第八章 继承与派生

目录

- 基本概念
- 派生类的声明方式及派生类对象的构成
- 派生类的成员访问属性
- 派生类的构造函数和析构函数
- 多重继承
- 虚基类
- 赋值兼容规则
- 继承和动态内存分配

8.1 基本概念

- 继承：C++中实现重用的机制（软件代码可重复利用）
- 类的继承：在一个已有类的基础上建立一个新类
- 类的派生：对继承而言，可以理解为已有类派生出新类
  - 可以在已有类的基础上添加新功能。例如：数组类可以添加数学运算
  - 可以给类添加数据。例如：字符串类，派生出的新类添加指定字符串显式颜色的数据成员
  - 可以修改类方法的行为。例如：对于代表提供给飞机乘客服务的Passenger类，派生出提供更高级别服务的FirstClassPassenger类

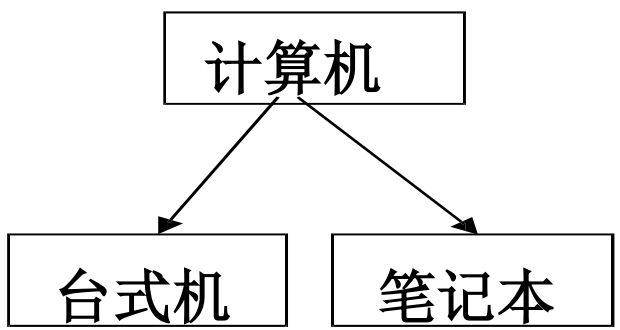
8.1 基本概念

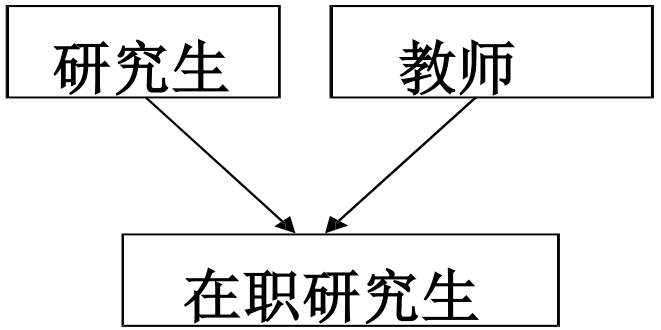
- 基类与派生类：基类(父类)，指已有的类

  派生类(子类)，新建立的类

- 派生类是基类的细化，基类是派生类的抽象

- 单继承与多继承：一个派生类从一个基类派生，则称为单继承，否则称为多继承





8.1 基本概念

- 一个简单的基类：

Webtown俱乐部决定跟踪乒乓球会会员。作为俱乐部的首席程序员，需要设计一个简单的TableTennisPlayer类：只记录会员的姓名以及是否有球桌。

数据成员：会员姓名（标准string类，比字符数组方便灵活且安全）

是否有球桌

成员函数：构造函数（建议使用成员初始化列表方式）

其他函数…

```
//tabtenn0.h -- a table-tennis base class
#ifndef TABTENN0_H_
#define TABTENN0_H_
#include <string>
class TableTennisPlayer
{
private:
    string firstname;  //比字符数组方便灵活且安全
    string lastname;
    bool hasTable;
public:
    TableTennisPlayer(const string & fn = "none",
                      const string & ln = "none", bool ht = false);
    void Name() const;
    bool HasTable() const { return hasTable; };
    void ResetTable(bool v) { hasTable = v; };
};
#endif
```

头文件
类的声明

```
//tabtenn0.cpp -- simple base-class methods
#include"tabtenn0.h"
#include <iostream>
using namespace std;
TableTennisPlayer::TableTennisPlayer(const string & fn, const string &
  ln, bool ht):firstname(fn),lastname(ln),hasTable(ht) {}
//初始化列表方式：直接使用string的复制构造函数将firstname初始化为fn…
void TableTennisPlayer::Name() const{
    cout << lastname << ", " << firstname;
}
```

源程序文件
函数的实现

- 将构造函数修改如下，注意区别：

```
TableTennisPlayer::TableTennisPlayer(const string & fn,
                                     const string & ln, bool ht){
    firstname = fn; lastname = ln; hasTable = ht;
}
//首先调用string的默认构造函数，再调用赋值运算符将firstname设置为fn…
```

```
//usett0.cpp -- using a base class
#include <iostream>
#include "tabtenn0.h"
using namespace std;
int main(){
    TableTennisPlayer player1("Chuck", "Blizzard", true);
    TableTennisPlayer player2("Tara", "Boomdea", false);
    player1.Name();
    if (player1.HasTable())
        cout << ": has a table.\n";
    else
        cout << ": hasn't a table.\n";
    player2.Name();
    if (player2.HasTable())
        cout << ": has a table.\n";
    else
        cout << ": hasn't a table.\n";
    return 0;
}
```

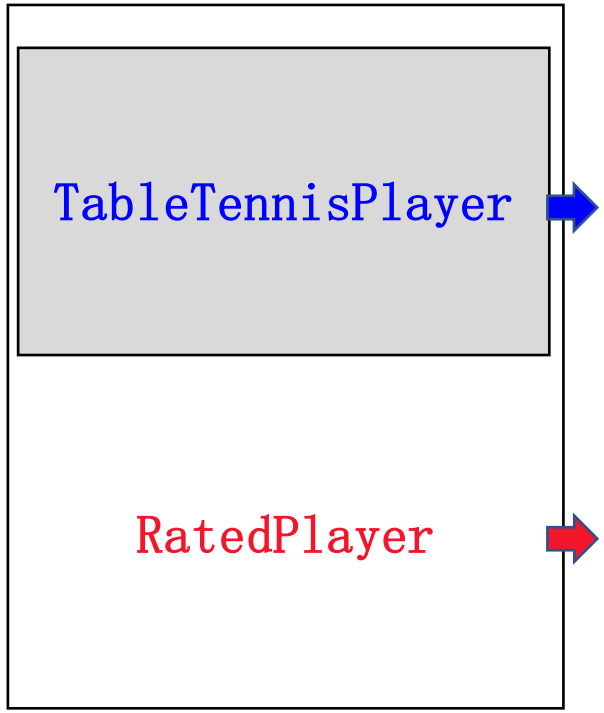
源程序文件
调用函数

构造函数的形参类型为const string &:
1)用string对象作构造函数的参数：调用接受const string&作为参数的string构造函数
2)用字符串作构造函数的参数（本例）：调用接受const char*作为参数的string构造函数

8.1 基本概念

- 派生一个类：

Webtown俱乐部的一些成员曾经参加过当地的乒乓球锦标赛，需要这样一个类，能包括成员在比赛中的比分。与其从零开始，不如从TableTennisPlayer类派生。



派生类对象存储了基类的数据成员（派生类继承了基类的实现）

派生类对象可以使用基类的方法（派生类继承了基类的接口）

派生类需要自己的构造函数

派生类可以根据需要添加额外的数据成员和成员函数

8.1 基本概念

- 派生一个类：

Webtown俱乐部的一些成员曾经参加过当地的乒乓球锦标赛，需要这样一个类，能包括成员在比赛中的比分。与其从零开始，不如从TableTennisPlayer类派生。

```
// simple derived class
class RatedPlayer : public TableTennisPlayer{
private:
   unsigned int rating;  //add a data member
public:
   RatedPlayer(unsigned int r = 0, const string & fn = "none",
               const string & ln = "none", bool ht = false);
   RatedPlayer(unsigned int r, const TableTennisPlayer & tp);
   unsigned int Rating() const { return rating; }     //add a method
   void ResetRating(unsigned int r) { rating = r; } //add a method
};
```

- 构造函数必须给新成员和继承的成员提供数据：//实现方法1

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
          const string & ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{
   rating = r;
}
```

若有：RatedPlayer rplayer1(1140, "Mallory", "Duck", true);

- RatedPlayer构造函数将实参"Mallory", "Duck", true 赋值给形参fn, ln, ht；
- 参数fn, ln, ht作为实参传递给TableTennisPlayer构造函数，创建一个嵌套的TableTennisPlayer对象，将数据"Mallory", "Duck", true 存储在该对象中；
- 进入RatedPlayer构造函数体，完成RatedPlayer对象的创建，并将r的值（1140）赋值给rating成员。

- 构造函数必须给新成员和继承的成员提供数据： //实现方法1-引申

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
          const string & ln, bool ht) //省略成员初始化列表，等效于
                                      :TableTennisPlayer()
{
   rating = r;
}
```

若有：RatedPlayer rplayer1(1140, "Mallory", "Duck", true);

- RatedPlayer构造函数将调用默认的基类构造函数；
- 除非要使用默认构造函数，否则应显式调用正确的基类构造函数。

- 构造函数必须给新成员和继承的成员提供数据：//实现方法2

```
RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
        : TableTennisPlayer(tp)
{
    rating = r;
}
```

```
RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
        : TableTennisPlayer(tp), rating(r){}
```

若有：RatedPlayer rplayer1(1140, tp);

- 派生类RatedPlayer的构造函数将基类信息tp传递给基类TableTennisPlayer构造函数，即通过调用基类的复制构造函数完成；
- 使用动态内存分配时，注意隐式复制构造函数的合理性；
- 构造和析构的顺序（8.4讨论）。

```
//tabtenn1.h -- a table-tennis base class
class TableTennisPlayer{…}
class RatedPlayer : public TableTennisPlayer{…}
```

头文件
类的声明

```
//tabtenn1.cpp -- simple base-class methods
#include"tabtenn1.h"
TableTennisPlayer::TableTennisPlayer(const string & fn, const string &
   ln, bool ht):firstname(fn),lastname(ln),hasTable(ht){}
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
          const string & ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{ rating = r; }
…
```

源程序文件
函数的实现

```
//usett1.cpp -- using base class and derived class
#include"tabtenn1.h"
int main()
{  TableTennisPlayer player1("Tara", "Boomdea", false);
   RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
   …
}
```

源程序文件
调用函数

完整程序见primer书

8.1 基本概念

- 继承关系的本质：is-a关系

建议不建立下述关系，否则导致编程问题：

➢has-a关系：午餐有水果，但通常午餐并不是水果；

➢is-like-a关系：律师像鲨鱼，但律师不是鲨鱼（不可以在水下生活）；

➢is-implemented-as-a关系：使用数组来实现栈，但栈不是数组；

➢uses-a关系：计算机可以使用激光打印机，但从计算机派生出打印机类没有意义。

目录

- 基本概念
- 派生类的声明方式及派生类对象的构成
- 派生类的成员访问属性
- 派生类的构造函数和析构函数
- 多重继承
- 虚基类
- 赋值兼容规则
- 继承和动态内存分配

8.2 派生类的声明方式及派生类对象的构成

8.2.1 派生类的声明方式：

```
class 派生类名:private/public 基类名1,
               private/public 基类名2,
                              ...
               private/public 基类名n {
    private:
        私有成员;
    public:
        公有成员;
}
```

当只有一个基类名时，单继承

当多于一个基类名时，多继承

- 根据需要加入派生类自己特有的成员

8.2 派生类的声明方式及派生类对象的构成

8.2.1 派生类的声明方式：

```
class 派生类名:private/public 基类名1,
               private/public 基类名2,
                              ...
               private/public 基类名n {
    private:
        私有成员;
    public:
        公有成员;
}
```

