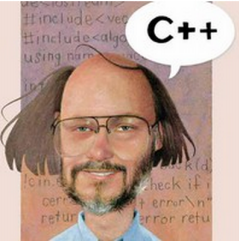
1. 综述 **C++** 1.1. 作者

1982 年,美国 AT&T 公司贝尔实验室的 Bjarne Stroustrup 博士在 c 语言的 基础上引 入并扩充了面向对象的概念,发明了一种新的程序语言。为了表达该 语言与 c 语言的渊 源关系,它被命名为 C++。而 Bjarne Stroustrup(本贾尼·斯特 劳斯特卢普)博士被尊称 为 C++语言之父。



C++语言之父



1.2. 历史背景

1.2.1. 应“运”而生?运为何?

C 语言作为结构化和模块化的语言,在处理较小规模的程序时,比较得心应 手。但 是当问题比较复杂,程序的规模较大的时,需要高度的抽象和建模时,c 语 言显得力不 从心。

为了解决软件危机,20 世纪 80 年代,计算机界提出了 OOP(object oriented programming)思想,这需要设计出支持面向对象的程序设计语言。 Smalltalk 就是当时 问世的一种面向对象的语言。而在实践中,人们发现 c 是语 此深入人心,使用如此之广 泛,以至于最好的办法,不是发明一种新的语言去取代 它,而是在原有的基础上发展它。 在这种情况下 c++应运而生,最初这门语言并 不叫 c++而是 c with class (带类的 c)。

1.2.2. C++发展大记事

|  |
| --- |
| 1983 年 8 月, 第一个 C++实现投入使用  1983 年 12 月,Rick Mascitti 建议命名为 CPlusPlus,即 C++。  1985 年 2 月, 第一个 C++ Release E 发布。  10 月,CFront 的第一个商业发布,CFront Release 1.0。  10 月,Bjarne 博士完成了经典巨著 The C++ Programming Language 第一版 1986 年 11 月,C++第一个商业移植 CFront 1.1,Glockenspiel。  1987 年 2 月, CFront Release 1.2 发布。  11 月,第一个 USENIX C++会议在新墨西哥州举行。  1988 年 10 月,第一次 USENIX C++实现者工作会议在科罗拉多州举行  1989 年 12 月,ANSI X3J16 在华盛顿组织会议。  1990 年 3 月, 第一次 ANSI X3J16 技术会议在新泽西州召开.  1990 年 5 月, C++的又一个传世经典 ARM 诞生。 1990 年 7 月, 模板被加入。  1990 年 11 月,异常被加入。  1991 年 6 月, The C++ Programming Language 第二版完成。  1991 年 6 月, 第一次 ISO WG21 会议在瑞典召开。  1991 年 10 月,CFront Release 3.0 发布。  1993 年 3 月, 运行时类型识别在俄勒冈州被加入。  1993 年 7 月, 名字空间在德国慕尼黑被加入。  1994 年 8 月, ANSI/ISO 委员会草案登记。  1997 年 7 月, The C++ Programming Language 第三版完成。 |

|  |
| --- |
| 1997 年 10 月,ISO 标准通过表决被接受  1998 年 11 月,ISO 标准被批准。 |

**1.3.** 应用领域

如果项目中,既要求效率又要建模和高度抽像,那就选择 c++吧。

**1.3.1.** 系统层软件开发

C++的语言本身的高效。

**1.3.2.** 服务器程序开发

面向对像,具有较强的抽像和建模能力。

**1.3.3.** 游戏,网络,分布式,云计算

效率与建模

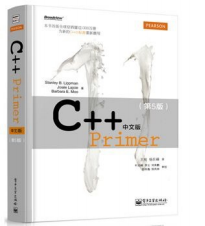
**1.3.4.** 科学计算

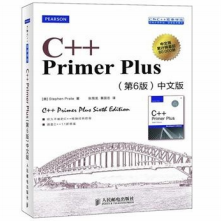
比如大名鼎鼎的 ACE 等科学类库。

1.4. 内容

C++语言的名字,如果看作 c 的基本语法,是由操作数 c 和运算符后++构 成。 C++ 是本身这门语言先是 c,是完全兼容 c.然后在此基础上++。这个++包含

三大部分,c++对 c 的基础语法的扩展,面向对像(继承,封装,多态),STL 等。

1.5. 书籍推荐



2. C++对C语言的加强

2.1 namespace命名空间

2.1.1 C++命名空间基本常识

所谓namespace，是指标识符的各种可见范围。C++标准程序库中的所 有标识符都被定义于一个名为std的namespace中。

一 ：<iostream>和<iostream.h>格式不一样，前者没有后缀，实际上， 在你的编译器include文件夹里面可以看到，二者是两个文件，打开文件就会 发现，里面的代码是不一样的。后缀为.h的头文件c++标准已经明确提出不支 持了，早些的实现将标准库功能定义在全局空间里，声明在带.h后缀的头文件 里，c++标准为了和C区别开，也为了正确使用命名空间，规定头文件不使用 后缀.h。 因此，

1)当使用<iostream.h>时，相当于在c中调用库函数，使用的是全局命 名空间，也就是早期的c++实现；

2) 当使用<iostream>的时候，该头文件没有定义全局命名空间，必须 使用namespace std；这样才能正确使用cout。

二： 由于namespace的概念，使用C++标准程序库的任何标识符时，可 以有三种选择：

1)直接指定标识符。例如std::ostream而不是ostream。完整语句如 下：

|  |
| --- |
| std::cout << std::hex << 3.4 << std::endl; |

2)使用using关键字。

|  |
| --- |
| using std::cout;  using std::endl;  using std::cin; |

以上程序可以写成 :

|  |
| --- |
| cout << std::hex << 3.4 << endl; |

3) 最方便的就是使用using namespace std; 例如： using namespace std;这样命名空间std内定义的所有标识符都有效(曝光)。就好像它们被声明 为全局变量一样。那么以上语句可以如下写: cout <<hex << 3.4 << endl;因 为标准库非常的庞大，所以程序员在选择的类的名称或函数名 时就很有可能和 标准库中的某个名字相同。所以为了避免这种情况所造成的名字冲突，就把标 准库中的一切都被放在名字空间std中。但这又会带来了一个新问 题。无数原 有的C++代码都依赖于使用了多年的伪标准库中的功能，他们都是在全局空间 下的。所以就有了<iostream.h> 和<iostream>等等这样的头文件，一个是 为了兼容以前的C++代码，一个是为了支持新的标准。命名空间std封装的是 标准程序库的名称，标准程序库为了和以前的头文件区别，一般不加".h"

2.1.2 C++命名空间定义以及使用方法

在C++中，名称( name)可以是符号常量、变量、宏、函数、结构、枚 举、类和对象等等。为了避免，在大规模程序的设计中，以及在程序员使用各 种各样的C++库时，这些标识符的命名发生冲突。

标准C++引入了关键字namespace (命名空间/名字空间/名称空间/名 域)，可以更好地控制标识符的作用域。

std是c++标准命名空间，c++标准程序库中的所有标识符都被定义在std 中，比如标准库中的类iostream、vector等都定义在该命名空间中，使用时要

加上using声明(using namespace std) 或using指示(如std::string、 std::vector<int>).

|  |
| --- |
| C中的命名空间  在C语言中只有一个全局作用域  C语言中所有的全局标识符共享同一个作用域  标识符之间可能发生冲突  C++中的命名空间  命名空间将全局作用域分成不同的部分  不同命名空间中的标识符可以同名而不会发生冲突  命名空间可以相互嵌套  全局作用域也叫默认命名空间 |

C++命名空间的定义：

|  |
| --- |
| namespace name { … } |

C++命名空间的使用：

使用整个命名空间：using namespace name;

使用命名空间中的变量：using name::variable;

使用默认命名空间中的变量：::variable

默认情况下可以直接使用默 认命名空间中的所有标识符

2.1.3 C++命名空间编程实践

|  |  |
| --- | --- |
| #include <stdio.h> | |
| namespace  {  int a  }  namespace | NameSpaceA  = 0;  NameSpaceB |

|  |  |
| --- | --- |
| {   |  | | --- | | #include <iostream> |   int a = 1;  namespace NameSpaceC  {  struct Teacher  {  char name[10];  int age;  };  }  }  int main(void)  {  using namespace NameSpaceA;  using NameSpaceB::NameSpaceC::Teacher;  printf("a = %d\n", a); //0  printf("a = %d\n", NameSpaceB::a);//1  NameSpaceB::NameSpaceC::Teacher t2;  Teacher t1 = {"aaa", 3};  printf("t1.name = %s\n", t1.name); //aaa  printf("t1.age = %d\n", t1.age); //3  return 0;  } |

2.1.4 结论

|  |
| --- |
| 1) 当使用<iostream>的时候，该[头文件](http://baike.baidu.com/view/668911.htm%22%20%5Ct%20%22_blank)没有定义全局命名空间，必须使用 namespace std；这样才能正确使用cout。若不引入using namespace std ,需要这 样做。std::cout。  2) c++标准为了和C区别开，也为了正确使用命名空间，规定[头文件](http://baike.baidu.com/view/668911.htm%22%20%5Ct%20%22_blank)不使用 后缀.h。  3) C++命名空间的定义： namespace name { … }  4) using namespace NameSpaceA;  5) namespce定义可嵌套。 |

2.2 “实用性”增强

|  |
| --- |
| using namespace std;  //C语⾔中的变量都必须在作⽤域开始的位置定义！！  //C++中更强调语⾔的“实⽤性”，所有的变量都可以在需要使⽤时再定义。  int main(void)  {  int i = 0;  cout << "i = " <<i <<endl;  int k;  k = 4;  cout << "k = " <<k <<endl;  return 0;  } |

2.3 变量检测增强

|  |
| --- |
| /\*  在C语⾔中，重复定义多个同名的全局变量是合法的  在C++中，不允许定义多个同名的全局变量  C语⾔中多个同名的全局变量最终会被链接到全局数据区的同⼀个地址空间上 int g\_var;  int g\_var = 1;  C++直接拒绝这种⼆义性的做法。  \*/  #include <iostream>  int g\_var;  int g\_var = 1;  int main(int argc, char \*argv[])  {  printf("g\_var = %d\n", g\_var);  return 0;  } |

2.4 struct 类型增强

|  |
| --- |
| /\*  C语⾔的struct定义了⼀组变量的集合， C编译器并不认为这是⼀种新的类型 C++中的struct是⼀个新类型的定义声明  \*/  #include <iostream>  struct Student  9 |

|  |
| --- |
| {  char name[100];  int age;  };  int main(int argc, char \*argv[])  {  Student s1 = {"wang", 1};  Student s2 = {"wang2", 2};  return 0;  } |

2.5 C++中所有变量和函数都必须有类型

|  |
| --- |
| /\*  C++中所有的变量和函数都必须有类型  C语⾔中的默认类型在C++中是不合法的  函数f的返回值是什么类型，参数⼜是什么类型？  函数g可以接受多少个参数？  \*/  //更换成 .cpp试试  f(i)  {  printf("i = %d\n", i);  }  g()  {  return 5;  }  int main(int argc, char \*argv[])  {  f(10);  printf("g() = %d\n", g(1, 2, 3, 4, 5));  getchar();  return 0;  } |

|  |
| --- |
| 在C语言中  int f( )；表示返回值为int，接受任意参数的函数  10 |

|  |
| --- |
| int f(void)；表示返回值为int的无参函数  在C++中  int f( );和int f(void)具有相同的意义，都表示返回值为int的无参函数  C++更加强调类型，任意的程序元素都必须显示指明类型 |

2.6 新增bool类型关键字

|  |
| --- |
| /\*  C++中的布尔类型  C++在C语⾔的基本类型系统之上增加了bool  C++中的bool可取的值只有true和false  理论上bool只占⽤⼀个字节，  如果多个bool变量定义在⼀起，可能会各占⼀个bit，这取决于编译器的实现  true代表真值，编译器内部⽤1来表⽰  false代表⾮真值，编译器内部⽤0来表⽰  bool类型只有true (⾮0)和false ( 0)两个值  C++编译器会在赋值时将⾮0值转换为true， 0值转换为false  \*/  #include <iostream>  using namespace std;  int main(int argc, char \*argv[])  {  int a;  bool b = true;  printf("b = %d, sizeof(b) = %d\n", b, sizeof(b));  b = 4;  a = b;  printf("a = %d, b = %d\n", a, b);  b = -4;  a = b;  printf("a = %d, b = %d\n", a, b);  a = 10;  b = a;  printf("a = %d, b = %d\n", a, b);  b = 0;  printf("b = %d\n", b);  11 |

|  |
| --- |
| return 0;  } |

2.7 三⽬运算符功能增强

|  |
| --- |
| #include <iostream>  using namespace std;  int main(void)  {  int a = 10;  int b = 20;  //返回⼀个最⼩数 并且给最⼩数赋值成30  //三⺫运算符是⼀个表达式 ，表达式不可能做左值  (a < b ? a : b ) = 30;  printf("a = %d, b = %d\n", a, b);  return 0;  } |

|  |
| --- |
| 1)C语言返回变量的值 C++语言是返回变量本身  C语言中的三目运算符返回的是变量值，不能作为左值使用  C++中的三目运算符可直接返回变量本身，因此可以出现在程序的任何地 方 2)注意：三目运算符可能返回的值中如果有一个是常量值，则不能作为左值  使用  (a < b ? 1 : b )= 30;  3)C语言如何支持类似C++的特性呢？  当左值的条件：要有内存空间；C++编译器帮助程序员取了一个地址而已 |

2.8 const增强

2.8.1 const基础知识

|  |
| --- |
| #include <iostream>  int main(void)  {  //const 定义常量 ---> const 意味只读  const int a;  int const b;  //第⼀个第⼆个意思⼀样 代表⼀个常整形数  const int \*c;  //第三个 c是⼀个指向常整形数的指针(所指向的内存数据不能被修改，但是本⾝可以修改)  int \* const d;  //第四个 d 常指针(指针变量不能被修改，但是它所指向内存空间可以被修改)  const int \* const e ;  //第五个 e⼀个指向常整形的常指针(指针和它所指向的内存空间，均不能被修改)  return 0;  } |

|  |
| --- |
| 合理的利用const的好处，  1指针做函数参数，可以有效的提高代码可读性，减少bug；  2清楚的分清参数的输入和输出特性 |

int setTeacher\_err( const Teacher \*p)

Const修改形参的时候，在利用形参不能修改指针所向的内存空间

2.8.2 C语言中的“冒牌货”

|  |
| --- |
| #include <stdio.h>  int main()  {  const int a = 10;  int \*p = (int\*)&a; |

|  |
| --- |
| printf("a===>%d\n", a);  \*p = 11;  printf("a===>%d\n", a);  return 0;  } |

2.8.3 const 和 #define 的相同

|  |
| --- |
| #include <iostream>  //#define N 10  int main()  {  const int a = 1;  const int b = 2;  int array[a + b] = {0};  int i = 0;  for(i = 0; i < (a+b); i++)  {  printf("array[%d] = %d\n", i, array[i]);  }  return 0;  } |

C++中的const修饰的，是一个真正的常量，而不是C中变量(只读)。在 const修饰的常量编译期间，就已经确定下来了

2.8.4 const 和 #define 的区别

|  |
| --- |
| #include <iostream>  void fun1()  {  #define a 10  const int b = 20;  }  void fun2()  {  printf("a = %d\n", a);  //printf("b = %d\n", b);  }  14 |

|  |  |
| --- | --- |
| int  {  } | main()  fun1();  fun2();  return 0; |

C++中的const常量类似于宏定义

const int c = 5; ≈ #define c [5](#_bookmark1)

C++中的const常量与宏定义不同

const常量是由编译器处理的，提供类型检查和作用域检查

宏定义由预处理器处理，单纯的文本替换

|  |
| --- |
| C语言中的const变量  C语言中const变量是只读变量，有自己的存储空间  C++中的const常量  可能分配存储空间,也可能不分配存储空间  当const常量为全局，并且需要在其它文件中使用，会分配存储空间  当使用&操作符，取const常量的地址时，会分配存储空间  当const int &a = 10; const修饰引用时，也会分配存储空间 |

2.9 真正的枚举

c 语言中枚举本质就是整型,枚举变量可以用任意整型赋值。而 c++中枚举 变量, 只能用被枚举出来的元素初始化。

|  |
| --- |
| #include <iostream>  using namespace std;  enum season {SPR,SUM,AUT,WIN};  int main()  {  enum season s = SPR;  //s = 0; // error, 但是C语言可以通过  s = SUM;  15 |

|  |
| --- |
| cout << "s = " << s <<endl; //1  return 0;  } |

3. C++对C语言的拓展

3.1 引用

3.1.1 变量名

变量名实质上是一段连续存储空间的别名，是一个标号(门牌号) 通过变量来申请并命名内存空间.

通过变量的名字可以使用存储空间.

|  |
| --- |
| 问题：对一段连续的内存空间只能取一个别名吗？ |

3.1.2 引用的概念

变量名，本身是一段内存的引用，即别名(alias). 引用可以看作一个已定 义变量的别名。

引用的语法：Type& name = var;

用法如下:

|  |
| --- |
| #include <iostream>  using namespace std;  int main(void)  {  int a = 10; //c编译器分配4个字节内存 , a内存空间的别名  int &b = a; //b就是a的别名  a = 11; //直接赋值  {  int \*p = &a;  \*p = 12;  16 |

|  |
| --- |
| cout << a <<endl;  }  b = 14;  cout << "a = " <<a << ", b = " << b <<endl;  return 0;  } |

3.1.3 规则

1 引用没有定义,是一种关系型声明。声明它和原有某一变量(实体)的关 系。故 而类型与原类型保持一致,且不分配内存。与被引用的变量有相同的地 址。

2 声明的时候必须初始化,一经声明,不可变更。

3 可对引用,再次引用。多次引用的结果,是某一变量具有多个别名。 4 &符号前有数据类型时,是引用。其它皆为取地址。

|  |
| --- |
| int main(void)  {  int a,b;  int &r = a;  int &r = b; //错误 ,不可更改原有的引⽤关系  float &rr = b; //错误 ,引⽤类型不匹配 cout<<&a<<&r<<endl; //变量与引⽤具有相**>**同 的地址。  int &ra = r; //可对引⽤更次引⽤ ,表⽰ a 变量有两个别名 ,分别是 r 和 ra  return 0;  } |

3.1.4 引用作为函数参数

普通引用在声明时必须用其它的变量进行初始化，引用作为函数参数声 明时不进行初始化。

|  |
| --- |
| #include <iostream>  using namespace std;  struct Teacher  {  char name[64];  int age ;  };  17 |

|  |
| --- |
| void printfT(Teacher \*pT)  {  cout<< pT->age <<endl;  }  //pT是t1的别名 ,相当于修改了t1  void printfT2(Teacher &pT)  {  pT.age = 33;  cout<<pT.age<<endl;  }  //pT和t1的是两个不同的变量  void printfT3(Teacher pT)  {  cout<<pT.age<<endl;  pT.age = 45; //只会修改pT变量 ,不会修改t1变量  }  int main(void)  {  Teacher t1;  t1.age = 35;  printfT(&t1);  printfT2(t1); //pT是t1的别名  printf("t1.age:%d \n", t1.age); //33  printfT3(t1) ;// pT是形参 ,t1 copy⼀份数据 给pT  printf("t1.age:%d \n", t1.age); //33  return 0;  } |

3.1.5 引用的意义

1)引用作为其它变量的别名而存在，因此在一些场合可以代替指针 2 )引用相对于指针来说具有更好的可读性和实用性

|  |
| --- |
| void swap(int a, int b); //⽆法实现两数据的交换  void swap(int \*p, int \*q); //开辟了两个指针空间实现交换 |

|  |
| --- |
| void swap(int &a, int &b){  int tmp;  tmp = a; a = b;  b = tmp; |

|  |
| --- |
| }  int main()  {  int a = 3,b = 5;  cout<<"a = "<<a<<"b = "<<b<<endl;  swap(a,b);  cout<<"a = "<<a<<"b = "<<b<<endl;  return 0;  } |

c++中引入引用后,可以用引用解决的问题。避免用指针来解决。

3.1.6 引用的本质

|  |
| --- |
| #include <iostream>  int main()  {  int a = 10;  int &b = a; // 注意： 单独定义的引⽤时，必须初始化。  b = 11;  printf("a:%d\n", a);  printf("b:%d\n", b);  printf("&a:%p\n", &a);  printf("&b:%p\n", &b);  return 0;  } |

思考1：c++编译器定※引用时，计后曲团什么工作。

|  |
| --- |
| #include <iostream>  struct Teacher {  int &a;  int &b;  };  int main()  {  printf("sizeof(Teacher) %d\n", sizeof(Teacher));  return 0;  } |

趮考3：!普滷引用有白鎈的空间吗?

1) 引用在C++中的内部实现是一个常指针

Type& name <===> Type\* const name

2)C++编译器在编译过程中使用常指针作为引用的内部实现，因此引用所占 用的空间大小与指针相同。

3)从使用的角度，引用会让人误会其只是一个别名，没有自己的存储空间。 这是C++为了实用性而做出的细节隐藏。

|  |
| --- |
| void func(int &a)  {  a = 5;  }  void func(int \*const a)  {  \*a = 5;  } |

|  |  |
| --- | --- |
| int  {  } | main()  int x = 10;  func(x);  return 0; |

|  |
| --- |
| 间接赋值的3各必要条件  1定义两个变量 (一个实参一个形参)  2建立关联 实参取地址传给形参  3\*p形参去间接的修改实参的值 |

引用在实现上，只不过是把：间接赋值成立的三个条件的后两步和二为一.

当实参传给形参引用的时候，只不过是c++编译器帮我们程序员手工取了

一个实参地址，传给了形参引用(常量指针) 。

3.1.7 引用作为函数的返回值(引用当左值)

|  |
| --- |
| I. 当函数返回值为引用时, 若返回栈变量:  不能成为其它引用的初始值(不能作为左值使用) |

|  |  |
| --- | --- |
| include <iostream>  using namespace std; | |
| int  {  } | getA1()  int a;  a = 10;  return a; |
| int& getA2()  {  int a;  a = 10;  return a;  } | |
| int main(void)  {  int a1 = 0;  int a2 = 0;  //值拷⻉  a1 = getA1();  //将⼀个引⽤赋给⼀个变量，会有拷⻉动作  //理解： 编译器类似做了如下隐藏操作， a2 = \*(getA2())  a2 = getA2();  //将⼀个引⽤赋给另⼀个引⽤作为初始值， 由于是栈的引⽤，内存⾮法  int &a3 = getA2();  cout <<"a1 = " <<a1<<endl;  cout <<"a2 = " <<a2<<endl;  cout <<"a3 = " <<a3<<endl;  return 0;  } | |

|  |
| --- |
| 若返回静态变量或全局变量  可以成为其他引用的初始值(可作为右值使用，也可作为左值使用 ) |

|  |  |
| --- | --- |
| #include <iostream>  using namespace std; | |
| int  {  } | getA1()  static int a;  a = 10;  return a; |
| int& getA2()  {  static int a;  a = 10;  return a;  } | |
| int main(void)  {  int a1 = 0;  int a2 = 0;  //值拷⻉  a1 = getA1();  //将⼀个引⽤赋给⼀个变量，会有拷⻉动作  //理解： 编译器类似做了如下隐藏操作， a2 = \*(getA2())  a2 = getA2();  //将⼀个引⽤赋给另⼀个引⽤作为初始值， 由于是静态区域，内存合法  int &a3 = getA2();  cout <<"a1 = " <<a1<<endl;  cout <<"a2 = " <<a2<<endl;  cout <<"a3 = " <<a3<<endl;  return 0;  } | |

|  |
| --- |
| 引用作为函数返回值，  如果返回值为引用可以当左值， |

|  |
| --- |
| 如果返回值为普通变量不可以当左值。 |

|  |
| --- |
| #include <iostream>  using namespace std;  //函数当左值  //返回变量的值  int func1()  {  static int a1 = 10;  return a1;  }  //返回变量本⾝ ,  int& func2()  {  static int a2 = 10;  return a2;  }  int main(void)  {  //函数当右值  int c1 = func1();  cout << "c1 = " << c1 <<endl;  int c2 = func2(); //函数返回值是⼀个引⽤ ,并且当右值  cout << "c2 = " << c2 <<endl;  //函数当左值  //func1() = 100; //error  func2() = 100; //函数返回值是⼀个引⽤ ,并且当左值  c2 = func2();  cout << "c2 = " << c2 <<endl;  return 0;  } |

3.1.8 指针引用

|  |
| --- |
| #include <iostream>  using namespace std;  struct Teacher  {  char name[64];  int age ;  };  23 |

|  |
| --- |
| //在被调⽤函数 获取资源  int getTeacher(Teacher \*\*p)  {  Teacher \*tmp = NULL;  if (p == NULL)  {  return -1;  }  tmp = (Teacher \*)malloc(sizeof(Teacher));  if (tmp == NULL)  {  return -2;  }  tmp->age = 33;  // p是实参的地址 \*实参的地址 去间接的修改实参的值  \*p = tmp;  return 0;  }  //指针的引⽤ 做函数参数  int getTeacher2(Teacher\* &myp)  {  //给myp赋值 相当于给main函数中的pT1赋值  myp = (Teacher \*)malloc(sizeof(Teacher));  if (myp == NULL)  {  return -1;  }  myp->age = 36;  return 0;  }  void FreeTeacher(Teacher \*pT1)  {  if (pT1 == NULL)  {  return ;  }  free(pT1);  }  int main(void)  {  Teacher \*pT1 = NULL;  //1 c语⾔中的⼆级指针  getTeacher(&pT1);  cout<<"age:"<<pT1->age<<endl;  FreeTeacher(pT1);  //2 c++中的引⽤ (指针的引⽤)  //引⽤的本质 间接赋值后2个条件 让c++编译器帮我们程序员做了。  24 |

|  |
| --- |
| getTeacher2(pT1);  cout<<"age:"<<pT1->age<<endl;  FreeTeacher(pT1);  return 0;  } |

3.1.9 const 引用

const 引用有较多使用。它可以防止对象的值被随意修改。因而具有一 些特性。

(1)const 对象的引用必须是 const 的,将普通引用绑定到 const 对象是不 合法的。 这个原因比较简单。既然对象是 const 的,表示不能被修改,引用当然 也不 能修改,必须使用 const 引用。实际上,

const int a=1;

int &b=a;

这种写法是不合法 的,编译不过。

(2)const 引用可使用相关类型的对象(常量,非同类型的变量或表达式)初 始化。 这个是 const 引用与普通引用最大的区别。

const int &a=2;

是合法的。

double x=3.14;

const int &b=a;

也是合法的。

|  |
| --- |
| #include <iostream>  //常引⽤是限制变量为只读 不能通过y去修改x了 /error  using namespace std;  int main(void)  {  //普通引⽤  int a = 10;  int &b = a;  cout << "b = " << b << endl;  //常引⽤  int x = 20;  const int &y = x;  //y = 21;  return 0;  25 |

|  |
| --- |
| } |

3.1.10 const引用的原理

const 引用的目的是,禁止通过修改引用值来改变被引用的对象。const 引用的 初始化特性较为微妙,可通过如下代码说明:

|  |
| --- |
| double val = 3.14;  const int &ref = val;  double & ref2 = val;  cout<<ref<<" "<<ref2<<endl;  val = 4.14;  cout<<ref<<" "<<ref2<<endl; |

上述输出结果为 3 3.14 和 3 4.14。因为 ref 是 const 的,在初始化的过 程中已经给定值,不允许修改。而被引用的对象是 val,是非 const 的,所以 val 的修改并未 影响 ref 的值,而 ref2 的值发生了相应的改变。

那么,为什么非 const 的引用不能使用相关类型初始化呢?实际上,const 引用 使用相关类型对象初始化时发生了如下过程:

|  |
| --- |
| int temp = val;  const int &ref = temp; |

如果 ref 不是 const 的,那么改变 ref 值,修改的是 temp,而不是 val。期 望对 ref 的赋值会修改 val 的程序员会发现 val 实际并未修改。

|  |
| --- |
| #include <iostream>  using namespace std;  int main(void)  {  //1> ⽤变量 初始化 常引⽤  int x1 = 30;  const int &y1 = x1; //⽤x1变量去初始化 常引⽤  //2> ⽤字⾯量 初始化 常量引⽤  const int a = 40; //c++编译器把a放在符号表中  //int &m = 41; //error , 普通引⽤ 引⽤⼀个字⾯量 请问字⾯量有没有内存地址 |

|  |
| --- |
| const int &m = 43; //c++编译器 会 分配内存空间  // int temp = 43  // const int &m = temp;  return 0;  } |

|  |
| --- |
| #include <iostream>  using namespace std;  struct Teacher  {  char name[64];  int age ;  };  void printTeacher(const Teacher &myt)  {  //常引⽤ 让 实参变量 拥有只读属性  //myt.age = 33;  printf("myt.age:%d \n", myt.age);  }  int main(void)  {  Teacher t1;  t1.age = 36;  printTeacher(t1);  return 0;  } |

|  |
| --- |
| 结论：  1)const int & e 相当于 const int \* const e  2)普通引用 相当于 int \*const e  3)当使用常量(字面量)对const引用进行初始化时，C++编译器会为常量值 分配空间，并将引用名作为这段空间的别名 4)使用字面量对const引用初始化后，将生成一个只读变量 |

