step further

专题

起步：

认知与体验（硬件、软件、程序与C语言）

进阶：

判断与推理（流程控制方法、语句）

抽象与联系（模块设计方法、函数）

表达与转换（基本操作、数据类型）

提高：

构造与访问（数组、**指针**、结构体）

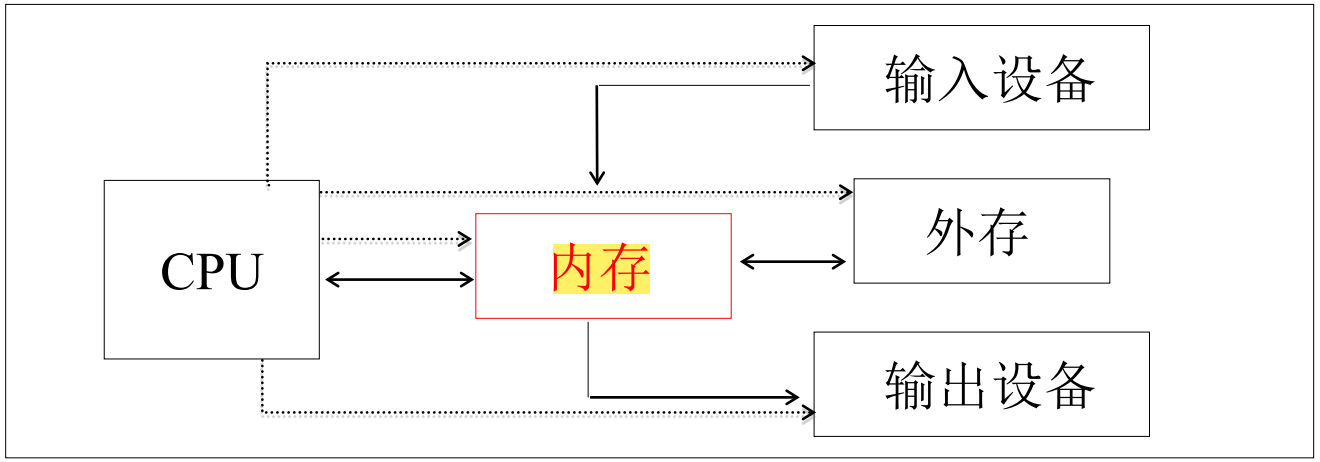
归纳与推广（程序设计的本质）

指针及其运用

- 指针的基本概念
  - 指针类型的构造
  - 指针变量的定义与初始化
  - 指针的基本操作
- 用指针操纵数组
- **用指针在函数间传递数据**
- **用指针访问动态变量**
- 用指针操纵函数*

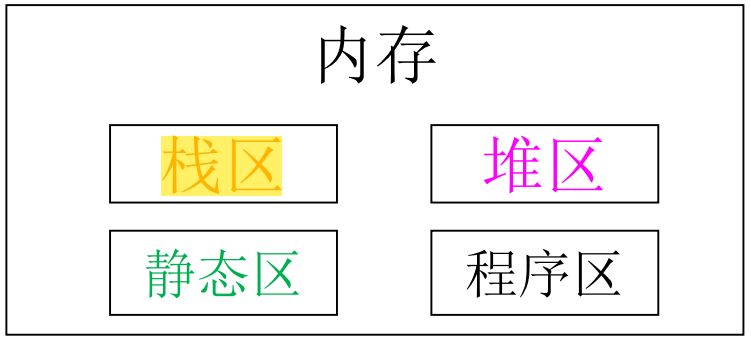
回顾

- **冯诺依曼体系结构示意图**



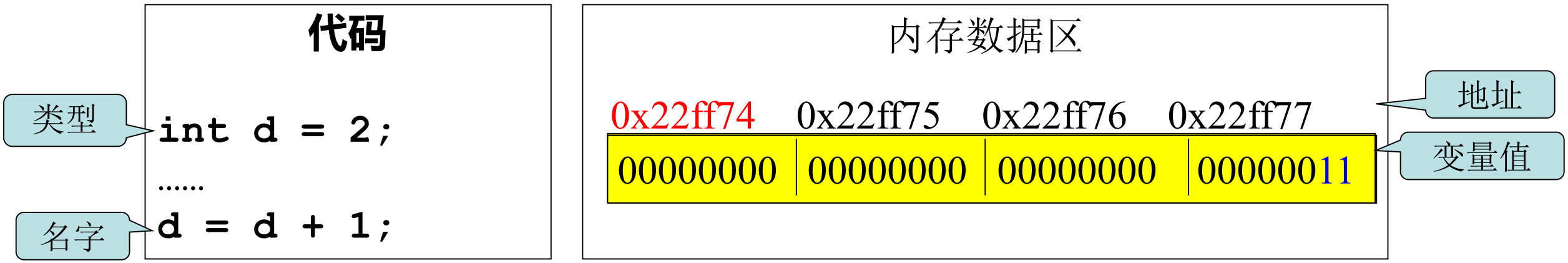
虚线箭头表示控制流

实线箭头表示数据流



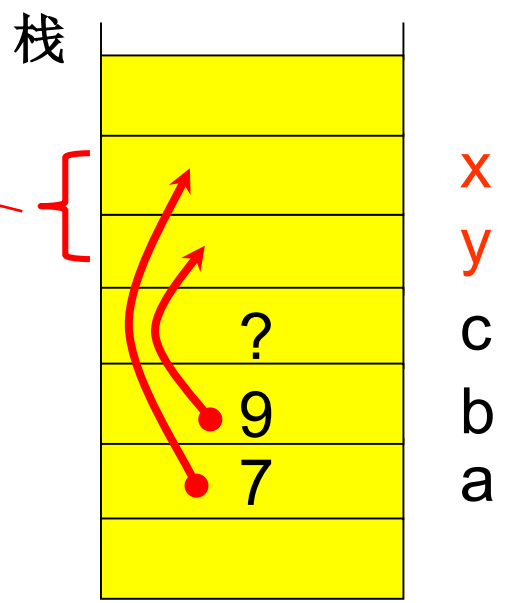
变量的属性

- 变量名
- 变量的类型：决定变量所占用内存单元的个数
- 变量的地址：每个变量实际占用的若干个内存单元中第一个内存单元的地址（即首地址）。
- 变量的值
- …



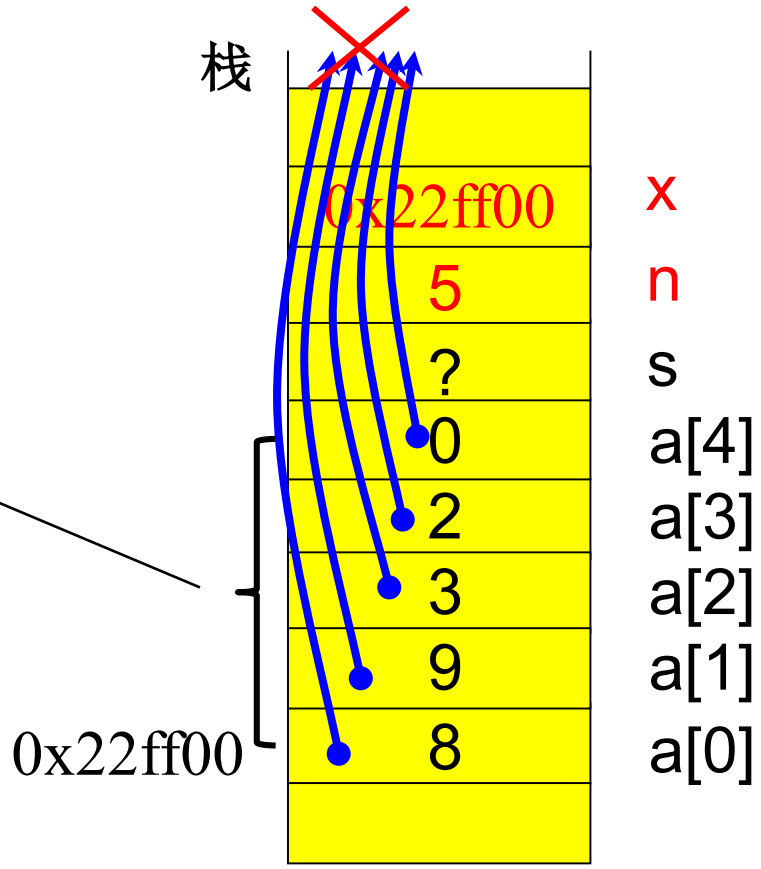
函数调用的过程

```
int Sum(int x, int y)
{
    return x + y;
}

int main()
{   int a, b, c;
    scanf("%d%d",&a, &b);
    c = Sum(a, b);
    printf("%d", c);
    return 0;
}
```



函数调用的过程

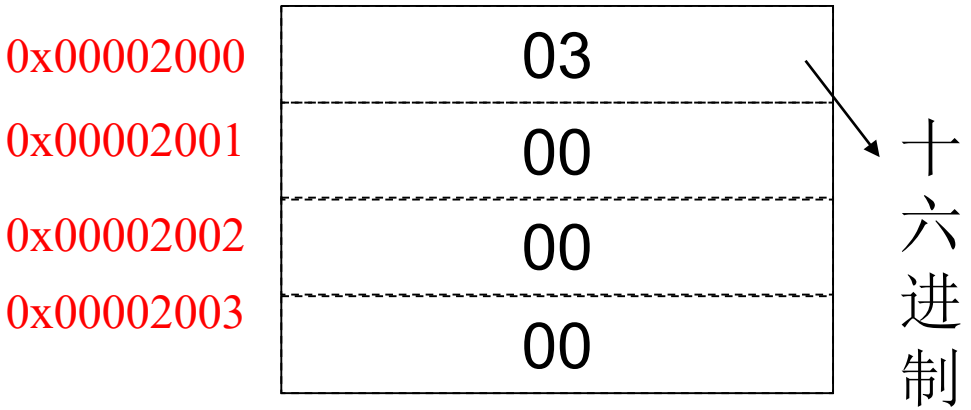
```
int Sum(int x[ ], int n)
{   int s = 0;
    for(int i = 0; i < n; ++i)
        s += x[i];
    return s;
}
int main()
{   int a[5] = {8,9,3,2,0};
    int s = Sum(a, 5);
    printf("%d", s);
    return 0;
}
```



