

基础知识5

#类作用域const限定

- ▶ C++有不少措施保护数据的安全性，如private保护类的数据成员等。但对于一些共用的数据，如函数实参与形参等，我们可以在不同的场合通过不同的途径访问同一个数据对象。有时不经意的误操作会改变数据的值，而这是人们所不希望出现的。
- ▶ 既要使数据能在函数间共享，又要保证它不被任意修改，**可以使用const限定，即把数据定义为只读的。**

31.3 常对象、常数据成员、常成员函数

- ▶ 1. 常对象
- ▶ 在定义对象时使用const限定，称它为常对象，定义的一般形式为：

类名 **const** 对象名1(实参列表), 对象名2,.....;
- ▶ 或

const 类名 对象名1(实参列表), 对象名2,.....;
- ▶ 例如，如已定义Data类，定义常对象如下：

const Data d1; //定义常对象
Data **const** d2(10,20,100); //定义常对象

31.3 常对象、常数据成员、常成员函数

- ▶ 常对象中的数据成员均是const的，因此必须要有初值。无论什么情况下，常对象中的数据成员都不能被修改。
- ▶ 除了合成的默认构造函数和默认析构函数外，也不能调用常对象的非const型的成员函数。例如：

```
d1.data=10; //错误，常对象数据成员data为const，不能成为左值  
d2.show(); //错误，不能调用常对象中非const型成员函数
```

必须有初值

31.3 常对象、常数据成员、常成员函数

- ▶ 在实际编程中，有时一定要修改常对象中的某个数据成员的值，这时可以将数据成员声明为mutable（可变的）来修改它的值。声明形式为：

```
mutable 数据成员类型 数据成员名列表; //可变的数据成员声明
```

- ▶ 其中mutable为C++关键字，表示可变的数据成员。
- ▶ 例如：

```
mutable int data; //可变的数据成员
```

- ▶ 2. 常数据成员
- ▶ 在声明数据成员时使用const限定，称它为常数据成员。声明的一般形式为：

```
const 数据成员类型 数据成员名列表; //常数据成员声明
```

- ▶ 例如：

```
const int data; //常数据成员声明
```

- ▶ 无论是成员函数还是非成员函数都不允许修改常数据成员的值。
- ▶ 常数据成员只能通过构造函数初始化列表进行初始化。

只能通过构造函数进行初始化

▶ 3. 常成员函数

- ▶ 在定义成员函数时使用const限定，称它为常成员函数。定义的一般形式为：

```
class 类名 { //类体
    ...
    返回类型 函数名(形式参数列表) const //常成员函数定义
    {
        函数体
    }
    ...
};
```

- ▶ 或者：

```
class 类名 { //类体
    ...
    返回类型 函数名(类型1 参数名1,类型2 参数名2,...) const;
    ...
};
返回类型 类名::函数名(形式参数列表) const //常成员函数外部定义
{
    函数体
}
```

► 例如：

```
int getx() const; //类体声明常成员函数
int Data::getx() const //外部定义常成员函数
{
    return x;
}
```

► 需要注意const的位置在函数头和函数体之间，不要写成：

```
const 返回类型 函数名(类型1 参数名1,类型2 参数名2,...);
```

► 这种写法表示函数返回值只读（如返回只读引用）。

C++程序设计

31.3 常对象、常数据成员、常成员函数

- 无论声明还是定义常成员函数都要有const关键字。常成员函数可以访问const数据成员，也可以访问非const的数据成员。const数据成员可以被const成员函数访问，也可以被非const的成员函数访问。具体情况见下表：

表31-1 const限定访问关系

数据成员	非常成员函数	常成员函数
非常数据成员	允许访问，可以修改	允许访问，不能修改
常数据成员	允许访问，不能修改	允许访问，不能修改
常对象数据成员	不允许访问和修改	允许访问，不能修改

► 关于常成员函数的说明。

► (1) 在一个类中，如果有些数据成员的值允许修改，另一些数据成员的值不允许修改，那么可以将一部分数据成员声明为const（常数据成员），使得其值不能被修改。而普通的成员函数可以修改普通的数据成员，但只能访问常数据成员的值。

► (2) 如果要求所有数据成员的值都不允许改变，可以将对象声明为const的（常对象），那么只能用const成员函数访问数据成员，且不能修改其值。这样，数据成员无论如何也不会被修改。

► (3) 如果定义了一个常对象，只能调用其中的const成员函数，而不能调用非const成员函数。如果需要访问对象中的数据成员，可将常对象中所有成员函数都声明为const成员函数，但应确保在函数中不会修改对象中的数据成员。

► (4) 常对象中的成员函数不一定是常成员函数。如果在常对象中的成员函数未加const声明，C++把它作为非常成员函数处理。

► (5) 常成员函数不能调用另一个非常成员函数。

► 1. 指向对象的常指针

► 指向对象的常指针定义形式如下：

```
类名 * const 指针变量名=对象地址; //常指针
```

► 其含义是这样的指针始终保持其初值，程序中不能修改其指向。如：

```
Data d(10,20,100), d1;  
Data *const p=&d; //定义指向对象的常指针  
p=&d1; //错误，不能修改常指针的指针值
```

► 请注意，对象的常指针必须在定义时初始化，因为其后就不能再指向别的对象了。

常指针定义是就要有初始值

- ▶ 虽然常指针是const的不能改变指向，但常指针所指向的对象却不一定是const的。
- ▶ 通常，使用常指针作为函数的形参，目的是不允许在函数执行过程中改变指针变量的值，使其始终指向原来的对象。如果在函数执行过程中试图修改常指针形参的值，就会出现编译错误。

