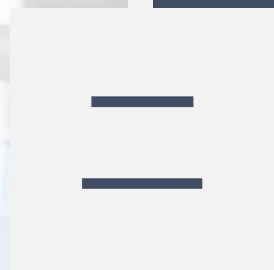


算法与数据结构

华东理工大学 叶琪





算法与数据结构

第二章 实现基础



CONTENTS

目录

1 数据存储基础

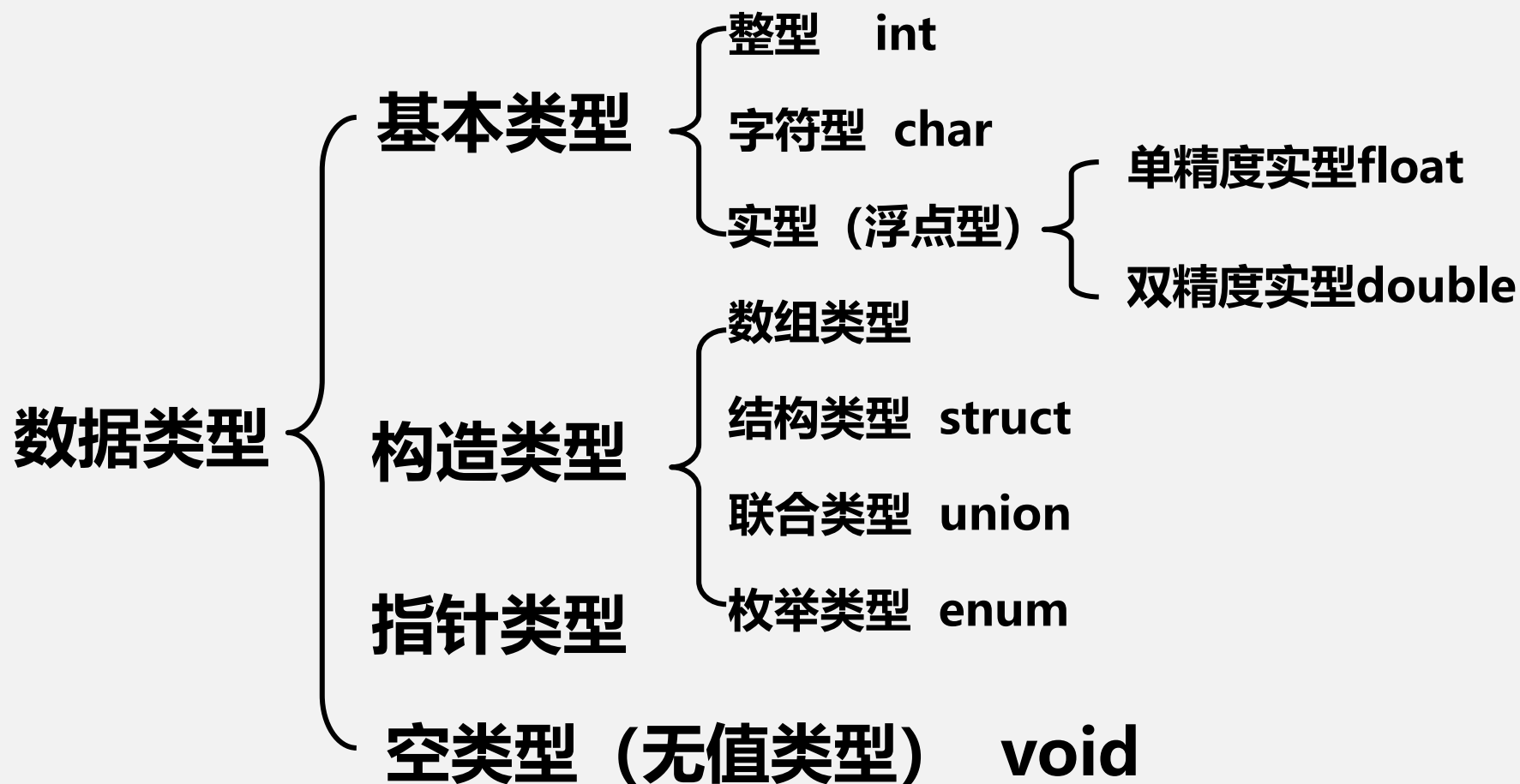
数组、指针、结构、链表、类型定义

2 流程控制基础

分支、循环、函数和递归

2.1 数据存储基础

❖ 数据存储的基本单位是**变量**，变量的**类型**决定了**存储和操作**。



2.1 数据存储基础

1. 数组

- 数组是最基本的构造类型，它是一组**相同类型**数据的**有序集合**。
 - 数组中的元素具有**固定格式和数量**；
 - 一旦被定义，每一维的**大小及上下界都不能改变**；
 - 用**数组名和下标**可以唯一地确定数组元素；
 - 数组中的元素在内存中**连续**存放。
- **优点**：随机存取