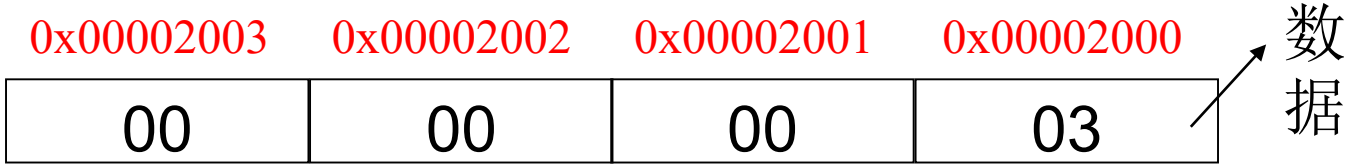
关于课件中内存单元示意图的说明

1个字节（1byte，8bit）

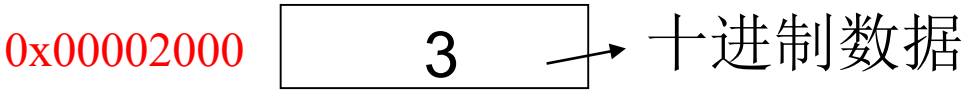
4个字节（4byte， 32bit）



有时压缩成：



或：



（一般用十进制或十六进制描述数据）

每一个内存单元都有一个地址

（一般用十六进制数描述地址，开头的00可以省略）

概述

- **地址：特殊的整数**

  - 计算机内存可看作由一系列内存单元组成，内存单元中可以存储不同的内容。每个内存单元（容量为1个字节）由一个特殊的整数（即地址）进行标识。

  - 不是所有内存单元的地址都能在任一程序中使用，一般地，一个程序只能使用执行环境在编译和执行期间分配给该程序的内存单元的地址；
  - 地址能参与的操作很有限。

  - C语言用指针类型描述地址，通过对指针类型数据的相关操作，可以实现与地址有关的数据访问功能。
    - 指针类型数据通常占用1个字空间（与int型数据占用相等大小的空间）
    - 值集、操作集与 int 不同

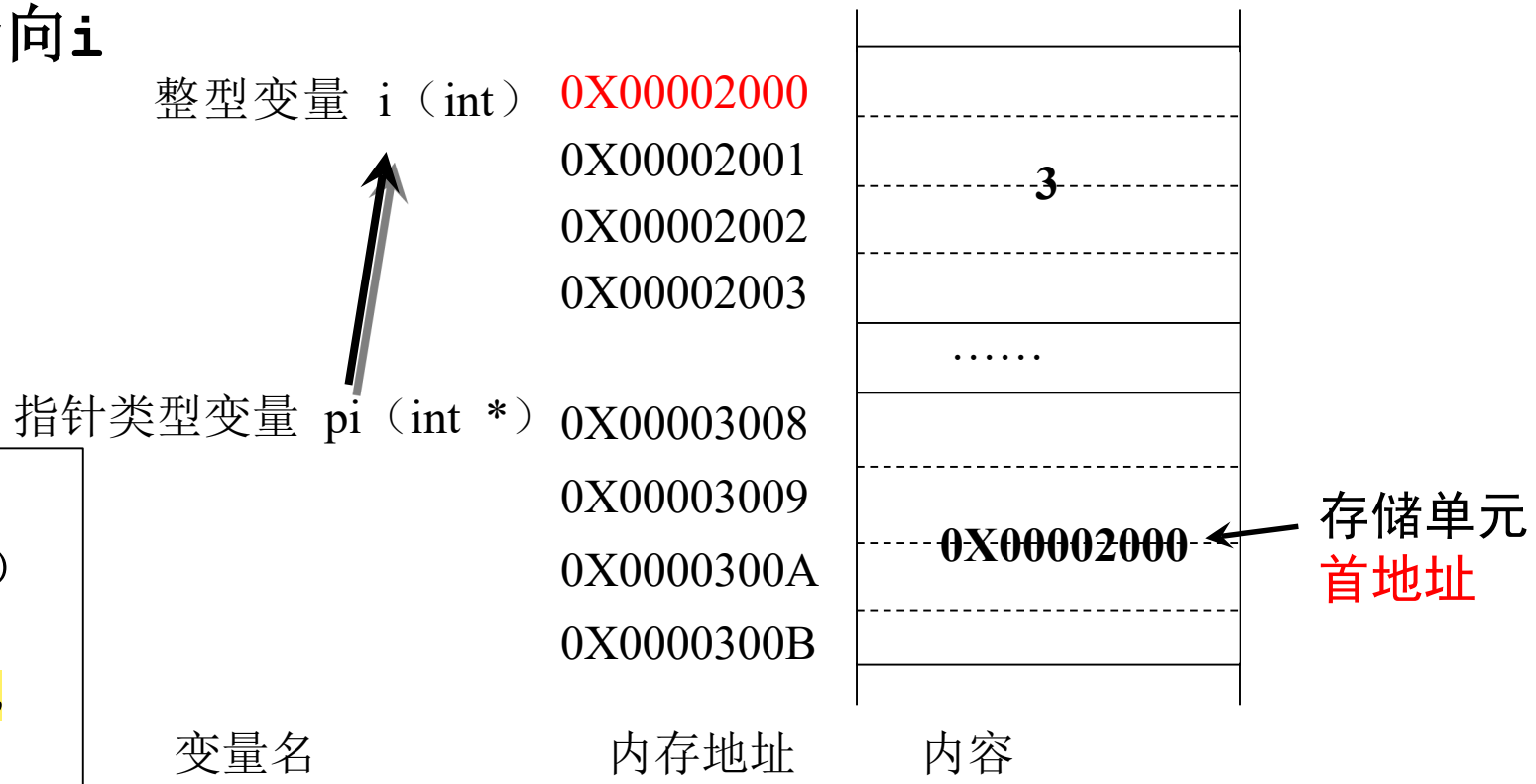
- 变量的地址与变量名之间存在映射关系，一般情况下，高级语言程序通过这种映射关系用变量名访问数据。

- 如果在程序中知道其他函数中变量的地址，则可以省略数据传输环节，从而提高数据的访问效率。
- 在没有变量名的情况下，这种访问方式更具意义。

C语言用指针类型变量表示和存储另一变量的地址。

```
int  i = 3;
int  *pi = &i;  //pi指向i
```

基类型



取地址操作

用地址操作符（Address Operator）

& 获取操作数的地址。

& 是一个单目操作符，优先级较高，结合性为自右向左。

其操作数应为各种类型的内存变量、数组元素等，不能是表达式、字面常量、寄存器变量。

指针（pointer）

- “指针”的含义未被严格界定，可能是指“地址”，也可能是指“指针类型”或“指针类型变量”

指针类型的构造

- 指针类型由一个表示变量类型（在这里又叫基类型）的关键字和一个*构造而成。
- 可以给构造好的指针类型取一个别名，作为指针类型标识符。
- 比如，

```
typedef int *Pointer;
// Pointer是类型标识符
//其值集为本程序中 int 型变量的地址
```

指针变量的定义

- 可以用构造好的指针类型来定义指针变量。
  - 比如，

```
typedef int *Pointer;
Pointer p1, p2;   //定义了两个同类型的指针变量p1、p2
```

- 也可以在构造指针类型的同时直接定义指针变量。
  - 比如，

```
int *p;
//int和*构造了一个指针类型，同时定义了一个指针变量p
//指针变量的变量名不包括*
```

- 多个定义有时可以合并写（只共用基类型），但不主张合并写。
  - 比如，

```
int *p1, *p2; //p2前的*不能省略
int m = 3, n = 5, *px;     或     int *px, m = 3, n = 5;
```

指针变量的初始化

- 指针变量实际存储的是哪一个内存单元的地址，可以通过初始化来指定。
- 用来初始化指针变量的变量要预先定义，且类型与指针变量的基类型要一致，初始化后称指针变量指向该变量。
- 指针变量的初始化通常是在函数调用过程中完成的，即实际参数的地址值传递给形式参数的指针变量。下面先用简单的直接初始化 例子 示意这一过程中的注意事项。
  - 比如，

```
int i = 0;
int *pi = &i;     //取出变量i的地址，用来初始化指针变量pi，即pi指向i
```

  - 而

```
double f = 3.2;
double *pf = &f;        //不可以 double *pf = &i;
```

- **也可以用另一个指针变量或指针常量（基类型要一致）来初始化一个指针变量。**
  - 比如，

```
int i = 0;
int *pi = &i;
int *pj = pi;    //pj也指向变量i
```

  - 又比如，

```
int a[10];
int *pa = a;     //pa指向数组a
```

- **还可以用0来初始化一个指针变量，表示该指针变量暂时不指向任何变量。**
  - 比如，

```
int *pv = 0;     // 这里的 0 是一个空地址，通常用 NULL 代替
int *pv = NULL; // NULL 是系统定义的符号常量（一般在头文件stdio.h中）
```

- **不可以将一个非0整数赋给一个指针变量，因为程序员指定的地址不一定是系统分配给该程序的内存单元地址，不一定能在该程序中访问其对应的空间。**
  - 比如，

```
int *pn = 0x2000;
//编译器不一定会报错，但执行时可能会引起系统故障
```

指针类型相关的基本操作

- **取值操作**
- 赋值操作
- 关系/逻辑操作
- 加/减一个整数
- 减法操作
- 下标操作

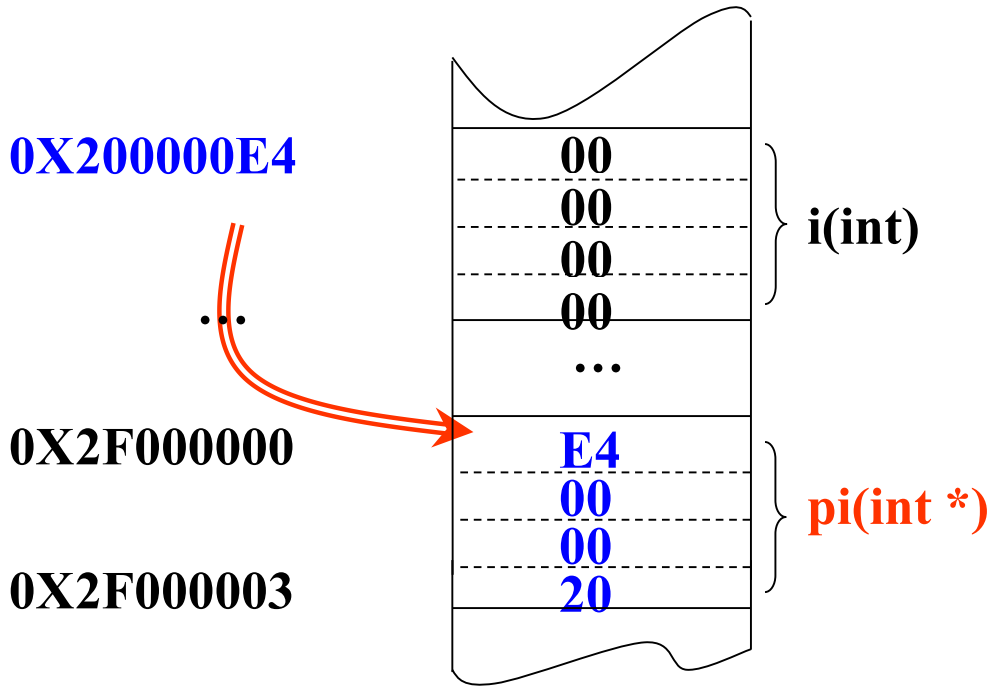
- **这些操作只在一定的约束条件下才有意义。不能参与的操作是没有意义的操作，不一定会出语法错误，但可能会造成难以预料的结果（例如，将两个指针类型数据相加或乘/除法运算，结果不一定是有效的内存地址，或者即使地址有效，其对应的内存单元未必能被该程序访问）。**

