

C++基础

C++代码编译运行

从C到C++

const限定符

const和#define的区别

运算符

作用域标识符：:

new和delete运算符

函数

函数重载

带默认形参值的函数-可能产生二义性

函数的参数传递

引用

const引用

引用作为函数返回值

内联函数、带参数宏、类型转换运算符

内联函数：见知识点辨析

类型转换

const_cast

static_cast

reinterpret_cast

dynamic_cast,向下转型

类与对象

面向对象介绍

面向对象的基本特征

类的声明

数据抽象的编程思想和封装

成员函数

内联成员函数

成员函数的重载及缺省参数

隐含的this指针

类作用域

嵌套类

局部类

构造函数

构造函数和new运算符

转换构造函数

构造函数初始化列表

对象成员及其初始化：见Effective C++

const成员、引用成员初始化

枚举 enum

拷贝构造函数

深拷贝与浅拷贝

禁止拷贝
空类默认产生的成员
析构函数
类与类之间的关系：StarUML绘制

对象的使用
对象的两种语义
static成员
static成员函数【见知识点辨析】
类、对象大小计算
四种对象的作用域与生存期
static与单例模式
const成员函数
const对象

数据抽象与封装

友元
友元介绍
友元函数
友元类

运算符重载
运算符重载
成员函数重载
友元函数重载
重载的规则
++运算符重载
! 运算符重载
字符串类的[]运算符重载
+运算符的重载：二元运算符通过友元函数进行重载
流运算符重载
类型转换运算符重载
指针访问运算符 `->` 重载
operator new和operator delete重载

标准库类型
标准库string类型
map容器使用

继承
代码重用
继承的语法
继承的级别
接口继承和实现继承
继承与重定义
继承于构造函数
拷贝构造函数的实现：应该逐成员赋值
静态成员与继承
转换与继承
将基类对象转换为派生类对象的实现（不推荐这种不安全的转型操作）

多重继承

虚继承和虚基类：用于钻石继承

虚基类及其派生类构造函数

虚继承对C++对象内存模型造成的影响

多态

静态绑定与动态绑定

虚函数

虚析构函数

object slicing与虚函数

纯虚函数

对象的动态创建

runtime type information (RTTI) : 都不如虚函数的虚函数表效率来得高

异常

C语言错误处理方法

程序错误

异常的语法

异常抛出：

异常的捕获

异常的传播

栈展开

异常与继承

I/O流类库

STL

模板

函数模板

 函数模板特化

类模板

 类模板定义

 类模板的使用

非类型模板参数

缺省模板参数

成员模板

关键字typename

派生类和模板

面向对象与泛型

模板的应用

 用模板实现单例模式

小结

泛型程序设计 (generic programming)

STL的六大组件

容器

适配器

函数对象

分配器allocator

 内存池设计实现

算法

迭代器

源码分析

vector源码分析

vector的push_back所做的工作

vector的capacity和size

vector元素的删除erase和remove

小结

迭代器

迭代器类型

算法

非变动性算法

变动性算法

已序区间算法

数值算法

算法尾词

函数对象

函数对象

函数对象与容器

函数对象与算法

STL中内置的函数对象

适配器

容器适配器

函数适配器：用于将函数适配成函数对象，用于algorithm

针对成员函数的适配器 `mem_fun`

针对一般函数的函数适配器 `ptr_fun`

`not1`

反向迭代器适配器`rbegin`, `rend`

插入迭代器

IO流迭代器

STL中适配器的实现

小结

容器的对比

迭代器

算法示例

用STL算法解决八皇后问题

`inserter`、`back_inserter`和`front_inserter`

函数适配器分类

小项目

面向对象计算器设计【未】

面向泛型版的计算器实现

银行储蓄系统

MFC框架

C++新特性

智能指针

右值引用?

function/bind (基于对象编程)

作用：解决的问题

std::function

std::bind

lambda表达式

MySQL

关系数据库

拓展总结

boost智能指针

scoped_ptr<T>

shared_ptr<T> (线程安全的)

循环引用

weak_ptr

scoped_array和shared_array

单例模式和auto_ptr

单例模式

auto_ptr在单例模式中的应用

muduo中单例模式实现

ThreadLocalSingleton模板类实现

Noncopyable实现

用宏实现sizeof的功能

编程技巧

工厂模式

常量是否可以更改?

使用PIMPL

知识点辨析

重载 (overload) 、重写 (重定义overwrite) 、覆盖 (override)

组合和继承

只能在构造函数的参数列表初始化的情况下

构造函数的调用顺序

泛型程序设计

拓展阅读

1. C++基础

□ C++支持的编程范式(paradigm)

- ▢ □ 过程式(procedural)
- ▢ □ 数据抽象(data abstraction)
- ▢ □ 基于对象(object-based)
- ▢ □ 面向对象式(object-oriented)
- ▢ □ 函数式(functional)
- ▢ □ 泛型形式
- ▢ □ 模板元形式

ADT

封装
封装
继承、多态

□ 值语义与对象语义

- ▢ 值语义可以拷贝与赋值、对象语义不可进行拷贝与赋值

□ function/bind的救赎（上）、这篇文章对编程范式进行一些探讨

▢ <http://blog.csdn.net/myan/article/details/5928531>

数据抽象：针对数据结构而言

基于对象编程：通过function/bind来实现（muduo库构建）

面向对象编程：程序由对象+对象+对象+消息传递构成

面向泛型编程：程序由对象+对象+抽象行为（能够施加在不同类型而又大相径庭的对象之上，**STL**）

模板元编程：

- 给出代码的产生规则，让编译器根据模板元产生新代码，实现我们预期的功能
- 让某些运行时的工作被提前到编译器来完成，从而增加了编译时间。但是提高了运行效率
- 可以实现神奇的类型推导（见STL源码剖析）

1.1 C++代码编译运行

- 通过 `gcc/g++ -c 源文件` 生成 `.obj` 对象文件
- 然后通过 `gcc/g++ x1.obj x2.obj` 链接生成可执行文件 `.exe` 或 `.out`

windows下编写的文件，可以在linux系统下使用。但是需要执行相应的编译链接命令。通常利用 `MakeFiles` 或者 `CMake` 实现

1.2 从C到C++

在C语言中没有bool类型，可以用int来替换

1.2.1 const限定符

```
int main(void)
{
    //const int a;           Error, 常量必须初始化
    const int a = 100;
    //a = 200;              Error, 常量不能重新被赋值

    int b = 22;
    const int * p;          //const在*左边, 表示*p为常量, 经由*p不能更改指针所指向的内容
    p = &b;
    /*p = 200;             Error, 常量不能重新被赋值
    int * const p2;         Error, p2为常量, 常量必须初始化
    int * const p2 = &b;     //const在*右边, 表示p2为常量
    //int c =100;
    //p2 = &c;
    *p2 = 200;              Error, 常量不能重新被赋值

    cout<<b<<endl;

    return 0;
}
```

- 通过const来定义一个常量，因此必须初始化（常量必须初始化，同理，引用也必须初始化）
- 常变量在初始化之后，不能再被赋值

1.2.2 const和#define的区别

C++
教程网

const与#define

- const定义的常量与#define定义的符号常量的区别
 - const定义的常量有类型，而#define定义的没有类型，编译可以对前者进行类型安全检查，而后者仅仅只是做简单替换
 - const定义的常量在编译时分配内存，而#define定义的常量是在预编译时进行替换，不分配内存。
 - 作用域不同，const定义的常变量的作用域为该变量的作用域范围。而#define定义的常量作用域为它的定义点到程序结束，当然也可以在某个地方用#undef取消
- 定义常量还可以用enum，尽量用const、enum替换#define定义常量。

高层次编程：const, enum, inline定义常量

而底层编程，则更多的是使用#define定义常量，宏具有灵活性

```

#include <iostream>
using namespace std;

#define STR(a) #a
#define CAT(a, b) a##b

int main(void)
{
    int xy = 100;
    cout<<STR(ABCD)<<endl;           // #ABCD <=> "ABCD"
    cout<<CAT(x, y)<<endl;           // x##y <=> xy 100
    return 0;
}

```

教程网

const与#define

- ❑ #define 定义的常量，容易产生副作用。

```

//Effective C++ 3rd的一个例子。
#define CALL_WITH_MAX(a, b) f((a) > (b) ? (a) : (b))

int a = 5;
int b = 0;
CALL_WITH_MAX(++a, b);           // a被累加二次
CALL_WITH_MAX(++a, b+10);        // a被累加一次

```

6 0 7
6 10 10

在这里，调用f之前，a的递增次数竟然取决于“它被拿来和谁比较”

- ❑ 定义常量还可以用enum，尽量用const、enum替换#define 定义常量。effectiveC++

1.3 运算符

1.3.1 作用域标识符：：

- 用于同局部变量同名的全局变量进行访问

```

1 #include <iostream>
2
3 using namespace std;
4
5 int var = 100;
6
7 int main(void)
8 {

```

```

9     int var = 50;
10    cout << var << endl; // 50, 局部变量隐藏了全局变量
11    cout << ::var << endl; // 100, 利用作用域::访问全局
12    // 变量
13
14 }

```

- 用于表示类的成员

1.3.2 new和delete运算符

1. new运算符可以用于创建堆空间

- 成功时返回首地址
- 而失败时抛出 `bad_alloc` 异常

2. 语法

- 指针变量 = `new` 数据类型
- 指针变量 = `new` 数据类型[长度n]

```

1 int * p = new int;
2 char *pStr = new char[50];

```

3. new和delete需要配对, new [] 和delete[] 配对

new operator	malloc只进行内存分配
<input checked="" type="checkbox"/> new 一个新对象 <input checked="" type="checkbox"/> 内存分配(operator new) <input type="checkbox"/> 调用构造函数	<input type="checkbox"/> delete 释放一个对象 <input type="checkbox"/> 调用析构函数 <input type="checkbox"/> 释放内存(operator delete)

new运算符的作用分类

- `new operator`: 分配内存+调用构造函数
- `operator new`: 只分配内存

- placement new: 不分配内存, 调用拷贝构造函数

1.4 函数

1.4.1 函数重载

C++ 教程网 **重载**

- 相同的作用域, 如果两个函数名称相同, 而参数不同, 我们把它们称为重载overload
- 函数重载又称为函数的多态性 **静态多态**
- 函数重载不同形式:
 - 形参数量不同
 - 形参类型不同
 - 形参的顺序不同
 - 形参数量和形参类型都不同
- 调用重载函数时, 编译器通过检查实际参数的个数、类型和顺序来确定相应的被调用函数

