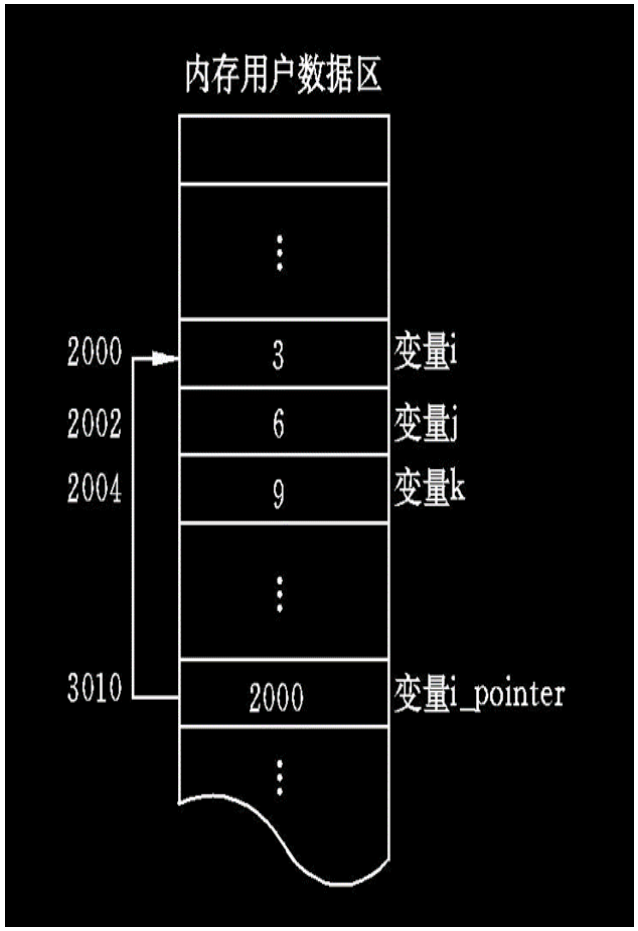
第08-09章 指针与引用

目录

地址和指针的概念

- 变量定义时的内存分配
  - 如果在程序中定义了一个变量，在编译时就给这个变量分配内存单元。系统根据程序中定义的变量类型，分配一定长度的空间。
- 内存单元的地址和内存单元的内容这两个概念的区别
  - 内存区的每一个字节有一个编号，这就是“地址”，它相当于旅馆中的房间号。
  - 在地址所标志的内存单元中存放数据，这相当于旅馆中各个房间中居住旅客一样。



地址和指针的概念

- 变量的“直接访问”方式：通过变量名来访问
  - 在程序中一般是通过变量名来对内存单元进行存取操作的。其实程序经过编译以后已经将变量名转换为变量的地址，对变量值的存取都是通过地址进行的。

- 变量的“间接访问”方式
  - 保存变量地址的变量：

    i_pointer=&i;

    i_pointer的值就是2000，即变量i所占用单元的起始地址
  - 变量的“间接访问”过程：先找到存放“i的地址”的变量，即i_pointer，从中取出i的地址(2000)，然后到2000、2001字节取出i的值(3)。

地址和指针的概念

- “直接访问”和“间接访问”方式下变量的存储过程

  为了表示将数值3送到变量中，可以有两种表达方法:

  (1) 将3送到变量i所标志的单元中。

  (2) 将3送到变量i_pointer所“指向”的单元(即i所标志的单元)中。见上图。所谓“指向”就是通过地址来体现的。

- “指针”和“指针变量”
  - 一个变量的地址称为该变量的“指针”。
  - 如果有一个变量专门用来存放另一变量的地址(即指针)，则它称为“指针变量”。
  - 指针变量的值(即指针变量中存放的值)是指针(地址)。请区分“指针”和“指针变量”这两个概念。

指针变量

- 定义指针变量的一般形式：
  - 基类型 *指针变量名
- 指针变量的基类型：
  - 用来指定该指针变量可以指向的变量的类型。
  - 指针变量只能用来指向和其基类型相同类型的变量，如基类型为int的指针变量只能用来指向整型变量，绝对不能指向实型变量。
  - int *pointer_1，*pointer_2， a;
  - 用赋值语句可使一个指针变量指向另一个变量
- 在定义指针变量时要注意两点：
  - (1) 指针变量前面的“*”，表示该变量的类型为指针型变量。
    - 注意：指针变量名是pointer_1、pointer_2，而不是*pointer_1、*pointer_2。
  - (2) 在定义指针变量时必须指定基类型。例如只有整型变量的地址才能放到指向整型变量的指针变量中。