基类存取限定符

- 根据限定符分为私有继承和公有继承，缺省是private

8.2 派生类的声明方式及派生类对象的构成

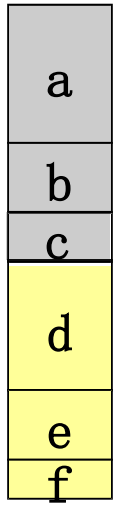
8.2.2 派生类对象的构成：

```
class A {
   private:
      int a;
      short b;
      char c;
   public:
      void f1();
      int f2();
}
```

```
class B:public A {
   private:
      int d;
      short e;
      char f;
   public:
      void f3();
      int f4();
}
```

```
B b1;
b1.f1();
b1.f2();
b1.f3();
b1.f4();
```

内存：



存在≠可访问

- 继承基类的全部数据成员(不一定都可以访问)
- 继承基类除构造函数和析构函数外的全部成员函数(不一定都可以访问)

8.2 派生类的声明方式及派生类对象的构成

- 派生类对象所占的空间：

  基类数据成员所占空间总和 + 派生类数据成员所占空间的总和

- 派生类可访问的成员函数：

  基类成员函数 + 派生类成员函数

- 友元不能继承
- 派生类的数据成员/成员函数允许和基类的同名，不同的继承方式访问方法不同(8.3讨论)

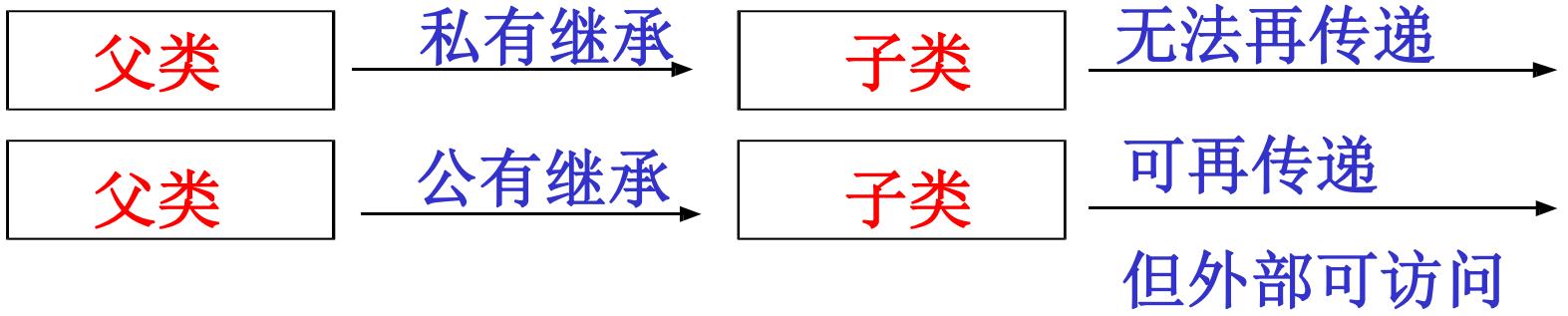
目录

- 基本概念
- 派生类的声明方式及派生类对象的构成
- 派生类的成员访问属性
- 派生类的构造函数和析构函数
- 多重继承
- 虚基类
- 赋值兼容规则
- 继承和动态内存分配

8.3 派生类的成员访问属性

- 公有继承与私有继承

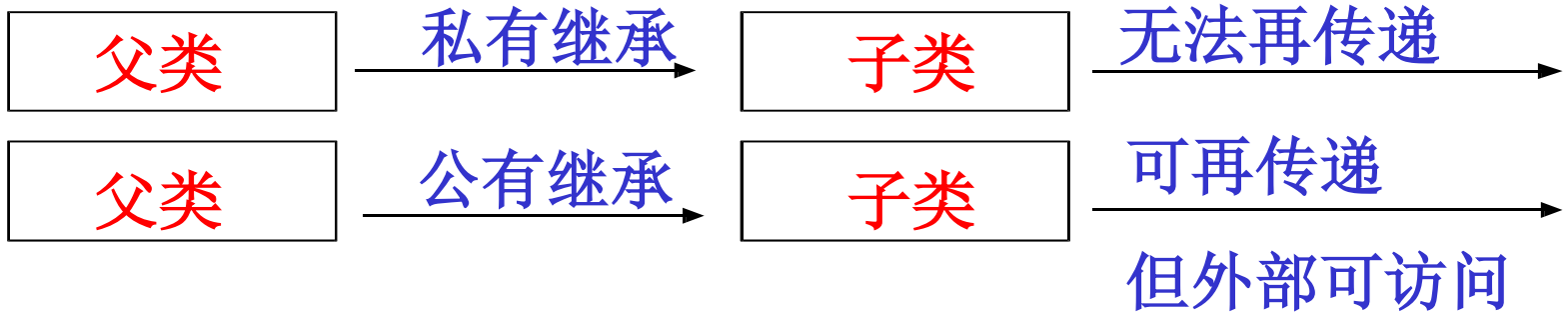
基类成员访问控制	继承访问控制	在派生类中的访问控制
public	public	public
private		不可访问
public	private	private
private		不可访问



8.3 派生类的成员访问属性

- 保护段与保护继承

在多级继承中，基类的private不可访问，public部分被私有继承则不可再传递，若公有继承则对于整个继承序列的外部均可访问：



为了加强灵活性，引入保护段及保护继承：

(1) 父类的成员在派生类的继承序列中内部能够访问

(2) 父类的成员在派生类的继承序列中外部不能访问

8.3 派生类的成员访问属性

- 保护段的定义：

```
class 类名 {
    private:
        ...
    protected:
        ...
    public:
        ...
}
```

➢protected被private继承后当做派生类的private

➢protected被public继承后当做派生类的protected

➢若该类不被其它类所继承，则声明保护段无意义（不被继承的情况下与声明为private等价）

8.3 派生类的成员访问属性

- 派生类的最终定义形式：

```
class 派生类名: private/protected/public 基类名{
        private:
            ...
        protected:
            ...
        public:
            ...
};
```

不同继承方式/不同限定成员的访问属性？

8.3 派生类的成员访问属性

- 继承方式与访问属性：

基类成员访问控制	继承访问控制	在派生类中的访问控制	
public	基类访问限定	public	(1)公有继承
protected	public	protected	
private		不可访问	
public	继承访问控制	protected	(2)保护继承
protected	protected	protected	
private		不可访问	
public	继承访问控制	private	(3)私有继承
protected	private	private	
private		不可访问	

```
class Base
{public:
      int x;
  protected:
      int y;
  private:
      int z;
}
```

Base B;

x
y
z

Base B;

int k=B.x; (√)

int k=B.y; (×)

int k=B.z; (×)

基类成员访问控制	继承访问控制	在派生类中的访问控制
public	基类访问限定	public
protected	public	protected
private		不可访问

(1) 公有继承

```
class C1:public Base
{public:
      int p;
      void fun()
      {只有z不可访问}
}
```

C1 C;

x, p, fun()
y
z

C1 C;

int k=C.p; (√)

int k=C.x; (√)

int k=C.y; (×)

int k=C.z; (×)

- 例1：B公有继承A

```
class A {
   private:
      int a;
      void f1();
   protected:
      int b;
      void f2();
   public:
      int c;
      void f3();
};
```

```
class B:public A{
   private:
      int d;
      void f4();
   protected:
      int e;
      void f5();
   public:
      int f;
      void f6();
      void fun();
};
```

c, f3(), f, f6(), fun()
b, f2(), e, f5()
d, f4()
a, f1()

```
void fun() //成员
{
    a=10; ×
    f1(); ×
    b=10;
    f2();
    c=10;
    f3();
    d=10;
    f4();
    e=10;
    f5();
    f=10;
    f6();
}
```

```
int main()  //域外
{  C c1;
   c1.a=10;×
   c1.f1();×
   c1.b=10;×
   c1.f2();×
   c1.c=10;
   c1.f3();
   c1.d=10;×
   c1.f4();×
   c1.e=10;×
   c1.f5();×
   c1.f=10;
   c1.f6();
}
```

```
class Base
{public:
      int x;
  protected:
      int y;
  private:
      int z;
}
```

Base B;

x
y
z

```
Base B;
int k=B.x;  (√)
int k=B.y;  (×)
int k=B.z;  (×)
```

继承访问控制	
public	protected
protected	protected
private	不可访问

(2) 保护继承

```
class C1:protected Base
{public:
      int p;
      void fun()
      {只有z不可访问}
}
```

C1 C;

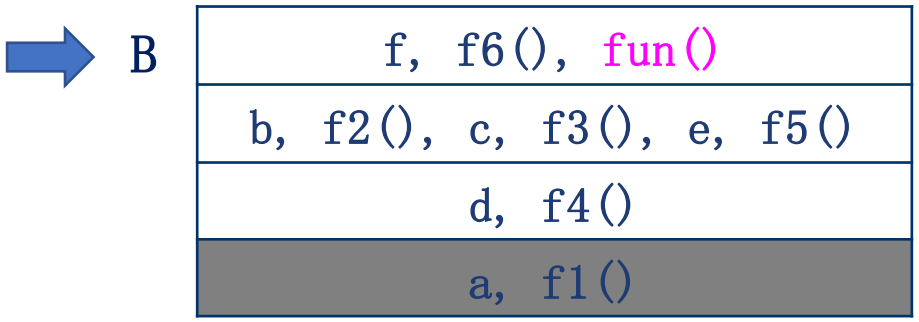
p,fun()
x, y
z

```
C1 C;
int k=C.p; (√)
int k=C.x; (×)
int k=C.y; (×)
int k=C.z; (×)
```

- 例2：B保护继承A

```
class A {
    private:
        int a;
        void f1();
    protected:
        int b;
        void f2();
    public:
        int c;
        void f3();
};
```

```
class B:protected A{
    private:
        int d;
        void f4();
    protected:
        int e;
        void f5();
    public:
        int f;
        void f6();
        void fun();
};
```




```
void fun() //成员
{
    a=10; ×
    f1(); ×
    b=10;
    f2();
    c=10;
    f3();
    d=10;
    f4();
    e=10;
    f5();
    f=10;
    f6();
}
```

```
int main() //域外
{  C c1;
   c1.a=10;×
   c1.f1();×
   c1.b=10;×
   c1.f2();×
   c1.c=10;×
   c1.f3();×
   c1.d=10;×
   c1.f4();×
   c1.e=10;×
   c1.f5();×
   c1.f=10;
   c1.f6();
}
```

```
class Base
{public:
      int x;
  protected:
      int y;
  private:
      int z;
}
```

Base B;

x
y
z

```
Base B;
int k=B.x;   (√)
int k=B.y;   (×)
int k=B.z;   (×)
```

继承访问控制		
public		private
protected	private	private
private		不可访问

(3)私有继承

```
class C1:private Base
{public:
      int p;
      void fun()
      {只有z不可访问}
}
```

C1 C;