不能仅根据返回类型来定义函数的重载

为了支持重载, 编译器会对重载的函数名进行**名字改编**

C++ 教程网 name mangling 与 extern “C”

- name mangling这里把它翻译为名字改编。
- C++为了支持重载, 需要进行name mangling
- extern “C”实现C与C++混合编程

```
#ifdef __cplusplus
extern "C"
{
#endif
...
#ifndef __cplusplus
}
```

为了让C++写的函数能够在C语言中使用
 1. 需要extern “C”让编译器不进行名字改编
 2. 但是在C语言中, 则不需要定义extern "C"
 3. 所以, 应该定义宏来保证, 是C++头文件的时候, 才定义相关的extern "C"

1.4.2 带默认形参值的函数-可能产生二义性

- 函数没有声明时，在函数定义中指定形参的默认值
- 函数既有定义又有声明时，声明时指定后，定义后就不能再指定默认值
- 默认值的定义必须遵守从右到左的顺序，如果某个形参没有默认值，则它左边的参数就不能有默认值。
 - void func1(int a, double b=4.5, int c=3); //合法
 - void func1(int a=1, double b, int c=3); //不合法
- 函数调用时，实参与形参按从左到右的顺序进行匹配

C++ 教程网 带默认形参值的函数的二义性

- 重载的函数中如果形参带有默认值时，可能产生二义性

```
int add(int x=5, int y=6); ①  
int add(int x=5, int y=6, int z=7);  
int main() {  
    int sum;  
    sum= add(10,20);  
    return 0;  
}
```

```
int add(int x, int y)  
{  
    return x+y;  
}  
int add(int x, int y, int z)  
{  
    return x+y;  
}
```

sum=add(10, 20)语句产生二义性，可以认为该语句是调用第一个函数，也可以是第二个，因此编译器不能确定调用的是哪一个函数。

1.4.3 函数的参数传递

1. 值传递
2. 引用传递：形参初始化的时候，不需要分配内存空间
3. 指针传递：本质也是值传递，实参在初始化形参的时候，也要分配内存空间，分配的时候4字节的空间。如果要修改指针的地址，单纯的用指针传递无法实现，必须使用指针的指针

值传递和引用传递的区别

- 值传递得到的是实参的副本，形参的改变无法影响到实参。
- 引用传递后，形参是实参的别名，和实参共享一块内存。可以通过形参的修改实现对实参的修改。
- 对于自定义数据类型，在进行参数传递的时候，如果采用值传递，那么会涉及到相应的拷贝构造操作。因此，建议使用引用传递

1.5 引用

变量的属性

- 名称
- 空间

引用的特性：

- 引用不是变量，只是所引用变量的别名
- 没有自己的独立空间
- 与所引用的变量共享内存空间
- 对引用所做的改变实际上是对他引用的变量的改变
- **引用一定要初始化**
- 引用一经初始化，不能重新指向其他变量（**指向其他变量，只是获得该变量的值**）

引用的作用：

1. 作为函数的参数传入
2. 作为函数的返回值

1.5.1 const引用

定义：对一个const对象的引用，即不能通过引用修改该对象的值

- 无法对一个const对象进行普通引用，因为可能存在通过该普通引用修改常量的风险
- 可以通过 `const reference to a nonconst obj`

```

#include <iostream>
using namespace std;

//const引用是指向const对象的引用

int main(void)
{
    const int val = 1024;
    const int& refval = val;

    //int& ref2 = val; // Error, nonconst reference to const object

    //refval = 200;    Error, refval是一个常量

    int val2 = 1024;
    const int& ref3 = val2; //const reference to nonconst object

    double val3 = 3.14;
    const int& ref4 = val3; // int temp = val3;
                           // const int& ref4 = temp;

    cout<<"ref4="<<ref4<<endl;
    cout<<"val3="<<val3<<endl;

    //int& ref5 = val3; Error

    return 0;
}

```

这样可行的原因是：
可以对临时变量进行
常量引用

1.5.2 引用作为函数返回值

函数返回引用的一个主要目的：为了可以将函数放在赋值运算符的左边

- 前置递增递减运算符重载返回值
- 这时候的引用在函数返回时初始化，初始化为函数体内部的被引用变量
- 不能返回对局部变量的引用**

```

#include <iostream>
using namespace std;

// 引用作为函数返回值

int& add(int a, int b)
{
    int sum;
    sum = a + b;
    return sum;
}

int main(void)
{
    int n = add(3, 4);
    int& n2 = add(5, 6); // n2是引用，没有自己独立的空间
                          // n2的值依赖于它所引用的变量
                          // 如果n2所引用的变量的生命期结束了，也就是说n2是一个
                          // 无效的引用，那么n2的值将是不确定的。
    cout<<"n2="<<n2<<endl; 这时候，sum的缓存还在
    cout<<"n="<<n<<endl;
    cout<<"n2="<<n2<<endl;
    return 0;
}

```

```

1 #include <iostream>
2
3 using namespace std;

```

```
4  
5 // 引用作为函数返回值  
6 int a[] = {0,1,2,3,4,5};  
7  
8 int& index(int i)  
9 {  
10     return a[i];  
11 }  
12  
13 int main(void)  
14 {  
15     index(3) = 100; // a[3] = 100;  
16     // 引用作为函数的返回值，使得函数可以放在赋值运算符左边  
17     // 函数返回引用，引用在函数返回的时候初始化  
18  
19     return 0;  
20 }
```

1.6 内联函数、带参数宏、类型转换运算符

宏 `#define`

常量
带参数的宏（类似于函数调用） `const enum`
`inline`

C++高层次编程 推荐用`const`、`enum`、`inline`替换宏

低层次编程 宏是很灵活。

1.6.1 内联函数：见知识点辨析

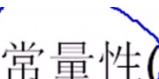
1.6.2 类型转换

隐式转换：编译器可以自动完成的（一般来说是安全的）

显示转换：`*_cast`

- 旧式转型
 - (T)expr
 - T(expr)
- 新式转型+
 - const_cast<T>(expr)
 - static_cast<T>(expr)
 - reinterpret_cast<T>(expr)
 - dynamic_cast<T>(expr)  在派生类与基类之间使用
 - 执行“安全向下”转型操作，也就是说支持运行时识别指针或所指向的对象，这是唯一一个无法用旧式语来进行的转型操作。

1.6.2.1 const_cast

- 用来移除对象的常量性  (cast away the constness)
- const_cast 一般用于指针或者引用 
- 使用 const_cast 去除 const 限定的目的不是为了修改它的内容
- 使用 const_cast 去除 const 限定，通常是为了函数能够接受这个实际参数

```

1 #include <iostream>
2
3 using namespace std;
4
5 // cast_cast用来移除常量性
6 // const_cast一般用于指针或者引用
7 void func(int &val)
8 {
9     cout << "val = " << val << endl;
10 }
11 int main(void)
12 {
13     const int val = 100;
14     int n = val;
15
16     // int *p = &val; error, 无法从const int* 转换为int*

```

```
17     int *p = const_cast<int*>(&val); // 正确
18     *p = 200; //
19
20     cout << *p << endl; // 200; 更改的是临时对象的值
21
22     // 引用测试
23     const int val2 = 200;
24     int& refval2 = const_cast<int&>(val2); // 目的不应该
25     是用来修改其引用对象的内容
26
27     func(const_cast<int&>(val2)); // 为的是让函数能够接受这
28     个实参
29 }
```

1.6.2.2 static_cast

- 编译器隐式执行的任何类型转换都可以由 **static_cast** 完成
- 当一个较大的算术类型赋值给较小的类型时，可以用 **static_cast** 进行强制转换。
- 可以将 **void*** 指针转换为某一类型的指针
- 可以将基类指针指向派生类指针
- 无法将 **const** 转化为 **nonconst**，这个只有 **const_cast** 才可以办得到

1.6.2.3 reinterpret_cast

❑ reinterpret_cast “通常为操作数的位模式提供较低层的重新解释”也就是说将数据以二进制存在形式的重新解释。

```
int i;  
char *p = "This is a example.";  
i = reinterpret_cast<int>(p);  
//此时结果, i与p的值是完全相同的。  
  
int *ip  
char *pc = reinterpret_cast<char*>(ip);  
//程序员需要记得pc所指向的真实对象是int型, 并非字符串。  
//如果将pc当作字符指针进行操作, 可能会造成运行时错误  
//如int len = strlen(pc);
```

1.6.2.4 dynamic_cast, 向下转型

1.7 类与对象

编写程序 是为了让计算机 解决现实生活中的实际问题

用一定的数据来表示
按一定的逻辑来处理这些数据

程序的定义

pascal之父 结构化程序设计先驱 迪杰斯特拉

程序 = 算法 + 数据结构

程序是完成一定功能的一系列有序指令的集合

指令 = 操作码 + 操作数

按一定的逻辑来处理这些数据

将指令按一定的顺序进行整合 就形成了程序

结构化程序设计的思想：

- 采用自顶向下，将系统视为分层的子程序的集合
- 可以将程序分为功能不同的模块，使得整个程序更有条理性

- 但是很多数据仍然属于整个程序，因而结构化程序设计思想还是需要很多的全局变量。在某个地方进行更改，所以会对整个程序产生难以预料的影响

结构化程序设计的缺点

- 结构化程序设计为处理复杂问题提供了有力手段，但到80年代末，这种设计方法逐渐暴露出以下缺陷：
 - 程序难以管理
 - 数据修改存在问题
 - 程序可重用性差
 - 用户要求难以在系统分析阶段准确定义，致使系统在交付使用时产生许多问题。
 - 用系统开发每个阶段的成果来进行控制，不能适应事物变化的要求。
- 面向过程程序设计缺点的根源在于数据与数据处理分离