- **取值操作**
  - 用指针操作符（Indirection Operator）* 获得指针变量指向的内存数据。
  - * 是一个单目操作符，优先级较高，结合性为自右向左。
  - 其操作数应为地址类型的数据。
  - 注意，构造指针类型或定义指针变量时的 * 不是指针操作符。

- **取值操作与取地址操作是一对逆操作。**

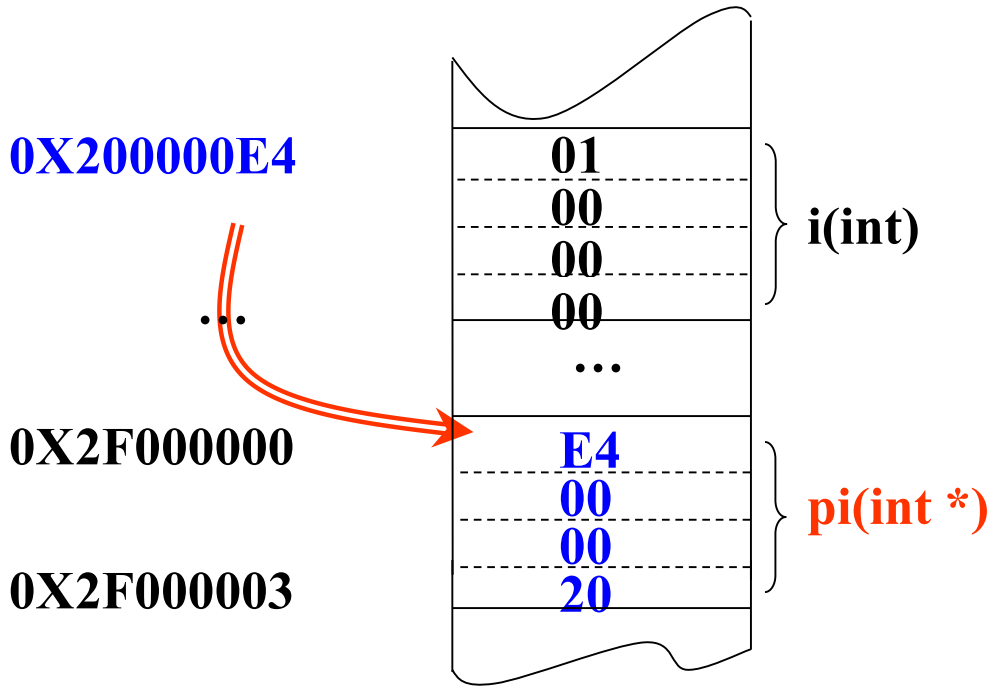
- 比如，

```
int i = 0;
int *pi = &i;     //pi指向i，这里的 * 不是指针操作符
```



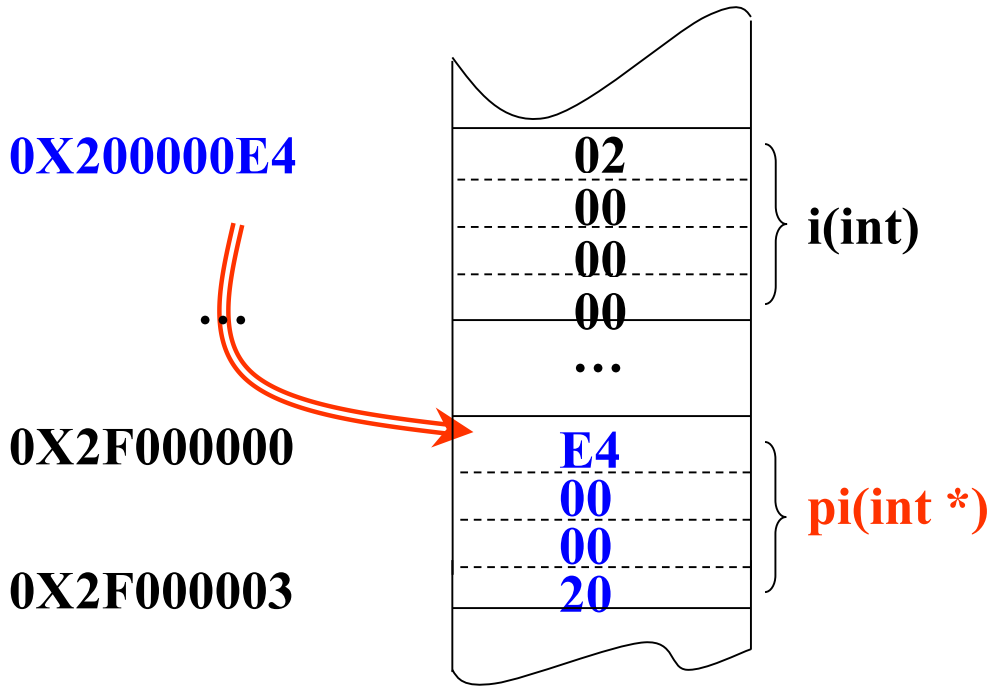
- 比如，

```
int i = 0;
int *pi = &i;     //pi指向i，这里的 * 不是指针操作符

*pi = 1;          //对pi进行取值操作，然后赋值，相当于 i = 1;
```



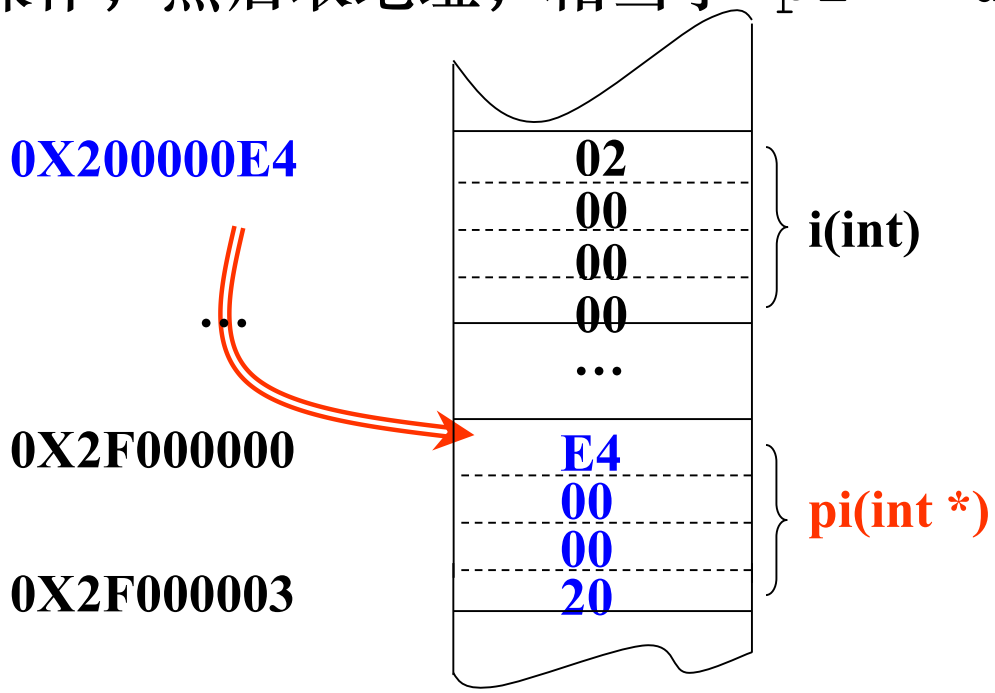
- 比如，

```
int i = 0;
int *pi = &i;      //pi指向i，这里的 * 不是指针操作符

*pi = 1;           //对pi进行取值操作，然后赋值，相当于 i = 1;
*(&i) = 2;         //对&i进行取值操作，然后赋值，相当于 i = 2;
```



- 比如，

```
int i = 0;
int *pi = &i;      //pi指向i，这里的 * 不是指针操作符

*pi = 1;           //对pi进行取值操作，然后赋值，相当于 i = 1;
*(&i) = 2;         //对&i进行取值操作，然后赋值，相当于 i = 2;
pi = &(*pi);       //对pi进行取值操作，然后取地址，相当于 pi = &i;
```



指针类型数据的赋值操作

- 指针变量除初始化外，还可以通过赋值操作指定一个地址，使之指向某一个变量。
  - 比如，

```
int i = 0;
int *pi = &i; //初始化，pi指向i
……
int x = 3;
……
pi = &x; //赋值，pi指向x
*pi = 2; //改变x的值
```

- 对指针变量进行赋值操作的右操作数

  也应为地址类型的数据，
  还可以为NULL（0），
  不可以是非0整数，

  右操作数中的变量要预先定义，其类型与左边指针变量的基类型要一致，
  若是指针变量或指针常量，其基类型要与左边指针变量的基类型一致，

  必要时可小心使用强制类型转换。

- **指针变量必须先初始化或赋值，然后才能进行其他操作，否则其所存储的地址是不确定的，对它的操作会引起不确定的错误。**

```
int *p;
p++;   ?
```

p没有初始化或赋值就使用，警告

```
int i;
int b[3];
int *pi, *pb, *pj, *pv;
pi = &i;
pb = b;
pj = pi;
pv = NULL;
```

将首地址赋给指针变量

- **初始化或赋值之后的指针变量，可以用它通过取值操作来访问数据**
  - 指针变量的值一般是某个数据的地址，用指针变量通过取值操作访问数据时，地址决定访问的起点，基类型决定访问的终点（通过指定起点和终点即可访问数据）

指针类型数据的加/减一个整数操作

- 一个指针类型的数据加/减一个整数，可以使其成为另一个地址，前提是操作结果仍然是一个有效的内存单元地址。比如，操作前指针变量指向某数组的一个元素，操作后的结果指向该数组的另一个元素（不能超出数组范围）。

- 注意，加/减一个整数n后的结果并非在原来地址值的基础上加/减n，而是n的若干倍，具体倍数由基类型决定，即加/减n*sizeof(基类型)。
  - 比如，

```
int i = 0;
int *pi = &i;
pi++;
            //设int型数据占4个字节空间，
            //则pi的地址值实际增加了4，而不是1
```

两个相同类型的指针类型数据的相减操作

- 两个相同类型的指针类型数据的相减操作，结果为两个地址之间可存储基类型数据的个数。通常用来计算某数组两个元素之间的偏移量。
  - 比如，

```
int *pi, *pj;
……
int offset = pj - pi;
//offset为pj与pi之间可存储int型数据的个数
```

指针类型数据的关系/逻辑操作

- 两个指针类型的数据可以进行比较，以判断在内存的位置前后关系，前提是它们表示同一组数据的地址。
- 比如，
  - 两个指针变量都指向某数组中的元素，用关系操作比较这两个地址在数组中的前后位置关系。
  - 也可以判断一个地址是否为NULL，以明确该地址是否为某个实际内存单元的地址。

