

C++ 程序设计与结构

北京邮电大学

网络与交换技术国家重点实验室

宽带网研究中心

主要内容

- 作用域与可见性
- 对象的生存期
- 数据与函数
- 静态成员
- 友元
- 常量类型

标识符的作用域

➤ 最小作用域优先原则（与 C 语言一样）

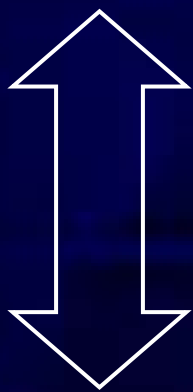
- 在使用一个标识符时，若不限定，编译程序将其理解为在有效作用域内同名标识符中，作用域最小的一个

函数原型的作用域

函数原型中的参数，其作用域始于
"(", 结束于")"。

➤ 例如，设有下列原型声明：

```
double Area(double radius);
```



radius 的作用域仅在于此，不能用于程序正文其它地方，因而可有可无。

```
double Area(double);
```

程序块作用域

- 在块中声明的标识符，其作用域自声明处起，限于块中（与C语言相同），例如：

```
void fun(int a)
{
    int b(a);
    cin>>b;

    if (b>0)
    {
        int b;
        int c;
        b = 1
        .....
    }
}
```

b c 的作用域

b 的作用域

类作用域

➤ 在 C++ 中引入，适用于 class 和 struct 定义类

➤ 类作用域作用于特定的类的成员

□ 成员变量和成员函数

➤ 类 X 的成员 M 具有类作用域：

□ 在 X 的成员函数中可直接访问

✉ 如果函数中声明了与 M 同名的
局部标识符，那么需要加限定符

□ 对外部，在对象 X 的作用域内

✉ 通过表达式访问 public 成员

x.M x.X::M

ptr->M ptr->X::M

✉ 成员的作用域与对象变量 x
的作用域相同

```
void f2();  
class X  
{  
    public:  
        int M;  
        void f1();  
        void f2();  
    public:  
        void abc(void)  
        {  
            M = 1;  
            f1();  
            {  
                int M;  
                M = 2;  
                X::M = 3; //或 this->M  
                f2();      // X::f2()  
                ::f2();    // 全局 f2()  
            }  
        }  
};
```

文件作用域

➤ 文件作用域

- ❑ 不在前述各个作用域中出现的标识符的声明
- ❑ 标识符的作用域开始于声明点，结束于文件尾

```
#include <iostream>
```

```
void func() ←
```

```
{
```

```
    i++;
```

```
}
```

```
int i = 0; ←
```

```
void func2() ←
```

```
{
```

```
    i++;
```

```
}
```

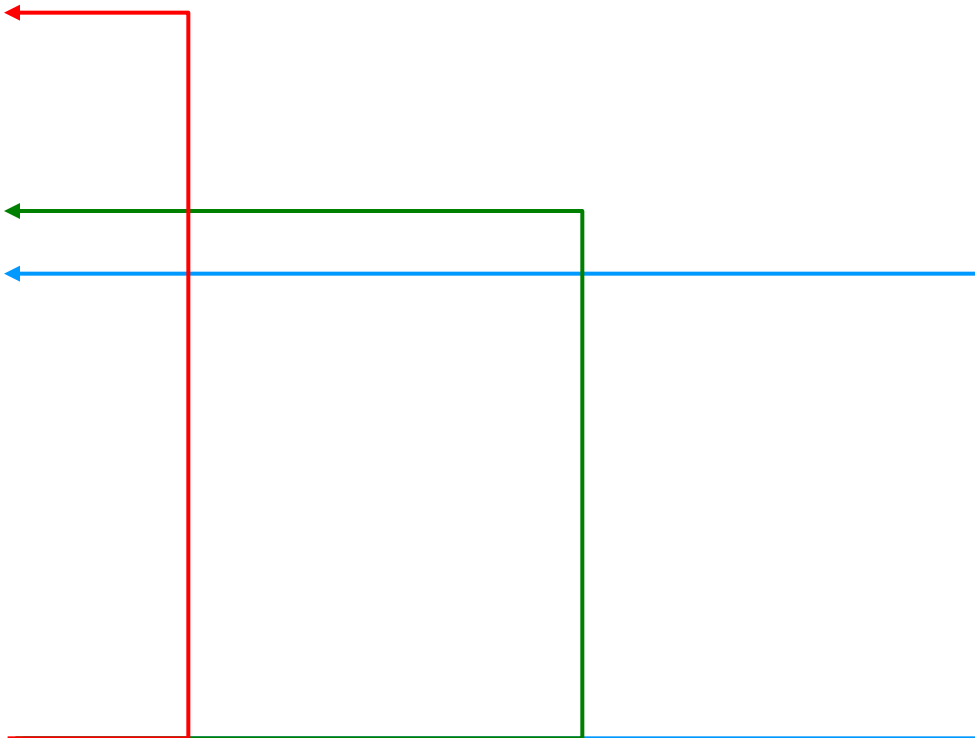
```
int main()
```

```
{
```

```
    i++;
```

