C + + 语言程序设计

UNIT 13 课程内容复习

高级语言程序设计课程组

绪论

简单程序设计：标识符、基本数据类型、运算符和表达式、基本控制结构

函数：函数定义和使用、内联函数、默认参数、函数重载

类与对象：类与对象、构造函数和析构函数、类的组合、UML

数据的共享和保护：标识符的可见性和生存期、类的静态成员、友元、共享数据的保护

数组、指针和字符串

继承和派生

多态性

1. 绪论

面向过程的结构化程序设计方法

设计思路

- ✓ 自顶向下、逐步求精。采用模块分解与功能抽象，自顶向下、分而治之。

程序结构：

- ✓ 按功能划分为若干个基本模块，形成一个树状结构。
- ✓ 各模块间的关系尽可能简单，功能上相对独立；每一模块内部均是由顺序、选择和循环三种基本结构组成。
- ✓ 其模块化实现的具体方法是使用子程序（函数）。

面向对象的基本概

念对象

面向对象方法中的对象：

系统中用来描述客观事物的一个实体，它是用来构成系统的一个基本单位。对象由一组属性和一组行为（或方法）构成。

属性：用来描述对象静态特征的数据项。

方法：用来描述对象动态特征的操作序列。

类

类的定义：

类是具有相同属性和行为的一组对象的集合，它为属于它的全部对象提供了统一的抽象描述，其内部包括属性和行为两个主要部分。

类是对象集合的再抽象。

例如，台式计算机是类，小明的台式计算机是对象。

面向对象的基本特征（抽象性）

抽象性(Abstract)

抽象就是忽略事物中与当前目标无关的非本质特征，更充分地注意与当前目标有关的本质特征。

数据抽象和行为抽象

面向对象的基本特征（封装性）

封装性(Encapsulation)

- ✓ 封装就是把对象的属性和行为结合成一个独立的单位，并尽可能隐蔽对象的内部细节。
- ✓ 其有两个含义：一是封装性，另一个是“信息隐蔽”。其一，是把对象的全部属性和行为结合在一起，形成一个不可分割的独立单位。

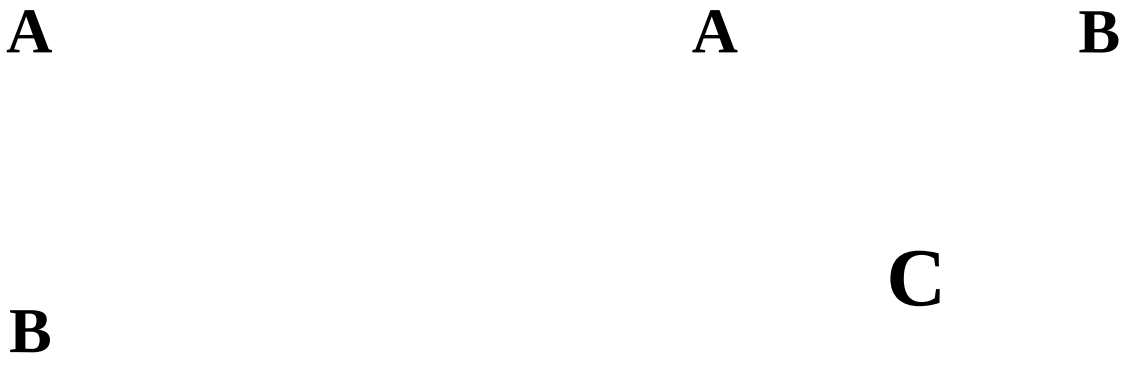
  其二，尽可能隐蔽对象的内部细节，对外形成一道屏障，与外部的联系只能通过外部接口实现。

好处：对象的使用者和设计者分开，提供代码的复用性，减轻开发软件系统的难度。

面向对象的基本特征（继承性）

继承性（Encapsulation）

- ✓ 继承是一种联结类与类的层次模型。继承性是指特殊类的对象拥有其一般类的属性和行为的特性。
- ✓ 继承意味着“自动地拥有”，即特殊类中不必重新定义已在一般类中定义过的属性和行为，而它却自动地、隐含地拥有其一般类的属性与行为
- ✓ 继承性的分类：单继承、多继承



面向对象的基本特征（多态性）

多态性（**Polymorphism**）是指类中同一函数名对应多个具有相似功能的不同函数，可以使用相同的调用方式来调用这些具有不同功能的同名函数的特性。

C++支持两种多态性：编译时的多态性和运行时的多态性。

- ✓ 重载：编译时多态性的实现，多个函数具有相同的名字但具有不同的作用。
  函数重载 操作符重载
- ✓ 虚函数：运行时多态性的实现，虚函数使用户在一个类等级中可以使用相同函数的多个版本。

2.1 简单程序设计：

标识符

字符集

C++语言中可用到的字符集有：

- **数字： 0 、 1 、……、 9 。**
- **字母：注意 C 程序中严格区分大小写字母，如 A 和 a 是不同的字符。**
- **空白符：空格符、制表符、换行符和换页符统称为空白符。它们主要用于分隔单词，一般无其它特殊意义。**
- **图形符号：图形（可见）符号，即 ！ “ # % &**

  ‘()*+,-./;:<=>?[\]^{|}

  ~ 主要用作各种运算符。

标识符

标识符：在 C++语言中要使用的对象，如符号常量、变量、函数、标号、数组、文件、数据类型和其他各种用户定义的对象，标识符就是这些对象的名字。

标识符的构成规则：

- 标识符由三类字符构成：英文大小写字母；数字 0……9；下划线。
- 必须由字母或下划线开头；后面可以跟随字母、数字或下划线。
- C++语言区分大小写，即大小写字母有不同的含义，例如：num，Num，NUM 为 3 个不同的标识符。
- 标识符不能与关键字同名，不能与库函数或自定义函数同名。

2.2 简单程序设计：

基本数据类型

基本数据类型

C++能够处理的基本数据类型

- ✓ 整数类型
- ✓ 浮点数类型
- ✓ 字符类型
- ✓ 布尔类型

程序中的数据

- ✓ 常量：在源程序中直接写明的数据，其值在整个程序运行期间不可改变，这样的数据称为常量。
- ✓ 变量：在程序运行过程中允许改变的数据，称为变量。

负数采用补码表示

基本数据类型

整型

有符号整型

短整型： **short int**
2 个字节长度，数据范围-32768～32767

整型： **int**
4 个字节

长整型： **long int**
4 个字节，数据范围-2 147 483 648~2 147 483 647

无符号整型

无符号短整型： **unsigned short**
int 2 个字节长度，数据范围 0～65535

无符号整型： **unsigned int**
4 个字节

无符号长整型： **unsigned long int**
4 个字节，数据范围 0~4 284 967 295

基本数据类型

字符型

有符号

char

1 个字节长度，数据范围-128～127

无符号

unsigned char

1 个字长，数据范围 0～255

基本数据类型

浮点型

浮点型： **float**

可以保存 **7** 位精度，**3.4E - 38～3.4E 38**

双精度浮点型： **double**

可以保存 **15** 位精度，**1.7E -308～1.7E308**

长双精度浮点型： **long double**

可以保存 **15** 位精度，**1.7E -308～1.7E308**

基本数据类型

布尔型

布尔型，又称逻辑型： **bool**

VC++等环境中占 1 个字节，不同编译系统中占据字节数可能不同

数据取值：false 或者 true ，分别表示假、真

数值类型的计算误差问题

计算误差问题：受字长限制，计算机无法表示无限位数的整数或实数；实数还存在表示精度的问题。一旦操作数过大、过小或者两个操作数相差过大，可能产生不准确或者完全错误的数据。