指针的输出

```
int i = 1;
int *p = &i;
printf("%d \n", p);      //输出p的值(i的地址) %x
printf("%d \n", *p);     //输出i的值
```

eg. 0x22ff40

```
int *p;
int i = 1;
p = &i;
printf("%d \n", p);      //输出p的值(i的地址) %x
```

eg. 0x22ff44

用指针操纵数组

- **指向一位数组元素的指针变量**
- 二级指针
- 数组的指针

指向一维数组元素的指针变量

- 由于一维数组名表示第一个元素的地址，所以可将一维数组名赋给一个一级指针变量，此时称该指针变量指向这个数组的元素。
  - 比如，

```
int a[5];
int *pa = a;      //相当于int *pa = &a[0];
```

  - 当一个指针变量指向一个数组的元素时，该指针变量可以存储数组的任何一个元素的地址，即pa先存储a[0]的地址，不妨设为0x00002000(简作2000)，然后，pa的值可以变化为2004，2008，200C，2010，于是可以通过pa来访问数组a的各个元素。

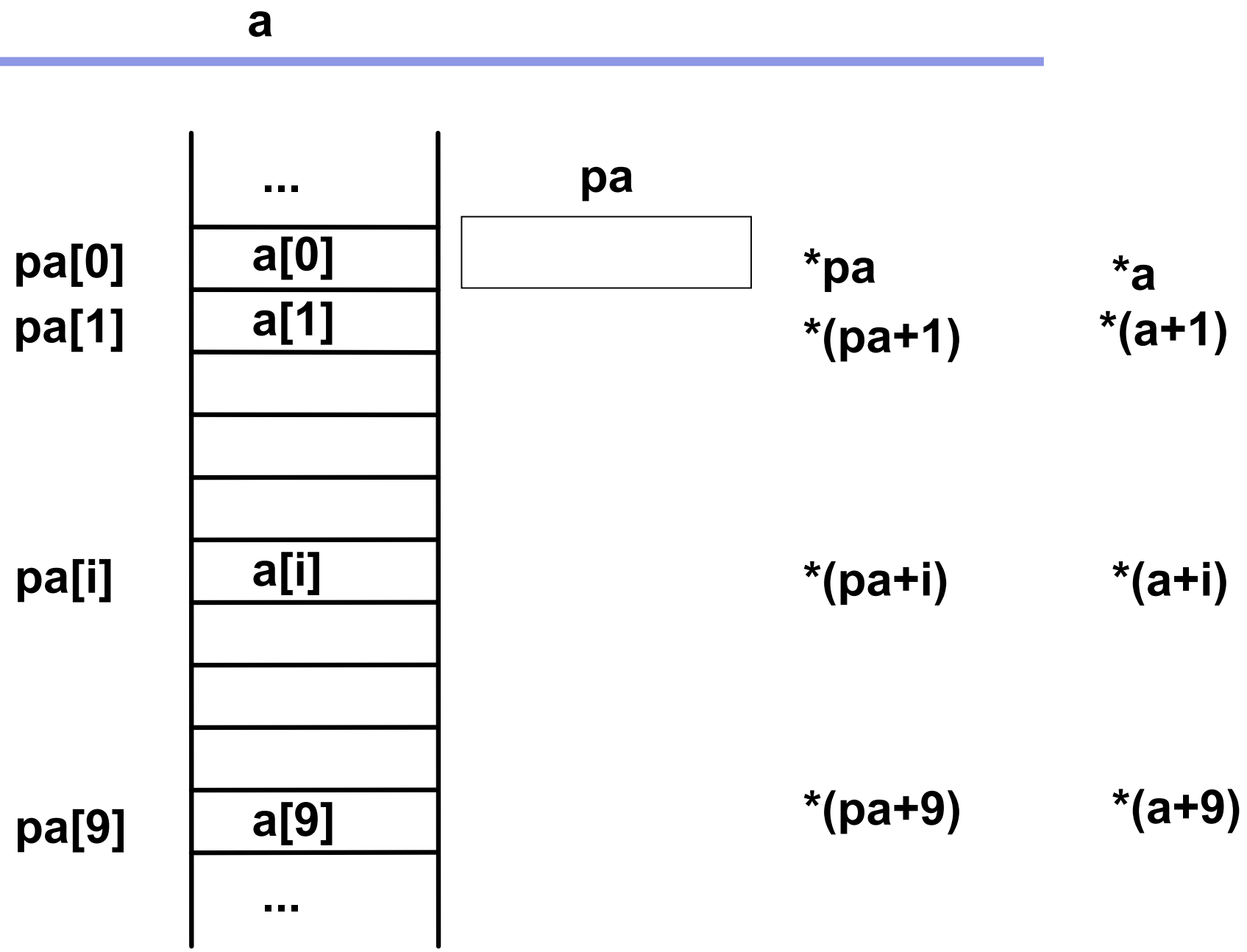
- **访问方法有三种：**
  - 下标法：通过下标操作指定元素；
  - 指针移动法：通过自加/减一个整数操作指定元素地址；
  - 偏移量法：通过取值操作和加/减一个整数操作指定元素。

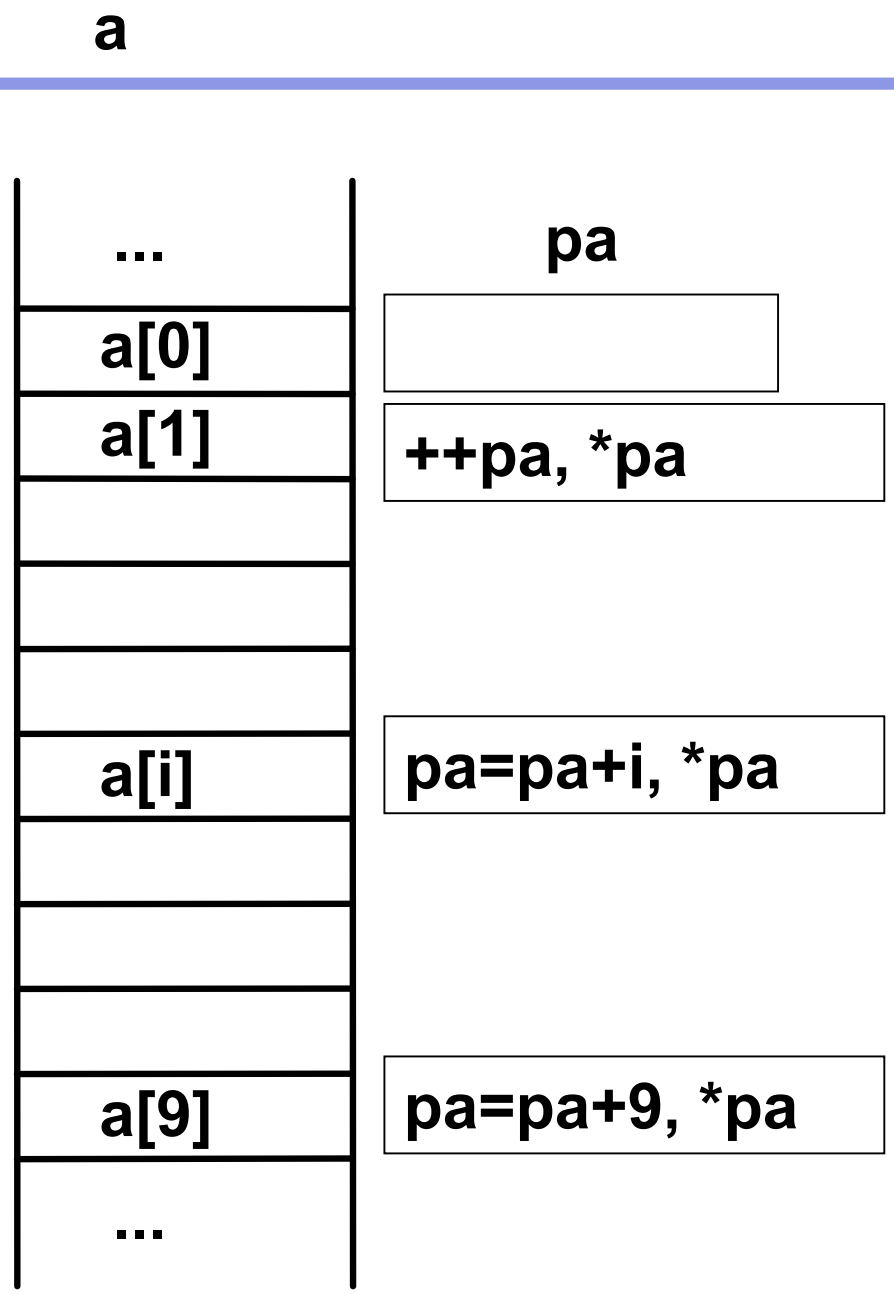
- **通过指针变量操纵数组时，要注意防止下标越界。**

- 例1 用指针变量操纵数组。

…

```
#define N 10
int main( )
{ int i;
  int a[N] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
  int *pa;
  for(pa = a, i = 0; i < N; ++i)
      printf("%d ", pa[i]);    //下标法，用a[i]也行
  for( ; pa < (a + N); ++pa)   //指针移动法，用++a行不行?
      printf("%d ", *pa);
  for(pa = a, i = 0; i < N; ++i)//这里没有“pa = a, ”，结果会怎样?
      printf("%d ", *(pa + i));//偏移量法，用*(a + i)也行
  return 0; }
```


a
...
pa
pa[0]
a[0]
*pa
*a
pa[1]
a[1]
*(pa+1)
*(a+1)
pa[i]
a[i]
*(pa+i)
*(a+i)
pa[9]
a[9]
*(pa+9)
*(a+9)
...


a
...
a[0]
a[1]
a[i]
a[9]
...
pa
++pa, *pa
pa=pa+i, *pa
pa=pa+9, *pa

指向数组元素的指针变量 vs. 数组名

- 数组名可以代表第一个元素的地址，是地址常量
  - 数组名的值不能被修改

- 指向数组元素的指针变量
  - 用来存放一个数组各个元素的地址
  - 一般情况下，这个变量的初始值为数组名代表的地址
  - 经过操作，其值可能会是其他元素的地址

指针及其运用

- 指针的基本概念
- 用指针操纵数组
- 用指针在函数间传递数据
  - 指针类型参数
  - const的作用
  - 引用类型参数
  - 指针类型返回值
- 用指针访问动态变量
  - 通用指针与void类型
  - 动态变量的创建、访问和撤销
  - 内存泄漏与悬浮指针
- 用指针操纵函数*

指针型参数

重点

- 指针变量可以用作函数的形参，以提高函数间大量数据（比如数组）的传递效率。

例2 指针变量作为函数形参。

…

```
#define N 10
int Fun(int *pa, int n);
int main( )
{ int a[N] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
  int sum = Fun(a, N);    // int *pa = a;
  printf("%d \n", sum);
  return 0;
}
```

```
int Fun(int pa[], int n
```

```
 int Fun(int *pa, int n)
{    int s = 0;
     for(int i = 0; i < n; ++i)
          s += *(pa + i);
     return s;
}
```

```
s += pa[i];
```