指向常对象的指针

- ▶ 2. 指向常对象的指针变量
- ▶ 指向常对象的指针变量定义形式如下：

```
const 类名 * 指针变量名;
```

- ▶ 其含义是指针变量指向的对象为const（即常对象）。例如：

```
const Data *p; //指向常对象的指针变量
```

- ▶ 指向常对象的指针变量，是不能通过它改变所指向的对象的值，但是指针变量本身的值是可以改变的，因此可以在定义时不初始化。

- ▶ 请注意指向对象的常指针变量与指向常对象的指针变量的区别：

```
Data *const p=&d; //定义指向对象的常指针  
const Data *p; //指向常对象的指针变量
```

- ▶ 如果一个对象已被声明为常对象，只能用指向常对象的指针变量指向它。
- ▶ 如果定义了一个指向常对象的指针变量，即使它指向一个非const的对象，其指向的对象也是不能通过指针来改变的。
- ▶ 指向常对象的指针常用作函数形参，目的是在保护形参指针所指向的对象，使它在函数执行过程中不被修改。

这个定义时不用初始化

▶ 3. 对象的常引用

▶ 在C++程序中，经常用对象的常指针和常引用作函数参数。这样既能保证数据安全，使数据在函数中不能被随意修改，又在调用函数时又不必传递实参对象的副本，大幅减少函数调用的空间和时间的开销。

▶ 对象常引用定义形式如下：

```
const 类名 & 引用变量名;
```

▶ 例如，复制构造函数就使用了常引用作为函数唯一形参：

```
Data(const Data& r) : x(r.x),y(r.y),data(r.data) { } //复制构造
```

【例31.2】

```
1 #include <iostream>
2 #include <string.h>
3 using namespace std;
4 class A{
5 public:
6     int getArea() const
7     {
8         return w*h;
9     }
10    void setWH(int x,int y)
11    {
12        w=x,h=y;
13    }
```

```
14    A(int x,int y){w=x,h=y;}
15    A(){}
16 private:
17    int w,h;
18 };
19 int main()
20 {
21     A a;//普通对象可以不初始化
22     a.setWH(3,9);
23     A const b;//错误，常对象必须声明的同时初始化，正确的是：
24                                     A const b(3,6);
25     b.setWH(3,7);//错误，常对象不能调用非常成员函数
26     cout<<a.getArea()<<endl<<b.getArea()<<endl;;
27     return 0;
28 }
```

在这个代码中就是那个setwh这个函数改变了值，所以这个函数有问题。

在结构体中，对于结构体对象用“.”，对于指针用“->”，

将形参改为指针，减少内存

如果是类访问就用::

如果是对象就是.

如果是指针就是->

如果运用域运算符，就是用了类，他俩是紧密相连的

类作用域运算符::

- ▶ 可以定义类的静态成员，能够实现同类的多个对象之间数据共享。

使用类的静态成员的优点是：

- ▶ ①静态成员的名字是在类的作用域中，因此可以避免与其他类的成员或全局对象名字冲突；
- ▶ ②静态成员可以实施封装，可以是私有成员，而全局对象不可以。
- ▶ ③静态成员是与特定类关联的，结构清晰。

- ▶ 静态数据成员是类的一种特殊数据成员，它以关键字static开始，声明形式为：

```
class 类名 { //类体
    ...
    static 数据成员类型 数据成员名列表; //静态数据成员声明
    ...
};
```


► 例如：

```
class Data { //Data类定义
public:
    static int count; //静态数据成员
    int maxlevel; //非静态公有数据成员
    Data(int i=0){ ..... , count++; } //构造函数
private:
    int level; //非静态私有数据成员
};
int Data::count=0; //静态数据成员定义且初始化
```

► count设计的目的是计数Data类总共有多少个实例化对象。

► 关于静态数据成员的说明：

- （1）通常，非静态数据成员存在于类类型的每个对象中，静态数据成员则独立于该类的任何对象，在所有对象之外单独开辟空间存储。在为对象所分配的空间中不包括静态数据成员所占的空间。
- （2）如果只声明了类而未定义对象，则类的非静态数据成员是不占存储空间的，只有在定义对象时，才为对象的数据成员分配空间。但是只要在类中定义了静态数据成员，即使不定义任何对象，也为静态数据成员分配空间，它可以在尚未建立对象时就被引用。

- ▶ (3) 访问静态成员时同样需要遵守公有及私有访问规则。
- ▶ (4) 静态数据成员必须在类外部定义一次（仅有一次），静态成员不能通过类构造函数进行初始化，而是在类外定义时进行初始化。定义静态数据成员的方式为：

数据成员类型 类名::静态数据成员名=初始化式;

- ▶ (5) 静态数据成员可用作默认实参，非静态数据成员不能用作默认实参，因为它的值不能独立于所属的对象而使用。例如：

```
class Data { //Data类定义
    ...
    Data& setbkcolor(int=bkcolor);
    static const int bkcolor = 5;
};
```

- ▶ (6) 有了静态数据成员，各对象之间实现了数据共享，因此可以不使用全局变量。

```
class test
{ private:
    int x;
    int y;
public:
    static int num;
    static int Getnum()
    {    x+=5; //错误, 静态成员函数不能调用非静态数据成员
        num+=15;
        return num;
    }
};
int test::num = 10;
```

```
int main(void)
{
    test a;
    cout<<test::num<<endl;           //输出10
    test::num = 20;
    cout<<test::num<<endl;           //输出20
    cout<<test::Getnum()<<endl;       //输出35
    cout<<a.Getnum()<<endl;           //输出50
    return 0;
}
```

