

基础知识4

现有一个电话号码簿,其中有姓名、电话号码,当输入电话号码时,查找出姓名与电话号码:当输入姓名时,同样查找出姓名与电话号码,还允许不完全输入查找,例如输入01时查找出所有以010开头的号码,输入“杨”时列出所有姓名以“杨”开头的号码。

```
#include <iostream>
using namespace std;

struct Phone {
    char number[20];
    char name[16];
};

Phone PhoneBook[] = {
    {"027888888888", "Name1"},
    {"010678923658", "Name2"},
    {"0755628328178", "Name3"},
    {"139012345608", "Name4"},
    {"632478988", "Name5"},
    {"872537828", "Name6"},
    {"972638288", "Name7"},
    {"0010678923658", "Name8"}
};

bool SubStrNum(char *str, char *substr) {
    int h1, h2; // 存储 str 和 substr 的长度
    int p1, p2; // 指向 str、substr

    h1 = strlen(str);
    h2 = strlen(substr);

    if (h2 > h1)
        return false;

    p1 = 0;
    while (p1 < h1) { // str 未到尾
        p2 = 0; // 重新开始比较
        while (str[p1] == substr[p2] && p2 < h2 && p1 < h1) {
            p1++; // 指针向后移
```

```

        p2++; // 指针向后移
    }

    if (p1 == h1 && p2 < h2) // 如果 str 到尾但 substr 未到尾，结束比较
        return false;

    if (p2 == h2) // 如果 substr 到尾，找到一个子串
        return true;
    else // 有字符不相等，指针 p1 回移
        p1 = p1 - p2 + 1;
    }
    return false;
}

int main() {
    int phones;
    char string[20];

    phones = sizeof(PhoneBook) / sizeof(PhoneBook[0]);

    cout << "Input phone number or name:";
    cin >> string;

    for (int i = 0; i < phones; i++) { // 在号码中找
        if (SubStrNum(PhoneBook[i].number, string))
            cout << i << ")\t" << PhoneBook[i].number << "\t" <<
PhoneBook[i].name << endl;
    }

    for (int i = 0; i < phones; i++) { // 在名字中找
        if (SubStrNum(PhoneBook[i].name, string))
            cout << i << "\t" << PhoneBook[i].number << "\t" <<
PhoneBook[i].name << endl;
    }

    return 0;
}

```

这上面那段代码就是为了实现在一个大字符串中寻找一个小的字符串

```

#include <iostream>
#include <cstring>

```

```

bool SubStrNum(const char* str, const char* substr) {
    int h1, h2; // 存储 str 和 substr 的长度
    int p1, p2; // 指向 str、substr

    h1 = strlen(str);
    h2 = strlen(substr);

    if (h2 > h1)
        return false;

    p1 = 0;
    while (p1 < h1) { // str 未到尾
        p2 = 0; // 重新开始比较
        while (str[p1] == substr[p2] && p2 < h2 && p1 < h1) {
            p1++; // 指针向后移
            p2++;
        }

        if (p1 == h1 && p2 < h2) // 如果 str 到尾但 substr 未到尾，结束比较
            return false;

        if (p2 == h2) // 如果 substr 到尾，找到一个子串
            return true;
        else // 有字符不相等，指针 p1 回移
            p1 = p1 - p2 + 1;
    }
    return false;
}

```

```

int main() {
    const char* str1 = "ABCMGHI";
    const char* str2 = "CM";

    if (SubStrNum(str1, str2))
        std::cout << str1 << " 包含 " << str2 << std::endl;
    else
        std::cout << str1 << " 不包含 " << str2 << std::endl;

    return 0;
}