2.1 数据存储基础

1. 数组 ——(1)一维数组

类型说明符 数组名 [数组长度] ;

【例】用数组求解Fibonacci数列。公式： $a_1=1$, $a_2=1$..., $a_n=a_{n-1}+a_{n-2}$ 。

例如:1, 1, 2, 3, 5, 8, 13.....

```
#include <stdio.h>
void main()
{   int i;
    int f [20] = {1, 1}; /*数组初始化*/
    for (i=2; i<20; i++)
        f [i] = f [i-2] + f [i-1] ; /*Fibonacci计算过程*/
    for (i=0; i<20; i++)
    {   if (i%5==0) printf (" \n");
        printf ("%12d", f [i] );
    }
}
```

数组的存储

- 例如：数组 `int list[4];`

变量	内存地址
List[0]	基址 α
List[1]	$\alpha + \text{sizeof}(\text{int})$
List[2]	$\alpha + 2 * \text{sizeof}(\text{int})$
List[3]	$\alpha + 3 * \text{sizeof}(\text{int})$

数组名list中存储了基址 α

```
#define MAX_SIZE 100
void main()
{
    int list[MAX_SIZE], answer;
    for (i = 0; i < MAX_SIZE; i++){list[i] = i;}
    printf("list[0]=    %d\n",list[0]);
    printf("list[0]的地址是    %d\n",&list[0]);
    printf("list[0]的大小是    %d\n",sizeof(list[0]));
    printf("list=    %d\n", list);
    printf("list的地址是    %d\n",&list);
    printf("list的大小是    %d\n", sizeof(list));
    printf("数组长度为:");
}
```

Microsoft Visual Studio 调试控制台

```
list[0]=    0
list[0]的地址是    17824788
list[0]的大小是    4
list=    17824788
list的地址是    17824788
list的大小是    400
数组长度为: 100
```

【例】 数组作为函数的参数传递。

```
#define MAX_SIZE 100
int i;
float sum(float list[], int n)
{
    int i;
    float tempsum = 0;
    for (i = 0; i < n; i++)
        tempsum += list[i];
    list[0] = 99;
    return tempsum;
}

void main()
{
    float input[MAX_SIZE], answer;
    for (i = 0; i < MAX_SIZE; i++){input[i] = i;}
    printf("input[0]为: %d\n", input[0]);
    answer = sum(input, MAX_SIZE);
    printf("1-99的和为: %f\n", answer);
    printf("input[0]为: %d\n", input[0]);
    printf("数组长度为: %d\n", sizeof(input) / sizeof(int));
}
```

```
#define MAX_SIZE 100
int i;
float sum(float list[])
{
    int i, n;
    n = sizeof(list) / sizeof(int);
    float tempsum = 0;
    for (i = 0; i < n; i++)
        tempsum += list[i];
    list[0] = 99;
    return tempsum;
}
```

Microsoft Visual Studio 调试控制台

函数中，list只是一个指针，sizeof(list)的结果是指针变量list占内存的大小

2.1 数据存储基础

1. 数组 ——(2)二维数组

类型说明符 数组名 [行长度][列长度];

- 二维数组的定义、初始化是使用与一维数组相似。可以把二维数组看作是一种特殊的一维数组：它的元素是一个一维数组。

【例】 $a[3][4]$ 看作是一个一维数组，它有3个元素： $a[0]$ 、 $a[1]$ 、 $a[2]$ ，每个元素又是一个包含4个元素的一维数组。

a	$a[0]$	-----	a_{00}	a_{01}	a_{02}	a_{03}
	$a[1]$	-----	a_{10}	a_{11}	a_{12}	a_{13}
	$a[2]$	-----	a_{20}	a_{21}	a_{22}	a_{23}

【例】 将一个二维数组行和列元素互换，存到另一个二维数组中。

```
void main()
{
    int a[2][3]={1, 2, 3}, {4, 5, 6}; /*二维数组赋初始值*/
    int b[3][2], i, j;
    printf("array a: \n");
    for (i=0; i<=1; i++)
    {
        for (j=0; j<=2; j++)
        {
            printf("%5d", a[i][j]);
            b[j][i]=a[i][j]; /*数组互换*/
        }
        printf("\n");
    }
    printf("array b: \n");
    for (i=0; i<=2; i++)
    {
        for (j=0; j<=1; j++)    printf("%5d", b[i][j]);
        printf("\n");
    }
}
```