面向对象程序设计：将系统看成通过交互作用来完成特定功能的对象集合。每个对象用自己的方法管理数据。

也就是说，只有对象内部的代码能够操作对象内部的数据

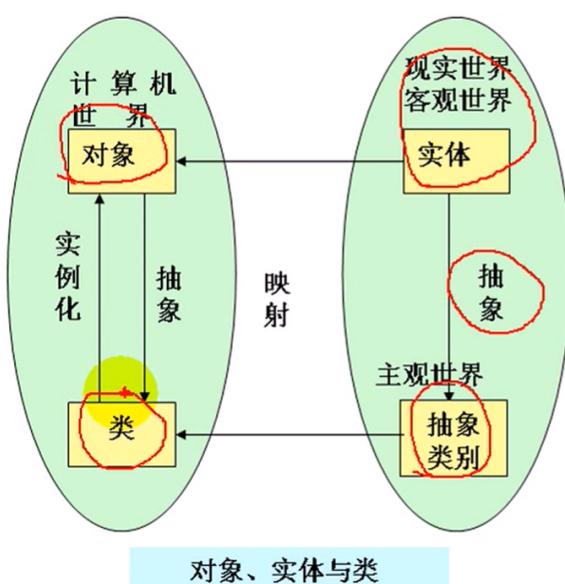
程序 = 对象 + 对象 + 对象 + 信息交互

对象 = 算法 + 数据结构 (封装的思想)

1.7.1 面向对象介绍

- 面向对象 (Object Oriented) 是认识事务的一种方法，是一种以对象为中心的思维方式
- 面向对象的程序设计：
 - 对象 = (算法 + 数据结构) 结合
 - 程序 = 对象 + 对象 + + 对象 +
- 面向对象程序设计模拟自然界认识和处理事物的方法，将数据和对数据的操作方法放在一起，形成一个相对独立的整体——对象 (object)，同类对象还可抽象出共性，形成类 (class)。一个类中的数据通常只能通过本类提供的方法进行处理，这些方法成为该类与外部的接口。对象之间通过消息 (message) 进行通讯。

实体、对象、类之间的关系



现实世界中的实体可以抽象出类别的概念。对应于计算机世界就有一个类 (class) 的概念，因为类是一个抽象的概念的对应体，所以计算机不给它分配内存，只给对象分配内存。左图表达了计算机世界与现实世界之间的对应关系。

- 对象是计算机内存中的一块区域。通过内存分块每个对象在功能上相对保持独立。
- 这些内存不但存储数据，也存储代码。这保证对象是受保护的，只有对象中的代码能访问存储于对象中的数据。这清楚地限定了对象所具有的功能，并且使得对象不受未知外部事件的影响，从而使自己的数据和功能不会因此遭受破坏。
- 对象之间只能通过函数调用也就是发送消息来实现相互通信。
- 当对象的一个函数被调用时，对象执行内部的代码来响应该调用，从而使对象呈现一定的行为。这个行为及其呈现出来的结果就是该对象所具有的功能。

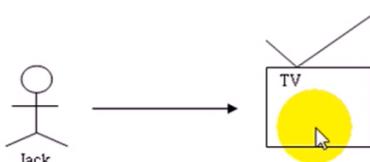
1.7.2 面向对象的基本特征

1. **抽象**: 抓住事物本质，而不是内部具体细节或具体实现。

- 从具体到一般的过程，归纳，归类
- 对象->类（归类）
- 大类->小类（分类）

2. **封装**: 隐藏内部细节，只暴露需要对外展示的接口

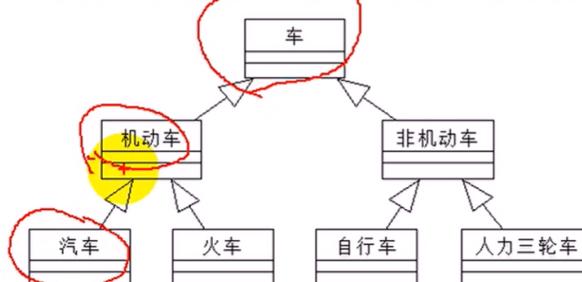
- 封装是指按照信息屏蔽的原则，把对象的属性和操作结合在一起，构成一个独立的对象。
- 通过限制对属性和操作的访问权限，可以将属性“隐藏”在对象内部，对外提供一定的接口，在对象之外只能通过接口对对象进行操作。
- 封装性增加了对象的独立性，从而保证了数据的可靠性。
- 外部对象不能直接操作对象的属性，只能使用对象提供的服务。



我们不用关心电视机的内部工作原理，
电视机提供了选台、调节音量等功能让我们使用。

3. **继承**:

- 继承表达了对象的一般与特殊的关系。特殊类的对象具有一般类的全部属性和服务。
- 当定义了一个类后，又需定义一个新类，这个新类与原来的类相比，只是增加或修改了部分属性和操作，这时可以用原来的类派生出新类，**新类中只需描述自己所特有的属性和操作。**
- 继承性大大简化了对问题的描述，**大大提高了程序的可重用性**，从而提高了程序设计、修改、扩充的效率。



继承具有传递性，如汽车具有车的全部属性和行为。

4. 多态：使得我们能够以一致的观点看待不同（但又大相径庭）的对象

同一个接口被不同的对象调用时，产生不同的结果。

C++
教程网

继承与多态

- 继承和多态性组合，可以生成很多相似但又独一无二的对象。继承性使得这些对象可以共享许多相似特性，而多态又使同一个操作对不同对象产生不同表现形式。这样不仅提高了程序设计的灵活性，而且减轻了分别设计的负担。

开闭原则：对增加功能开放、对修改、删除关闭。使得使用该对象的原有系统不会发生改变。

1.7.3 类的声明

成员变量的命名规范

```

1 int hour_;
2 int m_hour; // 微软命名规范
  
```

类的三种访问权限

- 在关键字**public**后面声明，它们是类与外部的接口，任何外部函数都可以访问公有类型数据和函数。
- 在关键字**private**后面声明，只允许本类中的函数访问，而类外部的任何函数都不能访问。
- 在关键字**protected**后面声明，与**private**类似，其差别表现在继承与派生时对派生类的影响不同

protected的对象继承后变为private，而private属性直接被隐藏不可访问

1.7.4 数据抽象的编程思想和封装

- **数据抽象**是一种依赖于接口和实现分离的编程（和设计）技术。**类设计者必须关心类是如何实现的**，但**使用该类的程序员不必了解这些细节**。使用者只要抽象地考虑该类型做什么，而不必具体地考虑该类如何工作。
- **封装**是一项将低层次的元素组合起来形成新的、高层次的实体的技术。**函数是封装的一种形式**：函数所执行的细节行为被封装在函数这个更大的实体中。被封装的元素隐藏了它们的实现细节——可以调用函数，但是不能直接访问函数所执行的语句。同样地，**类也是一个封装的实体**：它代表若干成员的聚集，**设计良好的类隐藏了类实现的细节**。

```
class Stack
{
public:
    void Push(int elem);
    int Pop();

private:
};
```

类设计者

类的使用者来说不需要关注类内部是怎么实现，也不需要关注类内部所采用的数据
他只需要关注这个类能提供什么功能。只要这个类所提供的功能不变，也就是类所
暴露的接口不变稳定

那么对类使用者所编写的代码的代码就是稳定的。

数组 链表

并非给客户越多的东西就是越好的服务
相反，仅仅给他需要关注的东西才是最好的服务

I

这样子能防止粗心的程序员破坏类内部的数据结构。|

1.7.5 成员函数

1.7.5.1 内联成员函数

内联函数的优点：

- 提高效率
- 在编译的时候将代码直接嵌入到调用的地方，从而减少了函数调用的开销

缺点：体积增大，以空间换时间的做法。

内联函数仅仅是给编译器一个提示，如果函数中有switch和for等循环分支语句，那么编译器可能不会进行内联

内联函数的定义

1. 直接在类体中给出实现
2. 在定义的时候，添加关键字 `inline`

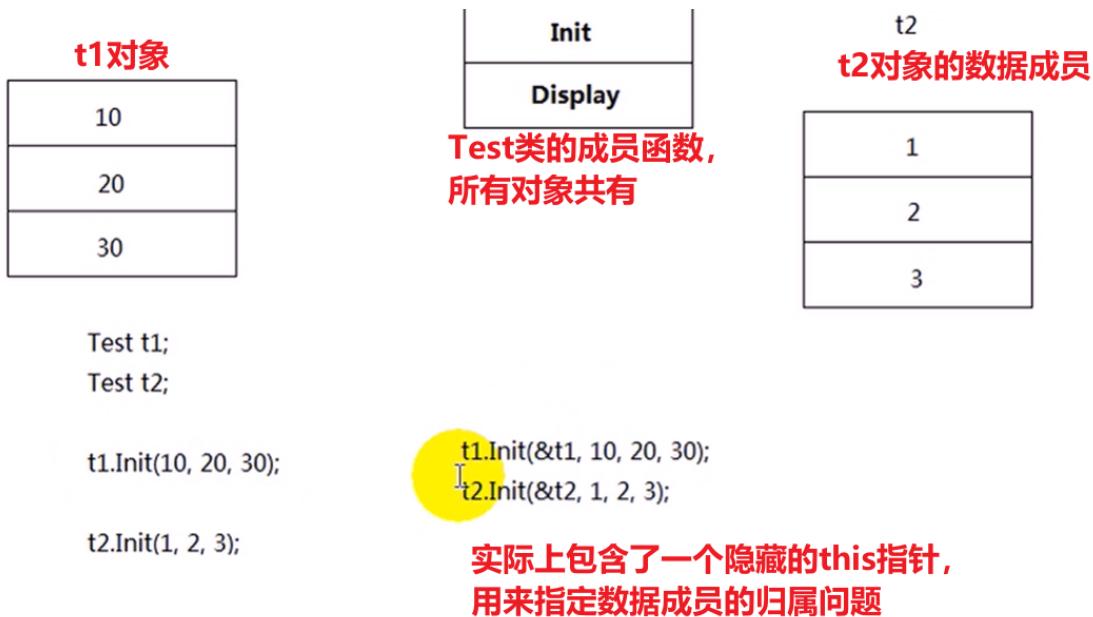
1.7.5.2 成员函数的重载及缺省参数

缺省参数的时候需要注意 `二义性问题`

1.7.5.3 隐含的this指针

- 成员函数有一个隐含的附加形参, 即指向该对象的指针, 这个隐含的形参叫做this指针
- 使用this指针保证了每个对象可以拥有不同的数据成员, 但处理这些成员的代码可以被所有对象共
享

