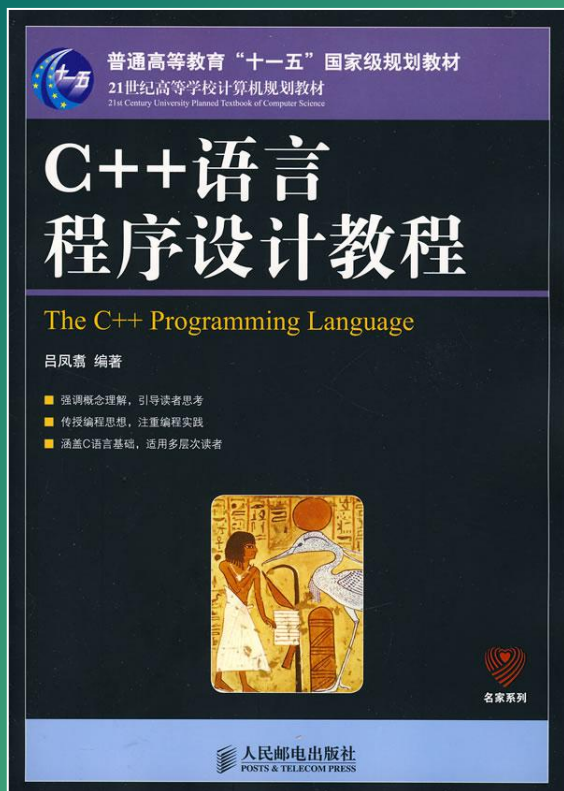


■ The C++ Programming Language



面向对象的程序设计

C++ 程序设计

第12讲 复习



§ 2.6 数组



THE C++ PROGRAMMING LANGUAGE

❖ 字符数组和字符串



- `char c1[5] = {'a', 'b', 'c', 'd'+1, 'e'-1};`

- `char c2[2][3] = {{ 'a', 'b', 'c' }, { 'd', 'e', 'f' } };`



- `char c3[] = "abcdef";` 或 `char c3[6] = "abcdef";`

- 能否写成:

- `char c3[7] = { 'a', 'b', 'c', 'd', 'e', 'f', '\0' }; ?`

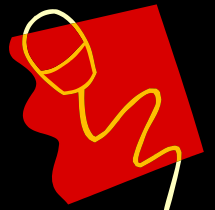


- `char c4[3][5] = {"abcd", "efgh", "ijkl"};`

- 表示3行5列元素，其中 `c[0]`、`c[1]`、`c[2]` 分别代表各行首地址

- 如 `cout << c[0] << endl;` 输出 "abcd"。





- 字符数据的使用方法
 - 字符数据和整型数据之间可以运算。
 - 字符数据与整型数据可以互相赋值。
- 字符串常量

例:"CHINA"

"a"

'a'

所以: `char c;`
`c="a";`

C	H	I	N	A	\0
---	---	---	---	---	----

a	\0
---	----

a



`char b=5; //给char赋值时要注意数值范围<128`

`int i='A' ;`

`int c=i+b;`





ASCII码表



L \ H	0000	0001	0010	0011	0100	0101	0110	0111
0000	NUL	DLE	SP	0	@	P	,	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2		R	b	r
0011	ETX							s
0100								t
0101								u
0110								v
0111								w
1000								x
1001								y
1010	LF	SD	*	:	S	Z	j	z
1011	VT	ESC	+	;	K	[k	{
1100	FF	FS	,	<	L	\	l	
1101	CR	GS	-	=	M]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

A: 0100 0001 = $2^6 + 1 = 65$

§ 4.3 指针和数组

THE C++ PROGRAMMING LANGUAGE

❖ 【例4.9】 编程对若干个字符串进行排序，可用冒泡排序法。

```
#include <iostream.h>
```

```
#include <string.h>
```

增加一个头文件string.h

```
void main( )
```

```
{
```

```
    char s[][8]={"while","switch","break","case","for","typedef"};
```

```
    char t[8];
```

```
    for(int i=0;i<5;i++)
```

```
        for(int j=5;j>i;j--)
```

```
            if(strcmp(s[j],s[j-1])<0)
```

```
            {
```

```
                strcpy(t,s[j]);
```

```
                strcpy(s[j],s[j-1]);
```

```
                strcpy(s[j-1],t);
```

```
            }
```

```
    for (i=0;i<6;i++)
```

```
        cout<<s[i]<<endl;
```

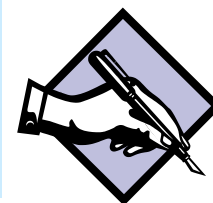
```
}
```

比较字符串的大小，即按
ASCII码顺序区分大小

当发现前一个字符串大于后
一个字符串时，两个字符串
交换位置。

程序输出：

```
break
case
for
switch
typedef
while
```





#include <string.h>

字符串长度函数

int strlen(char *s)

字符串比较函数

int strcmp(char *s1, char *s2)

❖ 字符串处理函数

■ 字符串连接函数

■ **char* strcat(char s1[], char s2[])**

■ 字符串复制函数

■ **char* strcpy(char s1[], char s2[])**





1

变量与表达式

2

语句和预处理

3

指针和引用

4

函数



§ 4.3 指针和数组



❖ 指针可表示数组元素

■ 二维数组元素的指针表示:

b是只读的指针，不能移动

- `int b[3][5];` 一般表示法 `b[i][j]`。
- 数组元素的地址表示法:
- `*(&b[0][0]+5*i+j)`, 其中 $i=0\sim 2, j=0\sim 4$ 。
- 指针表示法: `*(* (b+i)+j)`

❖ 行数组用指针，列数组用下标

- `(* (b+i)) [j]`

❖ 行数组用下标，列数组用指针

- `*(b[i]+j)`

为什么不是 `*(b+5i+j)` ?

