指针

引言

假如你要在微信上联系小王，你可以直接点他的头像，然后开始聊天，也可以通过输入微信号来查找小明然后

开始聊天。指针就是一种类似于微信号的变量，但它存储的可不是微信号，而是内存地址。

指针基础

指针是什么

指针是存储计算机内存地址的一个变量，也是一种变量类型，它拥有一个int类型长度，这个长度会随着系统长度来变化，例如32位系统中，指针的长度为4个字节，而在64位系统中则为8个字节。变量存放的是值，而指针变量存放的是变量的内存首地址，相当于一个快捷方式，访问它就相当于通过该地址得到这个地址上这一段内存存放的值，属于间接访问。所以我们说，用指针指向一个变量，实际上就是用指针存储该变量的地址值。知道了一个变量的内存地址，那我们想访问这个地址上存储的变量，就可以通过指针运算符(*)来对该内存地址进行访问，得到这一地址下的变量值。

用一个例子举例，假设某段short类型数组a[9]中有如下的数据：

数组编号	0	1	2	3	4	5	6	7	8
内存地址	0x3200	0x3202	0x3204	0x3206	0x3208	0x320a	0x320c	0x320e	0x3210
内容	0x0702	0x0202	0x0000	0x320c	0x077f	0x032f0	0x0003	0x0002	0x0008

使用short *b = a;或short *b = &a[0];来声明一个指向"short类型，且值为a[0]"的指针，它存储的内容是a[0]的内存地址0x3201。指针运算符有(*)和(&)，如果需要获取某个变量的地址，可以使用取地址运算符(&)：

- char *pa = &a;
- &a:得到存放变量a的地址，将该地址值存放进指针变量pa

如果需要访问指针变量指向的数据，可以使用取值运算符（*);

- int a = 1;
- int *pa = &a;
- printf("a = %d",*pa);

*pa:访问该指针变量存放的地址所对应的变量值，运行结果应为: a = 1。

(*)在不同的场景下有不同的作用：(*)可以用在指针变量的定义中，表明这是一个指针变量，以和普通变量区 分开；使用指针变量时在前面加(*)表示获取指针指向的数据，或者说表示的是指针指向的数据本身。 也就是说，定义指针变量时的(*)和使用指针变量时的(*)意义完全不同。

指针的类型

诸如int*、char*、short*、float*等这些类型称为指针类型，其含义是指针指向的元素的变量类型，方便编译器对其进行正确处理。

特殊的，void*也是一种指针类型，代表无类型指针，什么类型都可以指，如(int*)能够直接赋值给无类型指针，而将void*类型的指针赋值给其他类型需要强制类型转换,例如有void*型指针pa，强制转成(int*)型指针就是(int*)pa。

指针的运算

指针支持+、-、++、--的运算，加减运算符其含义是对指针指向的内存地址的值进行偏移，自增/自减运算符(b++/b--)是对b加上(减去)一个指针指向类型的大小。上文(short*)型指针b，指向的是数组中首元素a[0]的内存首地址，数组中各个元素的内存地址是连续的，a[0]占用着两个字节内存，接着便是a[1]的两个字节内存，而(b + 1)是对b加上一个(short*)指针类型的大小，也就是2个字节，当b从a[0]的首地址跨过两个字节内存后，便来到了a[1]的内存首地址，即:(b + 1) = &a[1],同理，(b + 2) = &a[2];

例如：*(b+3)=0x320c，也就是b+3对应的内存地址是0x3206。

指针的作用

- 动态内存分配

  动态内存分配相比静态内存分配更具有灵活性，可以随用随分配，对于各类数据结果使用会灵活很多

- 传递参数

  我们可以把指针当作参数传递给函数，在函数里修改的数值指针能够在调用时同步修改函数外的变量。可以弥补只有返回值的缺点。

一般来说，如果该函数内修改的参数在函数外有使用，都需要使用指针传参。

例如：函数fun会修改a变量，主函数会使用fun修改后a的值，那么则需要将a的指针作为参数传递给fun。若a为指针同理。

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

void sort(int *a, int *b, int *c)
{ //对三个元素进行排序:a>b>c
  int   temp;
```

```
  //  冒泡排序
  if (*b > *a)
  {
    temp = *a;
    *a = *b;
    *b = temp;
  }
  if (*c > *b)
  {
    temp = *b;
    *b = *c;
    *c = temp;
  }
  if (*b > *a)
  {
    temp = *a;
    *a = *b;
    *b = temp;
  }
}
int main()
{
  int a = 1, b = 3, c = 2;
  sort(&a, &b, &c);
  printf("a=%d,b=%d,c=%d", a, b, c);
  return 0;
}
```

> **运行结果：**
>
> a=3,b=2,c=1

- 强制类型转换

比较少见的用途，c语言中浮点数是不支持位运算的，但可以取它地址再将其强制类型转换成一个整型来使用。比较出名的运用是快速平方根倒数算法，《雷神之锤III竞技场》源代码中平方根倒数速算法如下：

```
float Q_rsqrt(float number)
{
  long i;
  float x2,y;
  x2 = number * 0.5F;
  y = number;
  i = *(long *)&y;       //将y转化为long型
  i = 0x5f3759df - ( i >> 1);       //利用   浮点数   性质求出来的魔数
```

```
  y = *(  float  *)&i;       //将i转回float型
  y *= 1.5F - (x2 * y * y);       //牛顿迭代
  y *= 1.5F - (x2 * y * y);
  return y;
}
```

- 存储函数参数

在C语言开发中十分常见，将来学习stm32的库函数会大量接触，学习C++时接触面向对象(OOP)的类也和此类似。

用结构体来存储函数内部参数，从而只需要使用不同的结构体就能将同一功能函数用在不同地方。

例如常用的PID算法为了实现复用可以定义为(代码不要求理解，只观察如何使用结构体作为函数参数)：