计算误差的类别：

- 上溢：运算结果过大，超过数据类型的可表示范围之外的一种错误状态
- 下溢：数据小于最小可表示数值时产生的一种错误状态；整数不会下溢；
- 可表示误差：计算机无法表示无限数位的实数

可计算误差小结：数据类型的选择

C++语言中那么多种数据类型，我该定义变量为哪种类型？

- 合适的数据范围：足够表示所有可能出现的数据取值；
- 满足精度要求；
- 尽量少占据内存空间：满足上述条件的前提下，尽可能少占据内存空间。

哪些基本数据类型是有序类型？

常量：数学中的常量和计算机中的常量

数学中的常量：取值不变的量。

计算机程序中的常量：

- 程序中取值不变的量；
- 常量的数据类型：根据字面形式可将常量区分为不同的数据类型。
  - ✓ 整型常量
  - ✓ 浮点型常量
  - ✓ 字符型常量
  - ✓ 字符串常量
  - ✓ 布尔常量

常量：整型常量（表示方法）

十进制整数：不带任何修饰的整数常量

123　-99

八进制整数：以 0 开头的整数常量

0123　067

68 ✘

十六进制整数：以 0X 或者 0x 开头的整数常量

0Xa123　0x67fe

常量后缀：L，长型；　U，无符号。

35000L　long int

10000U　unsigned int

常量：浮点型常量（表示方法）

小数表示法：

123.45　-99.0　0.234　.456

科学表示法：用字母 **e** 表示十进制指数中的 **10**，前面为尾数（小数形式或者整数），后面为阶码（整数）。

0.123E12　　0.123×10^{12}

67e-2　　67×10^{-2}

6.8E2.3 ✘

常量后缀：**F**，浮点型； **L**，长型。

3.5L　　long double

123.23F　　float

常量：字符型常量

字符常量：一对单引号中的单个字符。

'1'　　'+'　　'%'　　'a'

转义字符，又称反斜线字符常量：

多数可打印字符适用于直接放在一对单引号的方法。但是少数（回车键等）不能通过键盘放在字符常量中，为此，C++采用特殊的反斜线字符常量。

\b	退格	\t	水平制表	\v 垂直制表 \a 报警 \? 问号	
\f	换页	\”	双引号		
\n	换行	\’	单引号	\N	8 进制常量
\r	回车	\\	反斜线	\xN	16 进制常量

常量：字符串常量

字符串常量：一对双引号中的一系列字符。

"123.23 " " This is a test. "　　" "

字符串结束标记：编译程序在编译源程序时自动在每个字符串末尾放空字符'0',作为字符串结束标记。

"A "　　和　　**'A'**　　的区别

A　\0　　**A**

常量：布尔常量

布尔常量

false

true

变量：变量的定义形式

【限定词】类型 对象名称【=初始值】；

【……】：可选项

限定词：类型限定词或者存储类型限定词。

类型：有效的 C++数据类型。

对象名称：一个或多个用逗号间隔的标识符。

说明：变量必须先定义，后使用

变量：变量的初始化

C++定义了多种变量初始化的形式，例如：

```
int units_sold = 0 ;          //我们最熟悉的初始化形式
int units_sold = {0} ;        //列表初始化形式
int units_sold {0} ;          //第二种列表初始化形式
int units_sold (0);           //第四种初始化形式
```

符号常量

符号常量在声明时一定要赋初值，而在程序中间不能改变其值。

const 数据类型说明符常量名=常量值；

或：

数据类型说明符 const 常量名=常量值；

例： const float PI = 3.1415926；

2.3 简单程序设计：

运算符

运算符和表达式：算术运算符

运算符	作用
-	减法、负号
+	加法
*	乘法
/	除法
%	模除
--	减量
++	增量

/ 除法

○ 两个整数除法的结果是整数，整数除法的商。

5/2 结果 2

-5/2 结果 -2

% 模除，整数除法的余数。

○ 二元运算符，操作数均为整数

○ %运算的符号只取决于第一个运算数的符号。

++ -- 有前缀和后缀形式

运算符和表达式：
赋值运算符

对象名称=表达式

对象名称：变量或指针，赋值目标，要求有左值（可放在赋值运算符的左边→有用户访问的存储空间）。

复合赋值运算符

对象名称运算符=表达式

运算符和表达式：
逗号运算符

Exp1 , Exp2

Exp1 , Exp2, ……,Expn

1. 计算 **Exp1** 的值；
2. 计算 **Exp2** 的值；
3. 以此类推，以最后一项 **Expn** 的值作为表达式的结果。

例：1）a=3*5,a*4　　结果为 60 2）

(a=3*5,a*4),a+5　　结果为 20

运算符和表达式：关系和逻辑运算符

关系运算符		逻辑运算符	
运算符	作用	运算符	作用
<	小于	&&	与
>	大于	||	或
<=	小于等于	!	非
>=	大于等于		
==	等于		
!=	不等		

优先级别

!

> >= < <=

== !=

&&

||

运算符和表达式：
逻辑运算符（短路原则）

a&&b

当 **a** 为 **false** 时，可提前计算表达式结果为 **false**，因此不再处理 **b**。

例如，设变量 **int m,n,a,b** 的值均为 **0**，则执行表达式 **(m=a>b)&&(n=a>=b)**后，**m**、**n** 的值分别为（ **0** ）和（ **0**）。

a||b

当 **a** 为 **true** 时，可提前计算表达式结果为 **true**，因此不再处理 **b**。

例如，设变量 **int m,n,a,b** 的值均为 **0**，则执行表达式 **(m=a>=b)||(n=a>=b)**后，**m**、**n** 的值分别为（ **1**）和（**0**）。

运算符和表达式：
条件运算符

Exp1 ?Exp2:Exp3

1. 计算 **Exp1** 的值；
2. 如果 **Exp1** 的值为真，计算 **Exp2** 的值作为表达式的结果；
3. 如果 **Exp1** 的值为假，计算 **Exp3** 的值作为表达式的结果。

```
x=10;
y=(x>9)?100:200;
```

