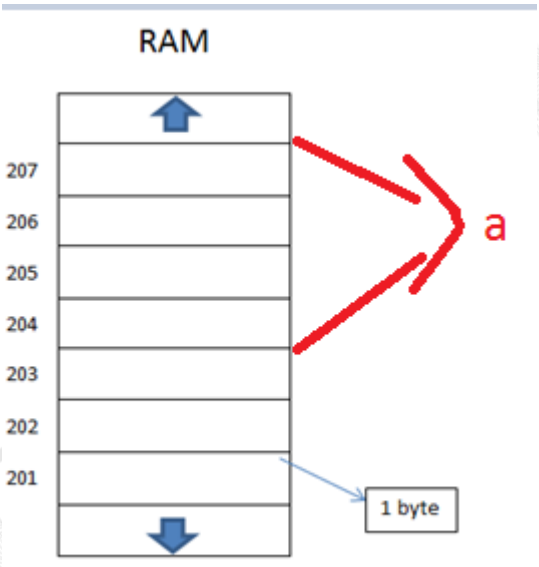
A pointer is a variable that represents the location of a particular data item within a specified domain (such as the hard drive, or an array). Within the computer's memory, every stored data item occupies one or more contiguous memory cell/s. The number of memory cells needed to store a data item depends on the data type of that item.

For example, a single character will typically be stored in 1 byte of memory; an integer usually requires 4 contiguous bytes, a floating-point number usually requires 4 contiguous bytes, and so on…

To understand pointers, you must first understand how data is stored in the computer's memory

In typical computer architecture of the memory, each byte of the memory has a unique address. Let us assume the first byte has an address of "201". Then the next address will be "202" and we'll go on "203","204" and so on…

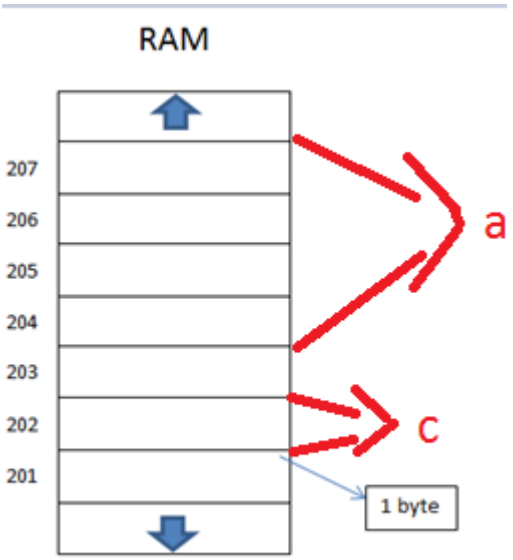
Now when we initialize a variable, the computer allocates some amount of memory corresponding to this particular variable depending on the data type of the variable.
So for example 'a' as type integer that means the computer allocates 4 bytes of memory, we can allocate memory from '204' to '207'



Lookup Table		
Variable name	Type	Located at:
a	integer	204

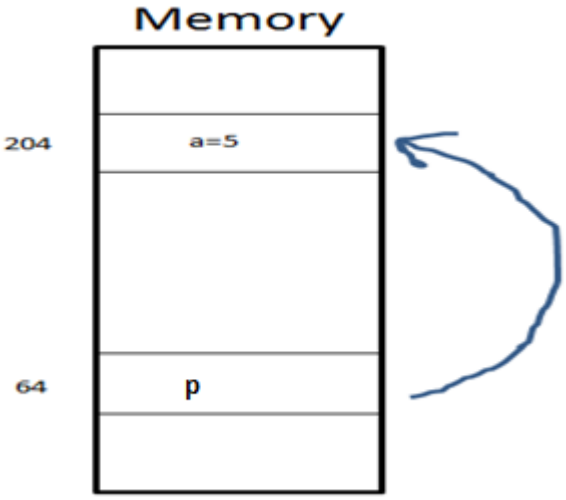
3.1.1 User-defined data types

And the computer has an internal structure called a **lookup table** where it stores data such as the variable name, its data type, and where it resides in the memory. Now if we declare another variable for instance, character 'c'. Once again when the machine sees this declaration, it knows that it is a character variable, so it looks for a free byte and places 'c' over there. In this case stored at '202'



