第四章：指针高级应用

指针数组和数组指针

- **找核心+找结合（最近的），与[]结合是数组、与*结合是指针、与（）结合是函数**
- 优先级：`[].`→这三个优先级比较高
  - int * p[5] // 指针数组。p是个数组，有5个元素，数组中的元素都是指针，指向int类型元素
  - int (*p)[5] // 数组指针。p是指针，指针指向数组，数组有五个元素，元素都是int型
  - int *(p[5]) // 指针数组。相当于int * p[5]

函数指针

- 函数指针
  - **函数为 void func(void)，函数指针为 void (*p)(void)，类型为void(*)(void)，指向函数地址（即函数第一句代码的首地址，用函数名表示）**
  - 函数指针、数组指针、普通指针没有本质区别，都是指针变量，占4个字节（32位系统），区别在于指针指向什么
  - 函数指针解引用可以调用该函数

    示例

```
void func1(void)
{...}
int main(void)
{
  void (*pFunc)(void);//定义函数指针
  pFunc=func1;//函数名代表函数指针
  //注意：&func1和func1做右值一模一样，没有区别（编译器规定）
  pFunc();//相当于func1()
}
```

- 函数指针实战
  - 函数指针执行函数

```
#include<stdio.h>
```

```c
#include<stdio.h>


int add(int a, int b);
int sub(int a, int b);
int multiply(int a, int b);
int divide(int a, int b);

typedef int (*pFunc)(int,int);
//为函数指针类型取别名为pFunc
//后面用其声明变量，使用时要为其赋值

int main()
{
    int a,b=2;
    char c=0;
    pFunc p=NULL;//局部变量定义一定要先初始化，指针声明先赋值为NULL

    printf("输入两个整数：\n");
    scanf("%d%d",&a,&b);

    printf("请输入操作类型：+|-|*|/\n");
    do
    {
        scanf("%c",&c);
    }while(c =='\n');//scanf遇到回车停止，取到数之后不丢弃/n


    switch(c)//这里是用C语言实现多态
    {
        case '+':
            p=add;break;
        case '-':
            p=sub;break;
        case '*':
            p=multiply;break;
        case '/':
            p=divide;break;
        default:
            p=NULL;break;
    }

    printf("结果为%d\n",p(a,b));
    return 0;
}

int add(int a, int b)
{
    return a+b;
```

```c
}
int sub(int a, int b)
{
    return a-b;
}
int multiply(int a, int b)
{
    return a*b;
}
int divide(int a, int b)
{
    return a/b;
}
```

- 分层实现上述内容（分层为了协作，linux内核就是如此分层）
  - 一个人写framework.c，写业务逻辑（实现功能，没有具体实现），并把相应的接口、结构体写到头文件（通过头文件协同工作）

framework.c

```c
#include "calcu.h"

int calculator(const struct cal_t *p)//传多个参数，打包成结构体，传指针
{
    return p->pfunc(p->a,p->b);
}
//架构代码里不要写printf调试信息，想要有所打印，应当使用返回值
```

calcu.h

```c
#ifndef __CALCU_H
#define __CALCU_H

typedef int(*pFunc)(int,int);

struct cal_t
{
   int a;
   int b;
   pFunc pfunc;
    /* data */
};

int calculator(const struct cal_t *p);


```

```
17 #endif
```

- 另一个人实现calcu.c完成具体计算器实现（填充数据），不需要看framework.c，看头文件calcu.h即可

calcu.c

```c
#include "calcu.h"
#include <stdio.h>

int add(int a, int b)
{
    return a+b;
}
int sub(int a, int b)
{
    return a-b;
}
int multiply(int a, int b)
{
    return a*b;
}
int divide(int a, int b)
{
    return a/b;
}

int main(void)
{
  int ret=0;
  struct cal_t myCal;
  myCal.a=2;
  myCal.b=3;
  myCal.pfunc=add;

  ret=calculator(&myCal);
  printf("return=%d\n",ret);

  return 0;
}


```

- 每个层次专注不同任务，不同层次用头文件交互，上层framework.c的存在是给下层调用的

> - 上层注重**业务逻辑（实现功能，与最终目标关联，没有具体实现）**，下层为上层填充变量，将变量传递给上层
> - 如何写下层代码？**定义结构体变量→填充结构体变量→调用上层接口函数，把结构体变量传给它**

typedef

- C语言中的两种类型：原生类型ADT、自定义类型UDT（数组类型、结构体类型、函数类型、函数指针、数组指针...），typedef用来为UDT重命名
- typedef重命名的是类型（相当于数据模板，不占内存，对应C++中的类），不是变量（真实数据，占用内存，对应C++中的对象）

> 为函数指针重命名

```
char *strcpy(char *dest, const char *src);

char *(*pFunc)(char *,const char*);//strcpy对应的函数指针
pFunc=strcpy;

typedef char *(*pType)(char *,const char*);
//将char *(*)(char *,const char*)类型重命名为pType
pType pFunc;//定义函数指针pFunc，相当于char *(*pFunc)(char *,const char*)
```