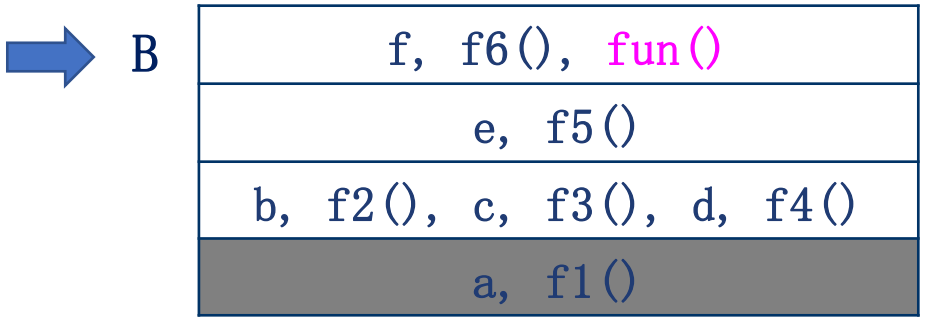
p, fun()
x, y
z

```
C1 C;
int k=C.p; (√)
int k=C.x; (×)
int k=C.y; (×)
int k=C.z; (×)
```

- 例3：B私有继承A

```
class A {
   private:
      int a;
      void f1();
   protected:
      int b;
      void f2();
   public:
      int c;
      void f3();
};
```

```
class B:private A{
   private:
      int d;
      void f4();
   protected:
      int e;
      void f5();
   public:
      int f;
      void f6();
      void fun();
};
```




```
void fun() //成员
{
    a=10; ×
    f1(); ×
    b=10;
    f2();
    c=10;
    f3();
    d=10;
    f4();
    e=10;
    f5();
    f=10;
    f6();
}
```

```
int main() //域外
{  C c1;
   c1.a=10;×
   c1.f1();×
   c1.b=10;×
   c1.f2();×
   c1.c=10;×
   c1.f3();×
   c1.d=10;×
   c1.f4();×
   c1.e=10;×
   c1.f5();×
   c1.f=10;
   c1.f6();
}
```

8.3 派生类的成员访问属性

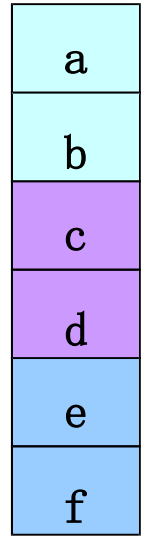
- 派生类的多级派生

```
class A {
  private:
    int a;
    void f1();
  public:
    int b;
    void f2();
}
```

```
class B: public A {
   private:
     int c;
     void f3();
  public:
     int d;
     void f4();
}
```

```
class C: public B {
  private:
    int e;
    void f5();
  public:
    int f;
    void f6();
}
```

C c1;



存在≠可访问

- 基类分为直接基类和间接基类：C的直接基类是B，间接基类是A，B的直接基类是A
- 派生类包含所有基类的数据成员(不一定可访问)
- 基类的成员的被访问关系逐级确定

8.3 派生类的成员访问属性

- 多级派生例1：B私有继承A，C私有继承B
- 多级派生例2：B公有继承A，C公有继承B
- 多级派生例3：B保护继承A，C保护继承B
- 多级派生例4：B私有继承A，C公有继承B
- 多级派生例5：B私有继承A，C保护继承B
- 多级派生例6：B公有继承A，C私有继承B
- 多级派生例7：B公有继承A，C保护继承B
- 多级派生例8：B保护继承A，C私有继承B
- 多级派生例9：B保护继承A，C公有继承B

自行补充

//基类的成员的被访问关系逐级确定，注意区分成员函数内访问和域外访问

- 多级派生例1：B私有继承A，C私有继承B

```
class A {
  private:
    int a1;
    void fa1();
  protected:
    int a2;
    void fa2();
  public:
    int a3;
    void fa3();
};
```

```
class B: private A{
  private:
    int b1;
    void fb1();
  protected:
    int b2;
    void fb2();
  public:
    int b3;
    void fb3();
};
```

```
class C: private B{
  private:
    int c1;
    void fc1();
  protected:
    int c2;
    void fc2();
  public:
    int c3;
    void fc3();
    void fun();
};
```

逐级确定

B

b3, fb3()
b2, fb2()
a2, fa2(), a3, fa3(), b1, fb1()
a1, fa1()

C

c3, fc3(), fun()
c2, fc2()
b2, fb2(), b3, fb3(), c1, fc1()
a1, fa1(), a2, fa2(), a3, fa3(), b1, fb1()

- 多级派生例1：B私有继承A，C私有继承B

```
void C::fun() //成员
{
    a1=10;×  fa1();×
    a2=10;×  fa2();×
    a3=10;×  fa3();×
    b1=10;×  fb1();×
    b2=10;   fb2();
    b3=10;   fb3();
    c1=10;   fc1();
    c2=10;   fc2();
    c3=10;   fc3();
}
```

```
int main()  //域外
{   C c;
    c.a1=10;×  c.fa1();×
    c.a2=10;×  c.fa2();×
    c.a3=10;×  c.fa3();×
    c.b1=10;×  c.fb1();×
    c.b2=10;×  c.fb2();×
    c.b3=10;×  c.fb3();×
    c.c1=10;×  c.fc1();×
    c.c2=10;×  c.fc2();×
    c.c3=10;   c.fc3();
}
```

逐级确定

B

b3, fb3()
b2, fb2()
a2, fa2(), a3, fa3(), b1, fb1()
a1, fa1()

C

c3, fc3(), fun()
c2, fc2()
b2, fb2(), b3, fb3(), c1, fc1()
a1, fa1(), a2, fa2(), a3, fa3(), b1, fb1()

- 多级派生例2：B公有继承A，C公有继承B

```
class A {
  private:
    int a1;
    void fa1();
  protected:
    int a2;
    void fa2();
  public:
    int a3;
    void fa3();
};
```

```
class B: public A{
  private:
    int b1;
    void fb1();
  protected:
    int b2;
    void fb2();
  public:
    int b3;
    void fb3();
};
```

```
class C: public B{
  private:
    int c1;
    void fc1();
  protected:
    int c2;
    void fc2();
  public:
    int c3;
    void fc3();
    void fun();
};
```

逐级确定

B

a3, fa3(), b3, fb3()
a2, fa2(), b2, fb2()
b1, fb1()
a1, fa1()

C

a3, fa3(), b3, fb3(), c3, fc3(), fun()
a2, fa2(), b2, fb2(), c2, fc2()
c1, fc1()
a1, fa1(), b1, fb1()

- 多级派生例2：B公有继承A，C公有继承B

```
void C::fun() //成员
{
    a1=10;×  fa1();×
    a2=10;   fa2();
    a3=10;   fa3();
    b1=10;×  fb1();×
    b2=10;   fb2();
    b3=10;   fb3();
    c1=10;   fc1();
    c2=10;   fc2();
    c3=10;   fc3();
}
```

```
int main()  //域外
{   C c;
    c.a1=10;×  c.fa1();×
    c.a2=10;×  c.fa2();×
    c.a3=10;   c.fa3();
    c.b1=10;×  c.fb1();×
    c.b2=10;×  c.fb2();×
    c.b3=10;   c.fb3();
    c.c1=10;×  c.fc1();×
    c.c2=10;×  c.fc2();×
    c.c3=10;   c.fc3();
}
```

逐级确定

B

a3, fa3(), b3, fb3()
a2, fa2(), b2, fb2()
b1, fb1()
a1, fa1()

C

a3, fa3(), b3, fb3(), c3, fc3(), fun()
a2, fa2(), b2, fb2(), c2, fc2()
c1, fc1()
a1, fa1(), b1, fb1()

8.3 派生类的成员访问属性

- 派生类与基类的成员同名

a）数据成员同名：

➢派生类屏蔽基类，通过直接访问方式可访问派生类的成员，通过加类的作用域符的方式可访问基类成员

b）成员函数同名：

➢参数的个数、类型完全相同：与数据成员同名的处理方式相同

➢参数的个数、类型不同：与数据成员同名的处理方式相同

上述规则一致：派生类优先于基类，称为支配规则

a）数据成员同名：

➢派生类屏蔽基类，通过直接访问方式可访问派生类的成员，通过加类的作用域符的方式可访问基类成员

```
class A{
public:
   int a;
   …
};
class B:public A{
public:
   short a;
   void fun();
   …
};
```

```
void B::fun()
{
   a=10;
   A::a=15;
}
int main()
{
   B b1;
   b1.a=10;
   b1.A::a=15;
}
```

```
int a;
class A{
public:
   int a;
   …
};
class B:public A{
public:
   short a;
   void fun();
   …
};
```

```
void B::fun()
{
   int a;
   a=10;
   A::a=15;
   B::a=20;
   ::a=30;
}
```

b）成员函数同名：

➢参数的个数、类型完全相同：与数据成员同名的处理方式相同

```
class A {
public:
    void f1();
    …
};
class B:public A {
public:
    void f1();
    void fun();
    …
};
```

```
void B::fun()
{
    f1();
    A::f1();
}
int main(){
    B b1;
    b1.f1();
    b1.A::f1();
}
```

b）成员函数同名：

➢参数的个数、类型不同：与数据成员同名的处理方式相同

```
class A {
public:
    void f1(int x);
    …
};
class B:public A {
public:
    void f1();
    void fun();
    …
};
```

```
void B::fun()
{
    f1();
    f1(10); //编译错误
}
int main()
{
    B b1;
    b1.f1();
    b1.f1(10); //编译错误
}
```

b）成员函数同名：

➢参数的个数、类型不同：与数据成员同名的处理方式相同

```
class A {
public:
    void f1(int x);
    …
};
class B:public A {
public:
    void f1();
    void fun();
    …
};
```

```
void B::fun()
{
    f1();
    A::f1(10);
}
int main()
{
    B b1;
    b1.f1();
    b1.A::f1(10);
}
```

8.3 派生类的成员访问属性

- 基类对象不可访问部分的强制访问

➢问题引入：

任何继承方式派生类均不能访问基类的private

派生类的对象中包含了基类对象(包括private部分)

=> 不可访问，又占空间？

➢解决方法：

继承后，基类的私有部分不可被派生类直接访问，但是可以通过基类的可访问的成员函数进行间接访问

- 基类对象不可访问部分的强制访问

```
#include <iostream>
using namespace std;
class A {
  private:
     int a;
  public:
     void set(int x)
     {
        a=x;
     }
     void show()
     {
        cout << a << endl;
     }
};
```

```
class B:public A {
  public:
     void fun()
     {
        a=10;   //编译错
        set(10); //但可间接访问
     }
};
int main(){
     B b1;
     b1.set(15);
     b1.show();
     b1.fun();
     b1.show();
}
```

目录

- 基本概念
- 派生类的声明方式及派生类对象的构成
- 派生类的成员访问属性
- 派生类的构造函数和析构函数
- 多重继承
- 虚基类
- 赋值兼容规则
- 继承和动态内存分配

8.4 派生类的构造函数和析构函数

8.4.1 简单的派生类构造函数

- 含义：在派生类的数据成员中不包含基类的对象
- 形式：派生类名(参数)：基类名(参数) 初始化表方式

```
{
    函数体(一般是对派生类新增成员的初始化)
}
```

