

程序设计 ppt 汇总

课程：2023 程序设计 D

任课老师：常军

助教：龙安迪

日期：2023/12

目录

一、绪论

二、C++语言简单程序设计

- (1) 标识符
- (2) 基本数据类型
- (3) 运算符和表达式
- (4) 常量与变量
- (5) 基本控制结构

三、函数

- (1) 函数定义和使用
- (2) 内联函数
- (3) 默认参数
- (4) 函数重载

四、类和对象

- (1) 类与对象
- (2) 构造函数和析构函数
- (3) 类的组合
- (4) UML

五、数据的共享和包含

- (1) 标识符的可见性和生存期
- (2) 类的静态成员
- (3) 友元
- (4) 共享数据的保护

六、数组、指针和字符串

七、类的继承

八、多态性

九、群体类和群体数据的组织

十、泛型程序设计与 C++语言标准模板库

十一、流类库与输入输出

十二、异常处理

一、绪论

1 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）。

2 面向对象的基本概念

类：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。

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。

3 面向对象的基本特征

（1）抽象性：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。

（2）封装性：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。

（3）继承性：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。

（4）多态性：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。

4 基本术语

源程序：用源语言编写的、有待翻译的程序

目标程序：是源程序通过翻译加工后生成的程序。目标程序可以用机器语言（此时称之为目标代码）、汇编语言、其他中间语言表示

翻译程序：用来把源程序翻译为目标程序的程序。有 3 种：汇编程序（负责把汇编语言写的源程序翻译成机器语言形式的目标程序）、编译程序（负责把高级语言写的源程序翻译成机器语言形式的目标程序）、解释程序（同样是把高级语言写的源程序翻译成机器指令。和编译程序的区别是边翻译边执行。）

二、简单程序设计

1 标识符

(1) **字符集**: C++语言中可用到的字符集有: 数字。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。

(2) **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 基本数据类型

字节数大小顺序满足: (signed/unsigned) signed char ≤ (unsigned) short int ≤ (unsigned) int ≤ (unsigned) long int ≤ long long int

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; 2 or 4 个字节

长整型: long int; 4 or 8 个字节, 4 字节的数据范围-2 147 483 648~2 147 483 647

长长整型: long long int; 至少有 8 个字节

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; 2 or 4 个字节

无符号长整型: unsigned long int; 4 or 8 个字节, 4 字节的数据范围 0~4 284 967 295

无符号长长整型: unsigned long long int; 至少有 8 个字节

字符型——char; 1 个字节长度, ISO C++标准没有规定是否有无符号!

有符号 signed char; 1 个 0 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

浮点型——浮点型: float 可以保存 7 位有效数字

双精度浮点型: double 可以保存 16 位有效数字

长双精度浮点型: long double 常用于特殊浮点需求的硬件, 精度各不相同

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

自定义数据类型:

typedef 声明——定义类型别名: typedef type name;

type: 原有数据类型名; name: 类型别名; 仅仅定义类型别名, 没有定义新的数据类型

auto 类型与 decltype 类型:

auto 类型说明符: 由编译器分析表达式类型。例如: auto val=val1+val2; //val 的类型由 val1+val2 决定; auto i=0, j=1; //正确: i、j 都是 int 类型; auto size=0, pi=3.14; //错误: size 和 pi

类型不一致

decltype:当我们定义一个变量与某一表达式类型相同,但是不想用该表达式初始化该变量时使用。

例如: `decltype (i) j=2; //j 初始值为 2, 类型和 i 一致`

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。

例如:正数 $32767+1=-32768$; 负数 $-128-1=127$

◆下溢:浮点值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。过小的数字需要更多数位的精度来表示它,因而无法存储在保存它的变量中。

◆可表示误差:计算机无法表示无限数位的实数。浮点数可表示误差示例:如果两个浮点数之间的差值非常小,小于问题的精度要求,就可认为二者相等。(可计算误差小结:数据类型的选择——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。)

3 计算机中的常量和变量

(1) 常量:程序中取值不变的量;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

字符型常量: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。

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

(2) 变量:程序中取值可变的量;内存空间中已分配、并且已命名的位置,数据类型决定系统分配给变量的内存空间的大小,以及该内存空间中数据的二进制编码规则。**!: 变量必须先定义,后使用。**

定义形式: (限定词) 类型 对象名称(=初始值); (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。)

定义位置:所有函数外:定义点之后的所有函数可以使用;全局变量或共有变量。函数内:{的后面,函数内定义的变量只有该函数可以使用,局部变量或者私有变量。函数参数定义:形式参数,属于局部变量。

(2.1) 变量的初始化:定义的同时初始化: `int i1=3, i2=4; float f1, f2=3.5;`

定义后,单独赋值: `int i1,i2; /* 定义整型变量 i1 和 i2 */ i1=3;i2=4; /*为 i1 赋初值为 3, i2 赋初值为 4 */` (初始化和赋值的区别?初始化是从无到有的过程,先分配空间,然后再填充数据;赋值是对已有的对象进行操作。)(默认初始化,是指定义变量没有指定初值时,则变量被默认初始化,即变量被赋予了“默认值”。基本数据类型的变量未被初始化,它的值由定义的位置决定。定义在函数之外的,默认初值为0,定义在函数之内的,则不被初始化!**未初始化的变量取值不确定,使用它可能引发运行时故障!**)