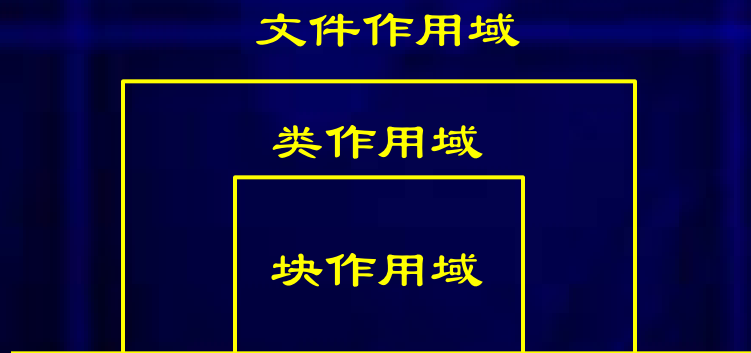
```
    ...
```

```
}
```



可见性

- 可见性是从对标识符的引用的角度来谈的概念
- 可见性表示从内层作用域向外层作用域“看”时能看见什么。
- 如果标识在某处可见，则就可以在该处引用此标识符。



可见性

- 标识符应声明在先，引用在后。
- 如果某个标识符在外层中声明，且在内层中没有同一标识符的声明，则该标识符在内层可见。
- 对于两个嵌套的作用域，如果在内层作用域内声明了与外层作用域中同名的标识符，则外层作用域的标识符在内层不可见。

同一作用域中的同名标识符

- 在同一作用域内的对象名、函数名、枚举常量名会隐藏同名的类名或枚举类型名。
- 重载的函数可以有相同的函数名。

作用域

```
#include<iostream>
using namespace std;
int i=1;      //文件作用域

int main()
{
    int i=5;      //块作用域, 隐藏了全局变量i
    cout << "i=" << i; //输出5

    {
        int i=7;      //块作用域, 隐藏了外层的i
        cout <<"i=" << i << endl; //输出7
    }

    cout <<"i=" << ::i << endl; //输出1,全局变量
    ...
}
```

对象的生存期

- 与基本类型的变量的生存期是一样的
 - ❑ 基本类型（如 int 等）的变量在 C++ 中也认为是一种对象
- 对象从创建到撤销的这段时间就是它的生存期
- 在对象生存期内，可以对对象进行操作(若其可见)
 - ❑ 如修改、更新成员变量（若允许）
 - ❑ 调用成员函数
- 没有操作时，对象将一直保持它的状态值