Microsoft Visual Studio 调试控制台

```
array a :
    1      2      3
    4      5      6
array b :
    1      4
    2      5
    3      6
```

2.1 数据存储基础

2.指针

- 使用指针可以对复杂数据进行处理，能对计算机的内存进行分配控制，在函数调用中使用指针还可以返回多个值。定义为：

类型名 *指针变量名；

【例】 `int *p; /* 指针变量名是p，而不是* p */`

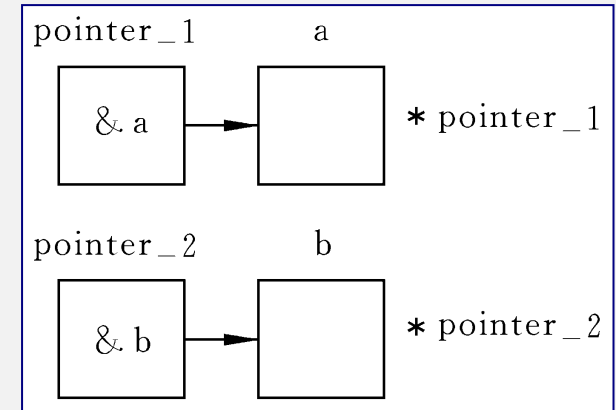
注意：

- (1) 指针变量只有在赋值后才能使用；
- (2) 基本运算：取地址运算符&，间接访问运算符*
- (3) 指针算术运算只包括：加，减

指针变量的引用

- 指针变量中只能**存放地址（指针）**，不要将一个整数（或任何其他非地址类型的数据）赋给一个指针变量。

```
int *pointer_1 , *pointer_2 ;  
a=100; b=10;  
pointer_1 = &a ;    /*把变量 a 的地址赋给 pointer_1 */  
pointer_2 = &b ;    /*把变量 b 的地址赋给 pointer_2 */  
printf ( "%d , %d \n " , a , b ) ;  
printf ( "%d , %d \n " , *pointer_1 , *pointer_2 ) ;
```



Microsoft Visual Studio 调试控制台

```
a= 100= 10  
*pointer_1= 100 , *pointer_2= 10
```

2.1 数据存储基础

2.指针 ——(1) 指针与数组

- 数组名是数组中第1个元素（下标为0）的地址，可以看作是常量指针，不能改变指针常量（数组名）的值。
- 引用一个数组元素，可以用：
 - (1) 下标法，如 $a[i]$ 形式；
 - (2) 指针法，如 $*(a + i)$ 或 $*(p + i)$ 。

其中 a 是数组名， p 是指向数组元素的指针变量，其初值 $p = a$ 。

【例】 通过指针变量输出 a 数组的10个元素。

```
void main ()
{int *p ,i,a[10];
  p=a;
  for(i=0;i<10;i++) scanf ("%d",p++);
  printf("\n"); p=a;
  for(i=0;i<10;i++,p++) printf("%d",*p);
}
```



2.1 数据存储基础

头文件 `stdlib.h` 或 `malloc.h`

2. 指针 —— (2) 用指针实现内存动态分配

- ① 分配函数 `void *malloc(unsigned size)` 。
 - 在内存中分配长度为size大小的连续空间，申请成功返回指向所分配内存空间的起始地址，不成功返回NULL；
 - malloc的返回是void *，使用时，需要将malloc的返回值转换为特定指针变量。

【例】 `int *p;`

- ② 释放函数 `void free(void *ptr)` 。
`p = (int *)malloc(sizeof(int));`

- 释放由动态存储分配函数申请到的内存空间，ptr为指向要释放空间的首地址

【例】 `free(p);`

2.1 数据存储基础

3.结构

【定义】 结构类型把一些**不同类型的数据分量聚合**成一个整体。同时，结构又是一个变量的集合，可以**单独使用其变量成员**。

结构类型定义的一般形式为：

```
struct 结构名{  
    类型名 结构成员名1;  
    类型名 结构成员名2;  
    .....  
    类型名 结构成员名n;  
};
```

【例】

```
struct student  
{  
    int num;  
    char name[20];  
    char sex;  
    int age;  
    float score;  
    char addr[30];  
}
```