3.2 inline内联函数

c 语言中有宏函数的概念。宏函数的特点是内嵌到调用代码中去,避免了 函数调用 的开销。但是由于宏函数的处理发生在预处理阶段,缺失了语法检测 和有可能带来的语 意差错。

3.2.1 内联函数基本概念

C++提供了 inline 关键字,实现了真正的内嵌。

|  |
| --- |
| #include <iostream>  using namespace std;  inline void func(int a)  {  a = 20;  cout << a <<endl;  }  int main(void)  {  func(10);  /\*  //编译器将内联函数的函数体直接展开  {  a = 20;  cout << a <<endl;  }  \*/  return 0;  } |

特点： 1)内联函数声明时inline关键字必须和函数定义结合在一起，否则编译器会直 接忽略内联请求。

2) C++编译器直接将函数体插入在函数调用的地方 。 3)内联函数没有普通函数调用时的额外开销(压栈，跳转，返回)。 4)内联函数是一种特殊的函数，具有普通函数的特征(参数检查，返回类型 等) 。

5) 内联函数由 编译器处理 ，直接将编译后的函数体插入调用的地方，

宏代码片段 由预处理器处理， 进行简单的文本替换，没有任何编译过程。

6) C++中内联编译的限制：

不能存在任何形式的循环语句

不能存在过多的条件判断语句

函数体不能过于庞大

不能对函数进行取址操作

函数内联声明必须在调用语句之前 7)编译器对于内联函数的限制并不是绝对的，内联函数相对于普通函数的优 势只是省去了函数调用时压栈，跳转和返回的开销。因此， 当函数体的执行开 销远大于压栈，跳转和返回所用的开销时，那么内联将无意义。

3.2.2 内联函数 vs 宏函数

|  |
| --- |
| #include <iostream>  #include <string.h>  using namespace std;  #if 0  优点 : 内嵌代码 ,辟免压栈与出栈的开销  缺点 : 代码替换 ,易使⽣成代码体积变⼤ ,易产⽣逻辑错误。  #endif  #define SQR(x) ((x)\*(x))  #if 0  优点 : ⾼度抽象 ,避免重复开发  缺点 : 压栈与出栈 ,带来开销  #endif  inline int sqr(int x)  {  return x\*x;  }  int main()  {  int i=0;  while(i<5)  {  // printf("%d\n",SQR(i++));  printf("%d\n",sqr(i++));  }  return 0;  } |

3.2.3 内联函数总结

|  |
| --- |
| 优点:避免调用时的额外开销(入栈与出栈操作)  代价:由于内联函数的函数体在代码段中会出现多个“副本”,因此会增加代码 段的空间。 |

|  |
| --- |
| 本质:以牺牲代码段空间为代价,提高程序的运行时间的效率。  适用场景: 函数体很“小”,且被“频繁”调用。 |

3.3 默认参数和占位参数

通常情况下,函数在调用时,形参从实参那里取得值。对于多次调用用一函 数同一 实参时,C++给出了更简单的处理办法。给形参以默认值,这样就不用从 实参那里取值了。

3.3.1 单个默认参数

|  |
| --- |
| //1 若 你填写参数,使⽤你填写的,不填写默认  void myPrint(int x = 3)  {  cout<<"x: “<<x<< endl;  } |

3.3.2 多个默认参数

|  |
| --- |
| //2 在默认参数规则 ，如果默认参数出现，那么右边的都必须有默认参数  float volume(float length, float weight = 4,float high = 5)  {  return length\*weight\*high;  }  int main()  {  float v = volume(10);  float v1 = volume(10,20);  float v2 = volume(10,20,30);  cout<<v<<endl;  cout<<v1<<endl;  cout<<v2<<endl;  return 0;  } |

3.3.3 默认参数规则

只有参数列表后面部分的参数才可以提供默认参数值

一旦在一个函数调用中开始使用默认参数值，那么这个参数后的所有参 数都必须使用默认参数值

3.3.4 占位参数

|  |
| --- |
| #include <iostream>  /\*  函数占位参数  占位参数只有参数类型声明，⽽没有参数名声明  ⼀般情况下，在函数体内部⽆法使⽤占位参数  \*/  int func(int a, int b, int)  {  return a + b;  }  int main()  {  func(1, 2); //error, 必须把最后⼀个占位参数补上。  //好悲剧的语法 -\_-!  printf("func(1, 2, 3) = %d\n", func(1, 2, 3));  return 0;  } |

|  |
| --- |
| #include <iostream>  /\*  可以将占位参数与默认参数结合起来使⽤  意义  为以后程序的扩展留下线索  兼容C语⾔程序中可能出现的不规范写法  \*/  //C++可以声明占位符参数，占位符参数⼀般⽤于程序扩展和对C代码的兼容  int func2(int a, int b, int = 0)  {  return a + b;  }  int main() |

|  |
| --- |
| {  //如果默认参数和占位参数在⼀起，都能调⽤起来  func2(1, 2);  func2(1, 2, 3);  return 0;  }  /\*  结论：如果默认参数和占位参数在⼀起，都能调⽤起来  \*/ |

3.4 函数重载

函数重载(Function Overload)：用同一个函数名定义不同的函数，当函 数名和不同的参数搭配时函数的含义不同。

3.4.1 重载规则

|  |
| --- |
| 1,函数名相同。  2,参数个数不同,参数的类型不同,参数顺序不同,均可构成重载。  3,返回值类型不同则不可以构成重载。 |

|  |
| --- |
| void func(int a); //ok  void func(char a); //ok  void func(char a,int b); //ok  void func(int a, char b); //ok  char func(int a); //与第⼀个函数有冲突 |

3.4.2 调用准则

|  |
| --- |
| 1,严格匹配,找到则调用。  2,通过隐式转换寻求一个匹配,找到则调用。 |

|  |
| --- |
| #include <iostream>  using namespace std; |

|  |
| --- |
| void print(double a){  cout<<a<<endl;  }  void print(int a){  cout<<a<<endl;  }  int main()  {  print(1); // print(int)  print(1.1); // print(double)  print('a'); // print(int)  print(1.11f); // print(double)  return 0;  } |

编译器调用重载函数的准则:

1.将所有同名函数作为候选者

2.尝试寻找可行的候选函数

3.精确匹配实参

4.通过默认参数能够匹配实参

5.通过默认类型转换匹配实参

6.匹配失败 7.最终寻找到的可行候选函数不唯一，则出现二义性，编译失败。 8.无法匹配所有候选者，函数未定义，编译失败。

3.4.3 重载底层实现(name mangling)

C++利用 name mangling(倾轧)技术,来改名函数名,区分参数不同的同 名函数。

实现原理:用 v c i f l d 表示 void char int float long double 及其引 用。

|  |
| --- |
| void func(char a); // func\_c(char a)  void func(char a, int b, double c); //func\_cid(char a, int b, double c) |

3.4.4 函数重载与函数默认参数

一个函数,不能既作重载,又作默认参数的函数。当你少写一个参数时,系统 无法确认是重载还是默认参数。

|  |
| --- |
| #include <iostream>  using namespace std;  int func(int a, int b, int c = 0)  {  return a \* b \* c;  }  int func(int a, int b)  {  return a + b;  }  int func(int a)  {  return a;  }  int main()  {  int c = 0;  c = func(1, 2); //error. 存在⼆义性，调⽤失败，编译不能通过  printf("c = %d\n", c);  return 0;  } |

3.4.5 函数重载和函数指针结合

|  |
| --- |
| /\*  函数重载与函数指针  当使⽤重载函数名对函数指针进⾏赋值时  根据重载规则挑选与函数指针参数列表⼀致的候选者  严格匹配候选者的函数类型与函数指针的函数类型  \*/  #include <iostream>  using namespace std;  int func(int x) // int(int a)  {  return x; |

|  |
| --- |
| }  int func(int a, int b)  {  return a + b;  }  int func(const char\* s)  {  return strlen(s);  }  typedef int(\*PFUNC)(int a); // int(\*)(int a)  typedef int(\*PFUNC2)(int a, int b); // int(\*)(int a, int b)  int main()  {  int c = 0;  PFUNC p = func;  c = p(1);  printf("c = %d\n", c);  PFUNC2 p2 = func;  c = p2(1, 2);  printf("c = %d\n", c);  return 0;  } |

函数指针基本语法

|  |
| --- |
| //⽅法⼀：  //声明⼀个函数类型  typedef void (myTypeFunc)(int a,int b);  //定义⼀个函数指针  myTypeFunc \*myfuncp = NULL; //定义⼀个函数指针 这个指针指向函数的⼊⼝地址  //⽅法⼆：  //声明⼀个函数指针类型  typedef void (\*myPTypeFunc)(int a,int b) ; //声明了⼀个指针的数据类型 //定义⼀个函数指针  myPTypeFunc fp = NULL; //通过 函数指针类型 定义了 ⼀个函数指针 ,  //⽅法三：  //定义⼀个函数指针 变量  void (\*myVarPFunc)(int a, int b);  35 |

3.4.6 函数重载总结

|  |
| --- |
| 重载函数在本质上是相互独立的不同函数。  函数的函数类型是不同的  函数返回值不能作为函数重载的依据  函数重载是由函数名和参数列表决定的。 |

4. 类和对象

4.1 基本概念

4.1.1 类与对象



4.1.2 成员变量和成员函数



4.2 封装和访问控制

4.2.1 从struct说起

当单一变量无法完成描述需求的时候,结构体类型解决了这一问题。可以 将多个类型打包成一体,形成新的类型。 这是 c 语言中封装的概念。

|  |
| --- |
| #include <iostream>  using namespace std;  struct Date  {  int  year; month; day;  int  int  };  void init(Date &d)  {  cout<<"year,month,day:"<<endl;  cin>>d.year>>d.month>>d.day;  }  void print(Date & d)  {  cout<<"year month day"<<endl;  cout<<d.year<<":"<<d.month<<":"<<d.day<<endl;  }  bool isLeapYear(Date & d)  {  if((d.year%4==0&& d.year%100 != 0) || d.year%400 == 0)  return true;  else  return false;  }  int main()  {  Date d;  init(d);  print(d); |

|  |
| --- |
| if(isLeapYear(d))  cout<<"leap year"<<endl;  else  cout<<"not leap year"<<endl;  return 0;  } |

对C语言中结构体的操作，都是通过外部函数来实现的。比如

|  |
| --- |
| void init(Date &d)；  void print(Date & d)；  bool isLeapYear(Date & d)； |

4.2.2 封装的访问属性

|  |  |  |  |
| --- | --- | --- | --- |
| 访问属性 属性 对象内部 对象外部 | | | |
| public | 公有 | 可访问 | 可访问 |
| protected | 保护 | 可访问 | 不可访问 |
| private | 私有 | 可访问 | 不可访问 |

struct 中所有行为和属性都是 public 的(默认)。C++中的 class 可以指定行 为和属性的访问方式。

封装,可以达到,对内开放数据,对外屏蔽数据,对外提供接口。 达到了信息 隐 蔽的功能。

比如我们用 struct 封装的类,即知其接口,又可以直接访问其内部数据,这样 却没有达到信息隐蔽的功效。而 class 则提供了这样的功能,屏蔽内部数据,对外 开放接口。

4.2.3 用class去封装带行为的类

class 封装的本质,在于将数据和行为,绑定在一起然后能过对象来完成操 作。

|  |
| --- |
| #include <iostream>  using namespace std;  class Date  39 |

|  |
| --- |
| {  public :  init(Date &d);  void  void  bool  print(Date & d);  isLeapYear(Date & d);  private :  int year;  int month;  int day;  };  void Date::init(Date &d)  {  cout<<"year,month,day:"<<endl;  cin>>d.year>>d.month>>d.day;  }  void Date::print(Date & d)  {  cout<<"year month day"<<endl;  cout<<d.year<<":"<<d.month<<":"<<d.day<<endl;  }  bool Date::isLeapYear(Date & d)  {  if((d.year%4==0 && d.year%100 != 0) || d.year%400 == 0)  return true;  else  return false;  }  int main()  {  Date d;  d.init(d);  d.print(d);  if(d.isLeapYear(d))  cout<<"leap year"<<endl;  else  cout<<"not leap year"<<endl;  return 0;  } |

Date 类 访问自己的成员,可以不需要传引用的方式

封装有2层含义(把属性和方法进行封装 对属性和方法进行访问控制)

Public修饰成员变量和成员函数可以在类的内部和类的外部被访问。

Private修饰成员变量和成员函数只能在类的内部被访问。

41

|  |
| --- |
| struct和class关键字区别  在用struct定义类时，所有成员的默认属性为public  在用class定义类时，所有成员的默认属性为private |

4.3 ⾯向对象编程案例练习



面向对象与面向过程

|  |
| --- |
| 面向对象： 狗.吃(屎)  面向过程： 吃(狗,屎) |

4.3.1 求圆的周长和面积

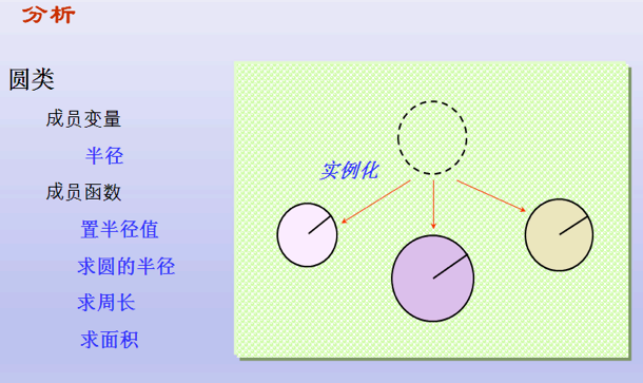
|  |
| --- |
| 数据描述：  半径，周长，面积均用实型数表示  数据处理：  输入半径 *r*；  计算周长 = 2\*π\**r* ；  计算面积 = π\* *r*2 ； |

|  |
| --- |
| 输出半径，周长，面积； |

店㟻**1** ：用结嚞葥店㟻编程，攰圆的慒长和面积

|  |
| --- |
| // count the girth and area of circle  #include <iostream>  using namespace std;  int main (void)  {  double r, girth, area ;  const double PI = 3.1415 ;  cout << "Please input radius:\n" ; //操作符重载  cin >> r ; //输入  girth = 2 \* PI \* r ;  area = PI \* r \* r ;  cout << "radius = " << r << endl ;  cout << "girth = " << girth << endl ;  cout << "area = " << area << endl ;  return 0;  } |





店㟻**2** ：用面向对袋店㟻编程，攰圆的慒长和面积

|  |
| --- |
| #include<iostream>  using namespace std;  class Circle  {  private:  double radius ; //成员变量  public : //类的访问控制  void Set\_Radius( double r )  {  radius = r;  } //成员函数  double Get\_Radius()  {  return radius;  } //通过成员函数设置成员变量  double Get\_Girth()  {  return 2 \* 3.14f \* radius;  } //通过成员函数获取成员变量  double Get\_Area()  {  return 3.14f \* radius \* radius;  }  };  int main(void)  43 |

|  |
| --- |
| {  Circle A, B ; //⽤类定义对象  A.Set\_Radius( 6.23 ) ; //类的调⽤  cout << "A.Radius = " << A.Get\_Radius() << endl ;  cout << "A.Girth = " << A.Get\_Girth() << endl ;  cout << "A.Area = " << A.Get\_Area() << endl ;  B.Set\_Radius( 10.5 ) ;  cout << "B.radius = " << B.Get\_Radius() << endl ;  cout << "B.Girth=" << B.Get\_Girth() << endl ;  cout << "B.Area = " << B.Get\_Area() << endl ;  return 0;  } |

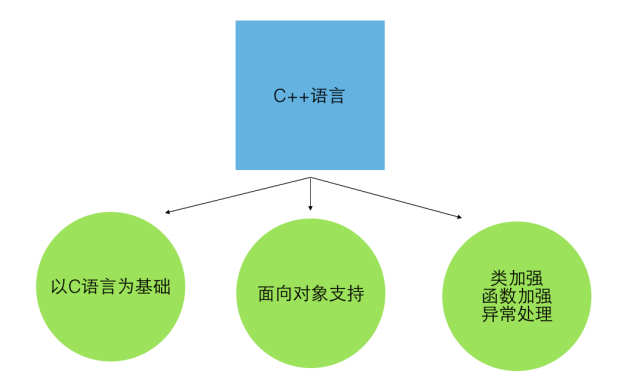
总结：建立类、对象、成员变量、成员函数，输入输入流基本概念。



4.3.2 初学者易犯错误模型

|  |
| --- |
| // demo02\_circle\_err.cpp  #include<iostream>  using namespace std;//c++的命名空间  class circle  {  public :  double r;  double pi = 3.1415926;  double area = pi\*r\*r;  };  int main(void)  {  circle pi;  cout << "请输⼊area" << endl;  cin >> pi.r;  cout << pi.area << endl; //乱码  return 0;  } |

4.3.3 C语言和C++语言的关系



C语言是在实践的过程中逐步完善起来的

|  |
| --- |
| 没有深思熟虑的设计过程  使用时存在很多“灰色地带”  残留量过多低级语言的特征  直接利用指针进行内存操作 |

C语言的目标是高效



当面向过程方法论暴露越来越多的缺陷的时候，业界开始考虑在工程项 目中引入面向对象的设计方法，而第一个需要解决的问题就是：高效的面向对 象语言，并且能够兼容已经存在的代码。

C语言 + 面向对象方法论 ===> Objective C /C++

|  |
| --- |
| C语言和C++并不是对立的竞争关系  C++是C语言的加强，是一种更好的C语言  C++是以C语言为基础的，并且完全兼容C语言的特性 |

学习C++并不会影响原有的C语言知识，相反会根据加深对C的认知； 学习C++可以接触到更多的软件设计方法，并带来更多的机会。

1) C++是一种更强大的C，通过学习C++能够掌握更多的软件设计方法.

2) C++是Java/C#/D等现代开发语言的基础，学习C++后能够快速掌握这 些语言.

3) C++是各大知名软件企业挑选人才的标准之一 .

4.3.4 综合面向对象案例练习

|  |
| --- |
| 面向对象练习1  设计立方体类(cube) ，求出立方体的面积和体积  求两个立方体，是否相等(全局函数和成员函数) |

|  |
| --- |
| 面向对象练习2  设计一个圆形类(AdvCircle)，和一个点类(Point)，计算点在圆内部 还是圆外  即：求点和圆的关系(圆内和圆外) |

|  |
| --- |
| 面向对象练习3  对于第二个案例，类的声明和类的实现分开 |



|  |
| --- |
| 作业1 ：编写C++程序完成冰下功能： 1)足义一个Point 类,其属性己沽点的坐标,堤侠计算两点乏间距离的万法f 2)足义一个圆玷类,其属性己沽圆心和半径f 3)创建两个圆玷对象,堤示用户输入圆心坐标和半径,判断两个圆是否相交,并 输出结呆。 |

|  |
| --- |
| 作业2 ：设计并测试一个沼为Rectangle的矩玷类,其属性为矩玷的左下角与右上角 两个点的坐标,根据坐标能计算出矩玷的面识 |

|  |
| --- |
| 作业3 ：足义一个Tree 类,有成员ages (树龄),成员凶软grow (int years)对ages 加上years,age ()显示tree对象的ages的值。 |

4.4 对象的构造和析构

4.4.1 如果没有构造函数？

面向对象的思想是从生活中来，手机、车出厂时，是一样的。生活中存 在的对象都是被初始化后才上市的；初始状态是对象普遍存在的一个状态的 。

如果不用构造函数初始化，该怎么办：

为每个类都提供一个public的initialize函数；

对象创建后立即调用initialize函数进行初始化。

缺点

1)initialize只是一个普通的函数，必须显示的调用 2)一旦由于失误的原因，对象没有初始化，那么结果将是不确定的 没有初始化的对象，其内部成员变量的值是不定的。

|  |
| --- |
| #include <iostream>  using namespace std;  class Test  {  public :  void init(int a, int b)  {  m\_a = a;  m\_b = b;  }  private:  int m\_a;  int m\_b;  };  int main(void)  {  Test t1;  int a = 10;  int b = 20;  t1.init(a, b);  Test tArray[3];  //⼿动调⽤显⽰初始化函数  tArray[0].init(0, 0);  tArray[1].init(0, 0);  tArray[2].init(0, 0); |

|  |
| --- |
| Test t21;  //⼿动调⽤显⽰初始化函数  t21.init(0, 0);  Test t22;  //⼿动调⽤显⽰初始化函数  t22.init(0, 0);  Test t23;  //⼿动调⽤显⽰初始化函数  t23.init(0, 0);  //在这种场景之下 显⽰的初始化⽅案 显得很蹩脚  Test tArray2[3] = {t21, t22, t23};  //在这种场景之下 ,满⾜不了 ,编程需要  Test tArray3[1999] = {t21, t22, t23};  return 0;  } |

所以C++对类提供了一个给对象的初始化方案，就是构造函数。

4.4.2 构造函数

定义

C++中的类可以定义与类名相同的特殊成员函数，这种与类名相同的成员 函数叫做构造函数.

|  |
| --- |
| class 类名 {  类名(形式参数) {  构造体  }  } |

|  |
| --- |
| class A  {  A(形参)  {  }  } |

调用

自动调用 ：一般情况下C++编译器会自动调用构造函数.

手动调用 ：在一些情况下则需要手工调用构造函数.

|  |
| --- |
| 规则**:**  **1** 在对象创建时自动调用**,** 完成初始化相关工作。  **2** 无返回值,与类名同,默认无参,可以重载,可默认参数。  **3** 一经实现,默认不复存在。 |

4.4.3 析构函数

定义

C++中的类可以定义一个特殊的成员函数清理对象，这个特殊的成员函数 叫做析构函数.

|  |
| --- |
| class 类名  {  ~类名() {  析构体  }  } |

|  |
| --- |
| class A  {  ~A()  {}  } |

|  |
| --- |
| 规则:  **1** 对象销毁时,自动调用。完成销毁的善后工作。  **2** 无返值 ,与类名同。无参。不可以重载与默认参数 |

析构函数的作用,并不是删除对象,而在对象销毁前完成的一些清理工作。

4.4.4 构造函数的分类及调用

|  |
| --- |
| class Test  {  public:  //⽆参数构造函数  Test()  {  ;  }  //带参数的构造函数  Test(int a, int b)  {  ;  }  //赋值构造函数  Test(const Test &obj)  {  ;  }  private:  int a;  int b;  }; |

(1) 无参构造函数

|  |
| --- |
| #include <iostream>  using namespace std;  class Test  {  public:  //⽆参数构造函数  Test()  {  a = 0;  b = 0;  cout <<"Test() ⽆参构造函数执⾏" <<endl;  }  private:  int a;  int b;  };  int main(void)  {  Test t; //调⽤⽆参构造函数 |

|  |
| --- |
| return 0;  } |

(2) 有参数构造函数

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;  class Test  {  private :  int a;  public:  //带参数的构造函数  Test(int a)  {  cout <<"a = " << a <<endl;  }  Test(int a, int b)  {  cout <<"a = " << a << ", b = " << b <<endl;  }  }; | |
| int main()  {  Test t1(10);  Test t2(10, 20);  return 0;  } | //调⽤有参构造函数 Test(int a)  //调⽤有参构造函数 Test(int a, int b) |

(3) 拷贝构造函数

由己存在的对象,创建新对象。也就是说新对象,不由构造器来构造,而是由 拷贝 构造器来完成。拷贝构造器的格式是固定的。

|  |
| --- |
| class 类名  {  类名(const 类名 & another)  {  拷⻉构造体  }  } |

|  |
| --- |
| class A  {  A(const A & another)  {}  } |

使用拷贝构造函数的几种场合

|  |
| --- |
| #include <iostream>  using namespace std;  class Test  {  public:  Test() //⽆参构造函数  {  cout<<"我是⽆参构造函数，被调⽤了"<<endl;  }  Test(int a) //带参数的构造函数  {  m\_a = a;  }  Test(const Test &another\_obj) //拷⻉构造函数  {  cout<<"我也是构造函数，我是通过另外⼀个对象 , 来初始化我⾃⼰"<<endl; m\_a = another\_obj.m\_a;  }  ~Test()  {  cout<<"我是析构函数，⾃动被调⽤了"<<endl;  }  void printT()  {  cout << "m\_a = " << m\_a <<endl;  }  private :  int m\_a;  }; |

(1)

|  |
| --- |
| //拷⻉构造函数的第⼀个应⽤场景  int main(void)  { |

54

|  |
| --- |
| Test t1(10);  Test t2 = t1; //⽤对象t1 初始化 对象 t2  t2.printT();  return 0;  } |

(2)

|  |
| --- |
| //拷⻉构造函数的第⼆个应⽤场景  int main(void)  {  Test t1(10);  Test t2(t1); //⽤对象t1 初始化 对象 t2  t2.printT();  return 0;  } |

(3)

|  |
| --- |
| //拷⻉构造函数的第三个应⽤场景  #include <iostream>  using namespace std;  class Location  {  public:  //带参数的构造函数  Location( int xx = 0 , int yy = 0 )  {  X = xx ;  Y = yy ;  cout << "Constructor Object." <<endl;  }  //copy构造函数 完成对象的初始化  Location(const Location & obj) //copy构造函数  {  X = obj.X;  Y = obj.Y;  cout <<"Copy Constructor." <<endl;  }  ~Location()  { |

|  |  |  |
| --- | --- | --- |
| cout << X << "," << Y << " Object destroyed." << endl ;  }  int GetX () {  return X;  }  int GetY () {  return Y;  }  private :  int X;  int Y;  };  void func(Location p) //会执⾏ p = b 的操作， p会调⽤copy构造函数进⾏初始化  {  cout <<"func begin" <<endl;  cout<<p.GetX()<<endl;  cout <<"func end" <<endl;  } | | |
| void test()  {  Location  Location | a(1, 2);  b = a; | //对象a 调⽤带参数的构造函数进⾏初始化  //对象b 调⽤copy构造函数进⾏初始化 |
| cout <<"----" <<endl;  func(b); //b实参取初始化形参p,会调⽤copy构造函数  }  int main(void)  {  test();  return 0;  } | | |

(+)

|  |
| --- |
| #include <iostream>  using namespace std;  class Location  {  //带参数的构造函数  Location( int xx = 0 , int yy = 0 )  {  X = xx ;  Y = yy ;  cout << "Constructor Object." <<endl;  }  55 |

|  |
| --- |
| //copy构造函数 完成对象的初始化  Location(const Location & obj) //copy构造函数  {  X = obj.X;  Y = obj.Y;  cout <<"Copy Constructor." <<endl;  }  ~Location()  {  cout << X << "," << Y << " Object destroyed." << endl ;  }  int GetX () {  return X;  }  int GetY () {  return Y;  }  private :  int X;  int Y;  };  /g函数 返回⼀个元素  //结论1 : 函数的返回值是⼀个元素 (复杂类型的), 返回的是⼀个新的匿名对象(所以会调⽤匿名 对象类的copy构造函数)  //  //结论2: 有关 匿名对象的去和留  //如果⽤匿名对象 初始化 另外⼀个同类型的对象 , 匿名对象 转成有名对象  //如果⽤匿名对象 赋值给 另外⼀个同类型的对象 , 匿名对象 被析构  //  //设计编译器的⼤⽜们:  //我就给你返回⼀个新对象(没有名字 匿名对象)  Location g()  {  Location temp(1, 2);  return temp;  }  void test1()  {  g();  }  void test2()  {  //⽤匿名对象初始化m 此时c++编译器 直接把匿名对转成m; (扶正) 从匿名转成有名字了m //就是将这个匿名对象起了名字m,他们都是同⼀个对象  56 |

|  |
| --- |
| Location m = g();  printf("匿名对象 ,被扶正 ,不会析构掉\n");  cout<<m.GetX()<<endl;;  }  void test3()  {  //⽤匿名对象 赋值给 m2后 , 匿名对象被析构  Location m2(1, 2);  m2 = g();  printf("因为⽤匿名对象=给m2, 匿名对象 ,被析构\n");  cout<<m2.GetX()<<endl;;  }  int main(void)  {  test1();  test2();  test3();  return 0;  } |

(5) 默认构造函数

二个特殊的构造函数

1)默认无参构造函数

当类中没有定义构造函数时，编译器默认提供一个无参构造函数，并且 其函数体为空

2)默认拷贝构造函数

当类中没有定义拷贝构造函数时，编译器默认提供一个默认拷贝构造函 数，简单的进行成员变量的值复制

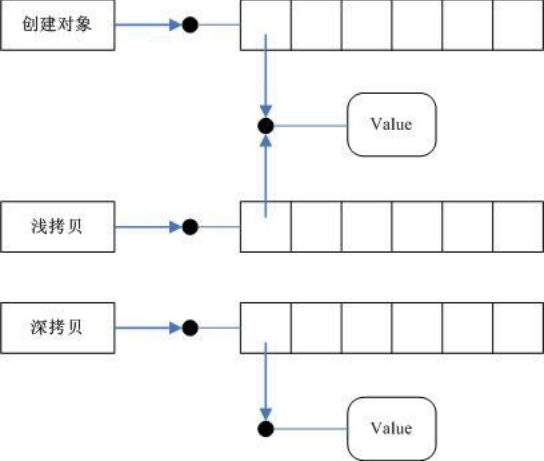
4.4.5 构造函数规则

|  |
| --- |
| 规则:  **1** 系统提供默认的拷贝构造器。一经实现,不复存在。  **2** 系统提供的时等位拷贝,也就是所谓的浅浅的拷贝。  **3** 要实现深拷贝,必须要自定义。  57 |

|  |
| --- |
| #include <iostream>  using namespace std;  //当类中定义了拷⻉构造函数时， c++编译器不会提供⽆参数构造函数  //当类中定义了有参数构造函数是,c++编译器不会提供⽆参数构造函数  //在定义类时, 只要你写了构造函数,则必须要⽤  class Test  {  public :  Test(const Test& obj) //copy构造函数 作⽤ : ⽤⼀个对象初始化另外⼀个对象  {  a = obj.a + 100;  b = obj.b + 100;  }  #if 0  Test()  {  }  #endif  void printT()  {  cout << "a:" << a << "b: "<<b<< endl;  }  private:  int a;  int b;  };  int main(void)  {  Test t1; //error, 没有合适的构造函数  return 0;  } |