指针变量的使用

- 指针变量中只能存放地址(指针)，不要随便将一个整型常量(或任何其他非地址类型的数据)赋给一个指针变量。

- 两个相关的运算符
  - (1) &： 取地址运算符。
  - (2) *： 指针运算符(或称“间接访问”运算符)。

  例如：&a为变量a的地址，*p为指针变量p所指向的存储单元。

- 注意：指针使用前必须先初始化！！！
  - pointer_1 = &i;

例如：
```
int* p1;
*p1 = 10;
//10存放在哪里?
```

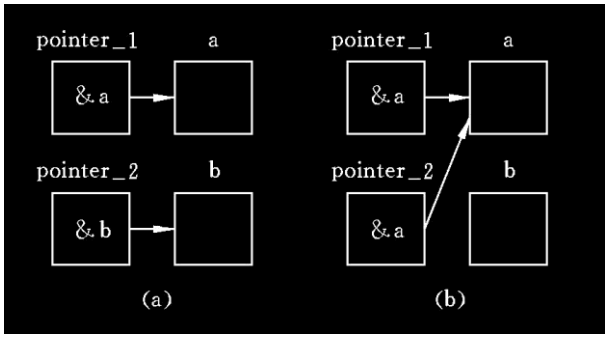
注意：规则修改：指针必须先定义并且初始化后，才可以使用

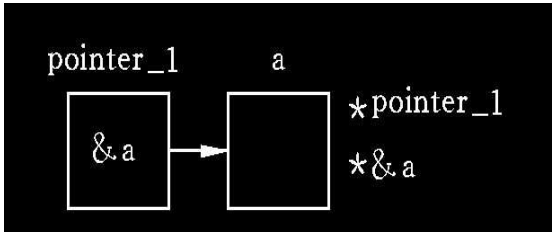
指针变量的使用举例

- 通过指针变量访问整型变量
  - int a,*pa;
  - pa=&a;　　//初始化指针变量pa
  - a=10;
  - *pa=10;　　//等价与a=10
  - pa=10;　　// 一种非常严重的错误，GNU编译报错。

对“&”和“*”运算符的深入说明

- (1) 如果已执行了“pointer_1=&a;”语句，若有&*pointer_1它的含义是什么?
  - &*pointer_1相当于&a
  - pointer_2 = &*pointer_1
  - pointer_2 = pointer_1
- (2) *&a的含义是什么?
  - 与a等价； *&a和*pointer_1的作用是一样的(假设已执行了“pointer_1=&a”)，它们等价于变量a。即*&a与a等价
- (3) (*pointer_1)++相当于a++。注意括号是必要的，如果没有括号，就成为了*pointer_1++，++和*为同一优先级别，相当于先做*pointer_1，然后做pointer_1++





使用指针交换两个变量的值

```
#include <iostream>
using namespace std;
int main()
{
    int a = 9, b = 5;
    int *p1 = &a, *p2 = &b;
    int* p;
    p = p1;
    p1 = p2;
    p2 = p;
    cout << a << "," << b <<
    endl;
    return 0;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int a = 9, b = 5;
    int *p1 = &a, *p2 = &b;
    int temp;
    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
    cout << a << "," << b <<
    endl;
    return 0;
}
```

结论：在使用指针时，要清楚操作的是指针本身，还是指针所指向的变量

指针与数组

- 数组的指针是指数组的起始地址
- 数组元素的指针是数组元素的地址。
- 指向数组元素的指针
  - 定义一个指向数组元素的指针变量的方法，与以前介绍的指向变量的指针变量相同。
  - 例如：

```
int a[10];
int  *p;
p=&a[0];
```

指针与数组

- C++规定数组名代表数组的首地址，也就是第0号元素的地址。下面语句等价：

```
p=&a[0];
p=a;
*p=10;    // 等价于a[0]=10;
```