- ▶和静态数据成员一样，静态成员函数是类的一部分，而不是对象的一部分。如果要在类外调用公有的静态成员函数，可以类作用域运算符 (::) 和通过对象名调用静态成员函数，例如：

```
cout<<Data::getcount()<<'\\t'<<d.getcount();
```

- ▶静态成员函数与非静态成员函数的根本区别是：非静态成员函数有this指针，而静态成员函数没有this指针。因此，静态成员函数不能访问本类中的非静态成员。**静态成员函数就是专门为了访问静态数据成员的。**
- ▶**静态成员函数不能被声明为const。**

32.2 静态成员函数

【例32.1】静态成员举例

```
1 #include <iostream>
2 using namespace std;
3 class CTest {
4 public:
5     CTest() { s_total++; id=s_total; cout<<"构造"<<id<<" ";}
6     ~CTest() { s_total--; cout<<"析构"<<id<<" "; }
7     static int gettotal() { return s_total; }
8 private:
9     static int s_total;
10    int id;
11 };
```

```
12 int CTest::s_total=0;
13 int main()
14 {
15     CTest a,b,c;
16     CTest *p=new CTest;
17     cout<<"合计="<<CTest::gettotal()<<" ";
18     delete p;
19     cout<<"合计="<<CTest::gettotal()<<" ";
20     return 0;
21 }
```

```

#include <iostream>
using namespace std;

class CTest {
public:
    CTest() {
        s_total++;
        id = s_total;
        cout << "构造" << id << " ";
    }

    static int gettotal() {
        return s_total;
    }

private:
    static int s_total;
    int id;
};


int CTest::s_total = 0;

int main() {
    CTest a, b, c;

    cout << "合计=" << CTest::gettotal() << endl;

    return 0;
}

```

 **创建对象**

这部分代码在创建 `CTest` 类的对象，包括两种方式：

1. **栈上创建对象**：通过声明 `CTest a, b, c;`，创建了三个对象 `a`、`b` 和 `c`，它们是在程序的栈上分配内存的局部变量。
2. **堆上动态分配对象**：通过 `CTest *p = new CTest;`，创建了一个对象，并将其地址分配给指针 `p`。这个对象是通过动态内存分配在堆上创建的，需要手动调用 `delete p;` 来释放这块内存。在这个例子中，没有释放动态分配的对象，这可能导致内存泄漏。

总体而言，这样的代码演示了两种创建对象的方式：栈上的局部对象和堆上的动态分配对象。栈上的对象在离开其作用域时会自动调用析构函数，而堆上的对象需要手动释放内存。



静态成员通常在以下情况下使用：

1. **共享数据**：如果你希望类的所有实例共享某个数据，而不是每个实例都有自己的一份，那么可以使用静态成员。例如，记录某个类的实例总数、共享某个配置信息等。

cppCopy code

```
class MyClass { public: static int instanceCount; // 所有实例共享的计数器
MyClass() { instanceCount++; // 在构造函数中增加实例计数 } ~MyClass() {
instanceCount--; // 在析构函数中减少实例计数 } }; // 在类外初始化静态成员 int
MyClass::instanceCount = 0;
```

2. **静态成员函数**：如果某个函数与类本身相关，而不涉及实例特定的数据，可以考虑将其声明为静态成员函数。静态成员函数只能访问静态成员和其他静态成员函数，而不能访问实例的非静态成员。

cppCopy code


```
class Utility { public: static double calculateSomething() { // 静态函数可以执
行与类本身相关的任务 // 但不能访问实例的非静态成员 return 42.0; } };
```

3. **全局性质**：如果你需要在整个程序的执行期间保持某种状态，而不依赖于类的实例，可以使用静态成员。静态成员在程序启动时创建，在程序结束时销毁，具有全局性质。
4. **类作用域**：静态成员属于整个类，而不是类的实例。这意味着你可以通过类名访问静态成员，而不需要创建类的实例。
5. **单例模式**：静态成员经常用于实现单例模式，确保类只有一个实例。

总的来说，使用静态成员的关键点是需要在整个类范围内共享某个状态或行为，而不是针对每个实例单独存在的情况。

1. `CTest` 类有一个静态成员变量 `s_total` 用于记录类的实例数量。
2. 构造函数和析构函数分别在实例创建和销毁时被调用，输出相应的信息。
3. 静态成员函数 `gettotal` 返回当前的合计数量。
4. 在 `main` 函数中，创建三个实例对象和一个动态分配的实例对象，然后输出当前的合计数量。
5. 删除动态分配的实例对象后，再次输出合计数量。

运行此程序，你会看到输出类似于以下内容：

 Copy code

```
构造1 构造2 构造3 构造4 合计=4 析构4 合计=3
```

- ▶ C++提供友元（friend）机制，允许一个类将其非公有成员的访问权授予指定的函数或类。友元的声明只能出现在类定义的内部的任何地方，由于友元不是授予友元关系（friendship）的那个类的成员，所以它们不受访问控制的影响。通常，将友元声明放在类定义的开始或结尾。
- ▶ 友元可以是普通的函数，或已定义的其他类的成员函数，或整个类。将一个函数设为友元，称为友元函数（friend function），将一个类设为友元，称为友元类（friend class）。友元类的所有成员函数都可以访问授予友元关系的那个类的非公有成员。
- ▶ 因此，访问类非公有成员可以有两个用户：类成员和友元。
- ▶ 如果在一个类以外的某个地方定义了一个函数，在类定义中用 `friend` 对其进行声明，此函数就称为这个类的友元函数。友元函数可以访问这个类中的私有成员。

【例32.7】友元函数举例

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4 class Point { //Point类
5 public:
6     Point(int _x=0,int _y=0) : x(_x), y(_y) { }
7 private:
8     int x, y; //私有数据成员
9     friend double distance(Point& r1, Point& r2); //友元函数
10 };
```