4.4.6 浅拷贝与深拷贝

系统提供默认的拷贝构造器,一经定义不再提供。但系统提供的默认拷贝 构造器是 等位拷贝,也就是通常意义上的浅拷贝。如果类中包含的数据元素全 部在栈上,浅拷贝 也可以满足需求的。但如果堆上的数据,则会发生多次析构行 为。



|  |
| --- |
| #include <iostream>  using namespace std;  class Name  {  public :  Name(const char \*myp)  {  m\_len = strlen(myp);  m\_p =(char \*) malloc(m\_len + 1); //  strcpy(m\_p, myp);  }  //Name obj2 = obj1;  //解决⽅案: ⼿⼯的编写拷⻉构造函数 使⽤深copy  Name(const Name& obj1)  {  m\_len = obj1.m\_len;  m\_p = (char \*)malloc(m\_len + 1);  strcpy(m\_p, obj1.m\_p);  }  ~Name()  {  if (m\_p != NULL)  {  free(m\_p);  59 |

|  |
| --- |
| m\_p = NULL;  m\_len = 0;  }  }  private :  char \*m\_p ;  int m\_len;  };  //对象析构的时候 出现coredump  void test()  {  Name obj1("abcdefg");  Name obj2 = obj1; //C++编译器提供的 默认的copy构造函数 浅拷⻉  Name obj3("abc");  //obj3 = obj2; // 当执⾏ =操作的时候， C++编译器也是使⽤的默认拷⻉构造函数，也是  浅拷⻉  }  int main(void)  {  test();  return 0;  } |

4.4.7 构造函数初始化列表

如果我们有一个类成员，它本身是一个类或者是一个结构，而且这个成 员它只有一个带参数的构造函数，没有默认构造函数。这时要对这个类成员进 行初始化，就必须调用这个类成员的带参数的构造函数，

如果没有初始化列表，那么他将无法完成第一步，就会报错。

|  |
| --- |
| #include <iostream>  using namespace std;  class A {  public :  A(int a) {  m\_a = a;  }  private:  int m\_a;  };  60 |

|  |
| --- |
| class B {  public :  B(int b) {  m\_b = b;  }  private:  int m\_b;  A obja; //当A的对象 是B类的⼀个成员的时候，在初始化B对象的时候，  //⽆法给B 分配空间，因为⽆法初始化A类对象  };  int main(void)  {  A obja(10);  B objb(20);//error,  return 0;  } |

|  |
| --- |
| #include <iostream>  using namespace std;  class ABC  {  public :  ABC(int a, int b, int c)  {  this->a = a;  this->b = b;  this->c = c;  printf("a:%d,b:%d,c:%d \n", a, b, c);  printf("ABC construct ..\n");  }  ~ABC()  {  printf("a:%d,b:%d,c:%d \n", a, b, c);  printf("~ABC() ..\n");  }  private :  int a;  int b;  int c;  };  class MyD  {  public :  MyD():abc1(1,2,3),abc2(4,5,6),m(100)  {  cout<<"MyD()"<<endl;  61 |

|  |
| --- |
| }  ~MyD()  {  cout<<"~MyD()"<<endl;  }  private:  ABC abc1;  ABC abc2;  const int m;  };  int main()  {  MyD myD;  return 0;  } |

当类成员中含有一个const对象时，或者是一个引用时，他们也必须要通 过成员初始化列表进行初始化 ，因为这两种对象要在声明后马上初始化，而在 构造函数中，做的是对他们的赋值，这样是不被允许的。

初始化列表中的初始化顺序,与声明顺序有关,与前后赋值顺序无关。

4.4.8 强化训练

|  |
| --- |
| 练习1,分析下列代码问造器和析问器的执行顺序 |

|  |
| --- |
| #include <iostream>  using namespace std;  class ABCD  {  public :  ABCD(int a, int b, int c)  {  \_a = a;  \_b = b;  \_c = c;  printf("ABCD() construct, a:%d,b:%d,c:%d \n", \_a, \_b, \_c);  }  ~ABCD()  {  printf("~ABCD() construct,a:%d,b:%d,c:%d \n", \_a, \_b, \_c);  }  int getA()  {  return \_a;  62 |

|  |
| --- |
| }  private :  int \_a;  int \_b;  int \_c;  };  class MyE  {  public :  MyE():abcd1(1,2,3),abcd2(4,5,6),m(100)  {  cout<<"MyD()"<<endl;  }  ~MyE()  {  cout<<"~MyD()"<<endl;  }  MyE(const MyE & obj):abcd1(7,8,9),abcd2(10,11,12),m(100)  {  printf("MyD(const MyD & obj)\n");  }  public:  ABCD abcd1; //c++编译器不知道如何构造abc1  ABCD abcd2;  const int m;  };  int doThing(MyE mye1)  {  printf("doThing() mye1.abc1.a:%d \n", mye1.abcd1.getA());  return 0;  }  int run()  {  MyE myE;  doThing(myE);  return 0;  }  int main(void)  {  run();  return 0;  } |

|  |
| --- |
| 练习2根据练习1分析冰下问造器和析问的调用顺序 |

|  |
| --- |
| int run2()  {  printf("run2 start..\n");  ABCD(400, 500, 600); //临时对象的⽣命周期  //ABCD abcd = ABCD(100, 200, 300);  printf("run2 end\n");  return 0;  } |

|  |
| --- |
| 练习3问造凶软再调问造凶软,分析冰下代码结呆 |

|  |
| --- |
| #include <iostream>  using namespace std;  //构造中调⽤构造是危险的⾏为  class MyTest  {  public :  MyTest(int a, int b, int c)  {  \_a = a;  \_b = b;  \_c = c;  }  MyTest(int a, int b)  {  \_a = a;  \_b = b;  MyTest(a, b, 100); //产⽣新的匿名对象  }  ~MyTest()  {  printf("MyTest~:%d, %d, %d\n", \_a, \_b, \_c);  }  int getC()  {  return \_c;  }  void setC(int val)  {  \_c = val;  }  64 |

|  |
| --- |
| private:  int \_a;  int \_b;  int \_c;  };  int main()  {  MyTest t1(1, 2);  printf("c:%d\n", t1.getC()); //请问c的值是？  return 0;  } |

4.5 对象动态建⽴和释放 new 和delete

在软件开发过程中，常常需要动态地分配和撤销内存空间，例如对动态 链表中结点的插入与删除。在C语言中是利用库函数malloc和free来分配和撤 销内存空间的。C++提供了较简便而功能较强的运算符new和delete来取代 malloc和free函数。

new和delete是运算符，不是函数，因此执行效率高。



虽然为了与C语言兼容， C++仍保留malloc和free函数，但建议用户不用 malloc和free函数，而用new和delete运算符。

|  |
| --- |
| new int;  //开辟⼀个存放整数的存储空间，返回⼀个指向该存储空间的地址(即指针)  new int(100);  //开辟⼀个存放整数的空间，并指定该整数的初值为100，返回⼀个指向该存储空间的地址  new char[10];  //开辟⼀个存放字符数组(包括10个元素)的空间，返回⾸元素的地址  new int[5][4];  //开辟⼀个存放⼆维整型数组(⼤⼩为5\*4)的空间，返回⾸元素的地址  float \*p=new float (3.14159);  //开辟⼀个存放单精度数的空间，并指定该实数的初值为//3.14159，将返回的该空间的地址 赋给指针变量p |



用new分配数组空间时不能指定初值。如果由于内存不足等原因而无法正 常分配空间，则new会返回一个空指针NULL ，用户可以根据该指针的值判断分

配空间是否成功。

|  |
| --- |
| malloc 不会调用类的构造函数,而new会调用类的构造函数  Free不会调用类的析构函数，而delete会调用类的析构函数 |

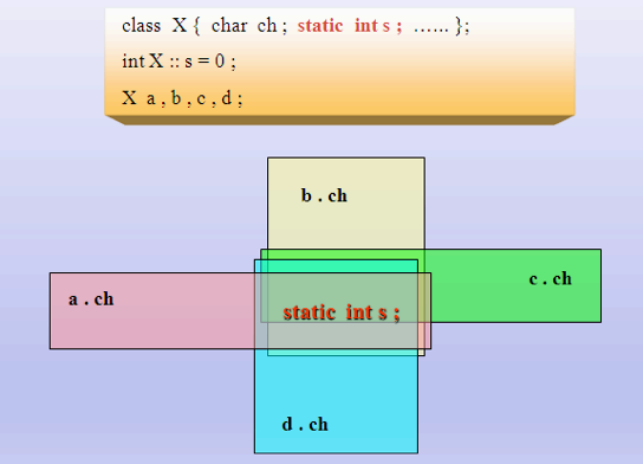
4.6 静态成员变量和成员函数

在 C++中,静态成员是属于整个类的而不是某个对象,静态成员变量只存储 一份供 所有对象共用。所以在所有对象中都可以共享它。使用静态成员变量实 现多个对象之间 的数据共享不会破坏隐藏的原则,保证了安全性还可以节省内 存。

类的静态成员,属于类,也属于对象,但终归属于类。

4.6.1 静态成员变量

|  |
| --- |
| //声明  static 数据类型 成员变量; //在类的内部  //初始化  数据类型 类名::静态数据成员 = 初值; //在类的外部  //调⽤  类名::静态数据成员  类对象 .静态数据成员 |



|  |
| --- |
| 案例：主成一个 Box 类,要示所在 Box 的高度 height 一致。 |

|  |
| --- |
| #include <iostream>  using namespace std;  class Box  {  public :  Box(int l, int w):length(l),width(w) {  }  int volume()  {  return length \* width \* height;  }  static int height;  int length;  int width;  };  int Box::height = 5;  int main()  {  // cout<<sizeof(Box)<<endl;  // Box b(2,3);  // cout<<sizeof(b)<<endl;  cout<<Box::height<<endl;  Box b(1,1);  cout<<b.height<<endl;  cout<<b.volume()<<endl;  return 0;  } |

|  |
| --- |
| 1,static 成员变量实现了同类对象间信息共享。  2,static 成员类外存储,求类大小,并不包含在内。  3,static 成员是命名空间属于类的全局变量,存储在 data 区。  4,static 成员只能类外初始化。  5,可以通过类名访问(无对象生成时亦可),也可以通过对象访问。 |

4.6.2 静态成员函数

|  |
| --- |
| //声明  static 函数声明  //调⽤  类名::函数调⽤  类对象 .函数调⽤ |

|  |
| --- |
| #include <iostream>  using namespace std;  class Student  {  public :  Student(int n,int a,float s):num(n),age(a),score(s){}  void total()  {  count++;  sum += score;  }  static float average();  private:  int num;  int age;  float score;  static float sum;  static int count;  };  float Student::sum = 0;  int Student::count = 0;  float Student::average() {  return sum/count;  }  int main()  {  Student stu[3]= {  Student(1001,14,70),  Student(1002,15,34),  Student(1003,16,90)  };  for(int i=0; i<3; i++) {  stu[i].total();  } |

|  |
| --- |
| cout<<Student::average()<<endl;  return 0;  } |

|  |
| --- |
| 1,静态成员函数的意义,不在于信息共享,数据沟通,而在于管理静态数据成员, 完  成对静态数据成员的封装。  2,静态成员函数只能访问静态数据成员。 原因:非静态成员函数,在调用时this 指 针被当作参数传进。而静态成员函数属于类,而不属于对象,没有 this 指针。 |

4.7 编译器对属性和⽅法的处理机制

4.7.1 静态成员占多大

|  |  |
| --- | --- |
| #include <iostream>  using namespace std; | |
| class C1  {  public :  int  int  int  }; //12 | i; //4  j; //4  k; //4 |
| class C2  {  public :  int i;  int j;  int k;  static int m; //4  public :  int getK() const { return k; } //4  void setK(int val) { k = val; } //4  };  struct S1  {  int i;  int j;  int k;  }; //12  struct S2 | |

|  |
| --- |
| {  int i;  int j;  int k;  static int m;  }; //12?  int main()  {  cout <<"c1 : " << sizeof(C1) <<endl;  cout <<"c1 : " << sizeof(C2) <<endl;  cout <<"c1 : " << sizeof(S1) <<endl;  cout <<"c1 : " << sizeof(S2) <<endl;  return 0;  } |

4.7.2 处理机制

通过上面的案例，我们可以的得出：

C++类对象中的成员变量和成员函数是分开存储的

成员变量：

普通成员变量：存储于对象中，与struct变量有相同的内存布局和字节对 齐方式

静态成员变量：存储于全局数据区中

成员函数：存储于代码段中。

祖?对袋共用 - 块代码?代码吕如何区h具体对袋

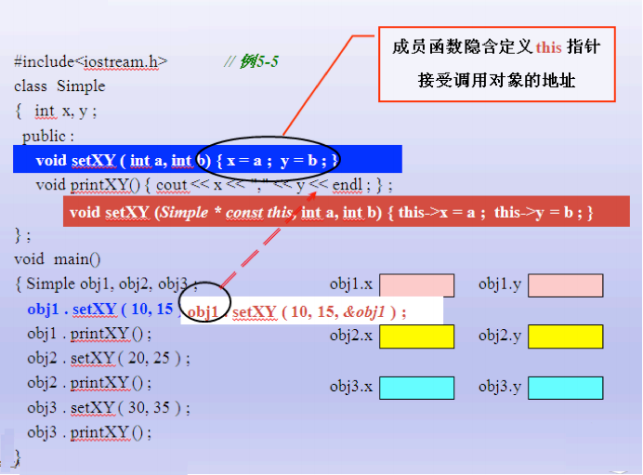
的非?

换句话说：int getK() const { return k; }，代码是如何区分，具体obj1、 obj2、obj3对象的k值？



|  |
| --- |
| **1** 、C++类对象中的成员变量和成员函数是分开存储的。 C语言中的内存四区模 型仍然有效！  **2** 、**C++**中类的普通成员函数都隐式包含一个指向当前对象的**this**指针。  3、静态成员函数、成员变量属于类  4、静态成员函数与普通成员函数的区别  静态成员函数不包含指向具体对象的指针  普通成员函数包含一个指向具体对象的指针 |

4.7.3 this指针



|  |
| --- |
| #include <iostream>  using namespace std;  class Test  {  public :  Test(int a, int b) //---> Test(Test \*this, int a, int b)  {  this->a = a;  this-> b = b;  }  void printT()  {  cout<<"a: " <<a <<endl;  cout<< "b: " << this->b <<endl;  }  protected :  private :  int a;  int b;  }; |

|  |  |
| --- | --- |
| int main(void)  {  Test t1(1, 2);  t1.printT();  return 0;  } | //===> Test(&t1, 1, 2);  // ===> printT(&t1) |

|  |
| --- |
| (1):若类成员凶软的玷冬 和 类的属性,沼早相同,遍过this指针采解决。  (2) :类的成员凶软可遍过const修饰。 |

4.7.4 全局函数与成员函数

1、把全局函数转化成成员函数，通过this指针隐藏左操作数 Test add(Test &t1, Test &t2)===》Test add(Test &t2)

2、把成员函数转换成全局函数，多了一个参数

void printAB()===》void printAB(Test \*pthis)

3、函数返回元素和返回引用

|  |
| --- |
| Test& add(Test &t2) //\*this //函数返回引⽤  {  this->a = this->a + t2.getA();  this->b = this->b + t2.getB();  return \*this; //\*操作让this指针回到元素状态  }  Test add2(Test &t2) //\*this //函数返回元素  {  //t3是局部变量  Test t3(this->a+t2.getA(), this->b + t2.getB()) ;  return t3;  } |

4.8 强化练习

|  |
| --- |
| 练习1  某商店经销一神货物。货物购进和卖出时冰箱为单位,各箱的重量不一样, 因此,商店需要记录目前库存的总重量。现在用C++模拟商店货物购进和卖出的晴 况。 |

|  |
| --- |
| 练习2 软组类封装  目标：解决实际问题,训练问造凶软、coP当问造凶软等,为橡作符重载做准 备 |

4.9 友元

采用类的机制后实现了数据的隐藏与封装,类的数据成员一般定义为私有成 员,成员函 数一般定义为公有的,依此提供类与外界间的通信接口。但是,有时需 要定义一些函数,这 些函数不是类的一部分,但又需要频繁地访问类的数据成员, 这时可以将这些函数定义为该 函数的友元函数。除了友元函数外,还有友元类, 两者统称为友元。友元的作用是提高了程 序的运行效率(即减少了类型检查和 安全性检查等都需要时间开销),但它破坏了类的封装 性和隐藏性,使得非成员函 数可以访问类的私有成员。

友元可以是一个函数,该函数被称为友元函数;友元也可以是一个类,该类被 称为友元 类。

同类对象间无私处

|  |
| --- |
| MyString::MyString(const MyString & other)  {  int len = strlen(other.\_str);  this->\_str = new char[len+1];  strcpy(this->\_str,other.\_str);  } |

异类对象间有友元

4.9.1 友元函数

友元函数是可以直接访问类的私有成员的非成员函数。它是定义在类外 的普通函 数,它不属于任何类,但需要在类的定义中加以声明,声明时只需在友元 的名称前加上 关键字 friend,其格式如下:

|  |
| --- |
| friend 类型 函数名(形式参数); |

一个函数可以是多个类的友元函数,只需要在各个类中分别声明。

全局函数作友元函数

|  |
| --- |
| #include<iostream>  #include<cmath>  using namespace std;  class Point  {  public :  Point(double xx, double yy)  {  x = xx;  y = yy;  }  void Getxy();  friend double Distance(Point &a, Point &b);  private:  double x, y;  };  void Point::Getxy()  {  cout << "(" << x << "," << y << ")" << endl;  }  double Distance(Point &a, Point &b)  {  double dx = a.x - b.x;  double dy = a.y - b.y;  return sqrt(dx\*dx + dy\*dy);  }  int main(void)  {  Point p1(3.0, 4.0), p2(6.0, 8.0);  p1.Getxy();  p2.Getxy();  double d = Distance(p1, p2);  cout << "Distance is " << d << endl;  return 0;  } |

类成员函数作友元函数

|  |
| --- |
| #include<iostream>  #include<cmath>  using namespace std;  class Point;  //前向声明,是⼀种不完全型声明,即只需提供类名**(**⽆需提供类实现**)**即可。仅可⽤ 于声明指针和引 ⽤。  class ManagerPoint  {  public :  double Distance(Point &a, Point &b);  };  class Point  {  public :  Point(double xx, double yy)  {  x = xx;  y = yy;  }  void Getxy();  friend double ManagerPoint::Distance(Point &a, Point &b);  private:  double x, y;  };  void Point::Getxy()  {  cout << "(" << x << "," << y << ")" << endl;  }  double ManagerPoint::Distance(Point &a, Point &b) {  double dx = a.x - b.x;  double dy = a.y - b.y;  return sqrt(dx\*dx + dy\*dy);  }  int main(void)  {  Point p1(3.0, 4.0), p2(6.0, 8.0);  p1.Getxy();  p2.Getxy();  ManagerPoint mp;  float d = mp.Distance(p1,p2);  cout << "Distance is " << d<< endl;  return 0;  } |

4.9.2 友元对象

友元类的所有成员函数都是另一个类的友元函数,都可以访问另一个类中 的隐藏信息(包括私有成员和保护成员)。

当希望一个类可以存取另一个类的私有成员时,可以将该类声明为另一类 的友元类。定义友元类的语句格式如下:

|  |
| --- |
| friend class 类名;  其中 :friend 和 class 是关键字 ,类名必须是程序中的⼀个已定义过的类。  例如 ,以下语句说明类 B 是类 A 的友元类 :  class A  {  ...  public:  friend class B;  }; ... |

经过以上说明后,类 B 的所有成员函数都是类 A 的友元函数,能存取类 A 的私有 成员和保护成员。

|  |
| --- |
| class A  {  public :  inline void Test()  {  }  private :  int x ,y; friend Class B;  }  class B  {  public :  inline void Test()  {  A a;  printf("x=%d,y=%d".a.x,a.y);  }  } |

4.9.3 论友元

声明位置

友元声明以关键字 friend 开始,它只能出现在类定义中。因为友元不是授 权类的 成员,所以它不受其所在类的声明区域 public private 和 protected 的影 响。通常我们 选择把所有友元声明组织在一起并放在类头之后.

友元的利弊

友元不是类成员,但是它可以访问类中的私有成员。友元的作用在于提高 程序的运 行效率,但是,它破坏了类的封装性和隐藏性,使得非成员函数可以访问 类的私有成员。 不过,类的访问权限确实在某些应用场合显得有些呆板,从而容 忍了友元这一特别语法 现象。

注急薯项

(1) 友元关系不能被继承。

(2) 友元关系是单向的,不具有交换性。若类 B 是类 A 的友元,类 A 不一定是类 B 的友元,要看在类中是否有相应的声明。

(3) 友元关系不具有传递性。若类 B 是类 A 的友元,类 C 是 B 的友元,类 C 不一 定 是类 A 的友元,同样要看类中是否有相应的声明。

4.10 运算符重载

所谓重载，就是重新赋予新的含义。 函数重载就是对一个已有的函数赋 予新的含义，使之实现新功能，因此，一个函数名就可以用来代表不同功能的 函数，也就是”一名多用”。

运算符也可以重载。实际上，我们已经在不知不觉之中使用了运算符重 载。例如，大家都已习惯于用加法运算符”+”对整数、单精度数和双精度数进行 加法运算，如5+8， 5.8 +3.67等，其实计算机对整数、单精度数和双精度数的 加法操作过程是很不相同的， 但由于C++已经对运算符”+”进行了重载，所以就 能适用于int, ﬂoat, doUble类型的运算。

又如”<<“是C++的位运算中的位移运算符(左移)，但在输出操作中又是 与流对 象cout 配合使用的流插入运算符， ”>>“也是位移运算符(右移)，但在输 入操作中又是与流对象 cin 配合使用的流提取运算符。这就是运算符重载 (operator overloading)。C++系统对”<<“和”>>“进行了重载，用户在不同的场合下 使用它们时，作用是不同 的。对”<<“和”>>“的重载处理是放在头文件stream中

的。因此，如果要在程序中用”<<“和”>>”作流插入运算符和流提取运算符，必 须在本文件模块中包含头文件stream(当然还应当包括”using namespace std“)。 现在要讨论的问题是：用户能否根据自己的需要对C++已提供的运算符进行重 载，赋予它们新的含义，使之一名多用.

运算符重载的本质是函数重载。

重载函数的一般格式如下:

|  |
| --- |
| 函数类型 operator 运算符名称(形参表列) {  重载实体;  } |

operator 运算符名称 在一起构成了新的函数名。比如

const Complex operator+(const Complex &c1,const Complex &c2);

我们会说,operator+ 重载了重载了运算符+。

4.10.1 友元重载

|  |
| --- |
| #include <iostream>  using namespace std;  class Complex  {  public :  Complex(float x=0, float y=0) :\_x(x),\_y(y){}  void dis() {  cout<<"("<<\_x<<","<<\_y<<")"<<endl;  }  friend const Complex operator+(const Complex &c1,const Complex &c2);  private :  \_x;  float  float  \_y;  };  const Complex operator+(const Complex &c1,const Complex &c2) {  return Complex(c1.\_x + c2.\_x,c1.\_y + c2.\_y);  }  int main() {  Complex c1(2,3);  Complex c2(3,4); |

|  |
| --- |
| c1.dis();  c2.dis();  Complex c3 = c1+c2;  //Complex c3 = operator+(c1,c2);  c3.dis();  return 0;  } |

4.10.2 成员重载

|  |
| --- |
| #include <iostream>  using namespace std;  class Complex  {  public :  Complex(float x=0, float y=0) :\_x(x),\_y(y){}  void dis() {  cout<<"("<<\_x<<","<<\_y<<")"<<endl;  }  friend const Complex operator+(const Complex &c1,const Complex &c2);  const Complex operator+(const Complex &another);  private :  \_x;  float  float  \_y;  };  const Complex operator+(const Complex &c1,const Complex &c2)  {  cout<<"友元函数重载"<<endl;  return Complex(c1.\_x + c2.\_x,c1.\_y + c2.\_y);  }  const Complex Complex::operator+(const Complex & another)  {  cout<<"成员函数重载"<<endl;  return Complex(this->\_x + another.\_x,this->\_y + another.\_y);  }  int main()  {  Complex c1(2,3);  Complex c2(3,4);  c1.dis();  c2.dis();  81 |

|  |
| --- |
| //Complex c3 = c1+c2;  //Complex c3 = operator+(c1,c2);  Complex c3 = c1+c2;  c3.dis();  return 0;  } |

|  |
| --- |
| int a = 3;  int b = 4;  (a+b) = 100; 这种语法是错的,所以重载函数的返回值必须是 const |

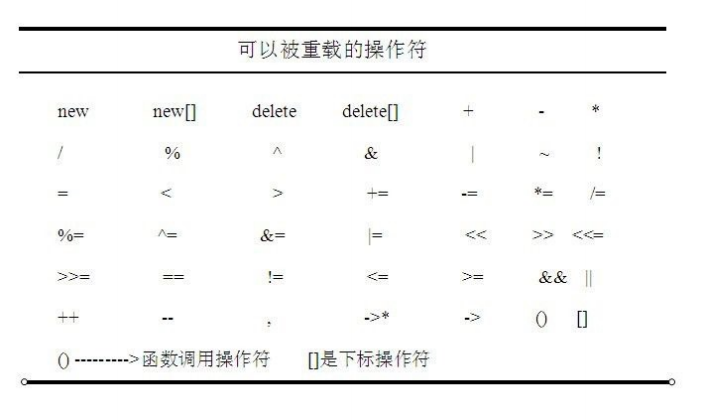
4.10.2 重载规则

(1)C++不允许用户自己定义新的运算符,只能对已有的 C++运算符进行重载。

例如,有人觉得 BASIC 中用“\* \*”作为幂运算符很方便,也想在 C++中将“\* \*”定义为幂运算符,用“3\* \*5”表示 35,这是不行的。

(2)C++允许重载的运算符

C++中绝大部分运算符都是可以被重载的。



不能重载的运算符只有 4 个:

前两个运算符不能重载是为了保证访问成员的功能不能被改变,域运算符合 sizeof 运算符的运算对象是类型而不是变量或一般表达式,不具备重载的特征。

3)重载不能改变运算符运算对象(即操作数)的个数。

如,关系运算符“>”和“<”等是双目运算符,重载后仍为双目运算符,需要两 个参数。运算符”+ “,”- “,”\* “,”& “等既可以作为单目运算符,也可以作为双 目运算符,可以分别将它们重载为单目运算符或双目运算符。

4)重载不能改变运算符的优先级别。

例如”\* “和”/ “优先级高于”+ “和”- “,不论怎样进行重载,各运算符之间 的优先级不会改变。有时在程序中希望改变某运算符的优先级,也只能使用加括号的 方法 强制改变重载运算符的运算顺序。

(5)重载不能改变运算符的结合性。

如,复制运算符”= “是右结合性(自右至左),重载后仍为右结合性。

(6)重载运算符的函数不能有默认的参数

否则就改变了运算符参数的个数,与前面第(3)点矛盾。

(7)重载的运算符必须和用户定义的自定义类型的对象一起使用,其参数至少应有 一 个是类对象(或类对象的引用)。

也就是说,参数不能全部是 C++的标准类型,以防止用户修改用于标准类型数据 成 员的运算符的性质,如下面这样是不对的:

复制代码 代码如下:

int operator + (int a,int b) {

return(a-b); }

原来运算符+的作用是对两个数相加,现在企图通过重载使它的作用改为两个数 相 减。如果允许这样重载的话,如果有表达式 4+3,它的结果是 7 还是 1 呢?显然,这 是 绝对要禁止的。

(8)用于类对象的运算符一般必须重载,但有两个例外,运算符”= “和运算 符”& “不 必用户重载。

复制运算符”= “可以用于每一个类对象,可以用它在同类对象之间相互赋值。 因 为系统已为每一个新声明的类重载了一个赋值运算符,它的作用是逐个复制类中的 数据 成员地址运算符&也不必重载,它能返回类对象在内存中的起始地址。

(9)应当使重载运算符的功能类似于该运算符作用于标准类型数据时候时所实现 的功 能。

例如,我们会去重载”+ “以实现对象的相加,而不会去重载”+“以实现对象相 减的功能,因为这样不符合我们对”+ “原来的认知。

(10)运算符重载函数可以是类的成员函数,也可以是类的友元函数,还可以是既非 类 的成员函数也不是友元函数的普通函数

4.10.3 双目运算符重载

|  |
| --- |
| //使用: L#R  operator#(L,R); //全局函数  L.operator#(R); //成员函数 |

oPQrafor+=

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;  class Complex  {  public :  Complex(float x=0, float y=0) :\_x(x),\_y(y){}  void dis()  {  cout<<"("<<\_x<<","<<\_y<<")"<<endl;  }  Complex& operator+=(const Complex &c)  {  this->\_x += c.\_x; this->\_y += c.\_y;  return \* this;  } | |
| private:  float  float  }; | \_x;  \_y; |
| int main()  {  // int a=10,b=20,c=30;  // a+=b;  // b+=c; | |