成员函数是共有的



1.7.6 类作用域

- 每个类都定义了自己的作用域称为类作用域
- 类作用域中说明的标识符只在类中可见。

- 1、块作用域
- 2、文件作用域
- 3、函数原型作用域
- 4、函数作用域
- 5、类作用域

前向声明:

- C++中类必须先定义，才能够实例化
- 两个类需要相互引用形成一个**环形引用时**，无法先定义使用，这时候需要使用到前向声明
- **前向声明的类不能实例化**，即不能够定义对象，只能够定义指针或者引用
- 作为类成员参数，也只能是指针或者引用

环形引用：

1.7.7 嵌套类

外部类需要使用嵌套类对象作为底层实现，并且该嵌套类只用于外围类的实现。且同时可以对用户隐藏该底层实现

代码示例

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Outer
6 {
7 public:
8     class Inner
9     {
10    public:
11        void func()
12        {
13            cout << "Inner func()." << endl;
14        }
15        void testFunc();
16    }
17 public:
18     Inner obj_;
19     void fun()
20     {
21         cout << "Outer func()." << endl;
22         obj_.func(); // "Inner func()."
23     }
24 };
25 // 嵌套类成员函数的类外实现
26 void Outer::Inner::testFunc(){
27     cout << "Outer::Inner::testFunc()";
28 }
```

```
29  
30 int main(void)  
31 {  
32     Outer::Inner i;  
33     i.func(); // 在外部也能够使用嵌套类的功能  
34  
35 }
```

注意事项：

- 从作用域的角度看，嵌套类被隐藏在外围类之中，该类名只能在外围类中使用。如果在外围类的作用域使用该类名时，需要加名字限定。
- 嵌套类中的成员函数可以在它的类体外定义。
- 嵌套类的成员函数对外围类的成员没有访问权，反之亦然。
- 嵌套类仅仅只是语法上的嵌入

1.7.8 局部类

- 类也可以定义在函数体内，这样的类被称为局部类（local class）。局部类只在定义它的局部域内可见。
- 局部类的成员函数必须被定义在类体中。
- 局部类中不能有静态成员

为什么局部类不能有静态成员？

1.7.9 构造函数

构造函数的作用：为了保证对象的每个数据成员都被正确初始化

构造函数的定义：

- 函数名和类名完全相同
- 不能定义构造函数的类型（返回类型），也不能使用void
- 通常情况下构造函数应声明为公有函数，否则它不能像其他成员函数那样被显式地调用
- 构造函数被声明为私有有特殊的用途。✓
- 构造函数可以有任意类型和任意个数的参数，一个类可以有多个构造函数（重载）

构造函数的特点：

- 如果类不提供任何一个构造函数，系统将为我们提供一个不带参数的默认的构造函数
- 全局对象的构造先于main函数

1.7.9.1 构造函数和new运算符

new运算符 (new operator)

- 分配内存（动态分配的内存，需要delete手动释放）
- 调用构造函数初始化

1.7.9.2 转换构造函数

带一个参数的构造函数的功能

- 普通构造函数（初始化）
- 转换构造函数（初始化和类型转化）

如何充当类型转化功能？

```

#include "Test.h"
int main(void)
{
    Test t(10);      // 带一个参数的构造函数，充当的是普通构造函数的功能

    t = 20;          // 将20这个整数赋值给t对象
    // 1、调用转换构造函数将20这个整数转换成类类型（生成一个临时对象）
    // 2、将临时对象赋值给t对象（调用的是=运算符）

    Test t2;
    t = temp;

    return 0;
}

```

I

**Test temp(20); 临时对象在当前行执行完毕就析构
t = temp;**

类的构造函数只有一个参数是非常危险的，因为编译器可以使用这种构造函数把参数的类型隐式转换为类类型

解决办法：声明为 **explicit**

1.7.9.3 构造函数初始化列表

构造函数执行的两个阶段：

- 初始化阶段：通过参数列表初始化
- 普通计算段（在构造函数体中执行的语句）

如果在普通计算段执行变量赋值操作，本质上不算是初始化操作。

因为此时类对象的空间已经分配好了

初始化语法：

```

1 class Clock{
2 public:
3     Clock(int hour, int minute, int second):
4         hour_(hour),
5         minute_(minute),
6         second_(second)
7     {
8         // 其他初始化操作
9     }
10 private:
11     int hour_;
12     int minute_;
13     int second_;
14 };

```

1.7.9.4 对象成员及其初始化：见Effective C++

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Object{
6 public:
7     Object(int num): num_(num){
8         cout << "Object" << endl;
9     }
10    ~Object()
11    {
12        cout << "~Object" << endl;
13    }
14 private:
15     int num_;
16 };
17
18 class Container{
19 public:
20     Container():
21     {
22         cout << "Container默认构造" << endl;
23     }
24     Container(int num1, int num2): obj_(num1),
25     obj2_(num2);
26     ~Container()
27     {
28         cout << "Container的析构" << endl;
29     }
30 private:
31     Object obj_;
32     Object obj2_;
33 };
34 int main(void)
35 {
36     Container c;
37
38     return 0;
39 }
```

```
39 }
```

执行流程：

- 先构造成员对象：obj_
- 再调用构造函数 Container
- 然后析构 ~Container
- 然后析构 ~Object

注意事项：

- 如果成员对象有多个，那么对象变量的构造与参数列表中出现顺序无关，而与在类中声明的顺序有关
- 如果对象成员 没有默认构造函数，那么就一定要在 参数列表中显式地构造对象成员

1.7.9.5 const成员、引用成员初始化

1. const成员只能通过参数列表初始化
2. 引用成员也只能通过参数列表初始化
3. 对象成员（对应的类没有默认构造函数）的初始化，只能通过参数列表初始化
4. 因为前面两者都是要求初始化，而实际上在构造函数体内执行的操作，不能称为初始化操作

1.7.9.6 枚举 enum

对于声明为常量的const成员，只能保证其对于某一个对象是常量。如果要该变量对于整个类都是常量。那么应该使用枚举 enum

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Object{
6 public:
7     enum KTYPE{
8         kTYPE_A = 100,
9         kTYPE_B = 200
10    }
11
12 };
13
14 int main(void)
15 {
```

```
16     cout << object::kTYPE_A << endl; // 100
17     cout << object::kTYPE_B << endl; // 200
18     Object obj1;
19     cout << obj1.kTYPE_B << endl; // 200
20     Object obj2;
21     cout << obj2.kTYPE_B << endl; // 200
22 }
```

1.7.10 拷贝构造函数

发生场景：用一个对象来初始化另一个对象

```
1 #include "Test.h"
2
3 int main(void)
4 {
5     Test t1(10);
6     Test t2(t1); // 拷贝构造函数
7     Test t3 = t1; // 仍然是拷贝构造,这时候的operator=等价于
8     Test t3(t1);
9
10    return 0;
11
12 // 拷贝构造函数实现
13 Test::Test(const Test& rhs) : num_(rhs.num_)
14 {
15     cout << "拷贝构造函数" << endl;
16 }
```

为什么拷贝构造的形参是引用传递？

因为如果拷贝构造的传入参数不是引用传递，那么就是值传递的形式，在实参传递给形参的过程中，会出现一次拷贝赋值的操作。这时候又需要进行一次拷贝赋值操作。

就进入了一个递归调用拷贝构造的情形，直到堆栈溢出。程序崩溃

这样会产生开销，因此建议使用引用传递对象实参

- 当函数的形参是类的对象，调用函数时，进行形参与实参结合时使用。这时要在内存新建立一个局部对象，并把实参拷贝到新的对象中。理所当然也调用拷贝构造函数。
- 当函数的返回值是类对象，函数执行完成返回调用者时使用。理由也是要建立一个临时对象中，再返回调用者。为什么不直接用要返回的局部对象呢？因为局部对象在离开建立它的函数时就消亡了，不可能在返回调用函数后继续生存，所以在处理这种情况时，编译系统会在调用函数的表达式中创建一个无名临时对象，该临时对象的生存周期只在函数调用处的表达式中。所谓return 对象，实际上是调用拷贝构造函数把该对象的值拷入临时对象。如果返回的是变量，处理过程类似，只是不调用构造函数。

```
1 #include <iostream>
2
3 using std::cout;
4
5 Test TestFunc(const Test& t)
6 {
7     return t;
8 }
9
10 Test& TestFunc2(const Test& t)
11 {
12     return t;
13 }
14
15 const Test& TestFunc3(const Test& t)
16 {
17     return t;
18 }
19
20 int main(void)
21 {
22     Test t(10);
23
24     TestFunc(t);
```

```

25 // 这里面, 当t作为实参传入的时候, 由于接受形参是引用传递, 不
会调用拷贝构造
26 // 但是, 作为函数返回值, 会构造一个临时对象, 如果没有一个外部
对象接受这个对象。
27 // 那么这个临时对象就是在该行结束的时候, 析构
28 cout << "-----" << endl;
29 Test t2 = TestFunc2(t);
30 // 这里会调用拷贝构造函数, 因为返回值是一个对象。临时对象构造
31 Test& t3 = TestFunc2(t);
32 // 这里会调用拷贝构造函数, 因为返回值是一个引用。临时对象构造
33 const Test& t4 = TestFunc3(t);
34 // 直接是引用原有的对象t, 因此没有任何函数调用
35 return 0;
36 }

```

```

Initializing 10
Initializing with other 10
Destroy 10
.....
Destroy 10
请按任意键继续...

```

```

}
Test TestFunc3(const Test& t)
{
    return t;
}

int main(void)
{
    Test t(10);
    //TestFun(t);
    //TestFun2(t);

    //t = TestFun3(t);
    Test t2 = TestFunc3(t);
    cout<<"....."<<endl;

    return 0;
}

```

临时对象被有名对象t2接管, 不会调用operator=和拷贝构造

1.7.10.1 深拷贝与浅拷贝

```

1 #include <iostream>
2 #include <cstring>
3
4 using namespace std;
5

```

```
6 class String{
7 public:
8     String(char* str = "");
9     ~String();
10
11    void printStr();
12    String& operator=(const String& other);
13 private:
14     char *str_;
15     char* AllocAndCpy(char* str);
16 };
17
18 String::String(char* str)
19 {
20     int len = strlen(str) + 1;
21     str_ = new char[len];
22     memset(str_, 0, len);
23     strcpy(str_, str);
24 }
25 String::~String()
26 {
27     delete[] str_;
28 }
29
30 // 深拷贝实现
31 String::String(const String& other)
32 {
33
34     int len = strlen(other.str_) + 1;
35     str_ = new char[len];
36     memset(str_, 0, len);
37     strcpy(str_, other.str_);
38 }
39
40 // operator=
41 String& operator=(const String& other)
42 {
43     // 前提是重载了operator==或者operator==
44     if (*this != other)
45     {
46         char* temp = str_;
```