- 构造函数的调用顺序：
  - 先基类，再派生类（在派生类的构造函数中先自动激活基类的构造函数）

8.4 派生类的构造函数和析构函数

8.4.1 简单的派生类构造函数

```
class RatedPlayer: public TableTennisPlayer {                    //体内实现
    …
    RatedPlayer(unsigned int r, const string & fn,
          const string & ln, bool ht) : TableTennisPlayer(fn, ln, ht)
    { rating = r; }
}
```

```
class RatedPlayer: public TableTennisPlayer {                    //体外实现
    …
    RatedPlayer(unsigned int r, const string & fn, const string & ln, bool ht);
}
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
          const string & ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{ rating = r; }
```

8.4 派生类的构造函数和析构函数

8.4.1 简单的派生类构造函数

- 若派生类的成员初始化表中不出现基类，则系统自动调用基类的无参构造函数

  （基类不定义构造函数时调用系统缺省的无参空体构造函数）

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
                         const string & ln, bool ht) //省略成员初始化列表，等效于
                                                     :TableTennisPlayer()
{
   rating = r;
}
```

- 派生类的成员初始化列表中出现基类

```
#include <iostream>
using namespace std;
class A {
public:
    A() { cout << "A()" << endl; }
    A(int x) { cout << "A(x)" << endl; }
};
class B:public A {
public:
    B(int x):A(x){ cout << "B(x)" << endl; }
    B(int x, int y):A(x){ cout << "B(x,y)" << endl; }
};
int main(){
    B b1(10);  cout << endl;
    B b2(100,200);
}
```

```
A(x)
B(x)

A(x)
B(x,y)
```

- 派生类的初始化列表中不出现基类（省略成员初始化列表）

```
#include <iostream>
using namespace std;
class A {
public:
    A() { cout << "A()" << endl; }
    A(int x) { cout << "A(x)" << endl; }
};
class B:public A {
public:
    B() { cout << "B()" << endl; }
    B(int x) { cout << "B(x)" << endl; }
};
int main(){
    B b1;  cout << endl;
    B b2(100);
}
```

```
A()
B()

A()
B(x)
```

- 不能以初始化列表的方式实现重载

```
#include <iostream>
using namespace std;
class A {
  public:
    A() { cout << "A()" << endl; }
    A(int x){ cout << "A(x)" << endl; }
  };
class B:public A {
   public:
    B() { cout << "B()" << endl; }
    B(int x) { cout << "B(x1)" << endl; }//重载错误
    B(int x):A(x) { cout << "B(x2)" << endl; }
    B(int x, int y):A(x){ cout << "B(x,y)" << endl; }
  };
```

```
int main()
{
   B b1; cout << endl;
   B b2(100); cout << endl;
   B b3(4, 5);
}
```

保留第一个：

```
A()
B()

A()
B(x1)

A(x)
B(x,y)
```

保留第二个：

```
A()
B()

A(x)
B(x2)

A(x)
B(x,y)
```

8.4 派生类的构造函数和析构函数

8.4.2 含有子对象的派生类构造函数

- 含义：在派生类定义中包含了基类的对象
- 形式：

```
派生类名(参数):基类名(参数),子对象名(参数)
{
    函数体(一般是对派生类新增成员的初始化)
}
```

- 构造函数的调用顺序：
  - 先基类，次子对象，再派生类

- 含有子对象的派生类构造函数

```
class student {
   protected:
      int num;
      char name[10];
   public:
      student(int n,char *nam){
         num = n; strcpy(name, nam);
      }
   …
};
class stu:public student {
   protected:
      int age;
      char addr[30];
      student monitor;
   public:
      stu(int n, char *nam, int n1, char *nam1,int a, char *ad)
                  : student(n,nam), monitor(n1,nam1){ age = a; strcpy(addr, ad);}
};
```

```
int main()
{
   stu s1(10001,"张三",10009,"李四",20,"上海");
   …
}
```

班长

s1所占空间为14*2+34=62字节，即双份student，其中1份继承，1份是monitor

8.4 派生类的构造函数和析构函数

8.4.3 多层派生时的构造函数

- 形式：

派生类名(参数):直接基类名(参数)

{

函数体(一般是对派生类新增成员的初始化)

}

  - 间接基类不需要出现在初始化表中

- 构造函数的调用顺序：
  - 按继承顺序依次调用，派生类放在最后

(派生类中先激活直接基类，直接基类再先激活其直接基类)

- 多层派生时的构造函数

```
class A {
public:
   A(int x) {
      cout << "A" << x << endl;}
};
class B: public A{
public:
   B(int x, int y):A(x) {
      cout << "B" << y << endl;}
};
class C: public B {
public:
   C(int x, int y, int z):B(x, y){
        cout << "C" << z << endl;}
};
```

```
int main()
{
   B b1(4,5);
   cout << endl;
   C c1(1,2,3);
}
```

```
A4
B5

A1
B2
C3
```

8.4 派生类的构造函数和析构函数

8.4.4 派生类的析构函数

- 析构函数的调用：
  - ➢在派生类对象出作用域时，自动调用派生类的析构函数，在其中再自动调用基类的析构函数
- 析构函数的调用顺序：
  - ➢先派生类，再基类(与构造函数的顺序相反)
- 析构函数的使用：
  - ➢派生类的数据成员无动态内存申请的情况下一般不需要定义派生类的析构函数
  - ➢若基类有动态内存申请，派生类无，则基类需要定义，派生类不需要（见8.8）

目录

- 基本概念
- 派生类的声明方式及派生类对象的构成
- 派生类的成员访问属性
- 派生类的构造函数和析构函数
- 多重继承
- 虚基类
- 赋值兼容规则
- 继承和动态内存分配

8.5 多重继承

8.5.1 多重继承的声明及使用

- 声明形式

```
class 派生类名: private/protected/public 基类名1, ... ,
                          private/protected/public 基类名n {
        private:
            私有成员;
        protected:
            保护成员;
        public:
            公有成员;
};
```

8.5 多重继承

8.5.1 多重继承的声明及使用

- 每个基类的基类存取限定符允许不同
- 每个基类被继承后的可访问性由其基类存取限定符确定
- 派生类继承所有基类的数据成员
- 派生类继承所有基类除构造函数和析构函数外的所有成员函数
- 派生类对象所占的空间 = 所有基类的数据成员之和 + 派生类的数据成员之和

8.5 多重继承

8.5.1 多重继承的声明及使用

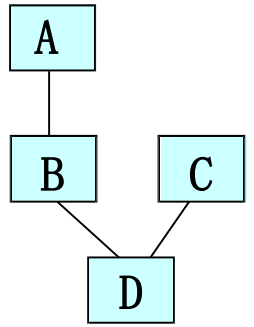
- 多重继承允许多层派生

```
class A {
  private:
    int a1;
  protected:
    int a2;
  public:
    int a3;
};
```

```
class B:public A {
  private:
    int b1;
  protected:
    int b2;
  public:
    int b3;
};
```

```
class C {
  private:
    int c1;
  protected:
    int c2;
  public:
    int c3;
};
```

```
class D:private B, public C {
 ...
};
```



8.5 多重继承

8.5.2 多重继承的派生类构造函数和析构函数

- 构造函数的形式

  派生类名(参数)：基类名1(参数),...， 基类名n(参数)

  {

  函数体(一般是对派生类新增成员的初始化)

  }

- 构造函数的使用：
  - 基类名出现的顺序无限制
  - 若调用基类的无参构造函数，则该基类可不出现在初始化参数表中
  - 若是多层派生，只出现直接基类
  - 若派生中含有基类的实例对象(子对象)，则子对象也可以出现在初始化参数表中

- 构造函数的使用：
  - 基类名出现的顺序无限制

```
class A {
  public:
    A(int x) { … }
}
class B {
  public:
    B(int y) { … }
}
```

```
class C:public A, private B{
  public:
    C(int a, int b, int c):A(a), B(b) {
                              B(b), A(a)
        …
    }
};
```

- 构造函数的使用：
  - 若调用基类的无参构造函数，则该基类可不出现在初始化参数表中

```
class A {
  public:
    A() { … }
}
class B {
  public:
    B(int y) { … }
}
```

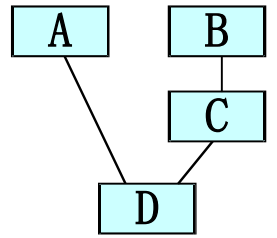
```
class C:public A, private B {
  public:
    C(int a, int b, int c):B(b) { … }
};
```

激活A的无参构造

- 构造函数的使用：
  - 若是多层派生，只出现直接基类

```
class A {
  public:
    A(int x) { … }
}
class B {
  public:
    B(int y) { … }
}
```

```
class C:public B {
  public:
     C(int z):B(z) { … };
}
class D:public A, private C {
  public:
     D(int a, int c, int d):A(a),C(c) { … }
};
```



B不需要出现

- 构造函数的使用：
  - 若派生中含有基类的实例对象(子对象)，则子对象也可以出现在初始化参数表中

```
class A {
  public:
    A(int x) { … }
}
class B {
  public:
    B(int y) { … }
}
```

```
class C:public A, private B {
  public:
    B bb;
    C(int a, int b, int b1):A(a), B(b), bb(b1){
    …
  }
};
```

8.5 多重继承

8.5.2 多重继承的派生类构造函数和析构函数

- 构造函数的调用顺序：
  - 按声明顺序依次调用基类的构造函数，最后调用派生类的构造函数
- 析构函数的调用顺序：
  - 先派生类的析构函数，再按声明顺序的反序依次调用基类的析构函数

//例1：

```
#include <iostream>
using namespace std;
class A {
   public:
      A() {cout << "A()" << endl;}
};
class B {
   public:
      B() {cout << "B()" << endl;}
};
class C {
   public:
      C() {cout << "C()" << endl;}
};
```

```
class D:public A, protected B, private C{
    public:
       D() {cout << "D()" << endl;}
};    //D():A(),B(),C() {…}
      //D():B(),A(),C() {…}
      //D():C(),B(),A() {…}
      …
int main(){
    D d1;
}
```

D对象的内存映像：

A的数据成员
B的数据成员
C的数据成员
D的数据成员

//例2：

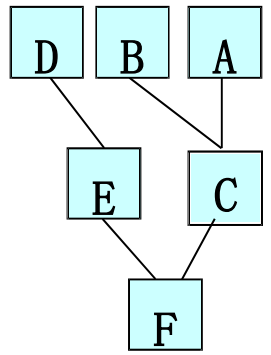
```
#include <iostream>
using namespace std;
class A {
  public:
   int a;
   A(){ a=1; }
};
class B {
  public:
   int b;
   B(){ b=2;}
};
class C {
  public:
   int c;
   C(){ c=3; }
};
class D:public A,
          protected B,
            private C {
 public:
    int d;
    D(){ d=4; }
};
int main()
{
   D d1;
   cout << sizeof(d1) << endl;
   int *p = (int *)&d1;
   cout << *p << endl;
   cout << *(p+1) << endl;
   cout << *(p+2) << endl;
   cout << *(p+3) << endl;
   return 0;
}
```