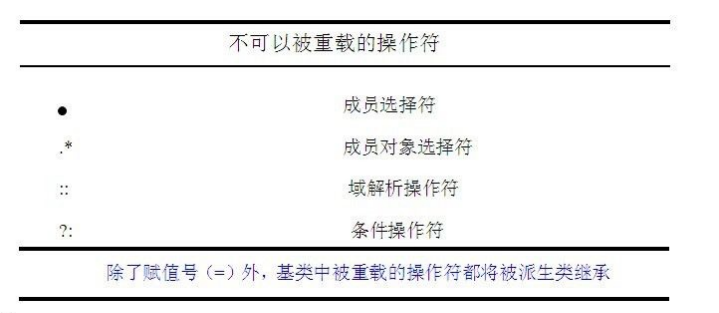
|  |
| --- |
| // cout<<"a = "<<a<<endl;  // cout<<"b = "<<b<<endl;  // cout<<"c = "<<c<<endl;  // Complex a1(10,0),b1(20,0), c1(30,0);  // 此时的+=重载函数返回 void  // a1 += b1;  // b1 += c1;  // a1.dis();  // b1.dis();  // c1.dis();  //----------------------------------------------------  // int a=10,b=20,c=30;  // a+=b+=c;  // cout<<"a = "<<a<<endl;  // cout<<"b = "<<b<<endl;  // cout<<"c = "<<c<<endl;  // Complex a1(10,0),b1(20,0), c1(30,0);  //此时重载函数+=返回的是 Complex // a1+=b1+=c1;  // a1.dis();  // b1.dis();  // c1.dis();  //----------------------------------------------------  int a = 10, b = 20,c = 30;  (a += b) += c;  cout<<"a = "<<a<<endl;  cout<<"b = "<<b<<endl;  cout<<"c = "<<c<<endl;  Complex a1(10,0),b1(20,0), c1(30,0);  // 此时重载函数+=返回的是 Complex &  // ⼀定要注意在连等式中 ,返回引⽤和返回对象的区别  (a1 += b1) += c1;  a1.dis();  b1.dis();  c1.dis();  return 0;  } |

oPQrafor-=

|  |
| --- |
| friend Complex& operator-=(Complex &c1, const Complex & c2)  {  } |

4.10.4 单目运算符重载

|  |  |
| --- | --- |
| operator#(M);  M.operator#() | //全局函数  //成员函数 |



oPQrafor++前加加

|  |
| --- |
| #include <iostream>  using namespace std;  class Complex  {  public :  Complex(float x=0, float y=0)  :\_x(x),\_y(y){}  void dis()  {  cout<<"("<<\_x<<","<<\_y<<")"<<endl;  }  friend Complex & operator++(Complex& c);  private :  \_x;  float  float  \_y;  };  Complex & operator++(Complex& c)  {  c.\_x++;  c.\_y++;  return c;  }  int main() |

87

|  |
| --- |
| {  int n = 10;  cout<<n<<endl;  //10  //11  //11  //13  cout<<++n<<endl;  cout<<n<<endl;  cout<<++++n<<endl;  cout<<n<<endl;  Complex c(10,10);  c.dis(); //10 10  Complex c2=++c;  c2.dis();  //11 11  //11 11  //13 13  //13 13  c.dis();  c2 = ++++c;  c2.dis();  c.dis();  return 0;  } |

oPQrafor++后加加

|  |
| --- |
| #include <iostream>  using namespace std;  class Complex  {  public :  Complex(float x=0, float y=0):\_x(x),\_y(y){}  void dis()  {  cout<<"("<<\_x<<","<<\_y<<")"<<endl;  }  #if 0  const Complex operator++(int)  {  Complex t = \*this; \_x++;  \_y++;  return t;  }  #endif  friend const Complex operator++(Complex &c,int);  private :  \_x;  float  float  \_y;  };  const Complex operator++(Complex &c,int)  {  Complex t(c.\_x,c.\_y); c.\_x++;  c.\_y++;  return t; |

88

|  |
| --- |
| }  int main()  {  int n = 10;  cout<<n<<endl;  //10  //10  //11  cout<<n++<<endl;  cout<<n<<endl;  // cout<<n++++<<endl; //13 后++表达式不能连⽤  cout<<n<<endl; //11  Complex c(10);  c.dis();  Complex c2 = c++;  c2.dis();  c.dis();  //c2 = c++++;  //c2.dis();  c.dis();  return 0;  } |

4.10.5 输入输出运算符重载

|  |
| --- |
| istream & operator>>(istream &,⾃定义类&);  ostream & operator<<(ostream &,⾃定义类&); |

通过友元来实现,避免修改 c++的标准库。

oPQrafor<<和oPQrafor>>

|  |
| --- |
| #include <iostream>  using namespace std;  class Complex {  public :  Complex(float x=0, float y=0)  :\_x(x),\_y(y){}  void dis() {  cout<<"("<<\_x<<","<<\_y<<")"<<endl;  }  friend ostream & operator<<(ostream &os, const Complex & c);  friend istream & operator>>(istream &is, Complex &c);  private:  \_x;  float  float  \_y;  };  operator<<(ostream &os, const Complex & c)  ostream & |

|  |
| --- |
| {  os<<"("<<c.\_x<<","<<c.\_y<<")";  return os;  }  istream & operator>>(istream &is, Complex &c)  {  is>>c.\_x>>c.\_y;  return is;  }  int main()  {  Complex c(2,3);  cout<<c<<endl;  cin>>c;  cout<<c<<endl;  return 0;  } |

4.10.6 友元还是成员

假设,我们有类 Sender 类和 Mail 类,实现发送邮件的功能。

sender<< mail;

sender 左操作数,决定了 operator<<为 Sender 的成员函数,而 mail 决 定了 operator<<要作 Mail 类的友员。

|  |
| --- |
| #include <iostream>  using namespace std;  class Mail;  class Sender  {  public :  Sender(string s):\_addr(s){}  Sender& operator<<(const Mail & mail); //成员  private:  string \_addr;  };  class Mail  {  public:  89 |

|  |
| --- |
| Mail(string \_t,string \_c ):\_title(\_t),\_content(\_c){}  friend Sender& Sender::operator<<(const Mail & mail);  private :  string \_title;  string \_content;  };  Sender& Sender::operator<<(const Mail & mail)  {  cout<<"Address:"<<\_addr<<endl;  cout<<"Title :"<<mail.\_title<<endl;  cout<<"Content:"<<mail.\_content<<endl;  return \*this;  }  int main()  {  Sender sender("danbing\_at@gmail.com");  Mail mail("note","meeting at 3:00 pm");  Mail mail2("tour","One night in beijing");  sender<<mail<<mail2;  return 0;  } |

|  |
| --- |
| 结论: 1,一个操作符的左右操作数不一定是相同类型的对象,这就涉及到将该操作符函  数定义为谁的友元,谁的成员问题。  2,一个操作符函数,被声明为哪个类的成员,取决于该函数的调用对象(通常是左 操作数) 。 3,一个操作符函数,被声明为哪个类的友员,取决于该函数的参数对象(通常是右 操作数)。 |

4.10.7 运算符重载提高

(1) 赋值运算符重载 (operator=)

用一个己有对象,给另外一个己有对象赋值。两个对象均己创建结束后,发

生的赋 值行为。

|  |
| --- |
| 类名  {  类名& operator=(const 类名& 源对象) 拷⻉体  } |

|  |
| --- |
| class A  {  A& operator=(const A& another)  {  //函数体  return \*this;  }  }; |

|  |
| --- |
| 规则  **1** 系统提供默认的赋值运算符重载,一经实现,不复存在。  **2** 系统提供的也是等位拷贝,也就浅拷贝,一个内存泄漏,重析构。  **3** 要实再深深的赋值,必须自定义。  **4** 自定义面临的问题有三个:  **1**,自赋值  **2**,内存泄漏  **3**,重析构。  **5** 返回引用,且不能用 **const** 修饰。其目的是实现连等式。 |

(2) 数组下标运算符 (operator[])

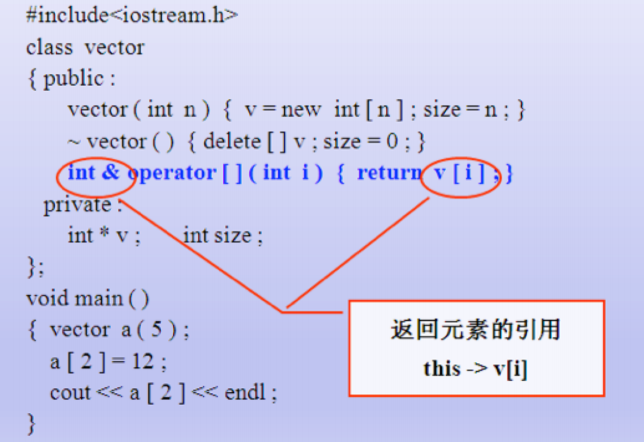
|  |
| --- |
| 类型 类 :: operator[] ( 类型 ) ； |

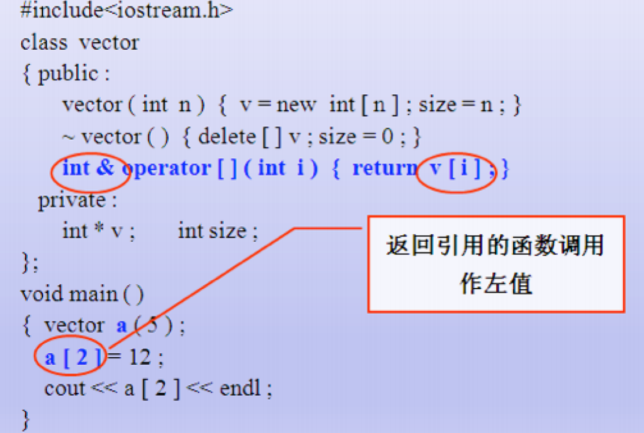
设 x 是类 X 的一个对象，则表达式

x [ y ]

可被解释为

x . operator [] ( y )





|  |
| --- |
| 练习：实现一个软组类,要示有<<,>>等重载 |

(3) 函数调用符号 (operator () )

把类对象像函数名一样使用。

仿函数(functor),就是使一个类的使用看上去象一个函数。其实现就是类

中实现一个operator(),这个类就有了类似函数的行为,就是一个仿函数类了。

|  |
| --- |
| class 类名 {  返值类型 operator()(参数类型) 函数体  } |

|  |
| --- |
| #include <iostream>  using namespace std;  class Sqr  {  public :  operator()(int i)  int  {  return i\*i;  }  double operator ()(double d)  {  return d\*d;  }  };  int main()  {  Sqr sqr;  int i = sqr(4);  double d = sqr(5.5);  cout<<i<<endl;  cout<<d<<endl;  return 0;  } |

(4) 不可重载**&&**和**||**操作符

|  |
| --- |
| #include <iostream>  using namespace std;  class Test  {  public :  Test(int i = 0)  {  this->i = i;  }  Test operator+ (const Test& obj)  {  cout<<"执⾏+号重载函数"<<endl;  Test ret;  ret.i = i + obj.i;  return ret;  }  bool operator&&(const Test& obj)  {  cout<<"执⾏&&重载函数"<<endl;  return i && obj.i;  }  private:  int i;  };  int main()  {  int a1 = 0;  int a2 = 1;  cout<<"注意： &&操作符的结合顺序是从左向右"<<endl;  if( a1 && (a1 + a2) )  {  cout<<"有⼀个是假，则不在执⾏下⼀个表达式的计算"<<endl;  }  Test t1(0);  Test t2(1);  if ( t1 && (t1 + t2) )  {  //t1 && t1.operator(t2)  // t1.operator&&( t1.operator+(t2) )  cout<<"两个函数都被执⾏了，⽽且是先执⾏了+"<<endl;  } |

|  |
| --- |
| return 0;  } |

|  |
| --- |
| C++如果重载&&或者|| 将无法实现短路规则 |

|  |
| --- |
| 练习 实现一个早符串类 |

|  |
| --- |
| 构道凶软要示  M」Sfri"9aj  M」Sfri"9a(“dddd”)j  M」Sfri"9b=aj  常用的橡作符  久久>>!===>久=[] |

(7)解引用与智能指针

常规意义上讲,new 或是 malloc 出来的堆上的空间,都需要手动 delete 和 free 的。但在其它高级语言中,只需申请无需释放的功能是存在的。

c++中也提供了这样的机制。我们先来探究一下实现原理。

常规应用

|  |
| --- |
| void foo()  {  A\*p = new A;  // do something  delete p;  } |

智能指针

|  |
| --- |
| #include <iostream>  #include <memory>  using namespace std;  class A |

|  |
| --- |
| {  public :  A() {  cout<<"A constructor"<<endl;  }  ~A() {  cout<<"A destructor"<<endl;  }  void dis() {  cout <<"class A's dis() " <<endl;  }  };  int main()  {  //使用智能指针 auto\_ptr  auto\_ptr<A> p (new A);  p->dis();  return 0;  } |

目足义智能指针

|  |
| --- |
| #include <iostream>  #include <memory>  using namespace std;  class A  {  public:  A()  {  cout<<"A constructor"<<endl;  }  ~A()  {  cout<<"A destructor"<<endl;  }  void dis()  {  cout<<"in class A's dis"<<endl;  }  };  class PMA  {  public :  PMA(A \*p) :\_p(p){}  96 |

|  |  |
| --- | --- |
| ~PMA()  {  delete  } | \_p; |
| A& operator\*()  {  return \*\_p;  }  A\* operator->()  {  return \_p;  }  private :  A \* \_p;  }; | |

**->**和**\*** 重载格式

|  |
| --- |
| 类名& operator\*() {  函数体  }  类名\* operator->() {  函数体  } |

作业



|  |
| --- |
| 作业1 设计TDaf&类 |

定义一个处理日期的类 TDate,它有 3 个私有数据成员:Month,Day,Year 和若干个 公 有成员函数,并实现如下要求:

1构造函数重载

2成员函数设置缺省参数

3可使用不同的构造函数来创建不同的对象

4定义一个友元函数来打印日期

|  |
| --- |
| 作业**2** 设计一个矩阵类 |

设计一个 3\*3 的矩阵类 class Matrix,通过一数组进行初始化。

要求如下:

1默认构造(初始化为 0),有参构造(数组作实参)

2重载+ / +=

3重载\* / \*=

4实现输出

|  |
| --- |
| class Matrix  {  public :  Matrix(void);  Matrix(int p[][3]);  private :  int data[3][3];  }; |

5. 继承和派生

在 C++中可重用性(software reusability)是通过继承(inheritance)这一机制来 实现的。 如果没有掌握继承性,就没有掌握类与对象的精华。

5.1 类和类之间的关系

has-A，uses-A 和 is-A

has-A 包含关系，用以描述一个类由多个“部件类”构成。实现has-A 关系用类成员表示，即一个类中的数据成员是另一种已经定义的类。

uses-A 一个类部分地使用另一个类。通过类之间成员函数的相互联系， 定义友员或对象参数传递实现。

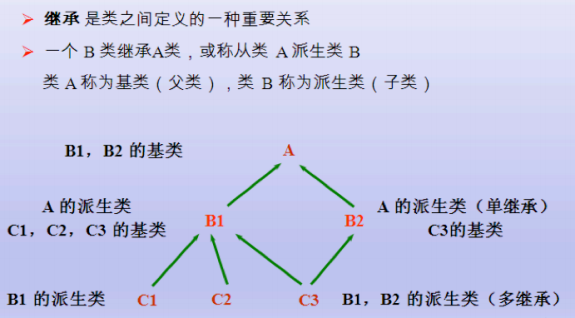
is-A 机制称为“继承”。关系具有传递性,不具有对称性。

5.2 什么是继承

|  |
| --- |
| #include <iostream>  using namespace std;  class Student  {  public :  void dis() {  cout<<name<<endl;  cout<<age<<endl;  }  string name;  int age;  }; |

重写：

|  |
| --- |
| class Student2  {  public :  void dis() {  cout<<name<<endl;  cout<<age<<endl;  cout<<sex<<endl;  cout<<score<<endl;  } |



|  |
| --- |
| private:  string name;  int age;  char sex;  float score;  }; |

继泳：

|  |
| --- |
| class Student2:public Student  {  public :  Student2(string n,int a,char s,float f)  {  name = n; age = a; sex = s; score = f;  }  void dis() {  Student::dis();  cout<<sex<<endl;  cout<<score<<endl;  }  char sex;  float score;  }; |

足义：

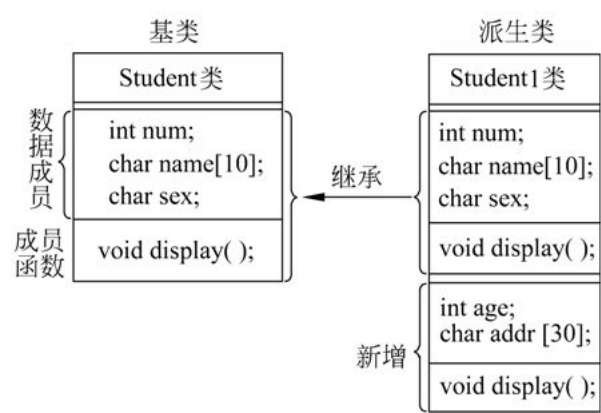
类的继承,是新的类从已有类那里得到已有的特性。或从已有类产生新类 的过程就是类的派生。原有的类称为基类或父类,产生的新类称为派生类或子类。

派生与继承,是同一种意义两种称谓。 isA 的关系。

101

派主类由组成

派生类中的成员,包含两大部分,一类是从基类继承过来的,一类是自己增加 的成员。从基类继承过过来的表现其共性,而新增的成员体现了其个性。



|  |
| --- |
| 几点说明: 1,全盘接收,除了构造器与析构器。 基类有可能会造成派生类的成员冗余,所以 说基类是需设计的。  2,派生类有了自己的个性,使派生类有了意义。 |

5.3 继承的方式

5.3.1 语法

|  |
| --- |
| class 派⽣类名:[继承⽅式] 基类名 {  派⽣类成员声明;  }; |

一个派生类可以同时有多个基类,这种情况称为多重继承,派生类只有一个 基类, 称为单继承。下面从单继承讲起。

5.3.2 protected 访问控制

protected 对于外界访问属性来说,等同于私有,但可以派生类中可见。

|  |
| --- |
| #include <iostream>  using namespace std;  class Base  {  public :  int pub;  protected :  int pro;  private :  int pri;  };  class Drive:public Base {  public :  void func() {  pub = 10;  pro = 100;  //pri = 1000;//error  }  };  int main(void)  {  Base b;  b.pub = 10;  //b.pro = 100; //error  //b.pri = 1000;//error  return 0;  } |

5.3.3 派生类成员的标识和访问

|  |  |  |  |
| --- | --- | --- | --- |
| public protected private | | | |
| 公有继承(public) | public | protected | 不可见 |
| 保护继承(protected) | protected | protected | 不可见 |
| 私有继承(private) | private | private | 不可见 |

P“b(ic公有继泳

当类的继承方式为公有继承时,基类的公有和保护成员的访问属性在派生 类中 不变,而基类的私有成员不可访问。 即基类的公有成员和保护成员被继承 到派生类 中仍作为派生类的公有成员和保护成员。派生类的其他成员可以直接 访问它们。无 论派生类的成员还是派生类的对象都无法访问基类的私有成员。

Privafc私有继泳

当类的继承方式为私有继承时,基类中的公有成员和保护成员都以私有成 员身 份出现在派生类中,而基类的私有成员在派生类中不可访问。 基类的公有 成员和保 护成员被继承后作为派生类的私有成员,派生类的其他成员可以直接 访问它们,但 是在类外部通过派生类的对象无法访问。无论是派生类的成员还 是通过派生类的对 象,都无法访问从基类继承的私有成员。通过多次私有继承 后,对于基类的成员都 会成为不可访问。因此私有继承比较少用。

Profccfcd保护继泳

保护继承中,基类的公有成员和私有成员都以保护成员的身份出现在派生 类 中,而基类的私有成员不可访问。 派生类的其他成员可以直接访问从基类继 承来的 公有和保护成员,但是类外部通过派生类的对象无法访问它们,无论派生 类的成员 还是派生类的对象,都无法访问基类的私有成员。

**private**成员在子类中依然存在，但是却无法访问到。不论何种方式继承

基类，派生类都不能直接使用基类的私有成员 。



如何恰当由便用P“b(ic,Profccfcd和Privafc为成员声明访问级别?

1、需要被外界访问的成员直接设置为public

2、只能在当前类中访问的成员设置为private

3、只能在当前类和子类中访问的成员设置为protected，protected成 员的访问权限介于public和private之间。

|  |
| --- |
| 练习：分析下列代码的访问权限 |

|  |
| --- |
| #include <iostream>  using namespace std;  class A  {  private :  int a;  protected :  int b;  public :  int c;  A()  {  a = 0;  b = 0;  c = 0;  }  void set(int a, int b, int c)  {  this->a = a;  this->b = b;  this->c = c;  }  };  class B : public A  {  public :  void print()  {  //能否访问???  cout<<"a = "<<a;  cout<<"b = "<<b;  cout<<"c = "<<endl;  //能否访问???  //能否访问???  }  };  class C : protected A  {  public :  void print()  {  //能否访问???  cout<<"a = "<<a;  cout<<"b = "<<b;  cout<<"c = "<<endl;  //能否访问???  //能否访问???  }  };  class D : private A  {  public :  void print() |

|  |
| --- |
| {  bb.print();  cc.print();  dd.print();  cout<<"a = "<<a; //能否访问???  cout<<"b = "<<b<<endl; //能否访问???  cout<<"c = "<<c<<endl; //能否访问???  }  };  main(void)  int  {  A aa;  B bb;  C cc;  D dd;  aa.c = 100;  //能否访问??? //能否访问??? //能否访问??? //能否访问???  bb.c = 100;  cc.c = 100;  dd.c = 100;  aa.set(1, 2, 3); //能否访问???  bb.set(10, 20, 30); //能否访问???  cc.set(40, 50, 60); //能否访问???  dd.set(70, 80, 90); //能否访问???  //能否访问??? //能否访问??? //能否访问???  return 0;  } |

5.4 继承中的构造和析构

5.4.1 类型兼容性原则

类型兼容规则是指在需要基类对象的任何地方，都可以使用公有派生类 的对象来替代。通过公有继承，派生类得到了基类中除构造函数、析构函数之 外的所有成员。这样，公有派生类实际就具备了基类的所有功能，凡是基类能 解决的问题，公有派生类都可以解决。

类型兼容规则中所指的替代包括以下情况：

|  |
| --- |
| 子类对象可以当作父类对象使用  子类对象可以直接赋值给父类对象 |

|  |
| --- |
| 子类对象可以直接初始化父类对象  父类指针可以直接指向子类对象  父类引用可以直接引用子类对象 |

在替代之后，派生类对象就可以作为基类的对象使用，但是只能使用从基类继 承的成员。

子类就是特殊的父类 (base \*p = &child;)

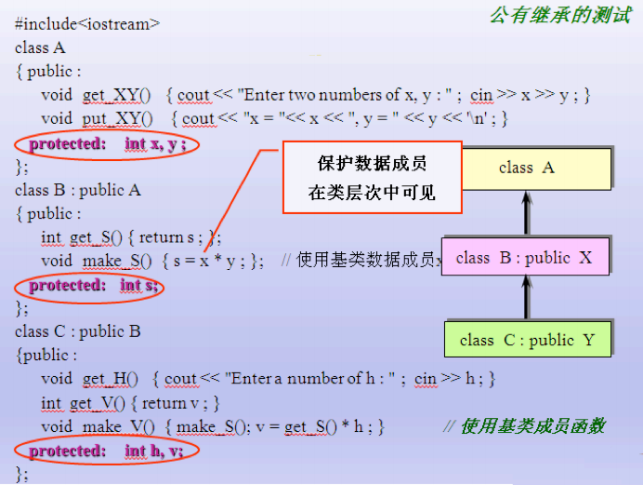


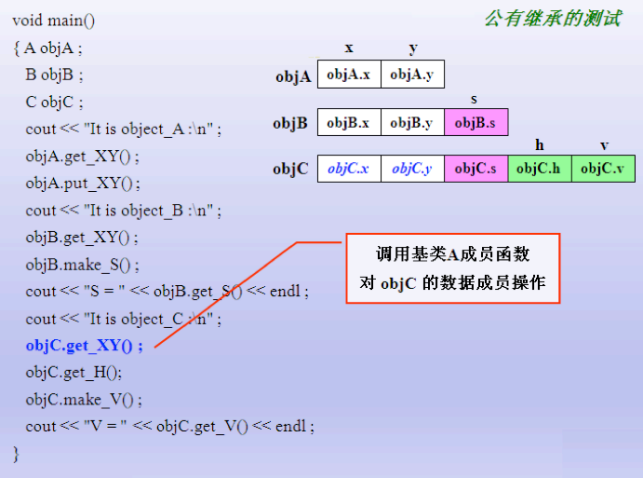
|  |
| --- |
| #include <iostream>  using namespace std;  class Parent  {  public :  void printP()  {  cout << "parent...." << endl;  }  };  class Child : public Parent  {  public :  void printC()  {  cout << "child..." << endl;  }  };  void print01(Parent \*p)  {  p->printP();  }  void print02(Parent &p)  {  p.printP();  }  int main()  {  Child c1;  c1.printC();  Parent \*p = NULL;  //可以⽤⽗类指针 指向 ⼦类对象  p = &c1;  p->printP(); //执⾏⽗类的函数  107 |

|  |  |  |  |
| --- | --- | --- | --- |
| Child  Parent | c2;  p2; | | |
| print01(&p2);  print01(&c2);  print02(p2);  print02(c2); | | | //⽗类指针指向⼦类对象  //⽗类引⽤指向⼦类对象 |
| //第⼆层含义  Child c3;  Parent p3 =  return 0;  } | | ⽤⼦类初始化⽗类对象  c3; | |

5.4.2 继承中的对象模型

类在C++编译器的内部可以理解为结构体,子类是由父类成员叠加子类新成 员得到的.