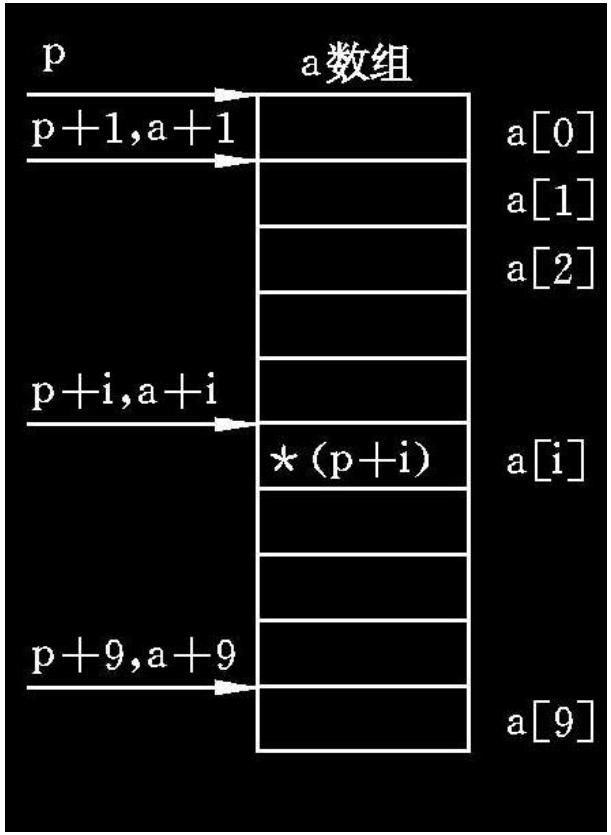
- 注意
  - 数组名a不代表整个数组，上述“p=a;”的作用是“把a数组的首地址赋给指针变量p”，而不是“把数组a各元素的值赋给p”。

指针的运算

- 通过指针对数组元素赋值，如：

```
*p=1;
```

- 如果指针变量p已指向数组中的一个元素，则p+1指向同一数组中的下一个元素(而不是将p值简单地加1)。



指针与数组的进一步理解

- 如果p的初值为&a[0]，则：
  - (1) p+i和a+i就是a[i]的地址
  - (2) *(p+i)或*(a+i)是p+i或a+i所指向的数组元素，即a[i]。例如，*(p+5)或*(a+5)就是a[5]。
  - (3) 指向数组的指针变量也可以带下标，如p[i]与*(p+i)等价

- 数组元素引用的两种方法：
  - (1) 下标法，如a[i]或p[i]形式，p是指向数组的指针变量，其初值p=a。
  - (2) 指针法，如*(a+i)或*(p+i)。其中a是数组名，p是指向数组的指针变量，其初值p=a。

指针与数组——举例

```
double GetAvg(int *arry, int len)
{
    double sum=0;

    for(k=1; k<len; k++){
        sum += arry[k];
    }

    return sum/len;
}
```

- sum += arry[k]等价的语句：

```
sum += *(arry+k)
```

```
sum += *arry;
arry++;
```

指针与数组——举例

- 请问输出结果是什么？
- 注意：
  - 数组名是常量，不可修改

```
int arry[5]={1,2,3};

for(int i=0; i<5; i++)
{
      cout<<*(arry+i)<< “  “
}
```

```
int arry[5]={1,2,3};

for(int i=0; i<5; i++)
{
      cout<<*arry<< “  ”
      ++arry;      //error
}
```

```
int arry[5]={1,2,3};
double avg;
avg = GetAvg(arry,5) //ok
```

指针的限定

- 指针限定的概念：常量性
- 指针常量：指针值不能改变，即该指针常量只能指向一个个体
- 常量指针：指向的值不能通过指针进行间接改变。决不意味着指向的只能是常量
- 修饰的主语是谁非常关键：区分依据为判别const和数据类型的先后次序
- 两级修饰的特殊性 const int * const icp = &c

指针的限定示例

- **注意：**
  - 未加限定的指针不可以指向常量

```
char s[]= "Gorm";
const char c='A';
const char *pc = s;   //常量指针
pc[3] = 'g';          // 不可以通过pc去修改指向的对象
pc = &c;              // 可以修改pc的指向
s[3] = 'g';           // s本身不是常量，可以修改

char *const cp = s; //指针常量
cp[3] = 'a';          // 可以通过指针常量修改指向的对象
cp = &c;              // 不可以改变指针常量的指向

const char *const cpc=s; // 指向常量的指针常量
cpc[3]='g';       // 不可以修改指向的对象
cpc = &c;         // 不可以修改其指向

char *p;      // 普通指针
p = s;        // 可以指向普通变量
p = &c;       // 不可以指向常量
char *const pc2=&c; // 不可以指向常量
```