Lookup Table		
Variable name	Type	Located at:
a	integer	204
c	character	202

Now the main question is, can we know the address of a variable in our program? Yes we can! Using the concept of "pointers".
We can have another variable, type of which is a pointer 'p'. Now this variable 'p' can store the address of 'a'. 'p' also takes some memory, so let's say that it's stored at location address 64 and it also takes 4 bytes of memory.

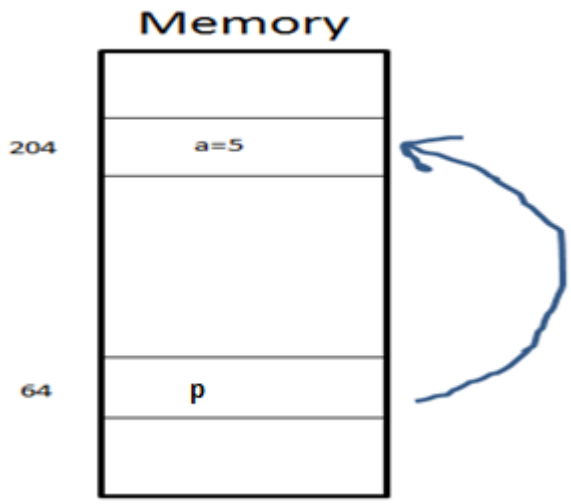



```
Int a;
Int *p;     / /is the pointer
p = &a   / /assigning address of 'a' to 'p'
a=5;
Print p      / /output is 204
Print &a  / /output is 204
Print &p / /output is 64
```

(Coding in C language)

3.1.1 User-defined data types

There is one more important feature of pointers. If we put as asterisk (*) in front of the pointer when declaring the variable, then it gives us the value of the variable that it points to. This concept is known as **“de-referencing”**




```
Int a;
Int *p;     / /is the pointer
p = &a   / /assigning address of ‘a’ to ‘p’
a=5;
Print p      / / 204
Print &a  / / 204
Print &p / / 64
Print *p / / 5


(Coding in C language)
```

‘p’ <points to> address

‘*p’ <points to> value at address

3.1.1 User-defined data types

Composite data types:

Set:

It is a composite data type that can store data in any particular order, in which each element is indexed. Sets cannot have data items repeated more than once

Constructing Sets

One way to construct sets is by passing any sequential object to the "set" constructor.

```
>>> set([0, 1, 2, 3])
set([0, 1, 2, 3])
>>> set("obtuse")
set(['b', 'e', 'o', 's', 'u', 't'])
```

We can also add elements to sets one by one, using the "add" function.

```
>>> s = set([12, 26, 54])
>>> s.add(32)
>>> s
set([32, 26, 12, 54])
```

The set function also provides a copy constructor. However, remember that the copy constructor will copy the set, but not the individual elements.

```
>>> s2 = s.copy()
>>> s2
set([32, 9, 12, 14, 54, 26])
```

3.1.1 User-defined data types

Membership testing

We can check if an object is in the set using the same "in" operator as with sequential data types.

```
>>> 32 in s
True
>>> 6 in s
False
>>> 6 not in s
True
```

We can also test the membership of entire sets. Given two sets S_1 and S_2, we check if S_1 is a subset or a superset of S_2.

```
>>> s.issubset(set([32, 8, 9, 12, 14, -4, 54, 26, 19]))
True
>>> s.issuperset(set([9, 12]))
True
```

Removing Items

There is the “remove” function to remove a specified element from the set

```
>>> s.remove(3)
>>> s
set([2,4,5,6])
```

However, removing an item which isn’t in the set causes an error.

3.1.1 User-defined data types

Intersection

Any element which is in both S_1 and S_2 will appear in their intersection.