(2.2) 限定词:

const 类型限定词:符号常量。程序中不能被修改。`const int a=10;`类型常量a,其值为10。程序

不能修改 a 的值。

volatile 限定词：变量可能非明确说明的方式改变。一般情况下，编译程序认为没有出现在赋值语句左侧的变量不会改变。volatile 型的变量例外，因此编译系统会自动优化。例如，变量保存 实时时钟值。

（2.3）赋值及左值和右值

左值：赋值运算符左边的值；有地址的值；右值：赋值运算符右边的值；

① `x = 5;` ✓

② `5 = x;` ✗

C++中有些运算符仅仅需要操作数需要右值，例如“+”的两侧、赋值运算符的右边；有些运算符仅仅需要左值，例如赋值运算符“=”的左边；有些运算符需要操作数同时有右值与左值的角色，例如++或--。

（2.4）变量的存储类别

局部变量，默认：局部动态生存期，函数形参和不加 static 说明的局部变量。

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。（全局变量：默认 extern 类别；局部变量：默认，局部动态生存期。）

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。

thread_local：具有线程存储生存期，是 Thread 的局部变量，用于编多线程程序。只能用于修饰命名空间以及块作用域中的变量，或者已经被指定为 static 的变量。

mutable：只能用于类数据成员，并且不能与 const 或者 static 同时使用，不能用来修饰引用变量。

mutable 关键字是为了突破 const 关键字的限制，被 mutable 关键字修饰的成员变量永远处于可变的状况，即使是在被 const 修饰的成员函数中。

（2.5）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。例：`const float PI = 3.1415926;`

`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。

4 运算符

（1）优先级和结合性

优先级	运算符	结合性
1	()、[]、.、->、后置++、后置--	左
2	前置++、前置--、sizeof、&、*、+（正号）、-（负号）、~、!	右
3	（强制类型转换）	右
4	.*、->*	左
5	*、%、\、/	左
6	+、-	左
7	<<、>>	左
8	<、>、<=、>=	左
9	=、!=	左
10	&	左
11	^	左
12		左
13	&&	左
14		左
15	? :	右
16	=、*=、/=、%=、+=、-=、<<=、>>=、&=、^=、 =	右
17	,	左

（2）运算符

算数运算符——

++/--: i++或++i 单独作为一个语句，此时上下两个语句效果是相同的，都是自增 1。++或 - 操作在其他表达式中时，++或 - 前后位置影响取值，后置++（-）先取值再自增（减），前置++（-）先自增（减）再取值。

%: 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。

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）

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

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。）

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。）

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	优先级别 ! > >= < <= == != &&	
==	等于		
!=	不等		

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。

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。

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

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。

-显式转换

语法形式（3种）：1. 类型说明符(表达式)；2. (类型说明符)表达式；3. 类型转换操作符<类型说明符>(表达式)。类型转换操作符可以是：const_cast、dynamic_cast、reinterpret_cast、static_cast

显式类型转换的作用：将表达式的结果类型转换为类型说明符所指定的类型。例：int(z)，(int)z，static_cast<int>(z) 三种完全等价。

(3) 语句：

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

5 基本控制结构

(1) if 语句：用于在程序中有条件地执行某一语句序列，它有两种基本语法格式。

-单分支形式：if (条件表达式) 单条语句 或者 复合语句

-双分支形式：

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

(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。
2. expression: 字符型、枚举类型或整型表达式；
如 swiath(day).
3. case constant: case 后面只能为常量表达式；各个 case 常量必须各异。如 case 1。
4. 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。
5. 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。

(3) 循环结构——while 语句：

initialization;

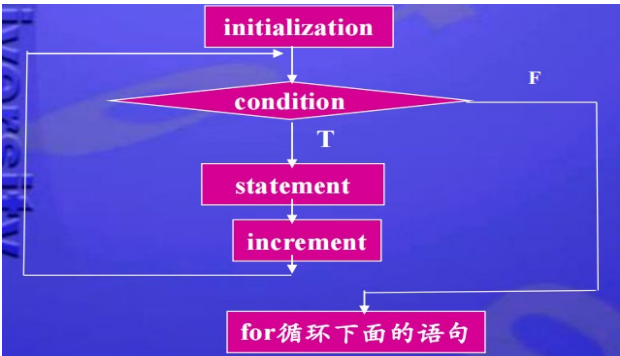
1. initialization: 初始化，一般为赋值语句；

<pre>while(condition) statement; （ture 则继续循环，false 则跳出循环，进入 while 循环下面的语句）</pre>	2. condition: 循环条件，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;
--	---

(3) 循环结构——do-while 语句:

<pre>initialization; do{ statement sequence }while(condition);</pre>	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;
--	---

(4) 循环结构——传统 for 语句:

<pre>for(initialization; condition; increment) statement;</pre> 	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。
--	---

(5) 循环结构——范围 for 语句:

C++11 新引入的更简单的 for 语句，用于遍历容器或其他序列的所有元素。其一般形式为：for (declaration : expression) 语句;其中，expression 必须是一个序列，例如花括号括起来的初始值列表、数组、vector、string 等类型的对象。declaration 定义一个变量，这个变量就是循环变量。最简单的方法就是用 auto 定义循环变量。范围 for 循环的执行过程为：每次迭代都重新定义循环变量；并将其初始化为序列中的下一个值；然后执行语句；直到 expression 中所有元素都处理完毕。

三、函数

1 函数定义和使用

(1) 定义:

<pre>return-value-type function-name(parameter list) { body of function }</pre>	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。
--	--

(2) 调用:

一般调用——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。

调用中函数原型的声明:

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；

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：—————→

$4! = 4 \times 3! \rightarrow 3! = 3 \times 2! \rightarrow 2! = 2 \times 1! \rightarrow 1! = 1 \times 0! \rightarrow 0! = 1$