指针的限定小结

- 指向常量的指针表示不能通过该指针改变其所指的量的值，但可以通过其他方式改变
- 指向常量的指针常用于保护函数参数
  - 例如
    - char* strcpy(char *p, const char *q); //不能修改*q
- 可以将一个变量的地址赋给一个指向常量的指针
- 不可以将一个常量的地址赋给一个非常量指针

```
int  a;
const int *p = &a; //OK
```

```
const int a = 5;
int* p = &a;   // error
int* const p1=&a; //error
```

指针与函数

- 指针作为函数参数
  - 将一个变量的地址传到另一个函数中。
- 例如：

```
void swapp(int *n1,int *n2)
{
    int temp;
    if (*n1> *n2){
        temp = *n1;
        *n1 = *n2;
        *n2 = temp;}
}
```

```
void main()
{
    int a,b;
    scanf("%d%d",&a,&b);
    printf("a=%d; b=%d",a,b);
    swapp(&a,&b);
    printf("a=%d; b=%d",a,b);
}
```

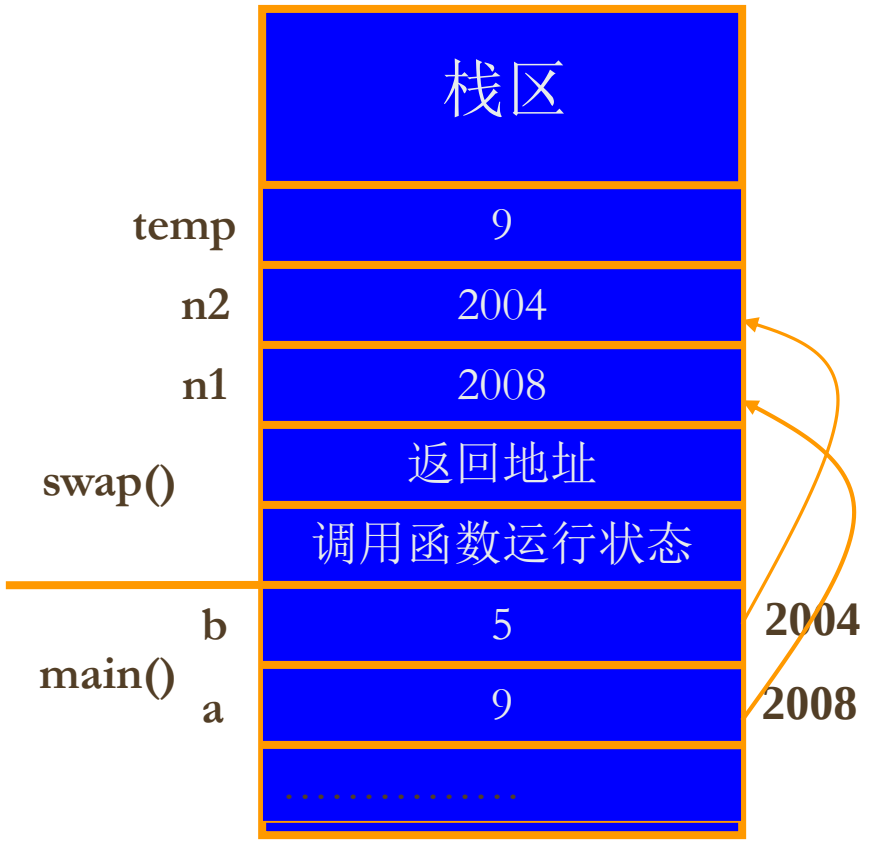
输入：9 5

输出：a=9; b=5

输出：a=5; b=9

指针与函数

- 函数的调用过程
  - 首先把a、b的地址值复制到swapp函数的堆栈区间，并赋值给行参n1、n2.
  - 交换n1、n2所指向内存单元的值。
  - 返回。



- 结果
  - 被调函数进行的操作结果反应在主调函数中。
  - 被调函数实际上是在主调函数的堆栈区间中进行的操作
  - 实际是一种地址传递方式

指针与函数

- 指针变量作为函数参数典型错误举例一

```
swap(int *n1，int *n2)
{     int   *temp;
      *temp=*n1;
      *n1=*n2;
      *n2=*temp;
}
```