```
11 double distance(Point& r1, Point& r2) //计算两个点的距离
12 {
13     double x = r2.x>r1.x ? r2.x-r1.x : r1.x-r2.x; //访问Point
类私有成员
14     double y = r2.y>r1.y ? r2.y-r1.y : r1.y-r2.y; //访问Point
类私有成员
15     return sqrt(x*x+y*y);
16 }
17 int main()
18 {
19     Point a(1,1),b(5,5); //定义两个点
20     cout<<distance(a, b); //输出它们的距离
21     return 0;
22 }
```

【例32.7】友元成员函数举例

```
1 class B; //类的前向声明
2 class A { //A类
3 public:
4     A(int _a=0) : a(_a) { }
5     void setb(B& r);
6 private:
7     int a; //私有数据成员
8 };
9 class B { //B类
10 public:
11     B(int _b=0) : b(_b) { }
12 private:
13     int b; //私有数据成员
14     friend void A::setb(B& r);
15 };
```

32.5 友元函数

```
16 void A::setb(B& r)
17 {
18     r.b=a; //访问B的私有成员b
19 }
20 int main()
21 {
22     return 0;
23 }
```

► 关于友元类的说明：

- （1）友元的关系是单向的而不是双向的。如果声明了类B是类A的友元类，不等于类A是类B的友元类，类A中的成员函数不能访问类B中的私有数据。
- （2）友元的关系不能传递或继承，如果类B是类A的友元类，类C是类B的友元类，不等于类C是类A的友元类。如果想让类C是类A的友元类，必须显式地在类A中另外声明。
- 面向对象程序设计的一个基本原则是封装性和信息隐蔽，而友元却可以访问其他类中的私有成员，**突破了封装原则**。友元的使用有助于数据共享，能提高程序的效率，但也不要滥用，要在数据共享与信息隐蔽之间选择一个恰当的平衡点。

```
#include<iostream>
#include<string>
#include<cstdlib>
#define MAX 1000
using namespace std;
struct Person
{
    string name;
    int age;
    string sex;
    string phone;
    string addr;
};
struct addressbooks
{
    struct Person personarry[MAX];
    int m_Size;
};
void mainMenu()
{
```



```

        cout << "-----" << endl;
        cout<<"1.增加联系人" << endl;
        cout<<"2.显示联系人" << endl;
        cout<<"3.删除联系人" << endl;
        cout<<"4.查找联系人" << endl;
        cout<<"5.修改联系人" << endl;
        cout<<"6.清空联系人" << endl;
        cout<<"0.退出通讯录" << endl;
        cout << "-----" << endl;
    }
    void addPerson(addrbooks* abs)
    {
        if (abs->m_Size == MAX)
        {
            cout << "联系人已满，无法添加" << endl;
            return;
        }
        string name;
        cout << "请输入姓名" << endl;
        cin >> name;
        abs->personarry[abs->m_Size].name = name;
        string sex;
        cout << "请输入性别" << endl;
        cin >> sex;
        abs->personarry[abs->m_Size].sex = sex;
        int age;
        cout << "请输入年龄" << endl;
        cin >> age;
        abs->personarry[abs->m_Size].age = age;
        string phone;
        cout << "请输入电话" << endl;
        cin >> phone;
        abs->personarry[abs->m_Size].phone = phone;
        string addr;
        cout << "请输入地址" << endl;
        cin >> addr;
        abs->personarry[abs->m_Size].addr = addr;
        //更新通讯录人数
        abs->m_Size++;
        cout << "添加成功" << endl;
        system("pause");
        system("cls");
    }
}

```

```

void printPerson(addrassbooks* abs)
{
    if (abs->m_Size == 0)
    {
        cout << "当前记录为空" << endl;
    }
    else
    {
        for (int i = 0; i < abs->m_Size; i++)
        {
            cout << "姓名\t" << abs->personarry[i].name << endl;
            cout << "性别\t" << abs->personarry[i].sex << endl;
            cout << "年龄\t" << abs->personarry[i].age << endl;
            cout << "电话\t" << abs->personarry[i].phone << endl;
            cout << "地址\t" << abs->personarry[i].addr << endl;
            cout << "\n";
        }
    }
    system("pause");
    system("cls");
}

int checkPerson(addrassbooks* abs, string name)
{
    for (int i = 0; i < abs->m_Size; i++)
    {
        if (abs->personarry[i].name == name)
        {
            return i;
        }
    }
    return -1;
}

void deletePerson(addrassbooks* abs)
{
    string dname;
    cout << "请输入你要删除的人名" << endl;
    cin >> dname;
    int ret = checkPerson(abs, dname);
    if (ret == -1)
    {
        cout << "查无此人" << endl;
    }
    else

```

```

    {
        for (int i = ret; i < abs->m_Size; i++)
        {
            abs->personarry[i] = abs->personarry[i + 1];
        }
        abs->m_Size--;
        cout << "删除成功" << endl;
    }
    system("pause");
    system("cls");
}

void findPerson(addrassbooks* abs)
{
    string fname;
    cout << "请输入要查找的联系人姓名" << endl;
    cin >> fname;
    int result = checkPerson(abs, fname);
    if (result == -1)
    {
        cout << "查无此人" << endl;
    }
    else
    {
        cout << "姓名\t" << abs->personarry[result].name << endl;
        cout << "性别\t" << abs->personarry[result].sex << endl;
        cout << "年龄\t" << abs->personarry[result].age << endl;
        cout << "电话\t" << abs->personarry[result].phone << endl;
        cout << "地址\t" << abs->personarry[result].addr << endl;
    }
    system("pause");
    system("cls");
}

void modifyPerson(addrassbooks* abs)
{
    string mname;
    cout << "请输入要修改的联系人姓名" << endl;
    cin >> mname;
    int result = checkPerson(abs, mname);
    if (result == -1)
    {
        cout << "查无此人" << endl;
    }
    else

```