未知

已知

回归：←—————

$4! = 4 \times 3! = 24 \leftarrow 3! = 3 \times 2! = 6 \leftarrow 2! = 2 \times 1! = 2 \leftarrow 1! = 1 \times 0! = 1 \leftarrow 0! = 1$

未知

已知

(3) 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。

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**。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。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) {...}`

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**。

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价值。

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。

2 内联函数

函数原型：`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。

3 默认参数

`constexpr` 函数，指能用于常量表达式的函数。该函数返回类型以及所有形参必须是常量，而且函数体必须有且仅有一条 `return` 语句。例如：`constexpr int get_size() {return 20;}`

`constexpr int ffoo=get_size();` //正确：foo 是一个常量表达式

`constexpr` 函数不一定返回常量表达式，例如：

`constexpr int len(int arg){return get_size()*arg;}` //如果 arg 是常量表达式是，则 `len(arg)` 也是常量表达式

用途：定义数组，例如 `int arr[len(2)];` //正确：`len(2)` 是常量表达式

`int i =2;` //i 不是常量表达式

`int a2[len(i)];` //错误：`len(i)` 不是常量表达式

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。

4 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。

四、类与对象

1 类与对象

类的定义：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;
    protected:
        数据成员或成员函数的声明;
};
各个成员函数的定义
```

说明部分

实现部分

对象：类的对象是该类的某一特定实体，即类类型的变量。

对象的定义格式与普通变量相同：类名 对象名表；

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。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。

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，如 `Student stu; stu.Age = 18;`

2 构造函数和析构函数

(1) 构造函数：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。

定义格式：类名::类名(参数表) {

函数体

}

构造函数中对类成员的初始化方式：

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！

②const 成员或引用成员必须将其初始化！

<p>默认构造函数</p> <p>调用时可以不需参数的构造函数都是默认构造函数。</p> <p>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。</p> <p>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：</p> <pre>Clock(); Clock(int newH=0,int newM=0,int newS=0);</pre> <p>委托构造函数</p> <p>委托构造函数使用它所属类的其他构造函数执行自己的初始化过程。例如：</p> <pre>Clock (int newH,int newN,int newS){ hour=newH; minute=newN; second=newS; } Clock():Clock(0,0,0){} //委托构造函数</pre>	<p>复制构造函数</p> <p>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。</p> <pre>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 }</pre> <p>被调用的三种情况：</p> <p>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。</p>
<p>移动构造函数——左值与右值</p> <p>左值是位于赋值语句左侧的对象变量，右值是位于赋值语句右侧的值，右值不依附于对象。</p> <pre>int i = 42; // i 是左值，42 是右值 int foolbar(); int j = foolbar(); //foolbar()是右值 左值和右值之间的转换 int i = 5, j = 6; // i 和 j 是左值 int k = i + j; // 自动转化为右值表达式</pre> <p>-右值引用</p> <p>对持久存在变量的引用称为左值引用，用&表示（即第 3 章引用类型）</p>	<p>基于右值引用，移动构造函数通过移动数据方式构造新对象，与复制构造函数类似，移动构造函数参数为该类对象的右值引用。示例如下</p> <pre>#include<utility> class astring { public : std::string s; astring (astring&& o) noexcept: s(std::move(o.s)) //显式移动所有成员 { 函数体 } }</pre> <p>移动构造函数不分配新内存，理论上不会报错，为</p>

对短暂存在可被移动的右值的引用称之为右值引用，用&&表示

```
float n = 6;
```

```
float &lr_n = n; //左值引用
```

```
float &&rr_n = n; //错误，右值引用不能绑定到左值
```

```
float &&rr_n = n * n; //右值表达式绑定到右值引用
```

通过标准库<utility>中的 move 函数可将左值对象移动为右值

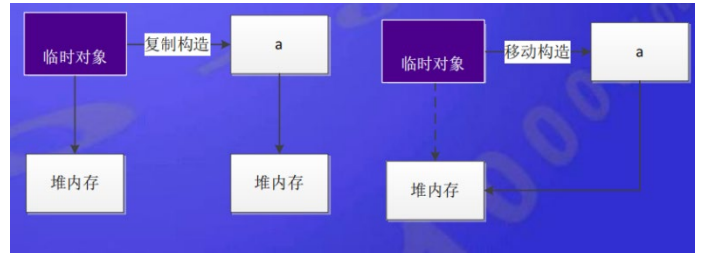
```
float n = 10;
```

```
float &&rr_n = std::move(n); //将 n 转为右值
```

使用 move 函数承诺除对 n 重新赋值或销毁外，不以 rr_n 以外方式使用

配合异常捕获机制，需声明 noexcept 表明不会抛出异常（将于 12 章异常处理介绍）被移动的对象不应再使用，需要销毁或重新赋值。

如果临时对象在被复制后就不再使用了，就完全可以把临时对象的资源直接移动，这样就避免了多余的复制操作。



什么时候该触发移动构造？

有可被利用的临时对象，如函数返回局部对象时，为避免不必要的复制构造，可以使用移动构造。

(2) 析构函数

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。

定义格式：类名::~~类名 () {
 <函数体>
}

(3) default、delete 函数

=default: 要求编译器生成默认或复制构造函数
例如：

```
class MyStr{  
public: string s;  
MyStr()=default; //默认合成的无参构造函数  
MyStr(string _s):s(std::move(_s)) {};//有参构造函数  
MyStr(MyStr &str)=default; //默认合成的复制构造函数  
~MyStr()=default;//默认合成的析构函数  
};
```

=delete: 要求编译器删除指定函数，除析构函数之外均可删除。例如：