```

这是一个具体讲解例子

编写一个函数，返回任意大的两整数之差(提示:大整数用字符串来表示)。

```
#include <iostream>
#include <cstring>

char* lsub(char* s1, char* s2) {
    int n1, n2, n;
    char *res, c = 0;
    n1 = strlen(s1);
    n2 = strlen(s2);
    n = n1 > n2 ? n1 : n2;

    res = new char[n + 2];
    res[0] = '0'; // 结果初值为0
    res[1] = '\0';
    res[n + 1] = '\0';

    if (n1 > n2)
        res[0] = '+';
    else if (n1 < n2)
        res[0] = '-';

    for (int i = 0; i < n; i++) {
        if (s1[i] < s2[i]) {
            res[0] = '-';
            break;
        } else if (s1[i] > s2[i]) {
            res[0] = '+';
            break;
        }
    }

    if (res[0] == '-') {
        for (int i = n; i > 0; i--) {
            res[i] = i > n - n1 ? s2[i - n + n2 - 1] : '0';
        }
        for (int i = n; i > 0; i--) {
            // 进行 s1-res
            if (s1[i - 1] - c >= res[i]) {
                res[i] = s1[i - 1] - res[i] - c + '0';
                c = '0';
            } else {
                // 为上次相减的借位
```

```

        res[i] = s1[i - 1] - res[i] - c + 10 + '0';
        c = '1';
    }
}
} else {
    for (int i = n; i > 0; i--) {
        res[i] = i > n - n2 ? s2[i + n2 - n1 - 1] : '0';
    }
    for (int i = n; i > 0; i--) {
        // 进行 s1-res
        if (s1[i - 1] - c >= res[i]) {
            res[i] = s1[i - 1] - res[i] - c + '0';
            c = '0';
        } else {
            // 为上次相减的借位
            res[i] = s1[i - 1] - res[i] - c + 10 + '0';
            c = '1';
        }
    }
}

return res;
}

int main() {
    char s1[100], s2[100];
    std::cin >> s1 >> s2;
    char* result = lsub(s1, s2);
    std::cout << "Result: " << result << std::endl;
    delete[] result; // 释放内存
    return 0;
}

```

在你提供的代码中，`i - n + n2 - 1` 这一部分主要用于确定在循环中对第二个数字串 `s2` 进行访问的索引位置。让我们来仔细解释一下：

在这个上下文中，`i` 是一个循环变量，表示当前循环的迭代次数。`n` 是两个数字串 `s1` 和 `s2` 中较大的长度。`n2` 是数字串 `s2` 的长度。

假设你正在计算 `s2` 中的某个位置，具体是 `s2` 的哪个位置呢？可以通过以下的方式来理解：

- `i`：当前循环的迭代次数，即从 `n` 到 `0` 的递减过程。
- `n`：两个数字串 `s1` 和 `s2` 中较大的长度。
- `n2`：数字串 `s2` 的长度。


现在，考虑以下情况：

- 当 `i` 较小时，`i` 代表的是数字串 `s1` 中的位置，而 `i - n + n2 - 1` 会是 `s2` 中的相应位置。
- 当 `i` 较大时，`i` 代表的是数字串 `s2` 中的位置，而 `i - n + n2 - 1` 会是 `s1` 中的相应位置。

这种计算方式使得在循环中能够同时访问两个数字串对应位置的数字。因此，`i - n + n2 - 1` 的作用是根据当前迭代次数 `i` 来确定在 `s2` 中的位置。

在 C++ 中，你可以用单引号 `' '` 来表示空格字符。空格字符的 ASCII 值是 32，因此你可以使用以下方式：


cpp

 Copy code

```
char space = ' ';
```

这样 `space` 就表示一个空格字符。你也可以在字符常量中直接使用空格，例如：

cpp

 Copy code

```
char space = ' ';
```

这里的 `' '` 就表示空格字符。

`while (1) {cin >> s >> sub}`这么写行么会造成死循环么

ChatGPT

是的，`while (1)` 表示一个无限循环，因为条件始终为真。在这种情况下，`cin >> s >> sub` 语句将在每次循环中尝试从输入流中读取两个字符串。如果输入有效，它将一直继续运行。如果输入流结束或者发生错误，它可能导致死循环。

如果你想在输入结束时退出循环，你可以考虑检查输入是否成功，例如：

cpp

Copy code

```
#include <iostream>
#include <string>

int main() {
    std::string s, sub;
    while (std::cin >> s >> sub) {
        // 在这里可以使用变量 s 和 sub 进行后续处理
        std::cout << "s: " << s << ", sub: " << sub << std::endl;
    }

    // 如果遇到输入结束或错误，循环会退出
    std::cout << "End of input." << std::endl;
```

