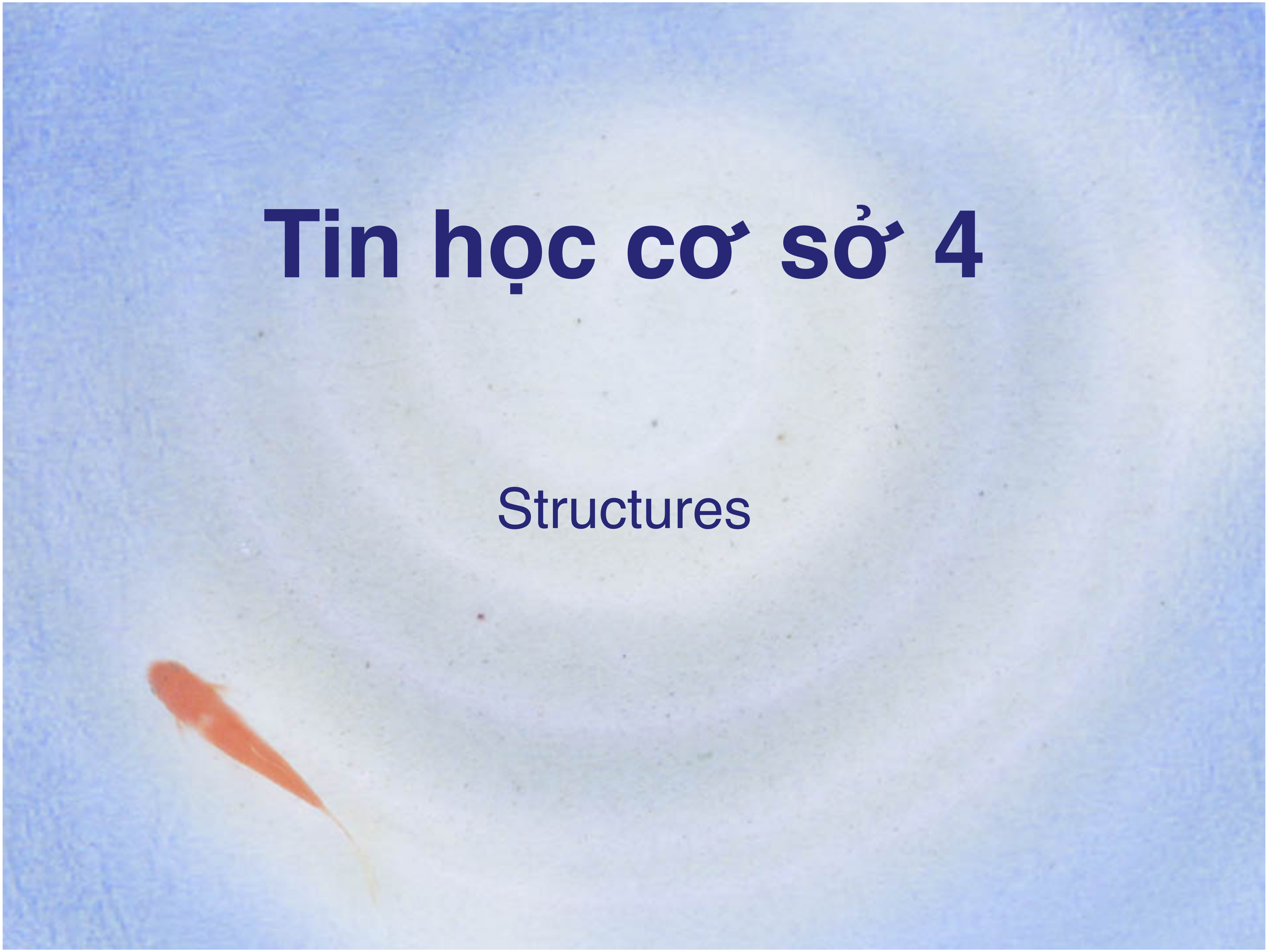


Tin học cơ sở 4

Structures

Outline

- Structured Data types
- Type definitions
- Self-referential types: linked list

Structure

- Many structured collections of data are:
  - Heterogeneous (components are different types)
  - Fixed-size (well-defined set of components)
- For example:

Student Record	
Name	Nguyen Van A
Student Id	1234567
Course	CSE
Date of Birth	01/01/1990
Gender	Male

Structured data types

- A structure is a collection of variables, perhaps of different types, grouped together under a single name.
- Structures:
  - Help to organize complicated data into manageable entities.
  - Expose the connection between data within an entity
  - Are defined using the *struct* keyword.