```
class MyStr{  
public: string s;  
MyStr()=default; //默认合成的无参构造函数  
MyStr(string _s):s(std::move(_s)) {};//有参构造函数  
MyStr(MyStr &str)=delete; //删除复制构造函数  
~MyStr()=default;//默认合成的析构函数  
};
```

3 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

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

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 }	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。
---	---

4 UML

Clock 类图	对象图
Clock	myClock : Clock
-hour : int -minute : int -second : int	- hour : int - minute : int - second : int
+showTime():void +setTime(newH:int=0,newM:int=0,newS:int=0):void	

-依赖关系：（左图）

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”

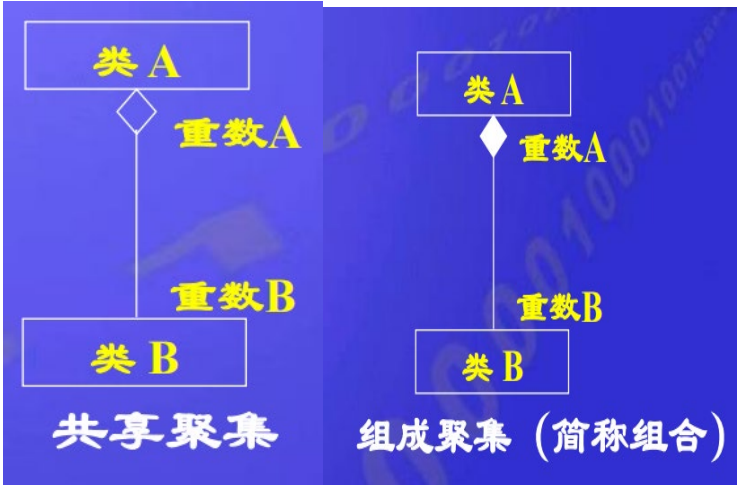


-作用关系：关联（右图）

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。

-包含关系：聚集和组合

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。



五、数据的共享和保护

1 标识符的可见性和生存期

作用域：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。有函数原型作用域、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

静态生存期

这种生存期与程序的运行期相同。在文件作用域中声明的对象（习惯称全局变量）具有这种生存期。在函数内部声明静态生存期对象，要冠以关键字 `static`。静态局部变量具有全局寿命，局部可见性。例如：`static int i;`静态生存期的变量未被初始化时，会被以 0 值初始化。

动态生存期

块作用域中声明的，没有用 `static` 修饰的对象是动态生存期的对象（习惯称局部生存期对象）。开始于程序执行到声明点时，结束于命名该标识符的作用域结束处。局部生存期变量不指定初值时，初值将不确定。

2 类的静态成员

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

3 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 ;*

4 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

5 C++程序的一般组织结构

一个源程序可以划分为多个源文件：类声明文件（.h 文件）、类实现文件（.cpp 文件）、类的使用

文件（main()所在的.cpp 文件），利用工程来组合各个文件。

外部变量 如果一个变量除了在定义它的源文件中可以使用外，还能被其它文件使用，那么就称这个变量是外部变量。 文件作用域中定义的变量，缺省情况下都是外部变量，但在其它文件中如果需要使用这一变量，需要用 extern 关键字加以声明。	外部函数 在所有类之外声明的函数（也就是非成员函数），都是具有文件作用域的。 这样的函数都可以在不同的编译单元中被调用，只要在调用之前进行引用性声明（即声明函数原型）即可。也可以在声明函数原型或定义函数时用 extern 修饰，其效果与不加修饰的缺省状态是一样的。
将变量和函数限制在编译单元内 使用匿名的命名空间：在匿名命名空间中定义的变量和函数，都不会暴露给其它的编译单元。 namespace { //匿名的命名空间 int n; void f() { n++; } } 这里被“namespace { }”括起的区域都属于匿名的命名空间。	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。

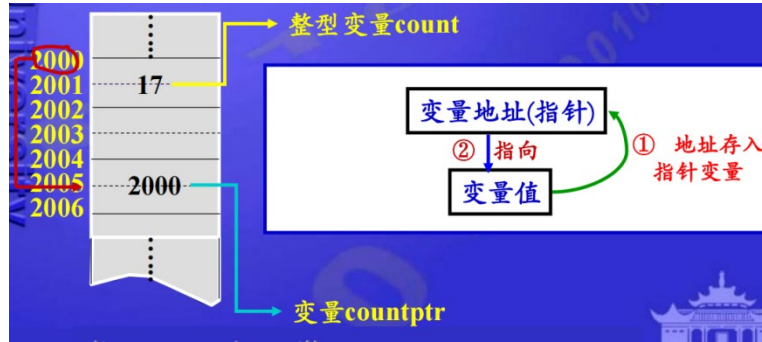
六、数组、指针和字符串

1 数组

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。 -常量表达式是不能放变量，可以是 1 等数字。 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。
---	---

2 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 <i>type *name = initialization;</i>
--	---



指针的典型应用：
动态数组；链表

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

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

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

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类型值的类型。

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;

指针变量的关系运算

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。

指针变量的算术运算

(1) 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)

(2) 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)

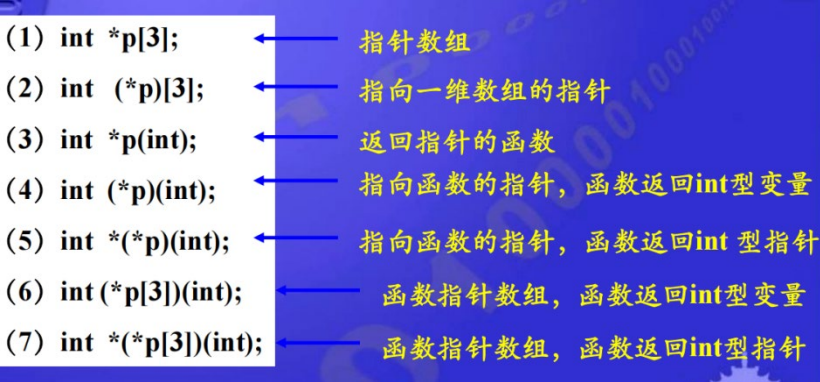
(3) 两个指针的减法 p1 - p2

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

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	隔的数据个数； p1、p2 通常指向同一个数组空间。
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);Point *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。	

深入理解多级指针

(1) 正确解读数据类型 基本准则：依据定义中出现的运算符优先级别和结合性来分析变量或数组的内存结构。	int (*p)(); p 为指向函数的指针变量，该函数返回整型数 int **p; p 为指针变量，它指向一个指向整型数据的指针变量
(2) 正确解读：指针的数据类型 <div> <div>定义</div> <div>含义</div> </div> int i; 定义整型变量 i int *p; p 为指向整型数据的指针变量 int a[n]; 定义含 n 个元素的整型数组 a int *p[n]; n 个指向整型数据的指针变量组成的指针数组 p int (*p)[n]; p 为指向含 n 个元素的一维整型数组的指针变量 int f();f 为返回整型数的函数	 <div> (1) int *p[3]; ← 指针数组 (2) int (*p)[3]; ← 指向一维数组的指针 (3) int *p(int); ← 返回指针的函数 (4) int (*p)(int); ← 指向函数的指针，函数返回int型变量 (5) int *(*p)(int); ← 指向函数的指针，函数返回int型指针 (6) int (*p[3])(int); ← 函数指针数组，函数返回int型变量 (7) int **p[3](int); ← 函数指针数组，函数返回int型指针 </div> int *p(); p 为返回指针的函数，该指针指向一个整型数据