- 可见，当函数的形参定义成指针变量，实参是数组名时，调用时不是将实参的副本通过赋值的形式一一传递给形参，而是通过指针变量直接访问实参，以提高数据的 传递效率。这种函数调用方式通常叫做传址调用。

- 实际上，C语言中，写成数组定义形式的形参也是按指针类型数据处理的，即只给形参分配存放一个字的内存空间，以存储实参第一个元素的地址。

函数的副作用

- 指针类型数据用作函数的形参，除了可以提高函数间大量数据的正向传递效率，还存在一种**函数的副作用**，即在被调函数中可以通过指针变量修改实参的值。

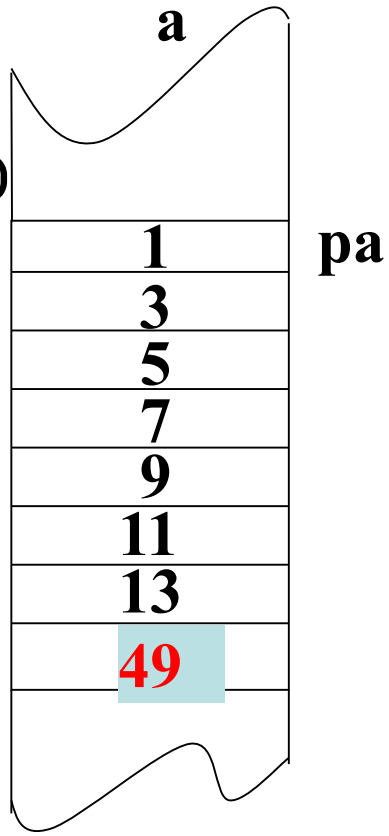
- 被调函数的return语句一般只能返回一个值，如果需要返回多个值，则可以利用这种函数的副作用实现函数间数据的反向传递。

例3 指针变量作为函数形参的双重作用。

…

```
#define N 8
void Fun(int *pa, int n);
int main( )
{ int a[N] = {1, 3, 5, 7, 9, 11, 13};//注意a[N-1]初始化为0
  Fun(a, N);    // int *pa = a
  printf("%d \n", a[N-1]);
  return 0;
}
void Fun(int *pa, int n)
{ for(int i = 0; i < n-1; ++i)
      *(pa + n - 1) += *(pa + i);
}//既利用传址调用提高了数据正向传递的效率
//又利用函数的副作用实现了数据的反向传递(修改了数组最后一个元素的值)
```

int *pa = a



- **例4 利用函数的副作用实现两个数据的交换。**

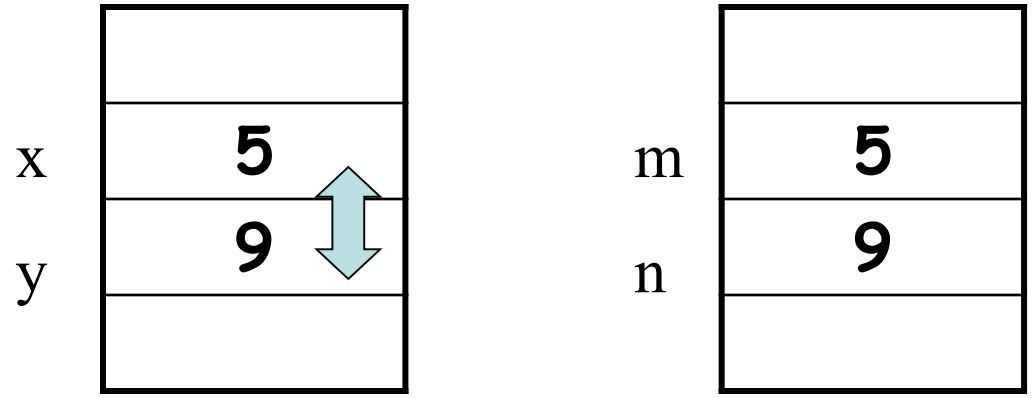
```
…
void Swap(int *pm, int *pn);
int main( )
{ int m = 5, n = 9;
  Swap(&m, &n);// int *pm = &m, int *pn = &n, 传址调用
  printf("%d, %d", m, n);
  return 0;
}
void Swap(int *pm, int *pn)
{ int temp = *pm;
  *pm = *pn;
  *pn = temp;
}
```

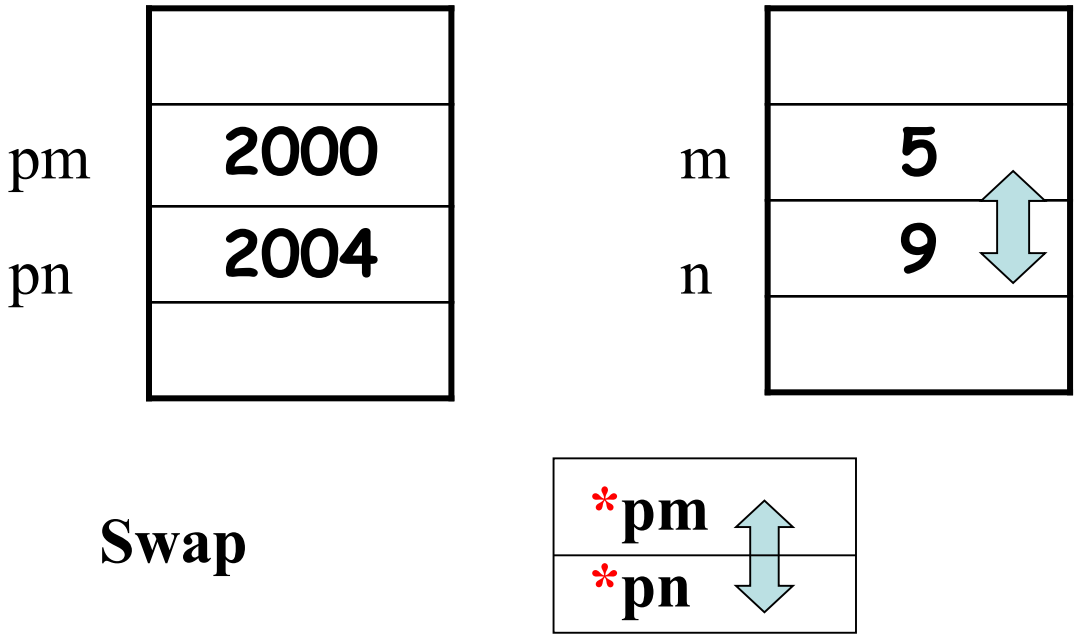
	形参	实参	特点	举例

传值调用	变量	常量的值
		变量的值
		表达式的值
			形参的改变不影响实参	`void Swap1(int x, int y)`
				`{`
				`  int temp = x;`
				`  x = y;`
				`  y = temp;`
				`}`
				`int main( )`
				`{`
				`  int m = 5, n = 9;`
				`  Swap1(m, n);`
				`  printf("%d, %d", m, n);`
				`  return 0;`
				`}`
				`int x=m`
				`int y=n`

传址调用	指针	地址值	改变形参所指向的变量值来影响实参	`void Swap(int *pm, int *pn)`
				`{`
				`  int temp = *pm;`
				`  *pm = *pn;`
				`  *pn = temp;`
				`}`
				`int main( )`
				`{`
				`  int m = 5, n = 9;`
				`  Swap(&m, &n);`
				`  printf("%d, %d", m, n);`
				`  return 0;`
				`}`
				`int *pm=&m`
				`int *pn=&n`

Swap1



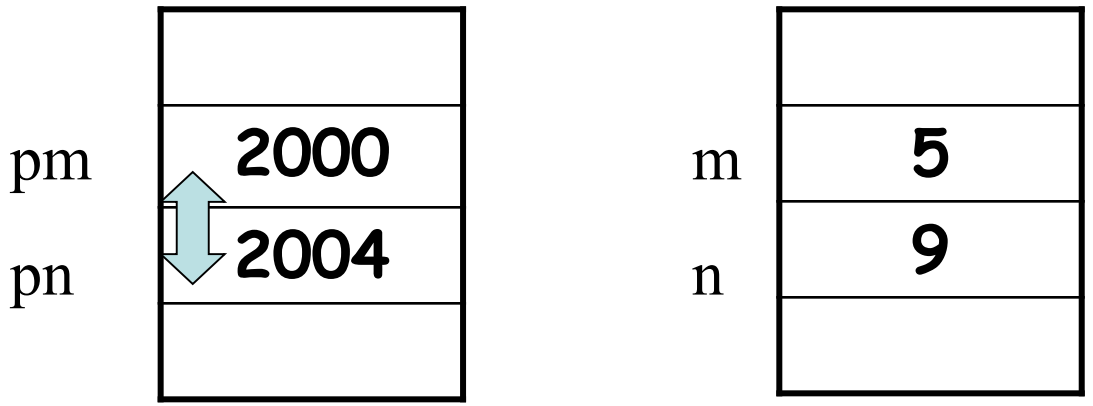


Swap


```
void Swap2(int *pm, int *pn)
{
  int *temp = pm;
  pm = pn;
  pn = temp;
}
```

```
int main( )
{
  int m = 5, n = 9;
  Swap2(&m, &n);
  printf("%d, %d", m, n);
  return 0;
}
```

```
int *pm=&m
int *pn=&n
```



Swap2

const的作用

- 指针变量作为形参一般可以产生提高数据正向传递效率与函数的副作用两个效果。
- 如果不希望函数的副作用起效，用关键字**const**将形参设置成指向常量的指针类型。
  - 比如，

```
void F(const int *p, int num)   // 或const int p[]
{
  ......
  *p = 1;     // 不允许该操作，同样，“p[i] = 1;”也是不允许的
  ......
}   //该程序通过const限制程序员在被调函数中修改实参的值
```

- **const还可以用来定义一个常量。**
  - 比如，

```
const double PI = 3.1415926;
```

- **这种常量的定义形式往往比“#define PI 3.14159”更好，因为系统会对其进行类型检查，而不是只进行文本替换。**

- **值得注意的是**
  - 用const定义常量时，必须初始化
  - const的不同位置含义不同

- 例如：

```
int n;
const int M = 0;  //M为常量
const int *p1;    //*p1是常量，可以p1 = &M，基类型相同
                  //可以p1 = &n，只不过不能通过p1修改n的值
int * const P2 = &n;   //P2是常量，*P2的值可以改变
        //不可以int * const P2 = &M，以防止通过P2修改M的值
const int * const P3 = &M;  //*P3、P3都是常量
        //可以const int * const P3 = &n，只不过不能通过P3修改n的值
n++;
M++;
p1++;
P2++;
P3++;
```