```
16
1
2
3
4
```

//例3：

```
#include <iostream>
using namespace std;
class A {
  public:
    A() {cout << "A()" << endl;}
};
class B {
  public:
    B() {cout << "B()" << endl;}
};
class C:public B, public A {
  public:
    C() {cout << "C()" << endl;}
};
class D {
  public:
    D() {cout << "D()" << endl;}
};
class E:public D {
  public:
    E() {cout << "E()" << endl;}
};
class F:public E, public C {
  public:
    F() {cout << "F()" << endl;}
};
int main(){
    F f1;
}
```

```
D()
E()
B()
A()
C()
F()
```



F对象的内存映像：

8.5 多重继承

- 8.5.3 多重继承引起的二义性问题(成员同名)
  - 以不产生二义性为基本准则

某一个基类与派生类的成员同名：

按单继承的方式进行处理(支配规则：派生类直接访问，基类加作用域符)

两个以上的基类中的成员同名：

分别加不同的基类作用域符区分

两个以上的基类与派生类的成员同名：

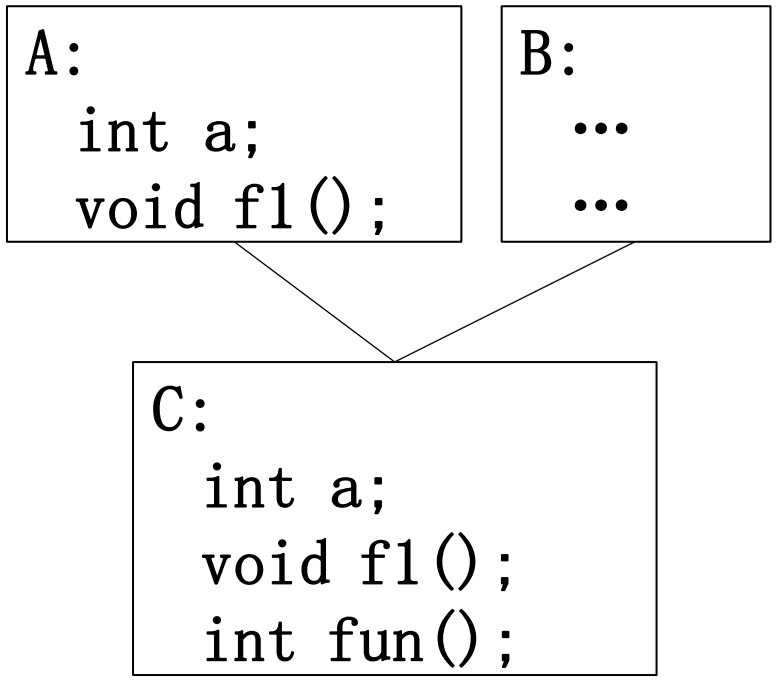
派生类直接访问，不同基类加作用域符区分

通过直接/间接方式继承同一个基类两个导致的同名：

通过可区分的类的作用域符来进行区分

- 某一个基类与派生类的成员同名：按单继承的方式进行处理

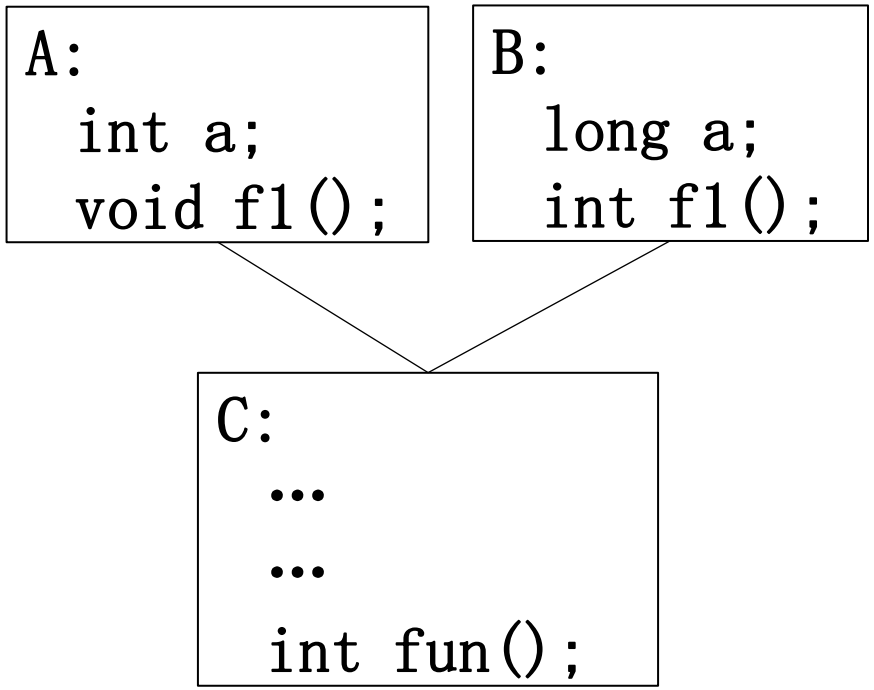
  （支配规则:派生类直接访问，基类加作用域符）




```
int C::fun()
{
    a=10;
    f1();
    A::a=15;
    A::f1();
}
```

```
int main()
{
    C c1;
    c1.a=10;
    c1.f1();
    c1.A::a=15;
    c1.A::f1();
}
```

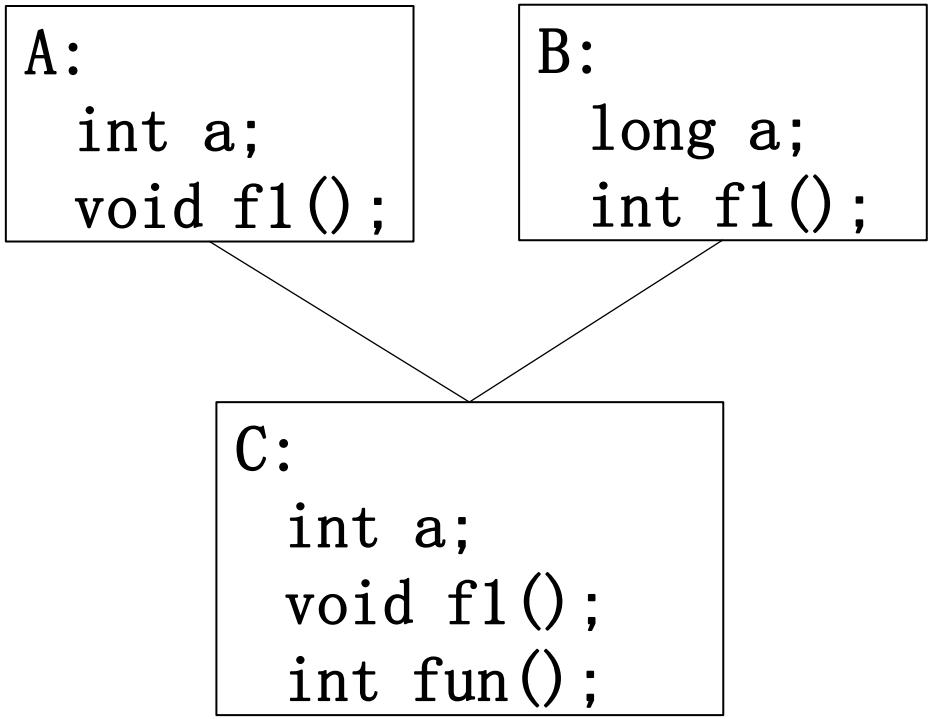
- 两个以上的基类中的成员同名：分别加不同的基类作用域符区分




```
int C::fun()
{
    A::a=10;
    A::f1();
    B::a=15;
    B::f1();
}
```

```
int main()
{
    C c1;
    c1.A::a=10;
    c1.A::f1();
    c1.B::a=15;
    c1.B::f1();
}
```

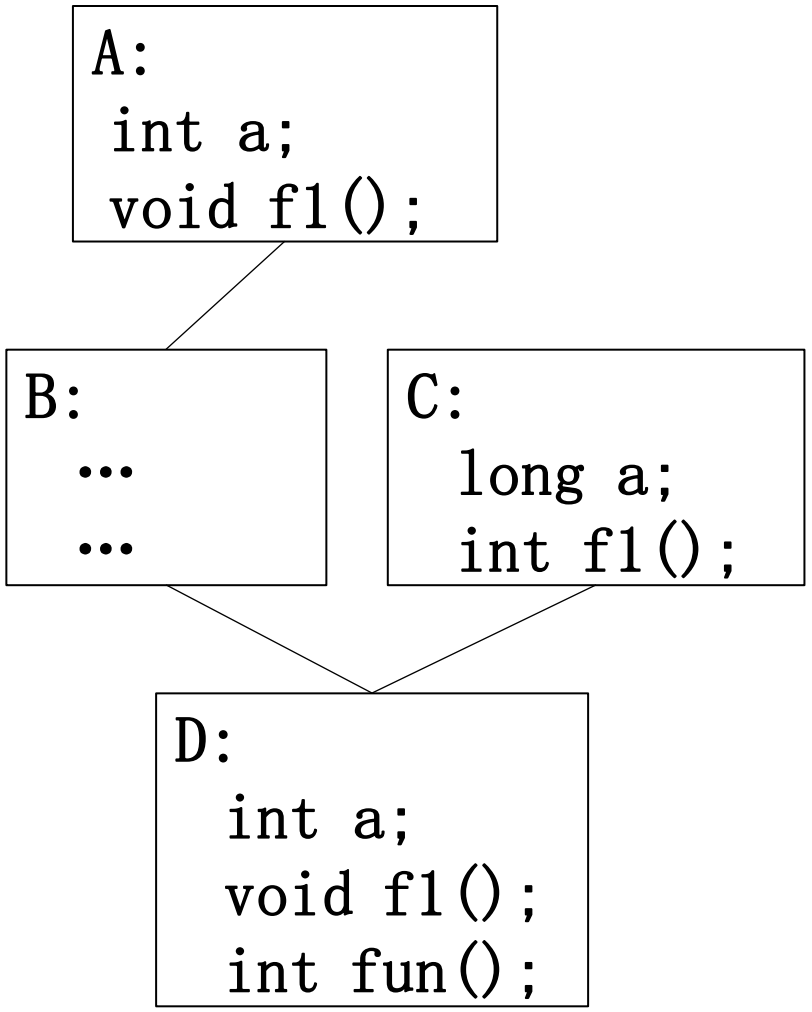
- 两个以上的基类与派生类的成员同名：派生类直接访问，不同基类加作用域符区分




```
int C::fun()
{
    a=10;
    f1();
    A::a=15;
    A::f1();
    B::a=20;
    B::f1();
}
```

```
int main()
{   C c1;
    c1.a=10;
    c1.f1();
    c1.A::a=15;
    c1.A::f1();
    c1.B::a=20;
    c1.B::f1();
}
```

- 通过直接/间接方式继承同一个基类两个导致的同名：通过可区分的类的作用域符来进行区分

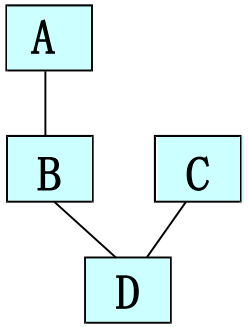



