C++编程(3)

Tang Xiaosheng
北京邮电大学电信工程学院

第五章 指针、数组和结构

- 指针
- 数组
- 到数组的指针
- 常量
- 引用
- 指向void的指针
- 结构
- 忠告

5.1 指针

- 对于类型T，T*是“到T的指针”，也就是说，一个类型为T*的变量能保存一个类型T的对象的地址
- 对指针的基本操作是dereference，书中翻译为间接引用，更多的资料中翻译为“提领”，间接运算符是(前缀的)*

```
char c = ‘a’;
char* p = &c;
//p 保存c的地址
```

```
char c = ‘a’;
char* p = &c;
char c2 = *p;
// c2 = ‘a’;
```

零

- 0是一个整数，可以用于任意整型、浮点类型、指针、指向成员的指针的常量等，其类型由具体环境确定
- 没有任何地址会被分配到地址0，所以0被当成一个指针常量，表示该指针没有指向任何对象
- C中，使用NULL代表0指针，C++中则使用数字0，可以const int NULL = 0;

5.2 数组

- 对于类型T，T[size]就是“具有size个T类型元素的数组”类型，元素下标从0到size-1
- 数组元素的个数必须是一个常量表达式
- 多维数组被表示为数组的数组

```
float v[3]; // v[0], v[1], v[2]
char* a[32]; // 32个到char的指针
// a[0] … a[31]
 int d2[10][20];
 int bad[2,5]; // Error
```

```
void f(int i)
{
    int v1[i]; // Error
    vector<int> v2(i);
}
```

5.2.1 数组初始化

```
int v1[] = {1, 2, 3, 4}; // Size = 4
char v2[] = { 'a', 'b', 'c', 0}; // Size = 4
char v3[2] = { 'a', 'b', 0}; // Error
char v4[3] = { 'a', 'b', 0}; // ok
int v5[8] = {1, 2, 3, 4};
```

相当于`int v5[8] = { 1, 2, 3, 4, 0, 0, 0, 0};`

注意：不存在与数组初始化相对应的数组赋值

```
void f()
{
    v4 = { 'c', 'd', 0}; // Error
}
```

可以使用vector或者valarray进行赋值

5.2.2 字符串文字量

- 字符串文字量是用双引号括起来的字符序列
- 一个字符串文字量里包含的字符个数比它看起来的字符数多一个，它总是由一个空字符’\0’结束，空字符的值是0
- 例如：sizeof(“test”)=5，但是strlen(“test”)=4
- 字符串文字量的类型是“适当个数的const字符的数组”，所以“test”的类型就是const char[5]，此为常量，而且是静态分配的
- 可以用字符串文字量给一个char*赋值，若想修改一个字符串，可以先将其复制到一个数组中去

字符串示例

```
void f()
{
    char* p = “Plato”;
    p[3] = ‘e’;
    //Error: 给常量赋值，
      结果无定义
}

void f2()
{
    char p[] = “Zeno”;
    p[0] = ‘R’;
    // ok
}
```

```
const char* error_msg()
{
    // …
    return “range error”;
} // ok

const char* p = “Heraclitus”;
const char* q = “Heraclitus”;

void g()
{
    if(p==q) cout << “One!”;
    //…
} // 结果由编译器实现决定
```

字符串示例

字符串中不能有真正的换行，比如

```
char str[] = " this is a two
    lines string";
//Error
```

```
char str[] = "this is a two \n lines string";
```

可以：

```
char alpha[] = "abcdefghijklmn"
  "opqrstuvwxyz";
```

等价于

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz";
```

另外：

```
char str[] = "Jers\000Munk"; // \0后面的将会被忽略
```

5.3 到数组的指针

- 数组名字可以被用作它的开始元素的指针
- 注意：取得“开始元素之前的一个位置”没有任何意义，因为数组常常被分配在机器地址的边界上
- 数组名可以被隐式地转换成数组的开始元素的指针(这在C风格代码的函数调用中被广泛使用)，但是这将丢失数组的大小信息，vector和string没有此类问题

数组指针示例

```
int v[] = {1, 2, 3, 4};
int* p1 = v;
int* p2 = &v[0];
int* p3 = &v[4];
//最后元素之后的一个位置
```