```
typedef struct
{
  float     KP  ;                    //比例因子
  float   KI;                      //积分因子
  float     KD  ;                    //微分因子
  float   IntegralMax;             //积分最大值
  float   PIDMax;                  //  PID  最大值
  float   IntegralValue;           //积分
  float   DiffValue;               //误差
  int16_t AngleMax;              //角度最大值
  FlagStatus AngleContinuity;  //消除角度零点
} s_PIDConfig;
float PID(s_PIDConfig* PIDConfig,int16_t E_value, int16_t C_value)
{
  float new_DiffValue = 0, miss = 0;
  int   CurrentValue = 0;
  if(PIDConfig->AngleContinuity ==   SET  )
  {
    //角度突变点处理
    //机械角度超过半数则需要往反方向调节，此时要消除突变点
      if (abs(C_value - E_value) > (PIDConfig->AngleMax >> 1))
      {
        E_value += (E_value > C_value) ? -PIDConfig->AngleMax : PIDConfig->Ar
      }
  }
  //当前误差
  new_DiffValue = E_value - C_value;
  //误差变量率(即误差的  导数  )
  miss = new_DiffValue - PIDConfig->DiffValue;
```

```
  PIDConfig->DiffValue = new_DiffValue;
  //积分求和
  PIDConfig->IntegralValue += E_value - C_value;
  //抗积分饱和
  if (PIDConfig->IntegralValue > PIDConfig->IntegralMax)
    PIDConfig->IntegralValue = PIDConfig->IntegralMax;
  else if (PIDConfig->IntegralValue < -PIDConfig->IntegralMax)
    PIDConfig->IntegralValue = -PIDConfig->IntegralMax;
  //  PID
  CurrentValue = (int)((PIDConfig->KP * PIDConfig->DiffValue) + (PIDConfig->K
  //抗  PID  饱和
  if (CurrentValue > PIDConfig->PIDMax)
    CurrentValue = PIDConfig->PIDMax;
  if (CurrentValue < -PIDConfig->PIDMax)
    CurrentValue = -PIDConfig->PIDMax    ;
  return CurrentValue;
}
```

数组指针

简述

数组指针可以说是“指向数组的指针”，首先数组指针是一个指针，这个指针存放着一个数组的首地址，该型指针访问下一个地址就是下一个数组首地址了，而非普通指针指向下一个地址得到的是下一个变量元素。a[9]中的&a就是一个数组指针。

- 定义一个数组指针：
- char(*pa)[4];
- char a[4];
- pa = &a;

a是一个长度为4的字符数组，a是这个数组的首元素地址，但不能将a赋值给pa，因为a是数组首元素地址，而数组指针pa存放的是数组的首地址，a是char 类型，所以a+1，是指向a中下一个元素的地址，而pa指向char[4]类型的数组，所以pa+1，读取到的是下一组数组的首地址，pa会加4，虽然数组的首地址和数组首元素地址的值相同，但是两者操作不同，所以类型不匹配不能直接赋值，但是可以这样：pa = &a，现在它指向数组a，a是首元素地址，&a则是读取数组的首地址。在新标准中数组名 a 在表达式中也会被转换为和 pa 等价的指针，故char(*pa) = a也是可以的。

通过指针引用数组

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

```c

int main()
{
  short a[] = {0x0702, 0x0202, 0x0000, 0x320c, 0x077f, 0x32f0, 0x0003, 0x0002
  //声明一个如上图所示的数组
  //其地址并不固定，所以可以观察程序输出地址与运算符的关系。

  //数组名实际也是个指针，可以直接同维度指针赋值。
  //变量必须使用取地址符
  short *b = a;
  printf("一个short类型大小为:%dbyte\n", sizeof(short));
  printf("该指针地址为:%#x,", b);    //#x为地址类型
  printf("该指针内容为:%#x\n", *b);
  //通过取地址符获取地址
  b = &a[0];
  printf("该指针地址为:%#x,", b);
  printf("该指针内容为:%#x\n", *b);
  //指针的自增运算，自减可自行测试
  short *c = b;
  b++;
  printf("该指针地址为:%#x,", b);
  printf("该指针内容为:%#x\n", *b);
  //如果直接相减是按照指针的运算求元素个数，所以需要强制类型转换。
  printf("与上一个地址的差为:%dbyte\n", (  int  )b - (int)c);
  //指针的加减运算 此处b是a[1]的地址，所以+2会访问a[3]
  printf("该指针存储的地址为:%#x,", b + 2);
  printf("该指针指向的内容为:%#x\n", *(b + 2));
  printf("使用数组形式访问:%#x\n", b[2]);
  //指针的访问
  *b = 0x1419;
  printf("该指针存储的地址为:%#x,", b);
  printf("该指针指向的内容为:%#x\n", *b);

  //声明一个short变量
  short d = 0x1145;
  //此处变量必须用取地址符号才能正确访问
  //若直接赋值编译器会报错。
  //b = d;//可以取消注释来尝试
  b = &d;
  printf("该指针存储的地址为:%#x,", b);
  printf("该指针指向的内容为:%#x\n", *b);
  //对指向变量+1或-1会引发指针越界
  //对超过数组大小的访问会引起数组越界
  //对其写操作可能会引发程序崩溃
  printf("该指针存储的地址为:%#x,", b + 1);
  printf("该指针指向的内容为:%#x\n", b[1]);
  return 0;
```