①结构变量的声明和使用

(1)先声明结构体类型再定义变量

【例】：struct student student1, student2;

结构体类型名

结构体变量名

(2)在声明类型的同时定义变量,这种形式的定义的一般形式为:

struct [结构体名]

{

成员表列

} 变量名表列;

➤ 调用格式为：结构变量名.结构成员名。

➤ 【例】：student1.num表示student1变量中的num成员。

➤ 结构变量不仅可以作为函数参数，也可以作为函数的返回值。

②结构数组：结构与数组的结合

对结构数组元素成员的引用是通过使用数组下标与结构成员操作符来完成的，其一般格式为：

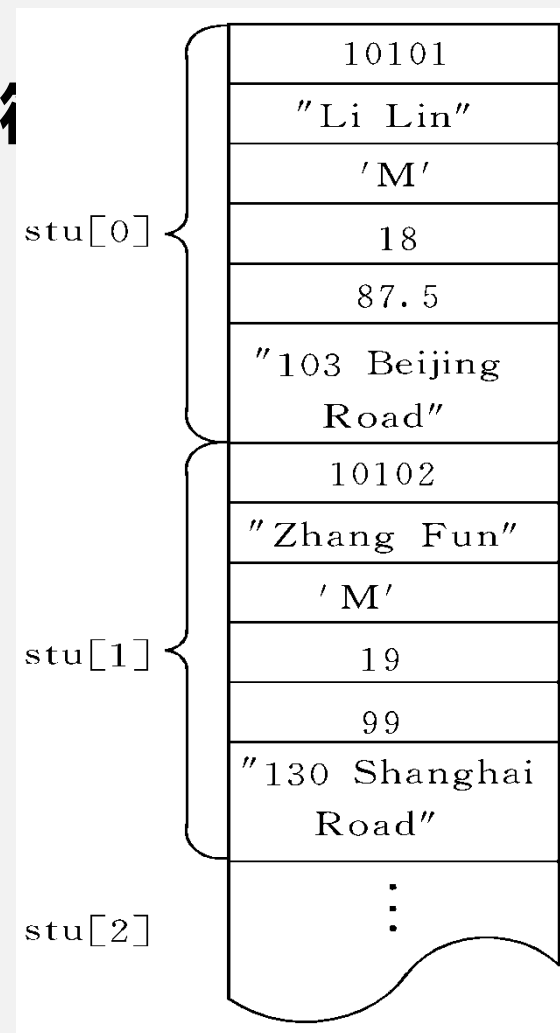
结构数组名[下标].结构成员名

【例】 **struct student**

```
{ int num;  
  char name[20];  
  char sex;  
  int age;  
  float score;  
  char addr[30];  
};
```

定义结构数组： **struct student stu[2];**

stu[2] = {{10101, "LiLin", 'M', 18, 87.5, "103 BeijingRoad"}, {10102, "Zhang Fun", 'M', 19, 99, "130 Shanghai Road"}};



③结构指针：指向结构的指针

(1) 用*方式访问，形式： **(*结构指针变量名).结构成员名**

(2) 用指向运算符“->”访问指针指向的结构成员，形式：

结构指针变量名->结构成员名

```
void main()
{
    struct student{
        long num;
        char name[20];
        float score;
    };
    struct student stu_1;
    struct student* p;
    p=&stu_1;
    stu_1.num=12170101;
    strcpy_s(stu_1.name," LiLin" );
    stu_1.score=89.5;
    printf("No.:%ld name:%s score:%f\n",stu_1.num,stu_1.name, stu_1.score);
    printf("No.:%ld name:%s score:%f\n",(*p).num,(*p).name, (*p).score);
    printf("%5d %-20s %4f\n", p->num, p->name,p->score);}
```

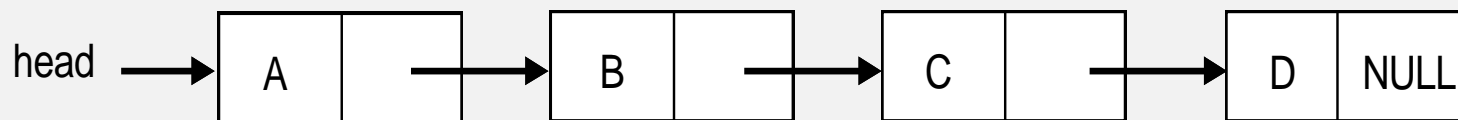
```
No.:12170101 name : LiLin score : 89.500000
No.:12170101 name : LiLin score : 89.500000
No.:12170101 name : LiLin score : 89.500000
```