```

{
    string name;
    cout << "请输入姓名" << endl;
    cin >> name;
    abs->personarry[result].name = name;
    string sex;
    cout << "请输入性别" << endl;
    cin >> sex;
    abs->personarry[result].sex = sex;
    int age;
    cout << "请输入年龄" << endl;
    cin >> age;
    abs->personarry[result].age = age;
    string phone;
    cout << "请输入电话" << endl;
    cin >> phone;
    abs->personarry[result].phone = phone;
    string addr;
    cout << "请输入地址" << endl;
    cin >> addr;
    abs->personarry[result].addr = addr;
    cout << "修改成功" << endl;
}
system("pause");
system("cls");
}
void cleanPerson(addrassbooks*abs)//逻辑清空
{
    abs->m_Size = 0;
    cout << "通讯录清空成功！" << endl;
    system("pause");
    system("cls");
}
int main(void)
{
    //创建通讯录结构体变量
    addrassbooks abs;
    //初始化通讯录中当前人员的个数
    abs.m_Size = 0;

    int select = 0;
    while (1)

```

```
{  
    mainMenu();  
    cin >> select;  
    switch (select)  
    {  
    case 1://添加联系人  
        addPerson(&abs);  
        break;  
    case 2://显示联系人  
        printPerson(&abs);  
        break;  
    case 3://删除联系人  
        deletePerson(&abs);  
        break;  
    case 4://查找联系人  
        findPerson(&abs);  
        break;  
    case 5://修改联系人  
        modifyPerson(&abs);  
        break;  
    case 6://清空联系人  
        cleanPerson(&abs);  
        break;  
    case 0://退出通讯录  
        cout << "欢迎下次使用" << endl;  
        system("pause");  
        return 0;  
    default:  
        break;  
    }  
}  
}
```


在main中或者其他函数中叫局部变量，不在函数中的变量叫全局变量。

```
int* func()
{
    int a = 10; //局部变量 存放在栈区，栈区的数据在函数执行完后自动释放
    return &a; //返回局部变量的地址
}

int main() {
    //接受func函数的返回值
    int * p = func();

    cout << *p << endl; //第一次可以打印正确的数字，是因为编译器做了保留
    cout << *p << endl; //第二次这个数据就不再保留了

    system("pause");
}
```

栈区：不要返回局部变量的地址，执行完之后数据就被销毁掉。

堆区：由程序员开辟和释放。

实例化：通过一个类，创建一个对象。

类中的属性和行为统一称为成员。

属性：成员属性，成员变量。

行为：成员函数，成员方法。

struct默认权限为共有，类的默认权限为私有。

使用引用作为函数参数有几种常见的情况：

1. **避免拷贝开销：** 当传递大型对象时，通过引用传递可以避免不必要的拷贝操作，提高程序的性能。拷贝大型对象可能会导致额外的开销，而通过引用传递只是传递对象的地址而已。

```
void processLargeObject(const LargeObject& obj) { // 处理大型对象的代码 }
```

2. **修改实参的值：** 通过引用传递参数，可以在函数内修改实参的值。这是因为引用提供了对实际参数的别名，对引用的修改会反映到实际参数上。

```
void modifyValue(int& value) { value = value * 2; }
```

3. **返回多个值：** 引用参数可以用于返回多个值，通过在函数参数中传递引用，函数可以修改传递的对象，从而使得调用者能够获取修改后的值。

```
void calculateSumAndProduct(int a, int b, int& sum, int& product) { sum = a + b; product = a * b; }
```

4. **对象的状态修改：** 如果函数需要修改传递对象的状态，而不仅仅是读取对象的值，使用引用是更合适的选择。

