Data Representation

A-Level Computer Science

User-defined data types

The built-in types (`INTEGER`, `REAL`, `STRING`, `CHAR`, `BOOLEAN`) cover the simplest cases. For richer problems you can define **user-defined types** 用户定义类型, making the code clearer and the compiler stricter.

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. And `DECLARE Taxi : Vehicle` is clearer (self-documenting) than `DECLARE Taxi : STRING`.

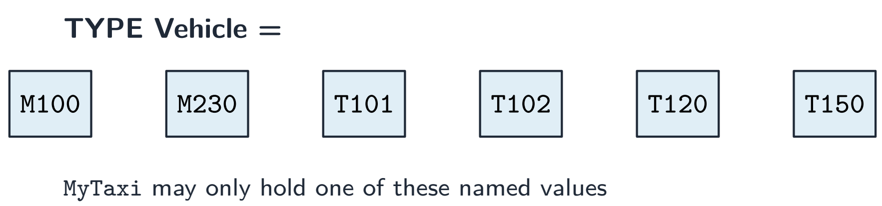
Non-composite types

Enumerated type

An **enumerated type** 枚举类型 has values that are a **fixed list of named constants**:

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

The names are values of the new type (stored internally as small integers); you cannot assign anything outside the list. Uses: days of the week, colours, status codes.



An enumerated type is a fixed list of named values

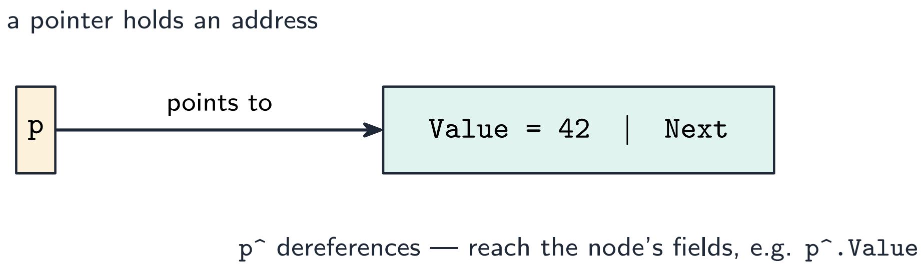
Pointer type

A **pointer** 指针 holds the **memory address of another variable** (or `NULL` for ”no target”). Pointers build dynamic structures (linked lists, trees) and pass references without copying.

```
TYPE PNode = ^TNode    // pointer to a TNode
DECLARE p : PNode
p ← NEW TNode
```

```
p^.Value ← 42            // dereference to reach the fields
```

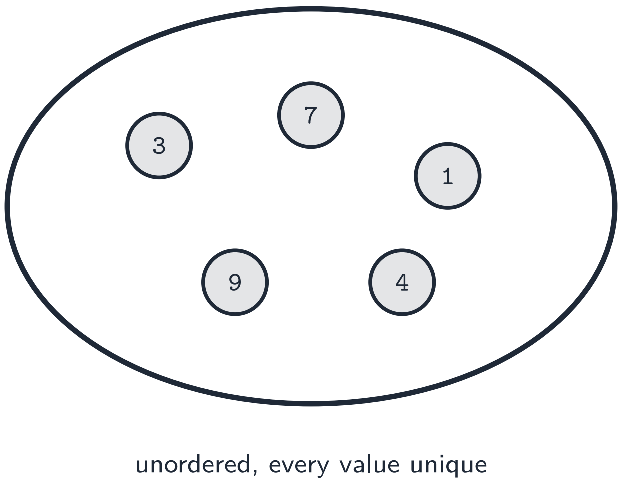
To **dereference** 解引用 (p^) means to reach the variable it points to.



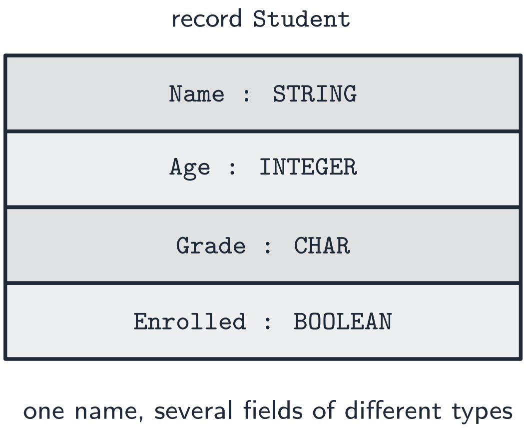
*A pointer holds an address; **p^** dereferences it to reach the node's fields*

Composite types

A **composite type** 复合类型 (one of the **composite data types**) groups several values under one name.