#插入字符串

方法一

```
#include <iostream>
using namespace std;

int main() {
    string s, sub;

    while (cin >> s >> sub) {
        int maxAscii = 0;
        int maxIdx = -1;

        // 寻找 ASCII 值最大的字符的索引
        for (int i = 0; i < s.size(); i++) {
            if (static_cast<int>(s[i]) > maxAscii) {
```

```

        maxAscii = static_cast<int>(s[i]);
        maxIdx = i;
    }
}

// 如果找到最大字符，则在它的后面插入子串
if (maxIdx != -1) {
    s.insert(maxIdx + 1, sub);
} else {
    // 如果字符串为空，直接插入子串到开头
    s.insert(0, sub);
}

cout << s << endl;
}

return 0;
}

```

方法二

```

#include <iostream>
using namespace std;
#include <string>
int main()
{
    string s,sub;
    while(cin >> s>>sub)
    {
        int idx=0;
        for(int i=0;i<s.size();i++)
            if(s[i] > s[idx])    用于比较ascii值
            {
                idx= i;
            }
        s.insert(idx+1,sub);
        cout<<s<<endl;
    }
}

```

#带参数的构造函数

在C++中，创建对象的方式不仅限于 `MyClass a(10, "hello");` 这种形式。这只是使用带

有参数的构造函数创建对象的一种方式，还有其他的创建对象的方式，以下是一些示例：

1. 默认构造函数：

cppCopy code

```
MyClass a; // 使用默认构造函数创建对象
```

2. 带参数的构造函数：

cppCopy code

```
MyClass b(20, "world"); // 使用带参数的构造函数创建对象
```

3. 动态内存分配（使用 new）：

cppCopy code

```
MyClass* c = new MyClass(30, "dynamic"); // 动态创建对象
```

4. 使用 std::make_shared（用于共享指针的动态内存分配）：

cppCopy code

```
std::shared_ptr<MyClass> d = std::make_shared<MyClass>(40, "shared"); // 动态创建 shared_ptr
```

5. 使用 std::vector 或其他容器：

cppCopy code

```
std::vector<MyClass> vec; vec.push_back(MyClass(50, "vector")); // 创建对象并将其添加到向量中
```

这只是一些示例，根据程序的需求，可以采用不同的方式创建对象。 `MyClass a(10, "hello");` 这样的语法只是在有带参数构造函数时的一种常见方式。

#复制函数加析构函数

```
#include<iostream>
#include<cstring> // 添加头文件以使用strcpy
using namespace std;

class person {
private:
    char* name;
    int id;
```

```

public:
    person(char* name1, int id1) {
        id = id1;
        name = new char[strlen(name1) + 1];
        strcpy(name, name1);
    }

    person(const person& a) {
        id = a.id;
        name = new char[strlen(a.name) + 1];
        strcpy(name, a.name);
    }

    ~person() {
        delete[] name;
    }

    void display() {
        cout << name << " " << id << endl;
    }
};

int main() {
    char name[10]; // 修改为数组，用于存储输入的名字
    int id;

    cin >> name >> id;

    // 调用构造函数
    person a(name, id);
    // 调用拷贝构造函数
    person b = a;

    // 调用display函数
    a.display();
    b.display();

    return 0;
}

```

```

#include <iostream>
using namespace std;

class Complex {
private:
    double real;
    double imag;

public:
    // 默认构造函数
    Complex() : real(0), imag(0) {}

    // 带参数的构造函数
    Complex(double r, double i) : real(r), imag(i) {}

    void displayComplex() {
        if (real == 0 && imag > 1)
            cout << imag << "i" << endl;
        else if (imag == 0)
            cout << real;
        else if (real == 0 && imag < 0)
        {
            cout << imag << "i";
        }
        else if (imag > 1)
        {
            cout << real << "+" << imag << "i";
        }
        else if (real == 0 && imag == 0)
        {
            cout << endl;
        }
        else if (imag == 1)
        {
            cout << real << "+i";
        }
        else
            cout << real << imag << "i";

    }
};

int main() {

```

```
double inputReal, inputImag;  
cin >> inputReal >> inputImag;  
  
// 使用带参数的构造函数  
Complex c1(inputReal, inputImag);  
c1.displayComplex();  
  
return 0;  
}
```

#类对象数组和指针

- ▶ 将具有相同类类型的对象有序地集合在一起便构成了对象数组，以一维对象数组为例，其定义形式为：

类名 对象数组名[常量表达式];