Structs

- In C, struct models such data.
- To define a record type, we must give:
  - Name of struct
  - Name of each field
  - Type of each field (could be another record/struct)
- Example:

```
struct employee {
    char    name[30];
    int     id;
    char    position[20];
    float   salary;
};
struct employee e1, e2;
```

Structs

- A record variable can be used as follows:
  - Assigned to another variable of the same type
  - Passed as a parameter to a function
  - To select a particular field (component) using the “dot” operator. Fields of a record behave *exactly* like variables of the appropriate type.

```
struct student john, betty;
struct student comp1721_students[200];
strcpy(john.name, "John Smith");
john.student_number = 2116344;
betty = john;
strcpy(betty.name, "Betty Smith");
betty.student_number = 2116345;
comp1721_students[0] = john;
comp1721_students[1] = betty;
```

Structs

- Operations which are NOT defined for whole records:
  - compare for equality/inequality
    (i.e. john == betty is not a legal expression)
  - compare based on ordering (<, >, ...)
    (i.e. john < betty is not a legal expression)
  - arithmetic operations
    (i.e. john + betty is not a legal expression)
  - reading from or writing to text files
    (i.e. no cout << john; or equivalent)
- If you need such operations, write your own procedures and functions.
- Unlike arrays, it is possible to copy all components of a structure in a single assignment.

Structs and Pointers

- As for other types:
  - (struct x *) is a pointer to struct x
  - & gives the address of a structure
- There are precedence problems because . binds more tightly than *
- If p is a pointer to a struct and a is a field of the struct:
  - *p.a means *(p.a) which is illegal.
  - You must use (*p).a
  - For convenience , the operator -> combines indirection and component selection.
  - p->a is equivalent to (*p).a

Structs and Pointers

- For example

```
struct point {
    int x, y;
};
struct point a;
struct point *ap;
ap = &a;
*ap.x = 0;   /* wrong  - equivalent to *(ap.x) = 0;*/
(*ap).y = 0; /* right */
ap->x = 0;   /* right */
```

Type Definition

- We can use the keyword typedef to make our own definitions:

typedef float Floating;

- This means variables can be declared as Floating but they will actually be of type float.
- If we later decide we need more precision, we can change to:

typedef double Floating;

without changing the codes.

Combining struct and typedef

```
typedef struct employee Employee;
struct employee {
    char    name[30];
    int     id;
    char    position[20];
    float   salary;
 };
Employee john;
```

- Note: we use the convention that the name of the defined type is the same as the *struct* modifier, but with the first letter capitalized.

Passing Structures as parameters

- A structure can be passed as a parameter to a function:

```
void print_employee (Employee e){
   printf(“Name: %s\n”, e.name);
}
```

- Because parameters in C are “call-by-value”, a copy of the entire structure will be made, and only this copy will be passed to the function.
- If a function needs to modify components within the structure, or if we want to avoid the inefficiency of copying the entire structure, we can instead pass a pointer to the structure as a parameter (*Employee *e*) or use “pass-by-reference” (*Employee &e*).

Self-referential types

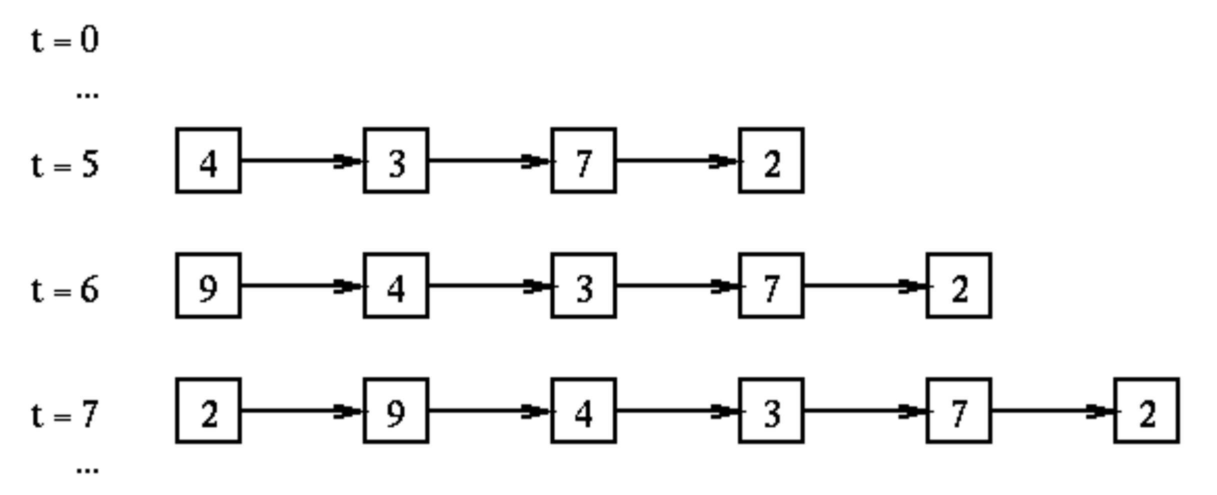
- A very powerful programming technique is to create struct with fields which contain a reference to an object in the same struct. For example:

```
struct list_node {
    int  data;
    struct list_node *next;
};
```

- This approach can be used to create some very useful data structures.