A set is an unordered collection of unique values



A record groups fields of different types under one name

- **record** 记录 (Topic 10) —fields of different types in a `TYPE ... ENDTYPE` block.
- **set** 集合—an unordered collection of unique values, with operations add, remove, membership test, union, intersection:

```
DECLARE Available : SET OF Colour
Available ← {Red, Blue}
IF Green IN Available THEN ...
```

- **class** 类 / **object** 对象—the OOP composite type, combining data fields (**attributes** 属性) with operations on them (**methods** 方法). An object is an instance of a class:

```
CLASS Taxi
    PRIVATE Capacity : INTEGER
    PUBLIC FUNCTION GetCapacity() RETURNS INTEGER
        RETURN Capacity
    ENDFUNCTION
ENDCLASS
```

Choosing a type

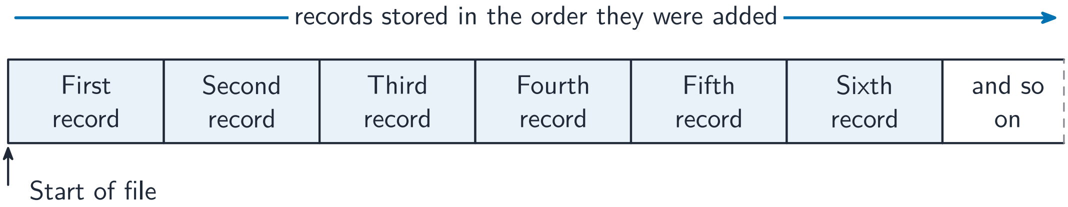
Use **enumerated** for a value from a fixed list, **pointer** for indirection, **record** for a group of fields, **set** for an unordered unique collection, and **class** when you need state **and** behaviour together.

File organisation and access

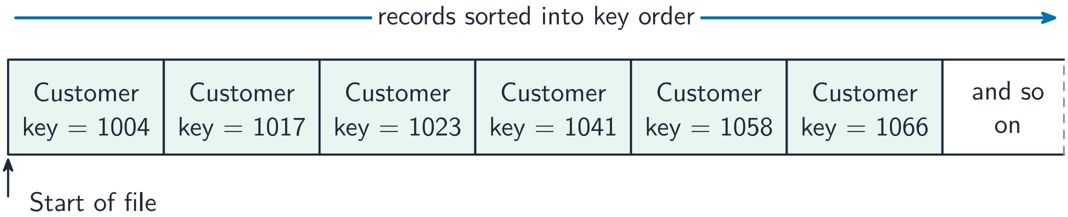
File organisation 文件组织 is how the data is laid out; **file access** is how the program reaches a record.

- **serial file** 串行文件—records in the **order added**, no sorting. Access is sequential only; appending is fast; searching is slow. Used for logs and audit trails.

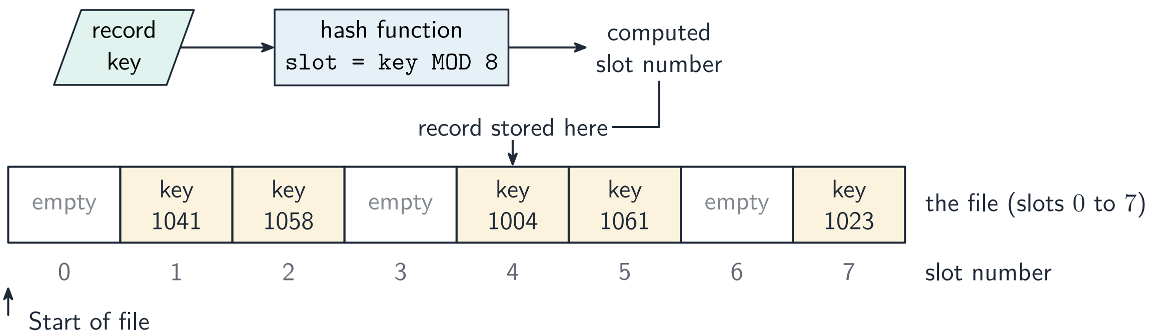
- **sequential file** 顺序文件—records **sorted by a key**. Searching is faster (you can stop early or binary-search); inserting is slow (records must shift). Used for master files updated in batch.
- **random (direct-access) file** 随机文件—records at positions computed from the key (often by a hash). Direct access by key is very fast; reading in key order is harder. Used for large lookup tables and customer accounts.