- ▶ 一维对象数组有时也称为对象向量，它的每个元素都是相同类类型的对象。
- ▶ 例如表示平面上若干个点，可以这样定义：

Point points[100]; //表示100个点

► 关于对象数组的说明：

- (1) 在建立对象数组时，需要调用构造函数。如果对象数组有100个元素，就需要调用100次构造函数。
- (2) 如果对象数组所属类有带参数的构造函数时，可用初始化列表按顺序调用构造函数，使用复制初始化来初始化每个数组元素。
- (3) 如果对象数组所属类有单个参数的构造函数时，定义数组时可以直接在初值列表中提供实参。

```
Point P[3]={Point(1,2),Point(5,6),Point(7,8)}; //三个实参
Student S[5]={20,21,19,20,19}; //Student类只有一个数据成员
```

- (4) 对象数组创建时若没有初始化，则其所属类要么有合成默认构造函数（此时无其他的构造函数），要么定义无参数的构造函数或全部参数为默认参数的构造函数（此时编译器不再合成默认构造函数）。
- (5) 对象数组的初始化式究竟是什么形式，本质上取决于所属类的构造函数。因此，需要明晰初始化实参与构造函数形参的对应关系，避免出现歧义性。
- (6) 如果对象数组所属类含有析构函数，那末每当建立对象数组时，按每个元素的排列顺序调用构造函数；每当撤销数组时，按相反的顺序调用析构函数。

```
class Time {
public:
    Time(int h=0,int m=0,int s=0):
        hour(h),minute(m),second(s) { }
    void set(int h=0,int m=0,int s=0)
        { hour=h,minute=m,second=s; }
    int hour,minute,second;
};
Time now(12,0,0), *pt; //指向对象的指针变量
pt=&now; //指向对象
```

► 可以通过对象指针访问对象和对象的成员。如：

```
pt->set(13,13,0);
pt->hour=1;
```

- 对象的成员要占用存储空间，因此也有地址，可以定义指向对象成员的指针变量，一般形式为：

数据成员类型 *指针变量名=初值;

- 例如：

```
int *ptr=&now.hour; //指向对象数据成员的指针变量
```

- C++比C语言有更严格的静态类型，更加强调类型安全和编译时检查。
- 因此，C++的指针被分成数据指针、函数指针、数据成员指针、成员函数指针四种，而且不能随便相互转换。其中前两种是C语言的，称为普通指针（ordinary pointer）；后两种是C++专门为类扩展的，称为成员指针（pointer to member）。
- **成员指针与类的类型和成员的类型相关，它只应用于类的非静态成员。**由于静态类成员不是任何对象的组成部分，所以静态成员指针可用普通指针。

- 1. 数据成员指针

- 定义数据成员指针的一般形式为：

数据成员类型 类名::*指针变量名=成员地址初值;

- 例如：

```
class Data { //Data类
public:
    typedef unsigned int index; //类型成员
    char get() const; //成员函数
    char get(index st, index eb) const; //成员函数
    string content; //数据成员
    index cursor,top,bottom; //数据成员
};
```

- ▶ 指向content的指针的完全类型是“指向string类型的Data类成员的指针”，即：

```
String Data::*ps=&Data::content; //指向Data::content的成员指针
```

▶ 2.成员函数指针

- ▶ 定义成员函数的指针时必须确保在三个方面与它所指函数的类型相匹配：

- ▶ ①函数形参的类型和数目，包括成员是否为const。②返回类型。
- ▶ ③所属类的类型。

- ▶ 定义的一般形式为：

```
返回类型 (类名::*指针变量名)(形式参数列表)=成员地址初值;
```

- ▶ 或

```
返回类型 (类名::*指针变量名)(形式参数列表) const =成员地址初值;
```


- ▶ 例如 “char get() const”成员函数的指针可以这样定义和初始化：

```
char (Data::*pmf)() const = &Data::get; //指向Data::get()的成员指针
```

- ▶ 可以为成员指针使用类型别名，例如：


```
typedef char (Data::*GETFUNC)(Data::index, Data::index) const; //类型别名GETFUNC
```

- ▶ 这样指向get成员函数的指针的定义可以简化为：

```
GETFUNC pfget = &Data::get; //定义成员函数指针pfget
```

在这个语句中，`Data::index` 并没有被写两遍，而是定义了一个指向成员函数的指针类型。让我们逐步解释这个语句：

cpp

 Copy code

```
typedef char (Data::*GETFUNC)(Data::index, Data::index) const;
```