```
int D::fun()
{
   a=10;
   f1();
   A::a=15;可以B
   A::f1();可以B
   C::a=20;
   C::f1();
}
```

```
int main()
{  D d1;
   d1.a=10;
   d1.f1();
   d1.A::a=15;可以B
   d1.A::f1();可以B
   d1.C::a=20;
   d1.C::f1();
}
```

```
#include <iostream>
using namespace std;
class A {
  public:
    int a;
    void f1() { return; }
};
class B:public A {
  public:
    int b;
};
class C {
  public:
    long a;
    int f1() { return 0; }
};
class D:public B, public C {
  public:
    int a;
    char f1() { return 'A'; }
    void fun();
};
```

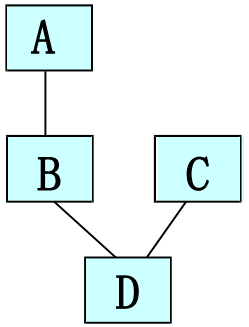
```
void D::fun(){
      a = 10;
      f1();
      A::a = 15; // B::a = 15;
      A::f1();   // B::f1();
      C::a = 20;
      C::f1();
      int *p = (int *)this;
      cout << sizeof(*this) <<endl;
      cout << *p << endl;
      cout << *(p+1) << endl;
      cout << *(p+2) << endl;
      cout << *(p+3) << endl;
}
int main(){
      D d1;
      d1.fun();
      return 0;
}
```




```
16
15
-858993460
20
10
```

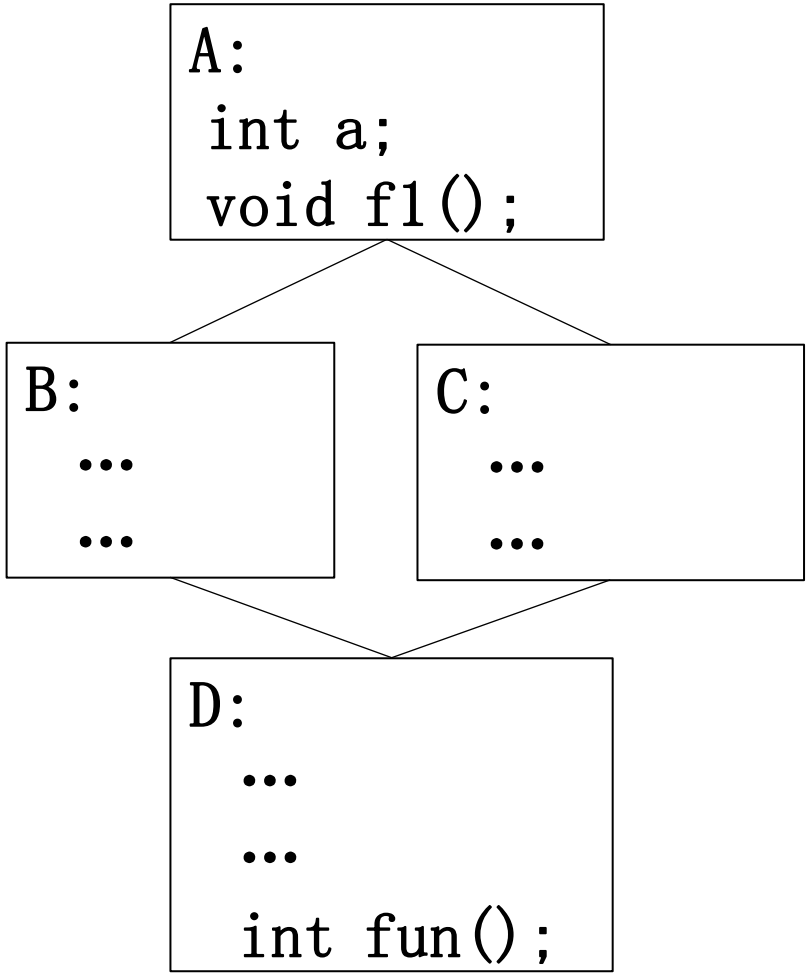
```
#include <iostream>
using namespace std;
class A {
  public:
    int a;
    void f1() { return; }
};
class B:public A {
  public:
    int b;
};
class C {
  public:
    long a;
    int f1() { return 0; }
};
class D:public B, public C {
  public:
    int a;
    char f1() { return 'A'; }
    void fun();
};
```

```
void D::fun(){
    return;
}
int main(){
    D d1;
    d1.a = 10;
    d1.f1();
    d1.A::a = 15;    // B::a = 15;
    d1.A::f1();      // B::f1();
    d1.C::a = 20;
    d1.C::f1();
    int *p = (int *)&d1;
    cout << sizeof(d1) <<endl;
    cout << *p << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;
    return 0;
}
```

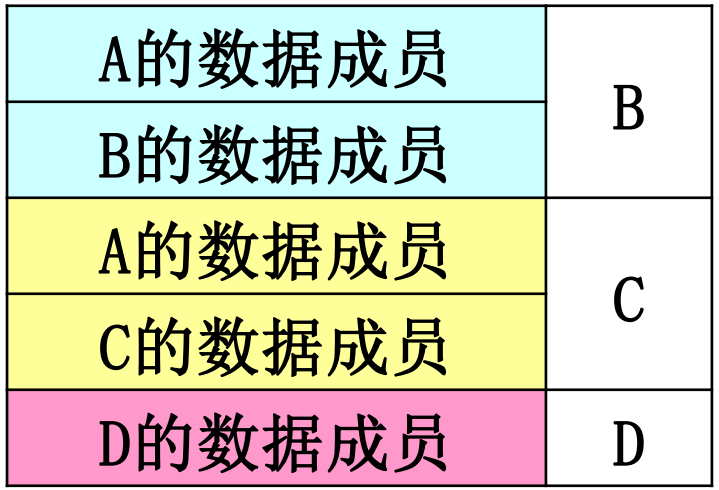



```
16
15
-858993460
20
10
```

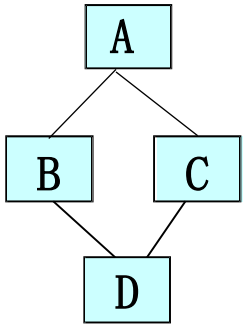
- 通过直接/间接方式继承同一个基类两个导致的同名：
  - 问题1：有几份A?



D对象的内存映象(两份A)



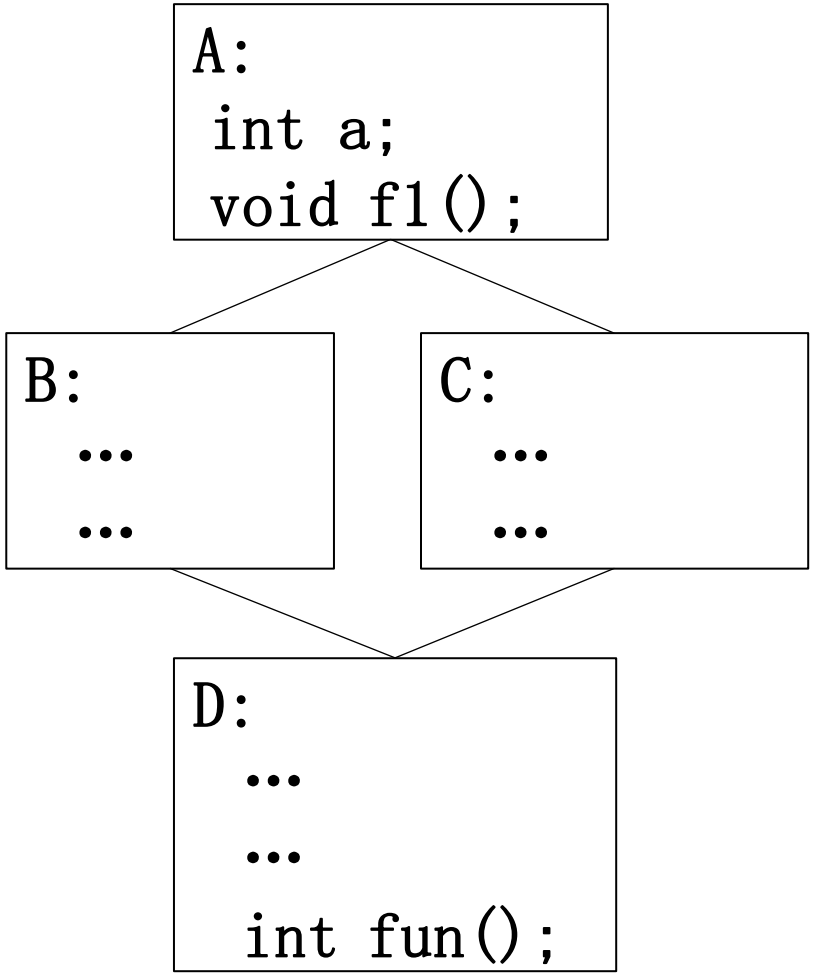
```
#include <iostream>
using namespace std;
static int x=0;
class A {
  public:
     int a;
     A() {
       a = ++x;
       cout << "A()" << endl; }
};
class B:public A {
  public:
     int b;
     B() {
       b = 20;
       cout << "B()" << endl; }
};
class C:public A {
  public:
     int c;
     C() {
       c = 30;
       cout << "C()" << endl; }
};
class D:public B, public C {
  public:
     int d;
     D(){
       d = 40;
       cout << "D()" <<endl; }
};
int main(){
  D d1;
  cout << sizeof(d1) << endl;
  int *p = (int *)&d1;
  cout << *p << endl;
  cout << *(p+1) << endl;
  cout << *(p+2) << endl;
  cout << *(p+3) << endl;
  cout << *(p+4) << endl;
  return 0;
}
```




```
A()
B()
A()
C()
D()
20
1
20
2
30
40
```

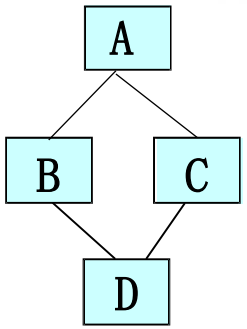
- 通过直接/间接方式继承同一个基类两个导致的同名：
  - 问题2：如何区分？

    通过可区分的类的作用域符来进行区分



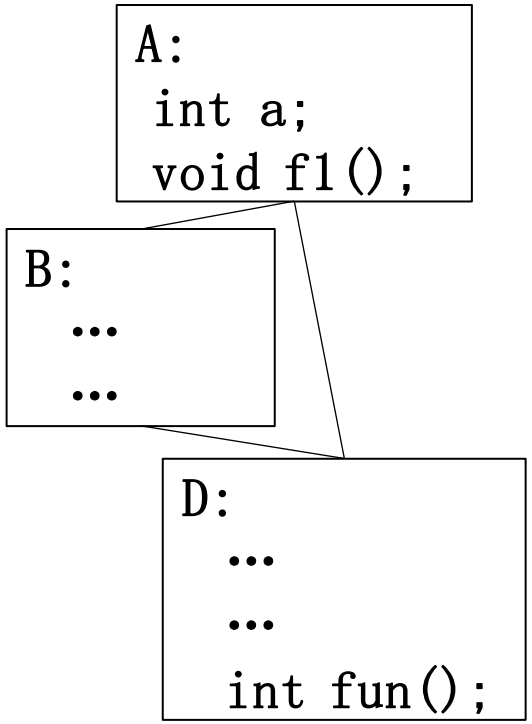