Serial file: records are kept in the order they were added



Sequential file: records are sorted by a key field



Random file: records sit at positions computed from the key

The two access methods are **sequential access** 顺序存取 (read from start to end) and **direct access** 直接存取 (jump straight to a known position). Match the structure to the dominant operation: single-key lookups favour random; in-order reports favour sequential.

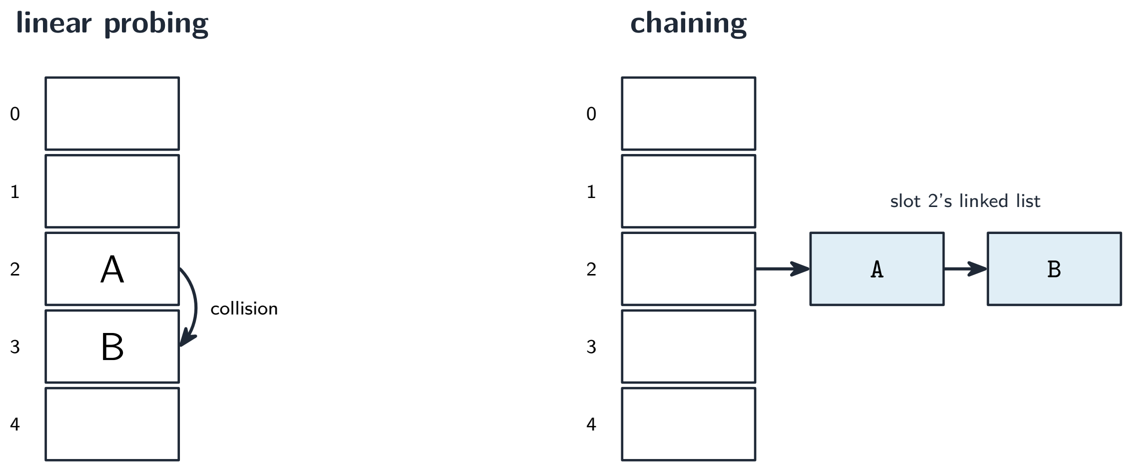
Hashing

A **hash function** 散列函数 (a **hashing algorithm**) takes a record key and produces an **address** where the record is stored. A good one is fast, **deterministic** 确定性, and spreads keys evenly.

Common **hashing algorithms** for N slots: modulo hash `address ← key MOD N`; folding (split the key, add the pieces, MOD N); a string hash (sum the character codes, MOD N).

A **collision** 冲突 is when two keys hash to the same address. Three ways to resolve it:

Strategy	How it works	Trade-off
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 of records	no clustering, but uses more memory
rehashing	apply a second hash function	spreads keys, but more work



Resolving a hash collision: linear probing uses the next free slot; chaining keeps a linked list per slot

To search: hash the key, read that slot; if the keys match you are done, else follow the resolution strategy until a match or an empty slot. To insert: hash the key, write to that slot or the next free one. Keep the **load factor** 装填因子 (records ÷ slots) below about 70% for near-O(1) lookups.