```
void updateObjectState(MyObject& obj) { // 修改对象状态的代码 }
```

总体来说，使用引用作为函数参数可以提高程序的效率，并允许函数对实参进行修改。然而，需要小心引用悬垂的问题，确保引用指向的对象在函数执行期间仍然有效。用原始的数据，就不会再拷贝一副数据，这样可以提高效率，所以用&

#继承派生

派生类，不能访问私有成员。但是派生类可以访问保护成员。

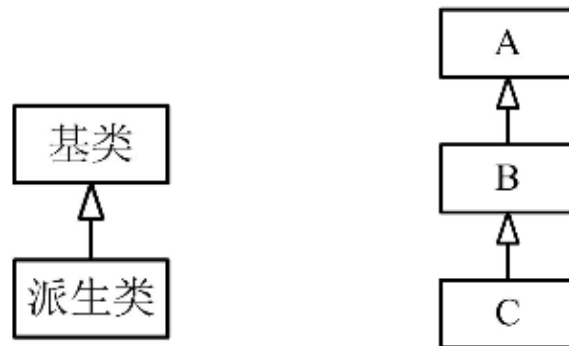
是的，您说得对。在面向对象的编程中，如果一个类派生自另一个类，那么派生类通常有访问其基类的成员的权限。私有成员是指只能在类内部访问的成员，而保护成员则是指可以在类内部和派生类中访问的成员。

具体来说，派生类不能直接访问基类的私有成员，但可以通过基类提供的公共或保护的成员函数来间接访问。而对于保护成员，派生类可以直接访问它们，这使得派生类能够继承并重用基类的一些功能。

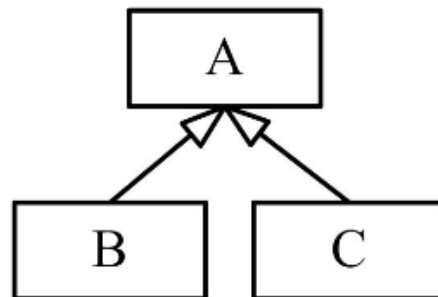
这种访问权限的设计有助于封装和继承的概念，同时保护基类的内部实现细节。这也使得在派生类中可以扩展或修改基类的功能，而不必暴露所有的实现细节。

- ▶ **继承**（inheritance）是面向对象程序设计的一个重要特性，是软件复用（software reusability）的一种重要形式。
- ▶ **继承允许在原有类的基础上创建新的类**，新类可以从一个或多个原有类中继承数据成员和成员函数，并且可以重新定义或增加新的成员，从而形成类的层次。继承具有传递性，不仅支持系统的可重用性，而且还促进系统的可扩充性。
- ▶ 在C++中，**继承就是在一个已存在的类的基础上建立一个新的类**。已存在的类称为**基类**（base class），又称为**父类**；新建立的类称为**派生类**（derived class），又称为**子类**。
- ▶ **一个新类从已有的类那里获得其特性，这种现象称为类的继承。**
- ▶ 另一方面，从已有的父类产生一个新的子类，称为**类的派生**。派生类继承了基类的所有数据成员和成员函数，具有基类的特性，派生类还可以对成员作必要的增加或调整，定义自己的新特性。

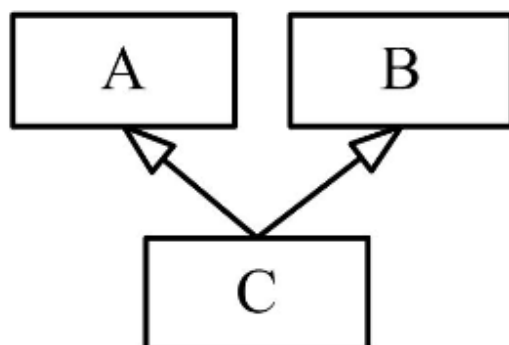
- ▶ 一个基类可以派生出多个派生类，每一个派生类又可以作为基类再派生出新的派生类，因此基类和派生类是相对而言的。
- ▶ 派生分为为单级派生和多级派生。



- ▶ 一个派生类可以只从一个基类派生，称为单继承（single inheritance），这是最常见的继承形式，如图所示，类B和类C都只从类A派生。



- ▶ 一个派生类有两个及两个以上的基类，称为**多重继承**（multiple inheritance），如图所示，类C从类A和类B派生。



- ▶ 基类与派生类之间的关系为：
 - ▶ （1）**基类是对派生类的抽象，派生类是对基类的具体化**。基类抽取了它与派生类的公共特征，而派生类通过增加信息将抽象的基类变为某种具体的类型。
 - ▶ （2）派生类是基类的组合，可以把多重继承看作是多个单一继承的简单组合。

- ▶ 定义派生类的一般形式为：

```
class 派生类名 : 类派生列表 { //类体  
    成员列表  
};
```

- ▶ 除了增加类派生列表外，派生类的定义与类定义并无区别。
- ▶ **类派生列表**（class derivation list）指定了一个或多个基类（base class），具有如下形式：

```
访问标号 基类名
```

33.2 派生类的定义

- ▶ 类派生列表可以指定多个基类，中间用逗号（，）间隔，基类名必须是已定义的类的名字。
- ▶ 访问标号表示继承方式，可以是public（公有继承）、protected（保护继承）或private（私有继承），**继承方式决定了对继承成员的访问权限**。如果未给出访问标号则默认为private（私有继承）。
- ▶ 派生类的成员列表描述的是派生类自己新增加的数据成员和成员函数。

33.2 派生类的定义

【例33.1】派生类举例


```
1 #include <iostream>
2 using namespace std;
3 class Base //基类
4 {
5 private:
6     int b_number;
7 public:
8     Base( ){} //基类不带参数的构造函数
9     Base(int i) : b_number (i) { } //基类带参数的构造函数
10    int get_number( ) {return b_number;}
11    void print( ) {cout << b_number << endl;}
12 };
13 class Derived : public Base //派生类
14 {
15 private:
16     int d_number; //派生类增加的数据成员
```


33.2 派生类的定义

```
17 public:
18     Derived(int i,int j):Base(i),d_number(j){}; //派生类构造函数
19     void print( ) //派生类增加的成员函数
20     {
21         cout << get_number() << " "; //派生类继承的成员函数
22         cout << d_number << endl;
23     }
24 };
25 int main( )
26 {
27     Base a(2);    Derived b(3, 4);
29     cout << "a is "; a.print( );    //基类的print
31     cout << "b is "; b.print( );    //派生类的print
33     cout << "base part of b is ";
34     b.Base::print( );    //基类的print
35     return 0;
36 }
```

这些修饰符用于限定派生类对基类成员的访问权限。当你在定义派生类时，可以使用这些修饰符来指定继承的成员的访问权限。例如：

cpp

 Copy code