问毁：如何和始化父类六员? 父类与了类的构浩召新有什么关系?

|  |
| --- |
| 在子类对象构造时，需要调用父类构造函数对其继承得来的成员进行初 始化.  在子类对象析构时，需要调用父类析构函数对其继承得来的成员进行清 理. |

|  |
| --- |
| #include <iostream>  using namespace std;  class Parent  {  public :  Parent(const char\* s)  {  this->s = s;  cout<<"Parent()"<<" "<<s<<endl;  }  109 |

|  |
| --- |
| ~Parent()  {  cout<<"~Parent()"<<endl;  }  private :  const char \*s;  };  class Child : public Parent  {  public :  Child(int a) : Parent("Parameter from Child!")  {  cout<<"Child()"<<endl;  this->a = a;  }  Child(int a, const char \*s) : Parent(s)  {  cout<<"Child()"<<endl;  this->a = a;  }  ~Child()  {  cout<<"~Child()"<<endl;  }  private :  int a;  };  void run()  {  //Child child(10);  Child child(10, "Parameter form child...");  }  int main(int argc, char \*argv[])  {  run();  return 0;  } |

5.4.3 继承中构造析构调用原则

|  |
| --- |
| 1、子类对象在创建时会首先调用父类的构造函数  2、父类构造函数执行结束后，执行子类的构造函数  3、当父类的构造函数有参数时，需要在子类的初始化列表中显示调用  110 |

|  |
| --- |
| 4、析构函数调用的先后顺序与构造函数相反 |

5.4.4 继承和组合并存，构造和析构原则

|  |
| --- |
| 先构造父类，再构造成员变量、最后构造自己  先析构自己，在析构成员变量、最后析构父类 |

|  |
| --- |
| #include <iostream>  using namespace std;  class Object  {  public :  Object(const char\* s)  {  cout<<"Object()"<<" "<<s<<endl;  }  ~Object()  {  cout<<"~Object()"<<endl;  }  };  class Parent : public Object  {  public :  Parent(const char\* s) : Object(s)  {  cout<<"Parent()"<<" "<<s<<endl;  }  ~Parent()  {  cout<<"~Parent()"<<endl;  }  };  class Child : public Parent  {  public :  Child() : o2("o2"), o1("o1"), Parent("Parameter from Child!")  {  cout<<"Child()"<<endl;  }  ~Child()  {  cout<<"~Child()"<<endl;  } |

|  |
| --- |
| private:  Object o1;  Object o2;  };  void run()  {  Child child;  }  int main(int argc, char \*argv[])  {  run();  return 0;  } |

结果

|  |
| --- |
| Object() o1  Object() o2  Child()  ~Child()  ~Object()  ~Object()  ~Parent()  ~Object() |

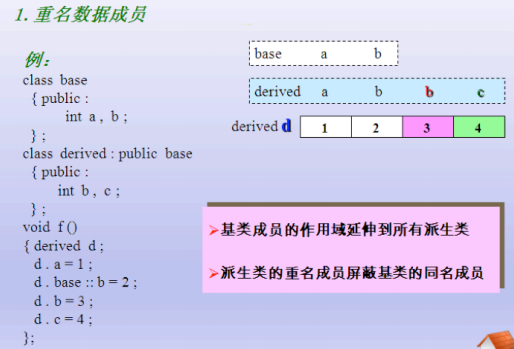
5.4.5 继承中同名成员变量处理方法

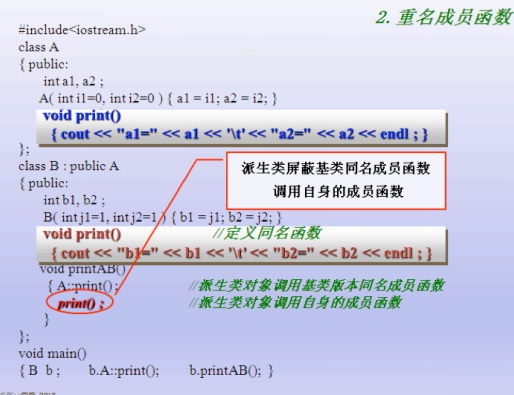
1、当子类成员变量与父类成员变量同名时

2、子类依然从父类继承同名成员

3、在子类中通过作用域分辨符::进行同名成员区分(在派生类中使用基 类的同名成员，显式地使用类名限定符)

4、同名成员存储在内存中的不同位置





同名成员变量和成员函数通过作用域分辨符进行区分



5.4.6 派生类中的static关键字

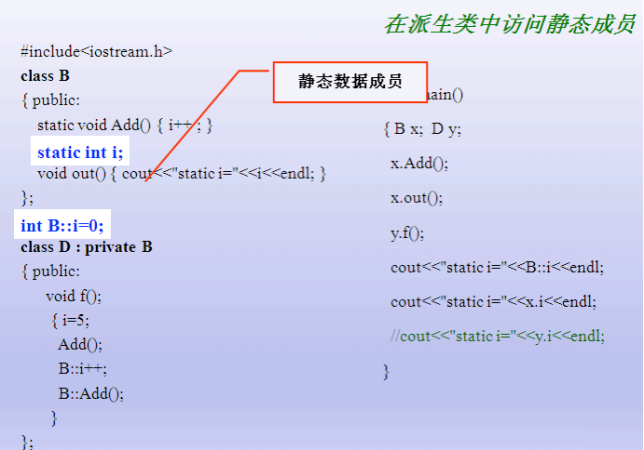
¬ 基类定义的静态成员，将被所有派生类共享

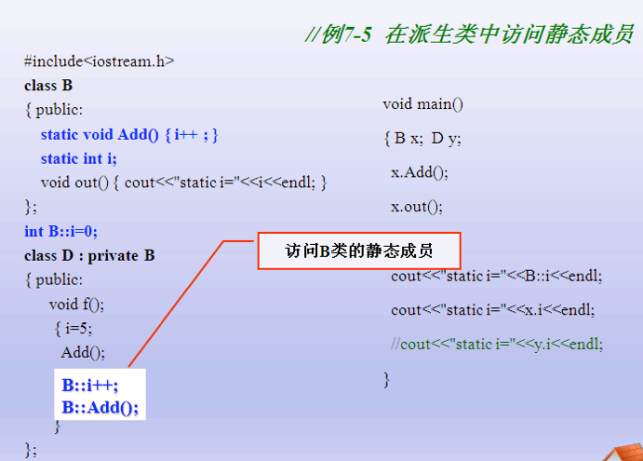
¬ 根据静态成员自身的访问特性和派生类的继承方式，在类层次体系中具 有不同的访问性质 (遵守派生类的访问控制)

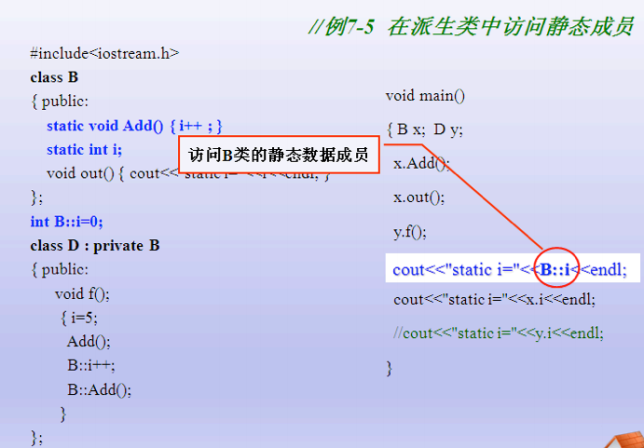
¬ 派生类中访问静态成员，用以下形式显式说明： 类名 :: 成员

或通过对象访问

对象名 . 成员







static函数也遵守3个访问原则

static易犯错误(不但要初始化，更重要的显示的告诉编译器分配内存)

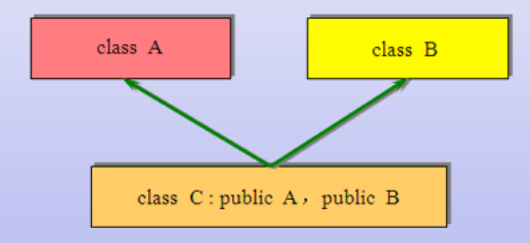
5.5 多继承

俗话讲的,鱼与熊掌不可兼得,而在计算机就可以实现,生成一种新的对象, 叫熊掌鱼,多继承自鱼和熊掌即可。还比如生活中,“兼”。

5.5.1 语法

|  |
| --- |
| 派⽣类名::派⽣类名(参数总表):基类名 1(参数表 1),基类名(参数名 2).... 基类名 n(参数名 n),内嵌⼦对象 1(参数表 1),  内嵌⼦对象 2(参数表 2).... 内嵌⼦对象 n(参数表 n) {  派⽣类新增成员的初始化语句;  } |

一个类有多个直接基类的继承关系称为多继承



5.5.2 沙发床实现



|  |
| --- |
| //bed.h  #ifndef BED\_H  #define BED\_H  //床类  class Bed  {  public:  Bed();  ~Bed();  void sleep();  };  #endif // BED\_H |

|  |
| --- |
| //bed.cpp  #include "bed.h"  #include <iostream>  using namespace std;  Bed::Bed()  { |

|  |
| --- |
| }  Bed::~Bed()  {  }  void Bed::sleep()  {  cout<<"take a good sleep"<<endl;  } |

|  |
| --- |
| //sofa.h  #ifndef SOFA\_H  #define SOFA\_H  //沙发类  class Sofa {  public:  Sofa();  ~Sofa();  void sit();  };  #endif // SOFA\_H |

|  |
| --- |
| //sofa.cpp  #include "sofa.h"  #include <iostream>  using namespace std;  Sofa::Sofa()  {  }  Sofa::~Sofa()  {  }  void Sofa::sit()  {  cout<<"take a rest"<<endl;  } |

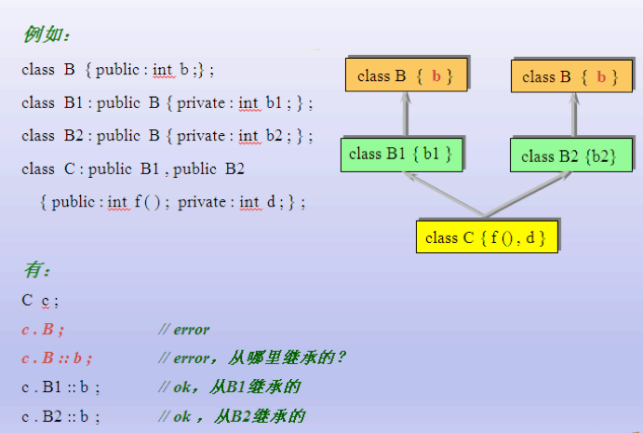
|  |
| --- |
| //main.cpp  #include <iostream>  #include "sofa.h"  #include "bed.h"  #include "sofabed.h"  using namespace std;  int main()  {  Sofa s;  118 |

|  |
| --- |
| s.sit();  Bed b;  b.sleep();  SofaBed sb;  sb.sit();  sb.sleep();  return 0;  } |

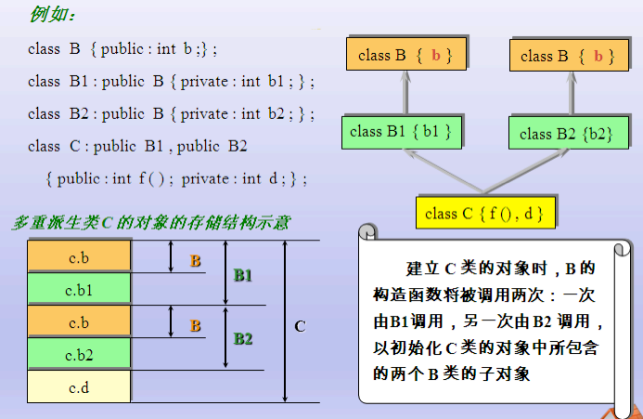
5.6虚继承

如果一个派生类从多个基类派生，而这些基类又有一个共同的基类，则 在对该基类中声明的名字进行访问时，可能产生二义性

5.6.1 多继承中二义性问题



分析：



5.6.2 虚继承virtual

¬ 如果一个派生类从多个基类派生，而这些基类又有一个共同

的基类，则在对该基类中声明的名字进行访问时，可能产生二义性

¬ 如果在多条继承路径上有一个公共的基类，那么在继承路径的某处 汇合点，这个公共基类就会在派生类的对象中产生多个基类子对象

¬ 要使这个公共基类在派生类中只产生一个子对象，必须对这个基类 声明为虚继承，使这个基类成为虚基类。

¬ 虚继承声明使用关键字 virtual





6. 多态

6.1 什么是多态

6.1.1 浅析多态的意义

如果有几个上似而不完全相同的对象,有时人们要求在向它们发出同一个消息 时, 它们的反应各不相同,分别执行不同的操作。这种情况就是多态现象。

例如,甲乙丙 **3** 个班都是高二年级,他们有基本相同的属性和行为,在同时听到 上课铃声的时候,他们会分别走向 **3** 个不同的教室,而不会走向同一个教室。

同样,如果有两支军队,当在战场上听到同种号声,由于事先约定不同,**A** 军队 可能实施进攻,而 **B** 军队可能准备 **kalalok**。

C++中所谓的多态(polymorphism)是指,由继承而产生的相关的不同的类,其对象 对同一消息会作出不同的响应。

多态性是面向对象程序设计的一个重要特征,能增加程序的灵活性。可以减轻 系统 升级,维护,调试的工作量和复杂度.

6.1.2 赋值兼容(多态实现的前提)

赋值兼容规则是指在需要基类对象的任何地方都可以使用公有派生类的 对象来替代。

赋值兼容是一种默认行为,不需要任何的显示的转化步骤。

赋值兼容规则中所指的替代包括以下的情况:

派生类的对象可以赋值给基类对象。

派生类的对象可以初始化基类的引用。

派生类对象的地址可以赋给指向基类的指针。

在替代之后,派生类对象就可以作为基类的对象使用,但只能使用从基类继 承的成员。

|  |
| --- |
| #include <iostream>  using namespace std;  class Parent  {  public: |

|  |
| --- |
| Parent(int a)  {  this->a = a;  cout<<"Parent a"<<a<<endl;  }  void print() //⼦类的和⽗类的函数名字⼀样  {  cout<<"Parent 打印 a:"<<a<<endl;  }  private :  int a ;  };  class Child : public Parent  {  public :  Child(int b) : Parent(10)  {  this->b = b;  cout<<"Child b"<<b<<endl;  }  void print()  {  cout<<"Child 打印 b:"<<b<<endl;  }  private :  int b;  };  void howToPrint(Parent \*base)  {  base->print(); //⼀种调⽤语句 有多种表现形态 ...  }  void howToPrint2(Parent &base)  {  base.print();  }  int main(void)  {  Parent  \*base = NULL; p1(20); c1(30);  Parent  Child  base = &p1;  base->print(); //执⾏⽗类的打印函数  base = &c1;  123 |

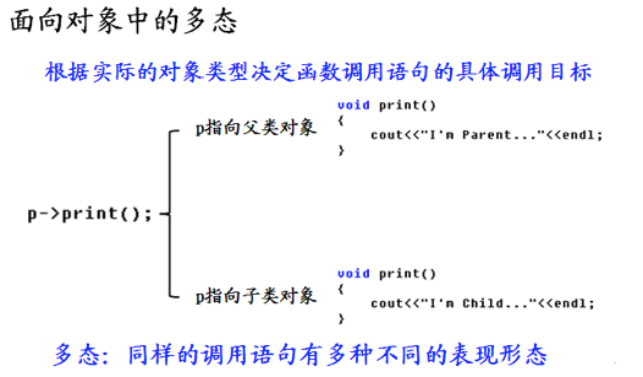
|  |
| --- |
| /\*  编译器认为最安全的做法是编译到⽗类的print函数，  因为⽗类和⼦类肯定都有相同的print函数。  \*/  base->print(); //执⾏谁的函数 ? //貌似我们希望之星Child的print函数  Parent &base2 = p1;  base2.print(); //执⾏⽗类的打印函数  Parent &base3 = c1;  base3.print(); //执⾏谁的函数?  //函数调⽤  howToPrint(&p1);  howToPrint(&c1);  howToPrint2(p1);  howToPrint2(c1);  return 0;  } |

6.1.3 面向对象新需求

编译器的做法不是我们期望的；

根据实际的对象类型来判断重写函数的调用；

如果父类指针指向的是父类对象则调用父类中定义的函数； 如果父类指针指向的是子类对象则调用子类中定义的重写函数；



6.1.4 解决方案

¬ C++中通过virtual关键字对多态进行支持

¬ 使用virtual声明的函数被重写后即可展现多态特性

|  |
| --- |
| #include <iostream>  using namespace std;  class HeroFighter  {  public :  virtual int ackPower()  {  return 10;  }  };  class AdvHeroFighter : public HeroFighter  {  public: |

|  |
| --- |
| virtual int ackPower()  {  return HeroFighter::ackPower()\*2;  }  };  class enemyFighter  {  public :  int destoryPower()  {  return 15;  }  };  //如果把这个结构放在动态库⾥⾯  void objPK(HeroFighter \*hf, enemyFighter \*enemyF)  {  if (hf->ackPower() >enemyF->destoryPower())  {  printf("英雄打败敌⼈。。。胜利\n");  }  else  {  printf("英雄。。。牺牲\n");  }  }  void main()  {  HeroFighter hf;  enemyFighter ef;  objPK(&hf, &ef);  AdvHeroFighter advhf;  objPK(&advhf, &ef);  return 0;  } |

6.1.5 多态工程的意义

封装

突破了C语言函数的概念。

继承

代码复用， 复用原来写好的代码。

多态

多态可以使用未来， 80年代写了一个框架， 90人写的代码。 多态是软件行业追寻的一个目标。

6.1.6 多态成立的条件

|  |
| --- |
| 1 要有继承  2 要有虚函数重写  3 要有父类指针(父类引用)指向子类对象 |

多态是设计模式的基础，多态是框架的基础



6.1.6 静态联编和动态联编

1、联编是指一个程序模块、代码之间互相关联的过程。

2、静态联编(sta5cbinding)，是程序的匹配、连接在编译阶段实现，也称为 早期匹配。 重载函数使用静态联编。

3、动态联编是指程序联编推迟到运行时进行，所以又称为晚期联编(迟绑 定)。 switch 语句和 if 语句是动态联编的例子。

|  |
| --- |
| 1 、C++与C相同，是静态编译型语言  2 、在编译时，编译器自动根据指针的类型判断指向的是一个什么样的对象； 所以编译器认为父类指针指向的是父类对象。  3、由于程序没有运行，所以不可能知道父类指针指向的具体是父类对象还是 子类对象， 从程序安全的角度，编译器假设父类指针只指向父类对象，因此编  译的结果为调用父类的成员函数。这种特性就是静态联编。  4、多态的发生是动态联编，实在程序执行的时候判断具体父类指针应该调用  的方法。 |

6.2 虚析构函数

¬ 构造函数不能是虚函数。建立一个派生类对象时，必须从类层次的根开 始，沿着继承路径逐个调用基类的构造函数。

¬ 析构函数可以是虚的。虚析构函数用于指引 delete 运算符正确析构动态 对象 。

|  |
| --- |
| #include <iostream>  using namespace std;  class A  {  public:  A()  {  p = new char[20];  strcpy(p, "obja");  printf("A()\n");  }  virtual ~A()  {  delete [] p;  printf("~A()\n");  }  private :  char \*p;  };  class B : public A  {  public:  B()  {  p = new char[20];  strcpy(p, "objb");  printf("B()\n");  }  ~B()  {  delete [] p;  printf("~B()\n");  }  private :  char \*p;  };  class C : public B  {  public: |

|  |
| --- |
| C()  {  p = new char[20];  strcpy(p, "objc");  printf("C()\n");  }  ~C()  {  delete [] p;  printf("~C()\n");  }  private :  char \*p;  };  //通过⽗类指针 把 所有的⼦类对象的析构函数 都执⾏⼀遍  //通过⽗类指针 释放所有的⼦类资源  void howtodelete(A \*base)  {  delete base;  }  int main()  {  C \*myC = new C;  //delete myC; //直接通过⼦类对象释放资源 不需要写virtual  howtodelete(myC);//通过⽗类的指针调⽤释放⼦类的资源  return 0;  } |

6.3 重载、重写、重定义

|  |
| --- |
| 重载(添加) ：  a 相同的范围(在同一个类中)  b 函数名字相同  c 参数不同  d virtual关键字可有可无 |

|  |
| --- |
| 重写(覆盖) 是指派生类函数覆盖基类函数，特征是：  a 不同的范围，分别位于基类和派生类中  129 |



|  |
| --- |
| b 函数的名字相同  c 参数相同  d 基类函数必须有virtual关键字 |

|  |
| --- |
| 重定义(隐藏) 是指派生类的函数屏蔽了与其同名的基类函数，规则如下：  a 如果派生类的函数和基类的函数同名，但是参数不同，此时，不管有无 virtual，基类的函数被隐藏。  b 如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没 有vitual关键字，此时，基类的函数被隐藏。 |

6.4 多态的实现原理

6.4.1 虚函数表和vptr指针

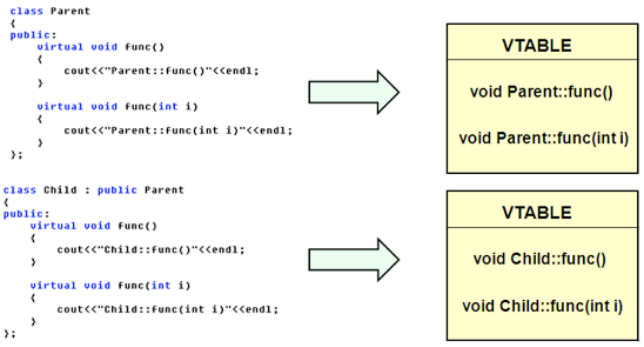
当类中声明虚函数时，编译器会在类中生成一个虚函数表；

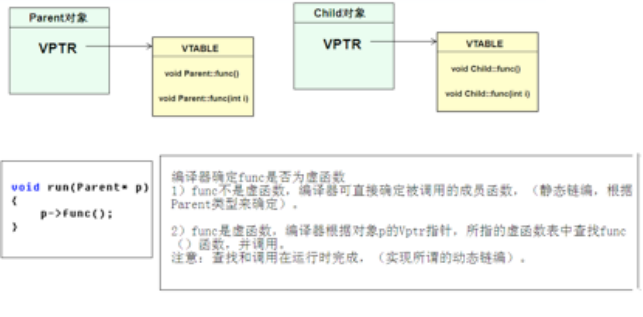
虚函数表是一个存储类成员函数指针的数据结构；

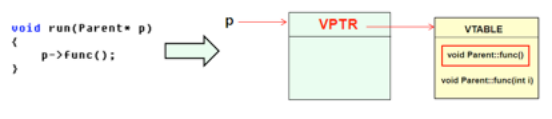
虚函数表是由编译器自动生成与维护的；

virtual成员函数会被编译器放入虚函数表中；

存在虚函数时，每个对象中都有一个指向虚函数表的指针(vptr指针)。







|  |
| --- |
| 说明：  1. 通过虚函数表指针VPTR调用重写函数是在程序运行时进行的，因此需 要通过寻址操作才能确定真正应该调用的函数。 而普通成员函数是在编译时就 确定了调用的函数。在效率上， 虚函数的效率要低很多。  2.出于效率考虑，没有必要将所有成员函数都声明为虚函数.  3.C++编译器，执行run 函数，不需要区分是子类对象还是父类对象,而是 直接通过p的VPTR指针所指向的对象函数执行即可。 |

6.4.2 证明vptr指针的存在

|  |
| --- |
| #include <iostream>  using namespace std;  class Parent1  {  public :  Parent1(int a=0)  {  this->a = a;  }  131 |

|  |
| --- |
| void print()  {  cout<<"我是爹"<<endl;  }  private :  int a;  };  class Parent2  {  public :  Parent2(int a=0)  {  this->a = a;  }  virtual void print()  {  cout<<"我也是爹"<<endl;  }  private :  int a;  };  int main()  {  cout <<"sizeof(Parent1): " << sizeof(Parent1) <<endl;  cout <<"sizeof(Parent2): " << sizeof(Parent2) <<endl;  return 0;  } |

6.4.3 构造函数中能否调用虚函数，实现多态？

对象在创建的时,由编译器对VPTR指针进行初始化

只有当对象的构造完全结束后VPTR的指向才最终确定

父类对象的VPTR指向父类虚函数表

子类对象的VPTR指向子类虚函数表

|  |
| --- |
| #include <iostream>  using namespace std;  //构造函数中调⽤虚函数能发⽣多态吗?  class Parent  {  public :  Parent(int a=0)  {  this->a = a;  print();  132 |

|  |
| --- |
| }  virtual void print()  {  cout<<"我是爹"<<endl;  }  private:  int a;  };  class Child : public Parent  {  public :  Child(int a = 0, int b=0):Parent(a)  {  this->b = b;  print();  }  virtual void print()  {  cout<<"我是⼉⼦"<<endl;  }  private :  int b;  };  void HowToPlay(Parent \*base)  {  base->print(); //有多态发⽣  }  int main(void)  {  Child c1; //定义⼀个⼦类对象 ,在这个过程中,在⽗类构造函数中调⽤虚函数print 能发 ⽣多态吗?  HowToPlay(&c1);  return 0;  } |

6.4.4 父类指针和子类指针的步长

|  |
| --- |
| #include <iostream>  using namespace std;  class Parent  {  public :  Parent(int a=0)  {  this->a = a;  }  virtual void print()  {  cout<<"我是爹"<<endl;  }  private:  int a;  };  class Child : public Parent  {  public :  Child(int b = 0):Parent(0)  {  this->b = b;  }  virtual void print()  {  cout<<"我是儿子"<<endl;  }  private :  int b;  };  int main()  {  Parent \*pP = NULL;  Child \*pC = NULL;  Child array[] = {Child(1), Child(2), Child(3)};  pP = array;  pC = array;  pP->print();//发生多态  pC->print();  pP++;  pC++;  pP->print();//发生多态  pC->print();  return 0;  } |

6.5 有关多态的理解

多态的实现效果

多态：同样的调用语句有多种不同的表现形态；

多态实现的三个系件

有继承、有virtual重写、有父类指针(引用)指向子类对象。

多态的**C++** 实现

virtual关键字，告诉编译器这个函数要支持多态；不是根据指针类型判断 如何调用；而是要根据指针所指向的实际对象类型来判断如何调用

多态的理论基础

动态联编PK静态联编。根据实际的对象类型来判断重写函数的调用。

多态的重要管义

设计模式的基础 是框架的基石。

多态原理琢究

虚函数表和vptr指针。

6.6 纯虚函数和抽象类

6.6.1 基本概念

纯虚函数是一个在基类中说明的虚函数，在基类中没有定义，要求任何 派生类都定义自己的版本

纯虚函数为个派生类提供一个公共界面(接口的封装和设计、软件的模 块功能划分)

纯虚凶软由语法：

|  |
| --- |
| virtual 类型 函数名(参数表) = 0; |

一个具有纯虚函数的基类称为抽象类。

6.6.2 纯虚函数和抽象类

|  |
| --- |
| using namespace std;  ////⾯向抽象类编程(⾯向⼀套预先定义好的接⼝编程)  class Figure //抽象类  {  public:  //阅读⼀个统⼀的界⾯(接⼝),让⼦类使⽤ ,让⼦类必须去实现  virtual void getArea() = 0 ; //纯虚函数  };  class Circle : public Figure  {  public :  Circle(int a, int b)  {  this->a = a;  this->b = b;  }  virtual void getArea()  {  cout<<"圆形的⾯积: "<<3.14\*a\*a<<endl;;  }  private:  int a;  int b;  };  class Tri : public Figure  {  public :  Tri(int a, int b)  {  this->a = a;  this->b = b;  }  virtual void getArea()  {  cout<<"三⾓形的⾯积: "<<a\*b/2<<endl;;  }  private:  int a;  int b;  };  class Square : public Figure  {  public :  Square(int a, int b)  {  this->a = a;  this->b = b;  136 |

|  |
| --- |
| }  virtual void getArea()  {  cout<<"四边形的⾯积: "<<a\*b<<endl;;  }  private:  int a;  int b;  };  void area\_func(Figure \*base)  {  base->getArea(); //会发⽣多态  }  int main()  {  //Figure f; //抽象类不能被实例化  Figure \*base = NULL; //抽象类不能被实例化  Circle c1(10, 20);  Tri t1(20, 30);  Square s1(50, 60);  //⾯向抽象类编程(⾯向⼀套预先定义好的接⼝编程)  area\_func(&c1);  area\_func(&t1);  area\_func(&s1);  return 0;  } |

|  |
| --- |
| 1,含有纯虚函数的类,称为抽象基类,不可实列化。 即不能创建对象,存在 的意义  就是被继承,提供族类的公共接口。  2,纯虚函数只有声明,没有实现,被“初始化”为 0。  3,如果一个类中声明了纯虚函数,而在派生类中没有对该函数定义,则该虚函数在 派生类中仍然为纯虚函数,派生类仍然为纯虚基类。 |

6.6.3 抽象类在多继承中的应用

绝大多数面向对象语言都不支持多继承,绝大多数面向对象语言都支持接 口的概念

C++中没有接口的概念,C++中可以使用纯虚函数实现接口

接口类中只有函数原型定义，没有任何数据的定义.

|  |  |
| --- | --- |
| class Interface  {  public :  virtual  void  void  void  virtual  virtual  }; | func1() = 0;  func2(int i) = 0;  func3(int i) = 0; |

|  |
| --- |
| #include <iostream>  using namespace std;  /\*  C++中没有接⼝的概念  C++中可以使⽤纯虚函数实现接⼝  接⼝类中只有函数原型定义，没有任何数据的定义。  \*/  class Interface1  {  public :  virtual void print() = 0;  virtual int add(int a, int b) = 0;  };  class Interface2  {  public :  virtual void print() = 0;  virtual int add(int a, int b) = 0;  virtual int sub(int a, int b) = 0;  };  class parent  {  public :  int a;  };  class Child : public parent, public Interface1, public Interface2  {  public: |

|  |
| --- |
| void print()  {  cout<<"Child::print"<<endl;  }  int add(int a, int b)  {  return a + b;  }  int sub(int a, int b)  {  return a - b;  }  };  int main()  {  Child c;  c.print();  cout<<c.add(3, 5)<<endl;  cout<<c.sub(4, 6)<<endl;  Interface1\* i1 = &c;  Interface2\* i2 = &c;  cout<<i1->add(7, 8)<<endl;  cout<<i2->add(7, 8)<<endl;  return 0;  } |

6.7 ⾯向抽象类编程案例

案例1动物园里欢乐多

|  |
| --- |
| //animal.h  #ifndef ANIMAL\_H  #define ANIMAL\_H  class Animal  {  public:  Animal();  virtual ~Animal();  virtual void voice() = 0;  };  #endif // ANIMAL\_H |

|  |
| --- |
| //animal.cpp  #include <iostream>  #include "animal.h"  using namespace std;  Animal::Animal()  {  cout<<"Animal::Animal()"<<endl;  }  Animal::~Animal()  {  cout<<"Animal::~Animal()"<<endl;  } |

|  |
| --- |
| //dog.h  #ifndef DOG\_H  #define DOG\_H  #include "animal.h"  class Dog:public Animal  {  public:  Dog();  ~Dog();  virtual void voice();  };  #endif // DOG\_H |

|  |
| --- |
| //dog.cpp  #include "dog.h"  #include <iostream> using namespace std;  Dog::Dog()  {  cout<<"Dog::Dog()"<<endl;  }  Dog::~Dog()  {  cout<<"Dog::~Dog()"<<endl;  }  void Dog:: voice()  {  cout<<"wang wang"<<endl;  } |

|  |
| --- |
| //cat.h  #ifndef CAT\_H  #define CAT\_H  #include "animal.h"  class Cat:public Animal  {  public:  Cat();  140 |

|  |
| --- |
| ~Cat();  virtual void voice();  };  #endif // CAT\_H |

|  |
| --- |
| //cat.cpp  #include "cat.h"  #include <iostream>  using namespace std;  Cat::Cat() {  cout<<"Cat::Cat()"<<endl;  }  Cat::~Cat()  {  cout<<"Cat::~Cat()"<<endl;  }  void Cat::voice()  {  cout<<"miao miao "<<endl;  } |

|  |
| --- |
| //main.cpp  int main()  {  // Animal ani; 抽象基类 ,不能实例化。  Animal \* pa = new Dog;  pa->voice();  delete pa;  cout<<"---------------"<<endl;  pa = new Cat;  pa->voice();  delete pa;  return 0;  } |