`b00 b01 b02 b03 b04`

`b10 b11 b12 b13 b14`

`b20 b21 b22 b23 b24`

指针不仅有地址，而且有级别和大小！

二维数组地址的表示形式:

`&b[i][j], b[i]+j, *(b+i)+j, &b[0][0]+5*i+j, &(* (b+i)) [j]`



§ 4.3 指针和数组



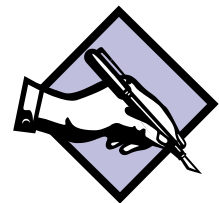
❖ 指向数组的指针和指针数组

■ 指向一维数组元素的指针：（表示一个二维数组）

■ `int a[2][5] = {0,1,2,3,4,10,11,12,13,14};`

■ `int (*pa)[5]=a; // 这样pa就与a一样使用`

■ 因为a是数组的地址，其指针大小为 `5*sizeof(int)`。



§ 4.3 指针和数组



❖ 指向数组的指针和指针数组

■ 指针数组：（表示一个指针的数组）

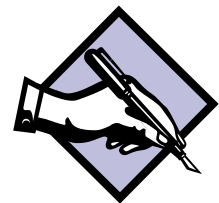
■ `char* s1 = "point";`

■ `char* s2 = "array";`

■ `char *ps[2];`

■ `ps[0] = s1; ➔ *ps = s1; // 效果相同`

■ `ps[1] = s2; ➔ *(ps+1) = s2; // 效果相同`



§ 4.4 引用



THE C++ PROGRAMMING LANGUAGE

❖ 【例4.14】 分析下列程序的输出结果，熟悉引用特性。

```
#include <iostream.h>
```

```
void main( )
```

```
{
```

```
    int *p;
```

```
    int*& rp = p;
```

```
    int a = 15;
```

```
    p = &a;
```

```
    cout << *p << ' , ' << *rp << endl;
```

```
    int b = 10;    rp = &b;
```

```
    cout << *p << ' , ' << *rp << endl;
```

```
    int c[] = {1,2,3,4};
```

```
    int& rc = c[2];
```

```
    rc = 8;
```

```
    cout << c[2] << endl;
```

```
}
```

关于指针的引用。

具有相同的地址，因此有相同的值

改变指针的值

指向数组元素

程序输出：

15,15

10,10

8





❖ 对象引用

- 对象引用常用来作函数的形参。当函数形参为对象引用时，则要求实参为对象名，实现引用调用。

❖ `A a(5);`

❖ `A &ra = a;`

❖ `A a(7);`

❖ `const A &ra = a;`

引用调用可以在被调用的函数中通过引用来改变调用函数中参数的值，为了避免这种改变，可以使用对象的常引用作形参。



§ 7.1 对象指针和对象引用



注意：函数参数中的对象常引用是保证对该对象不作修改。

❖ 【例7.5】分析下列程序的输出

```
#include <iostream.h>
class A
{
public:
    A(int i,int j)
    { x=i; y=j; }
    A()
    { x=y=0; }
    void Setxy(int i,int j)
    { x=i; y=j; }
    void Copy(const A &a);
    void Print()
    { cout << x << ' , ' << y << endl; }
private:
    int x,y;
};
```

10,15

20,25

```
void A::Copy(const A &a)
{
    x=a.x; y=a.y;
}
void fun(A *a1,A &ra)
{
    a1->Setxy(10,15);
    ra.Setxy(20,25);
}

void main()
{
    A a(5,8),b;
    b.Copy(a);
    fun(&a,b);
    a.Print();
    b.Print();
}
```



§ 4.4 引用



❖ 指针是变量，引用不是变量

❖ 指针可以引用，引用不可以引用

```
int *p; // p是一个指向int型变量的指针
int*& rp = p; // rp是一个指针p的引用
int a = 15;
rp = &a; // 给rp赋一个变量a的地址值
```

❖ 指针可以作数组元素，引用不可以作数组元素

```
int a[3];
int& refa[3] = a; // 错误!
```

■ 但可以这样:

```
int* p = a;
int*& rp = p; // 作为指针p的一个别名
```

❖ 可以有空指针，不能有空引用





1

变量与表达式

2

语句和预处理

3

指针和引用

4

函数



§ 5.3 函数的调用



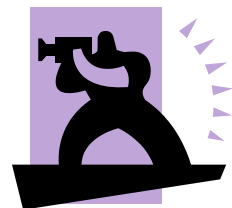
❖ 函数的传值调用

□ 传值调用方式

- `int fun(double a);` // 形参用变量名
- `int x = 5;`
- `int z = fun(x+1);` // 实参用表达式
- 将实参拷贝副本给形参，函数内部无法修改实参的值！