```
extern “C” int strlen(const char*); // string.h
void f()
{
    char v[] = “Annemarie”;
    char* p = v;  //隐式地从char[]转换到char*
    strlen(p);
    strlen(v);      //隐式地从char[]转换到char*
    v = p;          //Error: 不能给数组赋值
}
```

5.3.1 在数组中漫游

```
void fi(char v[])
{
    for(int i = 0; v[i] != 0; i++) use(v[i]);
}
```

等价于

```
void fi(char v[])
{
    for(char* p =v; *p!=0; p++) use(v[i]);
}
```

需要注意的是，普通数组(例如非char类型的数组)不具备自描述性，当遍历时，需要提供数组长度

```
#include <iostream>
int main()
{
    int vi[10];
    short vs[10];
    std::cout << &vi[0] << ‘ ‘ << &vi[1] << std::endl;
    std::cout << &vs[0] << ‘ ‘ << &vs[1] << std::endl
}
```

可能输出

0x7fffaef0 0x7fffaef4

0x7fffaedc 0x7fffaede

指针的运算

- 假设T* p;
- p++相当于p变为指向类型为T的下一个元素的地址，即p和p++之间的实际距离为sizeof(T)
- 只有两个指针指向同一个数组的元素时，指针相减才有意义，结果为两个指针之间数组元素个数(一个整数)
- 指针相加没有意义，是不允许的

指针运算示例

```
void f()
{
    int v1[10];
    int v2[10];

    int i1 = &v1[5] - &v1[3]; // i1 = 2
    int i2 = &v1[5] - &v2[3]; //结果无定义

    int* p1 = v2 + 2;  // p1= &v2[2]
    int* p2 = v2  - 2;  // p2无定义
}
```

5.4 常量

- const是为了描述概念“不变化的值”
- 常量在编程中非常有用：
  - 许多对象在初始化之后就不再改变自己的数值
  - 采用符号常量写出的代码更容易维护
  - 指针常常是边读边移动，而不是边写边移动
  - 许多函数参数是只读不写的

常量示例

```
const int model = 90;
const int v[] = {1, 2, 3, 4};
const int x; // Error

void f()
{
    model = 200; // Error
    v[2] ++;       // Error
}
```

```
void g(const X* p)
{
    //这里不能修改*p
}

void h()
{
    X val; //val可以被修改
    g(&val); // ok
    //…
}
```

编程中，应系统化地使用符号常量，以避免出现“神秘的数值”**(magic numbers)**

5.4.1 指针和常量

- 使用一个指针时涉及到两个对象：指针本身和被它所指的对象
- 要将指针本身而不是被指对象声明为常量，必须使用声明运算符*const，而不能只是简单地用const

指针和常量示例(1)

```
char* const cp;
//到char的const指针
char const* pc;
//到const char的指针
const char* pc2;
//到const char的指针
```

对于const定义，从右向左读有助于我们理解
例如：
cp是一个const指针到char
pc2是一个指针指向const char

```
void f1(char* p)
{
    char s[] = "Gorm";

    const char* pc = s; //指向常量
    pc[3] = 'g';  // Error
    pc = p; // ok

    char* const cp = s; //常量指针
    cp[3] = 'a';  // ok
    cp = p; // Error

    const char* const cpc = s;
    cpc[3] = 'a'; // Error
    cpc = p; // Error
}
```

指针和常量示例(2)

```
char* strcpy(char* p, const char* q);
//const表示strcpy函数的执行体不能修改*q
```

可以将一个变量的地址赋给一个指向常量的指针，因为这么做不会造成任何伤害，但是不能将常量的地址赋给一个未加限制的指针，因为这样将会允许修改该对象的值了

```
void f4()
{
    int a = 1; const int c = 2;
    const int* p1 = &c; // ok
    const int* p2 = &a; // ok
    int* p3 = &c; // Error
    *p3 = 7; // Error
}
```

5.5 引用

- 一个引用就是某对象的另一个名字，引用的主要用途是为了描述函数的参数和返回值，特别是为了运算符的重载
- 记法 X& 表示到X的引用
- 为了确保引用总是某个东西的名字(总能约束到某个对象)，必须对引用进行初始化，而且一旦被初始化后就不可能改变了
- 引用的一种最明显的实现方式是作为一个(常量)指针，每次使用它时都自动地做间接访问

引用示例