1. `typedef` 关键字用于创建新的类型别名。
2. `char (Data::*GETFUNC)(Data::index, Data::index) const;` 定义了一个指向成员函数的指针类型，该成员函数属于类 `Data`，接受两个参数，这两个参数的类型是 `Data::index`，返回类型是 `char`，并且这个成员函数是一个常量成员函数。
3. `GETFUNC` 是这个新类型的别名。

所以，`Data::index` 出现两次是因为该指针类型 `GETFUNC` 引用了 `Data` 类中的成员函数，这两个参数的类型都是 `Data::index`。这是用于指明成员函数的参数类型。



▶ 3.使用类成员指针

- ▶ ①通过对象成员指针引用（.*）可以从类对象或引用及成员指针间接访问类成员，或者通过指针成员指针引用（->.*）可以从指向类对象的指针及成员指针访问类成员。
- ▶ 对象成员指针引用运算符左边的运算对象必须是类类型的对象，指针成员指针引用运算符左边的运算对象必须是类类型的指针，两个运算符的右边运算对象必须是成员指针。

例如：

```
Data d, *p=&d; //指向对象d的指针
int Data::*pt = &Data::top; //pt为指向数据成员top的指针
int k = d.top; //对象成员引用，直接访问对象，直接访问成员，与下面等价
k = d.*pt; //对象成员指针引用，直接访问对象，间接访问成员
k = p->top; //指针成员引用，间接访问对象，直接访问成员，与下面等价
k = p->.*pt; //指针成员指针引用，间接访问对象，间接访问成员
char (Data::*pmf)(int,int) const; //pmf为成员函数指针
pmf = &Data::get; //指向有两个参数的get函数
char c1 = d.get(0,0); //对象直接调用成员函数，与下面等价
char c2 = (d.*pmf)(0,0); //对象通过成员函数指针间接调用成员函数
char c3 = (p->.*pmf)(0,0); //指针间接引用对象通过成员函数指针间接调用成员函数
```

- ▶ 除了静态成员函数外，每个成员函数都有一个额外的、隐含的形参this。在调用成员函数时，编译器向形参this传递调用成员函数的对象的地址。例如成员函数：

```
void Point::set(int a,int b) { x=a, y=b; } //成员函数定义
```

- ▶ 编译器实际上会重写这个函数为：

```
void Point::set(Point* const this,int a,int b)
{ this->x=a,this->y=b; }
```

- ▶ 当对象pt1调用MovePoint(2,2)函数时，即将pt1对象的地址传递给了this指针。MovePoint函数的原型应该是：

```
void MovePoint( Point *this, int a, int b);
```

- ▶ MovePoint函数中便成为：

```
void MovePoint(int a, int b) {this->x+=a; this->y+=b;}
```

- ▶ MovePoint函数体等价于：

```
pt1.x+=a; pt1.y+=b;
```

- ▶ 什么时候会用到this指针

- ▶ （1）在类的非静态成员函数中返回类对象本身的时候，直接使用return *this;
- ▶ （2）当参数与数据成员名相同时，如this->n = n （不能写成n=n）。

【例30.2】 this指针举例。

```
1 class point
2 {
3 public:
4     point(float x,float y)
5     {
6         this->x=x; //this->x 表示private中声明的 x; x 表示构造函数point(float x,float y)中的 x。
7         this->y=y; //this->y 表示private中声明的 y; y 表示构造函数point(float x,float y)中的 y。
8     }
9 private:
10     float x,y;
11 };
```

- ▶ this指针的const限定
- ▶ 假设Point类有getX这样一个非static函数：

```
double Point::getX();
```

- ▶ 编译以后形式如下：

```
double getX(Point *const this);
```

- ▶ 可以看出，this指针的指向不允许改变，所以this指针本身就是const指针。

- ▶ 如果成员函数是常函数也就是下面的定义：

```
double Point::getX()const;
```

- ▶ 编译后会变成：

```
double getX(const Point *const this);
```

- ▶ 可以看出，既不允许改变this指针的指向，也不允许改变this指向的内容。

#指针类示例

```
#include <iostream>

class MyClass {
public:
    void print() {
        std::cout << '1';
    }
};

int main() {
    MyClass a, b[2], *p[2];

    a.print(); // 输出：1

    for (int i = 0; i < 2; ++i) {
        b[i].print(); // 输出：11（因为有两个 b 对象）
    }
}
```

```

    p[0] = &a;
    p[1] = &b[0];

    p[0]->print(); // 输出: 1
    p[1]->print(); // 输出: 1

    return 0;
}