运算符和表达式：
编译时运算符，sizeof()

sizeof(操作数)

sizeof 变量名

- 操作数：变量、数据类型名。操作数为类型名必须用圆括号，操作数为变量可以不必用圆括号。
- 编译时一元运算符；
- 返回操作数对应的数据类型的字节数。

运算符和表达式：位运算符

运算符	作用
&	按位与
|	按位或
^	异或（没有进位的二进制加法运算）
~	求 1 的补
>>	右移
<<	左移

位运算运算规则

		&	|	^
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

运算符和表达式：
混合运算时数据类型的转换——隐含转换

一些二元运算符（算术运算符、关系运算符、逻辑运算符、位运算符和赋值运算符）要求两个操作数的类型一致。

在算术运算和关系运算中如果参与运算的操作数类型不一致，编译系统会自动对数据进行转换（即隐含转换），基本原则是将低类型数据转换为高类型数据。

char,short,int,unsigned,long,unsigned long,float,double

低　　高

运算符和表达式：*

混合运算时数据类型的转换——显式转换

语法形式：

1. 类型说明符(表达式)
2. (类型说明符)表达式

40

语句

C++中的语句有如下种类

- 空语句：只有一个语句结束符“;”
- 声明语句：例如，变量声明
- 表达式语句：在表达式末尾添加语句结束符构成表达式语句
  - 例如：a=3;
- 流程控制语句：选择语句、循环语句、跳转语句
- 标号语句：在语句前附加标号，通常用来与跳转语句配合
- 复合语句：用“{}”括起来的多条语句

2.4 简单程序设计：

基本控制结构

if 语句

if 语句用于在程序中有条件地执行某一语句序列，它有如下单分支和双分支两种基本语法格式。

单分支形式如下所示：

if (条件表达式) 单条语句或者复合语句 if

语句双分支形式如下所示：

```
if (条件表达式)
        单条语句 1 或者复合语句 1
else
        单条语句 2 或者复合语句 2
```

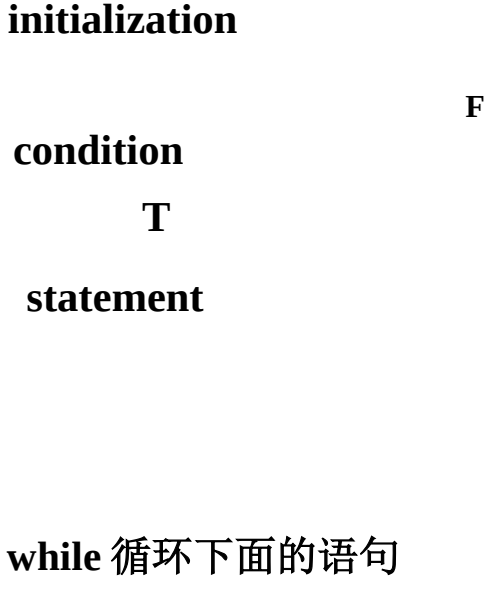
switch 语句

```
switch(expression){
    case constant1:
                statement sequence
                break;
    case constant2:
                statement sequence
                break;
    ……
    default:
                statement sequence
}
```

1. **break**：跳出 **case** 分支的跳转语句，必不可少。
3. **expression**：字符型、枚举类型或整型表达式；
4. **case constant**：**case** 后面只能为常量表达式；各个 **case** 常量必须各异。
5. 当表达式的值与 **case** 后面的常量表达式值相等时就执行此 **case** 后面的语句。
6. **case** 只能判断相等。遇第一个相等的 **case** 常量分支之后，顺序向下执行，不再进行相等与否的判断。

循环结构——while 语句

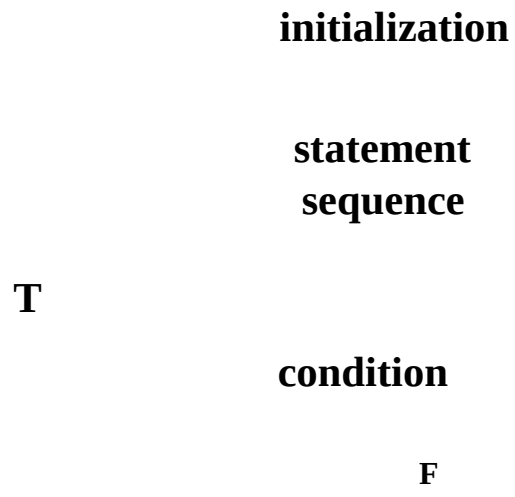
```
initialization;
while(condition)
        statement;
```



1. **initialization：** 初始化，一般为赋值语句；
2. **condition：** 循环条件，循环一直执行直到条件为假为止；
3. **statement：** 循环体，单个语句、块语句、空语句；

循环结构——do-while 语句

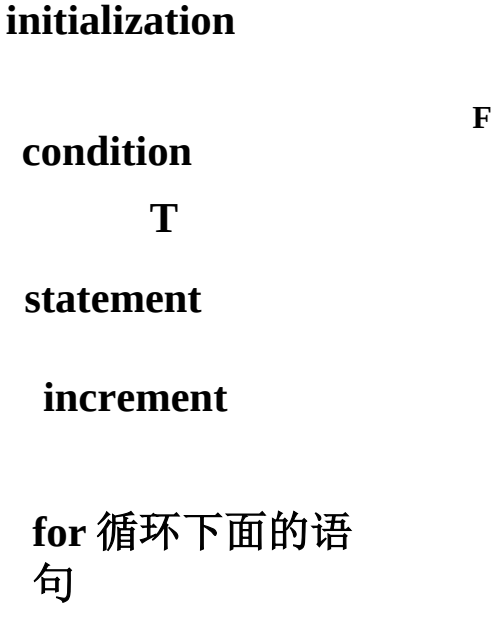
```
initialization;
do{
   statement
sequence }while(con
dition);
```



1. **initialization：初始化，一般为赋值语句；**
2. **condition：循环条件，循环一直执行直到条件为假为止；**
3. **statement sequence：循环体，语句序列；**

循环结构——传统的 for 语句

for(initialization; condition; increment)
statement;



1. **initialization**：初始化，一般为赋值语句；
2. **condition**：循环条件，循环一直执行直到条件为假为止；**condition** 缺省时表示 **true**。
3. **statement**：循环体，单个语句、块语句、空语句；
4. **increment**：修改控制变量。

3. 函数

函数定义

```
return-value-type function-name(parameter list)
{
    body of function
}
```

return-value-type: 函数返回值的数据类型；没有返回值写**void;**

function-name： 函数名；

parameter list： 形式参数表，用逗号间隔；

body of funcation： 函数体。

函数的调用：调用的一般形式

函数名(实参表)

说明：

- 实参与形参个数相等，类型一致，按顺序一一对应；
- 实参表求值顺序，因系统而定

函数的调用：函数原型

函数原型

return-value-type function-name(parameter list);

1. 把被调函数定义的位置放在主调函数之前，用这种方法也可以省去被调函数的原型说明
2. 被调函数定义的位置放在主调函数之后，则必须在函数调用之前使用被调函数的原型说明；