静态生存期

➤ 这种生存期与程序的运行期相同

□ 尽管与程序生命周期相同，但并非在程序的任何地方都是可见的

➤ 在文件作用域中声明的对象（全局变量）具有这种生存期

➤ 在函数内部声明静态生存期对象，要冠以关键字 `static`

➤ 对象的生存期结束时（程序结束时），将调用该对象的析构函数（若存在）

静态生存期

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

int i;                                //文件作用域，具有静态生存期

void foo(void)
{
    static int j=0;  //块作用域，具有静态生存期
    j++;
}
int main()
{
    i++;
    foo();
    return 0;
}
```

动态生存期

► 动态生存期的对象

- ❑ 在块作用域中声明的，用auto修饰、或者无修饰的对象
- ❑ 也称局部生存期对象
- ❑ 生存期开始于程序执行到声明点时，结束于标识符的作用域结束处
- ❑ 动态对象的生存期结束时，将调用该对象的析构函数（若存在）

例：对象的生存期与可见性

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

int i=1;                // i 为全局变量，具有静态生存期
int main()
{
    static int a=0;      // 静态局部变量，有全局寿命，局部可见。
    int b=-10;           // b为局部变量，具有动态生存期。

    i=i+10;
    void other(void);
    other();
    ...
}

void other(void)
{
    static int b, a=2;    // a,b为静态变量，全局寿命，局部可见，第一次进入函数时初始化
    int c=10;            // c 为局部变量，具有动态生存期，每次进入函数时都初始化。

    b = (a+=2);
    ...
}
```


例: 对象的生存期与可见性

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

class Clock          //时钟类声明
{
    ...              // Clock类 的定义
};

Clock globClock;     //声明对象globClock, 具有静态生存期, 文件作用域

void func()
{
    Clock myClock(globClock);    //创建具有块作用域的对象 myClock
    myClock.ShowTime();          //引用具有块作用域的对象
}

int main()
{
    globClock.SetTime(8,30,30);  //引用全局作用域的对象:
    func();
}
```



MyClock 在生存期结束时被销毁

数据与函数的关系

➤ 若需要处理的数据存储在局部对象中，通过参数传递实现共享

□ 在函数（过程）之间的传递参数，返回结果

➤ 若数据存储在全局对象中

□ 各函数（过程）对全局对象进行处理，结果也保存在全局对象中

➤ 将数据和使用数据的函数封装在类中

面向
过程

面向
对象

使用全局对象

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

int global;

void f()
{
    global=5;
}

void g()
{
    cout<<global<<endl;
}

int main()
{
    f();
    g();
    return 0;
}
```

**需要处理的数据
很多的时候怎么办？**

数据与行为的封装

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

class Application
{
    public:
        void f();
        void g();
    private:
        int global;
};

void Application::f()
{
    global=5;
}
```

```
void Application::g()
{
    cout<<global<<endl;
}

int main()
{
    Application  MyApp;

    MyApp.f();

    MyApp.g();

    return 0;
}
```

类的静态成员

➤ 静态数据成员

- ❑ 用关键字static声明
- ❑ 该类的所有对象维护该成员的一个拷贝
- ❑ 必须在类外定义和初始化，用(::)来指明所属的类。
- ❑ 不需要实例化对象就可以访问类的静态数据成员

➤ 静态成员函数

- ❑ 静态成员函数只能引用属于该类的静态数据成员或静态成员函数。
- ❑ 类外代码可以使用类名和作用域操作符来调用静态成员函数
- ❑ 不需要实例化对象就可以调用类的静态成员函数

例：静态成员

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

class Point
{
private:
    int X,Y;
    static int countP;
public:
    Point(int xx=0, int yy=0)
    {
        X=xx; Y=yy; countP++;
    };
    Point(Point &p)
    {
        X=p.X; Y=p.Y; countP++;
    };
    int GetX() {return X;};
    int GetY() {return Y;};
    static int GetC()
    {
        return countP;
    };
};
```

```
int Point::countP = 0;

int main()
{
    cout << Point::GetC() << endl;

    Point A(4,5);
    cout << A.GetC() << endl;

    Point B(A);
    cout << B.GetC() << endl;

    /* 如果 countP 是 public 的,
       则下列语句也是正确的:

    cout << Point::countP << endl;
    */
}
```

例：静态成员

```
class A
{
    public:
        static void f();
        static void g();
        void h();
    private:
        static int s;
        int x;
};

void A::f()
{
    cout << s;
    g();

    cout << x;    // 错误，在静态函数中的只能访问静态成员变量
    h();          // 错误，在静态函数中的只能调用静态成员函数

    A a;          // a 是一个实例化的对象，
    a.h();         // 可以访问 a 的非静态成员
    ...
}
```