3 动态内存分配

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。	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。
---	---

释放内存操作符 delete 声明: <i>delete 指针 p</i> 功能: 释放指针 p 所指向的内存。p 必须是 new 操作的返回值。	
--	--

4 用 vector 创建数组对象

被封装的动态数组 vector（这是一个类模板）。
 定义形式: vector<元素类型>数组对象名（数组长度）。如 x=10;vector<int>arr(x)。

5 深复制和浅复制

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。

6 字符串

(1) 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: <code>const char *STRING1 = "program";</code> 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: <code>char str[8] = "program";</code> -字符指针模拟字符数组, 表示字符串变量, 例如: <code>char *str= "program";</code>	(2) 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: <code>string s1; //建立一个空字符串</code> <code>string s2 = "abc"; //用常量建立一个初值为"abc"的字符串</code> <code>string s3 = s2; //执行拷贝构造函数, 用 s2 的值作为 s3 的初值</code>
--	--

七、继承

1 类的继承与派生

(1) 定义:
 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（或子类）。
 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。

(2) 派生类的声明

单继承	多继承
-----	-----

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	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 };
--	--

2 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

3 类型兼容规则

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。

4 派生类的构造和析构函数

(1) 构造函数

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。

(2) 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。

5 派生类成员的标识与访问

(1) 作用域分辨符

作用域限定 p270

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。如类名::成员名 //数据成员 ； 类名::成员名(参数表) //函数成员

二义性问题 p273

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。

(2) 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。

(3) 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。

八、多态性

1 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。

2 运算符重载

(1) 规则

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。

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

成员函数重载运算符的一般形式	<pre>返回类型 类名::operator重载运算符(参数表) { 函数体 }</pre> <div>由关键字和要重载运算符组成函数名</div>
友元函数重载运算符的一般形式	<pre>friend 返回类型 operator重载运算符(参数表) { 函数体 }</pre>
不可重载运算符	<pre>. (成员引用运算符) * (成员指针运算符) :: ?: sizeof</pre>

(2) 运算符成员函数的设计（运算符重载为成员函数）

<p>前置单目运算符 U</p> <p>-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。</p> <p>-经重载后，表达式 U oprd 相当于 oprd.operator U()</p>	<p>后置单目运算符 ++ 和 --</p> <p>-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。</p> <p>-经重载后，表达式 oprd++相当于 oprd.operator ++(0)</p>
<p>重载赋值运算符（重载格式）</p> <p>X & X :: operator = (const X & from)</p>	<p>重载函数调用运算符（ ）</p> <p>函数调用的一般格式为 基本表达式（表达式</p>

<pre>{ // 复制 X 的成员 }</pre> <p>返回类类型的引用是为了与 C++ 原有赋值号的语义相匹配</p> <p>用 const 修饰参数，保证函数体内不修改实参</p> <p>-重载()和[]</p> <p>只能用成员函数重载，不能用友元函数重载</p>	<p>表)</p> <p>可以看成是一个二元运算符“()”：</p> <p>左操作数 为 基本表达式</p> <p>右操作数 为 表达式表</p> <p>对应的函数是 <i>operator ()</i></p> <p>设 x 是类 X 的一个对象，则表达式 <i>x (arg1 , arg2)</i></p> <p>可被解释为 <i>x . operator () (arg1 , arg2)</i></p>
<p>重载下标运算符[]</p> <p>下标运算的格式为 基本表达式 [< 表达式 >]</p> <p>可以看成是一个二元运算符“[]”：</p> <p>左操作数 为 基本表达式</p> <p>右操作数 为 表达式</p> <p>对应的函数是 <i>operator []</i></p>	<p>(跟着左边) 设 x 是类 X 的一个对象，则表达式 <i>x [y]</i></p> <p>可被解释为</p> <p><i>x . operator [] (y)</i></p> <p>重载时，只能显式地声明一个参数</p>

3 虚函数

C++中的虚函数是动态绑定的函数，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。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。

<p>虚函数成员</p> <p>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）。</p> <p>虚函数的声明</p> <p>virtual 函数类型 函数名（形参表）</p> <pre>{ 函数体 }</pre>	<p>虚析构函数</p> <p>为什么需要虚析构函数？</p> <p>可能通过基类指针删除派生类对象；</p> <p>如果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。</p>
--	--

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

4 纯虚函数和抽象类

(1) 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
...
}
```