函数的调用：递归调用

函数直接或间接地调用自身，称为递归调用。

递归过程的两个阶段：

递推：

4!=4×3!→3!=3×2!→2!=2×1!→1!=1×0!→0!=1

未知　　已知

回归：

4!=4×3!=24←3!=3×2!=6←2!=2×1!=2←1!=1×0!=1←0!=1

未知　　已知

函数的参数传递：模块间的数据通信方式

主调函数与被调函数之间的三种数据通信方式：

两种基本通信方式：传入、传出；

双向传递：传入和传出的混合使用方式。

主调函数	主调函数	主调函数
传入方式	传出方式	双向传递
被调函数	被调函数	被调函数

函数的参数传递：参数传递方式

形参的类型决定了形参和实参交互的方式：值传递和引用传递

- 引用传递：当形参是引用类型时称为引用传递，此时形参绑定实参对象
- 值传递：当实参的值拷贝给形参时，形参和实参是相关独立的对象，称为值传递
  - ✓ 什么是传值：指初始化一个非引用类型变量，初始值被拷贝给变量。变量的改动不影响初始值
  - ✓ 传值参数的机理完全一样，函数对形参做的所有操作都不会影响实参

函数的参数传递：值传递

指针形参：当形参是指针类型时

- 指针的行为和其他非引用类型一样
- 执行指针拷贝操作时，拷贝的是指针的值
- 拷贝之后，两个指针是不同的指针
- 指针可间接访问它所指的对象，通过指针可修改它所指的对象的值

```
void reset(int *ip)
{  *ip = 0;// 改变指针 ip 所指向的对象的值
    ip = 0;              // 只改变了 ip 的局部拷贝，实参未被改变
}
int i = 42;
reset(&i);
cout << “i= ” << i << endl; // 输出 i=0
```

函数的参数传递：引用传递

引用(&)是标识符的别名，例如:

```
int i, j;
int &ri = i;
    //建立一个 int 型的引用 ri，并将其
    //初始化为变量 i 的一个别名
j = 10;
ri = j;//相当于 i = j;
```

声明一个引用时，必须同时对它进行初始化，使它指向一个已存在的对象。

一旦一个引用被初始化后，就不能改为指向其它对象。

引用可以作为形参

```
void swap(int &a, int &b) {...}
```

函数的参数传递：引用传递

实参和形参之间的传递二：按引用传递

形参变量/ 实参变量	特点	形参和实参共用同一个内存空间； 双向传递：对形参的取值的改变就是对实参取值的改变。
按引用传递		

内联函数

内联函数的函数原型是：

inline 返回值类型函数名（<参数列表>）

{函数体**}**

- 函数调用有一定的时间和空间开销，影响程序的执行效率。特别是对于一些函数体代码不是很大，但又频繁地被调用的函数，则引入内联函数。
- 在程序编译时，编译系统将程序中出现内联函数调用的地方用函数体进行替换。引入内联函数可以提高程序的运行效率，节省调用函数的时间开销，是一种以空间换时间的方案。

带默认参数值的函数

函数在声明时可以预先给出默认的形参值，调用时如给出实参，则采用实参值，否则采用预先给出的默认参数值。

有默认参数的形参必须在形参列表的最后，也就是说默认参数值的右面不能有无默认值的参数。因为调用时实参与形参的结合是从左向右的顺序。

如果一个函数有原型声明，且原型声明在定义之前，则默认参数值必须在函数原型声明中给出；而如果只有函数的定义，或函数定义在前，则默认参数值需在函数定义中给出。

函数重载

C++允许功能相近的函数在相同的作用域内以相同函数名声明，从而形成重载。方便使用，便于记忆。

- ✓ 重载函数的形参必须不同：个数不同或类型不同。不以形参名或者返回类型不同区分重载函数
- ✓ 编译程序将根据实参和形参的类型及个数的最佳匹配来选择调用哪一个函数。
- ✓ 不要将不同功能的函数声明为重载函数，以免出现调用结果的误解、混淆。这样不好：

4. 类与对象

类的定义

类定义一般分为：说明部分和实现部分。

说明部分是说明该类中的成员实现部分

是对成员函数的定义。

```
class 类名
{
    public:
        数据成员或成员函数的声明;
    private:
        数据成员或成员函数的声明; 说明部分
    protected:
        数据成员或成员函数的声明;
};
```

各个成员函数的定义　　实现部分

62

对象

类的对象是该类的某一特定实体，即类类型的变量。

类名 对象名表;

- 对象的定义格式与普通变量相同。
- 对象名表：可以有一个或多个对象名。当有多个对象名时，用逗号分隔。
- 对象名表：还可以是指向对象的指针名或引用名，也可以是对象数组名。

对象——类成员的访问

- 定义了类及其对象，就可以通过对象来使用其公有成员，从而达到对对象内部属性的访问和修改。
- 对象对其成员的访问有圆点访问形式

对象名.公有成员

构造函数

构造函数的功能是在定义对象时被编译系统自动调用来创建对象并初始化对象。

其定义格式如下：

```
类名::类名(参数表)
{
    函数体
}
```

构造函数——初始值列表

在构造函数中对类成员初始化的方式如下所示：

```
类名::类名(参数表)【:数据成员名 1(初始值 1)…】
{
    //构造函数体
}
```

注意：

①成员初始化顺序与它们在类定义中的出现顺序一致，而不是在初始值中出现的顺序！

② **const** 成员或引用成员必须将其初始化！

默认构造函数

调用时可以不需要参数的构造函数都是默认构造函数。

- ✓ 当不定义构造函数时，编译器自动产生默认构造函数
- ✓ 在类中可以自定义无参数的构造函数，也是默认构造函数
- ✓ 全部参数都有默认形参值的构造函数也是默认构造函数

下面两个都是默认构造函数，如果在类中同时出现，将产生编译错误：

```
Clock();
Clock(int newH=0,int newM=0,int newS=0);
```

复制构造函数

复制构造函数是一种特殊的构造函数，其形参为本类的对象引用。作用是用一个已存在的对象去初始化同类型的新对象。

```
class 类名 {
public :
    类名（形参）；//构造函数
    类名（const 类名 &对象名）；//复制构造函数
        ...
};
 类名::类（ const 类名 &对象名）//复制构造函数的实现
{函数体}
```

复制构造函数

复制构造函数被调用的三种情况

- ✓ 定义一个对象时，以本类另一个对象作为初始值，发生复制构造；
- ✓ 如果函数的形参是类的对象，调用函数时，将使用实参对象初始化形参对象，发生复制构造；
- ✓ 如果函数的返回值是类的对象，函数执行完成返回主调函数时，将使用 **return** 语句中的对象初始化一个临时无名对象，传递给主调函数，此时发生复制构造。

析构函数

- 析构函数的功能是在对象的生存期即将结束的时刻，由编译系统自动调用来完成一些清理工作。它的调用完成之后，对象也就消失了，相应的内存空间也被释放。
- 析构函数也是类的一个公有成员函数，它的名称是由类名前面加“~”构成，也不指定返回值类型。
- 和构造函数不同的是，析构函数不能有参数，因此不能重载。

```
类名::~类名()
{
    <函数体>
}
```

类的组合

类中的成员数据是另一个类的对象。

可以在已有抽象的基础上实现更复杂的抽象。

类组合的构造函数设计

原则：不仅要负责对本类中的基本类型成员数据赋初值，也要对对象成员初始化。

声明形式：

```
类名::类名(对象成员所需的形参，本类成员形参)
      :对象 1(参数)，对象 2(参数)，.....
{
//函数体其他语句
}
```