```
50 }
```

> **运行结果：**
>
> 一个short类型大小为:2byte
>
> 该指针地址为:0x65febe,该指针内容为:0x702
>
> 该指针地址为:0x65febe,该指针内容为:0x702
>
> 该指针地址为:0x65fec0,该指针内容为:0x202
>
> 与上一个地址的差为:2
>
> 该指针存储的地址为:0x65fec4,该指针指向的内容为:0x320c
>
> 使用数组形式访问:0x320c
>
> 该指针存储的地址为:0x65fec0,该指针指向的内容为:0x1419
>
> 该指针存储的地址为:0x65febc,该指针指向的内容为:0x1145
>
> 该指针存储的地址为:0x65febe,该指针指向的内容为:0x702

通过指针引用二维数组

- 首先引入二维数组的定义：二维数组在概念上是二维的，有行有列，但在内存中所有的元素都是连续排列的，以下面的二维数组为例：

  int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};

  - 从概念上理解，a的分布就像一个矩阵;
  - 从内存上理解,整个数组占用一块连续的内存;

  C语言中的二维数组是按行排列的，也就是先存放 a[0] 行，再存放 a[1] 行，最后存放 a[2] 行；每行中的 4 个元素也是依次存放。数组 a 为 int 类型，每个元素占用 4 个字节，整个数组共占用 4×(3×4) = 48 个字节。C语言允许把一个二维数组分解成多个一维数组来处理。对于数组a，可以理解为存储了三个一维数组的一个数组arr[3]，即 arr[0]、arr[1]、arr[2]。每一个一维数组又包含了 4 个元素，例如 arr[0] 包含 a[0][0]、a[0][1]、a[0][2]、a[0][3]。

  假设数组a中第0个元素的地址为1000，那么每个一维数组的首地址如下图所示：

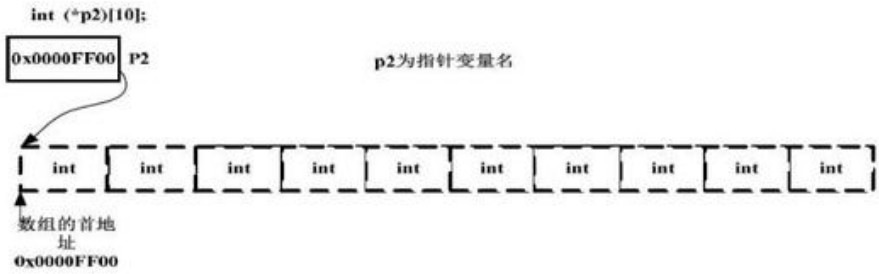
  为了更好的理解指针和二维数组的关系，我们先来定义一个指向 a 的指针变量 pa：

  int (*pa)[4] = a ;

  括号中的*表明 pa 是一个指针，它指向一个数组，数组的类型为int [4]，这正是 a 所包含的每个一维数组的类型。[]的优先级高于*，()是必须要加的，如果直接写作int *pa[4]，那么应该理解为int *(pa[4])，pa 就成了一个指针数组，而不是二维数组指针。对指针进行加法（减法）运算时，它前进（后退）的步长与它指向的数据类型有关，pa 指向的数据类型是int [4]，那么pa + 1就前进 4 × 4 = 16 个字节，pa - 1就后退 16 个字节，这正好是数组 a 所包含的每个一维数组

的长度。也就是说，pa + 1会使得指针指向二维数组的下一行，pa - 1会使得指针指向数组的上一行。数组名 a 在表达式中也会被转换为和 pa 等价的指针！

概念图如以下所示：



下面我们就来探索一下如何使用指针 pa 来访问二维数组中的每个元素。按照上面的定义：

- pa指向数组 a 的开头，也即第 0 行；pa + 1前进一行，指向第 1 行。
- *(pa + 1)表示取地址上的数据，也就是整个第 1 行数据。注意是一行数据，是多个数据，不是第 1 行中的第 0 个元素的地址，下面的运行结果有力地证明了这一点：

```
#include <stdio.h>
int main()
{
    int a[3][4] = { {0, 1, 2, 3}, {4, 5, 6, 7}, {8, 9, 10, 11} };
    int (*pa)[4] = a;
    printf("%d\n", sizeof(*(pa+1)));
    return 0;
}
```

> **运行结果：**
>
> 16

*(pa + 1) + 1表示第 1 行第 1 个元素的地址。如何理解呢？

pa指向的是一个数组，每一次pa + 1都是递增一个数组，而对pa解引用(*pa)得到的就是数组的首元素地址，pa指向最外层的数组，每次解引用就再进去一层，*pa就进去到数组中的元素的地址，**pa就是进去到元素的值。*(pa + 1)单独使用时表示的是第 1 行数据，放在表达式中会被转换为第 1 行数组的首元素地址，也就是第 1 行第 0 个元素的地址，因为使用整行数据没有实际的含义，编译器遇到这种情况都会转换为指向该行第 0 个元素的指针；就像一维数组的名字，在定义时或者和 sizeof、& 一起使用时才表示整个数组，出现在表达式中就会被转换为指向数组第 0 个元素的指针

((p+1)+1)表示第 1 行第 1 个元素的值。很明显，增加一个 * 表示取地址上的数据。

根据上面的结论，可以很容易推出以下的等价关系：

a+i == p+i

a[i] == p[i] == *(a+i) == *(p+i)

a[i][j] == p[i][j] == *(a[i]+j) == *(p[i]+j) == *(*(a+i)+j) == *(*(p+i)+j)

【实例】使用指针遍历二维数组。

```
#include <stdio.h>
int main()
{
    int a[3][4]={0,1,2,3,4,5,6,7,8,9,10,11};
    int(*p)[4];
    int i,j;
    p=a;
    for(i=0; i<3; i++)
  {
    for(j=0; j<4; j++)
      printf("%2d  ",*(*(p+i)+j)      ;
    printf("\n");
  }
  return 0;
}
```

运行结果：