- #define和typedef区别（注意定义时的差别）
  - typedef 为类型取别名，define 简单替换
  - typedef 编译时处理，define 预编译处理

> 示例

```
#define dpchar char*
typedef char* tpchar

dpchar p1,p2;//相当于char *p1,p2;
tpchar p3,p4;//相当于char *p3, char* p4;

sizeof(p1);//4
sizeof(p2);//1
sizeof(p3);//4
sizeof(p4);//4
```

- typedef与结构体
  - struct先定义再使用，每次使用前面加上struct（用typedef重命名则可以省去struct）

结构体定义方式

```
struct tag {
    member-list
    member-list
    member-list
    ...
} variable-list ;//tag和variable-list至少出现一个


struct student
{
  char name[20];
  int age;
};

struct student
{
  char name[20];
  int age;
}student1;

typedef struct student
{
  char name[20];
  int age;
}student_t;//结构体有两个名字：struct student=student_t

typedef struct
{
  char name[20];
  int age;
}student_t;//结构体仅有一个名字：student_t
```

- 结构体指针：

示例

```
struct student*p;//方法1
student_t*p;//方法2

//方法3：一次定义两个类型（结构体类型、结构体指针类型）
typedef struct student
{
  char name[20];
  int age;
}student_t,*pStudent_t;
//第一个变量：结构体变量（struct student = student_t）
```

```
11 //第二个变量：结构体指针变量（struct student* = pStudent_t）
```

- typedef与const

示例

```
1 typedef int * PINT;
2 const PINT p;//不代表const int * P，等同于int* const p
3 PINT const p;//等同于int* const p
4
5 //若想表示const int *p，必须写成
6 typedef const int * PINT
7 PINT p;
```

- 为什么要使用typedef?
  - int、double等ADT在不同机器上位数不同，因此对类型取别名，在Linux内核中常用这种方式，另外在stm32库中全部使用了自定义类型。例如：typedef int size_t（一种类型可能会有很多别名）
  - 未来迁移到不同平台只要修改typedef即可，提高了可迁移性

二重指针

- 二重指针也是指针变量，占四个字节，指针指向的变量是一重指针/指针数组，可以看成指针数组的指针，编译器会帮忙做类型检查
- 二重指针的用法
  - 二重指针指向一重指针的地址

示例

```
1 char a;
2 char **p1;
3 char *p2;
4 p2=&a;
5 p1=&p2;
```

  - 二重指针指向指针数组（二重指针用的少，常是这种用法）

示例

```
1 int *p1[5];
2 //p1是数组，有五个元素，数组里存指针，指针指向int类型，p1是数组首元素的首地址int **
3 int **p2;
```

```
p2=p1;
```

- 将二重指针作为函数参数，目的是为了通过二重指针改变指针指向

示例

```
void func(int **p)
{
  *p=(int *)0x12345678;
}
int main(void)
{
  int a=4;
  int *p=&a;//p指向a
  func(&p);//改变p的指向
}
```

二维数组

- 内存角度看，一维数组和二维数组没有本质差别，都是连续分布的格子，访问效率完全一样，但更好理解（另：平面直角坐标系、四轴飞行器用三维数组）
- 二维数组的下标式访问和指针式访问

示例

```
//一维数组
int a[3];
int *p=a;
a[i]或者 *(p+i)

//二维数组
int a[2][3];
int (*p)[3];//p是个指针，指向数组，数组有3个元素，存的都是int类型
p=a;//a相当于&a[0]，二维数组第一维首元素首地址
*(*(p+i)+j) //相当于a[i][j]
```

- 二维数组初始化：int a[2][2]={{1,2},{3,4}}或int a[2][2]={1,2,3,4}
- 二维数组数组名：二维数组第一维数组首元素的首地址，a等同于&a[0]
- 指针指向二维数组第一维
  - a[0]指向二维数组第一维第一个元素，相当于&a[0][0]

示例

```
int a[2][4];
```

```
int *p1=a[0];
*(p1+2);//相当于a[0][2]
int *p2=a[1];
*(p2+2);//相当于a[1][2]
```

其它

- C语言是强类型语言，每个变量都有类型，编译器对变量类型进行检查；makefile是弱类型脚本语言，变量不需要类型

  示例

  ```
  int a[5];
  int (*p1)[5];//p1类型为int (*)5，与&a的类型相同
  p1=&a
  ```

- 函数本质是一段代码，在内存中连续分布
- 出现段错误怎么办？定位段错误（可疑处打印信息）
- **每个printf都要加上\n**
  - linux行缓冲，一行输入完了再一次性把一行输出（为了效率），通过换行符判定，不加上\n会不断缓冲，看不到内容输出
  - 程序终止也会输出缓冲区内容
  - linux中换行符为\n，windows换行符为\r\n，IOS换行符为\r
- scanf问题（现实中很少用，不要研究scanf）
  - scanf空格等都为分隔符，拿走字符后，分隔符还留在缓冲区→使用do...while(c=='/n')去除/n继续接收