- 又例如：

```
int n;
const int M = 0;  //M为常量
const int *p1;    //*p1是常量，可以p1 = &M，基类型相同
                  //可以p1 = &n，只不过不能通过p1修改n的值
int * const P2 = &n;   //P2是常量，*P2的值可以改变
        //不可以int * const P2 = &M，以防止通过P2修改M的值
const int * const P3 = &M;  //*P3、P3都是常量
        //可以const int * const P3 = &n，只不过不能通过P3修改n的值

int *q = &n;
*q = 1;
q++;
q = &n;
q = &M;
```

函数间有多种通讯方式

- **传值方式(把实参的副本复制给形参)**
- **利用函数返回值传递数据**

- **通过全局变量传递数据**
  (函数副作用问题，尽量不用)

- **通过指针类型参数传递数据**
  (函数副作用问题，可以通过指向常量的指针来避免)

C++：引用类型参数

重点

注意：Typedef 是给类型名取一个别名

- 什么是引用
  - 引用类型用于给一个变量取一个别名。
    - 例如：

```
int x = 0;
int &y = x; //y为引用类型的变量
cout << x << ',' << y << endl; //结果为: 0,0
y = 1;
cout << x << ',' << y << endl; //结果为: 1,1
x = 2;
cout << x << ',' << y << endl; //结果为: 2,2
```

在语法上，对引用类型变量的访问与非引用类型相同；
但在语义上，对引用类型变量的访问实际访问的是另一个变量（被引用的变量），
通常用来定义形式参数。

引用类型的变量定义与初始化

- &

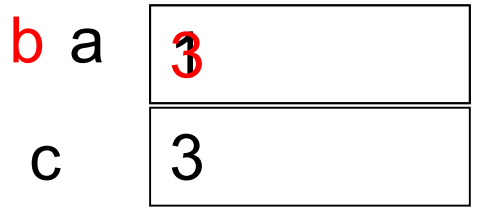
```
int x;
int &y = x;
```

- 注意：
  - 定义引用变量时必须要有初始化，并且引用变量的基类型和被引用变量的类型相同
  - 引用类型的变量定义之后，它不能再引用其他变量。（可以编译执行，但含义不是引用其他变量）

引用与指针的对比

```
int a=1, c=3;
int &b = a;      //b引用a
cout << a << ',' << b << ',' << c << endl;
b = c;           //不是b引用c，而是将c的值赋给b
cout << a << ',' << b << ',' << c << endl;
```

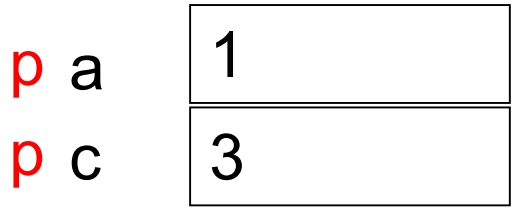
1, 1, 3



3, 3, 3

```
a=1, c=3;
int *p = &a;     //p指向a
cout << a << ',' << *p << ',' << c << endl;
p = &c;          //p指向c
cout << a << ',' << *p << ',' << c << endl;
```

1, 1, 3



1, 3, 3

引用与指针的异同点

- 引用类型与指针类型都可以实现通过一个变量访问另一个变量，但访问的语法形式不同：引用采用变量名直接访问，指针则采用取值操作间接访问。引用类型的访问过程对使用者而言是透明的，因此更安全。

- 在作为函数参数类型时，引用类型参数的实参是一个变量的名字，而指针类型参数的实参是一个变量的地址。引用更方便。除了在定义时指定的被引用变量外，引用类型变量不能再引用其他变量；而指针变量定义后可以指向其他同类型的变量。因此，引用类型比指针类型要安全。

引用类型参数

C++ 有这样的库函数 swap!

- **引用传递：把实参的地址传给相应的形式参数**

```
void Swap(int &x, int &y)
{
    int t = x;
    x = y;
    y = t;
}
int main()
{   int a=5, b=9;
    cout << a << ', ' << b << endl; //结果为: 5, 9
    Swap(a, b);
    cout << a << ', ' << b << endl; //结果为: 9, 5
    return 0;
}
```

```
void MySwap(int *pm, int *pn)
{
      int temp = *pm;
      *pm = *pn;
      *pn = temp;
}
```

```
MySwap(&m, &n);
```

常量引用-防止函数副作用问题

```
void F(const int &x)
{   ......
    x = 1;   //Error
    ......
}

int main()
{   int a;
    F(a);
    return 0;
}
```

函数间有多种通讯方式

- **传值方式(把实参的副本复制给形参)**
- **利用函数返回值传递数据**

- **通过全局变量传递数据**
  (函数副作用问题，尽量不用)

- **通过指针类型参数传递数据**
  (函数副作用问题，可以通过指向常量的指针来避免)

- **通过引用类型参数传递数据（C++）**
  (函数副作用问题，可以通过指向常量的引用来避免)

指针类型返回值

- C语言的return语句一般只能返回一个值，所以不能返回数组类型的数据，但可以返回数组或数组元素的地址。

```
int *Max(int ac[ ], int num)
{
      int max_index = 0;
      for(int i = 1; i < num; ++i)
            if(ac[i] > ac[max_index])
                  max_index = i;
      return &ac[max_index];
}
```

指针类型返回值

- C语言的return语句一般只能返回一个值，所以不能返回数组类型的数据，但可以返回数组或数组元素的地址。

```
int *Max(const int ac[ ], int num)
{
      int max_index = 0;
      for(int i = 1; i < num; ++i)
            if(ac[i] > ac[max_index])
                  max_index = i;
      return (int *)&ac[max_index];
}
```

将const int 型地址转换成 int 型地址

例5 指针类型返回值示例。

```
…
#define N 5
int *Max(const int ac[ ], int num);
int main( )
{
  int a[ ] = {1, 2, 5, 4, 3};
  *Max(a, N) = 0;          // 用0替换数组a中的最大数
  for(int i = 0; i < N; ++i)
     printf("%d \t", a[i]);
  return 0;
}
```

```
int *p = Max(a, N);
*p = 0;
```

- **注意，函数一般不能返回局部变量的地址。**
  - 例如，

```
int *F( )
{
      int m, n;
      scanf("%d%d", &m, &n);
      if(m > n)
            return &m;
      else
            return &n;
}//调用者获得地址时，m、n空间已经释放，地址无意义
```

用指针操纵数组

- 指向一位数组元素的指针变量
- 二级指针
- 数组的指针

二级指针

- C语言的指针变量还可以存储另一个指针变量的地址。
  - 比如，

```
int i = 0;
int *pi = &i;
int **ppi = &pi;  //指针变量ppi存储的是指针变量pi的地址
```

- 这时，称指针变量ppi是一个二级指针变量，它指向的变量是一个一级指针变量pi。
- 二级指针的实质是其基类型表示的是一个数据群体（有起点、有终点）。二级指针变量通常用于指针类型数据的传址调用，也可以用作数组的指针，操纵数组。
- 如果再定义一个指针变量pppi存储ppi的地址，则pppi是一个三级指针变量，以此类推。

数组的指针

- 指向整个数组的指针变量，即数组的指针。
  - 一个指针类型的基类型可以是int、float这样的基本类型，也可以是数组这样的构造类型。比如，

```
typedef int A[10];
A *q;
```

  - 或者合并写成，

```
int (*q)[10];
```

  - 该指针变量q可以存储一个类型为A的数组的地址，比如，

```
int a[10];
q = &a;      //q相当于一个二级指针变量
```

- **对于二维数组，可以通过不同级别的指针变量来操纵。注意，二维数组名表示第一行的地址。**

  - 比如，

```
int b[5][10];
int *p;
p = &b[0][0];    //或“p = b[0];”
//一级指针变量可以存储数组b某一元素的地址
```

  - 然后可以通过指针移动法指定某个元素地址，再取值指定任一元素，比如 ++p，*p;

- 或，

  ```
  int (*q)[10];
  q = &b[0]; //或“q = b;”
  ```

  //q相当于一个二级指针变量，可存储b某一行的地址

- 然后可通过下标法指定元素，比如q[i][j]（相当于b[i][j]）；
- 也可以通过指针移动法指定某一行的地址，再取值指定某一行的首元素，比如++q，**q;
- 还可以通过偏移量法指定任一元素，比如
  - *(*(q+i)+j)（相当于*(*(b+i)+j)）
  - *(&q[0][0]+10*i+j)（相当于*(&b[0][0]+10*i+j)）
  - *(q[i]+j)（相当于*(b[i]+j)）
  - (*(q+i))[j]（相当于(*(b+i))[j]）

- 或

```
int (*r)[5][10];
r = &b;
```

//r可存储整个二维数组b的地址，相当于一个三级指针变量

二维数组的指针*

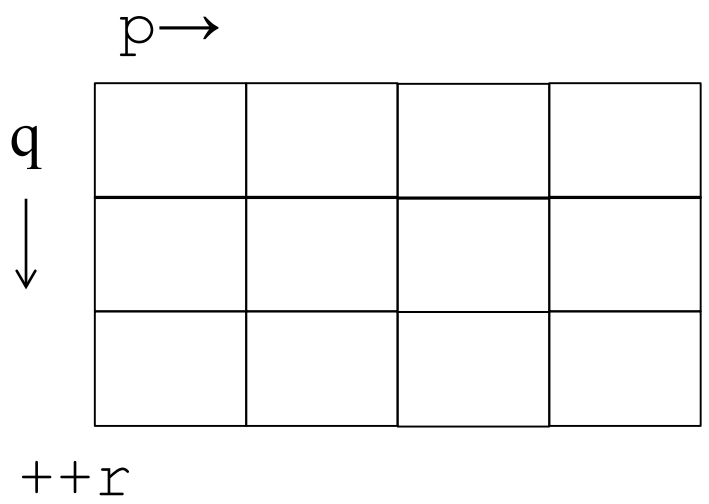
```
int b[5][10];
int *p;
p = &b[0][0];//或“p = b[0];”
```

第一行某个元素 `p[j]`

```
int (*q)[10];
q = &b[0];   //或“q = b;”
```

某个元素 `q[i][j]`

```
int (*r)[5][10];
r = &b;
```



指针及其运用

- 指针的基本概念
- 用指针操纵数组
- **用指针在函数间传递数据**
  - 指针类型参数
  - const的作用
  - 引用类型参数
  - 指针类型返回值
- **用指针访问动态变量**
  - 通用指针与void类型
  - 动态变量的创建、访问和撤销
  - 内存泄漏与悬浮指针
- 用指针操纵函数*