```
0  1  2  3
4  5  6  7
8  9 10  11
```

多维指针

简述

我们平常用的指针*p是一维指针，它直接指向数据。

但也存在诸如short **p等这类指针，这类指针称为多维指针，**p称为二维指针。

多维指针是指向指针的指针，拿现实中的例子举例：你想找小王，但你没小王的联系方式，但你有小王的好朋友小张的联系方式，于是你找到小张要到了小王的联系方式再联系小王。此处小张的联系方式(地址)就是一个二维指针，用来存储小王的联系方式(地址)。

访问多维指针

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

```c

int main()
{
  short a[] = {0x0702, 0x0202, 0x0000, 0x320c, 0x077f, 0x32f0, 0x0003, 0x0002
  short d[2][4] =
    {    0x1145, 0x1419, 0x1981, 0x0000,
    0x0478, 0x0273, 0x0231, 0x0413};
  //声明一个如上图所示的数组
  //数组名实际也是个指针，可以直接同维度指针赋值。
  //变量必须使用取地址符
  short *b = a;
  //此处声明了一个二维指针,内部存储的是b的地址
  short **c = &b;
  printf("数组a的地址为:%#x,", a);
  printf("数组b的地址为:%#x,", b);
  printf("b的地址为：%#x\n", &b);
  printf("该指针存储的地址为:%#x,", c);
  printf("该指针指向的内容为:%#x\n", *c);
  printf("该指针指向的指针的指针为：%#x\n", **c);
  printf("用数组的形式访问：%#x\n", c[0][1]);
  //将  二维数组  第1行赋值给b
  b = d[1];
  printf("  二维数组  第1行第1列的元素为：%#x\n", d[1][1]);
  printf("使用b指针来访问：%#x", b[1]);
  return 0;
}
```

> **运行结果：**
>
> 数组a的地址为:0x65febe,数组b的地址为:0x65febe,b的地址为：0x65fea8
>
> 该指针存储的地址为:0x65fea8,该指针指向的内容为:0x65febe
>
> 该指针指向的指针的指针为：0x702
>
> 用数组的形式访问：0x202
>
> 二维数组第1行第1列的元素为：0x273
>
> 使用b指针来访问：0x273

结构体指针

简述

所谓结构体指针就是指向结构体变量的指针，一个结构体变量的起始地址就是这个结构体变量的指针。

指向结构体变量的指针

- 定义结构体变量的指针

```
struct student  //定义了一个结构体
{
  long stuid;
  char   name[10];
  int   age;
  int   score;
}stu1;   //定义了一个结构体变量

struct student stu2;  //因为结构体含有名字，所以可以继续定义结构体变量stu2
struct student *stu3;   //定义了一个结构体指针
stu3 = &stu1; //同指针一样，结构体指针指向结构体变量
/*可替换为
struct student * stu3 = &stu1; //定义的同时初始化
struct student *stu3;
*stu3=stu1;
*/
struct student stu4[10];  //定义了一个结构体数组c
```

- 访问结构体变量的成员

```
//方法一：
//同结构体变量一样，通过成员选择运算符
stu1.age = 18;
(*stu3).age = 19;   //通过对stu3的解引用
//方法二：
//通过指向运算符来访问，例如：
stu3 -> age = 19;
```

- 结构体嵌套的访问

```
stu1.birthday.year = 1998;
(*stu3).birthday.year =1998;
stu3 -> birthday . year =1998;
```

指向结构体数组的指针

- 定义结构体数组的指针

```
info stu[40]; //定义了一个结构体数组
info *pt;   //定义了结构体指针
pt = stu;   //指针初始化方法一
info *pt1 = &stu[0]; //初始化方法二
info *pt2 = stu;   //方法三
```

- 访问结构体数组的成员

```
#include <stdio.h>

int main()
{
  typedef struct information // 定义了一个结构体
  {
    long stuid;
    char name[10];
    int age;
    int score;
  } info;       // 别名
  info stu[40]; // info为结构体标签的别名
  info *pt;
  pt = stu;

  stu[0].stuid = 1001;
  (*pt).stuid = 1001;
  pt->stuid = 1001; // 三条语句等价

  stu[1].stuid = 1002;
  (pt++)->stuid = 1002;
  (*pt++).stuid = 1002; // 语句等价
}
```

函数指针

概念

函数指针是指向函数的指针变量。

通常我们说的指针变量是指向一个整型、字符型或数组等变量，而函数指针是指向函数。

函数指针可以像一般函数一样，用于调用函数、传递参数。

函数指针的定义方式为：

函数返回值类型 (* 指针变量名) (函数参数列表);

“函数返回值类型”表示该指针变量可以指向具有什么返回值类型的函数；“函数参数列表”表示该指针变量可以指向具有什么参数列表的函数。这个参数列表中只需要写函数的参数类型即可。

如何用函数指针调用函数

给大家举一个例子：

```
int Func(int x);   /*声明一个函数*/
int (*p) (int x);  /*定义一个函数指针*/
p = Func;          /*将Func函数的首地址赋给指针变量p*/
p = &Func;          /*将Func函数的首地址赋给指针变量p*/
```

赋值时函数 Func 不带括号，也不带参数。由于函数名 Func 代表函数的首地址，因此经过赋值以后，指针变量 p 就指向函数 Func() 代码的首地址了。

例程：