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.intersection(s2)
```

Union

The union is the merger of 2 sets. Any element in S_1 or S_2 will appear in their union

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.union(s2)
```

Set difference

It will return all the elements that are in S_1 but not in S_2

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.difference(s2)
set([9, 4])
```

frozenset

A frozenset is basically the same as a set, except that its members cannot be changed. This means that they can be used as members in other sets. frozensets have the same functions as normal sets, except the functions which change the contents (add,remove,update, etc.)

3.1.1 User-defined data types

Record:

A record is a value that contains other values, indexed by names.

Field – an element of a record

Records are collections of data items (fields) stored about something. They allow you to combine several data items (or fields) into one variable. An example at your college they will have a database of records for each student. This student record would contain fields such as ID, Name, and Date of Birth.

```
Dim studentID As Integer
    Dim studentName As String
    Dim studentDoB As Date
Sub Main()
     Console.write("insert the id: ")
     newStudentid = console.readline()
     console.write("insert the name: ")
     newStudentname = console.readline()
     console.write("insert the Date of Birth: ")
     newStudentDoB = console.readline()

     console.writeline("new record created: " & newStudentid & " " & newStudentname & " " & newStudentDoB)
End Sub
```

```
Code Output
insert the id: 12
insert the name: Nigel
insert the Date of Birth: 12/12/1994
new record created: 12 Nigel 12/12/1994
```

This code helps store data about a student in record form, however after entering the data of one student, the program will terminate. How can we edit this code to allow for multiple records to be added?

3.1.1 User-defined data types

One way of doing this is defining more fields to accommodate more data entry.
i.e. 3 fields for each student

```
    Dim studentID1 As Integer 'field
    Dim studentName1 As String 'field
    Dim studentDoB1 As Date 'field
    Dim studentID2 As Integer 'field
    Dim studentName2 As String 'field
    Dim studentDoB2 As Date 'field
    Dim studentID3 As Integer 'field
    Dim studentName3 As String 'field
    Dim studentDoB3 As Date 'field
    ...
    ...
    Dim studentID 400 As Integer 'field
    Dim studentName400 As String 'field
    Dim studentDoB400 As Date 'field
```

It would take an awfully long time to declare them all, as well as writing data in them. So how do we solve this? Well we need to combine two things we've already learnt so far, the **record** and the **array**. We are going to make an array of student records.

```
Structure student 'record declaration
        Dim id As Integer 'field
        Dim name As String 'field
        Dim DoB As Date 'field
    End Structure

    Sub Main()
        Dim newStudents(400) As student 'declare an array of
student records,  a school with 401 students

        for x = 0 to 400 'insert the details for each student
            Console.WriteLine("insert the id")
            newStudents(x).id = Console.ReadLine()
            Console.WriteLine("insert the name")
            newStudents(x).name = Console.ReadLine()
            Console.WriteLine("insert the Date of Birth")
            newStudents(x).DoB = Console.ReadLine()
        next
        for x = 0 to 400 'print out each student
            Console.WriteLine("new record created: " &
newStudents(x).id & " " & newStudents(x).name & " " &
newStudents(x).DoB)
        next
    End Sub
```

3.1.1 User-defined data types

This seems to solve our problem, you are free to try it out yourself but decrease the number of students slightly!

Exercise 1: Declare a record called “player” to store the following Role Playing Game attributes: health, name, class (barbarian, wizard, elf), gold, gender