九、模板和群体数据

(1) 函数模板可以用来创建一个通用功能的函数，以支持多种不同形参，进一步简化重载函数的函数体设计。定义方法：

```
template <模板参数表>
函数定义
```

模板参数表的内容：类型参数：class（或 typename） 标识符；常量参数：类型说明符 标识符；
模板参数：template <参数表> class 标识符

(2) 类模板的声明

类模板：

```
template <模板参数表>
class 类名
{类成员声明}
```

如果需要在类模板以外定义其成员函数，则要采用以下的形式：

```
template <模板参数表>
类型名 类名<模板参数标识符列表>::函数名（参数表）
```

(3) 线性群体；线性群体的概念；直接访问群体—数组类；顺序访问群体—链表类；栈类；队列类

(4) 群体数据的组织：插入排序、选择排序、交换排序、顺序查找、折半查找。

排序：

排序是计算机程序设计中的一种重要操作，它的功能是将一个数据元素的任意序列，重新排列成一个按关键字有序的序列。

◆ 数据元素：数据的基本单位。在计算机中通常作为一个整体进行考虑。一个数据元素可由若干数据项组成。

◆ 关键字：数据元素中某个数据项的值，用它可以标识（识别）一个数据元素。

在排序过程中需要完成两种基本操作：

比较两个数的大小；调整元素在序列中的位置

内部排序：待排序的数据元素存放在计算机内存中进行的排序过程。包括插入排序、选择排序、交换排序。

外部排序：待排序的数据元素数量很大，以致内存存中一次不能容纳全部数据，在排序过程中尚需对外存进行访问的排序过程。

(1) 插入排序：每一步将一个待排序元素按其关键字值的大小插入到已排序序列的适当位置上，直到待排序元素插入完为止。

(2) 选择排序：每次从待排序序列中选择一个关键字最小的元素，（当需要按关键字升序排列时），顺序排在已排序序列的最后，直至全部排完。

(3) 交换排序：两两比较待排序序列中的元素，并交换不满足顺序要求的各对元素，直到全部满足顺序要求为止。（最简单的交换排序——起泡排序）

(4) 顺序查找：从序列的首元素开始，逐个元素与待查找的关键字进行比较，直到找到相等的。若整个序列中没有与待查找关键字相等的元素，就是查找不成功。

(5) 折半查找（二分法查找）：对于已按关键字排序的序列，经过一次比较，可将序列分割成两部分，然后只在有可能包含待查元素的一部分中继续查找，并根据试探结果继续分割，逐步缩小查找范围，直至找到或找不到为止。

(5) 深度探索

类模板和类；函数模板和函数；隐含实例化；多文件结构中模板的组织；显式实例化。

十、泛型程序设计与 C++ 语言标准模板库

1 泛型程序设计及 STL 的结构

泛型程序设计的**基本概念**：编写不依赖于具体数据类型的程序；将算法从特定的数据结构中抽象出来，成为通用的 C++ 的模板为泛型程序设计奠定了关键的基础。

几个术语：

-概念（concept）：用来界定具备一定功能的数据类型，如“支持‘<’运算符”的数据类型构成 Comparable 这一概念；

-模型（model）：符合一个概念的数据类型称为该概念的模型，如 int 型是 Comparable 概念的模型。

为概念赋予一个名称，并使用该名称作为模板参数名；例如，我们将用这样的方式表示 insertionSort 这样一个函数模板的原型：（事实上，很多 STL 的实现代码就是使用概念来命名模板参数的。）

```
template <class Sortable>
void insertionSort(Sortable a[], int n);
```

概念：用来界定具备一定功能的数据类型。例如：将“可以比大小的所有数据类型（有比较运算符）”这一概念记为 Comparable；将“具有公有的复制构造函数并可以用‘=’赋值的数据类型”这一概念记为 Assignable；将“可以比大小、具有公有的复制构造函数并可以用‘=’赋值的所有数据类型”这个概念记作 Sortable。

对于两个不同的概念 A 和 B，如果概念 A 所需求的所有功能也是概念 B 所需求的功能，那么就说概念 B 是概念 A 的子概念。例如：Sortable 既是 Comparable 的子概念，也是 Assignable 的子概念

模型：符合一个概念的数据类型称为该概念的模型，例如：int 型是 Comparable 概念的模型；静态数组类型不是 Assignable 概念的模型（无法用“=”给整个静态数组赋值）