```

将指针数组 `p` 的元素指向不同的对象的目的是在程序中引入更灵活的指针管理，允许你通过循环或其他控制结构来处理不同的对象。

在这个特定的例子中，`p[0]` 指向了对象 `a`，而 `p[1]` 指向了数组 `b` 的第一个元素（`b[0]`）。这样做的一些可能原因包括：

1. **多态性 (Polymorphism)**：如果 `MyClass` 类是多态的基类，那么可以通过基类指针数组来存储不同派生类的对象。通过这种方式，可以通过基类指针来调用派生类中的相同名称的虚函数，实现多态。
2. **循环处理**：如果需要循环处理一组对象，可以使用指针数组来轻松访问不同的对象。这在处理动态数组或容器中的对象时特别有用。
3. **参数化**：如果有一组相关的对象，可以通过循环或其他控制结构，使用指针数组来对它们进行参数化处理，而不需要显式引用每个对象。
4. **动态创建**：在运行时，可以动态创建和释放对象，并通过指针数组来进行管理。

这只是一些可能的原因，具体取决于代码的上下文和设计。在实际应用中，指向不同对象的指针数组提供了更灵活的代码结构和对象管理。

#数组类

```

#include <iostream>
using namespace std;
const int m= 100; //数组中元素的最大个数
class Array
{
public:
    Array(); //初始化length=0
    int getlength(); //返回数组中元素实际个数
    void insertX(int i, double x); //在下标i处插入x
    void displa();
    //输出数组中实际元素

```

```

    int length;

private:
    double data[m]; //存储元素

    //数组中实际元素个数
};
Array::Array()
{
    length = 0;
}
int Array::getlength()
{
    return length;
}
void Array::insertX(int i, double x)
{
    if(length<m&&i>=0&&i<=length)
        for (int j = length; j > i; j--)
        {
            data[j] = data[j - 1];
        }
    data[i] = x;
    length++;
}
void Array::displa()
{
    for (int i = 0; i < length; ++i) {
        cout << data[i] << " ";
    }
    cout << endl;
}

int main()
{
    Array myarray;
    double input; int ge = 0;
    while (cin >> input && input != 0)
    {
        myarray.insertX(myarray.getlength(), input);
        ge++;
    }
    cout << "Length:" << ge;
}

```

```
cout << "Elements:";
myarray.displa();
}
```

在这个代码中主要是为了呈现插入，就是用类，输入一组数据。再用类输出，一组数据。

#类的作用域和生命周期

同变量，函数一样有自己的作用域。

- ▶ (1) 每个类都定义了自己的作用域和唯一的类型。在类体内声明类成员，将成员名引入类的作用域中。两个不同的类具有两个独立的类作用域。即使两个类具有完全相同的成员列表，它们也是不同的类型。每个类的成员不同于任何其他类的成员。例如：

```
class A { public: int i; double d; };
class B { public: int i; double d; };
class Data{public: int member; void memfunc();};
A obj1;
B obj2 = obj1; //错误，不同的类类型对象不能赋值
int a;
```

- ▶ (2) 在类作用域之外，成员只能通过对象、指针或引用的方式（使用成员访问操作符“.”或“->”）来访问。这些运算符左边的运算对象分别是一个类对象、指向类对象的指针或对象的引用，后面的成员名字必须在相对应的类的作用域中声明。

```
Data obj; //定义对象
Data *ptr=&obj , &r=obj; //定义指针和引用
a=member; //错误，member需要作用域限定
obj.member; //正确，通过对象使用member
a=ptr->member; //正确，通过指针使用member
r.memfunc(); //正确，通过引用使用memfunc
```


- ▶ (3) 静态成员、类中定义的类型成员需要直接通过类作用域运算符“::”来访问。

```
class Data {
public:
    enum COLORS {RED, GREEN, BLUE, BLACK, WHITE};
    //声明枚举类型
    COLORS getcolor();
};
COLORS c1; //错误, COLORS只能在类作用域中
Data::COLORS cc; //正确, Data:: 限定COLORS在类作用域中
```