Linked Lists

- Consider the following data structure:
  - A list that grow over time
  - Items are added one by one
  - Each item is a number
- It might grow like this:



Linked Lists

- How do you implement such a list in C?
- You can use an array but you must check the array doesn't overflow and allocate memory for a new larger array if the array fills.
- Also can be implemented using a self-referential type.
- Each element in the linked list need:
  - some information (the data)
  - a connection to the next item
- The individual elements in data structures like this are often called **nodes.**

Linked Lists

- A node could be represented by this

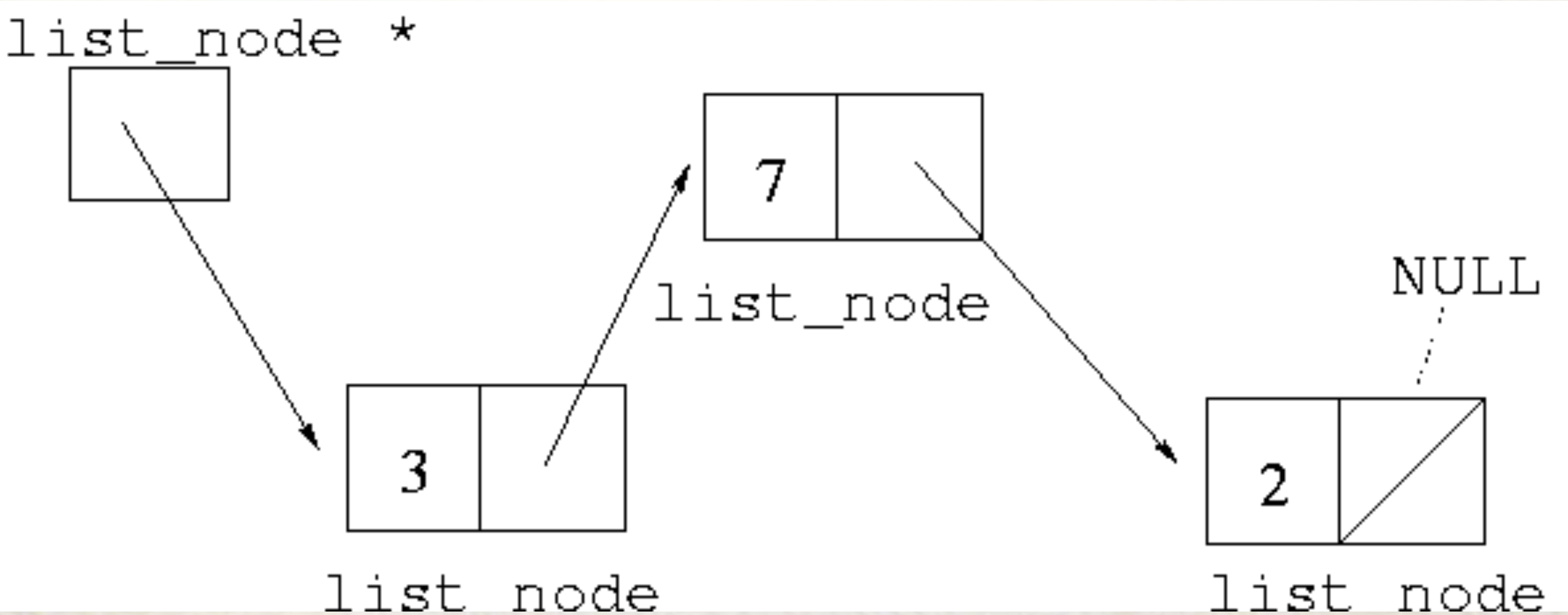
```
struct list_node {
    int  data;
    struct list_node *next;
}
```

- How can we represent the whole list?
- We need a pointer to the first node:

```
    struct list_node *list_start;
```

- How can we represent the ``end of list"?
  The next field of the struct will be NULL.

Linked Lists



Linked Lists

- Some methods:
  - insert a node
  - append a node
  - delete a node
  - print a list

File Handling

- stdin: keyboard, stdout: screen
- FILE in <stdio.h>

FILE *fp;

FILE *fopen(char *name, char* mode);

Mode: r: read, w:write, a:append, b: binary

int fscanf(FILE *fp, char *format, ….);

int fprintf(FILE *fp, char *format, ….);

- All files must be closed: fclose(fp);

References

- The C Programming Language. Chapter 6.