STL 简介：标准模板库（Standard Template Library，简称 STL）提供了一些非常常用的数据结构和算法，为泛型程序设计提供了逻辑基础。STL 中的各个类模板、函数模板的参数都是用这个体系中的概念来规定的。使用 STL 的模板时，类型参数既可以是 C++ 标准库中已有的类型，也可以是自定义的类型——只要这些类型是所要求概念的模型。STL 是泛型程序设计的一个范例

STL 的基本组件：容器(container)、迭代器(iterator)、算法(algorithms)、函数对象(function

object)。

2 迭代器

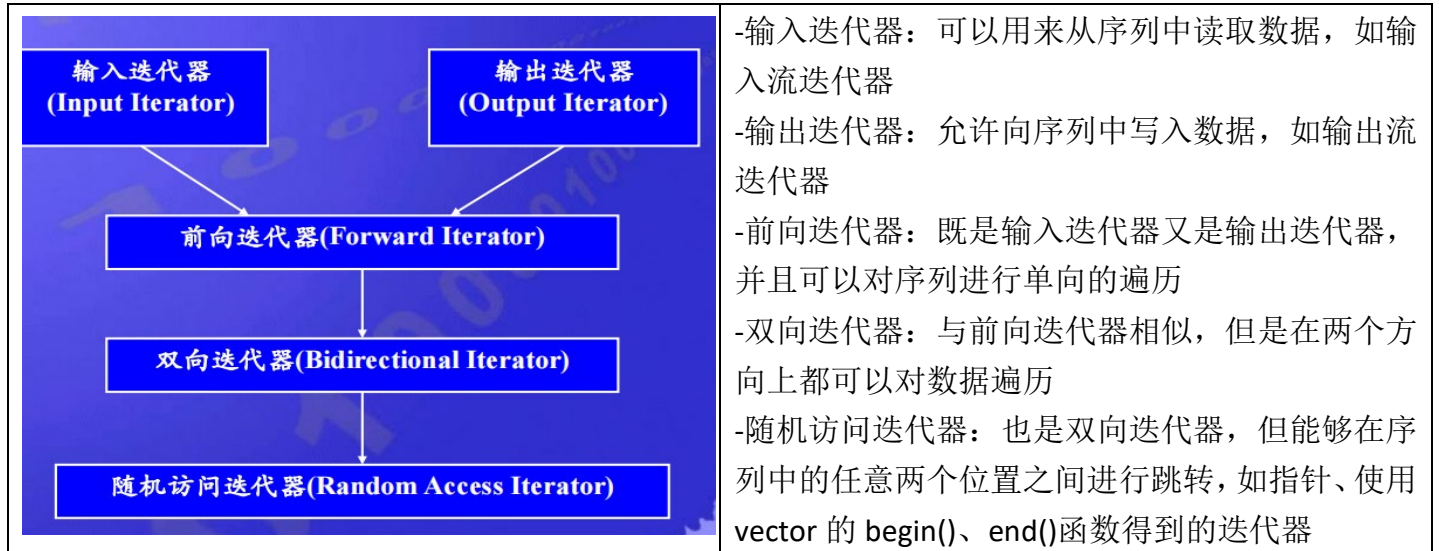
迭代器是算法和容器的桥梁

- 迭代器用作访问容器中的元素
- 算法不直接操作容器中的数据，而是通过迭代器间接操作

算法和容器独立

- 增加新的算法，无需影响容器的实现
- 增加新的容器，原有的算法也能适用

分类：迭代器是泛化的指针，提供了类似指针的操作（诸如++、*、->运算符）



迭代器的区间：两个迭代器表示一个区间： $[p1, p2)$ 。STL 算法常以迭代器的区间作为输入，传递输入数据，表示输入数据的序列。合法的区间：当且仅当 $p1$ 经过 n 次 ($n \geq 0$) 自增(++)操作后满足 $p1 == p2$ ；区间包含 $p1$ ，但不包含 $p2$ 。

迭代器的辅助函数：

`advance(p, n)` //对 p 执行 n 次自增操作

`distance(first, last)` //计算两个迭代器 $first$ 和 $last$ 的距离，即对 $first$ 执行多少次“++”操作后能够使得 $first == last$