```
#include <stdio.h>
int Max(int, int);  //函数声明
int main(void)
{
    int(*p)(int, int);  //定义一个函数指针
    int a, b, c;
    p = Max;  //把函数Max赋给指针变量p, 使p指向Max函数
    printf("please enter a and b:");
    scanf("%d%d", &a, &b);
    c = (*p)(a, b);  //通过函数指针调用Max函数
    printf("a = %d\nb = %d\nmax = %d\n", a, b, c);
    return 0;
}
int Max(int x, int y)  //定义Max函数
{
    int z;
    if (x > y)
    {
        z = x;
    }
    else
    {
        z = y;
    }
    return z;
}
```

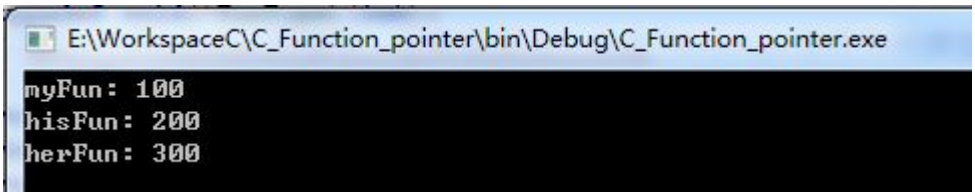
函数指针作为某个函数的参数

既然函数指针变量是一个变量，当然也可以作为某个函数的参数来使用的。

示例：

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

typedef void(*FunType)(int);
//前加一个typedef关键字，这样就定义一个名为FunType函数指针类型，而不是一个FunType变量。
//形式同 typedef int* PINT;
void myFun(int x);
void hisFun(int x);
void herFun(int x);
void callFun(FunType fp,int x);
int main()
{
    callFun(myFun,100);//传入函数指针常量，作为回调函数
    callFun(hisFun,200);
    callFun(herFun,300);

    return 0;
}

void callFun(FunType fp,int x)
{
    fp(x);//通过fp的指针执行传递进来的函数，注意fp所指的函数有一个参数
}

void myFun(int x)
{
    printf("myFun: %d\n",x);
}
void hisFun(int x)
{
    printf("hisFun: %d\n",x);
}
void herFun(int x)
{
    printf("herFun: %d\n",x);
}
```

输出：



回调函数

什么是回调函数

回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。

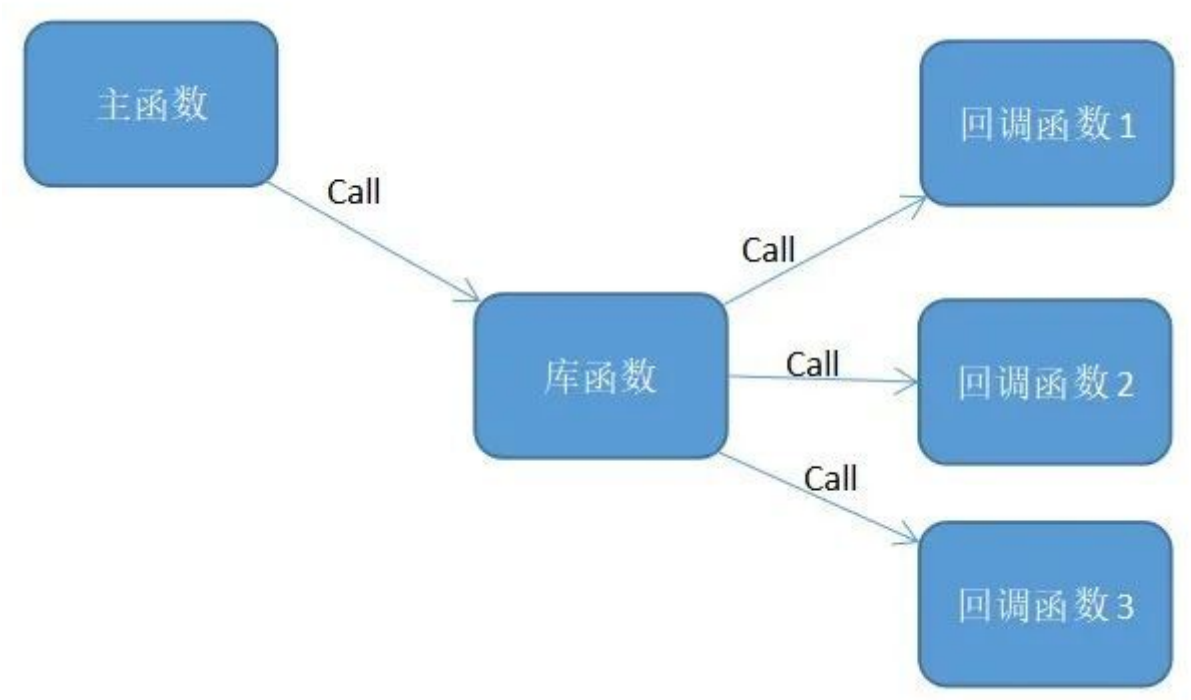
把一段可执行的代码像参数传递那样传给其他代码，而这段代码会在某个时刻被调用执行，这就叫做 回调 。

如果代码立即被执行就称为 同步回调 ，如果过后再执行，则称之为 异步回调 。

为什么要用回调函数?