构造组合类对象时的初始化次序

首先对构造函数初始化列表中列出的成员（包括基本类型成员和对象成员）进行初始化，初始化次序是成员在类体中定义的次序。

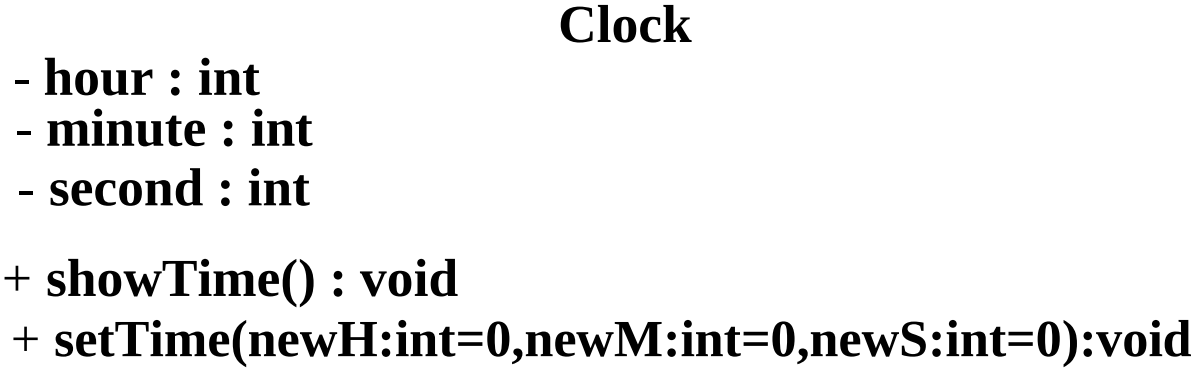
成员对象构造函数调用顺序：按对象成员的声明顺序，先声明者先构造。

初始化列表中未出现的成员对象，调用用默认构造函数（即无形参的）初始化

处理完初始化列表之后，再执行构造函数的函数体。

UML 类图

举例：Clock 类的完整表示



Clock 类的简洁表示

对象图

myClock : Clock

- hour : int
- minute : int
- second : int

myClock : Clock

几种关系的图形标识

依赖关系

类 A　　类 B

图中的“类 A”是源，“类 B”是目标，表示“类 A”使用了“类 B”，或称“类 A”依赖“类 B”

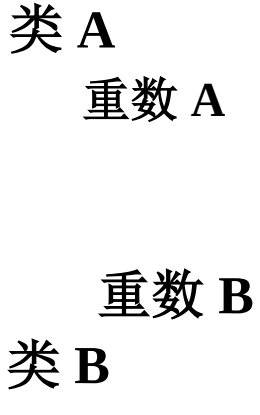
几种关系的图形标识

作用关系——关联

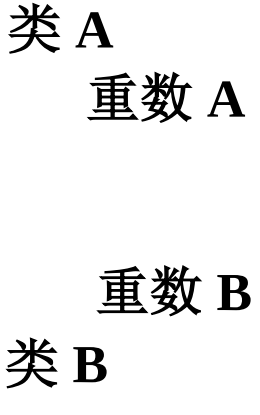
图中的“重数 A”决定了类 B 的每个对象与类 A 的多少个对象发生作用，同样“重数 B”决定了类 A 的每个对象与类 B 的多少个对象发生作用。

几种关系的图形标识

包含关系——聚集和组合



共享聚集



组成聚集（简称组合）

聚集表示类之间的关系是整体与部分的关系，“包含”、“组成”、“分为……部分”等都是聚集关系。共享聚集：部分可以参加多个整体；组成聚集（组合）：整体拥有各个部分，整体与部分共存，如果整体不存在了，那么部分也就不存在了。

5. 数据的共享和保护

作用域

作用域是一个标识符在程序正文中有效的区域，即程序中可以使用该标识符的区域。

函数原型作用域

局部作用域(块作用域)

类作用域

文件作用域

命名空间作用域

对象的生存期

四种存储类别：

- **auto**：自动类别，函数形参和不加 **static** 说明的局部变量，动态生存期。
- **register**：寄存器类别，用于说明自动存储期的变量，以达到目标代码优化的目的。
- **extern**：外部类别，说明需要使用在程序的其他地方具有外部链接的对象，被说明的对象必须是静态生存期的变量。
- **static**：静态生存期，可用于全局变量和局部变量。
  静态全局变量：设置为内部链接（本文件可访问）。
  静态局部变量：具有静态生存期。

全局变量：默认 **extern** 类别；

局部变量：默认 **auto** 类别。

类的静态成员

类的静态数据成员

- ✓ 用关键字 **static** 声明
- ✓ 不管产生多少个对象，类的静态数据成员拥有单一的存储空间，为该类的所有对象共享。静态数据成员具有静态生存期。
- ✓ 必须在类外定义和初始化，用**(::)**来指明所属的类。
- ✓ 静态数据成员可以是 **public**、**private**、**protected** 权限之一。

静态函数成员

- ✓ 类外代码可以使用类名和作用域操作符来调用静态成员函数。
- ✓ 静态成员函数只能引用属于该类的静态数据成员或静态成员函数。
- ✓ 静态成员函数属于整个类，不属于类中的某个对象，能在类的范围内共享

类的友元

通过将一个模块声明为另一个模块的友元，一个模块能够引用到另一个模块中本是被隐藏的信息。可以使用友元函数和友元类。

- 友元函数（独立函数或者类成员函数）是在类声明中由关键字 **friend** 修饰说明的非成员函数。

friend 返回值类型 函数名 (参数表);

- 友元类：若一个类为另一个类的友元，则此类的所有成员函数都能访问对方类的私有成员。友元关系不具有传递性

friend class 类名 B ;

共享数据的保护

对于既需要共享、又需要防止改变的数据应该声明为常类型（用 **const** 进行修饰）。对于不改变对象状态的成员函数应该声明为常函数。

- ✓ 常对象：必须进行初始化,不能被更新。

  const 类名对象名
- ✓ 常成员：用 **const** 进行修饰的类成员：常数据成员和常函数成员
- ✓ 常引用：被引用的对象不能被更新。

  const 类型说明符 **&**引用名
- ✓ 常数组：数组元素不能被更新。类型说明符 **const** 数组名[大小]...
- ✓ 常指针：指向常量的指针