3 容器的基本功能与分类

容器类是容纳、包含一组元素或元素集合的对象。（基于容器中元素的组织方式：顺序容器、关联容器。按照与容器所关联的迭代器类型划分：可逆容器-》随机访问容器。）

容器的通用功能：用默认构造函数构造空容器。支持关系运算符： $==$ 、 $!=$ 、 $<$ 、 $<=$ 、 $>$ 、 $>=$

`begin()`、`end()`：获得容器首、尾迭代器

`clear()`：将容器清空

`empty()`：判断容器是否为空

`size()`：得到容器元素个数

`s1.swap(s2)`：将 $s1$ 和 $s2$ 两容器内容交换

相关数据类型（ S 表示容器类型）

`S::iterator`：指向容器元素的迭代器类型

S::const_iterator: 常迭代器类型

4 顺序容器

顺序容器的特性：向量、双端队列、列表、单向链表、数组。

5 关联容器

6 函数对象

7 算法

8 深度探索

十一、流类库与输入输出

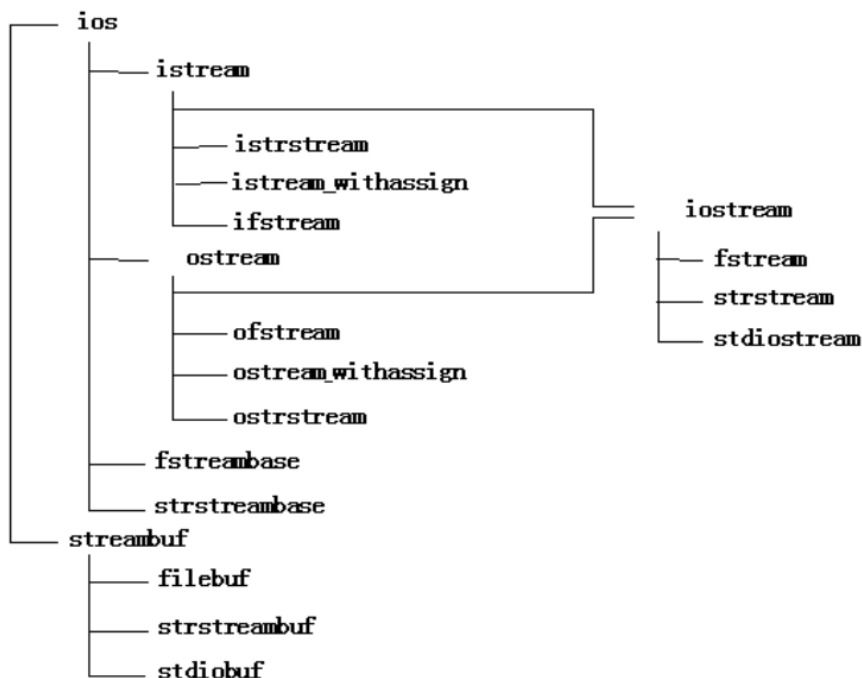
1 I/O 流的概念及流类库结构

当程序与外界环境进行信息交换时，存在着两个对象，一个是程序中的对象，另一个是文件对象。

流是一种抽象，它负责在数据的生产者和数据的消费者之间建立联系，并管理数据的流动。

程序建立一个流对象，并指定这个流对象与某个文件对象建立连接，程序操作流对象，流对象通过文件系统对所连接的文件对象产生作用。读操作在流数据抽象中被称为（从流中）提取，写操作被称为（向流中）插入。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。

C++编译系统提供的 I/O 流库含有两个平行基类：streambuf 和 ios，所有的流类都是由它们派生出来的。ios 类有 4 个直接派生类，即输入流类 istream、输出流类 ostream、文件流类 fstreambase、串流类 strstreambase，这 4 种流作为流类库中的**基本流类**。



2 输出流

最重要的三个输出流是

- ✓ ostream
- ✓ ofstream

✓ ostreamstream

3 输入流

重要的输入流类：

✓ istream 类最适合用于顺序文本模式输入。cin 是其实例。

✓ ifstream 类支持磁盘文件输入。

✓ istreamstream

4 输入/输出流

一个 iostream 对象可以是数据的源或目的。

两个重要的 I/O 流类都是从 iostream 派生的，它们是 fstream 和 stringstream。这些类继承了前面描述的 istream 和 ostream 类的功能

5 深度探索

宽字符、宽字符串与宽流；对象的串行化。

十二、异常处理

1. 异常处理的语法

抛出异常	捕获并处理异常
..... throw 表达式;	try 复合语句 catch (异常声明) 复合语句 catch (异常声明) 复合语句 ...

- 有异常则通过 throw 操作创建一个异常对象并抛出。
- 将可能抛出异常的程序段嵌在 try 块之中。控制流通过正常的顺序执行到达 try 语句，然后执行 try 块内的保护段。
- 如果在保护段执行期间没有引起异常，那么跟在 try 块后的 catch 子句就不执行。程序从 try 块后跟随的最后一个 catch 子句后面的语句继续执行下去。
- catch 子句按其在 try 块后出现的顺序被检查。匹配的 catch 子句将捕获并处理异常（或继续抛掷异常）。
- 如果匹配的处理器未找到，则运行库函数 terminate 将被自动调用，其缺省功能是调用 abort 终止程序。

2. 异常接口声明

- 可以在函数的声明中列出这个函数可能抛掷的所有异常类型。

例如：

```
void fun() throw(A, B, C, D);
```

- 若无异常接口声明，则此函数可以抛掷任何类型的异常。

- 不抛掷任何类型异常的函数声明如下：

```
void fun() throw();
```

3.标准程序库异常处理

异常类	头文件	异常的含义
bad_alloc	exception	用 new 动态分配空间失败
bad_cast	new	执行 dynamic_cast 失败（dynamic_cast 参见 8.7.2 节）
bad_typeid	typeid	对某个空指针 p 执行 typeid(*p)（typeid 参见 8.7.2 节）
bad_exception	typeid	当某个函数 fun()因在执行过程中抛出了异常声明所不允许的异常而调用 unexpected()函数时，若 unexpected()函数又一次抛出了 fun()的异常声明所不允许的异常，且 fun()的异常声明列表中有 bad_exception，则会有一个 bad_exception 异常在 fun()的调用点被抛出
ios_base::failure	ios	用来表示 C++的输入输出流执行过程中发生的错误
underflow_error	stdexcept	算术运算时向下溢出
overflow_error	stdexcept	算术运算时向上溢出
range_error	stdexcept	内部计算时发生作用域的错误
out_of_range	stdexcept	表示一个参数值不在允许的范围之内
length_error	stdexcept	尝试创建一个长度超过最大允许值的对象
invalid_argument	stdexcept	表示向函数传入无效参数
domain_error	stdexcept	执行一段程序所需要的先决条件不满足

- exception: 标准程序库异常类的公共基类
- logic_error 表示可以在程序中被预先检测到的异常
如果小心地编写程序，这类异常能够避免
- runtime_error 表示难以被预先检测到的异常