```
int D::fun()
{
    A::a = 5; //建议不用A::
    A::f1();  //建议不用A::
    B::a = 10;
    B::f1();
    C::a = 15;
    C::f1();
    return 0;
}
```

```
#include <iostream>
using namespace std;
static int x = 0;
class A {
public:
    int a;
    A() { a = ++x; }
    void f1() { cout << a << endl; }
};
class B :public A {
public:
    int b;
};
class C :public A {
public:
    int c;
};
class D :public B, public C {
public:
    int d;
    int fun();
};
int D::fun(){
    A::a = 5; //不建议
    A::f1();  //不建议
    B::a = 10;
    B::f1();
    C::a = 15;
    C::f1();
    return 0;
}
int main() {
    D d1;
    cout << sizeof(d1) << endl;
    d1.fun();
    d1.f1();   //报错：对f1的访问不明确
    d1.A::f1();//报错：对A的访问不明确
    d1.B::a=10;
    d1.B::f1();
    d1.C::a=15;
    d1.C::f1();
    return 0;
}
```



//删除报错语句后的运行结果：

```
20
5
10
15
10
15
```



D对象的内存映象(两份A)

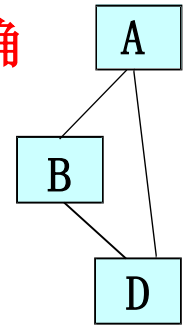
A的数据成员	B
B的数据成员	
A的数据成员	A
D的数据成员	D

```
#include <iostream>
using namespace std;
static int x=0;
class A {
  public:
    int a;
    A() {
      a = ++x;
      cout << "A()" << endl; }
};
class B :public A {
  public:
    int b;
    B() {
      b = 20;
      cout << "B()" << endl; }
};
```

```
class D :public B, public C {
  public:
    int d;
    D(){
      d = 40;
      cout << "D()" <<endl;}
};
int main(){
    D d1;
    cout << sizeof(d1) << endl;
    int *p = (int *)&d1;
    cout << *p << endl;
    cout << *(p+1) << endl;
    cout << *(p+2) << endl;
    cout << *(p+3) << endl;
    return 0;
}
```

```
A()
B()
A()
D()
16
1
20
2
40
```

```
#include <iostream>
using namespace std;
static int x = 0;
class A {
public:
    int a;
    A() { a = ++x; }
    void f1() { cout << a << endl; }
};
class B :public A {
public:
    int b;
};
class D :public B, public A {
public:
    int d;
    int fun();
};
int D::fun()
{
    A::a = 10; //不建议
    A::f1();   //不建议
    B::a = 15;
    B::f1();
    a = 15;    //报错：对a的访问不明确
    f1();      //报错：对f1的访问不明确
    return 0;
}
int main() {
    D d1;
    cout << sizeof(d1) << endl;
    d1.fun();
    d1.f1();   //报错：对f1的访问不明确
    d1.a = 15;  //报错：对a的访问不明确
    d1.A::a = 10; //不建议
    d1.A::f1();   //不建议
    d1.B::a = 15;
    d1.B::f1();
    return 0;
}
```



//删除报错语句后的运行结果：

16
10
15
10
15

目录

- 基本概念
- 派生类的声明方式及派生类对象的构成
- 派生类的成员访问属性
- 派生类的构造函数和析构函数
- 多重继承
- 虚基类
- 赋值兼容规则
- 继承和动态内存分配

8.6 虚基类

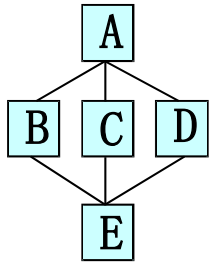
- 多重继承的问题：
  - 派生类中有多份相同的数据成员拷贝，占用较多的存储空间
  - 多个基类的相互交织可能会带来错综复杂的设计问题，命名冲突不可回避

➡引入虚基类，使相同基类只保留一份数据成员

- 虚基类声明的形式：

```
class 派生类名: virtual 存取限定符 基类名 {
    …
};
```

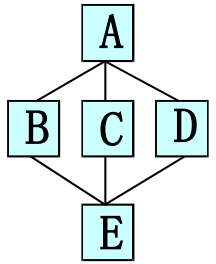
- 多重继承的构造函数调用




```
#include <iostream>
using namespace std;
class A {
  public:
    A() {cout << "A()" << endl;}
};
class B:public A {
  public:
    B() {cout << "B()" << endl;}
};
class C:public A {
  public:
    C() {cout << "C()" << endl;}
};
class D:public A {
  public:
    D() {cout << "D()" << endl;}
};
class E:public B,public C,public D{
  public:
    E() {cout << "E()" << endl;}
};
int main(){
    E e1;  //e1中有三份A的拷贝
}
```

```
A()
B()
A()
C()
A()
D()
E()
```

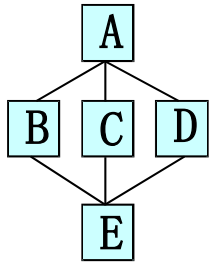
- 多重继承的构造函数调用：含虚基类




```
#include <iostream>
using namespace std;
class A {
  public:
    A() {cout << "A()" << endl;}
};
class B: virtual public A {
  public:
    B() {cout << "B()" << endl;}
};
class C: virtual public A {
  public:
    C() {cout << "C()" << endl;}
};
class D: virtual public A {
  public:
    D() {cout << "D()" << endl;}
};
class E:public B,public C,public D{
  public:
    E() {cout << "E()" << endl;}
};
int main(){
    E e1;  //e1中有一份A的拷贝
}
```

```
A()
B()
C()
D()
E()
```

- 多重继承的构造函数调用：含虚基类




```
#include <iostream>
using namespace std;
class A {
  public:
    A() {cout << "A()" << endl;}
};
class B: virtual public A {
  public:
    B() {cout << "B()" << endl;}
};
class C: virtual public A {
  public:
    C() {cout << "C()" << endl;}
};
```

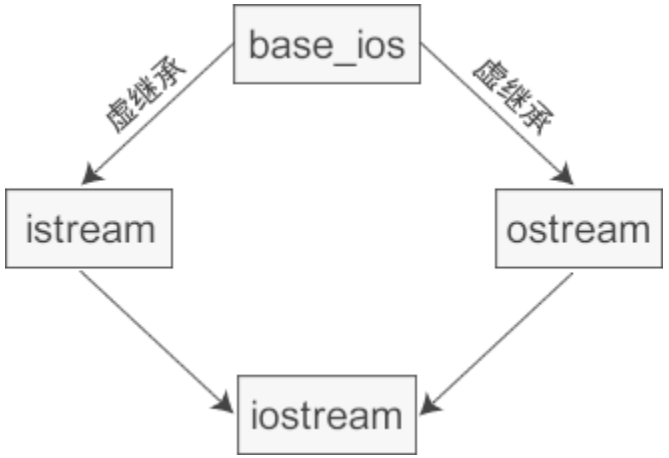
```
class D: public A {
  public:
    D() {cout << "D()" << endl;}
};
class E:public B,public C,public D{
  public:
    E() {cout << "E()" << endl;}
};
int main(){
    E e1;  //e1中有两份A的拷贝
}
```

所有继承该基类的直接派生类都应声明为虚基类才能保证只有一份数据拷贝

```
A()
B()
C()
A()
D()
E()
```

8.6 虚基类

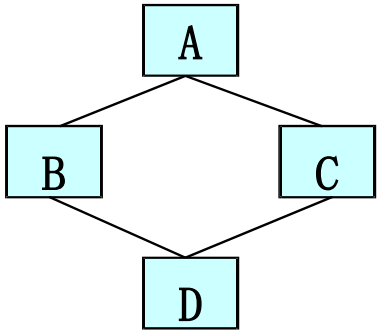
- 虚基类的实例：
  - C++标准库中的 iostream 类就是一个虚继承的实际应用案例。



  - iostream从istream和ostream直接继承而来，而istream和ostream又都继承自一个共同的名为base_ios的类，是典型的菱形继承
  - istream和ostream必须采用虚继承，否则将导致iostream 类中保留两份base_ios类的成员

8.6 虚基类

- 构造函数的冲突问题：
  - 给基类自动传递信息时，存在多条不同的途径



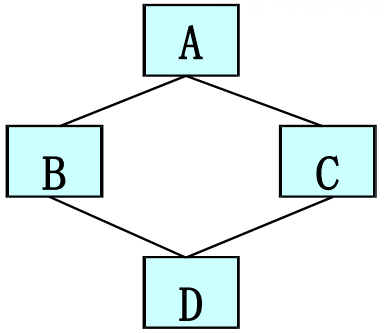
- 虚基类的构造函数：

  禁止信息通过中间类自动传递给基类

  - 使用虚基类时，必须在初始化列表最前方，显式的调用虚基类的构造函数；否则虚基类将调用默认构造函数初始化变量
  - 调用时，由派生类直接激活间接虚基类的构造函数，其直接基类不再自动激活虚基类的构造

- 禁止信息通过中间类自动传递给基类

```
class A {
   public:
      A() { cout << "A()" << endl; } // 无参构造
      A(int x) { cout << "A(" << x << ")" << endl; } //一参构造
};
class B:virtual public A {
   public:
     B(int y) { cout << "B()" << endl; }
};
class C:virtual public A {
   public:
     C(int z):A(z) { cout << "C()" << endl; }
};
class D:public B, public C {
   public:
      D(int x, int y, int z):A(x),B(y),C(z) {
              cout << "D()" << endl;}
};
```

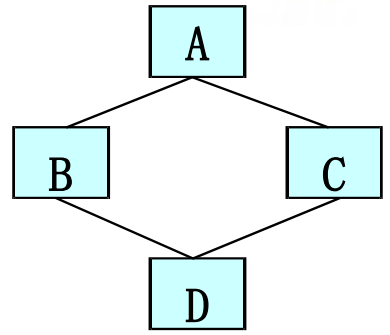



```
int main(){
  D d1(1,2,3);
  cout << endl;
  C c1(4);
  cout << endl;
  B b1(5);
}
```

```
A(1)
B()
C()
D()

A(4)
C()

A()
B()
```

- 禁止信息通过中间类自动传递给基类

```
class A {
   public:
      A() { cout << "A()" << endl; } // 无参构造
      A(int x) { cout << "A(" << x << ")" << endl; } //一参构造
};
class B:virtual public A {
   public:
     B(int y) { cout << "B()" << endl; }
};
class C:virtual public A {
   public:
     C(int z):A(z) { cout << "C()" << endl; }
};
class D:public B, public C {
   public:
      D(int x, int y, int z): B(y),C(z) {
               cout << "D()" << endl;}
};
```