友元

- 友元是C++提供的一种破坏数据封装和隐藏的机制
 - ❑ 目的是为了更方便编程、提高代码效率，增加灵活性
 - ❑ 使程序员可以在封装性和快速性方面做合理折衷
 - ❑ C++ 允许将一个类或函数声明为另一个类的友元，这样在这个类或函数中就能够直接访问第二个类的隐藏信息（即 private/protected 类型的成员变量和成员函数）
- 可以使用友元函数和友元类。
- 为了确保数据的完整性，及数据封装与隐藏的原则，建议尽量不使用或少使用友元。

友元函数

➤ 友元函数

- ❑ 在类声明中由关键字 friend 修饰说明的非成员函数
- ❑ 友元函数可通过对象名访问 private 和 protected 成员

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

class Point
{
    friend double
    Distance(Point &a, Point &b);

public:    //外部接口
    ...
private: //私有数据成员
    int X,Y;
};

//成员函数的实现
...
```

```
double
Distance(Point &a, Point &b)
{
    double dx = a.X-b.X;
    double dy = a.Y-b.Y;
    return sqrt(dx*dx + dy*dy);
}

int main()
{
    Point p1(3.0, 5.0);
    Point p2(4.0, 6.0);
    double d = Distance(p1, p2);

    cout<< d <<endl;
    return 0;
}
```

友元类

➤ 友元类

- ❑ 所有成员函数都能访问对方类的 `private/protected` 成员
- ❑ 声明语法：在类中将友元类的类名用 `friend` 修饰

➤ 友元关系是单向的

- ❑ 如果声明B类是A类的友元，B类的成员函数就可以访问A类的私有和保护数据，但A类的成员函数却不能访问B类的私有、保护数据。

例：友元类

```
class A;
class B
{
    public:
        void Set(A& a, int i);
        void Display(A& a);
    private:
        ...
};
```

```
class A
{
    friend class B;
    private:
        void Display();
    private:
        int x;
};
```

```
void B::Set(A& a, int i)
{
    a.x=i;
}
```

```
void B::Display(A& a)
{
    a.Display();
}
```

```
class A;
class B
{
    public:
        void Set(A& a, int i);
        void Display(A& a);
    private:
        ...
};
```

```
class A
{
    friend void B::Set(A& a,int i);
    private:
        void Display();
    private:
        int x;
};
```

```
void B::Set(A& a, int i)
{
    a.x=i;
}
```

```
void B::Display(A& a)
{
    a.Display();
}
```

将 B 的一个成员
函数设为友元



常量类型

➤ 常类型的变量（对象）必须进行初始化，在程序中不能被更新，常量不能作为左值

➤ 常量引用：被引用的对象不能被更新

`const` 类型说明符 &引用名

➤ 常量对象：必须进行初始化，不能被更新

类名 `const` 对象名

➤ 常量数组：数组元素不能被更新

类型说明符 `const` 数组名[大小]...

➤ 常量指针

常量类型

```
const int x = 100;
x++;           // 错误
int array[x];  // C++中正确, C中错误
int &rx = x;    // 错误
const int &crx = x;

int a;
int &ra = a;
a = 1;
ra = 2;
cout << a << endl; // output: 2
const int &cra = a;
cra = 3; // 错误
```

```
class A
{
public:
    A(int i, int j) {x=i; y=j;}
    Set(int i, int j) {x=i; y=j;}
public:
    int x, y;
};

main()
{
    A a(3,4);
    a.Set(5,6);

    A const b(7,8);
    b.x = 9;           // 错误
    b.Set(9,10);       // 错误
    ...
}
```

常量类型作为参数

```
class A
{
    public:
        A(int i,int j)    {x=i; y=j;}
        Set(int i, int j) {x=i; y=j;}
    public:
        int x,y;
};
```

// 常量引用做形参，要求在函数中不能更新引用的对象，有助于提高安全性，防止无意的修改
// 编程中应提倡