案例z员工待遇

|  |
| --- |
| 编写一个C++程序, 计算程序员( programmer )工资  1 要求能计算出初级程序员( junior\_programmer ) 中级程序员 ( mid\_programmer )高级程序员( adv\_programmer)的工资  2 要求利用抽象类统一界面,方便程序的扩展, 比如:新增, 计算 架构师  (architect ) 的工资 |

案例3电脑组装案例,面向抽家层编程

组装电脑有3个抽象类，抽象的cpu，抽象的显卡，抽象的内存。 现在要求组装两台电脑，一台是Intel的CPU，Intel的显卡和Intel的内存。

另一台是Intel的CPU、Nvidia的显卡和Kingston的内存条。针对抽象层编程 练习完成此题。

案例+几何图不多态练习

设计一个基类Shape包含成员函数Show(), 将Show()声明为纯虚函数。 Shape类公有派生矩形类Square(正方形)和圆类Circle(圆形)，

问题1： 分别定义Show()实现其主要集合元素的显示。使用抽象类Shape 类型的指针，当它指向某个派生类的对象时，就可以通过访问该对象的虚函数 成员Show()。

问题2：用ShowArea()分别显示各种图形的面积.最后还要显示所有图形 的各个面积。要求积累指针数组，数组的每个元素指向一个派生类对象。

案例5企业员工信息管理系统

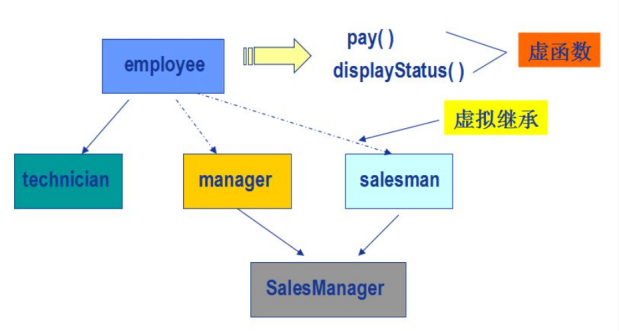
一个小型公司的人员信息管理系统

某小型公司,主要有四类人员:经理、技术人员、销售经理和推销员。现在, 需要存储这些人员的姓名、编号、级别、当月薪水.计算月薪总额并显示全部信 息。

人员编号基数为 1000,每输入一个人员信息编号顺序加 1。

程序要有对所有人员提升级别的功能。本例中为简单起见,所有人员的初 始级 别均为 1 级。然后进行升级,经理升为 4 级,技术人员和销售经理升为 3 级, 推销 员仍为1级。

月薪计算办法是: 经理拿固定月薪 8000 元;技术人员按每小时 100 元领取 月 薪; 推销员的月薪按该推销员当月销售额的 4%提成;销售经理既拿固定月薪 也 领取销售提成,固定月薪为 5000 元,销售提成为所管辖部门当月销售总额的 5%。



6.8 C语⾔中的⾯向接⼝编程

函数三要素： 名称、参数、返回值

趮号： 澮斞瞫訬莤蚻⃞祖幸篎：

澮斞瞫訬莤蚻⃞祖顳针：

6.8.1 函数类型语法基础

函数指针用于指向一个函数, 函数名是函数体的入口地址

1)可通过函数类型定义函数指针: FuncType\* pointer;

2)也可以直接定义：type (\*pointer)(parameter list);

pointer为函数指针变量名

type为指向函数的返回值类型

parameter list为指向函数的参数类型列表

|  |  |  |
| --- | --- | --- |
| #include <stdio.h>  typedef int(FUNC)(int);//定义⼀个函数类型  int test(int i) //定义⼀个函数  {  return i \* i;  }  void f() //定义⼀个函数  {  printf("Call f()...\n");  } | | |
| int main()  {  FUNC\* pt = test; | //定义⼀个指向函数的类型FUNC的指针pt,  //并初始化指向 test 函数 | |
| printf("Function pointer call: %d\n", pt(3)); //通过函数指针pt 间接调⽤test | | |
| void(\*pf)() = &f;  pf();  (\*pf)();  return 0;  } | | //直接定义⼀个函数指针 指向f  //通过函数指针间接调⽤  //通过函数指针间接调⽤ 等价于上述调⽤⽅法 |

趮考： 澮斞瞫訬莤蚻楅⃞类篎？

澮斞瞫訬莤蚻楅⃞顳针类篎？

澮斞瞫訬莤蚻!楅⃞顳针!!)顳嘓莤蚻楅⃞

的喿頎澕熤\*？

6.8.2 函数指针做函数参数

当函数指针 做为函数的参数，传递给一个被调用函数，被调用函数就可 以通过这个指针调用外部的函数，这就形成了回调。

|  |
| --- |
| #include <stdio.h>  int add(int a, int b);  int libfun(int (\*pDis)(int a, int b));  int main(void)  {  int (\*pfun)(int a, int b);//定义⼀个函数指针pfun 指向 int ()(int, int)函数类型  pfun = add;  libfun(pfun);  return 0;  }  int add(int a, int b)  {  return a + b;  }  int libfun(int (\*pDis)(int a, int b))  {  int a, b;  a = 1;  b = 2;  add(1,3); //直接调⽤add函数  printf("%d", pDis(a, b)); //通过函数指针做函数参数 ,间接调⽤add函数  return 0;  } |

回调凶软由优点

1 函数的调用 和 函数的实现 有效的分离

2 类似C++的多态,可扩展

现在这几个函数是在同一个文件当中

int libfun(int (\*pDis)(int a, int b))

是一个库中的函数，就只有使用回调了，通过函数指针参数将外部函数地址传 入来实现调用。

函数 add 的代码作了修改，也不必改动库的代码，就可以正常实现调用 便于程序的维护和升级。

|  |
| --- |
| 刘备利用周逾、曹仁厨杀乏际,乘虚袭取了南补、荆州、裹阳,冰后又征服  了长沙等四补。周逾想想十分气恨,正无处报夏冰夺还荆州。不久,刘备忍然丧 偶,周逾计上(采,对孙权说： “ 您的娇娇,美丽、刚强,我们冰联姻坑曹沼义向 刘备招采,把他骗采南徐幽禁,逼他们拿荆州采块。 ”孙权大喜,訣派人到荆州说 采。  刘备认为这是骗局,想要拒纯,诸葛亮笑道： “迷个好票子上门何不答应? 您R管去东吴,我叫赵云陪您去,目有子排,己您得了夫人又不夫荆州。 ”  接着,诸葛亮暗暗关照赵云道： “ 我这里有三个锦囊,内藏三系妙计。到南徐时打 开第一个,到年底时打开第二个,向急无路时打开第三个。 ”  第一个锦囊  一到东吴就拜会乔国老  第二个锦囊  刘备被孙权设计留下就对他谎称曹揉大军压境  第三个锦囊  被东吴军队追赶就示孙夫人解围 |

回调函数的本质：

提前做了一个协议的约定(把函数的参数、函数返回值提前约定)

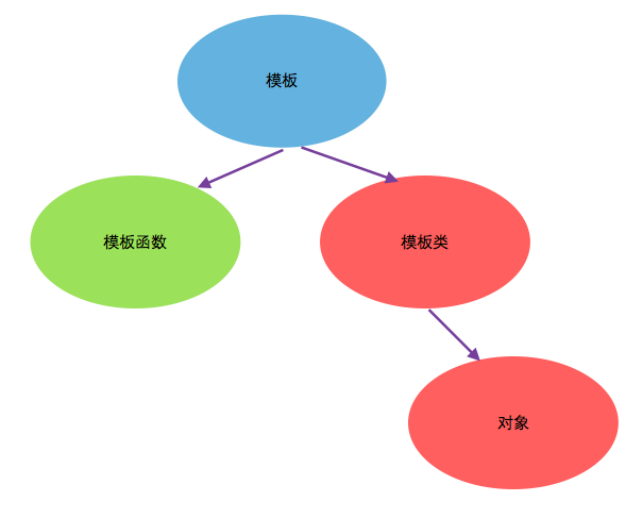


7. 模板

泛型(Generic Programming)即是指具有在多种数据类型上皆可操作的含 意。 泛型编 程的代表作品 STL 是一种高效、泛型、可交互操作的软件组件。

泛型编程最初诞生于 C++中,目的是为了实现 C++的 STL(标准模板库)。 其语 言支持机制就是模板(**Templates**)。模板的精神其实很简单:参数化类型。 换句话说, 把一个原本特定于某个类型的算法或类当中的类型信息抽掉,抽出来 做成模板参数 T。

所谓函数模板，实际上是建立一个通用函数，其函数类型和形参类型不 具体指定，用一个虚拟的类型来代表。这个通用函数就称为函数模板。



7.1 函数模板

7.1.1 函数重载实现的泛型

写n个函数，交换char类型、 int类型变量的值。

|  |
| --- |
| #include <iostream>  using namespace std;  void myswap(int &a, int &b)  {  int t = a;  a = b;  b = t;  }  void myswap(char &a, char &b)  {  char t = a;  a = b;  b = t;  }  int main(void)  {  int x = 1;  int y = 2;  myswap(x, y);  cout <<"x: " << x <<", y:" << y <<endl;  char a = 'c';  char b = 'b';  myswap(a, b);  cout <<"a: " << a <<", b:" << b <<endl;  return 0;  } |

7.1.2 函数模板的引入

语法格式

|  |
| --- |
| template<typename T>  template<class T>  -----------------------------  template<typename 类型参数表>  返回类型 函数模板名(函数参数列表) {  函数模板定义体  } |

template 是语义是模板的意思,尖括号中先写关键字 typename 或是 class ,后 面跟一个类型 T,此类即是虚拟的类型。至于为什么用 T,用的人多了,也 就是 T 了。

7.1.3 函数模板的实例

调用过程是这样的,先将函数模板实再化为函数,然后再发生函数调用。

|  |
| --- |
| #include <iostream>  using namespace std;  template<typename T>  void myswap(T &a, T &b)  {  T t = a;  a = b;  b = t;  }  int main(void)  {  int x = 1;  int y = 2;  myswap(x, y);  cout <<"x: " << x <<", y:" << y <<endl;  myswap<int>(x, y);  cout <<"x: " << x <<", y:" << y <<endl;  char a = 'a';  char b = 'b';  myswap(a, b);  cout <<"a: " << a <<", b:" << b <<endl;  myswap<char>(a, b);  cout <<"a: " << a <<", b:" << b <<endl; |

|  |
| --- |
| return 0;  } |

函数模板,只适用于函数的参数个数相同而类型不同,且函数体相同的情况。 如 果个数不同,则不能用函数模板。

|  |
| --- |
| 练习写一个沒化的排序程序,用f也wP(af也T采遍用基本类型 |

|  |
| --- |
| template<typename T>  void sortArray(T \*array, int num); |

7.1.4 函数模板与函数重载

|  |
| --- |
| #include <iostream>  using namespace std;  template <typename T>  void myswap(T &a,  T &b)  {  T t;  t = a;  a = b;  b = t;  cout<<"myswap  模板函数do"<<endl;  }  void myswap(char &a, int &b)  {  int t;  t = a;  a = b;  b = t;  cout<<"myswap 普通函数do"<<endl;  }  int main()  {  char cData =  'a';  2;  int iData =  myswap(cData, iData);  myswap(iData, cData);  // 普通函数会进⾏隐⼠的数据类型转换  myswap<int>(cData, iData); |

|  |
| --- |
| // 函数模板不提供隐式的数据类型转换 必须是严格的匹配  return 0;  } |

普通函数会进行隐士的数据类型转换, 函数模板不提供隐式的数据类型转 换 必须是严格的匹配。

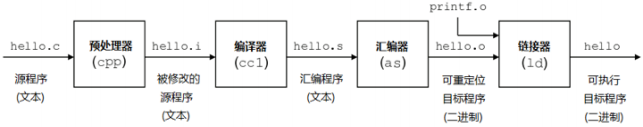
|  |
| --- |
| #include <iostream>  using namespace std;  int Max(int a, int b)  {  cout<<"int Max(int a, int b)"<<endl;  return a > b ? a : b;  }  template<typename T>  T Max(T a, T b)  {  cout<<"T Max(T a, T b)"<<endl;  return a > b ? a : b;  }  template<typename T>  T Max(T a, T b, T c)  {  cout<<"T Max(T a, T b, T c)"<<endl;  return Max(Max(a, b), c);  }  int main()  {  int a = 1;  int b = 2;  cout<<Max(a, b)<<endl; //当函数模板和普通函数都符合调⽤时 ,优先选择普通函数  cout<<Max<>(a, b)<<endl; //若显⽰使⽤函数模板 ,则使⽤<> 类型列表 cout<<Max(3.0, 4.0)<<endl; //如果 函数模板产⽣更好的匹配 使⽤函数模板 cout<<Max(5.0, 6.0, 7.0)<<endl; //重载  cout<<Max('a', 100)<<endl; //调⽤普通函数 可以隐式类型转换  return 0;  } |

153

|  |
| --- |
| 1. 当函数模板和普通函数都符合调用时,优先选择普通函数  2. 若显示使用函数模板,则使用<> 类型列表  3. 如果 函数模板产生更好的匹配 使用函数模板 |

7.1.5 编译器对模板机制剖析

简师编译器由编译过程



|  |
| --- |
| g++ -E hello.c -o hello.i (预处理)  g++ -S hello.i -o hello.s (编译)  g++ -c hello.s -o hello.o (汇编)  g++ hello.o -o hello (链接)  以上四个步骤，可合成一个步骤  g++ hello.c -o hello (直接编译链接成可执行目标文件) |

f也wP(af也.CPP

|  |
| --- |
| #include <iostream>  using namespace std;  template<class T>  void mySwap(T &a, T& b)  {  T c = a;  a = b;  b = c; |

154

|  |
| --- |
| }  int main(void)  {  int x = 10;  int y = 20;  mySwap<int>(x, y);  cout <<"x: " << x <<", y: " << y <<endl;  char a = 'a';  char b = 'b';  mySwap<char>(a, b);  cout <<"a: " << a <<", b: " << b << endl;  return 0;  } |

|  |
| --- |
| g++ -E template.cpp -o template.s |

fQwP(afQ.S

|  |
| --- |
| 2 .file "template.cpp"  3 .local \_ZStL8\_\_ioinit  4 .comm \_ZStL8\_\_ioinit,1,1  5 .section .rodata  6 .LC0:  7 .string "x: "  8 .LC1 :  9 .string ", y : "  10 .LC2 :  11 .string "a : "  12 .LC3 :  13 .string ", b : "  14 .text  15 .globl main  16 .type main, @function  17 main : # int main(void)  18 .LFB972 :  19 .cfi\_startproc  20 pushq %rbp  21 .cfi\_def\_cfa\_offset 16  22 .cfi\_offset 6, -16  23 movq %rsp, %rbp  24 .cfi\_def\_cfa\_register 6  25 pushq %r12  26 pushq %rbx  27 subq $16, %rsp  28 .cfi\_offset 12, -24 |

|  |
| --- |
| 29 .cfi\_offset 3, -32  30 movl $10, -24(%rbp)  31 movl $20, -20(%rbp)  32 leaq -20(%rbp), %rdx  33 leaq -24(%rbp), %rax  34 movq %rdx, %rsi  35 movq %rax, %rdi  36 call **\_Z6mySwapIiEvRT\_S1\_** # mySwap<int>(x, y);  37 movl -20(%rbp), %ebx  38 movl -24(%rbp), %r12d  39 movl $.LC0, %esi  40 movl $\_ZSt4cout, %edi  41 call \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc  42 movl %r12d, %esi  43 movq %rax, %rdi  44 call \_ZNSolsEi  45 movl $.LC1, %esi  46 movq %rax, %rdi  47 call \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc  48 movl %ebx, %esi  49 movq %rax, %rdi  50 call \_ZNSolsEi  51 movl $\_ZSt4endlIcSt11char\_traitsIcEERSt13basic\_ostreamIT\_T0\_ES6\_, %esi  52 movq %rax, %rdi  53 call \_ZNSolsEPFRSoS\_E  54 movb $97, -26(%rbp)  55 movb $98, -25(%rbp)  56 leaq -25(%rbp), %rdx  57 leaq -26(%rbp), %rax  58 movq %rdx, %rsi  59 movq %rax, %rdi  60 call **\_Z6mySwapIcEvRT\_S1\_** # mySwap<char>(a, b);  61 movzbl -25(%rbp), %eax  62 movsbl %al, %ebx  63 movzbl -26(%rbp), %eax  64 movsbl %al, %r12d  65 movl $.LC2, %esi  66 movl $\_ZSt4cout, %edi  67 call \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc  68 movl %r12d, %esi  69 movq %rax, %rdi  70 call \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_c  71 movl $.LC3, %esi  72 movq %rax, %rdi  73 call \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_PKc  74 movl %ebx, %esi  75 movq %rax, %rdi  76 call \_ZStlsISt11char\_traitsIcEERSt13basic\_ostreamIcT\_ES5\_c  77 movl $\_ZSt4endlIcSt11char\_traitsIcEERSt13basic\_ostreamIT\_T0\_ES6\_,  %esi  78 movq %rax, %rdi  79 call \_ZNSolsEPFRSoS\_E  80 movl $0, %eax  81 addq $16, %rsp  155 |

|  |
| --- |
| 82 popq %rbx  83 popq %r12  84 popq %rbp  85 .cfi\_def\_cfa 7, 8  86 ret  87 .cfi\_endproc  88 .LFE972 :  89 .size main, .-main  90 .section .text.\_Z6mySwapIiEvRT\_S1\_,"axG",@progbits,\_Z6mySwapIiEvRT\_S1  \_,comdat  91 .weak \_Z6mySwapIiEvRT\_S1\_  92 .type \_Z6mySwapIiEvRT\_S1\_, @function  93 **\_Z6mySwapIiEvRT\_S1\_** : # void mySwap<int>(int &a, int &b)  94 .LFB973 :  95 .cfi\_startproc  96 pushq %rbp  97 .cfi\_def\_cfa\_offset 16  98 .cfi\_offset 6, -16  99 movq %rsp, %rbp  100 .cfi\_def\_cfa\_register 6  101 movq %rdi, -24(%rbp)  102 movq %rsi, -32(%rbp)  103 movq -24(%rbp), %rax  104 movl (%rax), %eax  105 movl %eax, -4(%rbp)  106 movq -32(%rbp), %rax  107 movl (%rax), %edx  108 movq -24(%rbp), %rax  109 movl %edx, (%rax)  110 movq -32(%rbp), %rax  111 movl -4(%rbp), %edx  112 movl %edx, (%rax)  113 popq %rbp  114 .cfi\_def\_cfa 7, 8  115 ret  116 .cfi\_endproc  117 .LFE973 :  118 .size \_Z6mySwapIiEvRT\_S1\_, .-\_Z6mySwapIiEvRT\_S1\_  119 .section .text.\_Z6mySwapIcEvRT\_S1\_,"axG",@progbits,\_Z6mySwapIcEvRT\_S  1\_,comdat  120 .weak \_Z6mySwapIcEvRT\_S1\_  121 .type \_Z6mySwapIcEvRT\_S1\_, @function  122 **\_Z6mySwapIcEvRT\_S1\_** : # void mySwap<char>(char &a, char &a)  123 .LFB977 :  124 .cfi\_startproc  125 pushq %rbp  126 .cfi\_def\_cfa\_offset 16  127 .cfi\_offset 6, -16  128 movq %rsp, %rbp  129 .cfi\_def\_cfa\_register 6  130 movq %rdi, -24(%rbp)  131 movq %rsi, -32(%rbp)  132 movq -24(%rbp), %rax  133 movzbl (%rax), %eax  156 |

|  |
| --- |
| 134 movb %al, -1(%rbp)  135 movq -32(%rbp), %rax  136 movzbl (%rax), %edx  137 movq -24(%rbp), %rax  138 movb %dl, (%rax)  139 movq -32(%rbp), %rax  140 movzbl -1(%rbp), %edx  141 movb %dl, (%rax)  142 popq %rbp  143 .cfi\_def\_cfa 7, 8  144 ret  145 .cfi\_endproc  146 .LFE977 :  147 .size \_Z6mySwapIcEvRT\_S1\_, .-\_Z6mySwapIcEvRT\_S1\_  148 .text  149 .type \_Z41\_\_static\_initialization\_and\_destruction\_0ii, @function  150 \_Z41\_\_static\_initialization\_and\_destruction\_0ii :  151 .LFB984 :  152 .cfi\_startproc  153 pushq %rbp  154 .cfi\_def\_cfa\_offset 16  155 .cfi\_offset 6, -16  156 movq %rsp, %rbp  157 .cfi\_def\_cfa\_register 6  158 subq $16, %rsp  159 movl %edi, -4(%rbp)  160 movl %esi, -8(%rbp)  161 cmpl $1, -4(%rbp)  162 jne .L5  163 cmpl $65535, -8(%rbp)  164 jne .L5  165 movl $\_ZStL8\_\_ioinit, %edi  166 call \_ZNSt8ios\_base4InitC1Ev  167 movl $\_\_dso\_handle, %edx  168 movl $\_ZStL8\_\_ioinit, %esi  169 movl $\_ZNSt8ios\_base4InitD1Ev, %edi  170 call \_\_cxa\_atexit  171 .L5 :  172 leave  173 .cfi\_def\_cfa 7, 8  174 ret  175 .cfi\_endproc  176 .LFE984 :  177 .size \_Z41\_\_static\_initialization\_and\_destruction\_0ii, .-  \_Z41\_\_static\_initialization\_and\_destruction\_0ii  178 .type \_GLOBAL\_\_sub\_I\_main, @function  179 \_GLOBAL\_\_sub\_I\_main :  180 .LFB985 :  181 .cfi\_startproc  182 pushq %rbp  183 .cfi\_def\_cfa\_offset 16  184 .cfi\_offset 6, -16  185 movq %rsp, %rbp  186 .cfi\_def\_cfa\_register 6  187 movl $65535, %esi  157 |

|  |
| --- |
| 188 movl $1, %edi  189 call \_Z41\_\_static\_initialization\_and\_destruction\_0ii  190 popq %rbp  191 .cfi\_def\_cfa 7, 8  192 ret  193 .cfi\_endproc  194 .LFE985 :  195 .size \_GLOBAL\_\_sub\_I\_main, .-\_GLOBAL\_\_sub\_I\_main  196 .section .init\_array,"aw"  197 .align 8  198 .quad \_GLOBAL\_\_sub\_I\_main  199 .hidden \_\_dso\_handle  200 .ident "GCC : (Ubuntu 4.8.4-2ubuntu1~14.04) 4.8.4"  201 .section .note.GNU-stack,"",@progbits |

|  |
| --- |
| 1. 编译器并不是把函数模板处理成能够处理任意类的函数  2. 编译器从函数模板通过具体类型产生不同的函数  3. 编译器会对函数模板进行两次编译,在声明的地方对模板代码本身进行编译； 在调用的地方对参数替换后的代码进行编译。 |

7.2 类模板

7.2.1 类模板定义

类模板与函数模板的定义和使用类似，我们已经进行了介绍。 有时，有 两个或多个类，其功能是相同的，仅仅是数据类型不同 ，所以将类中的类型进 行泛化。

|  |
| --- |
| template<typename T>  class A  {  } |

7.2.2 简单的模板类

|  |
| --- |
| #include <iostream>  using namespace std;  template<typename T> //定义类模板 要在类的头部之前加⼊template<typename T> 或者  template<class T>  class A  158 |

|  |
| --- |
| {  public :  A(T t)  {  this->t = t;  }  T &getT()  {  return t;  }  public:  T t;  };  int main(void)  {  //在定义⼀个对象的时候，要明确此类所泛化的具体类型  A<int> a(100);  cout << a.getT() << endl;  return 0;  } |

7.2.3 模板类的派生

|  |
| --- |
| #include <iostream>  using namespace std;  template<class T>  class A  {  public :  A(T a) {  this->a = a;  }  protected :  T a;  };  //模板类派⽣普通类  //结论: ⼦类从模板类继承的时候 ,需要让编译器知道 ⽗类的数据类型具体是什么(数据类型的本 质:固定⼤⼩内存块的别名)A<int>  class B : public A<int>  {  public :  B(int a, int b) : A<int>(a)  {  this->b = b;  159 |

|  |
| --- |
| }  void printB()  {  cout<<"b: "<<b <<endl;  }  private :  int b;  };  //模板类派生模板类  template <class T>  class C :public A<T>  {  public :  C(T a, T c): A<T>(a)  {  this->c = c;  }  void printC()  {  cout <<"C : " << c <<endl;  }  private :  T c;  }; |

7.3 类模板实现

7.3.1 函数体写在类中

|  |
| --- |
| #include <iostream>  using namespace std;  template <class T>  class Complex  {  friend ostream & operator<<(ostream &os, Complex &c)  {  os << "( " << c.a << " + " << c.b << "i" << " )";  return os;  }  public:  Complex()  {  }  Complex(T a, T b)  {  160 |

|  |
| --- |
| this->a = a;  this->b = b;  }  void printComplex()  {  cout << "( " << a << " + " << b << "i" << " )" << endl;  }  Complex operator+(Complex &another)  {  Complex temp(a +another.a, b +another.b);  return temp;  }  private:  T a;  T b;  };  int main(void)  {  Complex<int> a(10, 20); //让模板类具体化是为了告诉编译具体的⼤⼩，分配内存 Complex<int> b(3, 4);  a.printComplex();  Complex<int> c;  c = a + b; // Complex c.operator+(Complex &another)  c.printComplex();  cout << c << endl;  return 0;  } |

7.3.2 函数体写在类外(在一个cpp中)

|  |
| --- |
| #include <iostream>  using namespace std;  template <class T>  class Complex;  template <class T>  Complex<T> mySub(Complex<T> &one, Complex<T> &another);  template <class T>  class Complex  {  public:  161 |

|  |
| --- |
| friend ostream & operator<< <T> (ostream &os, Complex<T> &c);  //在模板类中 如果有友元重载操作符<<或者>>需要 在 operator<< 和 参数列表之间 //加⼊ <T>  //滥⽤友元函数，本来可以当成员函数，却要⽤友元函数  //如果说是⾮<< >> 在模板类中当友元函数  //在这个模板类 之前声明这个函数  friend Complex<T> mySub <T> (Complex<T> &one, Complex<T> &another);  //最终的结论， 模板类 不要轻易写友元函数， 要写的 就写<< 和>> 。  Complex();  Complex(T a, T b);  Complex operator+(Complex &another);  Complex operator-(Complex &another);  void printComplex();  private:  T a;  T b;  };  template <class T>  Complex<T>::Complex()  {  }  template <class T>  Complex<T>::Complex(T a, T b)  {  this->a = a;  this->b = b;  }  template <class T>  void Complex<T>::printComplex()  {  cout << "( " << a << " + " << b << "i" << " )" << endl;  }  template <class T>  Complex<T> Complex<T>::operator+(Complex<T> &another)  {  Complex temp(a + another.a, b + another.b);  return temp;  }  template <class T>  Complex<T> Complex<T>::operator-(Complex<T> &another)  {  Complex temp(this->a - another.a, this->b = another.b);  return temp;  }  162 |

|  |
| --- |
| //友元函数  template <class T>  ostream & operator<<(ostream &os, Complex<T> &c)  {  os << "( " << c.a << " + " << c.b << "i" << " )";  return os;  }  template <class T>  Complex<T> mySub(Complex<T> &one, Complex<T> &another)  {  Complex<T> temp(one.a - another.a, one.b - another.b);  return temp;  }  int main(void)  {  Complex<int> a(10, 20); //让模板类具体化是为了告诉编译具体的⼤⼩，分配内存 Complex<int> b(3, 4);  a.printComplex();  Complex<int> c;  c = a + b;  c.printComplex();  cout << c << endl;  c = mySub(a, b);  cout << c << endl;  return 0;  } |