编译预处理

#include 包含指令

- ✓ 将一个源文件嵌入到当前源文件中该点处。
- ✓ **#include<文件名>**

  按标准方式搜索，文件位于 C++系统目录的 include 子目录下
- ✓ **#include"文件名"**

  首先在当前目录中搜索，若没有，再按标准方式搜索。

#define 宏定义指令

- ✓ 定义符号常量，很多情况下已被 const 定义语句取代。
- ✓ 定义带参数宏，已被内联函数取代。

#undef

- ✓ 删除由#define 定义的宏，使之不再起作用。

6. 数组、指针和字符串

数组的声明与使用

数组的声明方法：

类型说明符　数组名[常量表达式] [常量表达式]……；

- ✓ 数组名是地址常量。
- ✓ 数组元素的访问：数组名[下标 1][下标 2]……
- ✓ 数组元素下标从 0，必须先声明，后使用。

  只能逐个引用数组元素，而不能一次引用整个数组
- ✓ 越界访问存储单元：C++编译系统不检查。
- ✓ 数组名作参数，形参被转换为指针，实参应是数组名。

对象数组

声明:

类名　数组名[元素个数];

访问方法:

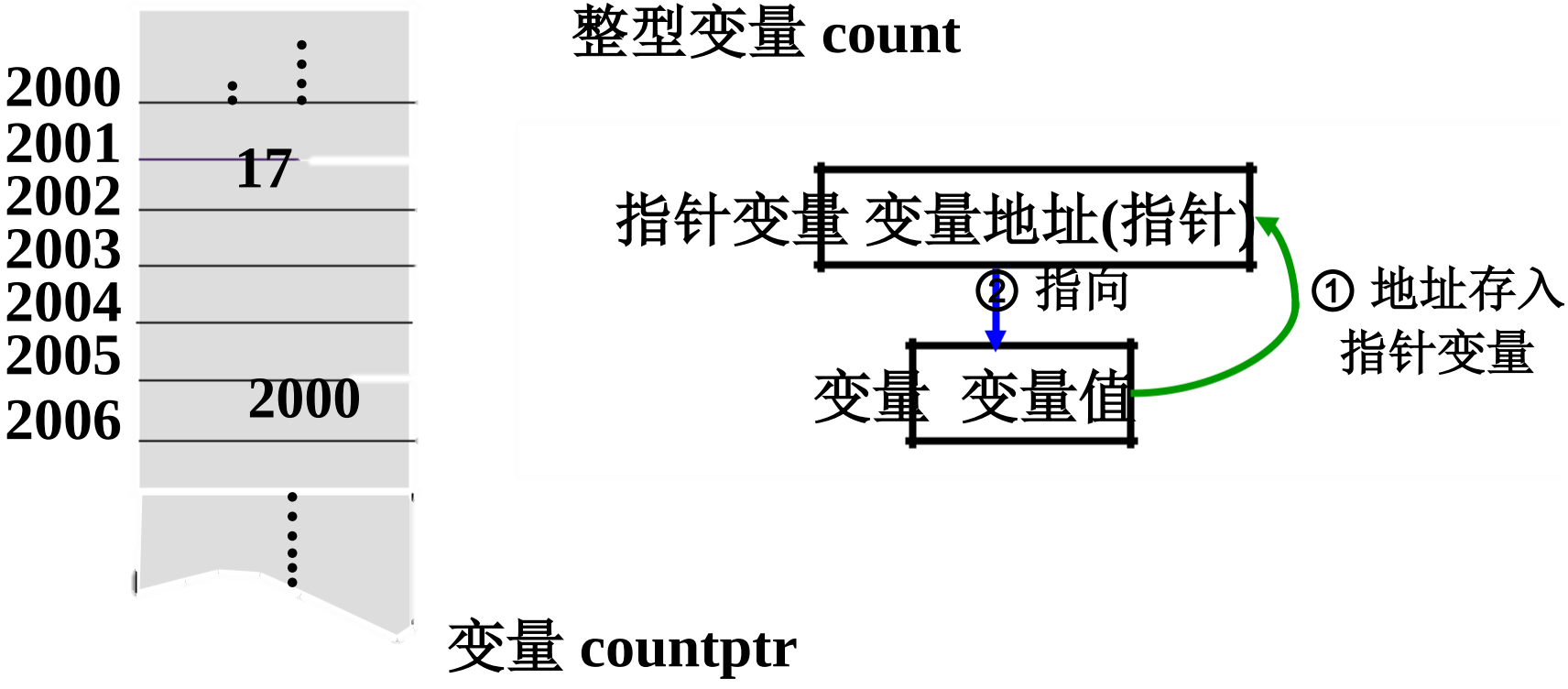
- ✓ 通过下标访问

数组名[下标].成员名

数组中每一个元素对象被创建时，系统都会调用类构造函数初始化该对象。通过初始化列表赋值。

指针和指针变量

- 指针：一个变量的地址；
- 指针变量：专门存放变量地址的变量。



指针变量初始化

type *name = initialization;

- 不要使用未初始化的指针变量：
  - 全局/静态指针变量未初始化，系统自动初始化为 NULL，此时指向对象不存在；
  - 自动指针变量未初始化，原来存储在内存空间中数据被保留；此时指针指向的对象没有意义。
- 不要用一个内部 **auto** 变量去初始化 **static** 指针。
- **initialization**：
  - **NULL**：不指向任何对象
  - 已经定义的地址或指针；
  - 动态分配内存。

与地址相关的运算——“*”和“&”

地址运算符：&

- ✓ 例：int var;

则 &var 表示变量 var 在内存中的起始地址

- ◆ 指针运算符：*
  - ✓ 例：*address

    表示内存地址为 address 的数据单元；

    其中，address 必须是指针（地址）表达式。

指针变量的赋值运算

指针名=地址

- ✓ “地址”中存放的数据类型与指针类型必须相符。
- ✓ 向指针变量赋的值必须是地址常量或变量，不能是普通整数。但可以赋值为整数 **0**，表示空指针。
- ✓ 指针的类型是它所指向变量的类型，而不是指针本身数据值的类型。
- ✓ 允许声明指向 **void** 类型的指针。该指针可以被赋予任何类型对象的地址。
- ✓ 例： **void *general;**

指向常量的指针

不能通过指针来改变所指对象的值，但指针本身可以改变，可以指向另外的对象。

例：int a;

```
const int *p1 = &a; //p1 是指向常量的指针
int b;
p1 = &b;        //正确，p1 本身的值可以改变
*p1 = 1;        //编译时出错，不能通过 p1 改变所指的对象
```

指针类型的常量

若声明指针常量，则指针本身的值不能被改变。

例：int a;

```
int * const p2 = &a;
p2 = &b;//错误，p2 是指针常量，值不能改变
```

指针运算：
指针变量的算术运算

指针与整数的加减运算

- ✓ 指针 **p** 加上或减去 **n**，其意义是指针当前指向位置的前方或后方第 **n** 个数据的地址。
- ✓ 这种运算的结果值取决于指针指向的数据类型。
- ✓ **p1[n1]**等价于***(p1 + n1)**

指针加 **1**，减 **1** 运算

- ✓ 指向下一个或前一个数据。
- ✓ 例如：**y=*px++** 相当于 **y=*(px++)**
  (*和**++**优先级相同，自右向左运算**)**

指针运算：
（两个指针的减法）

p1 – p2

- **p1、p2 必须是同类型的指针变量；**
- **结果为整数：p1 和 p2 指向的对象之间间隔的数据个数；**
- **p1、p2 通常指向同一个数组空间。**