```
void display(const A& a)
{
    cout<<a.x<<" "<<a.y<<endl;
}
```

```
main()
{
    A a(3,4);
    display(a);
    ...
}
```

const 修饰符

```
int const i=100; } 等价  
const int i=100;
```

```
int const &r = i; } 等价  
const int &r = i;
```

```
class A  
{  
    ...  
};  
const A a; } 等价  
A const a;
```

```
int a;  
int const *p = &a; } 等价  
const int *p = &a;
```

const 修饰指针

```
int a, b;
const int *p1 = &a;           // 指针指向的内容不能修改
int * const p2 = &a;          // 指针指向的地址不能修改
const int * const p3 = &a;    // 指针指向的地址和内容都不能修改
*p1 = 10;   X
p1 = &b;
*p2 = 10;
p2 = &b;   X
*p3 = 10;   X
p3 = &b;   X

char str[] = "abcdef";
const char *s1 = str;
char * const s2 = str;
const char * const s3 = str;
```


用 const 修饰的对象成员

➤ 常成员函数

- ❑ 使用const关键字说明的函数

- ❑ 常成员函数不能更新对象的数据成员

 - ✉ 常成员函数中，不能调用非常量成员函数

- ❑ 常成员函数说明格式：

 - 类型说明符 函数名（参数表） const;

 - 这里，const是函数类型的一个组成部分，因此在实现部分也要带 const 关键字

- ❑ const 关键字可以被用于参与对重载函数的区分

 - ✉ 相同的函数名可以带或不带 const，用于区分两个不同函数

➤ 使用常量对象时，只能调用其常成员函数

➤ 常数据成员

- ❑ 使用 const 说明的数据成员。

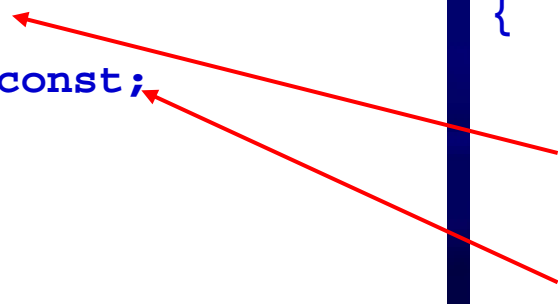
例：常成员函数

```
#include<iostream>
using namespace std;
class R
{
public:
    R(int r1, int r2)
    {
        R1=r1;R2=r2;
    }
    void print();
    void print() const;
private:
    int R1,R2;
};

void R::print()
{
    // 可以修改成员变量
    cout<<R1<<":"<<R2<<endl;
}

void R::print() const
{
    // 不可以修改成员
    cout<<R1<<";"<<R2<<endl;
}

int main()
{
    R a(5,4);
    a.print();    //调用print()
    const R b(20,52);
    b.print();    //调用print() const
}
```



例：常数据成员

```
#include<iostream>
using namespace std;
class A
{
    public:
        A(int i);
        void print();
    private:
        const int a;
};
A::A(int i):a(i)
{
    ...
}
```

```
void A::print()
{
    cout<<a<<endl;
}
int main()
{
    A a1(100);
    A a2(0);
    a1.print();    // output: 100
    a2.print();    // output: 0
}
```

在构造函数中，对常数据成员初始化，此后常量数据成员的值不能被修改

编译预处理命令

➤ #include 包含指令

- ❑ 将一个源文件嵌入到当前源文件中该点处。

- ❑ #include<文件名>

 - ☒ 按标准方式搜索，文件位于C++系统目录的include子目录下

- ❑ #include"文件名"

 - ☒ 首先在当前目录中搜索，若没有，再按标准方式搜索。

➤ #define 宏定义指令

- ❑ 定义符号常量，很多情况下已被const定义语句取代。

- ❑ 定义带参数宏，已被内联函数取代。

➤ #undef

- ❑ 删除由#define定义的宏，使之不再起作用。

条件编译指令 #if 和 #endif

#if 常量表达式1 //当“常量表达式”非零时编译

程序正文

#elif 常量表达式2

程序正文

#else

程序正文

#endif

条件编译指令

`#ifdef 标识符 或者 #ifndef 标识符`

`程序段1`

`#else`

`程序段2`

`#endif`

如果“标识符”已经用`#defined`定义过（或者未定义），
则编译程序段1，否则编译程序段2