□ 传址调用方式

- `int fun(double* pa);` // 形参用变量的指针
- `int x = 5;`
- `int z = fun(&x);` // 实参的地址传到函数中
- 用形参指针指向实参变量，有可能修改实参的值！



§ 5.3 函数的调用



❖ 函数的引用调用

□ 引用调用方式

- `int fun(int& a);` // 形参用引用名
- `int x = 5;`
- `int z = fun(x);` // 实参用变量名
- 将实参的变量名传给形参实现引用。

- **注意：**引用调用实质上是传递地址，引用的值被改变即所引用的变量的值被改变！





§ 5.2 函数的参数和返回值



函数三种返回值形式

- 数值类型 //返回一个**值**
 - `int fun()`
 - `{static int a=10;return a;}`
- 指针类型 //返回一个**地址值**
 - `int * fun()`
 - `{static int a=10;return &a;}`
- 引用类型 //返回一个**变量**
 - `int & fun()`
 - `{static int a=10; return a;}`
- 指针、引用：注意返回生存期有效的变量。



§ 5.2 指针和引用作函数的参数和返回值

THE C++ PROGRAMMING LANGUAGE



❖ 【例】分析下列程序的输出结果，熟悉引用作函数返回值的使用方法。

```
#include <iostream.h>
```

```
int& fun(int);
```

```
void main()
```

```
{
```

```
    int x(5), y(8);
```

```
    int s1 = fun(x)++;
```

```
    int s2 = fun(y);
```

```
    cout << s1 << '\n' << s2 << endl;
```

```
}
```

```
int& fun(int a)
```

```
{
```

```
    static int t;
```

```
    t = 2*a;
```

```
    return t;
```

```
}
```

引用作函数的返回值

2. 换成 `int s1=++fun(x)` 会出现什么现象?
++作用在什么地方?

建议：static变量最好给初始值！

程序输出：

10,16





1

类和对象

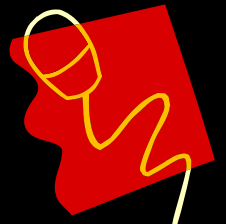
2

继承性与派生类

3

多态性和虚函数





对象创建

分配对象空间

构造函数（数据空间初始化） `Student(...){...}`

`Student s(10,"zhang");`

使用

析构函数（主要是堆清理） `~Student(){...}`

对象销毁

释放对象空间





1. 默认构造函数

```
class Date
{public:
    Date();    .....
};
```

2. 带参数的构造函数

构造函数可以带有一个或者多个参数。

```
class Date
{public:
    Date(int y, int m, int d);    .....
};
```

3. 拷贝构造函数

```
class Date
{public:
    Date(Date& date);    .....
};
```





对象数组和对象指针数组

中国传媒大学版权所有

- ❖ **Student a[10];** //调用10次无参构造
- ❖ **Student b[10]={Student("zhang")};**
//首先b[0]被分配内存,调用有参构造函数,
b[1]~b[9]: 分配内存后, 各自调用1次无参构造函数
- ❖ **Student *p[10];**
//创建10个指针空间, 没有对象创建, 不会调用任何构造函数
- ❖ **Student (*p)[10];**
//数组指针, 没有对象创建, 不调用构造函数





```
class Tdate
```

```
{
```

```
    public:
```

```
        Tdate(int m,int d,int y);
```

```
    private:
```

```
        int month;
```

```
        int day;
```

```
        int year;
```

```
};
```

```
Void main()
```

```
{
```

```
    Tdate *pD;
```

```
    pD=new Tdate(02,12,2010);
```

```
    .....
```

```
    delete pD;
```

```
}
```

没有默认的构造函数，
会报错

如果带参的构造函数
都有默认值也可以

pD=new Tdate;



§ 5.2 函数的参数和返回值



❖ 设置函数参数的默认值

■ `int fun(int a, int b=8, int c=10);`

- ① 多参数时，可以全部或部分设置默认值。
- ② 给定默认参数时必须从右到左设置。
- ③ 默认参数值只能在说明语句中给出。
- ④ 设置默认参数值时可以是相同类型的表达式。
- ⑤ 在函数调用时，对应参数如果有实参值，则将该实参值取代设置的默认值；如果没有给定实参值，则用参数的默认值。



§ 5.2 函数的参数和返回值



【例5.3】分析下列程序的输出结果，熟悉设置函数参数默认值的方法。

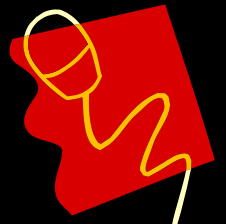
```
#include <iostream.h>
void fun(int a=1,int b=2,int c=3)
{
    cout<<"a="<<a<<"\t" <<"b="<<b<<"\t" <<"c="<<c<<endl;
}
void main( )
{
    fun();
    fun(9); // a=9,b=2,c=3
    fun(4,5); // a=4,b=5,c=3
    fun(7,8,9); // a=7,b=8,c=9
}
```

实参不写时使用默认值

程序输出:

```
a=1,b=2,c=3
a=9,b=2,c=3
a=4,b=5,c=3
a=7,b=8,c=9
```





1.

```
Point fun2()
{   Point A(1,2);
    return A; //调用拷贝构造函数
}
```

2.

```
int main()
{   Point B;
    B=fun2();
}
```

3.

```
cout<<p.GetX()<<endl;}
void main()
{
    Point A(1,2);
    fun1(A); //调用拷贝构造函数
}
```