```

47         // delete[] str_, 这里其实也不应该直接delete[]
48         str_;
49         // 因为一旦AllocAndCpy出了问题,那么this->str_数据就
50         // 被污染了
51         str_ = AllocAndCpy(other.str_);
52         delete[] temp;
53     }
54     return *this;
55 }
56 char* String::AllocAndCpy(char *str) // 但是这样容易造成内
57 存泄漏
58 {
59     int len = strlen(str) + 1;
60     char *temp = new char[len];
61     memset(temp, 0, len);
62     strcpy(temp, str);
63
64     return temp;
65 }
66 String::printStr()
67 {
68     cout << "str_:" << str_ << endl;
69 }
70 int main(void)
71 {
72     String str1("hello world");
73
74     str1.printStr();
75     // 这时候还没有办法使用cout << str1 << endl; 因为没有重载
76     // operator<<
77
78     // Error: String str2 = str1;
79     // 调用拷贝构造函数, 系统提供的默认拷贝构造函数实现的是浅拷贝
80     // str2.str_ = str1.str_;
81
82     // 默认operator=也是浅拷贝
83     String str3;
84     str3 = str1; // 浅拷贝, 需要自己实现operator=
85 }
```

浅拷贝：

- 多个指针指向同一块内存，存在的问题：在释放内存的时候，可能会造成内存的重复释放问题
- **解决办法：深拷贝**

重新开辟新的内存，让指针指向这块新开辟的内存

1.7.10.2 禁止拷贝

应用场景：要让对象是独一无二的，我们要禁止拷贝，方法如下：

1. 只需要将拷贝构造函数和operator=设置为 `private` 成员函数，并且不提供实现
2. `=delete`

1.7.10.3 空类默认产生的成员

C++ 教程网 空类默认产生的成员

```

class Empty {};
Empty();           // 默认构造函数
Empty( const Empty& ); // 默认拷贝构造函数
~Empty();          // 默认析构函数
Empty& operator=( const Empty& ); // 默认赋值运算符
Empty* operator&();           // 取址运算符
const Empty* operator&() const; // 取址运算符 const

```

```

#include <iostream>
using namespace std;

class Empty
{
public:
    Empty* operator&() // 默认取地址运算符&，返回this指针
    {
        return this;
    }
};

int main(void)
{
    Empty e;
    Empty* p = &e; // 等价于e.operator&();
    return 0;
}

```

e.operator&(this);

空类的大小：一个字节的空间，为了表示该类，并且方便生成实例

1.7.11 析构函数

析构函数的特点：

- 函数名和类名相似（前面多了一个字符“~”）
- 没有返回类型
- 没有参数
- 析构函数不能被重载
- 如果没有定义析构函数，编译器会自动生成一个默认析构函数，其格式如下：
类名::~默认析构函数名()
{
}
- 默认析构函数是一个空函数。

```
#include "Test.h"
int main(void)
{
    Test t[2] = {10, 20};

    Test* t2 = new Test(2);
    delete t2;

    Test* t3 = new Test[2];
    delete[] t3;

    return 0; }
```

1. new [] 和 delete[] 需要匹配

```
ca C:\Windows\system32\cmd.exe
Initializing 10
Initializing 20
Initializing 2
Destroy 2
Initializing Default
Initializing Default
Destroy 0
Destroy 0
Destroy 20
Destroy 10
请按任意键继续...
```

析构函数可以显式调用：但是不建议这么使用，因为如果在析构函数中有内存释放操作。那么会出现内存重复释放的问题

1.7.12 类与类之间的关系：StarUML绘制

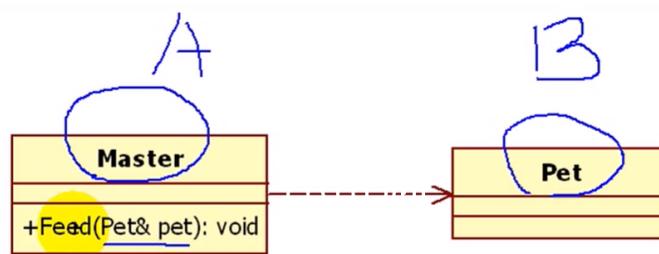
UML：为软件开发提供了一些标准的图例，统一开发思想，从而促进团队协作

- 类图(class diagram)
- 对象图(object diagram)
- 用例图(use case diagram)
- 组件图(component diagram)
- 部署图(deployment diagram)
- 组合结构图(composite structure diagram)
- 序列图(sequence diagram)
- 协作图(collaboration diagram)
- 状态图(statechart diagram)
- 活动图(activity diagram)

结构

行动

- 继承：类A继承自类B
- 关联：类A是类B的成员
 - 单向的，表示类B知道类A，而类B不知道类A
 - 双向的关联关系，（设计上应该避免）
- 聚合：比关联更强的关联关系。一个类是另一个类的成员，并且还存在整体与局部的关系（**整体并不负责局部的生命周期**）
- 组合：相比聚合而言，更强的关联关系。整体与局部的关系（**整体负责局部对象的生命周期**）
- 依赖关系



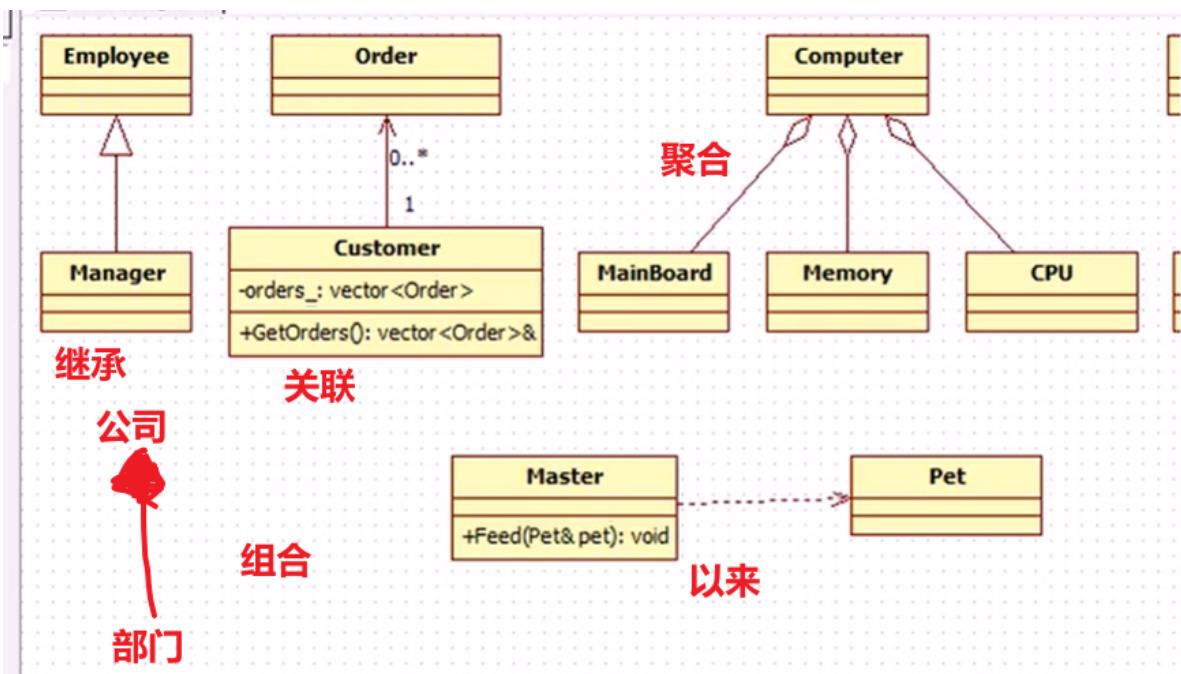
类A依赖于B：

从语义上来讲是A use B, 偶然的, 临时的

B作为A的成员函数参数

B作为A的成员函数的局部变量

A的成员函数调用B的静态方法



- 继承体现的是类与类之间的纵向关系，其他4种体现的是类与类之间的横向关系。
- 关联强弱
 - 依赖 < 关联 < 聚合 < 组合
- 继承 (A is B)
- 关联、聚合、组合 (A has B)
- 依赖 (A use B) | | **忌**

1.8 对象的使用

1.8.1 对象的两种语义

1. 值语义：可以拷贝的，拷贝之后，与原对象脱离关系
2. 对象语义：要么是不能拷贝的，要么可以拷贝，拷贝之后与原对象仍然存在一定关系。比如共享底层资源（要实现自己的拷贝构造函数）

1.8.2 static成员

需要某个变量被所有的对象访问，比如统计某种类型对象已创建的数量

- 全局变量
- static静态变量

- 对于特定类型的全体对象而言，有时候可能需要访问一个全局的变量。比如说统计某种类型对象已创建的数量。**共享 +1, -1 滥用**
- 如果我们用全局变量会破坏数据的封装，一般的用户代码都可以修改这个全局变量，这时我们可以用类的静态成员来解决这个问题。
- 非static数据成员存在于类类型的每个对象中，static数据成员独立该类的任意对象存在，它是与类关联的对象，不与类对象关联。

静态成员变量的声明和定义

- 静态成员的声明放在类内
- 静态成员的定义：应该在文件作用域，任意一个函数之外定义（**不能够在类声明的时候，实际上所有的类都成员变量都不能在类声明的时候定义**）
- 静态成员是共有的，不属于某一个具体的对象

1.8.3 static成员函数【见知识点辨析】

1.8.4 类、对象大小计算

- 类大小计算遵循前面学过的结构体对齐原则
- 类的大小与数据成员有关与成员函数无关
- 类的大小与静态数据成员无关
- 虚函数对类的大小的影响
- 虚继承对类的大小的影响

虚表指针

1.8.5 四种对象的作用域与生存期

栈对象

- 隐含调用构造函数（程序中没有显示调用）

堆对象

- 隐含调用构造函数（程序中没有显示调用）

全局对象、静态全局对象

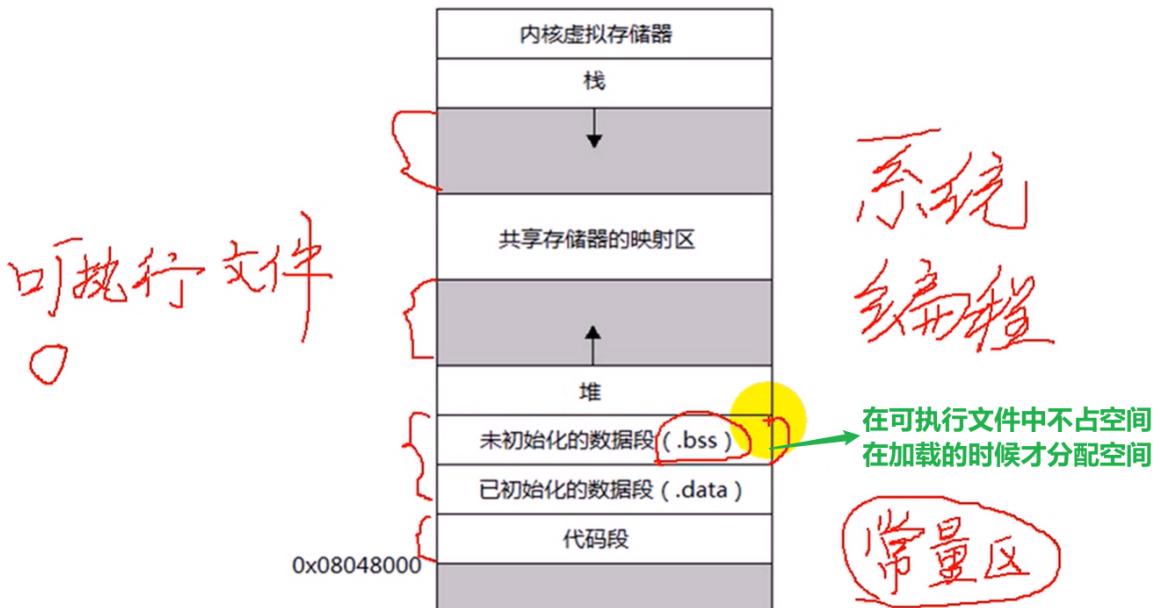
- 全局对象的构造先于**main**函数
- 已初始化的全局变量或静态全局对象存储于**.data**段中
- 未初始化的全局变量或静态全局对象存储于**.bss**段中

静态局部对象

- 已初始化的静态局部变量存储于**.data**段中
- 未初始化的静态局部变量存储于**.bss**段中

未初始化的全局变量，会采用默认初始化，存储在BSS (block started by symbol) 段中

两者区别见Linux系统编程



1.8.6 static与单例模式

1.8.7 const成员函数

- **const**成员函数不会修改对象的状态
- **const**成员函数只能访问数据成员的值，而不能修改它
- 可以通过**const**修饰符进行重载
- **const**成员函数可以修改被声明为**mutable**的数据成员

const成员函数定义：