2.1 数据存储基础

4.链表

- 链表是一种重要的基础数据结构，也是实现复杂数据结构的重要手段。每一个结点里保存着下一个结点的地址（指针）。
- 链表又分单向链表，双向链表以及循环链表等。

①单向链表的结构



【定义】使用结构的嵌套来定义单向链表结点的数据类型。

【使用】

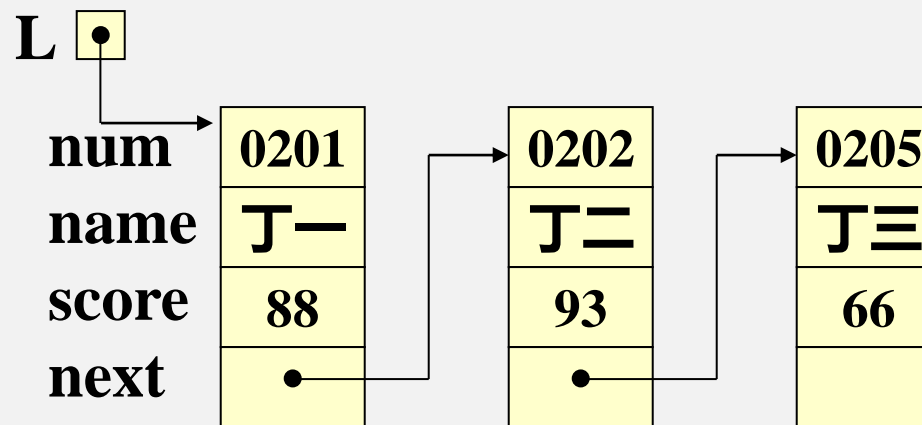
```
struct Node{  
    ElementType Data;  
    struct Node *Next;  
};
```

自引用结构

```
struct Node *p;  
p = (struct Node *) malloc(sizeof(struct Node));  
(*p).data='A';
```

例:

```
struct student
{ char num[8];          //数据域
  char name[8];         //数据域
  int score;            //数据域
```



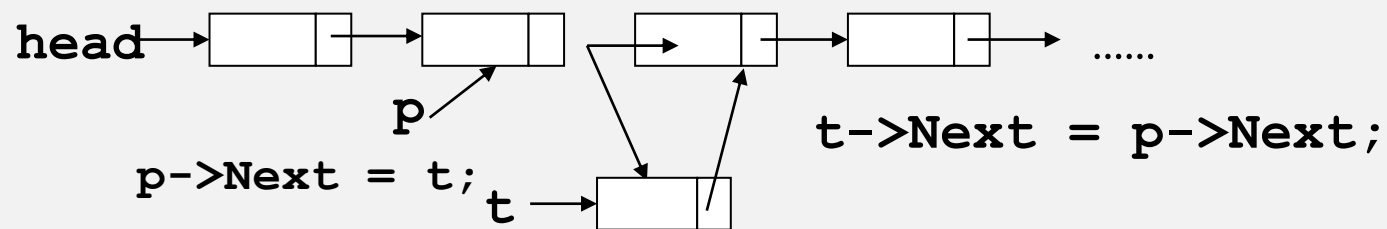
struct student *next; //next 既是 struct student 类型中的一个成员，又指向 struct student 类型的数据。

```
}Stu_1, Stu_2, Stu_3, *L;
```

```
Struct student *head;
```

②单向链表的常见操作

(1) 插入结点 (p之后插入新结点t)



(2) 删除结点

$t = p \rightarrow \text{Next};$

$p \rightarrow \text{Next} = t \rightarrow \text{next};$

(3) 单向链表的遍历

```
p = head;  
while (p!=NULL) {  
    .....  
    处理p所指的结点信息;  
    .....  
    p = p->Next;  
}
```

(4) 链表的建立

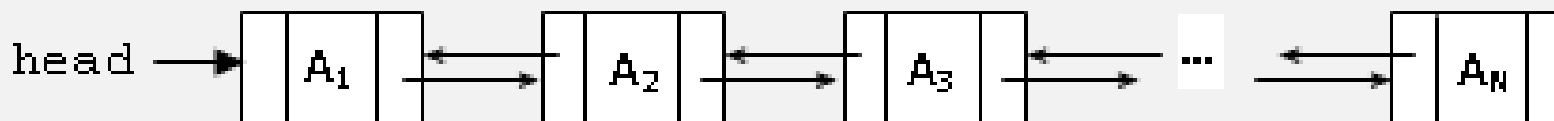
两种常见的插入结点方式:

- (1) 在链表的头上不断插入新结点;
- (2) 在链表的尾部不断插入新结点。

如果是后者，一般需要有一个**临时的结点指针**一直指向**当前链表的最后一个结点**，以方便新结点的插入。

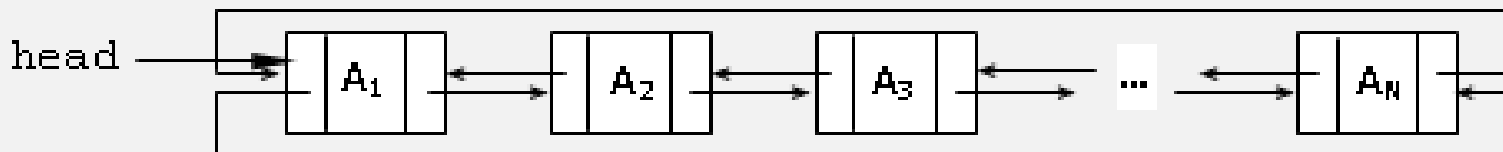
③双向链表

```
struct Node{  
    ElementType Data;  
    struct Node *Next;  
    struct Node * Previous;  
};
```

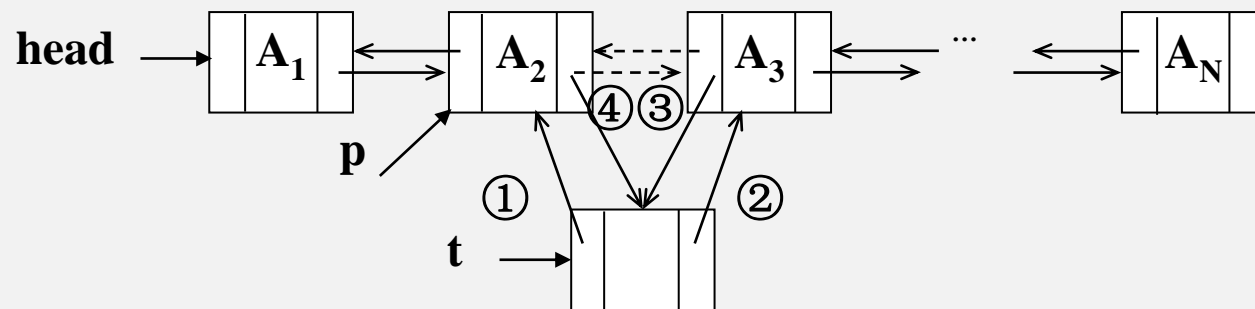


● 双向循环链表:

- 将双向链表最后一个单元的Next指针指向链表的第一个单元;
- 第一个单元的Previous指针指向链表的最后一个单元



❖ 双向链表的插入、删除和遍历基本思路与单向链表相同，但需要同时考虑前后两个指针。



```
struct DNode {  
    ElementType Data;  
    struct DNode *Next;  
    struct DNode *Previous;  
} *p,*t;
```

指针操作顺序:

- ① t->Previous = p;
- ② t->Next = p->Next;
- ③ p->Next->Previous = t;
- ④ p->Next = t;

【例】 链表转置 给定一个单链表L，设计函数Reverse将链表L就地逆转，即不需要申请新的结点，将链表的第一个元素转为最后一个元素，第二个元素转为倒数第二个元素，……

【分析】 基本思路：

- 定义两个链表p，q，p是一个待逆转的序列，而q是一个已经逆转好的序列。
- 利用循环，从链表头开始逐个处理。
- 每轮循环把p中的第一个元素插入到q的头上，直至p为空。

```
struct Node *Reverse(struct Node *L)
{
    struct Node *p, *q, *t;
    p = L, q = NULL;
    while ( p != NULL ) {
        t = p->Next;
        p->Next = q;  q = p;
        p = t;
    }
    return q;
}
```

2.1 数据存储基础

5. 类型定义typedef

利用typedef语句来建立已经定义好的数据类型的别名，定义为：

typedef 原有类型名 新类型名

【例】 `typedef char *STRING;` //声明STRING为字符指针类型

说明：

- 用typedef可以声明各种类型名，但不能用
- 用typedef只是对已经存在的类型增加一个
- 使用typedef有利于程序的通用与移植。

```
Typedef struct node{  
    long data;  
    struct node *next;  
} LinkedList;  
LinkedList *head;
```