因为可以把调用者与被调用者分开，所以调用者不关心谁是被调用者。它只需知道存在一个具有特定原型和限制条件的被调用函数。

简而言之，回调函数就是允许用户把需要调用的方法的指针作为参数传递给一个函数，以便该函数在处理相似事件的时候可以灵活的使用不同的方法。




```
1 int Callback()    ///< 回调函数
```

```
{
    // TODO
    return 0;
}
int main()      ///<  主函数
{
    // TODO
    Library(Callback);  ///< 库函数通过函数指针进行回调
    // TODO
    return 0;
}

```

回调似乎只是函数间的调用，和普通函数调用没啥区别。

但仔细看，可以发现两者之间的一个关键的不同：在回调中，主程序把回调函数像参数一样传入库函数。

这样一来，只要我们改变传进库函数的参数，就可以实现不同的功能，这样有没有觉得很灵活？并且当库函数很复杂或者不可见的时候利用回调函数就显得十分优秀。

怎么使用回调函数?

```
int Callback_1(int a)   ///< 回调函数1
{
    printf("Hello, this is Callback_1: a = %d ", a);
    return 0;
}

int Callback_2(int b)  ///< 回调函数2
{
    printf("Hello, this is Callback_2: b = %d ", b);
    return 0;
}

int Callback_3(int c)   ///< 回调函数3
{
    printf("Hello, this is Callback_3: c = %d ", c);
    return 0;
}

int Handle(int x, int (*Callback)(int)) ///< 注意这里用到的函数指针定义
{
    Callback(x);
}
```

```
23
24 int main()
25 {
26     Handle(4, Callback_1);
27     Handle(5, Callback_2);
28     Handle(6, Callback_3);
29     return 0;
30 }
31
```

如上述代码：可以看到，Handle()函数里面的参数是一个指针，在main()函数里调用Handle()函数的时候，给它传入了函数Callback_1()/Callback_2()/Callback_3()的函数名，这时候的函数名就是对应函数的指针，也就是说，回调函数其实就是函数指针的一种用法。

指针的动态内存分配

简述

指针跟数组最大区别就是指针是能够动态分配的，而数组只能在编译时分配一个固定长度。动态内存分配的函数主要是使用malloc、realloc、free三个函数组成，他们位于stdlib.h内。malloc用来分配内存，realloc用来重新分配内存，free用来释放内存。

建立内存的动态分配

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

int main()
{
  int   *p;
  p = (int *)  malloc  (10 * sizeof(int));
  //  malloc  用来声明一段长度为10*  int  类型大小的变量
  //  malloc  的参数为声明的  内存  大小，它等于声明元素个数*类型大小。
  //sizeof(  int  )为求int的类型大小。
  //(  int  *)是将void*强制类型转换为int*，去掉这个会报错。
  //等价于:
  //  int   a[10]; p=a;
  for (int i = 0; i < 10; i++)
  {
    p[i] = i;
    printf("%d ", p[i]);
  }
  //用一句  for循环  来为p[i]赋值为1,2,...,9。
```

```c
  //然后打印到屏幕上。
  p = (int *)realloc(p, 20 * sizeof(int));
  //使用realloc来重新分配  内存  ，上述语法用来声明一段长度为20*  int  类型大小的指针
  //realloc的第一个参数为需要重新分配的指针，第二个参数为新的  内存  大小
  //重新分配并不影响丢失原来的数据(重新分配大于原先大小)
  //下面用一个  for循环  来举例。
  printf("\n现在打印0~19\n");
  for (int i = 10; i < 20; i++)
  {
    p[i] = i;
  }
  for (int i = 0; i < 20; i++)
  {
    printf("%d ", p[i]);
  }
  //当指针不在使用时需要及时释放，不然会引起  内存  泄露
  //使用语句是free，它的参数为需要释放的指针。
  free(p);
  p =   NULL  ;
  //指针释放完需要将其指向  NULL  ，防止误操作。
  //  内存  泄露：使用完的内存未及时释放导致程序无法访问该内存，导致电脑内存被占用。
/***************
警告
***************
下面这段程序用来显示内存泄露的危害。
请开启任务管理器观察内存的使用情况。
请确保保存好当前正在运作的任务，
此操作会导致电脑死机而导致当前未保存的内容丢失。
删除下方"/*"开始此程序
***************/
/*
  for(;;)
  {
    p = (int *)    malloc    (sizeof(int));
  }//使用一个死循环一直申请  内存  而不释放，最终会占用电脑所有内存*/
  return 0;
}
```

运行结果：

0 1 2 3 4 5 6 7 8 9

现在打印0~19

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

拓展：在C语言中实现面向对象思想

从一个例子开始

一般的，要表示一名学生，而学生拥有很多属性，像是学生的学号、姓名、性别、考试分数等等。联想我们学过的结构体知识，我们会想到声明一个结构体来表示学生，如下我们定义一个用于表示学生的结构体

```
struct student
{
    int id; // 学号
    char name[20]; // 姓名
    int mark; // 成绩
};
```

学生的学号规定是由入学年份、班级、序号拼接而成。例如某一个同学是2023年入学的123班的26号学生，那么他的学号就是202312326。我们为了更加方便的设置学号，我们定义一个函数makeStudentID，参数即为入学年份、班级、序号，它将这些数据转成字符串，将字符串拼接完成后再将整个字符串转换成整形数据作为学生id返回。

```
int makeStudentId(int year, int classNum, int serialNum)
{
    char buffer[20];
    sprintf(buffer, "%d%d%d", year, classNum, serialNum);
    int id = atoi(buffer);
    return id;
}
```

sprintf和printf函数类似，printf函数会将占位符"%d%d%d"替换为其后的参数，将结果打印到控制台上。而sprintf不会将结果打印在控制台上，而是将结果存放在第一个参数buffer所指示的字符数组当中。

函数atoi能将buffer指示的字符串转换为整型并返回结果。

现在，表示学生属性的结构体和设置学生学号的函数都已经准备好了，我们来表示一个2023年入学、231班、11号的小原同学，我们将使用以下方式，调用学生属性结构体和学生id设置函数

```
int main()
{
    //设置一个新的学生属性结构体
    struct student stu;
    // 设置数值
    // 学号：202322111
    // 姓名：小原
```