```
int main(){
  D d1(1,2,3);
  cout << endl;
  C c1(4);
  cout << endl;
  B b1(5);
}
```

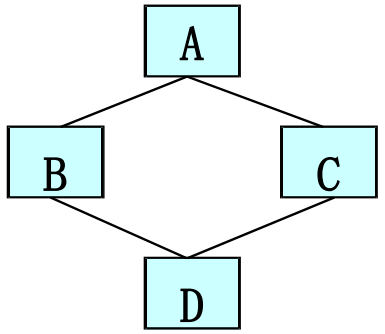
```
A()
B()
C()
D()

A(4)
C()

A()
B()
```

- 禁止信息通过中间类自动传递给基类

```
int main(){
  D d1(1,2,3);
  cout << endl;
  C c1(4);
  cout << endl;
  B b1(5);
}
```

➡ //d1对象生成时会自动激活A、B、C的构造函数，再调用D的构造函数。B/C的构造函数被调用时，不再自动激活A的构造函数

➡ //直接基类是虚基类时：与非虚基类一样，b1/c1生成时会自动激活A的无参/有参构造函数



目录

- 基本概念
- 派生类的声明方式及派生类对象的构成
- 派生类的成员访问属性
- 派生类的构造函数和析构函数
- 多重继承
- 虚基类
- 赋值兼容规则
- 继承和动态内存分配

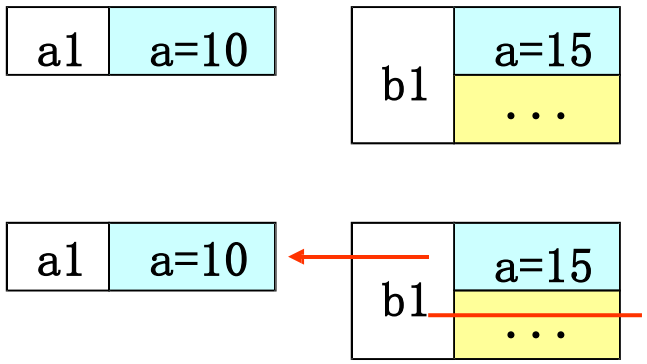
- 引例：基类与派生类的转换

```
#include <iostream>
using namespace std;
class A {
    public:
        int a;
};
class B:public A {
    public:
        int b;
};
```

```
int main()
{   A a1;
    B b1;
    a1.a = 10;
    b1.a = 15;
    cout << a1.a << endl;
    b1 = a1;  //编译错!!!
    cout << a1.a << endl;
}
```

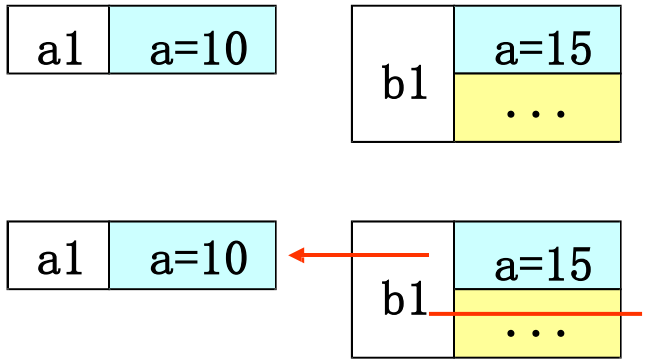
```
int main()
{   A a1;
    B b1;
    a1.a = 10;
    b1.a = 15;
    cout << a1.a << endl; 10
    a1 = b1;
    cout << a1.a << endl; 15
}
```

赋值兼容规则：



8.7 赋值兼容规则

- 赋值兼容规则：在需要基类对象的任何位置，均可以使用公有继承的派生类对象
  - ➢ 只有公有继承适用，私有及保护继承会导致基类的公有部分对外不可访问
  - ➢ 将派生类中的基类无名实例对象对应空间拷贝给基类，其余部分丢弃
    - ✓ 如果不希望直接对应拷贝，则可根据需要自行定义=重载或复制构造函数
    - ✓ 当基类中包含动态申请内存时，赋值兼容规则可能出错（参考=重载及复制构造函数）



- 派生类->基类：直接对应拷贝

```
class A {
    public:
        int a;
};
class B:public A {
    public:
      int b;
};
```

```
int main()
{
    B b1;
    b1.a = 15;
    A a1(b1); //复制构造
    A a2;
    a2 = b1;
    cout << a1.a << endl; //15
    cout << a2.a << endl; //15
}
```

- 派生类->基类：=重载

```
class B; //提前声明
class A {
  public:
      int a;
      A() {} //无参空体
      A(B &b);//不能体内实现
      A& operator=(B &b);//不能体内实现
};
class B:public A {
   public:
    int b;
};
A::A(B &b) //一参构造
{  a = b.a*2; }
```

```
A& A::operator=(B &b) //=重载
{   a = b.a*2;
    return (*this);
}
int main()
{
    B b1;
    b1.a = 15;
    A a1(b1), a2;
    //一参，无参构造
    a2 = b1; //= 重载
    cout << a1.a << endl; //30
    cout << a2.a << endl; //30
}
```

- 派生类->基类：动态内存申请 运行出错

```
class A {
   private:
     char *s;
   public:
     int a;
     A()
     {
        s = new char[20];
     }
     ~A()
     {
        cout << "A析构" << endl;
        delete s;
     }
};
```

```
class B:public A {
   public:
     int b;
     ~B() { cout << "B析构" << endl;}
};
int main()
{
   B b1;
   b1.a = 15;
   A a1(b1); //复制构造
}
```

8.7 赋值兼容规则

- 赋值兼容规则的使用位置：

  1. 派生类对象可初始化基类的对象或引用

```
//示例：
class A {…};
class B:public A {…};

B b1;
A a1(b1);    //需要A对象，用B对象替代
A &a2 = b1; //需要A对象，用B对象替代
```

8.7 赋值兼容规则

- 赋值兼容规则的使用位置：

  2. 派生类对象可出现在函数参数/返回值为基类的地方

```
//示例：
void f1(A  a1) { ... }  //形参为A对象
void f2(A &a1) { ... }  //形参为A对象的引用
void f3(A *pa) { ... }  //形参为A对象的指针
A f4(){ static B b1; return b1; //返回值为A对象 }
int main()
{   B b1;
    f1(b1); //B对象出现在要求A对象的位置
    f2(b1); //B对象出现在要求A对象引用的位置
    f3(&b1); //B对象出现在要求A对象指针的位置
}
```

8.7 赋值兼容规则

- 赋值兼容规则的使用位置：

3. 派生类对象可赋值给基类

```
//示例：
A a1;
B b1;
a1 = b1;  //需要A对象，用B对象替代
```

4. 派生类对象的指针可出现在基类指针出现的位置

```
//示例：
A *pa;
B b1, *pb = &b1;
pa = pb; //需要A对象的地址，用B对象的地址替代
pa = &b1; //需要A对象的地址，用B对象的地址替代
```

8.7 赋值兼容规则

- 赋值兼容规则：
  - 赋值兼容是实现多态的一个前提（模块09）
  - 赋值兼容规则+指针或引用+虚函数==多态（模块09）

目录

- 基本概念
- 派生类的声明方式及派生类对象的构成
- 派生类的成员访问属性
- 派生类的构造函数和析构函数
- 多重继承
- 虚基类
- 赋值兼容规则
- 继承和动态内存分配

8.8 继承和动态内存分配

- 引例

```
class baseDMA{    //基类使用了动态内存
private:
    char *label;
    int rating;
public:
    baseDMA(const char *lb = "null", int r = 0);
    baseDMA(const baseDMA &rs);                    //复制构造函数
    virtual ~baseDMA();                            //析构函数
    baseDMA & operator=(const baseDMA &rs);        //重载赋值运算符
…
};
```

8.8 继承和动态内存分配

- 情况1：派生类不使用new

```
class lacksDMA:public baseDMA{    //派生类
private:
    char color[40];
public:
…
};
```

不需要为lacksDMA类定义显示析构函数、复制构造函数和赋值运算符：

- lacksDMA成员不需执行任何特殊操作，所以默认析构函数是合适的
- lacksDMA类的默认复制构造函数使用显示baseDMA复制构造函数来复制lacksDMA对象的baseDMA部分，因此，默认复制构造函数也是合适的
- 默认的赋值构造函数将自动使用基类的赋值构造函数完成基类组件的赋值

8.8 继承和动态内存分配

- 情况2：派生类使用new

```
class hasDMA:public baseDMA{    //派生类
private:
    char *style;  //use new in constructors
public:
    ...
};
```

必须为hasDMA类定义显示析构函数、复制构造函数和赋值运算符！

8.8 继承和动态内存分配

- 情况2：派生类使用new
  - 显示析构函数

```
baseDMA::~baseDMA() //依赖于基类的析构函数来释放指针label管理的内存
{
   delete [] label;
}
hasDMA::~hasDMA()
{
   delete [] style;  //释放指针style管理的内存
}
```

8.8 继承和动态内存分配

- 情况2：派生类使用new
  - 复制构造函数

```
baseDMA::baseDMA(const baseDMA &rs){
    label = new char[strlen(rs.label) + 1];
    strcpy(label,rs.label);
    rating = rs.rating;
}
hasDMA::hasDMA(const hasDMA &hs):baseDMA(hs){
    style = new char[strlen(hs.style) + 1];
    strcpy(style,hs.style);
}
```

基类的指针或引用可以指向派生类的指针或引用

hasDMA复制构造函数只能访问hasDMA的数据，因此它必须调用baseDMA复制构造函数来处理baseDMA共享的数据

8.8 继承和动态内存分配

- 情况2：派生类使用new
  - 赋值运算符

```
baseDMA & baseDMA::operator=(const baseDMA &rs)  //基类
{
    if(this == &rs)
        return *this;
    delete [] label;
    label = new char[strlen(rs.label)+1];
    strcpy(label,rs.label);
    rating = rs.rating;
    return *this;
}
```

8.8 继承和动态内存分配

- 情况2：派生类使用new
  - 赋值运算符

```
hasDMA & hasDMA::operator=(const hasDMA &hs)    //派生类
{
        if(this == &hs)
            return *this;
        baseDMA::operator=(hs);  //copy base portion 函数表示法
        delete [] style;   //prepare for new style
        style = new char[strlen(hs.style)+1];
        strcpy(style,hs.style);
        return *this;
}
```

//含义为*this = hs; 运算符表示法
如按此写法，编译器将使用
hasDMA::operator=()，从而形成递归调用

8.8 继承和动态内存分配

- 情况2：派生类使用new（总结）
  - 显示析构函数：自动完成
  - 复制构造函数：在初始化成员列表中调用基类的复制构造函数完成；如果不这样做，将自动调用基类的默认构造函数
  - 赋值运算符：使用作用域解析运算符显示的调用基类的赋值运算符来完成
- 使用动态内存分配和友元的继承示例
  - primer书：集成baseDMA、lacksDMA、hasDMA类

```
//dma.h
//dma.cpp
//usedma.cpp
```

总结

- 基本概念（掌握）
- 派生类的声明方式及派生类对象的构成（掌握）
- 派生类的成员访问属性（掌握）
- 派生类的构造函数和析构函数（熟悉）
- 多重继承（熟悉）
- 虚基类（熟悉）
- 赋值兼容规则（熟悉）
- 继承和动态内存分配（了解）