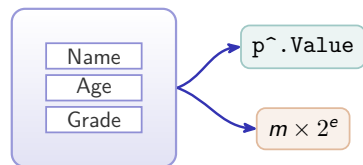


Data Representation

A-Level Computer Science ■ Topic 13

A-Level Computer Science



Data Representation

What we will cover

- 1 User-defined types
- 2 File organisation & access
- 3 Hashing
- 4 Floating-point numbers
- 5 Recap

1 User-defined types

Why define your own types?

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

Restrict

限制

a STRING will hold nonsense; an enumerated type will not

Group

组合

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

Name

自说明

DECLARE Taxi :
Vehicle reads
as what it is

User-defined types 用户定义类型 make the compiler stricter and the code self-documenting.

Enumerated type – a fixed named list

An **enumerated type** 枚举类型 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.

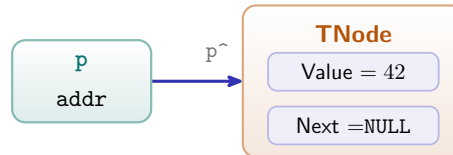
Pointer type – an address, not a value

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

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

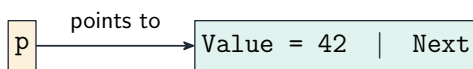
Point at a node

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



A pointer holds an address

a pointer holds an address



$p^{\wedge}$ dereferences — reach the node's fields, e.g. $p^{\wedge}$.Value

The variable stores **where** the data is, not the data — which is what lets a linked structure grow without moving anything.

Composite types – several values, one name

A **composite type** 复合类型 groups several values under one name.



Enumerated and pointer are **non-composite**. Record, set and class are composite.

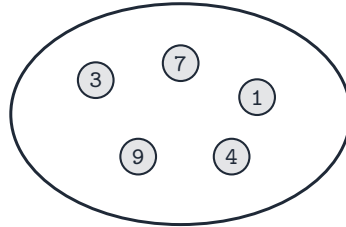
Two composite shapes

record Student

Name	: STRING
Age	: INTEGER
Grade	: CHAR
Enrolled	: BOOLEAN

one name, several fields of different types

A **record**: named fields of different types, held together.



unordered, every value unique

A **set**: unordered, no duplicates, tested with membership.

Choosing a type

Pick the type that matches the **kind of value**, not the first one that compiles.

a value from a fixed list

enumerated

a group of fields

record

an unordered unique collection

set

indirection / a linked structure

pointer

state *and* behaviour together

class

2

File organisation

Three ways to lay records out

File organisation 文件组织 is the layout; **file access** is how you reach a record.

serial

串行文件

records in the *order added*. Append is fast; search is slow. Logs.

sequential

顺序文件

sorted on a *key*. Search can stop early; insert must shift.

random

随机文件

position from the key (often a hash). Direct lookup is fast.

Serial – records in arrival order

— records stored in the order they were added —→

First record	Second record	Third record	Fourth record	Fifth record	Sixth record	and so on
-----------------	------------------	-----------------	------------------	-----------------	-----------------	--------------

↑
Start of file

New records go on the **end**. Simple to write, but any search reads from the start.

Sequential – ordered by a key

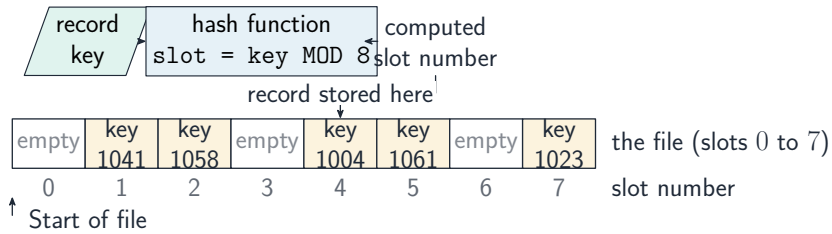
— records sorted into key order —→

Customer key = 100	Customer key = 101	Customer key = 102	Customer key = 104	Customer key = 105	Customer key = 106	and so on
-----------------------	-----------------------	-----------------------	-----------------------	-----------------------	-----------------------	--------------

↑
Start of file

Records are kept in **key order**, so a search can stop early — but an insertion means rewriting the file.

Random – the key computes the address



A **hash** 散列 of the key gives the record's position, so one read finds it — **direct access**, at the cost of managing collisions.

Sequential access vs direct access

- **sequential access** 顺序存取 – read from the start, one record after another
- **direct access** 直接存取 – jump straight to a known position

Serial and sequential files **need** sequential access. A random file **allows** direct access.