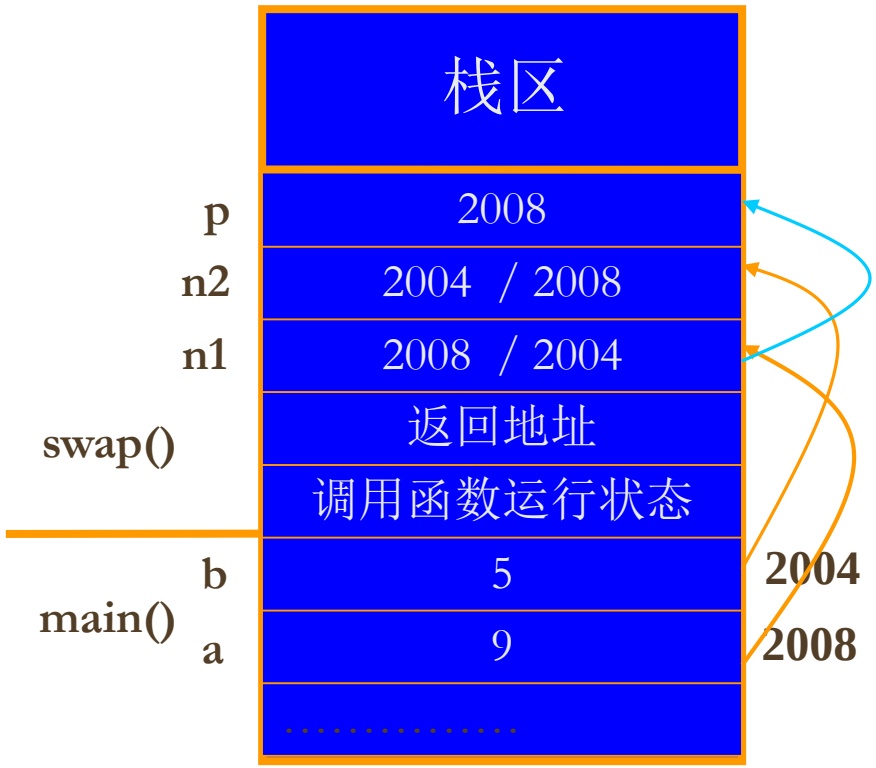
/*此语句有问题*/

- 问题：
  - temp没有初始化，指向的内存单元是不确定的
  - 对没有初始化的指针赋值非常危险！！！

指针与函数

- 指针变量作为函数参数典型错误举例二

```
swap(int *n1，int *n2)
{
        int   *p;
        p=n1;
        n1=n2;
        n2=p;
}
```



- 问题
  - 交换的只是两个形参所指向的内存单元。

指针与函数

- C++语言中实参变量和形参变量之间的数据传递是单向的“值传递”方式。指针变量作函数参数也要遵循这一规则，这时的值是指针的值，也就是地址值。

- 调用函数不可能改变实参指针变量的值，但可以改变实参指针变量所指变量的值。

- 运用指针变量作参数，可以从函数调用中得到多个变化了的值。如果不用指针变量是难以做到这一点的。参考下例：

指针与函数

```
swap(int *pt1,  int *pt2)
{
    int temp;
    temp=*pt1;
    *pt1=*pt2;
    *pt2=temp;
}
```

```
exchange(int *q1,  int *q2,  int *q3)
{
     if(*q1<*q2) swap(q1, q2);
     if(*q1<*q3) swap(q1, q3);
     if(*q2<*q3) swap(q2, q3);
}
```

```
main()
{
     int a, B, C, *p1, *p2, *p3;
    scanf("%d, %d, %d", &a, &B, &C);
    p1=&a; p2=&B; p3=&C;
    exchange(p1, p2, p3);
    printf("\n%d, %d, %d\n", a, B, C);
}
```

运行情况如下：

9, 0, 10

10, 9, 0

指针举例

- 编写一个函数，找出实型数组中的最大值、最小值，计算数组元素的平均值。在main函数中测试该函数的正确性。
- 分析：函数需要返回多个值：只能通过参数带出多个值

指针举例

```
void getMaxMinAvg(double* arry, int len,
        double *max,double* min, double* avg)
{
    *avg=*arry;
    *max=*arry;
    *min=*arry;
    for(int i=1; i<len; i++)
    {
        *max = *max>*(arry+i)? *max:*(arry+i);
        *min = *min<*(arry+i)?*min:*(arry+i);
        *avg += *(arry+i);
    }
    *avg /= len;
}
```