对象的初始化



- ❖ **【例6.3】** 分析下列程序的输出结果，熟悉拷贝构造函数的用法及对象赋值的含义。

```
#include <iostream.h>
```

```
class Point
```

```
{
```

```
public:
```

```
    Point(int i, int j)
```

```
    { X=i; Y=j; }
```

```
    Point(Point &rp);
```

```
    ~Point()
```

```
    { cout<<"Destructor called.\n"; }
```

```
    int Xcood()
```

```
    { return X; }
```

```
    int Ycood()
```

```
    { return Y; }
```

```
private:
```

```
    int X,Y;
```

```
};
```

带参数的构造函数

拷贝构造函数

析构函数



对象的初始化



THE C++ PROGRAMMING LANGUAGE

```
Point::Point(Point &rp)
{
    X = rp.X;
    Y = rp.Y;
    cout<<"Copy Constructor called.\n";
}
void main()
{
    Point p1(6,9);
    Point p2(p1);
    Point p3 = p2, p4(0,0);
    p4 = p1;
    cout<<"p3= ("<<p3.Xcood() <<
    cout<<"p4= ("<<p4.Xcood() <<
}
```

程序输出:

```
Copy Constructor called.
Copy Constructor called.
p3=(6,9)
p4=(6,9)
Destructor called.
Destructor called.
Destructor called.
Destructor called.
```





□ 何时需要虚析构造函数？

□ 当你可能通过基类指针删除派生类对象时。

```
class A
{
public:
    virtual ~A()
    {cout<<"A::~~A()  called.\n";}
};
```

```
class B:public A
{
public:
    B(int i)
    { buffer = new char[i]; }
    ~B()
    { delete [] buffer;
    cout<<"B::~~B()  called.\n";}
private:
    char *buffer;
};
```

```
void fun(A *a)
{
    delete a;
}
```

```
void main()
{
    B *b = new B(5);
    fun(b);
}
```

输出结果：

B::~~B() called.
A::~~A() called.



§ 9.4.2 堆对象



2. 使用运算符 **delete** 释放对象

■ 使用 **delete** 运算符释放一个对象或其他类型变量:

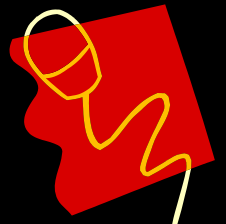
```
A *pa = new A(3,5);  
delete pa;  
int *p = new int(8);  
delete p;
```

■ 使用 **delete** 运算符释放一个对象数组或其他类型数组:

```
int *parray = new A[10];  
delete[] A;
```

new和**delete**成对出现,
加[]连续申请的空间被释放





使用**new**如果创建成功，则**new**表达式具有非0的地址值，并将它赋同一类型的指针。如果创建失败，则**new**表达式的值为0.

```
A *parray;
```

```
parray[0]=A(1,2);
```

```
parray=new A[10];
```

```
parray[1]=A(3,4);
```

```
parray[2]=A(5,6);
```

```
if(parray==NULL)
```

```
parray[3]=A(7,8);
```

```
{
```

```
    cout<<"数组创建失败！ /n";
```

```
    exit(1);
```

```
}
```





❖ 静态数据成员的特点

- ① 静态数据成员不是属于某个对象，而是属于整个类的。
 - 静态数据成员具有共享性和唯一性

- ② 静态数据成员不随对象的创建而分配内存空间，它也不随对象被释放而撤销。只有在程序结束时才被系统释放。
 - 静态数据成员具有文件生命期

- ③ 静态数据成员只能在类体外被初始化。
 - 静态数据成员必须初始化。





```
class Student
```

```
{ static int count; //类的所有对象共享这块数据空间
```

```
    int id;        //对象自己的空间
```

```
    char name[10]; //对象自己的空间
```

```
public:
```

```
    static int GetCount(){return count;}
```

```
    //静态函数成员访问静态数据成员
```

```
    static int SetCount(){count++;}
```

```
    //静态函数成员访问静态数据成员
```

```
};
```

静态成员在类外初始化:

```
int Student::count=0;
```

静态成员的访问方式（类名，对象）

数据成员：类名： `cout<<Student::count;` // 输出初始值0

对象： `Student s1,s2; s1.count=9;`

`cout<<s2.count;` //输出9

函数成员：类名： `Student::GetCount();`

对象： `s1.GetCount();`



常数据成员



```
class Student  
{public:
```

```
    Student():MAX1(100)//无参构造函数中的初始化
```

```
    { }
```

```
    Student(int max):MAX1(max)//有参构造函数中的初始化
```

```
    { }
```

```
    Student(Student &r):MAX1(r.MAX1)//拷贝构造函数中的初始化
```

```
    { }
```

```
private:
```

```
    const int MAX1;
```

```
    static const int MAX2;
```

```
};
```

```
const int Student::MAX2=20;
```

const数据成员:

通过初始化后确定该内存的初值, 此后不能
通过赋值方式修改其值, 否则编译出错。

const数据成员初始化位置:

每个构造函数初始化列表, 保证以不同方
式创建的每个对象的常成员都会被初始化

static const数据成员初始化位置:

不从属于某个对象, 放在类外初始化



```
class Student
```

```
{public:
```

```
    Student():MAX(100){...}
```

```
    void Print( ) const
```

```
{ cout<<id<<name<<endl; //可以访问非const成员
```

```
//id=10; //试图在常函数内修改数据成员，编译出错!
```

```
}
```

```
private:
```

```
    int id;
```

```
    char name[10];
```

```
    const int MAX;
```

```
};
```

1. 常成员函数内可以使用所有数据成员，但不能修改任何数据成员。
2. const函数内不能调用非const函数

1. const Student s("zhang",1001);
2. 以上定义常对象s，其对象内存不可被修改
3. 因此常对象S只能调用const函数，以防止通过函数成员间接修改对象数据成员的情况发生。

§ 6.5 常成员

❖ 常成员函数

- 常成员函数保证不修改任何外部变量。
- 常成员函数可以引用任何数据成员，但不可改变外部的变量。
- 只能调用常函数。

```
class B
{
public:
    B(int i, int j)
    { b1 = i; b2 = j; }
    void Print()
    { cout << b1 << ';' << b2 << endl; }
    void Print() const
    { cout << b1 << ':' << b2 << endl; }
private:
    int b1, b2;
};
```

.....

```
void main()
{
    B b1(5,10);
    b1.Print();
    const B b2(2,8);
    b2.Print();
}
```

常函数！只有当常对象时调用

程序输出：

5;10
2:8





- ❖ **Student a,b;**
- ❖ **Student *p=&a;**
- ❖ **Student * const p1=&a;** //指向对象的**常指针**
- ❖ **const Student * p2=&a;** //指向**常对象**的指针
 - **常对象指针只能调用常函数成员**
- ❖ **this指针：非静态成员函数的隐藏参数**



§ 7.1 对象指针和对象引用



THE C++ PROGRAMMING LANGUAGE

❖ 指向常对象的指针

❖ `const Date d1(2005,7,1);`

❖ `const Date *pd = &d1;`

对象指针指向常对象。

❖ 常对象必须用常指针指向它

❖ 常对象指针也可以指向一般对象，但该指针所指内容不可被改变。

❖ 用常指针引用的成员函数应该是常成员函数。

❖ 常对象指针只是其内容（数据成员的值）不可改变，指针所指地址值可以被改变。

❖ 常对象指针可以用于函数参数。





- ❖ 在类**A**中声明**A**的友元函数**fun**和友元类**B**:
- ❖ **class A**
- ❖ **{**
- ❖ **int num;**
- ❖ **public:**
- ❖ **friend void fun(A &a);**
- ❖ **friend class B;**
- ❖ **};**//可以在友元中访问**A**的私有成员





下面关于友元的描述正确的是

(**A**)

- A.** 友元函数不是类的成员，但可以访问所在类的任何成员。
- B.** 友元函数中能访问所在类的私有成员但不能访问类的保护成员。
- C.** 如果一个类成为另一个类的友元类，则两个类可以互相访问对方的私有成员。
- D.** 一个类与其他类的友元关系可以被其派生类继承。





2.类体内定义与实现

```
class Point {  
    private:  
        int X,Y;  
    public:  
        void SetPosition( int a,int b);  
        // void SetPosition( int a,int b){ X=a;Y=b;}  
};
```

```
inline void Point :: SetPosition ( int a, int b )  
{  
    X=a;  
    Y=b;  
}
```





- ❖ 在类体内部直接给出函数的实现或类外加**inline**关键字的函数实现
- ❖ 编译时在调用处用函数体进行替换,节省了参数传递、控制转移等开销。
- ❖ 注意:
 - 内联函数体内不能有**循环语句**和**switch**语句。
 - 内联函数的声明必须出现在内联函数第一次被调用之前。





1

类和对象

2

继承性与派生类

3

多态性和虚函数





- ❖ 派生类的构造函数
- ❖ 派生类的构造函数应该包含它的直接基类的构造函数。
 - 先执行基类构造函数；
 - 再执行子对象的构造函数（如有子对象的话）；
 - 最后执行派生类构造函数的函数体。

```
class A // C的基类
{ public: A(); };
class B
{ public: B(); };
class C : public A // A的派生类
{
public: C();
private: B b; // B为C的子对象
};
```

C c;
当用C构建对象c时，ABC各类的构造函数的调用顺序是：
 $A() \rightarrow B() \rightarrow C()$





❖ 派生类的析构函数

❖ 由于析构函数不能继承，因此在派生类的析构函数中要包含它的**直接基类**的析构函数。

- 先执行**派生类**析构函数的函数体。
- 再执行**子对象**所在类的析构函数（如果有子对象的话）
- 最后执行直接**基类**中的析构函数。

```
class A // C的基类
{ public: ~A(); };
class B
{ public: ~B(); };
class C : public A // A的派生类
{
public:   ~C();
private: B b; // B为C的子对象
};
```

C c;