通用指针与void类型

- void：空类型的关键字
- 空类型的值集为空，在计算机中不占空间，一般不能参与基本操作

- 通常用来描述不返回数据的函数的返回值，以及不需要参数的函数的形参

- 还可以作为指针类型的基类型，形成通用指针类型（void *）。

- 通用指针类型变量不指向具体的数据，不能用来访问数据，但任何指针类型都可以隐式转换为通用指针类型，而且在将通用指针类型转换回原来的指针类型时不会丢失信息。

- 通用指针类型经常作为函数的形参和返回值的类型，以提高所定义的函数（比如创建动态变量的库函数）的通用性。

内存分配方式

- **系统为程序中的数据分配内存空间一般有两种方式**
  - 静态：程序开始执行前在静态数据区分配空间
    - 先由编译器对其定义行进行特殊处理（规划所需内存，分析、处理其初始化值），程序执行时，由执行环境在静态数据区为之分配空间，并写入初始化值，若未初始化则写入初值0，此后程序可获取或修改其值，直到整个程序执行结束才收回其空间。
  - 动态：程序执行时在栈区或堆区分配空间
    - 在栈区：定义行有对应的目标代码，通过代码的执行在栈区获得空间，若已初始化则获得初始化值，若未初始化则其初值是内存里原有的值，此后程序可以访问其内存空间，获取或修改其值，一旦复合语句执行结束即收回其内存空间
    - 在堆区：由程序员编写的相关代码（调用malloc库函数）申请内存空间，通过代码的执行在堆区获得空间，由于没有初始化，其初值为内存里原有的值，此后程序可以访问其内存空间，进行赋值操作，以及获取或修改其值，最后通过程序员编写的相关代码（调用free库函数）释放其内存空间。如果程序员忘记编写释放其内存空间的代码，则要等整个程序执行结束时才收回其内存空间

重点

- **动态变量的创建、访问和撤销**

动态变量的创建

- 动态变量是指系统根据程序执行过程中的动态需求，临时在零星的空闲内存中寻找合适的若干单元来存储的一类数据。
- 动态变量没有定义过程，由malloc库函数创建。

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

#include <malloc.h>
```

动态变量的创建

- malloc库函数的原型是

```
void *malloc(unsigned int size);
```

  - 该函数在**stdlib**中声明，其功能是在程序的堆区分配size个单元，并返回该内存空间的首地址。调用时一般需要用操作符sizeof计算需要分配的单元个数作为实参，并对返回值进行强制类型转换，以便存储某具体类型的数据。
  - 比如，

```
(int *)malloc(sizeof(int));
        // 创建一个int型动态变量，可存储1个int型数据
(double *)malloc(sizeof(double) * n);
        // 创建一个double型动态数组，含n个元素
```

动态变量的访问

- 创建的动态变量没有变量名，需要通过指针变量来访问。

动态变量的访问

```
void *malloc(unsigned int size);
```

(1) 一般动态变量的访问

```
int *pd;
pd = (int *)malloc(sizeof(int));
        // 接下来可用 *pd 表示和访问该动态变量
```

```
void *malloc(unsigned int size);
```

(2) 动态数组的访问

```
int n;
scanf("%d", &n);   //假设输入的 n 为5
double *pda;
pda = (double *)malloc(sizeof(double) * n);
    //接下来，可用*(pda+3)或pda[3]表示和访问该动态数组中的第4个元素
```

```
pda[0] ~ pda[n-1]
*pda; ++pda, *pda; ……; ++pda, *pda;
*pda ~ *(pda + n-1)
```

动态变量的撤销

- C语言中的动态变量（包括动态数组）在所属的函数执行完后不会自动消亡，需要由free库函数在程序中显式地撤销。
- free库函数的原型是

```
void free(void *p);
```

  - 该函数在stdlib中声明，其功能是释放由malloc函数分配的p所指向的内存空间。
  - 比如，

```
free(pd);          // 撤销pd指向的动态变量
free(pda);         // 撤销pda指向的动态数组
```

```
int *pda = (int *)malloc(sizeof(int) * n * 2);//看成一维数组
```

二维动态数组

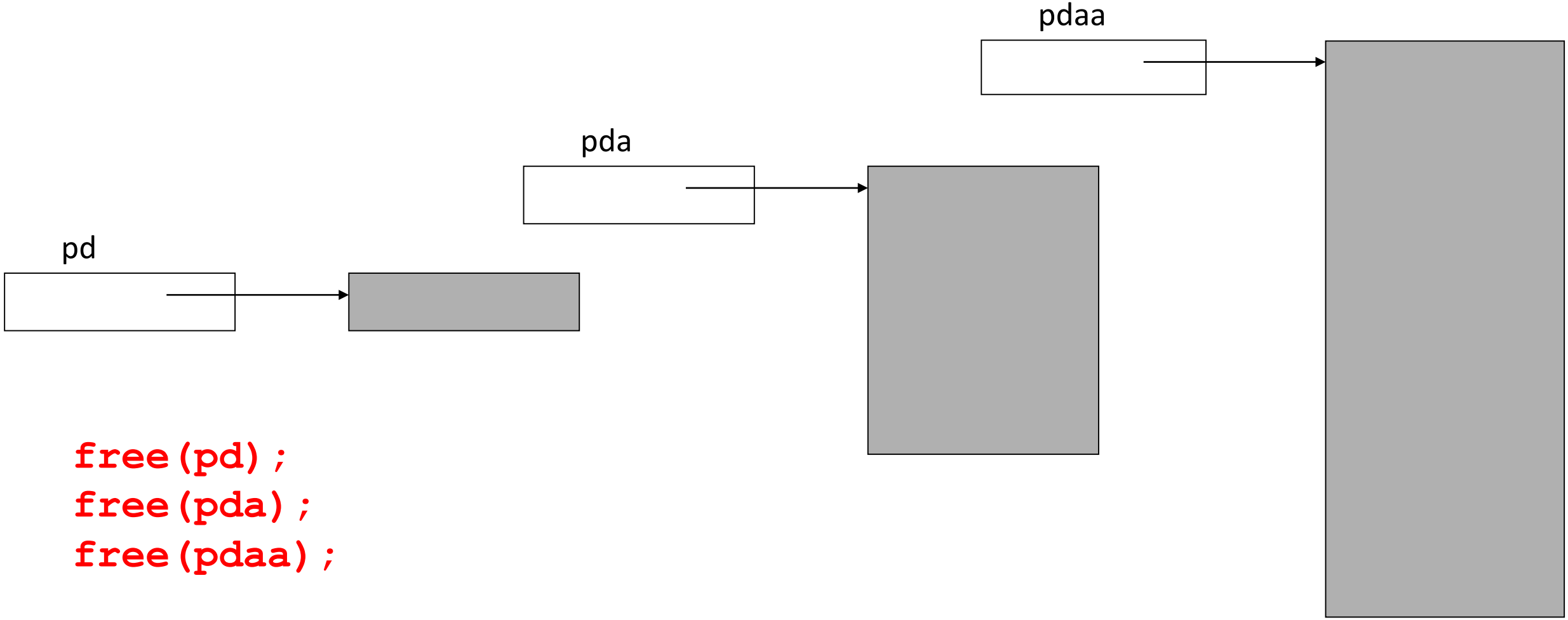
```
int (*pdaa)[2] = (int (*)[2])malloc(sizeof(int) * n * 2);
```

```
typedef int A[2];
A *pdaa;          //pdaa可以指向列数为2的int型二维数组中的一行
pdaa = (A *)malloc(sizeof(int) * n * 2);
                  //返回值强制转换成 A *（即int(*)[2]）类型
```

```
int **pdaa = (int **)malloc(sizeof(int) * n * 2);
                  //创建一个int型n行2列二维动态数组（C语言是弱类型）
```

```
free(pdaa);       //  撤销pdaa指向的二维动态数组
```

- **注意，撤销的是动态变量所在的堆区内存空间，即图中的阴影区域，而不是指向它的指针变量。**




```
free(pd);
free(pda);
free(pdaa);
```

C：创建与撤销动态变量

重点

```
#include <stdlib.h>

int *pd = (int *)malloc(sizeof(int));
double *pda = (double *)malloc(sizeof(double) * n);
int (*pdaa)[10] = (int (*)[10])malloc(sizeof(int) * n * 10);

free(pd);
free(pda);
free(pdaa);
```

C++：创建与撤销动态变量

重点

```
int *pd = new int;
double *pda = new double[n];
int (*pdaa)[10] = new int[n][10];

delete pd;
delete []pda;
delete []pdaa;
```

eg.

```
int *pd = new int;
*pd = 3;
cout << endl << *pd << endl;
```

```
delete pd;
```

eg.

```
int *pda = new int[n];
for(int i=0; i < n; ++i, ++pda)
      cin >> *pda;

pda -= n;
for(int i=0; i < n; ++i, ++pda)
      cout << *pda << ", ";
```

```
for(int i=0; i < n; ++i)
      cin >> pda[i];


for(int i=0; i < n; ++i)
      cout << pda[i] << ", ";
```

```
delete []pda;
```

eg.

```
const int N = 10;
int m;
cin >> m;
int (*pdaa)[N] = new int[m][N];
for(int i=0; i < m; ++i)
      for(int j=0; j < N; ++j)
            cin >> *(*(pdaa+i) + j);

for(int i=0; i < m; ++i)
{     for(int j=0; j < N; ++j)
            cout << *(*(pdaa+i) + j) << "   ";
      cout << endl;
}
```

```
for(int i=0; i < m; ++i)
      for(int j=0; j < N; ++j)
            cin >> pdaa[i][j];


for(int i=0; i < m; ++i)
{     for(int j=0; j < N; ++j)
            cout << pdaa[i][j] << "
      cout << endl;
}
```

```
delete []pdaa;
```

内存泄露*

难点

- 对于动态变量，如果没有显式地撤销，那么当指向它的指针变量的生存期结束后（比如其所属函数执行完毕但整个程序尚未执行完毕时），或者指向它的指针变量指向了别处，则该动态变量仍然存在，但却无法访问，从而造成内存空间的浪费，这一现象称作“内存泄露”，即上图中阴影区域的内存空间“泄露”了。
  - 比如，

```
int *pda;
int m, n;
scanf("%d", &n);  //假设输入的 n 为5
pda = (int *)malloc(sizeof(int) * n);  //pda指向动态数组
pda = &m;  //     pda指向m，上面的动态数组造成内存泄露
```

悬浮指针*

难点

- 动态变量在用free库函数撤销后，指向它的指针变量则指向一个无效空间，这时该指针变量变为“悬浮指针”（dangling pointer）。即前图中阴影区域的内存空间释放后，指针变量p则变为“悬浮指针”
  - 比如，

```
int *pda;
int n;
scanf("%d", &n);  //假设输入的 n 为5
pda = (int *)malloc(sizeof(int) * n); // pda指向动态数组
free(pda);
// pda变为“悬浮指针”，不能通过pda访问数据，比如不可*pda = 0
```

内存泄露与悬浮指针*

难点

```
int *pda;
int m;
pda = (int *)malloc(sizeof(int) * n);
……

pda = &m;
```

pda所指向的动态空间没有释放，但无法访问，泄漏了

pda所指向的动态空间释放了，不知道会分配给谁，
但pda 里存储的还是该动态空间的首地址

```
int *pda;
pda = (int *)malloc(sizeof(int) * n);
……
free(pda);
```

- **实际应用**

1）对输入的10个整数进行排序，可以用数组来实现：

```
const int N = 10;
int i, a[N];

for(i = 0; i < N; ++i)
   scanf("%d", &a[i]); //cin >> a[i];