综上： 模板类不要轻易使用友元函数。



7.3.3 函数体写在类外(在.h和.cpp中)

cowP(kx.h

|  |
| --- |
| #pragma once  #include <iostream>  using namespace std;  template <class T>  163 |

|  |
| --- |
| class Complex;  template <class T>  Complex<T> mySub(Complex<T> &one, Complex<T> &another);  template <class T>  ostream & operator<<(ostream &os, Complex<T> &c);  template <class T>  class Complex  {  friend ostream & operator<< <T> (ostream &os, Complex<T> &c);  //在模板类中 如果有友元重载操作符<<或者>>需要 在 operator<< 和 参数列表之间 //加⼊ <T>  //滥⽤友元函数，本来可以当成员函数，却要⽤友元函数  //如果说是⾮<< >> 在模板类中当友元函数  //在这个模板类 之前声明这个函数  friend Complex<T> mySub <T>(Complex<T> &one, Complex<T> &another);  //最终的结论， 模板类 不要轻易写友元函数， 要写的 就写<< 和>> 。  public:  Complex();  Complex(T a, T b);  void printComplex();  Complex operator+(Complex &another);  Complex operator-(Complex &another);  private:  T a;  T b;  }; |

cowP(kx.hPP

|  |
| --- |
| #include "Complex.h"  template <class T>  Complex<T>::Complex()  {  }  template <class T>  Complex<T>::Complex<T>(T a, T b)  {  this->a = a;  this->b = b;  }  template <class T>  164 |

|  |
| --- |
| void Complex<T>::printComplex()  {  cout << "( " << a << " + " << b << "i" << " )" << endl;  }  template <class T>  Complex<T> Complex<T>::operator+(Complex<T> &another)  {  Complex temp(a + another.a, b + another.b);  return temp;  }  template <class T>  Complex<T> Complex<T>::operator-(Complex<T> &another)  {  Complex temp(this->a - another.a, this->b = another.b);  return temp;  }  //友元函数  template <class T>  ostream & operator<<(ostream &os, Complex<T> &c)  {  os << "( " << c.a << " + " << c.b << "i" << " )";  return os;  }  template <class T>  Complex<T> mySub(Complex<T> &one, Complex<T> &another)  {  Complex<T> temp(one.a - another.a, one.b - another.b);  return temp;  } |

waiH.CPP

|  |
| --- |
| #include <iostream>  #include "Complex.h"  #include "Complex.hpp"  using namespace std;  int main(void)  { Complex<int> a(10, 20); //让模板类具体化是为了告诉编译具体的⼤⼩，分配内存  Complex<int> b(3, 4);  a.printComplex();  Complex<int> c;  c = a + b; // Complex c.operator+(Complex &another)  c.printComplex();  165 |

|  |
| --- |
| cout << c << endl;  c = mySub(a, b);  cout << c << endl;  return 0;  } |

由于二次编译，模板类在.h在第一次编译之后，并没有最终确定类的具 体实现，只是编译器的词法校验和分析。在第二次确定类的具体实现后，是 在.hpp文件生成的最后的具体类，所以main函数需要引入.hpp文件。

综上：引入hpp文件一说也是曲线救国之计，所以实现模板方法建议在同 一个文件.h中完成



7.3.4 类模板中的static

|  |  |
| --- | --- |
| #include <iostream>  using namespace std;  template <class T>  class A{  public :  static T s\_value;  };  //静态变量需要在类的外部初始化  template <class T>  T A<T>::s\_value = 0;  /\*  当编译器看⻅ A<int> 被调⽤， 将执⾏⼆次编译， ⽣成如下的类A  class A  {  public:  static int s\_value;  };  int A::s\_value = 0;  \*/ | |
| /\*  当编译器看⻅  class A  {  public:  static char | A<char> 被调⽤， 将执⾏⼆次编译， ⽣成如下的类A |
| s\_value; |
| 166 | |

|  |
| --- |
| };  char A::s\_value = 0;  \*/  int main(void)  {  A<int> a1, a2, a3; //class A <int>家族的 对象  A<char> b1, b2, b3; // class A <char>家族的 对象  a1.s\_value = 10;  b1.s\_value = 'a';  cout << a1.s\_value << endl;  cout << b1.s\_value << endl;  //打印出 a1.s\_value = 10, b1.s\_value = 'a' 说明 两个s\_value 在两个类中是不同  的  a1.s\_value++;  cout << a2.s\_value << endl;// 11  cout << a3.s\_value << endl;// 11  b1.s\_value++;  cout << b2.s\_value << endl; //'b'  cout << b3.s\_value << endl; //'b'  //通过以上结果，说明 a1， a2, a3 是属于A<int>家族的他们共享A<int>::s\_value;  // b1, b2, b3 是属于A<char>家族的他们共享  A<char>::s\_value;  return 0;  } |

练习 实现一个模板数组类

|  |
| --- |
| 请设计一个软组模扳类( M当V也cfor ),完成对iwf、char、T也ach也r类型元素的 管理。  需要实现 问造凶软 浇贝问造凶软 << [] 重载=橡作符。 |

8 类型转换

8.1 类型转换的名称和语法

类型转换有 c 风格的,当然还有 c++风格的。 c 风格的转换的格式很简单 (TYPE) EXPRESSION,但是 c 风格的类型转换有不少的缺点,有的时候用 c 风格 的转换是不合 适的,因为它可以在任意类型之间转换,比如你可以把一个指向 const 对象的指针转换 成指向非 const 对象的指针,把一个指向基类对象的指针 转换成指向一个派生类对象的 指针,这两种转换之间的差别是巨大的,但是传统 的 c 语言风格的类型转换没有区分这 些。还有一个缺点就是,c 风格的转换不容 易查找,他由一个括号加上一个标识符组成, 而这样的东西在 c++程序里一大 堆。所以 c++为了克服这些缺点,引进了 4 新的类型转换操作符。

**C**风格由强制类型转挾(T当P也casf)

|  |
| --- |
| TYPE b = (TYPE) a |

c++堤侠了+神类型转挾,分别处理不同由项合应用

static\_cast

reinterpreter\_cast

dynamic\_cast

const\_cast

静态类型转换。

重新解释类型转换。

子类和父类之间的多态类型转换。

去掉const属性转换。

8.2 转换⽅式

8.2.1 static\_cast 静态类型转换

|  |
| --- |
| static\_cast< 目标类型> (标识符) |

所谓的静态,即在编译期内即可决定其类型的转换,用的也是最多的一种。

|  |  |
| --- | --- |
| #include <iostream>  using namespace std; | |
| int main(void)  {  double dPi = 3.1415926;  int num1 = (int)dPi;  int num2 = dPi; | //c语⾔的 旧式类型转换  //隐式类型转换 |
| // 静态的类型转换 :  // 在编译的时 进⾏基本类型的转换 能替代c⻛格的类型转换 可以进⾏⼀部分检查 int num3 = static\_cast<int> (dPi); //c++的新式的类型转换运算符  cout << "num1:" << num1 << " num2:" << num2 << " num3:" << num3 << endl;  return 0;  } | |

8.2.2 dynamic\_cast 子类和父类之间的多态类型转换

|  |
| --- |
| dynamic\_cast< 目标类型> (标识符) |

用于多态中的父子类之间的强制转化。

|  |
| --- |
| #include <iostream>  using namespace std;  class Animal  {  public :  virtual void cry() = 0;  };  class Dog : public Animal  {  public :  virtual void cry()  {  cout << "旺旺~ " << endl;  }  void doHome()  {  cout << "看家" << endl;  }  };  class Cat : public Animal  169 |

|  |
| --- |
| {  public :  virtual void cry()  {  cout << "喵喵~ " << endl;  }  void doHome()  {  cout << "抓⽼⿏" << endl;  }  };  int main(void)  {  Animal \*base = NULL;  base = new Cat();  base->cry(); //此时⽗类指针指向 猫  //⽤于将⽗类指针转换成⼦类，  Dog \*pDog = dynamic\_cast<Dog \*>(base); //转换之后 讲⽗类指针转换成 ⼦类狗指针 // 但是由于⽗类指针此时指向的对象是猫， // 所以转换狗是失败的  if (pDog != NULL) //如果转换失败则返回 NULL  {  pDog->cry();  pDog->doHome();  }  Cat \*pCat = dynamic\_cast<Cat \*>(base);//转换之后 讲⽗类指针转换成 ⼦类猫指针 //向下转换  if (pCat != NULL)  {  pCat->cry();  pCat->doHome();  }  return 0;  } |

8.2.2 const\_cast 去掉const属性转换

|  |
| --- |
| const\_cast<目标类型> (标识符) //目标类类型只能是指针或引用。 |

|  |
| --- |
| #include <iostream>  using namespace std;  struct A { |

|  |
| --- |
| int data;  };  int main(void)  {  const A a = {200};  A a1 = const\_cast<A>(a);  a1.data = 300;  A &a2 = const\_cast<A&>(a);  a2.data = 300;  cout<<a.data<<a2.data<<endl;  A \*a3 = const\_cast<A\*>(&a);  a3->data = 400;  cout<<a.data<<a3->data<<endl;  const int x = 3;  int &x1 = const\_cast<int&>(x);  x1 = 300;  cout<<x<<x1<<endl;  int \*x2 = const\_cast<int\*>(&x);  \*x2 = 400;  cout<<x<<\*x2<<endl;  return 0;  } |

8.2.3 reinterpret\_cast 重新解释类型转换

|  |
| --- |
| reinterpret\_cast< 目标类型> (标识符) |

interpret 是解释的意思,reinterpret 即为重新解释,此标识符的意思即为 数据的二进制形式重新解释,但是不改变其值。

|  |
| --- |
| #include <iostream>  using namespace std;  class Animal  {  public :  virtual void cry() = 0;  };  class Dog : public Animal  {  171 |

|  |
| --- |
| public:  virtual void cry()  {  cout << "旺旺~ " << endl;  }  void doHome()  {  cout << "看家" << endl;  }  };  class Cat : public Animal  {  public :  virtual void cry()  {  cout << "喵喵~ " << endl;  }  void doHome()  {  cout << "抓⽼⿏" << endl;  }  };  class Book  {  public :  void printP()  {  cout << "book" << endl;  }  };  int main(void)  {  Animal \*base = NULL;  //1 可以把⼦类指针赋给 ⽗类指针 但是反过来是不可以的 需要 如下转换 //Dog \*pdog = base;  Dog \*pDog = static\_cast<Dog \*> (base);  //2 把base转换成其他 ⾮动物相关的 err  //Book \*book= static\_cast<Book \*> (base);  //3 reinterpret\_cast 可以强制类型转换  Book \*book = reinterpret\_cast<Book \*> (base);  return 0;  } |

|  |
| --- |
| 建议1：  程序员要清除的知道: 要转的变量，类型转换前是什么类型，类型转换后  是什么类型。转换后有什么后果。  建议2：  一般情况下，不建议进行类型转换。 |

9 异常

1 )异常是一种程序控制机制，与函数机制独立和互补

函数是一种以栈结构展开的上下函数衔接的程序控制系统,异常是另一种 控制结构,它依附于栈结构,却可以同时设置多个异常类型作为网捕条件,从而以 类型匹配在栈机制中跳跃回馈.

2 )异常设计目的：

栈机制是一种高度节律性控制机制,面向对象编程却要求对象之间有方 向、有目的的控制传动,从一开始，异常就是冲着改变程序控制结构，以适应面 向对象程序更有效地工作这个主题，而不是仅为了进行错误处理。

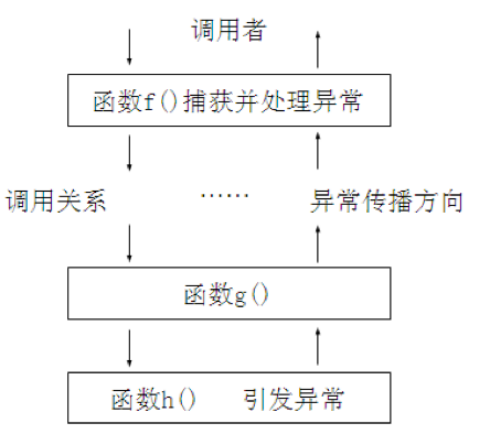
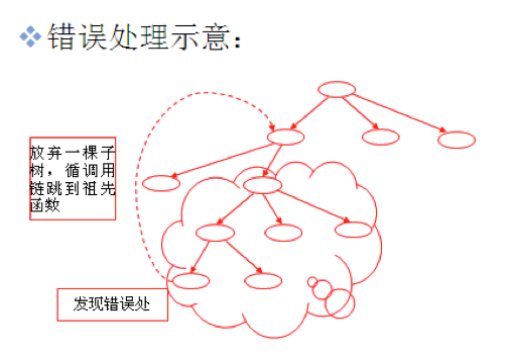
异常设计出来之后，却发现在错误处理方面获得了最大的好处。

9.1 异常处理的基本思想

9.1.1 传统的错误处理机制

通过函数返回值来处理错误。

9.1.2 异常的错误处理机制



1 ) C++的异常处理机制使得异常的引发和异常的处理不必在同一个函数中，这 样底层的函数可以着重解决具体问题，而不必过多的考虑异常的处理。上层调 用者可以再适当的位置设计对不同类型异常的处理。

2 )异常是专门针对抽象编程中的一系列错误处理的， C++中不能借助函数机 制，因为栈结构的本质是先进后出，依次访问，无法进行跳跃，但错误处理的 特征却是遇到错误信息就想要转到若干级之上进行重新尝试，如图

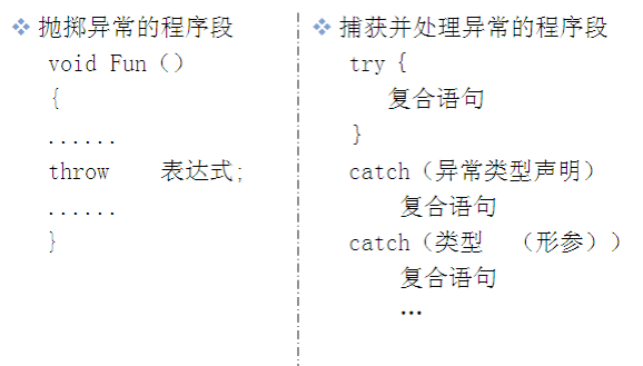
174

3 )异常超脱于函数机制，决定了其对函数的跨越式回跳。

4 )异常跨越函数

9.2 C++异常处理的实现

9.2.1 异常的基本语法



1 ) 若有异常则通过throw操作创建一个异常对象并抛掷。

2 ) 将可能抛出异常的程序段嵌在try块之中。控制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。

3 ) 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。

4 ) 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)。

5 ) 如果匹配的处理器未找到，则运行函数terminate将被自动调用，其缺省功 能是调用abort终止程序。 6)处理不了的异常，可以在catch的最后一个分支，使用throw语法，向上扔。

|  |
| --- |
| #include <iostream>  using namespace std;  int divide(int x, int y )  {  if (y ==0)  {  throw x;  }  return x/y;  }  int main(void)  {  try  {  cout << "8/2 = " << divide(8, 2) << endl;  cout << "10/0 =" << divide(10, 0) << endl;  }  catch (int e)  {  cout << "e" << " is divided by zero!" << endl;  }  catch(...)  {  cout << "未知异常" << endl;  }  return 0;  } |

9.2.2 栈解旋(unwinding)

异常被抛出后，从进入try块起，到异常被抛掷前，这期间在栈上的构造的 所有对象，都会被自动析构。析构的顺序与构造的顺序相反。这一过程称为栈 的解旋(unwinding)。

|  |
| --- |
| #include <iostream>  using namespace std;  class MyException {};  class Test  {  public :  Test(int a=0, int b=0)  {  176 |

|  |
| --- |
| this->a = a;  this->b = b;  cout << "Test 构造函数执⾏" << "a:" << a << " b: " << b << endl;  }  void printT()  {  cout << "a:" << a << " b: " << b << endl;  }  ~Test()  {  cout << "Test 析构函数执⾏" << "a:" << a << " b: " << b << endl;  }  private :  int a;  int b;  };  void myFunc() throw (MyException)  {  Test t1;  Test t2;  cout << "定义了两个栈变量,异常抛出后测试栈变量的如何被析构" << endl;  throw MyException();  }  int main(void)  {  //异常被抛出后，从进⼊try块起，到异常被抛掷前，这期间在栈上的构造的所有对象**>**， //都会被⾃动析构。析构的顺序与构造的顺序相反。  //这⼀过程称为栈的解旋(unwinding)  try  {  myFunc();  }  catch(MyException &e)  //catch(MyException ) //这⾥不能访问异常对象  {  cout << "接收到MyException类型异常" << endl;  }  catch(...)  {  cout << "未知类型异常" << endl;  }  return 0;  } |

9.2.3 异常接口声明 1 )为了加强程序的可读性，可以在函数声明中列出可能抛出的所有异常类 型，例如：

void func() throw (A, B, C , D); //这个函数func ()能够且只能抛出类 型A B C D及其子类型的异常。 2)如果在函数声明中没有包含异常接口声明，则次函数可以抛掷任何类型的 异常，例如：

void func();

3 )一个不抛掷任何类型异常的函数可以声明为：

void func() throw();

4 ) 如果一个函数抛出了它的异常接口声明所不允许抛出的异常， unexpected 函数会被调用，该函数默认行为调用terminate函数中止程序。

9.2.4 异常类型和异常变量的生命周期

1 ) throw的异常是有类型的，可以使，数字、字符串、类对象。

2 ) throw的异常是有类型的， catch严格按照类型进行匹配。

3 )注意 异常对象的内存模型 。

(一)传统由错误模型处理

|  |
| --- |
| //传统的错误处理机制  int my\_strcpy(char \*to, char \*from)  {  if (from == NULL)  {  return 1;  }  if (to == NULL)  {  return 2;  }  //copy是的 场景检查  if (\*from == 'a')  {  return 3; //copy时出错  }  while (\*from != '\0')  {  \*to = \*from;  to ++;  from ++;  } |

179

|  |
| --- |
| \*to = '\0';  return 0;  }  int main(void)  {  int ret = 0;  char buf1[] = "zbcdefg";  char buf2[1024] = {0};  ret = my\_strcpy(buf2, buf1);  if (ret != 0)  {  switch(ret)  {  case 1:  printf("源buf出错!\n");  break;  case 2:  printf("⺫的buf出错!\n");  break;  case 3:  printf("copy过程出错!\n");  break;  default :  printf("未知错误!\n");  break;  }  }  printf("buf2:%s \n", buf2);  return 0;  } |

(二)抛出普遍类型异常

|  |
| --- |
| //throw int类型异常  void my\_strcpy1(char \*to, char \*from)  {  if (from == NULL)  {  throw 1;  }  if (to == NULL)  {  throw 2;  }  //copy是的 场景检查  if (\*from == 'a')  {  throw 3; //copy时出错  } |

|  |
| --- |
| while (\*from != '\0')  {  \*to = \*from;  to ++;  from ++;  }  \*to = '\0';  } |

|  |
| --- |
| //throw char\*类型异常  void my\_strcpy2(char \*to, char \*from)  {  if (from == NULL)  {  throw "源buf出错";  }  if (to == NULL)  {  throw "⺫的buf出错";  }  //copy是的 场景检查  if (\*from == 'a')  {  throw "copy过程出错"; //copy时出错  }  while (\*from != '\0')  {  \*to = \*from;  to ++;  from ++;  }  \*to = '\0';  } |

(三) 抛出自定义类型异常

|  |
| --- |
| class BadSrcType {};  class BadDestType {};  class BadProcessType  {  public:  BadProcessType()  {  cout << "BadProcessType构造函数do \n";  }  BadProcessType(const BadProcessType &obj)  {  180 |

|  |
| --- |
| cout << "BadProcessType copy构造函数do \n";  }  ~BadProcessType()  {  cout << "BadProcessType析构函数do \n";  }  };  //throw 类对象 类型异常  void my\_strcpy3(char \*to, char \*from)  {  if (from == NULL)  {  throw BadSrcType();  }  if (to == NULL)  {  throw BadDestType();  }  //copy是的 场景检查  if (\*from == 'a')  {  printf("开始 BadProcessType类型异常 \n");  throw BadProcessType();  }  if (\*from == 'b')  {  throw &(BadProcessType());  }  if (\*from  == 'c')  new BadProcessType;  {  throw  }  while (\*from != '\0')  {  \*to = \*from;  to ++;  from ++;  }  \*to = '\0';  }  int main(void)  {  int ret = 0;  char buf1[] = "cbbcdefg";  char buf2[1024] = {0};  try  {  181 |

|  |
| --- |
| //my\_strcpy1(buf2, buf1);  //my\_strcpy2(buf2, buf1);  my\_strcpy3(buf2, buf1);  }  catch (int e) //e可以写 也可以不写  {  cout << e << " int类型异常" << endl;  }  catch(char \*e)  {  cout << e << " char\* 类型异常" << endl;  }  //---  catch(BadSrcType e)  {  cout << " BadSrcType 类型异常" << endl;  }  catch(BadDestType e)  {  cout << " BadDestType 类型异常" << endl;  }  //结论1: 如果 接受异常的时候 使⽤⼀个异常变量 ,则copy构造异常变量 .  /\*  catch( BadProcessType e)  {  cout << " BadProcessType 类型异常" << endl;  }  \*/  //结论2: 使⽤引⽤的话 会使⽤throw时候的那个对象  //catch( BadProcessType &e)  //{  // cout << " BadProcessType 类型异常" << endl;  //}  //结论3: 指针可以和引⽤/元素写在⼀块 但是引⽤/元素不能写在⼀块  catch( BadProcessType \*e)  {  cout << " BadProcessType 类型异常" << endl;  delete e;  }  //结论4: 类对象时 , 使⽤引⽤⽐较合适  // —  catch (...)  {  cout << "未知 类型异常" << endl;  }  return 0;  } |

9.2.5 异常的层次结构

设计一个数组类 MyArr6y ，重载[[操作，数组初始化时，对数组的个数进行有效 检查

1) index<0 抛出异常eNegative

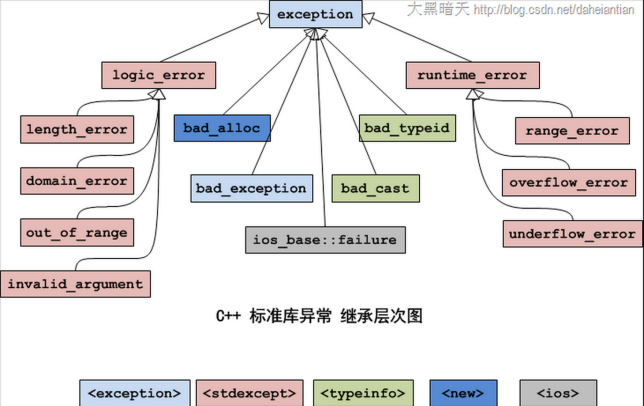
2) index = 0 抛出异常 eZero

3) index>1000抛出异常eTooBig

4) index<10 抛出异常eTooSmall

5) eSize类是以上类的父类，实现有参数构造、并定义virtual void printErr()输出错误。

9.3 标准程序库异常



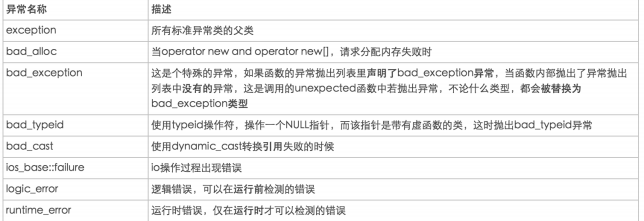
每个类所在的头文件在图下方标识出来.

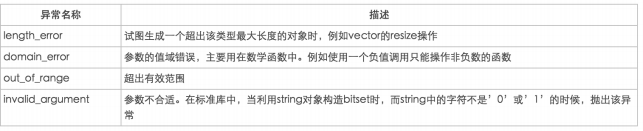
标准异常类的成员：

① 在上述继承体系中，每个类都有提供了构造函数、复制构造函数、和 赋值操作符重载。

② logic\_error类及其子类、runtime\_error类及其子类，它们的构造函数 是接受一个string类型的形式参数，用于异常信息的描述；

③ 所有的异常类都有一个what()方法，返回const char\* 类型(C风格字 符串)的值，描述异常信息。







案例1：

|  |
| --- |
| #include <iostream>  using namespace std;  #include <stdexcept>  class Teacher  {  public:  Teacher(int age) //构造函数 , 通过异常机制 处理错误  {  if (age > 100)  {  throw out\_of\_range("年龄太⼤");  }  this->age = age; |

|  |
| --- |
| }  protected :  private :  int age;  };  int main()  {  try  {  Teacher t1(102);  }  catch (out\_of\_range e)  {  cout << e.what() << endl;  }  return 0;  } |

案例2：

|  |
| --- |
| #include <iostream>  #include <stdexcept>  using namespace std;  class Dog  {  public:  Dog()  {  parr = new int[1024\*1024\*100]; //4MB  }  private :  int \*parr;  };  int main()  {  Dog \*pDog;  try{  for(int i=1; i<1024; i++) //40GB!  {  pDog = new Dog();  cout << i << ": new Dog 成功 ." << endl;  }  }  catch(bad\_alloc err)  {  cout << "new Dog 失败 : " << err.what() << endl;  185 |

|  |
| --- |
| }  return 0;  } |

10. 输入输出流

10.1 **I/O**流的概念和流类库的结构

程序的输入指的是从输入文件将数据传送给程序，程序的输出指的是从程序将 数据传送给输出文件。

C++输入输出包含以下三个方面的内容：

对系统指定的标准设备的输入和输出。即从键盘输入数据，输出到显示器屏 幕。这种输入输出称为标准的输入输出，简称标准I/O。

以外存磁盘文件为对象进行输入和输出，即从磁盘文件输入数据，数据输出 到磁盘文件。以外存文件为对象的输入输出称为文件的输入输出，简称文件I/

O。

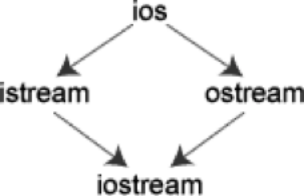
对内存中指定的空间进行输入和输出。通常指定一个字符数组作为存储空间 (实际上可以利用该空间存储任何信息)。这种输入和输出称为字符串输入输 出，简称串I/O。

**C++**的**I/O**对**C**的发展**--**类型安全和可扩展性

在C语言中，用prinV和scanf进行输入输出，往往不能保证所输入输出的 数据是可靠的安全的。 在C++的输入输出中，编译系统对数据类型进行严格的 检查，凡是类型不正确的数据都不可能通过编译。因此C++的I/O操作是类型安 全(type safe)的。 C++的I/O操作是可扩展的，不仅可以用来输入输出标准类型的 数据，也可以用于用户自定义类型的数据。

C++通过I/O类库来实现丰富的I/O功能。这样使C++的输人输出明显地优于 C 语言中的prinV和scanf ，但是也为之付出了代价， C++的I/O系统变得比较复 杂，要掌握许多细节。

C++编译系统提供了用于输入输出的iostream类库。 iostream这个单词是由 3个部 分组成的，即i-o-stream ，意为输入输出流。在iostream类库中包含许多 用于输入输出的 类。常用的见表

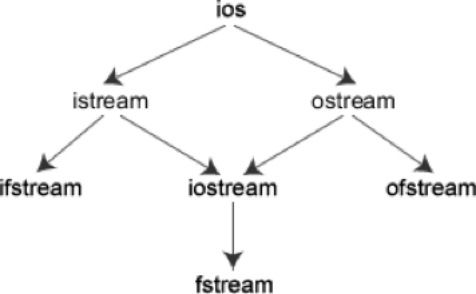




ios是抽象基类，由它派生出istream类和ostream类，两个类名中第1个字 母i和o分别代表输入(input)和输出(output)。 istream类支持输入操作， ostream 类支持输出操作， iostream类支持输入输出操作。 iostream类是从istream类和 ostream类通过多重继承而派生的类。其继承层次见上图表示。

C++对文件的输入输出需要用ifstrcam和ofstream类，两个类名中第1个字母i和o 分别代表输入和输出，第2个字母f代表文件 (ﬁle)。ifstream支持对文件的输入操

作， ofstream支持对文件的输出操作。类ifstream继承了类istream ，类ofstream 继承了类ostream ，类fstream继承了 类iostream。见图



I/O类库中还有其他一些类，但是对于一般用户来说，以上这些已能满足需要 了。

与iostream类库有关的头文件

iostream类库中不同的类的声明被放在不同的头文件中，用户在自己的程序中 用#include命令包含了有关的头文件就相当于在本程序中声明了所需 要用到的 类。可以换 —种说法：头文件是程序与类库的接口，iostream类库的接口分 别由不同的头文件来实现。常用的有

• iostream 包含了对输入输出流进行操作所需的基本信息。

• fstream 用于用户管理的文件的I/O操作。

• strstream 用于字符串流I/O。

• stdiostream 用于混合使用C和C + +的I/O机制时，例如想将C程序转变 为C++程序。

• iomanip 在使用格式化I/O时应包含此头文件。

在**iostream**头文件中定义的流对象

在 iostream 头文件中定义的类有 ios，istream，ostream，iostream， istream \_withassign， ostream\_withassign，iostream\_withassign 等。

在**iostream**头文件中重载运算符

“<<”和“>>”本来在C++中是被定义为左位移运算符和右位移运算符的，由于在 iostream头文件中对它们进行了重载， 使它们能用作标准类型数据的输入和输 出运算符。所以，在用它们的程序中必须用#include命令把iostream包含到程序 中。

#include <iostream>

1) >>a表示将数据放入a对象中。

2) <<a表示将a对象中存储的数据拿出。

10.2 标准I/O流

标准I/O对象:cin，cout，cerr，clog

cout流对象

cout是console output的缩写，意为在控制台(终端显示器)的输出。强调几 点。

1) cout不是C++预定义的关键字，它是ostream流类的对象，在iostream中 定义。 顾名思义，流是流动的数据，cout流是流向显示器的数据。cout流中 的数据是用流插入运算符“<<”顺序加入的。如果有

cout<<"I "<<"study C++ "<<"very hard. << “wang bao ming "; 按顺序将字符串"I ", "study C++ ", "very hard."插人到cout流中，cout就将

它们送到显示器，在显示器上输出字符串"I study C++ very hard."。cout流 是容纳数据的载体，它并不是一个运算符。人们关心的是cout流中的内容，也 就是向显示器输出什么。 2)用“ccmt<<”输出基本类型的数据时，可以不必考虑数据是什么类型，系 统会判断数据的类型，并根据其类型选择调用与之匹配的运算符重 载函数。这 个过程都是自动的，用户不必干预。如果在C语言中用prinf函数输出不同类型 的数据，必须分别指定相应的输出格式符，十分麻烦，而且容易出 错。C++ 的I/O机制对用户来说，显然是方便而安全的。

