

Exercise walk-through

13.1.1

Designing a user-defined type for a problem, and the linked list

A-Level Computer Science

Why define your own types?

Built-in types (INTEGER, STRING, ...) are too loose for real data.

Restrict

a STRING will hold nonsense; an enumerated type will not

Group

a taxi is a name *and* a capacity *and* a flag

Name

DECLARE Taxi :
Vehicle reads
as what it is

User-defined types
documenting.

make the compiler stricter and the code self-

Why they are needed

A built-in `STRING` lets you store nonsense in a field that should hold one of a few legal values; a user-defined type can restrict it.

Real entities are usually a **collection** of values of different types.

Enumerated type – a fixed named list

An **enumerated type** has values that are a **fixed list of named constants**.

A fleet of taxis

```
TYPE Vehicle = (M100, M230, T101,  
                T102, T120, T150)  
DECLARE MyTaxi : Vehicle  
MyTaxi ← T102
```

You cannot assign anything outside the list.
Internally the names are small integers.

Vehicle

M100

M230

T101

T102

T120

An enumerated type, drawn

TYPE Vehicle =

M100

M230

T101

T102

T120

T150

MyTaxi may only hold one of these named values

The values are **named constants** in a fixed order, so they can be compared and counted – but they are not strings and not integers.

An **enumerated type** lists its own allowed values by name, so the compiler can reject anything else – clearer than remembering that 3 means Wednesday.

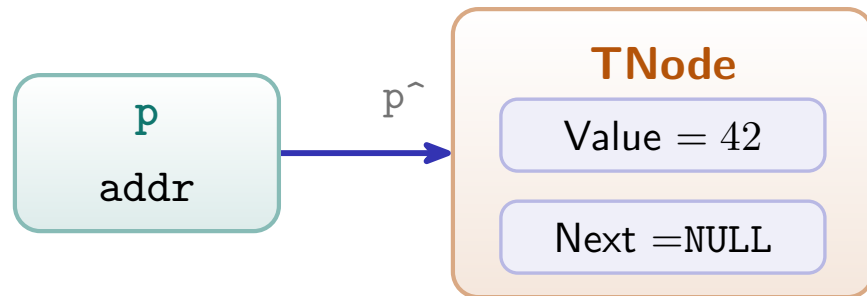
Pointer type – an address, not a value

A **pointer** holds the **memory address** of another variable (or NULL).

- builds dynamic structures: lists, trees
- passes a reference without copying
- **dereference** $p^\wedge$ reaches the target

Point at a node

```
TYPE PNode = ^TNode  
DECLARE p : PNode  
p ← NEW TNode  
p^.Value ← 42
```



Exercise 2.1

2.1 A program stores, for each member of a sports club, their name, their date of birth, and which one of the four teams Red, Blue, Green or Yellow they belong to. State the user-defined type you would use for the date and the one you would use for the team, and justify each choice. [3]

Exercise 2.1

2.1 A program stores, for each member of a sports club, their name, their date of birth, and which one of the four teams Red, Blue, Green or Yellow they belong to. State the user-defined type you would use for the date and the one you would use for the team, and justify each choice. [3]

2.1

- for the date, a **record** (a user-defined composite type)
- because a date has parts of its own – a day, a month and a year – which belong together in one field
- for the team, an **enumerated** type, because the value is exactly one of a fixed, known list of names, and declaring it that way lets the compiler reject anything else

Exercise 2.2

2.2 Declare a record type `DateType` with three integer fields for the day, month and year. [3]

Exercise 2.2

2.2 Declare a record type `DateType` with three integer fields for the day, month and year. [3]

2.2

```
TYPE DateType
    DECLARE Day : INTEGER
    DECLARE Month : INTEGER
    DECLARE Year : INTEGER
ENDTYPE
```

- `TYPE DateType ... ENDTYPE`
- three `DECLARE` lines inside it
- all three of type `INTEGER`

Exercise 2.3

2.3 Declare a record type `MemberType` whose fields are a name, a `DateType` and an enumerated type `TeamType`. You may assume `TeamType` has already been declared. [3]

Exercise 2.3

2.3 Declare a record type `MemberType` whose fields are a name, a `DateType` and an enumerated type `TeamType`. You may assume `TeamType` has already been declared. [3]

2.3

```
TYPE MemberType
    DECLARE Name : STRING
    DECLARE Born : DateType
    DECLARE Team : TeamType
ENDTYPE
```

- `TYPE MemberType ... ENDTYPE`
- a `STRING` field for the name
- fields of type `DateType` and `TeamType`

Exercises 2.4-2.5

2.4 State **two** advantages of a linked list over an array of records.

[2]

2.5 State what the `NextPtr` field of the **last** node in a linked list holds.

[1]

Exercises 2.4-2.5

2.4 State **two** advantages of a linked list over an array of records. [2]

2.4 any **two** of:

- it can grow and shrink while the program runs, so the number of items need not be known when it is declared
- inserting or deleting in the middle changes only two pointers, instead of shuffling every later item along
- memory is used one node at a time, so none is reserved for items that are never stored

2.5 State what the `NextPtr` field of the **last** node in a linked list holds. [1]

2.5

- `NULL` – there is no next node

Exercise 3.1

3.1 Which of these is a **non-composite** user-defined type?

[1]

☐

A record

B set

C enumerated

D class

Exercise 3.1

3.1 Which of these is a **non-composite** user-defined type?

[1]

☐

A record

B set

C enumerated

D class

3.1 C

- an enumerated type is a list of single named values, so it is **non-composite**
- a record, a set and a class each group several values together, so all three are composite

Exercise 3.2 (a)

3.2 A library program stores, for each book, its title, its author, the year it was published, and whether it is currently on the shelf, on loan or lost.