```
Enum type
   WIZARD
   BARBARIAN
   ELF
End Enum

Structure player 'remember Visual Basic uses structure instead of record
   name as string
   health as integer
   gold as integer
   gender as char
   ' class as string ' you might have already noticed, the word class is 'reserved'
   ' this means it is has a special purpose in VBNET and we'll have to use another
   characterclass as type 'a string would work, but it's better to use an enum
End player
```

Exercise 2: Create 2 characters, Gandolf and Conan using the player record

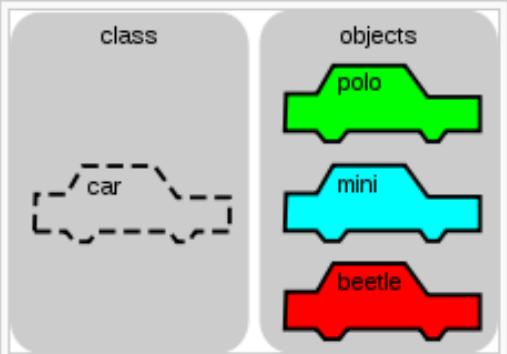
```
'you can of course give them different attributes
Dim Gandolf As player
Gandolf.name = "Gandolf"
Gandolf.health = 70
Gandolf.gold = 50
Gandolf.gender = "m"
Gandolf.class = type.WIZARD

Dim Conan As player
Conan.name = "Conan"
Conan.health = 100
Conan.gold = 30
Conan.gender = "m"
Conan.class = type.BARBARIAN
```

3.1.1 User-defined data types

Class/Object :

In object-oriented programming, a **class** is a construct that is used as a blueprint (or template) to create objects of that class. This blueprint describes the state and behavior that the objects of the class all share. An object of a given class is called an instance of the class. All the instances of a class have similar properties. For example, you can define a class called “Car” and create three instances of the class “Car” for “Polo”, “Mini” and “Beetle”.



You can use the same class as a template to make (instantiate) many objects

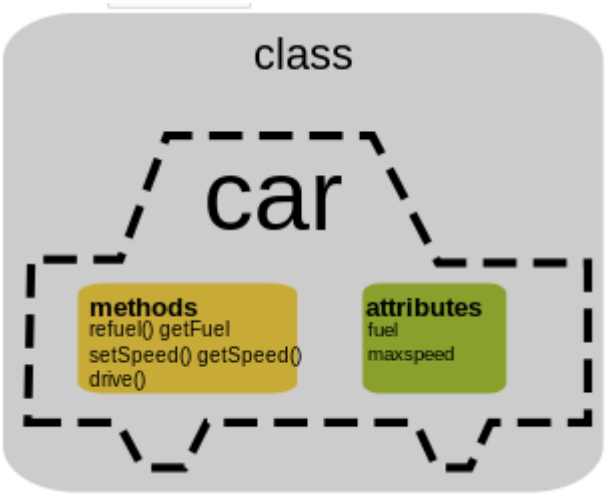
Structures (Records) are very similar to Classes in that they collect data together. However, classes extend this idea and are made from two different things:
Let’s take a look at the following example:

```
class car
  private maxSpeed as integer
  public fuel as integer
  public sub setSpeed(byVal s as integer)
    maxSpeed = s
  end sub
  public function getSpeed() as integer
    return maxSpeed
  end function
  public sub refuel(byVal x as integer)
    console.writeline("pumping gas!")
    fuel = fuel + x
  end sub
  public function getFuel() as integer
    return fuel
  end function
  public sub drive()
    fuel = fuel - 1
  end sub
end class
```

3.1.1 User-defined data types

You can see that the class is called ‘car’ and it has:

- 2 attributes:
  - maxSpeed
  - fuel
- 4 methods
  - 3 procedures:
    - setSpeed
    - refuel
    - drive
  - 1 function:
    - getSpeed



Remember this is a class and therefore only a template, we need to ‘create’ it using an object.

Attributes

These store information about the object. In the example above we store the fuel and maxSpeed. The attributes are attached to the classes, and if there are several instances (objects) of the classes then each will store its own version of these variables. The terms ‘private’ and ‘public’ are substitutes of the term ‘dim’ and belongs to another section in OOP.

Methods

Unlike structures, OOP allows you to attach functions and procedures to your code. This means that not only can you store details about your car (attributes), you can also allow for sub routines such as ‘drive()’ and ‘refuel’. Which are attached to each class.

3.1.1 User-defined data types

What is the difference between a class and an object?

- A class is a template which cannot be executed
- An object is an instance of a class which can be executed
- One class can be used to make many objects

What are the main components of a class?

- Attributes
- Methods

What is the difference between a structure and a class?

- Structures do not have any methods