▶ 2. 嵌套类

- ▶ 可以在类A的内部定义类B, 称类B为嵌套类 (nested class), 又称为嵌套类型 (nested type), 称类A为外围类 (enclosing class)。
- 例如:

```
class Queue { //外围类定义体
private:
    struct QueueItem { //嵌套类
        QueueItem(const int &);
        int item;
        QueueItem *next;
    };
    QueueItem *head, *tail;
};
```

▶ 说明:

- ▶ (1) 嵌套类是独立的类, 基本上与它们的外围类不相关, 因此, 外围类和嵌套类的对象是互相独立的。嵌套类型的对象不包含外围类所定义的成员, 同样, 外围类的成员也不包含嵌套类所定义的成员。
- ▶ (2) 嵌套类的名字只在其外围类的作用域中可见, 嵌套类的名字不会与另一作用域中声明的名字冲突。
- ▶ (3) 嵌套类可以具有与非嵌套类相同种类的成员。像任何其他类一样, 嵌套类使用访问标号控制对自己成员的访问。成员可以声明为public、private或protected。外围类对嵌套类的成员没有特殊访问权, 并且嵌套类对其外围类的成员也没有特殊访问权。

- ▶ (1) 局部类的所有成员（包括函数）必须完全定义在类体内。
- ▶ (2) 局部类可以访问的外围作用域中的名字是有限的。局部类只能访问在外围作用域中定义的类型名、静态变量和枚举成员，不能使用定义该类的函数中的变量。
- ▶ (3) 外围函数对局部类的私有成员没有特殊访问权，当然，局部类可以将外围函数设为友元。实际上，局部类中private成员几乎是不必要的，通常局部类的所有成员都为public成员。
- ▶ (4) 可以访问局部类的程序部分是非常有限的。局部类封装在它的局部作用域中，进一步通过信息隐藏进行封装通常是不必要的。

31.1 类的作用域

```
int a, v;
void fun(int v){
    static int s; //静态局部变量
    enum Loc { a=1024, b};
    class Bar { //局部类
    public:
        Loc locv; //正确，允许使用局部类型
        int barv;
        void setBar(Loc l=a) { //正确，默认参数值为枚举类型的枚举器a
            barv=v; //错误，v是函数fun的形参
            barv=:v; //正确，使用全局v
            barv=s; //正确，使用静态局部变量
            locv=b; //正确，使用枚举器b
        }
    };}
```


- ▶ 按生命期的不同，对象可分为如下四种：
 - ▶ （1）局部对象。
 - ▶ 局部对象在运行函数时被创建，调用构造函数；当函数运行结束时被释放，调用析构函数。
 - ▶ （2）静态局部对象。
 - ▶ 静态局部对象在程序执行函数第一次经过该对象的定义语句时被创建，调用构造函数。这种对象一旦被创建，在程序结束前都不会撤销。即使定义静态局部对象的函数结束时，静态局部对象也不会撤销。在该函数被多次调用的过程中，静态局部对象会持续存在并保持它的值。
 - ▶ 静态局部对象在程序运行结束时被释放，调用析构函数。
-

- ▶ （3）全局对象
 - ▶ 全局对象在程序开始运行时，main运行前创建对象，并调用构造函数；在程序运行结束时被释放，调用析构函数。
- ▶ （4）自由存储对象
 - ▶ 用new分配的自由存储对象在new运算时创建对象，并调用构造函数；在delete运算时被释放，调用析构函数。自由存储对象一经new运算创建，就会始终保持直到delete运算时，即使程序运行结束它也不会自动释放。

31.2 对象的生命期

【例31.1】

```
1 #include <iostream>
2 using namespace std;
3 class A {
4 public: int n;
5     A(int _n=10):n(_n) { cout<<"A("<<n<<")构造\t"; }
6     ~A() { cout<<"A("<<n<<")析构\t"; }
7 };
8 class B {
9 public: int m;
10    B(int _n=20):m(_n),a(_n) { cout<<"B("<<m<<")构造"<<endl; }
11    ~B() { cout<<"B("<<m<<")析构"<<endl; }
12    A a; //类类型对象
13 };
```

31.2 对象的生命期

```
14 B *gp, gb(30); //全局对象
15 void fun4()
16 {
17     static B b41(41); //静态局部对象
18     B b42(42); //局部对象
19     gp = new B(43); //自由存储对象
20 }
21 void fun5()
22 {
23     static B b51(51); //静态局部对象
24     B b52(52); //局部对象
25 }
```