第八章 继承与派生

模块8.1：继承与派生的基本概念

主讲教师：同济大学计算机科学与技术学院 陈宇飞
同济大学计算机科学与技术学院 龚晓亮

目录

- 继承与派生基础
- 派生类的成员访问属性
- 派生类的构造函数和析构函数

目录

- 继承与派生基础
  - 基本概念
  - 派生类的声明方式
  - 派生类对象的构成

1.1 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

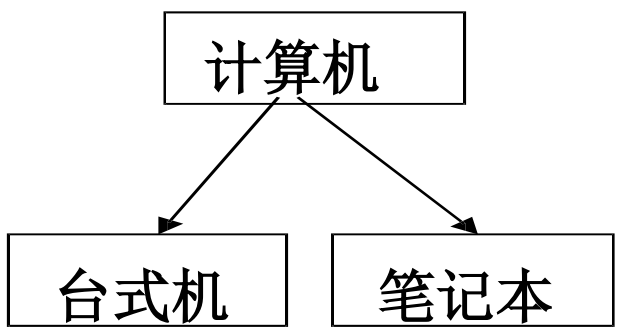
1.1 基本概念

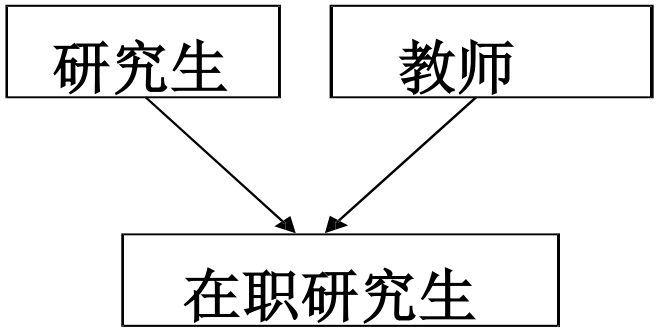
- 基类与派生类：基类(父类)，指已有的类

  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





1.1 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;
}
```

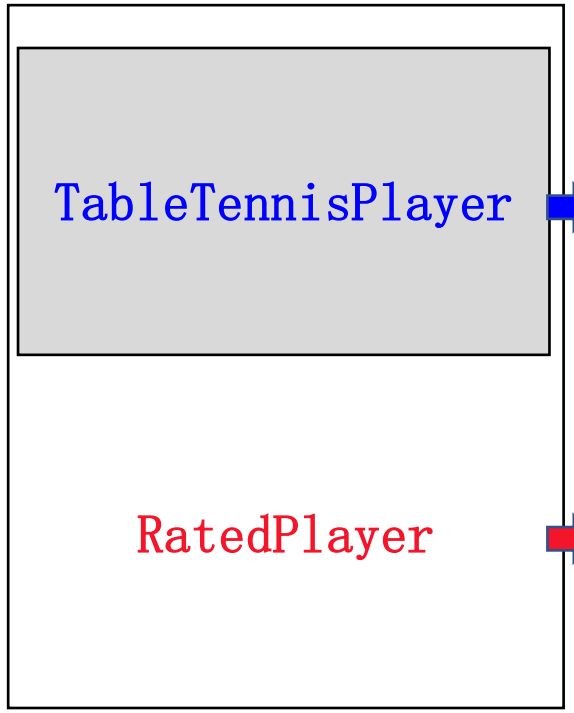
源程序文件
调用函数

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

1.1 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

1.1 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（后续讨论）。

```
//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

1.1 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。

1.2 派生类的声明方式

形式：

```
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

1.2 派生类的声明方式

形式：