当对象c析构时，ABC各类的析构函数的调用顺序是：

$\sim C() \rightarrow \sim B() \rightarrow \sim A()$



§ 8.2 单重继承



THE C++ PROGRAMMING LANGUAGE

❖ 【例8.5】分析下列程序的输出结果，熟悉派生类构造函数的定义格式。

```
#include <iostream.h>
#include <string.h>
class Student
{
    char name[20];
    int stuno;
    char sex;
public:
    Student(int no, char *str, char s)
    {
        stuno = no;
        strcpy(name, str);
        sex=s;
        cout<<"Constructor called.S\n";
    }
    void Print()
    { cout<<stuno<<'\\t'<<name<<'\\t'<<sex<<'\\t'; }
};
```



§ 8.2 单重继承



THE C++ PROGRAMMING LANGUAGE

```
class Student1:public Student
{
public:
    Student1(int no,char *str,char s,int a,int sco)
        : Student(no,str,s)
    {
        age=a;
        score=sco;
        cout<<"Constructor called.S1\n"
    }
    void Print()
    {
        Student::Print();
        cout<<age<<"\t"<<score<<endl;
    }
private:
    int age,score;
};
```

```
construct called.S
construct called.S1
construct called.S
construct called.S1
502001 Ma Li m 20 90
502002 Li Hua f 19 88
```

同名覆盖，如果没有Student: :，派
生类对象将调用自身的Print()函数



赋值兼容规则



当类型A是类型B的子类时，则满足下述的赋值兼容规则。**A是B的派生类，B是A的基类。**

① A类的对象可以赋值给B类的对象。**派生类的对象可以赋值给基类类型的对象。**

② A类的对象可以给B类对象引用赋值。**派生类对象可以给基类的对象引用赋值。**

③ A类的对象地址值可以给B类对象指针赋值。**派生类对象指针可以给基类对象指针赋值，即基类对象指针可以指向派生类的对象。**

例如：

```
class A:public B
A a;B b;A *pa;B *pb;
b = a;
B &rb = a;
pb =&a; pb = pa;
```

例8.8 分析下列程序的输出结果。

```
#include<iostream.h>
```

```
class A
```

```
{public: A()      {a=0;}
```

```
      A(int i)  {a=i;}
```

```
      void Print()      {cout<<a<<endl;}
```

```
      int Geta()        {return a;}
```

```
private: int a;
```

```
};
```

```
class B:public A
```

```
{public: B()      {b=0;}
```

```
      B(int i,int j):A(i),b(j) { }
```

```
      void Print()
```

```
      {          cout<<b<<',';
```

```
                A::Print();
```

```
      }
```

```
private: int b;
```

```
};
```

```
void fun(A &a)
```

```
{          cout<<a.Geta()+2<<',';
```

```
          a.Print();
```

```
}
```

```
void main()
```

```
{
```

```
    A a1(10),a2;
```

```
    B b(10,20);
```

```
    b.Print();
```

```
    a2=b;
```

```
    a2.Print();
```

```
    A *pa=new A(15);
```

```
    B *pb=new B(15,25);
```

```
    pa=pb;
```

```
    pa->Print();
```

```
    fun(*pb);
```

```
    delete pa;
```

```
}
```

20, 10

10

15

17,15





1

类和对象

2

继承性与派生类

3

多态性和虚函数



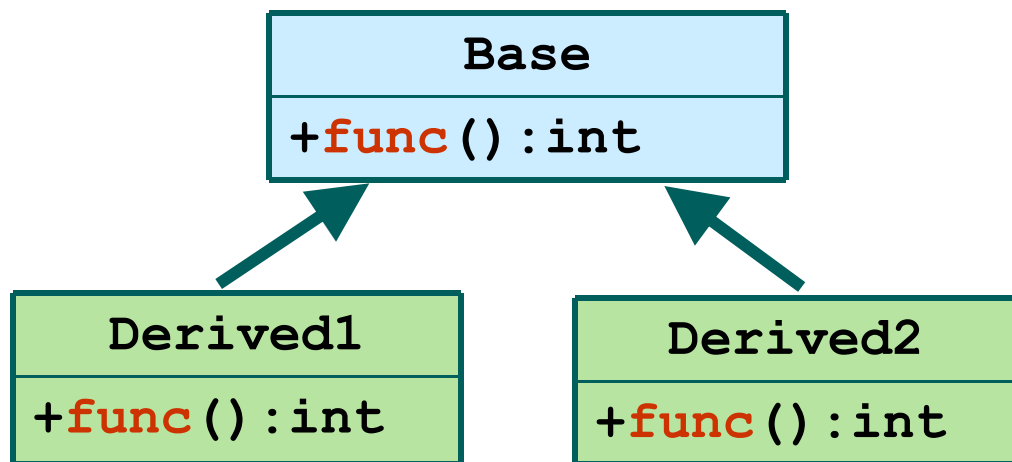
§ 9.2 静态联编和动态联编

THE C++ PROGRAMMING LANGUAGE

❖ 联编的概念：（也称绑定 $binding$ ）

- “联编”的含义是指对于相同名字的若干个函数的选择问题。
- 联编可分为静态联编（ $static\ binding$ ）和动态联编（ $dynamic\ binding$ ）两种。静态联编是在编译时进行的，又称早期联编。动态联编是在运行阶段进行的，又称滞后联编。

❖ 为什么会出现“同名函数选择”问题？



```
Derived1 d1;  
Derived2 d2;  
Base* p[2]={&d1,&d2};  
p[0]->func();  
p[1]->func();
```