```
1 class Test
2 {
3 public:
4     Test(int x) : x_(x), outputTimes(0)
5     {
6     }
7
8     int GetX() const
9     {
10         // ERROR: x_ = 100; const成员函数无法修改成员变量的
11         // 值
12         return x_;
13     }
14
15     void Output() const
16     {
17         cout << "x = " << x_ << endl;
18         outputTimes++; // 我们又希望能够在输出一次的情况下,
19         // 修改数据成员outputTimes
20     }
21
22     int GetOutputTimes() const
23     {
24         return outputTimes;
25     }
26 private:
27     int x_;
28     mutable int outputTimes;
29 };
30
31 int main(void)
32 {
33     const Test t(10);
34
35     t.GetX();
36
37     return 0;
38 }
```

1.8.8 const对象

const对象定义：

```
1 | const 类名 对象名;
```

- const对象只能调用const成员函数，不能调用非const成员函数

1.9 数据抽象与封装

1. C语言实现版本

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4
5 struct Link
6 {
7     int data;
8     struct Link* next;
9 };
10 struct Stack{
11     struct Link* head;
12     struct int size;
13 }
14
15 void StackInit(struct Stack* stack)
16 {
17     stack->head = nullptr;
18     stack->size = 0;
19 }
20
21 void StackPush(struct Stack* stack, const int data)
22 {
23     struct Link* node;
24     node = (struct Link*)malloc(sizeof(struct
Link));
25     assert(node != nullptr);
26
27     node->data = data;
28     node->next = stack->head;
29     stack->head = node;
```

```

30     stack->size++;
31 }
32
33 int StackEmpty(struct Stack* stack)
34 {
35     return (stack->size == 0); // C语言中没有bool类型
36 }
37 int StackPop(struct Stack* stack, int* data)
38 {
39     if (StackEmpty(stack))
40     {
41         return 0;
42     }
43     struct Link* temp = stack->head; // 保存头结点
44     *data = stack->head->data;
45     stack->head = stack->head->next;
46     free(temp); // 释放原来的头结点的内存
47
48     return 1;
49 }
50
51 void StackClear(struct Stack* stack)
52 {
53     struct Link* temp;
54     while (!StackEmpty(stack))
55     {
56         temp = stack->head;
57         stack->head = stack->head->next;
58         free(temp);
59     }
60
61     stack->size = 0;
62 }
```

2. C++版本实现

1.10 友元

- 友元的作用在于提高程序运行效率
- 但是不建议声明过多的友元，否则类的封装属性就不太明显了。暴露过多的 private 成员变量

- 友元关系是单向的

A是B的友元类（在B类中，friend class A），但是在A中，B不是A的友元。
即B不能够访问A的私有成员

- 友元关系不能够传递：A是B的友元类，B又是C的友元类，并不代表A是C的友元类
- 友元关系是不能够被继承：A是B的友元类，C继承自A，并不代表C是B的友元类

1.10.1 友元介绍

- 友元是一种允许非类成员函数访问类的非公有成员的一种机制。
- 可以把一个函数指定为类的友元，也可以把整个类指定为另一个类的友元。
 - 友元函数
 - 友元类

1.10.2 友元函数

- 友元函数在类作用域外定义，但它需要在类体中进行说明
- 为了与该类的成员函数加以区别，定义的方式是在类中用关键字friend说明该函数，格式如下：
 - friend 类型 友元函数名(参数表);
- 友元的作用在于提高程序的运行效率

- 友元函数不是类的成员函数，在函数体中访问对象的成员，必须用对象名加运算符“.”加对象成员名。但友元函数可以访问类中的所有成员（公有的、私有的、保护的），一般函数只能访问类中的公有成员。
- 友元函数不受类中的访问权限关键字限制，可以把它放在类的公有、私有、保护部分，但结果一样。
- 某类的友元函数的作用域并非该类作用域。如果该友元函数是另一类的成员函数，则其作用域为另一类的作用域，否则与一般函数相同。
- 友元函数破坏了面向对象程序设计类的封装性，所以友元函数如不是必须使用，则尽可能少用。或者用其他手段保证封装性。

1.10.3 友元类

```
1 class A  
2 {  
3     friend class B; // 声明B时A的友元，此时B可以访问A中的私有成员  
4 }
```

1.11 运算符重载

1.11.1 运算符重载

- 运算符重载允许把标准运算符（如+、-、*、/、<、>等）应用于自定义数据类型的对象
- 直观自然，可以提高程序的可读性
- 体现了C++的可扩充性

- 运算符重载仅仅只是语法上的方便，它是另一种函数调用的方式
- 运算符重载，本质上是函数重载
- 不要滥用重载、因为它只是语法上的方便，所以只有在涉及的代码更容易写、尤其是更易读时才有必要重载

最后一条如果不满足，就应该通过函数重载实现相关功能

1.11.2 成员函数重载

```
class Complex
{
public:
    Complex(int real, int imag);
    Complex();
    ~Complex();

    Complex& Add(const Complex& other);
    void Display() const;

    Complex operator+(const Complex& other); // I

private:
    int real_;
    int imag_;
};

#endif // _COMPLEX_H_
```

```
1 Complex c3 = c1 + c2;
2 // 如果是成员函数重载，等价于c1.operator+(c2);
3 // 如果是下面的友元函数重载，等价于operator+(c1, c2);
```

1.11.3 友元函数重载

```
Complex operator+(const Complex& c1, const Complex& c2)
{
    int r = c1.real_ + c2.real_;
    int i = c1.imag_ + c2.imag_;
    return Complex(r, i);
```

声明为友元函数，才能够访问类的私有成员

会比成员函数重载运算符的形参多一个

1.11.4 重载的规则

- 运算符重载不允许发明新的运算符。
- 不能改变运算符操作对象的个数。
- 运算符被重载后，其优先级和结合性不会改变。
- 不能重载的运算符：

运算符	符号
作用域解析运算符	::
条件运算符	? :
直接成员访问运算符	.
类成员指针引用的运算符	*
sizeof运算符	sizeof

- 一般情况下，单目运算符最好重载为类的成员函数；双目运算符则最好重载为类的友元函数。
- 以下一些双目运算符不能重载为类的友元函数：=、()、[]、->。
- 类型转换运算符只能以成员函数方式重载
- 流运算符只能以友元的方式重载

```

class Test
{
public:
    Test& operator=(const Test& other);
private:
    int x_;
};

Test t1;
Test t2;

t1 = t2;           // t1.operator(t2);

class D : public Test
{
};

Test& operator=(Test& t1, const Test& t2)
{
    t1.x_ = t2.x_;
    return t1;
}

D d;
test t;

d = t;
operator=(d, t);

```

**为什么不能将
operator=、
operator[]、
operator->
通过友元函数进行重载示例**

**因为调用运算符的对象此时可能
不是该class实例，涉及到转型操作**

1.11.5 ++运算符重载

□ 前置++运算符重载

□ 成员函数的方式重载，原型为：

函数类型 & operator++();

□ 友元函数的方式重载，原型为：

friend 函数类型 & operator++(类类型 &);

□ 后置自增和后置自减的重载

□ 成员函数的方式重载，原型为：

函数类型 & operator++(int);