指针运算：
指针变量的关系运算

关系运算

- ✓ 指向相同类型数据的指针之间可以进行各种关系运算。
- ✓ 指向不同数据类型的指针，以及指针与一般整数变量之间的关系运算是无意义的。
- ✓ 指针可以和零（NULL）之间进行等于或不等于的关系运算。例如：**p==NULL** 或 **p!=NULL**

用指针处理数组元素

声明与赋值

例：**int a[10], *pa;**

pa=&a[0]; 或 **pa=a;**

通过指针引用数组元素

- ✓ 经过上述声明及赋值后：

pa** 就是 **a[0]**，(pa+1)**就是 **a[1]**，**...** ，***(pa+i)**就是 **a[i] a[i], *(pa+i), *(a+i), pa[i]**都是等效的。

不能写 **a++**，因为 **a** 是数组首地址是常量。

用指针作为函数参数

以地址方式传递数据，可以用来返回函数处理结果。

实参是数组名时形参可以是指针或者数组名。

形参是数组名时，编译系统将其转换为对应的指针类型。

对象指针

声明形式

- 类名 *对象指针名;

  例: **Point a(5,10);**

  Piont *ptr;

  ptr=&a;

通过指针访问对象成员

- 对象指针名->成员名 **ptr->getx()**

  相当于 **(*ptr).getx();**

this 指针

隐含于每一个类的成员函数中的特殊指针。

明确地指出了成员函数当前所操作的数据所属的对象。

- ✓ 当通过一个对象调用成员函数时，系统先将该对象的地址赋给 **this** 指针，然后调用成员函数，成员函数对对象的数据成员进行操作时，就隐含使用了 **this** 指针。

动态申请内存操作符 new

声明：

new 类型名 **T**（初始化参数列表）

功能：在程序执行期间，申请用于存放 **T** 类型对象的内存空间，并依初值列表赋以初值。

结果值：

成功：**T** 类型的指针，指向新分配的内存；
失败：抛出异常。

释放内存操作符 delete

声明：

delete 指针 p

功能：释放指针 **p** 所指向的内存。**p** 必须是 **new** 操作的返回值。

申请和释放动态数组

分配：**new** 类型名 **T [** 数组长度 **]**

- ✓ 数组长度可以是任何表达式，在运行时计

算释放：**delete[]** 数组名 **p**

- ✓ 释放指针 **p** 所指向的数组。**p** 必须是用 **new** 分配得到的数组首地址。

深拷贝与浅拷贝

浅拷贝：实现对象间数据元素的一一对应复制。

深拷贝：当被复制的对象数据成员是指针类型时，不是复制该指针成员本身，而是将指针所指对象进行复制。

用字符数组存储和处理字符串

字符串常量（例："program"）

- ✓ 各字符连续、顺序存放，每个字符占一个字节，以‘\0’结尾，相当于一个隐含创建的字符常量数组
- ✓ “program”出现在表达式中，表示这一 char 数组的首地址
- ✓ 首地址可以赋给 char 常量指针：

```
const char *STRING1 = "program";
```

字符串变量

- ✓ 可以显式创建字符数组来表示字符串变量，例如：

```
char str[8] = "program";
```

- ✓ 字符指针模拟字符数组，表示字符串变量，例如：

```
char *str= "program";
```

string 类

常用构造函数

- **string(); //缺省构造函数，建立一个长度为 0 的串**
- **string(const char *s); //用指针 s 所指向的字符串常量初始化 string 类的对象**
- **string(const string& rhs); //拷贝构造函数**

例：

- **string s1; //建立一个空字符串**
- **string s2 = “abc”; //用常量建立一个初值为”abc”的字符串**
- **string s3 = s2;//执行拷贝构造函数，用 s2 的值作为 s3 的初值**

7. 继承和派生

类的继承与派生

继承与派生是同一过程从不同的角度来看

- ✓ 保持已有类的特性而构造新类的过程称为继承。
- ✓ 在已有类的基础上新增自己的特性而产生新类的过程称为派生。

被继承的已有类称为基类（或父类）。

派生出的新类称为派生类。

直接参与派生出某类的基类称为直接基类，基类的基类甚至更高层的基类称为间接基类。

派生类的声明(单继承)

class 派生类名**:[public/private/protected]** 基类名 **{**

派生类数据成员和函数成员定义

};

◆基类名： 已有的类的名称；

◆派生类名： 基础原有类的特性而生成的新类的名称；

◆派生方式： 默认为私有继承

public 公有继承
private 私有继承
protected 保护继承

◆派生方式的不同：

派生类中新增成员对基类成员的访问控制；
派生类外部，通过派生类对象对基类成员的访问控制。

◆派生类不能继承基类的构造函数和析构函数

派生类的声明（多继承）

- 多继承可以看作是单继承的扩展，派生类与每个基类之间的关系可以看作是一个单继承。在 C++中，多继承的定义格式如下：

class 派生类名:继承方式 1 基类名 1,…,继承方式 n 基类名 n {

　派生类新定义成员

};

访问控制

不同继承方式的影响主要体现在：

- ✓ 派生类成员对基类成员的访问权限
- ✓ 通过派生类对象对基类成员的访问权限三种继承方式
- ✓ 公有继承
- ✓ 私有继承
- ✓ 保护继承

类型转换规则

一个公有派生类的对象在使用上可以被当作基类的对象，反之则禁止。具体表现在：

- ✓ 派生类的对象可以隐含转换为基类对象。
- ✓ 派生类的对象可以初始化基类的引用。
- ✓ 派生类的指针可以隐含转换为基类的指针。通过基类对象名、指针只能使用从基类继承的成员

继承时的构造函数

基类的构造函数不被继承，派生类中需要声明自己的构造函数。

定义构造函数时，只需要对本类中新增成员进行初始化，对继承来的基类成员的初始化，自动调用基类构造函数完成。

派生类的构造函数需要给基类的构造函数传递参数

单一继承时的构造函数

派生类名::派生类名(基类所需的形参，本类成员所需的形参):基类名(参数表), 本类成员初始化列表

{//其他初始化；

};

多继承时的构造函数

派生类名::派生类名(参数表):基类名 1(基类 1 初始化参数表), 基类名 2(基类 2 初始化参数表), ...基类名 n(基类 n 初始化参数表), 本类成员初始化列表

{

//其他初始化;

};

多继承且有内嵌对象时的构造函数

派生类名::派生类名(形参表):基类名 1(参数), 基类名 2(参数), ...基类名 n(参数), 本类对象成员和基本类型成员初始化列表

{

//其他初始化

};