```
    // 成绩：98
    stu.id = makeStudentId(2023, 221, 11);
    strcpy(stu.name, "小原");
    stu.mark = 98;
    // 打印这些数值
    printf("学号:%d\n", stu.id);
    printf("姓名:%s\n", stu.name);
    const char* gender = numGenderToStrGender(stu.gender);
    printf("分数:%d\n", stu.mark);
    return 0;
}
```

这就是我们学到现在为止，我们要实现表示一名学生这个目的的一般步骤。接下来，我们以面向对象风格来重新审视这段代码，在面向对象风格中，结构体被看作数据(data)，而操作数据的函数称为方法。现在函数并不是直接操作数据，即并不直接操作结构体，我们只是拿到函数的返回值，再将返回值赋值给数据。而面向对象编程风格中的一大特性——封装，要求将数据和方法结合在一起，它们构成一个整体，这就构成了一个类。

让我们对设置学生学号的函数进行修改：将函数的第一个参数设置为 struct *student，让函数直接操作student结构体，修改函数名，将设置数据的方法命名为setXXX

```
void setStudentId(struct student* s, int year, int classNum, int serialNum)
{
    char buffer[20];
    sprintf(buffer, "%d%d%d", year, classNum, serialNum);
    int id = atoi(buffer);
    s->id = id;
}
```

修改以后，当我们想设置学号时，只需用修改后的函数，直接操作student结构体

```
setStudentId(&stu, 2023, 231, 11);
```

目前，函数可以直接操作数据了，但函数和数据仍然是两个独立的部分。要实现封装性的话，我们要将函数和数据封装进一个结构体成为一个整体，这样这个整体就能称作一个类，然后对类中的函数和数据分别去赋予具体函数和数据后，类就有了具体的属性和方法，成为一个具体的东西，我们称之为对象。对象是类的一个实例， 类是创建对象的蓝图或模板。结构体中的数据就是对象的属性，结构体中的函数就可以称作对象的方法。

大多数面向对象语言都是用以下格式调用一个对象的方法。

```
对象.方法(对象指针，参数1，参数2，参数3...)
```

在我们的例子中，就是下面这种格式：

```
stu.setStudentID(&stu,2023,221,11);
```

以上代码中，对象为stu，方法为setStudentID。通过对象.方法的形式调用对象stu中的setStudentID方法。在C语言中，为了在对象中储存一个方法，也就是一个函数，我们可以在学生类这个结构体中声明方法的函数指针

```
struct student
{
    void (*setStudentId)(struct student* s, int year, int classNum, int serialNu
    int id; // 学号
    char name[20]; // 姓名
    int mark; // 分数
};

```

为了让函数指针有一个正确的朝向，我们需要通过一个initStudent函数来初始化函数指针

```
void initStudent(struct student*)
{
    s->setStudentID = setStudentID;  //指向实际函数
}
```

做完了这些，我们就把一个学生对象的属性和方法都封装在一起了。

```
struct student stu;  //新建立一个对象
initStudent(&stu);  // 初始化student
// 学号：202323111
// 姓名：小原
// 分数：98
stu.setStudentId(&stu, 2023, 231, 11);
strcpy(stu.name, "小明");
stu.mark = 98;
// 打印这些数值
printf("学号:%d\n", stu.id);
```

```
11 printf("姓名:%s\n", stu.name);
12 printf("分数:%d\n", stu.mark);
13
```

这里有一个需要注意的地方，结构体声明后，结构体内的函数指针是无效的，必须使用initStudent函数，设置函数指针的正确指向。

上述步骤完成了面向对象的封装特性，接下来我们来完成继承特性

学校中除了学生以外，还需要有老师，老师也具有很多属性。例如

- 工号
- 姓名
- 任课科目

所以还是一样，首先声明一个结构体用于表示老师

```
1 struct teacher
2 {
3     int id;  //工号
4     char name[20];  //姓名
5     char subject[20];  //任课科目
6 };
```

比较老师和学生的结构体

```
1 //老师
2 struct teacher
3 {
4     int id;  //工号
5     char name[20];  //姓名
6     char subject[20];  //任课科目
7 };
8
9 //学生
10 struct student
11 {
12     int id; // 学号
13     char name[20]; // 姓名
14     int mark; // 分数
15 };
16
```

可以看到其中，姓名和学号是两者共有的。老师和学生都是学校里的人员，学校里的人员都有编号和名字，而老师和学生是在都是学校里的人员的基础上，再去拥有不同的性质，再分别是老师和学生。那我们可以把老师和学生的共同性质姓名和编号给抽象出来，成为一个person类

```
struct person
{
    int id;  //编号
    char name[20];  //姓名
};
```

接下来，我们再用分别用老师和学生的类去包含这个person类

```
struct teacher
{
    struct person person;
    char subject[20];  //任课科目
};

struct student
{
    struct person person;
    int mark;  //分数
};
```

在现在的代码，我们将编号姓名两个变量抽象成一个person类，这样一来，有两个好处：

1. 减少重复代码
2. 代码层次更加清晰

由于student和teacher类都是在person类的基础上建立的，因此我们可以说student类和teacher类均继承于person类，person是student和teacher的父类，student和teacher是person的子类，这就是一个简单的类的继承例子

老师和学生的类创建完成后，老师和学生还需要增加优秀度这一属性，老师是评价分数大于60分就可以评为优秀教师，学生是综评分数大于80可以评为优秀学生，为此我们要对老师和学生的类新增属性和方法

```
struct teacher
{
    struct person person;
```

```
    void (*checkgood)(struct teacher* t);
    char subject[20];  //任课科目
    int point;  //评价分数
    char evaluate[20];  //优秀or不优秀
};

struct student
{
    struct person person;
    void char* (*checkgood)(struct student* t);
    int mark;  //分数
    int point;  //综评分数
    char evaluate[20];  //优秀or不优秀
};
```

老师类中方法的实例