例



```
const double PI=3.14159265;
class Point
{
public:
    Point(double i,double j)
    {   x=i;y=j;   }
    double Area()
    {   return 0;   }
private:
    double x,y;
};
```

```
class Rectangle:public Point
{
public:
    Rectangle(double i,double j,
double k,double l):Point(i,j,k,l)
    {   w=k;h=l;   }
    double Area()
    {   return w*h;   }
private:
    double w,h;
};
```

```
class Circle:public Point
{
public:
    Circle(double i,double j,
double k):Point(i,j,k)
    {   r=k;   }
    double Area()
    {   return PI*r*r;   }
private:
    double r;
```

输出结果:

```
Area= 0
Area= 0
```

```
void fun(Point &p)
{cout<<"Area="<<p.Area()
<<endl;}

void main()
{ Rectangle r(3.5,4,5.2,6.6);
Circle c(4.5,6.2,5);
fun(r);
fun(c);
}
```

例9.7



```
const double PI=3.14159265;
class Point
{
public:
    Point(double i,double j)
    { x=i;y=j; }
    virtual double Area()
    { return 0; }
private:
    double x,y;
};
```

```
class Rectangle:public Point
{
public:
    Rectangle(double i,double j,
double k,double l):Point(i,j)
    { w=k;h=l; }
    double Area()
    { return w*h; }
private:
    double w,h;
};
```

```
class Circle:public Point
{
public:
    Circle(double i,double j,
double k):Point(i,j)
    { r=k; }
    double Area()
    { return PI*r*r; }
private:
    double r;
```

输出结果:

```
Area= 34.32
Area= 78.5398
```

```
void fun(Point &p)
{cout<<"Area="<<p.Area()
<<endl;}

void main()
{ Rectangle
r(3.5,4,5.2,6.6);
Circle c(4.5,6.2,5);
fun(r);
fun(c);
}
```

9.2.2 虚函数



1. 虚函数的说明方法

虚函数是实现运行时多态性的基础，是非静态成员函数。说明虚函数的格式如下：

virtual <类型> <成员函数名> (<参数表>)

2. 虚函数的作用

虚函数是实现动态联编的关键。

虚函数可以通过基类指针或引用访问基类和派生类中被说明为虚函数的同名函数。

3. 虚函数具有继承性

基类中声明了虚函数，派生类中无论是否说明，同原型(名字、参数、返回值均同)函数自动为虚函数

将p269页例10.8基类M的Print成员函数修改为虚函数，观察输出结果的变化。

输出结果：

8,3

3

9,5

15,9,5



例8.8 分析下列程序的输出结果。

```
#include<iostream.h>

class A
{public: A()      {a=0;}
      A(int i)  {a=i;}
      virtual void Print(){cout<<a<<endl;}
      int Geta()      {return a;}
private: int a;
};

class B:public A
{public: B()      {b=0;}
      B(int i,int j):A(i),b(j) { }
      void Print()
      {          cout<<b<<',';
        A::Print();
      }
private: int b;
};

void fun(A &a)
{      cout<<a.Geta()+2<<',';
      a.Print();
}
```

```
void main()
{
    A a1(10),a2;
    B b(10,20);
    b.Print();
    a2=b;
    a2.Print();
    A *pa=new A(15);
    B *pb=new B(15,25);
    pa=pb;
    pa->Print();
    fun(*pb);
    delete pa;
}
```

20, 10
10
25,15
17,25,15





多态性的概念



- ❖ **多态性**是同样的消息对应不同的响应
(相同的函数调用与不同执行代码)
- ❖ 多态的实现:
 1. 函数重载 (静态联编)
 2. 运算符重载 (静态联编)
 3. 虚函数 (动态联编)

静态联编 (绑定) 重载

联编工作出现在**编译阶段**, 用对象名或者类名来限定要调用的函数。

动态联编 (绑定) 虚函数

联编工作在**程序运行时执行**, 在程序运行时才确定将要调用的函数。通过基类的指针或引用访问派生类中的同名函数。



§ 9.1 运算符重载



THE C++ PROGRAMMING LANGUAGE

❖ 大多数运算符都可以重载，只有少数运算符不能重载。

❖ 不能重载的运算符：

- `.`（对象成员） 如：`obj.var;`
- `.*`（对象成员指针） 如：`obj.*ptr;`
- `::`（作用域） 如：`Class::age=18;`
- `?:`（三元条件） `int i = (a > b) ? a : b;`
- `sizeof`（字节数）如：

```
int size = sizeof(floatNum);
```