```
class Base {
public:
    int publicMember;
protected:
    int protectedMember;
private:
    int privateMember;
};

class Derived : public Base {
    // 在 Derived 中可以访问 publicMember 和 protectedMember
    // 不能直接访问 privateMember
};
```

► 派生类的说明：

- (1) 用作基类的类必须是已定义的，其原因是显而易见的：每个派生类包含并且可以访问其基类的成员，为了使用这些成员，派生类必须知道它们是什么。这一规则说明**不可能从类自身派生出一个类**。

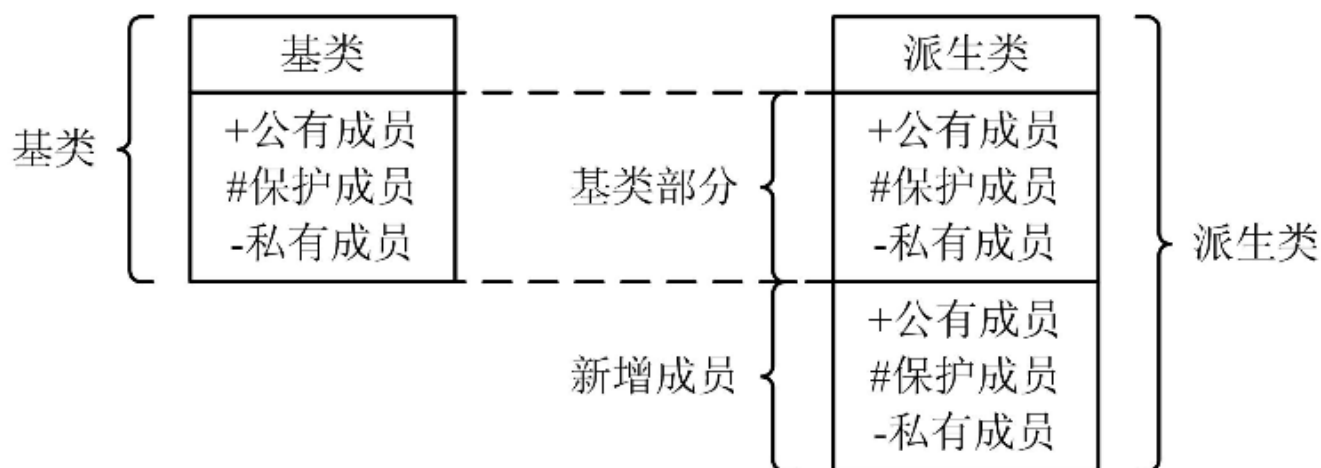
- (2) 如果需要声明（但并不实现）一个派生类，则只需要声明类名即可。例如：

```
class Derived : public Base; //错误，前向声明会导致编译错误

class Base;
class Derived; //正确
```

- (3) 显然，在一个派生类中，其成员由两部分构成：一部分是从基类继承得到的，另一部分是自己定义的新成员，所有这些成员仍然分为public（公有）、private（私有）和protected（保护）三种访问属性。
- (4) **友元关系不能继承**。一方面，基类的友元对派生类的成员没有特殊访问权限。另一方面，如果基类被授予友元关系，则只有基类具有特殊访问权限，该基类的派生类不能访问授予友元关系的类。
- (5) 如果基类定义了静态成员，则整个继承层次中只有一个这样的成员。无论从基类派生出多少个派生类，**每个静态成员只有一个实例**。
- (6) 静态成员遵循常规访问控制：如果静态成员在基类中为私有的，则派生类不能访问它。如果该静态成员在基类是公有的，则基类可以访问它，派生类也可以访问它。
- (7) 一般地，可以使用作用域运算符 (::) 也可以使用对象成员引用运算符 (.) 或指针成员引用运算符 (->) 访问静态成员。

图33.1 派生类的构成



- ▶ 实际编程中，设计一个派生类包括4个方面的工作：
- ▶ （1）从基类接收成员。
- ▶ 除了构造函数和析构函数，派生类会把基类全部的成员继承过来。这种继承是没有选择的，不能选择接收其中一部分成员，而舍弃另一部分成员。
- ▶ （2）调整基类成员的访问。
- ▶ 派生类接收基类成员是程序员不能选择的，但是程序员可以对这些成员作出访问策略。
- ▶ （3）修改基类成员
- ▶ 可以在派生类中声明一个与基类成员同名的成员，则派生类中的新成员会覆盖基类的同名成员，就实现了修改基类成员功能的效果。
- ▶ （4）在定义派生类时增加新的成员。
- ▶ 一般还应当自己定义派生类的构造函数和析构函数。

【例33.2】派生类举例

```

1  #include <iostream>
2  using namespace std;
3  class Base //基类
4  {
5  private:
6      int b_number;
7  public:
8      Base( ){} //基类不带参数的构造函数
9      Base(int i) : b_number (i) { } //基类带参数的构造函数
10     int get_number( ) {return b_number;}
11     void print( ) {cout << b_number << endl;}
12 };

```

```

13 class Derived : public Base //派生类
14 {
15 private:
16     int d_number; //派生类增加的数据成员
17 public:
18     Derived(int i,int j):Base(i),d_number(j){};
19     //派生类构造函数
20     void print()//派生类增加的成员函数
21     {
22         cout << get_number() << " "; //派生类继承的成员函数
23         cout << d_number << endl;
24     }
25 };

```

```

25 int main( )
26 {
27     Base a(2);
28     Derived b(3, 4);
29     cout << "a is ";
30     a.print( ); //基类的print
31     cout << "b is ";
32     b.print( ); //派生类的print

```

#派生类

```

#include <iostream>
#include <string>

```

```
// 基类 Animal
class Animal {
public:
    // 构造函数
    Animal(std::string name) : name(name) {}

    // 公有成员函数
    void speak() {
        std::cout << "Animal speaks" << std::endl;
    }

protected:
    // 保护成员变量
    std::string name;
};

// 派生类 Dog
class Dog : public Animal {
public:
    // 派生类构造函数
    Dog(std::string name, std::string breed) : Animal(name), breed(breed) {}

    // 新的公有成员函数
    void bark() {
        std::cout << "Dog barks" << std::endl;
    }

private:
    // 派生类私有成员变量
    std::string breed;
};

int main() {
    // 创建 Animal 对象
    Animal genericAnimal("Generic Animal");
    genericAnimal.speak();