返回指针值的函数

- 一般定义形式
  - 类型名 *函数名(参数表);
  - 说明：
    - 类型名： 表示函数返回的指针变量指向的数据类型
  - 注意区分
    - 类型名 (*指针变量名)(); //定义了一个指针变量
    - 类型名 *函数名(参数表); //定义了一个函数

返回指针值的函数的常见错误

- 将指向局部变量的指针值返回
- 例如：

```
char* getString()
{
        char str[128];
        strcpy(str,"You Are welcome.");
        return str;
}

main()
{
      char *ptr;
      ptr = getString();
      printf("%s",ptr);
}
```

注意：

随着*getString()*函数的返回，*str*数组已经没有意义了，因此再让*ptr*指向*str*就会出错。

不能将函数内部具有局部作用域的数据地址作为返回值！

返回的地址必须仍然是指向一个具体的地方！

返回指针值的函数小结

- 内存分配分为三种：
  - 1从静态存储区域分配。内存在程序编译时候已经分配好，这块内存在程序整个运行期间都存在。对应的就是：全局变量，static变量
  - 2从栈上创建。函数内局部变量申请的存储单元在栈上创建，函数结束时栈空间自动释放。比如：int a; int *p = &a; return p; 这样返回的地址就不知道指向的是什么东西了。
  - 3从堆上分配，亦称动态分配。运行时用malloc(c语言中使用)或者new(c++语言中使用)申请内存，程序员自己负责用free或者delete释放，因此生存期是由我们来定的。即使在函数体内声明的也不会随函数结束而消失。
- 返回的指针必须是以下两种情况之一
  - 指向全局变量的指针值
  - 指向堆中变量的指针值: 要注意内存的释放，否则，容易引起内存泄漏
  - 参数传入的指针值

指针数组

- 指针数组的概念
  - 定义：一个数组，其元素均为指针类型数据，称为指针数组，也就是说，指针数组中的每一个元素都相当于一个指针变量。
  - 一般定义形式

    类型名　*数组名[数组长度]
- 注意区分：
  - 类型名　*数组名[数组长度]; //指针数组
  - 类型名　(*数组名)[数组长度]; //行指针，指向一维数组的指针
- 用途：
  - 适合用于指向若干字符串
  - 用于指向若干不等长的数组

动态内存

- 两种方式
  - 向量等容器：需要处理的数据量变化不大
  - 动态内存分配：new和delete运算符

- new/delete运算符
  - 分配内存，同时进行初始化
  - 初始化就是会调用构造函数进行初始化
  - 调用析构函数释放对象。

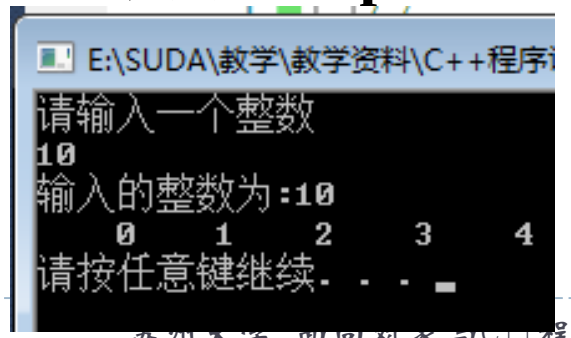
动态内存分配: new/delete

```
int *p1, *p2;
string *pstr;

p1 = new int;          //分配一个整型数
delete p1;             //释放一个整型数的内存空间
p2 = new int[5];       //分配一个包含5整型数的数组空间
delete [] p2;          //释放一个包含5整型数的数组空间
pstr = new string;     //分配一个string对象空间
delete pstr;           //释放一个string对象空间
```

```
//====================
// new/delete 运算实例
//====================
#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
    int *p;
    p = new int;
    cout<<"请输入一个整数:\n";
    cin>>*p;
    cout<<"输入的整数为:<<*p<<endl;
    delete p;
    p = new int[5];
    for(int i=0; i<5; i++)
        p[i] = i;
    for(int i=0; i<5; i++)
        cout<<setw(5)<<p[i];
    cout<<"\n';
    delete []p;

    return 0;
}
```