Match the layout to the dominant operation: single-key lookups favour random; in-order reports favour sequential.

serial / sequential

walk from the start

random

hash, then jump

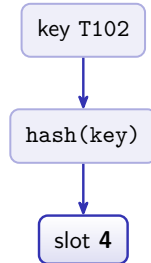
3 Hashing

A hash function turns a key into an address

A **hash function** 散列函数 takes a record key and produces the **slot** where the record is stored. A good one is fast,

deterministic 确定性, and spreads keys evenly. For N slots:

- modulo: $\text{key} \bmod N$
- folding: split, add the pieces, $\bmod N$
- string: sum character codes, $\bmod N$



Keep the **load factor** 装填因子 below about 70%.

Collisions – two keys, one slot

A **collision** 冲突 is when two keys hash to the same address.

linear probing 线性探测

use the next free slot, wrapping around. Simple, but keys *cluster*.

chaining 链接法

each slot points to a **linked list** 链表. No clustering; uses more memory.

rehashing 再散列

apply a second hash function. Spreads keys, more work.

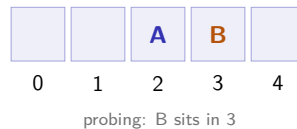
Worked example – resolve a collision

Keys A and B both hash to slot 2

Linear probing: B goes in the next free slot (3).

Chaining: slot 2 holds a list $A \rightarrow B$.

To search: hash the key, read that slot; if the keys do not match, follow the resolution strategy until a match or an empty slot.



Empty slot = “not found” when probing.

4 Floating-point

Data Representation

└ Floating-point

Mantissa × two-to-the-exponent

A **floating-point** 浮点 number is binary scientific notation:

$$\text{value} = \text{mantissa} \times 2^{\text{exponent}}$$

Both fields are **two's complement** 补码 integers.

- more mantissa bits $\Rightarrow$ more **precision**
- more exponent bits $\Rightarrow$ more **range**

The mantissa is a binary *fraction*: the first bit after the point is $\frac{1}{2}$, then $\frac{1}{4}$, $\frac{1}{8}$, ...

mantissa

0.1010000

exponent

00000010

$$0.625 \times 2^2 = 2.5$$

Data Representation

└ Floating-point

Worked example – binary to denary

Mantissa 10110000, exponent 00000011

Exponent 00000011 = +3.

Mantissa starts with 1, so it is **negative**. Sign bit = -1, fraction $\frac{1}{4} + \frac{1}{8} = 0.375$:

$$m = -1 + 0.375 = -0.625$$

$$\text{value} = -0.625 \times 2^3 = -5.0$$

A negative mantissa follows two's-complement rules. Convert the exponent first – it is the easy part.

Worked example – store +2.5

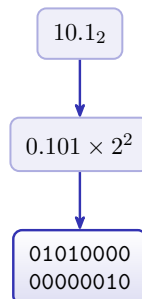
Write +2.5 in this 8+8-bit format

In binary $2.5 = 10.1$.

As a normalised fraction: $2.5 = 0.101 \times 2^2$.

mantissa **01010000** (sign 0, then .101)

exponent **00000010** ($= 2$)



Normalisation maximises precision

A number is **normalised** 规格化 when the first significant bit sits **immediately after the binary point** – no wasted leading zeros.

- positive: starts 0.1...
- negative (two's complement): starts 1.0...

Shift the mantissa left and decrease the exponent (or right and increase it) until that pattern holds. The value is unchanged.

Normalise 0.0101

Shift left until it reads 0.1...

$$0.0101 \rightarrow 0.101 \times 2^{-1}$$

mantissa 0.101, exponent -1

Each **left** shift subtracts one from the exponent.

What normalising does to the bits

before 0.0011010 exponent 4 (two wasted leading zeros)

↓ shift left 2, exponent -2

after 0.1101000 exponent 2 (normalised — same value)

Shifting until the first two bits differ uses every mantissa bit for **precision** instead of for leading zeros.

Why $0.1 + 0.2$ is not 0.3

Many denary reals **cannot be stored exactly** in binary – $0.1_{10} = 0.000110011\dots_2$ must be truncated.

rounding 舍入误差

errors build up over many operations

overflow 溢出

result too large for the exponent's range

underflow 下溢

result too small, rounds to zero

Never test reals with $=$. Use a tolerance, or use **fixed-point** 定点 / BCD for money.

5 Recap

Topic 13 – key takeaways

1 Types

enumerated, pointer, record, set, class

2 Files

serial, sequential, random; sequential vs direct access

3 Hashing

key $\rightarrow$ address; probing / chaining / rehash

4 Float

$m \times 2^e$; normalise; rounding / overflow

Check yourself

- 1 Which file type gives direct access by key?
- 2 Linear probing vs chaining – what is the trade-off?
- 3 How does a normalised *positive* mantissa start?
- 4 Why can 0.1_{10} not be stored exactly?

Answers 1. random 2. probing clusters; chaining uses extra memory 3. $0.1 \dots$ 4. it is a repeating binary fraction