❖ 但有些运算符很少被重载（不建议被重载），如

- `->`（指针选成员）、`->*`（指针选指针成员）、
，`(逗号)`、`&`（取地址）、`()`（函数调用）





❖ 运算符重载遵循的“4个不变”原则

① 操作数个数不变；

// 即运算符的目数，除了()运算

② 优先级不变； // 被执行的优先程度

③ 结合性不变；

// 同等优先级的操作数的执行的顺序，从左到右还是从右到左。

④ 语义不变。（不是强制要求）

// 运算符应该保持运算符在语义上的一致性，如+号的重载函数，如果在语义上与加法无关，调用者将无法理解。





❖ 声明形式

函数类型 operator 运算符（形参）
{.....}

❖ 重载为类成员函数时

参数个数=原操作数个数-1 （后置++、--除外）

❖ 重载为友元函数时

参数个数=原操作数个数，且至少应该有一个自定义类型的形参。





```
#include<iostream. h>
class complex //复数类声明
{
private:
    double real; //复数实部
    double imag; //复数虚部
public:
    complex(double r=0.0,double i=0.0)
        {real=r;imag=i;} //构造函数
    complex operator +(complex c2); //+重载为成员函数
    friend complex operator +(complex &c1, complex
        &c2); //+重载为友员函数

    void display(); //输出复数
};
```

c3=c1.operator +(c2);

c3=c1+c2;

c3=operator +(c1,c2);

§ 9.1 运算符重载



THE C++ PROGRAMMING LANGUAGE

❖ 【例9.3】将赋值运算符重载为成员函数。

```
class A
{
public:
    A() { a1=a2=0; }
    A(int i,int j) { a1=i; a2=j; }
    A& operator =(A &p);
    int Geta1() { return a1; }
    int Geta2() { return a2; }
private:
    int a1,a2;
};
```

重载运算符=

```
A& A::operator =(A &p)
{
    a1 = p.a1;
    a2 = p.a2;
    return *this;
}
```

当前对象被修改，并把当前对象按引用返回。

```
void main()
{
    A a1(5,7),a2;
    a2=a1; // 调用operator =
    ...
}
```



9.6 重载流插入和流提取运算符

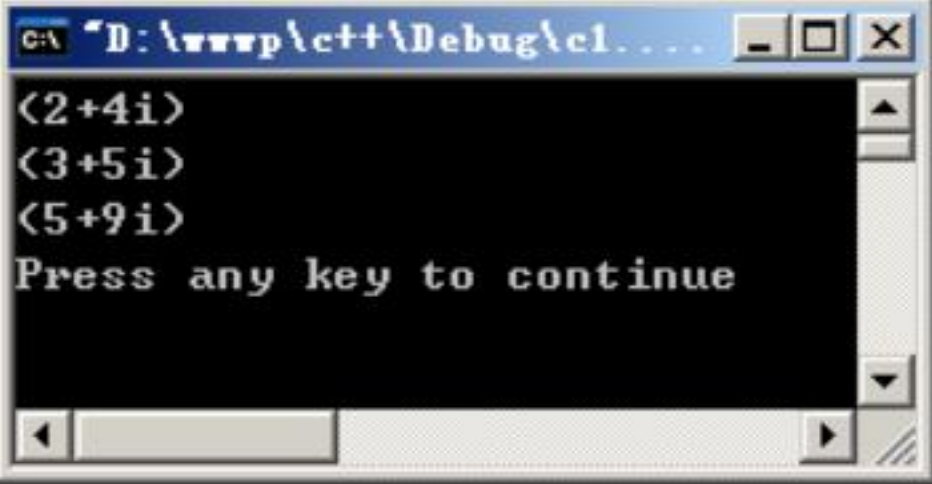


❖ 重载流插入运算符

我们希望“<<”运算符不仅能输出标准数据类型，而且能输出用户自己定义类对象。我们以复数对象输出为例。

```
#include <iostream.h>
class complex
{public:
    complex( ){real = 0; image = 0;}
    complex (double r, double i )
    { real = r; image = i; }
    complex operator + ( complex &c2)·
```

```
    fri
priv
    do
};
com
{ ret
ostr
{out
    ret
void main( )
{ complex c1(2,4), c2(3,5), c3;
  c3 = c2+c1;
  cout<<c1; cout<<c2; cout << c3;
}
```

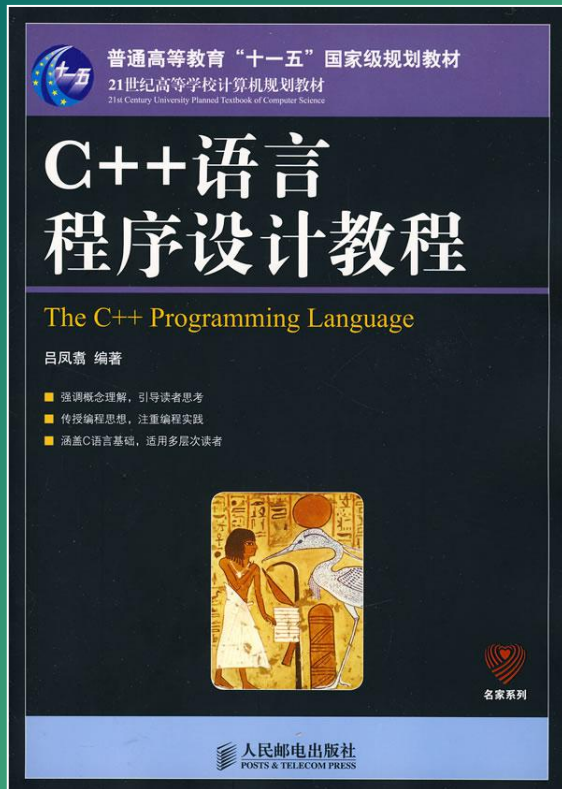


x&);

& c)



■ The C++ Programming Language



Thank You!