2.2 流程控制基础

❖ 三种基本的控制结构是**顺序、分支和循环**。

- 顺序结构是一种自然的控制结构，通过安排**语句或模块的顺序**就能实现。
- 分支控制提供了**if-else**和**switch**两类语句。
- 循环控制提供了**for、while**和**do-while**三类语句。

- 函数定义
- 函数调用
- 函数递归



语句级控制

单位级控制



【例】 求100到200之间的所有素数。

【分析】 可以设定两重循环：大循环（外层循环）控制整数*i*在100到200之间变化（用for语句），而小循环（内层循环）则用来判别*i*是否是素数（用while语句）。

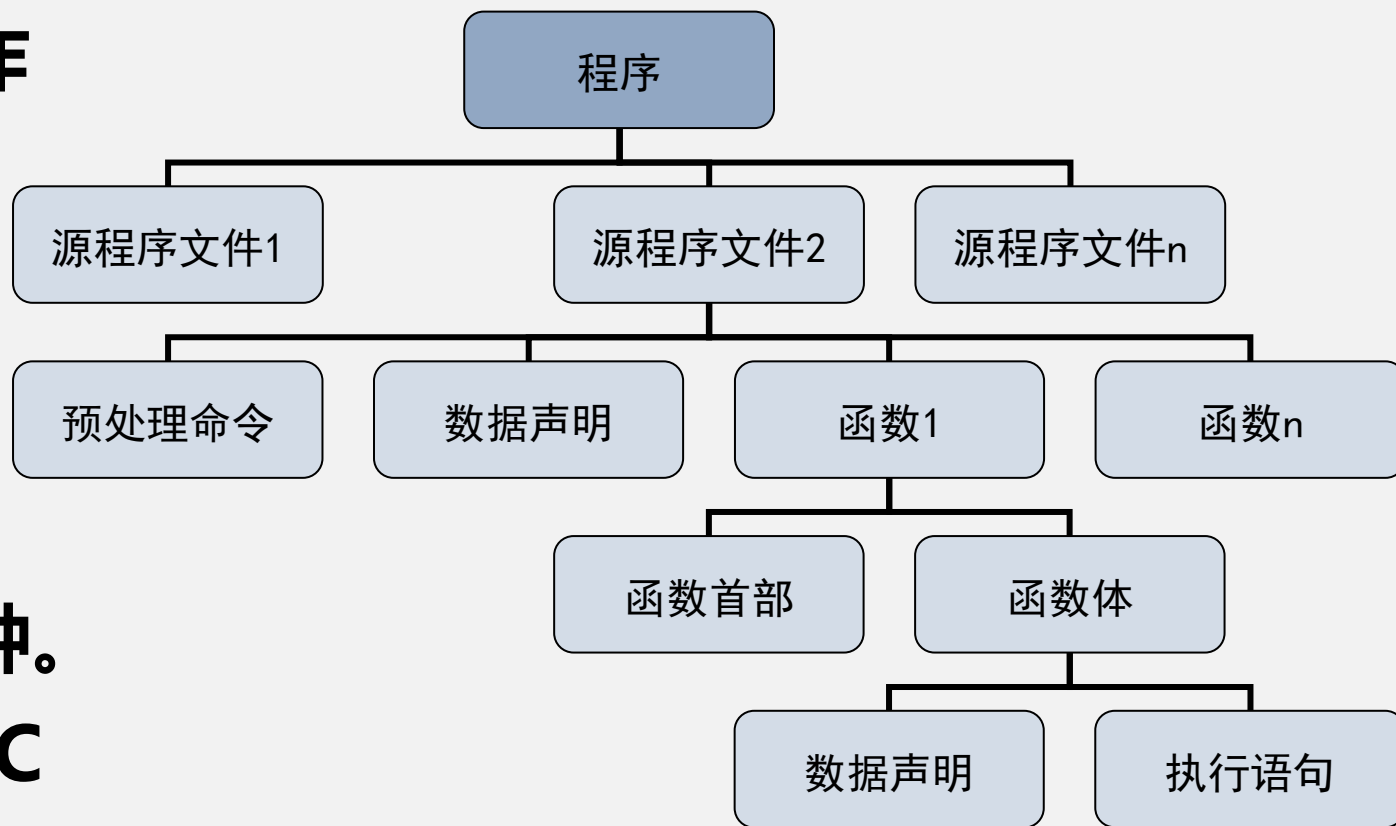
```
int main()
{
    int i, j;
    for( i = 100; i <= 200; i++ ){          /* 外层循环*/
        j = 2;
        while (j < i && i%j != 0 ) j++;
        /* 内层循环,判别i是否是素数*/
        if (j == i) printf("%d ", i);
        /* j == i 说明在上面的while循环中i都不能被j整除, 因此i是素数 */
    }
    return 0;
}
```