```
void f()
{
    int i = 1;
    int& r = i;
    //r和i都是同一个int数值
    int x = r; // x = 1
    r = 2;      // i = 2
}

int i = 1;
int& r1 = i; // ok
int& r2; // Error 没有初始化
extern int& r3;
// ok r3在别处初始化
```

```
void g()
{
    int ii = 0;
    int& rr = ii;
    rr ++;   // ii被增加1
    int* pp = &rr; //pp指向ii
}
void increment(int& aa)
{ aa++;}
void f()
{
    int x = 1;
    increment(x); //x = 2
}
```

为提高程序的可读性，通常应该尽可能避免让函数修改传递过来的参数，相反应该让函数明确地返回一个值，或者明确要求一个指针参数，使用引用参数，会给他人一种该函数要修改这个参数的强烈暗示

```
int next(int p)
  { return p+1; }
void incr(int* p) { (*p)++; }
void g()
{
    int x = 1;
    increment(x);  // x = 2
    x = next(x);   // x = 3
    incr(&x);      // x = 4
}
```

另外一种错误的引用声明方式

```
double& dr = 1;
// Error, cannot convert
from  'const int'  to
  'double &'

const double& cdr = 1;
// ok
```

5.6 指向void的指针

- 一个指向任何对象类型的指针(不包括函数指针和成员指针)都可以赋值给类型为void*的变量
- void*可以赋值给另一个void*
- 两个void*可以比较相等与否
- 可以显式地将void*转换到另一个类型
- 其他任何操作都是不安全的
- void*的最重要的用途是需要向函数传递一个指针，而又不能对对象的类型做任何假设，或者从函数返回一个无类型的对象，注意：使用前必须经过显式的类型转换

void*示例

```
void f(int* pi)
{
    void* pv = pi; // ok
    *pv;    // Error
    pv++; // Error
    int* pi2 = static_cast<int*>(pv); // ok
    double* pd1 = pv; // Error
    double* pd2 = pi;  // Error
    double* pd3 = static_cast<double*>(pv); //不安全
}
```

5.7 结构

- 数组是相同类型的元素的一个集合，struct则是(几乎)任意类型数组的一个集合
- 结构类型对象的大小未必是其成员的大小之和，这是因为许多机器要求将对象分配在某种与机器系统结构有关的边界上，比如说机器的字边界，这种情况下，对象被称为是具有**对齐**的性质

结构示例(1)

```
struct address {
    char* name;
    long int number; // 4 bytes
    char* street;
    char* town;
    char state[2];
    long zip;
}; //很多机器上，sizeof(address)是24
address jd;
jd.name = “Jim Dandy”;
jd.number = 61;

address jd = { “Jim Dandy”, 61, “South
St”, “New Providence”, {‘N’, ‘J’}, 7974};
```

```
address *p;
…

cout << p->name ;
cout << p->number;

p->name
等价于
(*p).name
```

结构示例(2)

- 类型的名字在出现后立即就可以使用，不必等到看到完整声明，但是在完整声明被看到之前，不能去声明这个结构类型的新对象

```
struct Link{
    Link* previous;
    Link* successor;
}; // ok

struct No_Good{
    No_Good member;
}; // Error
```

```
struct List; //先声明一下
struct Link{
    Link* pre;
    Link* suc;
    List* member_of;
};
struct List{
    Link* head;
}
```

5.7.1 类型等价

- 两个结构总是不同的类型，即使它们有相同的成员，此外，结构类型也与各种基本类型不同，每个结构在程序里都是唯一的

```
struct S1 { int a; };
struct S2 { int a; };

S1 x;
S2 y = x; // Error

int i = x;  // Error
```

5.8 忠告

- [1] 避免非平凡的指针算术
- [2] 当心，不要超出数组的界限去写
- [3] 尽量使用0而不是NULL
- [4] 尽量使用vector和valarray而不是内部数组
- [5] 尽量使用string而不是以0结尾的char数组
- [6] 尽量少用普通的引用参数
- [7] 避免void*，除了在某些低级代码中
- [8] 避免在代码中使用非平凡的文字量(magic number)，相反，应该定义和使用符号常量