注意：
(1) 申请单个变量和申请数组的区别
(2) 删除单个变量和删除数组的区别

命令行参数

- 程序运行源自main函数被操作系统调用
  - main函数可具有参数,由操作系统传递实参
  - 启动程序即发命令,命令有表示方式—命令行
  - 命令行中除了程序名还可有参数,参数被操作系统解释,作为main参数,回馈给程序
  - 发命令行时,给出参数,就能让程序获取
  - 对main函数参数的处理,就是对命令行参数的处理
  - 参数有二,第一参数为整型,说明第二参数(字串数组即字符数组的指针)的元素个数,第二参数为字串数组(依序列出命令行参数字串)

命令行参数

- main函数原型

```
#include<iostream>
using namespace std;
int main(int argc, char* argv[])
{
  for(int i=0; i<argc; i++)
    cout<<"arg"<<i<<": "<<argv[i]<<"\n";
}
```

- main函数返回类型
  - void main(){}非标准C++及C编程
  - int main(){} C++98标准
  - C++兼容标准和非标准写法,提倡标准写法int **main(){}**

命令行参数

- main函数返回语句
  - 针对标准写法,函数最后应有return 0;事实上有没有都对
  - A）操作系统有默认值0,编译器有对main的特殊做法,有无返回语句均可
  - B）过程化编程可省略return语句
  - C）对象化编程不要省略return语句

在对象化编程教学中,常涉及演示对象构造与析构的顺序,运行结果展示,若无main函数的return语句,则运行到main函数结束的花括号语句时,析构函数就不会执行,需要在C++之外的命令行提示符平台中运行,才会输出全部,或者main函数最后若有return语句,也会在C++平台中,输出全部.

函数指针

- 数据指针：
  - 指向数据区域的指针，eg:int a; int *p = &a;
- 函数指针：
  - 指向代码区域的指针称为指向函数的指针，简称为函数指针。
  - eg:
    - int (*gp)(int);　　//定义了一个函数指针
  - 说明：
    - 函数指针使得函数可以作为实际参数使用
    - 函数名作为实参时自动转换为函数指针

```
bool sortBylength(const string& s1,const string& s2);
sort(v_str.begin(), v_str.end(), sortBylength);
```

函数指针

- 函数指针和返回指针的函数的区别
  - 形式不同
    - int* func(int);　　// 声明了一个返回指针的函数
    - int (*gfunc)(int);　// 定义了一个函数指针，是指针变量
- 函数指针
  - 函数指针的初始化
    - int (*gfunc)(int) = func;　//ok
    - gfunc = func;　　// ok
  - 函数指针的类型必须接受编译器的检查。
    - void f();　　// 普通函数
    - void (*fp)();　　//　函数指针
    - gfunc = f;　　// error: 错误的指针转换
    - gfunc = fp;　　// error: 错误的指针转换

```
//=====================================
// f0506.cpp
// 函数指针传递
//=====================================
#include <iostream>
#include <algorithm>
#include <vector>
#include <iomanip>
using namespace std;
//---------------------------------
int bitSum(int a);
bool lessThanBitSum(int a, int b){ return bitSum(a)<bitSum(b); }
//---------------------------------
int bitSum(int a){
  int sum=0;
  for(int x=a; x; x/=10)  sum += x%10;
  return sum;
}//=====================================
```

功能：计算整数的各位数字之和

```
int main()
{
    int a[] = {33, 61, 12, 19, 14, 71, 78, 59};
    vector<int> aa(a, a+8);
    sort(aa.begin(), aa.end(), lessThanBitSum);
    for(int i=0; i<aa.size(); ++i)
    {
        cout<<setw(5)<<aa[i]<<" ";
    }
    cout<<"\n";
}//-------------------------------------
```

用数组a的前面8个数据来初始化向量aa;

函数指针作为实参

引用

- 引用的概念
  - 引用是程序中另外一个目标（变量）的别名。
  - 举例：
    - int someint;
    - int &rint = someint;
  - 引用只有声明，没有定义；引用不是值，不占存储空
  - 引用在声明时必须初始化
    - int someint;
    - int &rint;
    - rint = someint;

引用

- &的不同理解
  - **int *p = &a; //取址符**
  - **int &rint = someint; //引用符**
  - **a= b&c; //位与运算符**
  - **if (a&&b) //逻辑与运算符**
- 引用的操作
  - 引用的取地址操作的正确理解：只能找到所引用的目标的地址。引用本身在内存中是不占空间的，无地址
  - 引用一旦初始化，就和确定的目标紧密地联系在一起了，不能再建立它和其他目标的引用关系