```
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

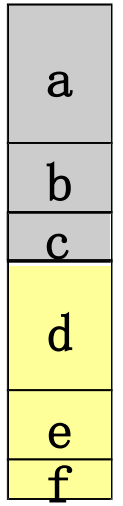
1.3 派生类对象的构成

```
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)

1.3 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（下一节讨论）

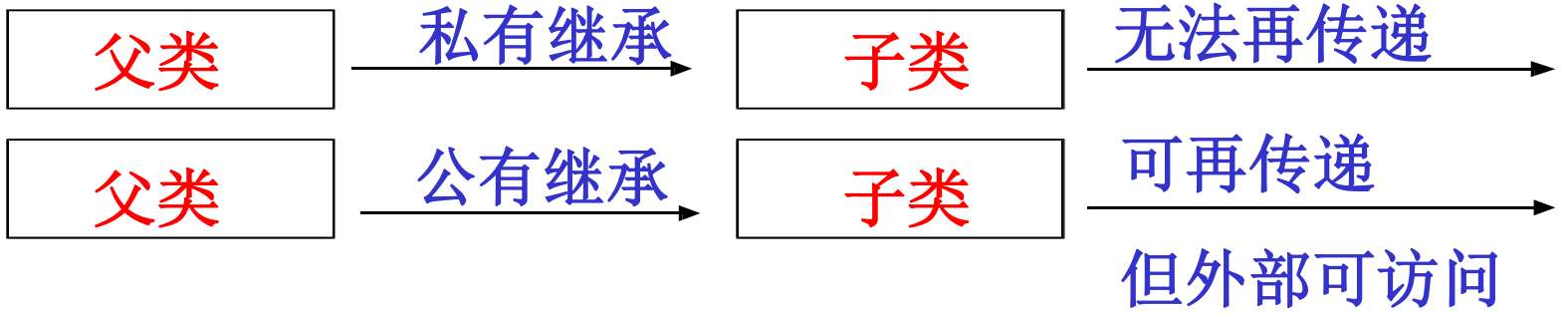
目录

- 派生类的成员访问属性
  - 继承访问控制权限
  - 继承方式与访问属性
  - 派生类的成员访问属性

2.1 继承访问控制权限

- 公有继承与私有继承

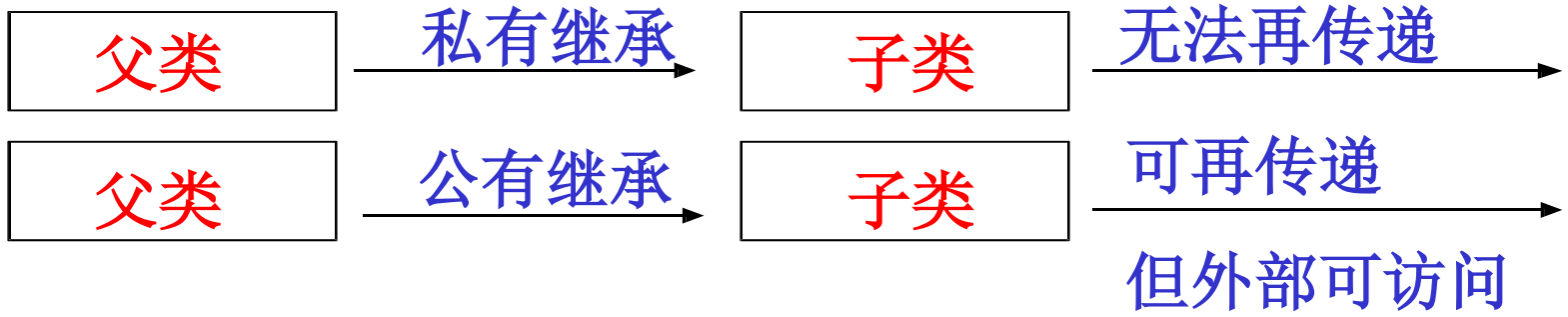
基类成员访问控制	继承访问控制	在派生类中的访问控制
public	public	public
private		不可访问
public	private	private
private		不可访问



2.1 继承访问控制权限

- 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

2.1 继承访问控制权限

- 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）

2.1 继承访问控制权限

- 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？

2.2 继承方式与访问属性

- 小结：

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; (×)

基类访问控制	继承访问控制	派生类访问控制
public	protected	protected
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; (×)

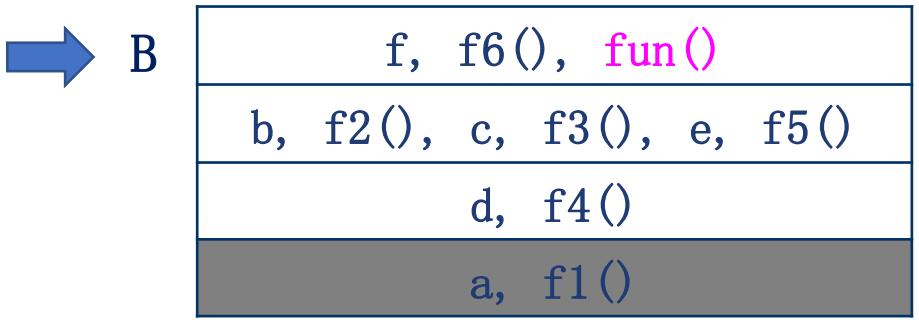
int k=C.y; (×)

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; (×)

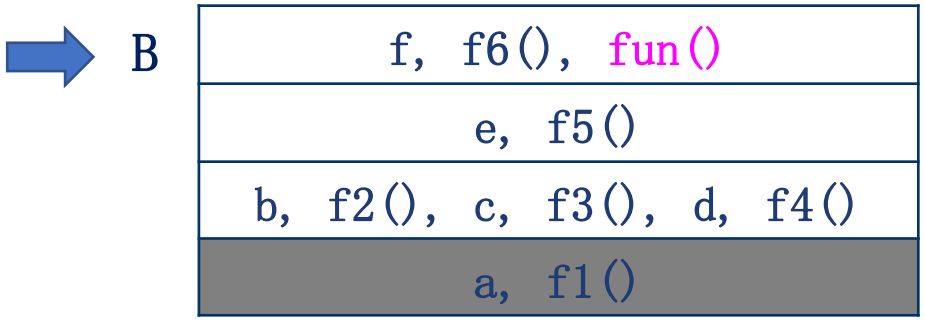
int k=C.y; (×)

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();
}
```

2.3 派生类的成员访问属性

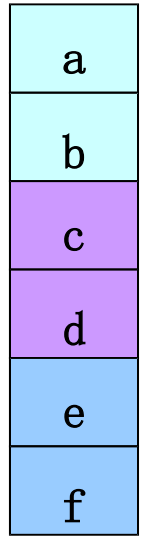
- 派生类的多级派生

```
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

2.3 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()

2.3 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);
}
```

2.3 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

- 派生类的构造函数和析构函数
  - 简单的派生类构造函数
  - 含有子对象的派生类构造函数
  - 多层派生时的构造函数
  - 派生类的析构函数

3.1 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)

3.1 简单的派生类构造函数

代码实现形式：

```
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; }
```

3.1 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)
```

3.2 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

3.3 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
```

3.4 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）

总结

- 继承与派生基础（掌握）
  - 基本概念
  - 派生类的声明方式
  - 派生类对象的构成
- 派生类的成员访问属性（掌握）
  - 继承访问控制权限
  - 继承方式与访问属性
  - 派生类的成员访问属性
- 派生类的构造函数和析构函数（熟悉）
  - 简单的派生类构造函数
  - 含有子对象的派生类构造函数
  - 多层派生时的构造函数
  - 派生类的析构函数