(a) Declare a suitable **enumerated** type `StatusType` for the last of these. [2]

Exercise 3.2 (a)

3.2 A library program stores, for each book, its title, its author, the year it was published, and whether it is currently on the shelf, on loan or lost.

(a) Declare a suitable **enumerated** type `StatusType` for the last of these. [2]

(a)

```
TYPE StatusType = (OnShelf, OnLoan, Lost)
```

- `TYPE StatusType = ( ... )`
- exactly the three values, and no others

Exercise 3.2 (b)

3.2 A library program stores, for each book, its title, its author, the year it was published, and whether it is currently on the shelf, on loan or lost.

(b) Declare a record type `BookType` for one book, using `StatusType`. [3]

Exercise 3.2 (b)

3.2 A library program stores, for each book, its title, its author, the year it was published, and whether it is currently on the shelf, on loan or lost.

(b) Declare a record type `BookType` for one book, using `StatusType`. [3]

(b)

```
TYPE BookType
  DECLARE Title : STRING
  DECLARE Author : STRING
  DECLARE YearPublished : INTEGER
  DECLARE Status : StatusType
ENDTYPE
```

- `TYPE BookType ... ENDTYPE`
- two `STRING` fields and an `INTEGER` field
- a field of type `StatusType`

Exercise 3.2 (c)

3.2 A library program stores, for each book, its title, its author, the year it was published, and whether it is currently on the shelf, on loan or lost.

(c) Declare an array **Catalogue** able to hold 500 books, and write the statement that sets the status of the first book in the array to on loan. [2]

Exercise 3.2 (c)

3.2 A library program stores, for each book, its title, its author, the year it was published, and whether it is currently on the shelf, on loan or lost.

(c) Declare an array **Catalogue** able to hold 500 books, and write the statement that sets the status of the first book in the array to on loan. [2]

(c)

```
DECLARE Catalogue : ARRAY[1:500] OF BookType  
Catalogue[1].Status ← OnLoan
```

- the array declaration, of 500 elements of **BookType**
- the assignment, reaching the field with a dot and using the enumerated value unquoted

Exercise 3.3 (a)

3.3 A program stores a list of integers in a linked list. Each node holds one integer and a pointer to the next node.

(a) Declare the pointer type **PNode** and the record type **NodeType**, in the correct order.[4]

Exercise 3.3 (a)

3.3 A program stores a list of integers in a linked list. Each node holds one integer and a pointer to the next node.

(a) Declare the pointer type **PNode** and the record type **NodeType**, in the correct order.[4]

(a)

```
TYPE PNode = ^NodeType

TYPE NodeType
    DECLARE Value : INTEGER
    DECLARE NextPtr : PNode
ENDTYPE
```

- `TYPE PNode = ^NodeType`
- the pointer type declared **first**, because the record needs a type that already exists
- `TYPE NodeType ... ENDTYPE` with an `INTEGER` field
- a `NextPtr` field of type `PNode`

Exercise 3.3 (b)

3.3 A program stores a list of integers in a linked list. Each node holds one integer and a pointer to the next node.

(b) The list is empty, and then the values 20, 12 and 7 are each inserted **at the front**, in that order. State the value in the node that **HeadPointer** points to afterwards, and state what the **NextPtr** of the node holding 20 contains. [3]

Exercise 3.3 (b)

3.3 A program stores a list of integers in a linked list. Each node holds one integer and a pointer to the next node.

(b) The list is empty, and then the values 20, 12 and 7 are each inserted **at the front**, in that order. State the value in the node that **HeadPointer** points to afterwards, and state what the **NextPtr** of the node holding 20 contains. [3]

(b)

- inserting at the front makes the **most recent** value the first one, so **HeadPointer** points to the node holding **7**
- 20 was inserted first, so nothing was ever attached after it
- its **NextPtr** therefore still holds **NULL**, and it is the last node in the list

Exercise 3.3 (c)

3.3 A program stores a list of integers in a linked list. Each node holds one integer and a pointer to the next node.

(c) Write pseudocode that outputs every value in the list, starting from `HeadPointer`.^[3]

Exercise 3.3 (c)

3.3 A program stores a list of integers in a linked list. Each node holds one integer and a pointer to the next node.

(c) Write pseudocode that outputs every value in the list, starting from `HeadPointer`. [3]

(c)

```
DECLARE Current : PNode
Current ← HeadPointer
WHILE Current <> NULL DO
    OUTPUT Current^.Value
    Current ← Current^.NextPtr
ENDWHILE
```

- start from the head: `Current ← HeadPointer`
- a loop that continues while `Current <> NULL`
- output the value and then move on with `Current ← Current^.NextPtr`, in that order
- $\triangle$ moving on before outputting loses the first value; testing `Current^.NextPtr <> NULL` instead loses the last one