构造函数的执行顺序

① 调用基类构造函数，调用顺序按照它们被继承时声明的顺序（从左向右）。

② 对本类成员初始化列表中的基本类型成员和对象成员进行初始化，初始化顺序按照它们在类中声明的顺序。对象成员初始化是自动调用对象所属类的构造函数完成的。

③ 执行派生类的构造函数体中的内容。

析构函数

析构函数也不被继承，派生类自行声明

声明方法与一般（无继承关系时）类的析构函数相同。

不需要显式地调用基类的析构函数，系统会自动隐式调用。

析构函数的调用次序与构造函数相反。

作用域限定

当派生类与基类中有相同成员时：

若未特别限定，则通过派生类对象使用的是派生类中的同名成员。

如要通过派生类对象访问基类中被隐藏的同名成员，应使用基类名和作用域操作符（::）来限定。

二义性问题

当派生类从多个基类派生，而这些基类又从同一个基类派生，则在访问此共同基类中的成员时，将产生二义性——采用虚基类来解决。

在多继承时，基类与派生类之间，或基类之间出现同名成员时，将出现访问时的二义性（不确定性）——采用虚函数或同名隐藏来解决。

虚基类

虚基类的引入

- ✓ 用于有共同基类的场

合声明

- ✓ 以 **virtual** 修饰说明基类
  例：**class B1:virtual public B**

作用

- ✓ 主要用来解决多继承时可能发生的对同一基类继承多次而产生的二义性问题.
- ✓ 为最远的派生类提供唯一的基类成员，而不重复产生多次拷贝

注意：

- ✓ 在第一级继承时就要将共同基类设计为虚基类。

虚基类及其派生类构造函数

建立对象时所指定的类称为最（远）派生类。

虚基类的成员是由最派生类的构造函数通过调用虚基类的构造函数进行初始化的。

在整个继承结构中，直接或间接继承虚基类的所有派生类，都必须在构造函数的成员初始化表中给出对虚基类的构造函数的调用。如果未列出，则表示调用该虚基类的默认构造函数。

在建立对象时，只有最派生类的构造函数调用虚基类的构造函数，该派生类的其他基类对虚基类构造函数的调用被忽略。

8. 多态性

多态性

多态是指操作接口具有表现多种形态的能力，即能根据操作环境的不同采用不同的处理方式。多态性是面向对象系统的主要特性之一，在这样的系统中，一组具有相同基本语义的方法能在同一接口下为不同的对象服务。

C++语言支持的多态性可以按其实现的时机分为编译时多态（例如，运算符重载等）和运行时多态（例如虚函数）两类。

运算符重载的规则

C++ 几乎可以重载全部的运算符，而且只能够重载C++中已经有的。

- ✓ 不能重载的运算符举例：“.”、“.*”、“::”、“?:”

重载之后运算符的优先级和结合性都不会改变。

运算符重载是针对新类型数据的实际需要，对原有运算符进行适当的改造。

两种重载方式：重载为类的非静态成员函数和重载为非成员函数。

运算符重载的规则

成员函数重载运算符的一般形式

返回类型类名::operator 重载运算符(参数表)
{
    函数体
}

由关键字和要重载算符组成函数名

友元函数重载运算符的一般形式

friend 返回类型 operator 重载运算符(参数表)
{
    函数体
}

运算符重载的规则

不可重载运算符 .（成员引用运算符） *（成员指针运算符） :: ?:

sizeof

运算符重载需要注意的问题：

◆重载的运算符要保持原运算符的意义。例如，单目运算符不能重载为双目运算符；

◆只能对已有的运算符重载，不能增加新的运算符；

◆重载的运算符不会改变优先级别和结合性；

◆以下运算符只允许用成员函数重载：= ( ) [ ] new delete

运算符成员函数的设计

前置单目运算符 U

- 如果要重载 U 为类成员函数，使之能够实现表达式 U oprd，其中 oprd 为 A 类对象，则 U 应被重载为 A 类的成员函数，无形参。
- 经重载后，
  表达式 U oprd 相当于 oprd.operator U()

后置单目运算符 ++和--

- 如果要重载 ++或--为类成员函数，使之能够实现表达式 oprd++ 或 oprd-- ，其中 oprd 为 A 类对象，则
  ++或-- 应被重载为 A 类的成员函数，且具有一个 int 类型形参。
- 经重载后，表达式 oprd++ 相当于 oprd.operator ++(0)

重载赋值运算符（重载格式）

```
X & X :: operator = ( const X & from)
{
        // 复制 X 的成员
}
```

- 返回类类型的引用是为了与 C + + 原有赋值号的语义相匹配
- 用 const 修饰参数，保证函数体内不修改实际参数

重载( )和[ ]

只能用成员函数重载，不能用友元函数重载

重载函数调用运算符( )

函数调用的一般格式为　　基本表达式(表达式表)

可以看成一个二元运算符“（）”：

左操作数为 基本表达式

右操作数为 表达式表

对应的函数是 operator ( )

设 x 是类 X 的一个对象，则表达式

x (arg1 , arg2)

可被解释为

x . operator ( ) (arg1 , arg2)

重载下标运算符[ ]

下标运算的格式为 基本表达式 [< 表达式>] 可以看成一个二元运算符“[ ]”：

左操作数为基本表达式

右操作数为表达式

对应的函数是 **operator [ ]**

设 **x** 是类 **X** 的一个对象，则表达式

x [y]

可被解释为

x . operator [ ] (y)

重载时，只能显式地声明一个参数

虚函数

用 **virtual** 关键字说明的函数

虚函数是实现运行时多态性的基础

C++中的虚函数是动态绑定的函数

虚函数必须是非静态的成员函数，虚函数经过派生之后，就可以实现运行过程中的多态。

虚函数成员

C++中引入了虚函数的机制在派生类中可以对基类中的成员函数进行覆盖（重定义）。

虚函数的声明

virtual 函数类型函数名（形参表）

{

函数体

}

虚析构函数

为什么需要虚析构函数？

可能通过基类指针删除派生类对象；

如果你打算允许其他人通过基类指针调用对象的析构函数（通过 **delete** 这样做是正常的），就需要让基类的析构函数成为虚函数，否则执行 **delete** 的结果是不确定的。

C++中不能定义为虚函数的？

① 普通函数（非成员函数）：只能被 **overload**，不能被 **override**

② 构造函数

③ 内联成员函数：**inline** 函数在编译时被展开，虚函数在运行时
才能动态的绑定函数

④ 静态成员函数：对于每个类来说只有一份代码，所有的对象都共享这一份代码，所以没有要动态绑定的必要性

⑤ 友元函数：没有继承特性的函数，就没有虚函数之说

纯虚函数

纯虚函数是一个在基类中声明的虚函数，它在该基类中没有定义具体的操作内容，要求各派生类根据实际需要定义自己的版本，纯虚函数的声明格式为：

virtual 函数类型函数名(参数表) = 0;

带有纯虚函数的类称为抽象类:

```
class 类名
{
    virtual 类型函数名(参数表)=0; //纯虚函数
    ...
}
```

9. 程序设计能力

9.1 问题的理解

如何理解一个问题

借鉴表达

问题的抽象与提升

站在“读者”的角度

9.2 分析问题

这个问题是一个什么样？什么样的方法、思路怎么解决

逻辑数据结构、表示

什么样的思想，“科普式”的构思

最好是图示法表示

9.3 算法、软件部件描述

数据结构是什么？

用 **UML?**

类、对象、实例

之间的各种调用、联系(如对象运行)

出现开发环境的东东了