Floating-point numbers

To store **real numbers** of very different sizes, computers use a **floating-point** 浮点 format —a binary form of scientific notation, with two fields:

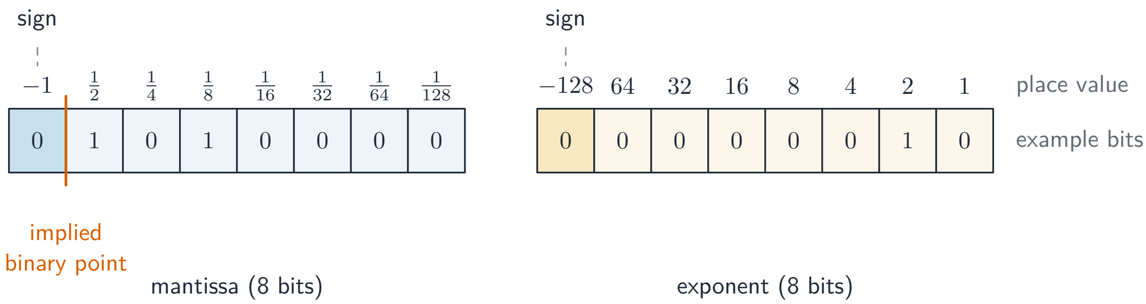
- a **mantissa** 尾数—the significant digits.
- an **exponent** 指数—the power of 2 to multiply by.

Both are stored as **two's complement** 补码 integers. The value is

$$\text{number} = \text{mantissa} \times 2^{\text{exponent}}.$$

Read the mantissa as a binary fraction —the first bit after the point is worth 1/2, the next

1/4, then 1/8, and so on. So `0.1010000` is $1/2 + 1/8 = 0.625$; with exponent `00000010` $(= 2)$ the value is $0.625 \times 2^2 = 2.5$.



The place values of an 8-bit mantissa and an 8-bit exponent

Converting

- **binary → denary**: read the mantissa (use two's-complement rules if negative) as a fraction, read the exponent as a signed integer, then multiply mantissa by 2^{exponent}.
- **denary → binary**: write the number as a binary fraction × a power of 2, then store the mantissa and exponent in the agreed formats.

Worked example. A number has mantissa `10110000` and exponent `00000011`. Find its denary value.

The exponent `00000011` is $+3$. The mantissa begins with a 1, so it is negative. Read as `1.0110000` in two's complement, the sign bit is worth -1 and the fraction bits add $\frac{1}{4} + \frac{1}{8} = 0.375$, so the mantissa is $-1 + 0.375 = -0.625$. Then

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

Worked example. Store $+2.5$ in this format.

In binary $2.5 = 10.1$. Written as a normalised fraction, $2.5 = 0.101 \times 2^2$. So the mantissa is `01010000` (sign bit 0, then `.101`) and the exponent is `00000010` $(= 2)$.

Normalisation

A number is **normalised 规格化** when the first significant bit is **immediately after the binary point** (no wasted leading zeros). This **maximises precision**, because every mantissa bit carries information. To normalise, shift the mantissa left and decrease the exponent (or shift right and increase it) until the first significant bit is in place; the value is unchanged. For negative (two's-complement) mantissas, the sign bit (1) is followed immediately by a 0.

before	`0.0011010`	exponent 4 (two wasted leading zeros)
	↓ shift left 2, exponent − 2	
after	`0.1101000`	exponent 2 (normalised — same value)

Normalising: shift the mantissa left to remove leading zeros, lowering the exponent by the same amount

Approximation and rounding errors

Many denary reals **cannot be stored exactly** in binary —e.g. 0.1_{10} is the repeating binary fraction $0.000110011\ldots_2$, which must be truncated. Consequences:

- **rounding errors** 舍入误差 build up over many operations (`0.1 + 0.2` is not exactly `0.3`).
- **comparisons fail** —test `ABS(x - 0.3) < 1e-9` instead of `x = 0.3`.
- subtracting two nearly-equal values loses precision.
- **overflow** 溢出 (a result too large for the exponent's range) and **underflow** 下溢 (a result too small, rounding to zero) occur when the exponent runs out of range.

For exact needs (currency), use **fixed-point** 定点 or **BCD** 二进码十进数 instead of floating-point.

Exam tips

- For **floating-point** binary → denary, read the mantissa as a fraction and the exponent as a signed integer, then multiply; a **negative mantissa** follows two's-complement rules.
- **Normalise** by shifting until the first digit after the point differs from the sign bit — this maximises precision.
- Explain **rounding, overflow and underflow** errors and why 0.1 cannot be stored exactly.
- Compare file organisation (serial, sequential, direct) and how **hashing** finds a record quickly.