```
void checkgood_t(struct teacher* t)
{
    if ((t -> point) >= 60)
    {
        strcpy(t->evaluate,"优秀");
    }
    else if ((t->point) < 60)
    {
        strcpy(t->evaluate,"  良好");
    }
}

```

学生类中方法的实例

```
void checkgood_s(struct student* s)
{
    if ((s -> point) >= 80)
    {
        strcpy(s->evaluate,"优秀");
    }
    else if ((s->point) < 80)
    {
        strcpy(s->evaluate,"  良好");
    }
}
```

然后对函数指针执行初始化

```
void initStudent(struct student* s)
{
    s->checkgood = checkgood_s;
}

void initTeacher(struct teacher* t)
{
    t->checkgood = checkgood_t;
}
```

我们可以发现老师和学生的类中都有一个checkgood方法用于评价优良程度，我们可以将checkgood这个方法抽象出来，放进person类中，让teacher和student分别都包含person类，这样teacher和student就都可以使用checkgood这个方法了。

```
struct person
{
    void char* (*checkgood)(struct person* p);
    int id;  //编号
    char name[20];  //姓名
};

struct teacher
{
    struct person person;
    char subject[20];  //任课科目
    int point;  //评价分数
    char evaluate[20];  //优秀or不优秀
};

struct student
{
    struct person person;
    int mark;  //分数
    int point;  //综评分数
    char evaluate[20];  //优秀or不优秀
};
```

这里有一个注意的地方，父类结构体和子类结构体的内存首地址必须是一样的，结构体的内存首地址就是结构体内第一个元素的内存首地址，所以像下面那样声明student才是正确的

```
struct student
{
    struct person person;
    int mark;  //分数
    int point;  //综评分数
    char evaluate[20];  //优秀or不优秀
};
```

student类的首地址 == 结构体内首元素的地址 == person类的首地址，这样就完成了父类和子类内存排布的重合。

接着，我们要修改各对象的初始化函数，将原有的 s -> checkgood 改为 s -> person.check。

```
void initStudent(struct student* s)
{
    s -> person.check = checkgood_s;
}

void initTeacher(struct teacher* t)
{
    t -> person.check = checkgood_t;
}
```

其实这里有一个问题，等号左边的s -> person.checkgood是void(*)(struct person*)类型，参数是struct person*，

而等号右边的函数指针类型分别是

```
void(*)(struct student*)
void(*)(struct teacher*)
```

两边的函数指针类型并不一致，无法进行赋值，但只要我们将函数指针强制类型转换为void(*)(struct person*)，赋值便能正常进行

```
void initStudent(struct student* s)
{
    s -> person.check = (void(*)(struct person*))checkgood_s;
}

void initTeacher(struct teacher* t)
```

```
{
    t -> person.check = (void(*)(struct person*))checkgood_t;
}
```

接下来我们来思考应该如何去使用这些由类创建的对象

首先声明student1和teacher1这2个对象，并使用初始化列表对其进行初始化，由于student1中的第1个成员是person，所以这里再使用1个列表{}，将person成员初始化

```
struct student student1 = { {NULL,0," "},0,0," " };
struct teacher teacher1 = { {NULL,0," "},0,0," " };
```

然后再使用各自的初始化函数

```
initStudent(&student1);
initTeacher(&teacher1);
```

然后声明一个struct person*类型的指针

```
struct person* Person;  //声明一个父对象指针
Person = (struct person*)(&student1);  //将子对象指针强制类型转换为父对象指针
Person -> checkgood(&Person);  //调用方法

Person = (struct person*)(&teacher1);
Person -> checkgood(&Person);
```

在上面的代码中，当我们使用父对象指针Person来操作两个子对象时，使用相同的接口调用了不同函数，也即实现了面向对象的最后一个特性——多态，多态意味着调用成员函数时，会根据对象的不同来执行不同的函数。

基础练习

- 请问下边代码是否可以正常执行？如果可以，会打印什么值？如果不行，请说明原因？(10分)

```
#include <stdio.h>

int main()
{
    int a[5] = {1, 2, 3, 4, 5};
```

```
    int *b;

    b = &a[3];
    printf("%d\n", b[-2]);
    return 0;
}
```

- 为什么不能使用 if (str1 == str2) 这样的形式来比较两个字符串？(10分)
- 使用指针+循环的形式访问一个一维数组并打印到屏幕(10分)
- 使用指针指向一个常量字符串(“Hello World!”),并调用printf输出到屏幕(10分)
- 设计一个函数，输入为一个长度为n的整型数组，输出为数组的最大值和最小值。(10分)
- 回答下列printf的输出值(10分)

```
int main()
{
    int data[2][5] = {1,2,3,4,5,6,7,8,9,0};
    int (*p)[5] = data;
    printf("%d\n",data[0][2]);
    printf("%d\n",p[1][2]);
    printf("%d\n",*(data[0]+2));
    printf("%d\n",*(data[1]+1));
    printf("%d\n",*(p[1]+3));
    printf("%d\n",*(p[0]+2));
    printf("%d\n",*(*(data+1)+3));
    printf("%d\n",*(*(data+0)+3));
    printf("%d\n",*(*(p+0)+0));
    printf("%d\n",*(*(p+1)+4));
}
```

- 设计一个函数，其功能为计算并返回kx+b，x为输入参数。使用结构体保存k和b的值，并使用多个结构体来保存不同的k和b的值从而让该函数能输出不同结果。(可参考9.4)(30分)
- 到洛谷题单（*https://www.luogu.com.cn/training/list*）中完成前1-6个入门级题目，每个小结根据自己个人需求刷一定数量（不一定要全部刷完，但需要会用）(10分)