□ 友元函数的方式重载，原型为：

friend 函数类型 & operator++(类类型 &, int);

注意要点：后置递增的重载

```

1 class Integer
2 {
3 public:
4     Integer(int i) : n_(i){}
5     void show()

```

```

6   {
7       cout << "n_ = " << n_ << endl;
8   }
9   Integer& operator++() // 前置递增
10  {
11      ++this->n_;
12
13      return *this;
14  }
15  Integer operator++(int) // 后置递增，并通过int区分
16  {
17      Integer temp(*this);
18      ++this->n_;
19
20      return temp; // 因为返回的是临时对象，所以不能返回引用。
21  }
22  // 以友元的方式重载后置递增
23  friend Integer operator++(Integer& i, int);
24 private:
25     int n_;
26 };
27
28 Integer operator++(Integer& i, int)
29 {
30     Integer temp(*this);
31     ++this->n_;
32
33     return temp; // 因为返回的是临时对象，所以不能返回引用。
34 }
```

1.11.6！运算符重载

1. 当字符串非空的时候返回为假
2. 当字符串为空的时候，返回为真

1.11.7 字符串类的[]运算符重载

```

1 char& String::operator[](int index)
2 {
3     // 为了缩短代码量，可以让non-const版本调用const版本
```

```

4     return const_cast<char&>(static_cast<const String&>
5         (*this)[index]);
6     //return str_[index];
7 } // 返回引用的目的是为了能够作为左值存在，能够被修改
8
9 String s1("hello world");
10 s1[1] = 'A';
11 // 为了避免const String被修改，应该重载operator[]
12 const char& String::operator[](int index) const
13 {
14     return str_[index];
15 }

```

1.11.8 +运算符的重载：二元运算符通过友元函数进行重载

```

String operator+(const String& s1, const String& s2)
{
    int len = strlen(s1.str_) + strlen(s2.str_) + 1;
    char* newstr = new char[len];
    memset(newstr, 0, len);
    strcpy(newstr, s1.str_);
    strcat(newstr, s2.str_);

    String tmp(newstr);
    delete newstr;
    return tmp;
}

```

为什么不能用成员函数重载？

因为成员函数的第一个隐含参数是自身，因此无法实现 `String str = "aa" + str2;` 这样的操作

1.11.9 流运算符重载

- 输出流运算符 `<<` 的重载，因为第一个参数为流对象。因此必须通过友元函数重载。并且返回一个流对象，为了能够连续输出

```
1 #include <iostream>
2
3 ostream& operator<<(ostream& os, const String& str)
4 {
5     os<<str.str_;
6
7     return os;
8 }
```

2. 插入运算符 `>>` 重载

```
1 istream& operator>>(istream& is, String& str)
2 {
3     char temp[1024];
4     cin >> temp;
5     str = temp;
6
7     return is; // 返回引用的目的是为了能够继续作为左值连续输入
8 }
```

1.11.10 类型转换运算符重载



类型转换运算符

- 必须是成员函数，不能是友元函数 因为要用到自身的this指针
- 没有参数（操作数是什么？）
- 不能指定返回类型（其实已经指定了）
- 函数原型： operator 类型名();

```
1 // 类型转换符的作用，将类类型转换为其他类型
2 // 而转换构造函数，是通过隐式转换将其他类型转换为类类型
3 class Integer
4 {
5 public:
6     operator int();
7 private:
```

```
8     int n_;
9 };
10
11 Integer::operator int()
12 {
13     return n_;
14 }
```

1.11.11 指针访问运算符 -> 重载

什么样的情况下需要重载 -> 运算符?

当我们通过对对象自动析构的原来来管理内存的时候，需要间接访问该内存的一些数据操作

```
1 #include <iostream>
2
3 class DBHelper
4 {
5 public:
6     DBHelper(){}
7     ~DBHelper(){}
8     void open()
9     {
10         std::cout << "open db..." << std::endl;
11     }
12     void close()
13     {
14         std::cout << "close db..." << std::endl;
15     }
16 };
17
18 // 由于我们不知道创建的DBHelper对象应该何时析构，所以可以将其内置为class DB的数据成员，
19 // 然后创建DB栈对象，利用栈对象的声明周期结束时，自动析构的特性来管理DBHelper的生命周期
20
21 class DB
22 {
23 public:
24     DB(){
25         db_ = new DBHelper();
```

```

26     }
27     ~DB()
28     {
29         delete db_;
30     }
31     DBHelper* operator->()
32     {
33         return db_;
34     }
35 private:
36     DBHelper* db_;
37 };
38
39 int main(void)
40 {
41     DB db;
42     db->open(); // 希望通过dp间接访问DBHelper的操作，那么需要
重载->
43
44     return 0;
45 }
```

好处：

- 实现了类似智能指针的内存管理方法
- 并且，由于DB有DBHelper的基类指针，如果后续有派生类继承自DBHelper，那么就可以产生多态

1.11.12 operator new和operator delete重载

- | | |
|---|--------|
|  <input type="checkbox"/> void* operator new(size_t size)
<input type="checkbox"/> void operator delete(void* p) | 需要配对重载 |
|  <input type="checkbox"/> void operator delete(void* p, size_t size)
<input type="checkbox"/> void* operator new(size_t size, const char* file, long line)
<input type="checkbox"/> void operator delete(void* p, const char* file, long line) | |
|  <input type="checkbox"/> void* operator new[](size_t size)
<input type="checkbox"/> void operator delete[](void* p)
<input type="checkbox"/> void operator delete[](void* p, size_t size) | 针对数组 |

new关键字的三种用法

- new operator: 在内部通过malloc分配内存, 然后调用对象的构造函数.**不可以被重载**
- operator new: 只分配内存。**可以被重载**
- placement new: 在已经存在内存上构造对象

```
1 #include <iostream>
2
3 using namespace std;
4
5 // 全局的operator new重载
6 void operator new(size_t size)
7 {
8     void* p = malloc(size);
9
10    return p;
11 }
12
13 void operator new[](size_t size)
14 {
15     void* p = malloc(size);
16
17    return p;
18 }
19
20 void operator delete[](void* p)
21 {
22     free(p);
23 }
24 void operator delete(void* p) // 调用优先级更高
25 {
26     free(p);
27 }
28 void operator delete(void* p, size_t size) // operator
29 // delete的重载形式2
30 {
31     free(p);
32 }
33 class Test
34 {
```

```
35 public:
36     Test(int n) : n_(n)
37     {
38     }
39     Test(const Test& other) : n_(other.n_)
40     {
41         cout << "Test(const Test& other)" << endl;
42     }
43     ~Test()
44     {
45         cout << "~Test()" << endl;
46     }
47     void operator new(size_t size) // operator new重载
48     {
49         void* p = malloc(size);
50
51         return p;
52     }
53
54
55     void* operator new(size_t size, void* p) // placement new重载
56     {
57         return p;
58     }
59
60     void operator new(void* p1, void* p2) // placement delete对应的delete
61     {
62         return p;
63     }
64
65     void operator delete(void* p) // 调用优先级更高
66     {
67         free(p);
68     }
69     void operator delete(void* p, size_t size) // operator delete的重载形式2
70     {
71         free(p);
72     }
```

```

73 private:
74     int n_;
75 }
76
77 int main(void)
78 {
79     Test* pt1 = new Test(100); // new operator =
operator new + constructor
80
81     char chunk[10];
82
83     Test* pt2 = new(chunk) Test(200); // placement new;
不分配内存+构造函数调用
84
85     pt2->~Test(); // 显示调用析构函数
86     Test* pt3 = reinterpret_cast<Test*>(chunk);
87     return 0;
88 }
```

operator new的用于跟踪的重载形式：

```

1 void operator new(size_t size, const char* file, long
line)
2 {
3     cout << file << ":" << line << endl;
4
5     void* p = malloc(size);
6
7     return p;
8 }
9
10 void operator delete(void* p, const char* file, long
line)
11 {
12     cout << file << ":" << line << endl;
13
14     free(p);
15 }
16
17 #define new new(__FILE__, __LINE__)
18 Test* pt4 = new(__FILE__, __LINE__) Test(300); // 便于排
查内存泄漏
```

19 `delete ptr4;`

1.12 标准库类型

1.12.1 标准库string类型

- **string**类型支持长度可变的字符串, C++标准库将负责管理与存储字符相关的内存, 以及提供各种有用的操作 模板类
- `typedef basic_string<char> string;`
- `typedef basic_string<wchar_t> wstring;`
- 要使用**string**类型对象, 必须包含相关头文件
 - `#include <string>`
 - `+ using std::string;`

成员函数	功能描述
<code>size()</code> ✓	得到字符串的大小 建议使用这种, 和stl其他容器类型更适配
<code>length()</code> ✓	同上
<code>empty()</code> ✓	判断是否为空
<code>substr()</code>	截取字符串
<code>find()</code> ✓	在字符串中查找字符或者字符串
<code>rfind()</code> ✓	反向查找
<code>replace()</code> ✓	替代 .c_str() // 转为C风格字符串
<code>compare()</code> ✓	比较字符串
<code>insert()</code> ✓	插入字符
<code>append()</code> ✓	追加字符
<code>swap()</code> ✓	交换字符串
重载运算符 ✓	[], +=, =, +, >, <, >=, <=, !=, ==, >>, << 等

1.12.2 map容器使用

map容器的元素插入方式

插入到map容器内部的元素默认是按照key从小到大来排序

因此, 必须要求插入的key类型支持operator<

当然, 也可以传入自定义的排序规则

```
1 #include <map>
2
3 map<string, int> mapTest;
4
5 mapTest["aaa"] = 100; // int& operator[](const string&
6 index)重载
7 mapTest.insert(map<string, int>::value_type("bbb",
8 200));
9 mapTest.insert(pair<string, int>("ccc", 300));
10 mapTest.insert(make_pair("ddd", 400));
```

1.13 继承

继承是面向对象设计的一个重要特性，没有使用到继承的程序设计，即使用到了class这一抽象数据类型，也不能成为面向对象编程设计。

- 派生类是基类的具体化
- 派生类的范围小，更具体
- 而基类的范围大，更抽象

1.13.1 代码重用

□ C++很重要的一个特征就是代码重用。在C语言中重用代码的方式就是拷贝代码、修改代码。C++可以用继承或组合的方式来重用。通过组合或继承现有的的类来创建新类，而不是重新创建它们。