引用

- 能够建立合法引用的范围
  - 引用的初始值不是左值的理解：临时变量

    double &rr=1;　等价于 double temp; temp=double(1); double &rr=temp;

    不建议对字面常量声明引用。
  - 指针变量的引用

    int *a;
    int * &p = a;
    int b = 8;
    p = &b;
  - 禁止建立void类型的引用
  - 不能建立数组的引用
  - 引用没有指针的概念，也不存在引用的引用
  - 不能用类型来进行引用的初始化 **int &ra = int;**

引用

- 引用和指针的关系
  - 引用和指针的区别：指针可以改变关联的实体，而引用只能操作一个实体
  - 从本质上而言，引用值是引自所指向的实体
  - 引用看似是直接访问实体，但实际上是指针的间接访问。编译器将会使引用转换为*rInt的操作。但这个操作是幕后进行的
  - 与指针相比，引用隐去了地址操作（幕后进行），或者说封锁了地址修改的可能性，使得间接访问操作更加安全可靠
- 引用作为函数参数
  - 将引用作为函数参数和将指针作为函数参数的效果是一样的，都不会在函数内部建立实参的值的副本

引用与指针使用对比

```
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}

//调用形式
int x=5, y=9;
swap(&x,&y);
```

```
void swap(int& a, int& b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

//调用形式
int x=5, y=9;
swap(x,y);
```

引用

- 引用作为函数参数的过程中存在的问题：在没有实际看到函数的定义和原型时，仅仅看到函数调用并不能准确地判断出执行效果
- 引用作为函数参数时导致的典型编译错误举例

```
void  fn(int s)      {        ……    }
void fn(int &s)    {        ……    }
void main( )
{
    int a = 5;
    fn( a );          //??????????
}
```

应用引用返回多个值

```
void getMaxMinAvg(double* arry, int len,
        double &max,double& min, double& avg)
{
    avg=*arry;
    max=*arry;
    min=*arry;
    for(int i=1; i<len; i++)
    {
        max = max>*(arry+i)? max:*(arry+i);
        min = min<*(arry+i)?min:*(arry+i);
        avg += *(arry+i);
    }
    avg /= len;
}
```

引用

- 函数的返回值类型为引用类型
  - 作用机制的理解：
    - 如果函数返回值类型为值，则要生成一个值的副本
    - 但如果函数返回值类型是引用，则不生成值的副本；
  - 不能建立一个临时对象的引用，这是非常危险的行为；
  - 返回的引用作为左值参与赋值运算也是程序设计中的禁忌。但有时又必须这样使用(运算符重载时就需要)
- 可以作为函数返回值的实体
  - 全局变量的引用
  - 引用类型的参数
  - 堆空间变量，但要注意内存泄漏的问题。

函数返回引用类型——分析

```
//--------------------
//    ch9_6.cpp
//--------------------
#include<iostream>
using namespace std;
//--------------------
float temp;
//--------------------
float fn1(float r){
  temp = r*r*3.14;
  return temp;
}//-------------------
float& fn2(float r){
  temp = r*r*3.14;
  return temp;
}//--------------
```

```
int main()
{
  float a=fn1(5.0);       //1
  float& b=fn1(5.0);      //2:warning
  float c=fn2(5.0);       //3
  float& d=fn2(5.0);      //4
  cout<<a<<endl;
  cout<<b<<endl;
  cout<<c<<endl;
  cout<<d<<endl;
}//---------------
```

引用

- 用const限定引用
  - 引用作为函数参数的利弊分析
  - const指针作为函数参数
  - const引用作为函数参数
- 可以对传值参数进行const限定，但没有意义。
- 举例
  - string_copy(string* ptarget_str, const string*psource_str);
  - string_copy(string&target_str, const string&source_str);

程序举例1

- 从键盘输入一个英文语句，请提出该语句中的所有单词，并对所有单词按照升序方式排序，然后输出到屏幕上，要求每个单词输出一行。
- 独立功能：
  - 提取单词：
  - 排序：
  - 输出：

程序举例2

- 编写程序：先产生一个100以内的随机整数n,然后再产生n个1000以内的随机整数，提取这些整数中没有重复数字的整数，对这些整数进行升序排序，然后输出到屏幕，每行输出8个整数，每个整数占5列。
- 独立功能：