    // 创建 Dog 对象
    Dog myDog("Buddy", "Golden Retriever");

    // Dog 类继承了 Animal 类的 speak 方法
    myDog.speak();
}
```

```
// Dog 类拥有自己的方法 bark
myDog.bark();

return 0;
}
```

```
Dog(std::string name, std::string breed) : Animal(name), breed(breed) {}
```

在这个行内，我们定义了 `Dog` 类的构造函数，它有两个参数：`name` 和 `breed`。然后，在冒号后面的初始化列表中，我们指定了如何初始化基类 `Animal` 的部分和派生类 `Dog` 的部分。

- `: Animal(name)`: 这部分表示调用基类 `Animal` 的构造函数，将参数 `name` 传递给它，以初始化基类的部分。这确保了基类 `Animal` 的构造函数被正确调用，使基类的属性得到初始化。
- `, breed(breed)`: 这部分是初始化派生类 `Dog` 的成员变量 `breed`。我们使用传递给构造函数的参数 `breed` 来初始化派生类的部分。

总体而言，这个初始化列表确保了在进入构造函数的函数体之前，基类和派生类的成员变量都被正确初始化。这种方式比在构造函数的函数体内初始化成员变量更高效，因为它避免了额外的赋值操作

运行上述C++代码的结果应该如下：

Copy code

```
Animal speaks Animal speaks Dog barks
```

让我解释一下每一步的执行过程：

1. 创建了一个 `Animal` 对象 `genericAnimal`，并调用其 `speak` 方法。输出 "Animal speaks"。
2. 创建了一个 `Dog` 对象 `myDog`，在构造函数中调用基类 `Animal` 的构造函数来初始化 `name`，并且初始化了 `breed`。然后调用 `speak` 方法，输出 "Animal speaks"。最后，调用 `bark` 方法，输出 "Dog barks"。

总结起来，`Dog` 类既能访问基类 `Animal` 的公有成员函数（如 `speak`），也能调用自己添加的成员函数（如 `bark`）。这是继承和派生类的基本概念，使得派生类能够重用基类的功能并添加自己的特有功能

#赋值兼容规则

赋值兼容规则是指在需要基类对象的任何地方，都可以使用公有派生类的对象来替代。通过公有继承，派生类得到了基类中除构造函数、析构函数之外的所有成员。这样，公有派生类实际就具备了基类的所有功能，凡是基类能解决的问题，公有派生类都可以解决。赋值兼容规则中所指的替代包括以下的情况！

1派生类的对象可以赋值给基类对象；

2派生类的对象可以初始化基类的引用

3派生类对象的地址可以赋给指向基类的指针

```
class Base { }; //基类
class Derive : public Base { }; //公有派生类
Base b,*pb; //定义基类对象、指针
Derive d; //定义派生类对象
```

▶这时，支持下面三种操作：

```
b=d; //派生类对象赋值给基类，复制基类继承部分
Base &rb=d; //基类引用到派生类对象
pb=&d; //基类指针指向派生类对象
```

▶赋值兼容规则是C++多态性的重要基础之一。

【例34.2】赋值兼容规则举例1

```
1 #include <iostream>
2 using namespace std;
3 class man //基类man
4 {
5     protected:
6         int age;
7         string name;
8     public:
9         man(int a,string n):age(a),name(n){ } //基类的构造函数
10        void print()
11        {cout<<"姓名: "<<name<<"，年龄: "<<age<< endl;}
12        int get_age(){return age;}
13        string get_name(){return name;}
14 };
```



```

15 class superman:public man //派生类superman
16 {
17 private:
18     int force_value; //派生类增加的数据成员
19 public:
20     superman(int a,string n,int f)
21         :man(a,n),force_value(f){ };//派生类的构造函数
22     void print( )
23     {
24         cout<<"姓名: "<<name<<"， 年龄: "<<age<<"武力值: "
25             <<force_value<<endl;
26     }
27     void fly(){cout<<"fly,fly,fly....."<<endl;}
28     //派生类增加的成员函数
29 };

```

```

30 int work(man a) //形参为基类对象
31 {
32     if(a.get_age()>=20) return 1;
33     else return 0;
34 }
35 int main( )
36 {
37     int n;
38     man i(25,"jerry");
39     superman j(20,"clark",100);
40     n=work(j); //实参为派生类对象
41     if(n!=0) cout<<j.get_name()<<"可以胜任此项工作。"<<endl;
42     else cout<<j.get_name()<<"不可以胜任此项工作。"<<endl;
43     return 0;
44 }

```

```

26 int work(man *a) //形参为基类指针
27 {
28     if(a->get_age()>=20) return 1;
29     else return 0;
30 }
31 int main( )
32 {
33     int n;
34     man i(25,"jerry");
35     superman j(20,"clark",100);
36     n=work(&j); //实参为派生类对象的地址
37     if(n!=0) cout<<j.get_name()<<"可以胜任此项工作。"<<endl;
38     else cout<<j.get_name()<<"不可以胜任此项工作。"<<endl;
39     return 0;
40 }

```

```

26 int work(man &a) //形参为基类引用
27 {
28     if(a.get_age()>=20) return 1;
29     else return 0;
30 }
31 int main( )
32 {
33     int n;
34     man i(25,"jerry");
35     superman j(20,"clark",100);
36     n=work(j); //实参为派生类对象
37     if(n!=0) cout<<j.get_name()<<"可以胜任此项工作。"<<endl;
38     else cout<<j.get_name()<<"不可以胜任此项工作。"<<endl;
39     return 0;
40 }

```