函数与递归

【定义】 函数是一个完成特定工作的独立程序模块。

➤ 函数类型 函数名 (形参表)
{函数实现过程}

函数包括**库函数**和**自定义函数**两种。
例如，scanf、printf等库函数由C语言系统提供定义，编程时只要直接调用即可。



【例】 复数无法使用 “+” 运算；可以写一个函数来实现复数加法的这个功能。

(1) 定义复数类型 `ImgType`，以约定何为复数：

```
struct Image {  
    double r;  
    double i;  
};  
typedef struct Image ImgType;
```

(2) 定义复数的加法函数：

```
ImgType ImgAdd(ImgType a, ImgType b)  
{  
    ImgType c;  
    c.r = a.r + b.r;  
    c.i = a.i + b.i; /*实部和虚部分别相加 */  
    return c;  
}
```

函数调用 函数名 (实参表)

```
ImgType a,b,c;  
a.r = 0.5;  
a.i = 2.5;  
b.r = 1;  
b.i = -5;  
c = ImgAdd(a,b);  
printf("complexAdd: c.r %f, c.i %f \r\n",c.r,c.i);
```

有了这个函数，以后可以在任何需要计算复数加法的地方调用它！

❖ 在设计函数时，注意掌握以下原则：

- (1) 函数**功能**的设计原则：结合模块的独立性原则，函数的**功能要单一**，不要设计多用途的函数，否则会降低模块的聚合度；
- (2) 函数**规模**的设计原则：函数的**规模要小**，尽量控制在**50行代码以内**，这样可以使得函数更易于维护；
- (3) 函数**接口**的设计原则：结合模块的独立性原则，函数的接口包括函数的参数（入口）和返回值（出口），**不要设计过于复杂的接口**，合理选择、设置并控制参数的数量，尽量不要使用全局变量，否则会
增加模块的耦合度。

递归函数


【定义】 函数支持自己调用自己的形式称为函数的递归调用，带有递归调用的函数也称为递归函数。

❖ 两个关键点：

- (1) 递归出口：即递归的结束条件，到何时不再递归调用下去；
- (2) 递归式：当前函数结果与准备调用的函数结果之间的关系，如求阶乘函数的递归式子

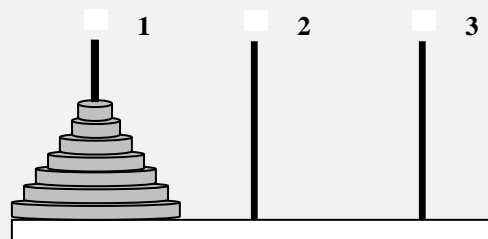
$$\text{Factorial}(n) = n * \text{Factorial}(n-1)$$

```
long int Factorial( int n )  
{  
    if( n == 0 ) return 1;  
    else      return n * Factorial(n-1);  
}
```

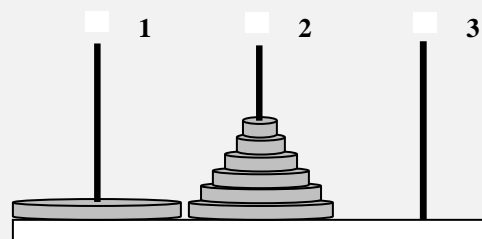


递归调用

【例】 汉诺塔 (Tower of Hanoi) 问题



(a) 初始状态



(b) 中间状态

【分析】 可以用递归方法来求解汉诺塔问题，也就是将 n 个盘片的移动问题转换为2个 $n-1$ 个盘片的移动问题。其中，当 $n=1$ 时，是递归出口。

```
void Move(int n, int start, int goal, int temp)
{
    递归调用
    if( n == 0 ) return;
    Move(n-1, start, temp, goal);
    printf("Move disk %d from %d to %d.\n", n, start, goal);
    Move(n-1, temp, goal, start);
}
```

本章小结

数据 存储

数组：一维数组、二维数组

指针：定义、引用、指针和数组、动态分配

结构：结构的定义和声明、结构数组、结构指针

链表：单向链表（插入、删除、建立、遍历）、双向链表

类型定义：typedef

流程 控制

分支、循环、函数（递归函数）



THANKS

华东理工大学 叶琪