Sort(a, N);
```

…

2）对输入的若干个整数进行排序（先输入整数的个数n，后输入n个整数），可以：

```
int n, i;
scanf("%d", &n); //cin >> n;
int a[n];

for(i = 0; i < n; ++i)
   scanf("%d", &a[i]); //cin >> a[i];

Sort(a, n);
```

有些新标准，
允许数组长度为变量，
此法适用于支持这类新标准的编译器。

…

2）对输入的若干个整数进行排序（先输入整数的个数n，后输入n个整数），可以用动态数组来实现：

老标准和有些新标准，
不允许数组长度为变量，
此法适用于支持这类标准的编译器。

```
int n, i;
int *pda;
scanf("%d", &n); //cin >> n;
pda = (int *)malloc(sizeof(int) * n);    // pda = new int[n];
for(i = 0; i < n; ++i)
  scanf("%d", &pda[i]); //cin >> pda[i];

Sort(pda, n);
…
free(pda);//delete []pda;
```

3）对输入的若干个正整数进行排序（先输入各个正整数，最后输入一个结束标志 -1）

```
?      //用不断调整动态数组的大小来实现

Sort(pda, count);
…      //输出排序后的数组
free(pda); //delete []pda;
```

用不断调整动态数组的大小来实现

```
const int INC = 5;
int max_len = 10, count = 0, m;
int *pda = (int *)malloc(sizeof(int) * max_len);
          //int *pda = new int[max_len];
scanf("%d", &m);
//cin >> m;
while(m != -1)
{
  ……


  pda[count] = m;
  ++count;
  scanf("%d", &m); //cin >> m;
}
```

```
if(count >= max_len)
{  max_len += INC;  //扩容
   int *q = (int *)malloc(sizeof(int) * max_len);
     //int *q = new int[max_len];
   for(int i = 0; i < count; ++i)
     q[i] = pda[i];
   free(pda);          //delete []pda;
   pda = q;
   q = NULL;
}
```

用指针操纵函数*

难点

- C程序运行期间，程序中每个函数的目标代码也占据一定的内存空间。C语言允许将该内存空间的首地址赋给函数指针类型变量（简称函数指针，注意与指针类型返回值的区别），然后通过函数指针来调用函数。

  - 比如，

```
typedef int (*PFUNC)(int);      // 构造了一个函数指针类型
PFUNC pf;  // 定义了一个函数指针
```

  - 也可以在构造函数指针类型的同时直接定义函数指针：

```
int (*pf)(int);
//第一个int为函数的返回值类型，第二个int为参数的类型。
```

- 对于一个函数，比如int F(int m) { ... }，可以用取地址操作符&（或直接用函数名）来获得其内存地址。
  - 比如，

    ```
    pf = &F;
    ```
  - 或

    ```
    pf = F;
    ```
  - 这样，函数指针pf指向内存的代码区（而不是数据区），接下来可以通过函数指针调用函数：

    ```
    (*pf)(10);
    ```
  - 或

    ```
    pf(10);     //    实参为10，调用函数F
    ```

- 例* 根据输入的要求，执行在函数表中定义的某个函数。

```
…
#include <cmath>
typedef double (*PF)(double);
PF func_list[8] = {sin, cos, tan, asin, acos, atan, log, log10};
int main( )
{ int index;
  double x;
  do
  {   printf("请输入要计算的函数(0:sin 1:cos 2:tan 3:asin 4:acos 5:atan 6:log 7:log10):\n");
      scanf("%d", &index);
  }while(index < 0 || index > 7);
  printf("请输入参数: ");
  scanf("%lf", &x);
  printf("结果为: %f \n", (*func_list[index])(x));
```

```
double Integrate(double (*pfun)(double x), double x1, double x2)
```

- 函数Integrate可以计算任意一个一元可积函数（由函数指针pfun操纵）在一个区间[x1, x2]上的定积分，该函数的调用形式：

```
double My_func(double x)
{   double f = x;
    return f;
}
```

- **Integrate(My_func, 1, 10);**
  //计算函数My_func在区间[1, 10]上的定积分
- **Integrate(sin, 0, 1);**
  //计算函数sin在区间[0, 1]上的定积分
- **Integrate(cos, 1, 2);**
  //计算函数cos在区间[1, 2]上的定积分

Integrate(cos, 1, 2);

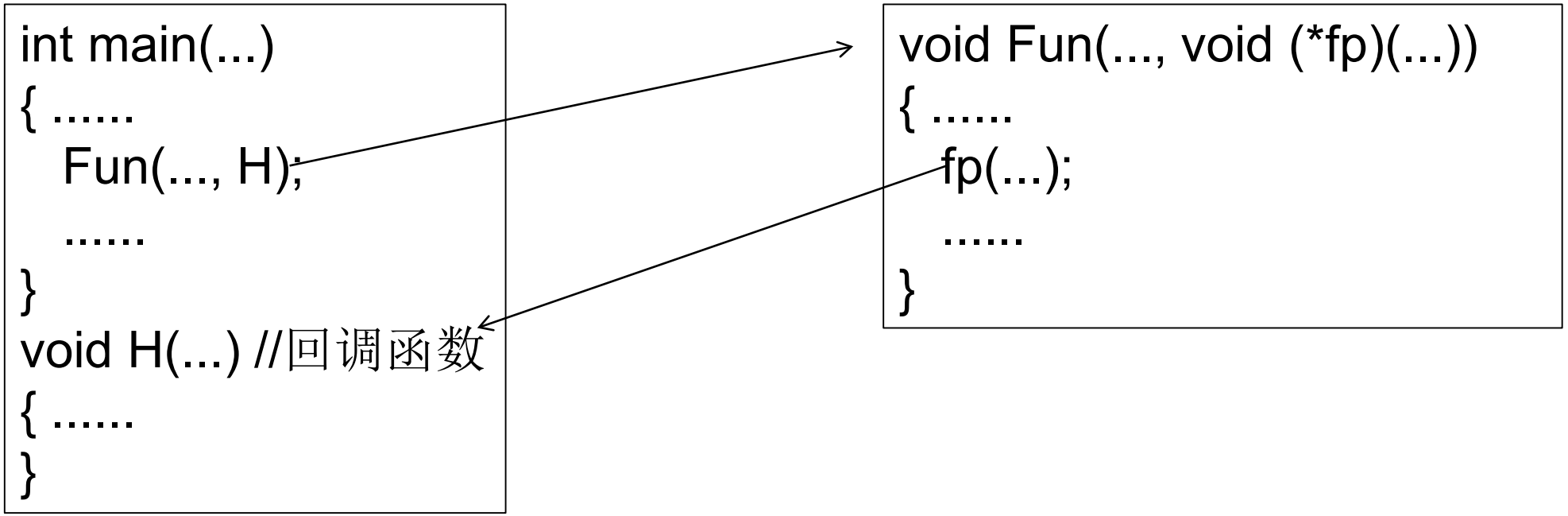
- 即可以把一个函数作为参数传给被调用函数，被调用函数的形参定义为一个函数指针类型，调用时的实参为一个函数的地址。
  - 比如，

```
double Integrate(double (*pfun)(double x), double x1, double x2)
{ double s = 0;
  int i = 1, n;// i为步长，n为等份的个数，n越大，计算结果精度越高
  printf("please input the precision: ");
  scanf("%d", &n);
  while(i <= n)
  {   s += (*pfun)(x1 + (x2 - x1) / n * i);    //窄条的上底边≈下底边
      ++i;
  }
  s *= (x2 - x1) / n;      //每个窄条的面积都应乘以高
  return s;}
```

横坐标

回调函数（Callback Functions）**

- 一个函数 Fun 在执行的过程中，需要主调函数配合做些事，于是会调用主调函数提供的一个函数 H，H 被称为回调函数。回调函数通常是用函数指针传给被调用者。



例：编写一个能根据不同要求进行排序的函数

```
struct Student
{     int no;
      char name[20];
      ......
};
void Sort(Student st[], int num,
                bool (*less_than)(Student *st1, Student *st2))
{ ......
    if (!less_than(&st[i], &st[j])   //比较数组元素大小
    { ......                         //交换st[i]与st[j]
    }
    ......
}
```

```
bool less_than_by_no(Student *st1, Student *st2)
{ return (st1->no < st2->no);
}
bool less_than_by_name(Student *st1, Student *st2)
{ return (strcmp(st1->name, st2->name) < 0);
}
void F()
{
      ......
      Student st[100];
      ......
      Sort(st, 100, less_than_by_no); //按学号从小到大排序
      ......
      Sort(st, 100, less_than_by_name); //按姓名从小到大排序
      ......
}//若想由大到小排序，重新定义比较函数
```

例：快速排序

```
const int N = 5;
void Sort(int a[], int n, bool (*compr)(int x, int y))
{
      if(n==1 || !n)  return;
      int key = a[0], t;
      int i=0, j=n-1;
      while(true)
      {
          while(!compr(a[j], key) && i < j) --j ;
          if(i==j)  break;
          t = a[i], a[i] = a[j], a[j] = t;
          while(!compr(key, a[i]) && i < j) ++i;
          if(i==j)  break;
          t = a[i], a[i] = a[j], a[j] = t;

      }
      Sort(a, i, compr);
      Sort(a + i + 1, n - i - 1, compr);
}
```

```
bool Less(int m, int n){ return m < n;}
bool More(int m, int n){ return m > n;}
Sort(a, N, Less);  //升序
Sort(a, N, More);  //降序
```

- 写一个函数Map，它有三个参数。第一个参数是一个一维double型数组，第二个参数为数组元素个数，第三个参数是一个函数指针，它指向带有一个double型参数、返回值类型为double的函数。函数Map的功能是：把数组（第一个参数）的每个元素替换成 用它原来的值（作为参数）调用第三个参数所指向的函数得到的值。

```
void Map(double d[], int n, double (*fp)(double d))
{
    for (int i=0; i < n; ++i)
        d[i] = (*fp)(d[i]);
    return;
}
```

解释下面表述的含义：

```
int * (*pfnPfnPp)(int(*)(int *, int), int **);
```

```
int *
      (*pfnPfnPp)
                 (                                  );
                                          , int **
                   int
                        (*)
                             (int *, int)
```

函数声明中的形参名可省略

函数声明中的形参名可省略

小结

- 指针：
  - 一种构造数据类型，
  - 地址是一种特殊的整数，将地址专门用指针类型来描述，可以限制该类型数据的操作集，从而得以保护数据。
  - 引用是基于指针封装的一种构造数据类型。

小结

- 要求：
  - 掌握指针/引用的基本概念及定义、初始化和操作方法
  - 掌握指针/引用类型的特征及其典型运用场合
    - 可以在函数间高效地传递数据，
      - ✓函数副作用如何利用或避免
    - 指针可以用来操作动态数据。
      - ✓动态变量和动态数组需要在程序中创建与撤销，使用过程中要注意避免内存泄露和悬浮指针等问题。
    - 一个程序代码量≈40行
  - 继续保持良好的编程习惯

Thanks!