3) cout流在内存中对应开辟了一个缓冲区，用来存放流中的数据，当向cout 流插 人一个endl时，不论缓冲区是否已满，都立即输出流中所有数据，然后插 入一个换行符， 并刷新流(清空缓冲区)。注意如果插人一个换行符”\n“(如 cout<<a<<"\n")，则只输出和换行，而不刷新cout 流(但并不是所有编译系 统都体现出这一区别)。

4) 在iostream中只对"<<"和">>"运算符用于标准类型数据的输入输出进行了 重载，但未对用户声明的类型数据的输入输出 进行重载。如果用户声明了新的

类型，并希望用”<<"和">>"运算符对其进行输入输出，按照重运算符重载来 做。

cerr流对象

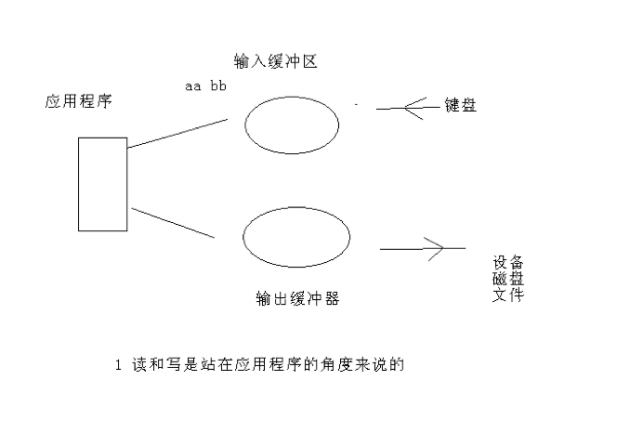
cerr流对象是标准错误流，cerr流已被指定为与显示器关联。cerr的 作用是向 标准错误设备(standard error device)输出有关出错信息。 cerr与标准输出流 cout的作用和用法差不多。但有一点不同：cout流通常是传送到显示器输出， 但也可以被重定向 输出到磁盘文件，而cerr流中的信息只能在显示器输出。当 调试程序时，往往不希望程序运行时的出错信息被送到其他文件，而要求在显 示器上及时输出，这时 应该用cerr。cerr流中的信息是用户根据需要指定的

。

clog流对象

clog流对象也是标准错误流，它是console log的缩写。它的作用和cerr相同， 都是在终端显示器上显示出错信息。 区别：cerr是不经过缓冲区，直接向显示 器上输出有关信息，而clog中的信息存放在缓冲区中，缓冲区满后或遇endl时 向显示器输出。

缓冲区概念：



10.2.1 标准的输入流

标准输入流对象cin ，重点掌握的函数

cin.get() //一次只能读取一个字符

cin.get(一个参数) //读一个字符

cin.get(三个参数) //可以读字符串

cin.getline()

cin.ignore()

cin.putback()

|  |
| --- |
| #include <iostream>  using namespace std;  //2 输⼊字符串 你 好 遇⻅空格,停⽌接受输⼊  void main()  {  char YourName[50];  int myInt;  long myLong;  double myDouble;  float myFloat;  unsigned int myUnsigned;  cout << "请输⼊⼀个Int: ";  cin >> myInt;  cout << "请输⼊⼀个Long: ";  cin >> myLong;  cout << "请输⼊⼀个Double: ";  cin >> myDouble;  cout << "请输⼊你的姓名: ";  cin >> YourName;  cout << "\n\n你输⼊的数是： " << endl;  cout << "Int: \t" << myInt << endl;  cout << "Long: \t" << myLong << endl;  cout << "Double: \t" << myDouble << endl;  cout << "姓名: \t" << YourName << endl;  cout<< endl << endl;  return 0;  } |

|  |
| --- |
| using namespace std;  //1 输⼊英⽂ ok  //2 ctr+z 会产⽣⼀个 EOF(-1)  int main()  {  char ch;  while( (ch= cin.get())!= EOF)  {  std::cout << "字符: " << ch << std::endl;  }  std::cout << "\n结束 .\n";  return 0;  } |

|  |
| --- |
| #include <iostream>  using namespace std;  //演⽰:读⼀个字符 链式编程  int main(void)  {  char a, b, c;  cin.get(a);  cin.get(b);  cin.get(c);  cout << a << b << c<< endl;  cout << "开始链式编程" << endl;  cin.get(a).get(b).get(c);  cout << a << b << c<< endl;  return 0;  } |

|  |
| --- |
| #include <iostream>  using namespace std;  //演⽰cin.getline() 可以接受空格  int main(void)  {  char buf1[256];  char buf2[256];  192 |

|  |
| --- |
| cout << "\n请输⼊你的字符串 不超过256" ;  cin.getline(buf1, 256, '\n');  cout << buf1 << endl;  //  cout << "注意: cin.getline() 和 cin >> buf2 的区别 , 能不能带空格 " << endl; cin >> buf2 ; //流提取操作符 遇⻅空格 停⽌提取输⼊流  cout << buf2 << endl;  return 0;  } |

|  |
| --- |
| #include <iostream>  using namespace std;  //缓冲区实验  /\*  1 输⼊ "aa bb cc dd" 字符串⼊缓冲区  2 通过 cin >> buf1; 提⾛了 aa  3 不需要输⼊ 可以再通过cin.getline() 把剩余的缓冲区数据提⾛  \*/  int main()  {  char buf1[256];  char buf2[256];  cout << "请输⼊带有空格的字符串 ,测试缓冲区" << endl;  cin >> buf1;  cout << "buf1:" << buf1 << endl;  cout << "请输⼊数据 ..." << endl;  //缓冲区没有数据 ,就等待; 缓冲区如果有数据直接从缓冲区中拿⾛数据  cin.getline(buf2, 256);  cout << "buf2:" << buf2 << endl;  return 0;  } |

|  |
| --- |
| #include <iostream>  using namespace std;  // ignore  int main(void)  {  int intchar;  char buf1[256];  char buf2[256];  cout << "请输⼊带有空格的字符串,测试缓冲区 aa bbccddee " << endl;  193 |

|  |
| --- |
| cin >> buf1;  cout << "buf1:" << buf1 << endl;  cout << "请输⼊数据 ..." << endl;  cin.ignore(2);  //缓冲区没有数据 ,就等待; 缓冲区如果有数据直接从缓冲区中拿⾛数据  cin.getline(buf2, 256);  cout << "buf2:" << buf2 << endl;  return 0;  } |

|  |
| --- |
| #include <iostream>  using namespace std;  //案例:输⼊的整数和字符串分开处理  int main()  {  cout << "Please, enter a number or a word: ";  char c = cin.get();  if ( (c >= '0') && (c <= '9') ) //输⼊的整数和字符串 分开处理  {  int n; //整数不可能 中间有空格 使⽤cin >>n  cin.putback (c);  cin >> n;  cout << "You entered a number: " << n << '\n';  }  else  {  string str;  cin.putback (c);  getline (cin,str); // //字符串 中间可能有空格 使⽤ cin.getline(); cout << "You entered a word: " << str << '\n';  }  return 0;  } |

10.2.2 标准的输出流

标准输出流对象cout

cout.put()

cout.write()

cout.width()

cout.fill()

cout.setf(标记)

操作符、控制符

flush

endl

oct

dec

hex

setbase

setw

setfill

setprecision

|  |
| --- |
| #include <iostream>  #include <iomanip>  using namespace std;  int main(void)  {  cout << "hello world" << endl;  cout.put('h').put('e').put('l').put('\n');  cout.write("hello world", 4); //输出的长度  char buf[] = "hello world";  printf("\n");  cout.write(buf, strlen(buf));  printf("\n");  cout.write(buf, strlen(buf) - 6);  printf("\n");  return 0;  } |

|  |
| --- |
| #include <iostream>  #include <iomanip>  using namespace std;  //使⽤cout.setf()控制符  int main(void)  {  //使⽤类成员函数  cout << "<start>";  cout.width(30);  cout.fill( '\*');  cout.setf(ios::showbase); //#include <iomanip>  cout.setf(ios::internal); //设置  cout << hex << 123 << "<End>\n";  cout << endl;  cout << endl;  //使⽤ 操作符、控制符  cout << "<Start>"  << setw(30)  << setfill( '\*')  << setiosflags(ios::showbase) //基数  << setiosflags(ios::internal)  << hex  << 123  << "<End>\n"  << endl;  return 0;  } |

10.2.3 输出格式化

在输出数据时，为简便起见，往往不指定输出的格式，由系统根据数据 的类型采取默认的格式，但有时希望数据按指定的格式输出，如要求以十六进 制或八进制形式 输出一个 整数，对输出的小数只保留两位小数等。有两种方 法可以达到此目的。

1 )使用控制符的方法；

2 )使用流对象的有关成员函数。分别叙述如下。



使用控制符的方法

|  |
| --- |
| #include <iostream>  #include <iomanip>  using namespace std;  int main()  {  int a;  cout<<"input a:";  cin>>a;  cout<<"dec:"<<dec<<a<<endl; //以⼗进制形式输出整数  cout<<"hex:"<<hex<<a<<endl; //以⼗六进制形式输出整数a  cout<<"oct:"<<setbase(8)<<a<<endl; //以⼋进制形式输出整数a  const char \*pt="China"; //pt指向字符串"China"  cout<<setw(10)<<pt<<endl; //指定域宽为 ,输出字符串  cout<<setfill( '\*')<<setw(10)<<pt<<endl; //指定域宽 ,输出字符串 ,空⽩处以'\*'填**>**充 double pi=22.0/7.0; //计算pi值  //按指数形式输出,8位⼩数  cout<<setiosflags(ios::scientific)<<setprecision(8);  197 |

|  |
| --- |
| cout<<"pi="<<pi<<endl; //输出pi值  cout<<"pi="<<setprecision(4)<<pi<<endl; //改为位小数  return 0;  } |

结果：

|  |
| --- |
| input a:16  dec:16  hex:10  oct:20  China  \*\*\*\*\*China  pi=3.14285714e+00  pi=3.1429e+00  pi=0x1.9249249249249p+1 |

人们在输入输出时有一些特殊的要求，如在输出实数时规定字段宽度， 只保留两位小数，数据向左或向右对齐等。 C++提供了在输入输出流中使用的 控制符(有的书中称为操纵符)

举例， 输出双精度数：

double a=123.456789012345; // 对a赋初值

1) cout<<a; 输出： 123.456

2) cout<<setprecision(9)<<a; 输出： 123.456789

3) cout<<setprecision(6); 恢复默认格式(精度为6)

4) cout<< setiosflags(ios∷fixed); 输出： 123.456789

5) cout<<setiosflags(ios∷fixed)<<setprecision(8)<<a; 输出： 123.45678901

6) cout<<setiosflags(ios∷scientific)<<a; 输出： 1.234568e+02

7) cout<<setiosflags(ios∷scientific)<<setprecision(4)<<a; 输出： 1.2346e02

下面是整数输出的例子：

int b=123456; // 对b赋初值

1) cout<<b; 输出： 123456

2) cout<<hex<<b; 输出： 1e240

3) cout<<setiosflags(ios∷uppercase)<<b; 输出： 1E240

4) cout<<setw(10)<<b<<'，'<<b; 输出： 123456，123456

5) cout<<setfill('\*')<<setw(10)<<b; 输出： \*\*\*\* 123456

6) cout<<setiosflags(ios∷showpos)<<b; 输出： +123456

如果在多个cout语句中使用相同的setw(n) ，并使用se5osﬂags(ios::right)， 可以实现各行数据右对齐，如果指定相同的精度，可以实现上下小数点对齐。

例如：各行小数点对齐。

int main( )

{

double a=123.456,b=3.14159,c=-3214.67;

cout<<setiosflags(ios::fixed)<<setiosflags(ios::right)<<setprecision(2); cout<<setw(10)<<a<<endl;

cout<<setw(10)<<b<<endl;

cout<<setw(10)<<c<<endl;

system("pause");

return 0;

}

输出如下：

123.46 (字段宽度为10 ，右对齐，取两位小数) 3.14

-3214.67

先统一设置定点形式输出、取两位小数、右对齐。这些设置对其后的输出 均有效(除非重新设置) ，而setw只对其后一个输出项有效，因此必须在输出a， b，c之前都要写setw(10)。

用流对象的成员函数控制输出格式

除了可以用控制符来控制输出格式外，还可以通过调用流对象cout中用于控制 输出格式的成员函数来控制输出格式。用于控制输出格式的常用的成员函数如 下：



流成员函数seV和控制符se5osﬂags括号中的参数表示格式状态，它是通过格式 标志来指定的。格式标志在类ios中被定义为枚举值。因此在引用这些格式标志 时要在前面加上类名ios和域运算符“::”。格式标志见表13.5。



201

|  |
| --- |
| #include <iostream>  #include <iomanip>  using namespace std;  int main( )  {  int a=21;  cout.setf(ios::showbase);//显⽰基数符号(0x或)  cout<<"dec:"<<a<<endl; //默认以⼗进制形式输出a  cout.unsetf(ios::dec); //终⽌⼗进制的格式设置  cout.setf(ios::hex); //设置以⼗六进制输出的状态  cout<<"hex:"<<a<<endl; //以⼗六进制形式输出a  cout.unsetf(ios::hex); //终⽌⼗六进制的格式设置  cout.setf(ios::oct); //设置以⼋进制输出的状态  cout<<"oct:"<<a<<endl; //以⼋进制形式输出a  cout.unsetf(ios::oct);  const char \*pt="China"; //pt指向字符串"China"  cout.width(10); //指定域宽为  cout<<pt<<endl; //输出字符串  cout.width(10); //指定域宽为  cout.fill( '\*'); //指定空⽩处以'\*'填充  cout<<pt<<endl; //输出字符串  double pi=22.0/7.0; //输出pi值  cout.setf(ios::scientific); //指定⽤科学记数法输出  cout<<"pi="; //输出"pi="  cout.width(14); //指定域宽为  cout<<pi<<endl; //输出pi值  cout.unsetf(ios::scientific); //终⽌科学记数法状态  cout.setf(ios::fixed); //指定⽤定点形式输出  cout.width(12); //指定域宽为  cout.setf(ios::showpos); //正数输出“+”号  cout.setf(ios::internal); //数符出现在左侧  cout.precision(6); //保留位⼩数  cout<<pi<<endl; //输出pi,注意数符“+”的位置  return 0;  } |

结果是：

|  |
| --- |
| dec:21  hex:0x15  oct:025  China  \*\*\*\*\*China  pi=\*\*3.142857e+00 |

|  |
| --- |
| +\*\*\*3.142857 |

对程序由几点说明：

1) 成员函数width(n)和控制符setw(n)只对其后的第一个输出项有效。如： cout. width(6);

cout <<20 <<3.14<<endl;

输出结果为 203.14

在输出第一个输出项20时，域宽为6 ，因此在20前面有4个空格，在输出 3.14时， width (6)已不起作用，此时按系统默认的域宽输出(按数据实际长度 输出)。如果要求在输出数据时都按指定的同一域宽n输出，不能只调用一次 width(n)， 而必须在输出每一项前都调用一次width(n> ，上面的程序中就是这样 做的。

2) 在表13.5中的输出格式状态分为5组，每一组中同时只能选用一种(例如 dec、 hex和oct中只能选一，它们是互相排斥的)。在用成员函数seV和 控制符 se5osﬂags设置输出格式状态后，如果想改设置为同组的另一状态，应当调用成 员函数unseV (对应于成员函数self)或 rese5osﬂags (对应于控制符 se5osﬂags)，先终止原来设置的状态。然后再设置其他状态，大家可以从本程 序中看到这点。程序在开 始虽然没有用成员函数self和控制符se5osﬂags设置用 dec输出格式状态，但系统默认指定为dec ，因此要改变为hex或oct ，也应当先 用unseV 函数终止原来设置。如果删去程序中的第7行和第10行，虽然在第8行 和第11行中用成员函数seV设置了hex和oct格式，由于未终止dec格式，因 此hex 和oct的设置均不起作用，系统依然以十进制形式输出。

同理，程序倒数第8行的unseV 函数的调用也是不可缺少的。

3) 用seV 函数设置格式状态时，可以包含两个或多个格式标志，由于这些格式 标志在ios类中被定义为枚举值，每一个格式标志以一个二进位代表，因此可以 用位或运算符“|”组合多个格式标志。如倒数第5、第6行可以用下面一行代替：

cout.setf(ios::internal | ios::showpos); //包含两个状态标志，用"|"组合

可以看到：对输出格式的控制，既可以用控制符(如例13.2) ，也可以用 cout流的有关成员函数(如例13.3) ，二者的作用是相同的。 控制符是在头文件

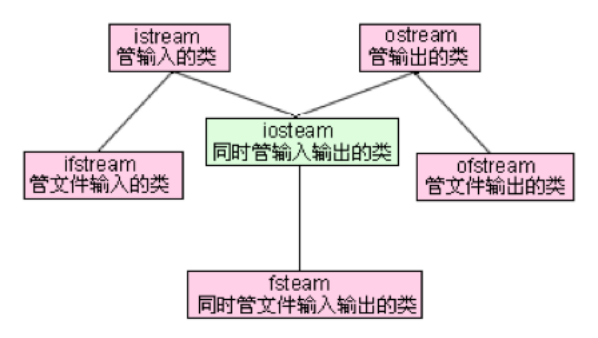
iomanip中定义的，因此用控制符时，必须包含iomanip头文件。 cout流的成员 函数是在头文件iostream 中定义的，因此只需包含头文件iostream ，不必包含 iomanip。许多程序人员感到使用控制符方便简单，可以在一个cout输出语句中 连续使用多种控制符。

10.3 文件IO

10.3.1 文件流类和文件流对象

输入输出是以系统指定的标准设备(输入设备为键盘，输出设备为显示 器)为对象的。在实际应用中，常以磁盘文件作为对象。即从磁盘文件读取数 据，将数据输出到磁盘文件。

和文件有关系的输入输出类主要在fstream.h这个头文件中被定义，在这 个头文件中主要被定义了三个类，由这三个类控制对文件的各种输入输出操 作， 他们分别是ifstream、ofstream、fstream，其中fstream类是由iostream类 派生而来，他们之间的继承关系见下图所 示。



由于文件设备并不像显示器屏幕与键盘那样是标准默认设备，所以它在 fstream.h头文件中是没有像cout那样预先定义的全局对象，所以我们必须自己 定义一个该类的对象。

ifstream类，它是从istream类派生的，用来支持从磁盘文件的输入。 ofstream类，它是从ostream类派生的，用来支持向磁盘文件的输出。 fstream类，它是从iostream类派生的，用来支持对磁盘文件的输入输出。

10.3.2 文件的打开与关闭

打开文件

所谓打开(open)文件是一种形象的说法，如同打开房门就可以进入房间活 动一样。 打开文件是指在文件读写之前做必要的准备工作，包括： 1 )为文件流对象和指定的磁盘文件建立关联，以便使文件流流向指定的磁盘 文件。 2 )指定文件的工作方式，如，该文件是作为输入文件还是输出文件，是ASCII 文件还是二进制文件等。

以上工作可以通过两种不同的方法实现。

1) 调用文件流的成员函数open。如

ofstream outfile; //定义ofstream类(输出文件流类)对象outfile

outfile.open("f1.dat",ios::out); //使文件流与f1.dat文件建立关联 第2行是调用输出文件流的成员函数open打开磁盘文件f1.dat ，并指定它为输出 文件， 文件流对象ouVile将向磁盘文件f1.dat输出数据。 ios::out是I/O模式的一 种，表示以输出方式打开一个文件。或者简单地说，此时f1.dat是一个输出文 件，接收从内存输出的数据。

调用成员函数open的一般形式为：

文件流对象.open(磁盘文件名, 输入输出方式);

磁盘文件名可以包括路径，如"c:\new\\f1.dat" ，如缺省路径，则默认为当 前目录下的文件。

2) 在定义文件流对象时指定参数

在声明文件流类时定义了带参数的构造函数，其中包含了打开磁盘文件 的功能。因此，可以在定义文件流对象时指定参数，调用文件流类的构造函数 来实现打开文件的功能。如

ostream outfile("f1.dat",ios::out); 一般多用此形式，比较方便。作用与 open函数相同。

输入输出方式是在ios类中定义的，它们是枚举常量，有多种选择，见表

13.6。



几点说明：

1) 新版本的I/O类库中不提供ios::nocreate和ios::noreplace。

2) 每一个打开的文件都有一个文件指针，该指针的初始位置由I/O方式指定， 每次读写都从文件指针的当前位置开始。每读入一个字节，指针就后移一个字 节。当文 件指针移到最后，就会遇到文件结束EOF(文件结束符也占一个字节， 其值为-1)，此时流对象的成员函数eof的值为非0值(一般设为1)，表示文件结 束 了。

3) 可以用“位或”运算符“|”对输入输出方式进行组合，如表13.6中最后3行 所示那样。还可以举出下面一些例子：

ios::in | ios:: noreplace //打开一个输入文件，若文件不存在则返回打开失 败的信息

ios::app | ios::nocreate //打开一个输出文件，在文件尾接着写数据，若文

件不存在，则返回打开失败的信息

ios::out l ios::noreplace //打开一个新文件作为输出文件，如果文件已存 在则返回打开失败的信息

ios::in l ios::out I ios::binary //打开一个二进制文件，可读可写 但不能组合互相排斥的方式，如 ios::nocreate l ios::noreplace。

4) 如果打开操作失败，open函数的返回值为0(假)，如果是用调用构造函数的 方式打开文件的，则流对象的值为0。可以据此测试打开是否成功。如

if(outfile.open("f1.bat", ios::app) ==0)

cout <<"open error";

或

if( !outfile.open("f1.bat", ios::app) )

cout <<"open error";

关闭文件

在对已打开的磁盘文件的读写操作完成后，应关闭该文件。关闭文件用 成员函数close。如

ouVile.close( ); //将输出文件流所关联的磁盘文件关闭

所谓关闭，实际上是解除该磁盘文件与文件流的关联，原来设置的工作方式也 失效，这样，就不能再通过文件流对该文件进行输入或输出。 此时可以将文件

流与其他磁盘文件建立关联，通过文件流对新的文件进行输入或输出。如 ouVile.open("f2.dat",ios::app|ios::nocreate);

此时文件流ouVile与f2.dat建立关联，并指定了f2.dat的工作方式。

10.3.3 C++对ASCII文件的读写操作

如果文件的每一个字节中均以ASCII代码形式存放数据,即一个字节存放一 个字符,这个文件就是ASCII文件(或称字符文件)。程序可以从ASCII文件中读入若 干个字符,也可以向它输出一些字符。

1) 用流插入运算符“<<”和流提取运算符“>>”输入输出标准类型的数据。 “<<” 和“>>”都巳在iostream中被重载为能用于ostream和istream类对象的标准类型的 输入输出。由于ifstream和 ofstream分别是ostream和istream类的派生类；因此 它们从ostream和istream类继承了公用的重载函数，所以在对磁盘文件的操作 中，可以通过文件流对象和流插入运算符“<<”及 流提取运算符“>>”实现对磁盘 文件的读写，如同用cin、cout和<<、>>对标准设备进行读写一样。

2) 用文件流的put、get、geiline等成员函数进行字符的输入输出，：用C++ 流成员函数put输出单个字符、 C++ get()函数读入一个字符和C++getline()函数读 入一行字符。

|  |
| --- |
| #include <iostream>  #include <fstream>  using namespace std;  int main(void)  {  char\* fname = "c:/aaaa.txt";  ofstream fout(fname, ios::app); //建⼀个 输出流对象 和⽂件关联; if (!fout)  {  cout << "打开⽂件失败" << endl;  return ;  }  fout << "hello....111" << endl;  fout << "hello....222" << endl;  fout << "hello....333" << endl;  fout.close();  //读⽂件  ifstream fin(fname); //建⽴⼀个输⼊流对象 和⽂件关联  char ch;  while (fin.get(ch))  {  cout <<ch ;  }  fin.close();  return 0;  } |

10.3.4 C++对二进制文件的读写操作

|  |
| --- |
| #include <iostream>  #include <fstream>  using namespace std;  class Teacher  {  public:  Teacher()  {  age = 33;  strcpy(name, "");  }  Teacher(int \_age,const char \*\_name)  {  age = \_age;  strcpy(name, \_name);  }  void printT()  {  cout << "age:" << age << "name:" << name <<endl;  }  protected :  private :  int age;  char name[32];  };  int main()  {  const char\* fname = "./11a.dat";  ofstream fout(fname, ios::binary); //建⼀个 输出流对象 和⽂件关联; if (!fout)  {  cout << "打开⽂件失败" << endl;  return -1;  }  Teacher t1(31, "t31");  Teacher t2(32, "t32");  fout.write((char \*)&t1, sizeof(Teacher));  fout.write((char \*)&t2, sizeof(Teacher));  fout.close();  //  ifstream fin(fname); //建⽴⼀个输⼊流对象 和⽂件关联  Teacher tmp;  fin.read( (char\*)&tmp,sizeof(Teacher) );  tmp.printT(); |

|  |
| --- |
| fin.read( (char\*)&tmp,sizeof(Teacher) );  tmp.printT();  fin.close();  return 0;  } |

10.3.5 作业及参考答案

1 编程实现以下数据输入/输出：

(1)以左对齐方式输出整数,域宽为12。

(2)以八进制、十进制、十六进制输入/输出整数。

(3)实现浮点数的指数格式和定点格式的输入/输出,并指定精度。

(4)把字符串读入字符型数组变量中,从键盘输入,要求输入串的空格也全部读

入,以回车符结束。

(5)将以上要求用流成员函数和操作符各做一遍。 2编写一程序，将两个文件合并成一个文件。 3编写一程序，统计一篇英文文章中单词的个数与行数。 4编写一程序，将C++源程序每行前加上行号与一个空格。 5编写一程序，输出 ASCII码值从20到127的ASCII码字符表，格式为每行10个。

参考答案

第一题：

|  |
| --- |
| //ios类成员函数实现  #include<iostream>  #include<iomanip>  using namespace std;  int main()  {  long a=234;  double b=2345.67890;  char c[100];  cout.fill( '\*');  cout.flags(ios\_base::left);  cout.width(12);  cout<<a<<endl;  cout.fill( '\*');  cout.flags(ios::right);  cout.width(12);  209 |

|  |
| --- |
| cout<<a<<endl;  cout.flags(ios::hex);  cout<<234<< '\t';  cout.flags(ios::dec);  cout<<234<< '\t';  cout.flags(ios::oct);  cout<<234<<endl;  cout.flags(ios::scientific);  cout<<b<< '\t';  cout.flags(ios::fixed);  cout<<b<<endl;  cin.get(c,99);  cout<<c<<endl;  return 0;  }  //操作符实现  #include<iostream>  #include<iomanip>  using namespace std;  int main()  {  long a=234;  double b=2345.67890;  char c[100];  cout<<setfill( '\*');  cout<<left<<setw(12)<<a<<endl;  cout<<right<<setw(12)<<a<<endl;  cout<<hex<<a<< '\t'<<dec<<a<< '\t'<<oct<<a<<endl;  cout<<scientific<<b<< '\t'<<fixed<<b<<endl;  return 0;  } |

第二题：

|  |
| --- |
| #include<iostream>  #include<fstream>  using namespace std;  int main()  {  int i=1;  char c[1000];  ifstream ifile1("D:\\1.cpp");  ifstream ifile2("D:\\2.cpp");  ofstream ofile("D:\\3.cpp");  210 |

211

|  |
| --- |
| while(!ifile1.eof()){  ifile1.getline(c,999);  ofile<<c<<endl;  }  while(!ifile2.eof()){  ifile2.getline(c,999);  ofile<<c<<endl;  }  ifile1.close();  ifile2.close();  ofile.close();  return 0;  } |

第三题：

|  |
| --- |
| #include<iostream>  #include<fstream>  using namespace std;  bool isalph(char);  int main()  {  ifstream ifile("C:\\daily.doc");  char text[1000];  bool inword=false;  int rows=0,words=0;  int i;  while(!ifile.eof()) {  ifile.getline(text,999);  rows++;  i=0;  while(text[i]!=0){  if(!isalph(text[i]))  inword=false;  else if(isalph(text[i]) && inword==false){  words++;  inword=true;  }  i++;  }  }  cout<<"rows= " << rows <<endl;  cout<<"words= "<< words <<endl; |

212

|  |
| --- |
| ifile.close();  return 0;  }  bool isalph(char c)  {  return ((c>= 'A' && c<= 'Z') || (c>= 'a' && c<='z'));  } |

第四题：

|  |
| --- |
| #include<iostream>  #include<fstream>  using namespace std;  int main()  {  int i=1;  char c[1000];  ifstream ifile("D:\\1.cpp");  ofstream ofile("D:\\2.cpp");  while(!ifile.eof()){  ofile<<i++<<": ";  ifile.getline(c,999);  ofile<<c<<endl;  }  ifile.close();  ofile.close();  return 0;  } |

第五题：

|  |
| --- |
| #include<iostream>  using namespace std;  int main()  {  int i,l;  for(i=32;i<127;i++){  cout<<char(i)<<" ";  l++;  if(l%10==0)cout<<endl;  }  cout<<endl; |

|  |
| --- |
| return 0;  } |

附录A