优势：
原有的类接口实现，都是经过良好测试的，
只需要对新实现的功能进行测试

组合：将一个类作为另一个类的对象成员

1.13.2 继承的语法

```
1 class 派生类名 : 继承方式 基类名  
2 {  
3     派生类新增成员的说明;  
4 }
```

1.13.3 继承的级别

C++ 教程网

公有、私有、保护继承

派生类成员函数是否可以访问

继承方式	基类成员特性	派生类成员特性	派生类对象访问
公有继承	public	public	可直接访问
	protected	protected	不可直接访问
	private	不可直接访问 <small>只能通过基类暴露的接口访问</small>	不可直接访问
私有继承	public	private	不可直接访问
	protected	private	不可直接访问
	private	不可直接访问	不可直接访问
保护继承	public	protected	不可直接访问
	protected	protected	不可直接访问
	private	不可直接访问	不可直接访问

C++ 教程网

默认继承保护级别

- class Base {};
- struct D1 : Base {}; // 公有继承
- class D2 : Base {}; // 私有继承

1.13.4 接口继承和实现继承

- 我们将类的公有成员函数称为接口。
- 公有继承，基类的公有成员函数在派生类中仍然是公有的，换句话说是基类的接口成为了派生类的接口，因而将它称为接口继承。
- 实现继承，对于私有、保护继承，派生类不继承基类的接口。派生类将不再支持基类的公有接口，它希望能重用基类的实现而已，因而将它称为实现继承。

这时候，派生类对象不能直接调用基类暴露的接口，但是对于基类中非private接口可以在派生类的新定义成员函数中调用。

即是：实现继承，在需要的时候，只能在新的成员函数内部调用

1.13.5 继承与重定义

1. 数据成员的重定义：**隐藏**
2. 成员函数的重定义：

- 对基类的数据成员的重定义
- 对基类成员函数的重定义分为两种
 - overwrite** → 这两种都会**隐藏基类的同名函数**，使得无法直接调用基类的接口
 - 与基类完全相同
 - 与基类成员函数名相同，参数不同
 - override**

解决办法：
1. 加上作用域
2. using Base::func();

1.13.6 继承于构造函数

不能够被派生类继承的成员函数

- 构造函数
- 析构函数
- operator=赋值

- 基类的构造函数不被继承，派生类中需要声明自己的构造函数。
- 声明构造函数时，只需要对本类中新增成员进行初始化，对继承来的基类成员的初始化（调用基类构造函数完成）~~（调用基类构造函数完成）~~
- 派生类的构造函数需要给基类的构造函数传递参数

代码示例：

```
1 #include <iostream>
2 using namespace std;
3
4 class Base
5 {
6 public:
7     // 编译器不再提供默认构造函数
8     Base(int b) : b_(b)
9     {
10         cout << "Base..." << endl;
11     }
12     ~Base()
13     {
14         cout << "~Base()" << endl;
15     }
16 private:
17     int b_;
18 };
19
20 class Derived : public Base
21 {
22 public:
23     Derived(int d) : d_(d), Base(10) // 调用基类的构造函数
24     {
25         cout << "Derived()..." << endl;
26     }
27     Derived(int b, int d) : d_(d), Base(b) // 调用基类的
28     构造函数
29     {
30         cout << "Derived(int b, int d)..." << endl;
31     }
32 }
```

```

30     }
31     ~Derived()
32     {
33         cout << "~Derived()..." << endl;
34     }
35
36 private:
37     int d_;
38 };
39
40 int main(void)
41 {
42     Derived d(100); // 要构造一个派生类对象，先调用基类的构造
函数
43     cout << d.b_ << " " << d.d_ << endl;
44     Derived bd(10, 200); // 给基类的构造函数传值
45     return 0;
46 }
```

小结：

如果基类没有默认构造函数，那么在派生类的构造函数中，就需要对基类构造进行显示调用。因为派生类对象的构造函数晚于基类的构造函数调用。

如果不这样做，就会出错。

```

1 Derived(int d) : d_(d), Base(10) // 调用基类的构造函数
2 {
3     cout << "Derived()..." << endl;
4 }
5 Derived(int b, int d) : d_(d), Base(b) // 调用基类的构
造函数
6 {
7     cout << "Derived(int b, int d)..." << endl;
8 }
```

1.13.7 拷贝构造函数的实现：应该逐成员赋值

```

class Base
{
public:
    Base(int b) : b_(b), objb_(111)
    {
        cout<<"Base ... "<<endl;
    }
    Base(const Base& other) : objb_(other.objb_), b_(other.b_)
    {
    }
    Base()
    {
        cout<<"~Base ... "<<endl;
    }
    int b_;
    ObjectB objb_;
};

class Derived : public Base
{
public:
    Derived(int b, int d) : d_(d), Base(b), objd_(222)
    {
        cout<<"Derived ... "<<endl;
    }
    Derived(const Derived& other) : d_(other.d), objd_(other.objd_), Base(other)
    {
    }
    Derived()
    {
        cout<<"~Derived ... "<<endl;
    }
    int d_;
    ObjectD objd_;
};

```

因为ObjectB没有默认构造实现

派生类的拷贝构造实现

1.13.8 静态成员与继承

静态成员被所有类共享，无所谓继承。在基类和派生类中都只有一份

1.13.9 转换与继承

派生类对象也是基类对象，这意味着在使用到基类对象的地方，也可以使用派生类对象

当public继承时

- 派生类对象指针可以转化为基类对象指针。
- 可以将派生类对象看成基类对象（会产生**对象切割object slicing**），但是派生类特有的成员消失
- 而基类指针无法转换为派生类指针
- **基类指针可以强制转换为派生类指针，但是不安全**
- 基类对象无法强制转换为派生类独享

- 当派生类以**public**方式继承基类时,编译器可**自动执行的转换(向上转型 upcasting 安全转换)**
 - 派生类对象指针自动转化为基类对象指针
 - 派生类对象引用自动转化为基类对象引用
 - 派生类对象自动转换为基类对象(特有的成员消失)
- 当派生类以**private/protected**方式继承基类时
 - 派生类对象指针(引用)转化为基类对象指针(引用)需用**强制类型转化**。但不能用**static_cast**, 要用**reinterpret_cast**
 - 不能把派生类**对象**强制转换为**基类对象**

1.13.9.1 将基类对象转换为派生类对象的实现 (不推荐这种不安全的转型操作)

实现方法:

- 在派生类中提供对应的转换构造函数
- 在基类中提供类型转换符

```
1 #include <iostream>
2 #include <string>
3
4 class Manager;
5
6 class Employee
7 {
8 public:
9     Employee(int id, int dpatId, string name) : id(id),
10             dpatId_(dpatId), name_(name)
11     {}
12     ~Employee(){}
13     operator Manager();
14     // 不能直接在这里实现, 因为这时候manager只是一个前向声明
15 private:
16     int id_;
17     int dpatId_;
18     string name_;
```

```

19
20 class Manager : public Employee
21 {
22 public:
23     Manager(int id, int dpatId, string name, int
24     level) : Employee(id, dpatId, name), level_(level)
25     {}
26     ~Manager(){}
27     Manager(const Employee& emp) : Employee(emp),
28     level_(-1) // 转换构造函数
29     {
30     }
31 private:
32     int level_;
33 };
34 // 类内的转型运算符
35 Employee::operator Manager()
36 {
37     return Manager(id, dpatId, name_, -1);
38 }
```

1.13.10 多重继承

- 单重继承——一个派生类最多只能有一个基类
- 多重继承——一个派生类可以有多个基类
 - class 类名: 继承方式 基类1, 继承方式 基类2,
{....};
- 派生类同时继承多个基类的成员，更好的软件重用
- 可能会有大量的二义性，多个基类中可能包含同名变量或函数
- 多重继承中解决访问歧义的方法
 - 基类名::数据成员名（或成员函数(参数表)）
 - 明确指明要访问定义于哪个基类中的成员

多重继承存在的问题：容易出现二义性，多个基类中包含同名变量或函数

解决办法：采用虚基类来解决

1.13.11 虚继承和虚基类：用于钻石继承

- 当派生类从多个基类派生，而这些基类又从同一个基类派生，则在访问此共同基类中的成员时，将产生二义性——采用虚基类来解决。
- 虚基类的引入
 - 用于有共同基类的场合
- 声明
 - 以 **virtual** 修饰说明基类
- 例： class B1:virtual public BB₊
- 作用
 - 主要用来解决多继承时可能发生的对同一基类继承多次而产生的二义性问题。
 - 为最远的派生类提供唯一的基类成员，而不重复产生多次拷贝

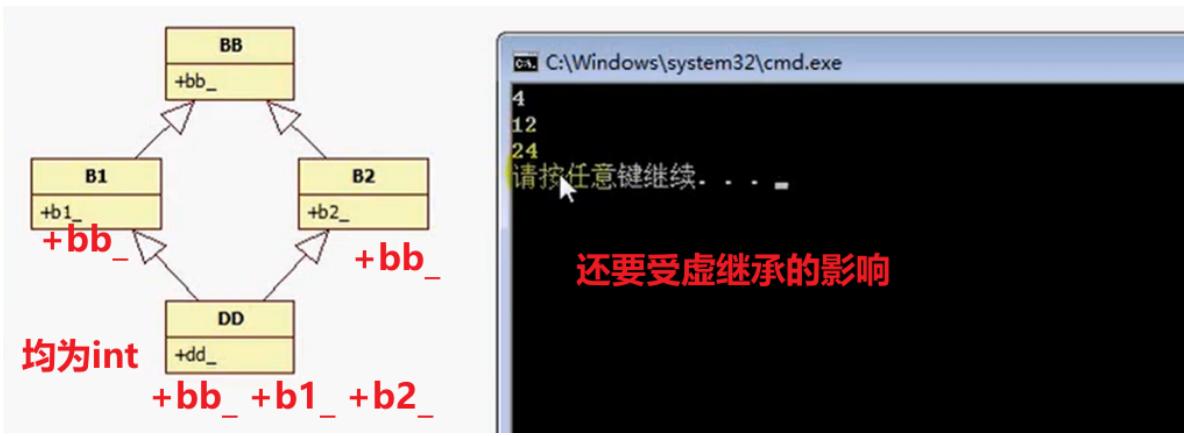
重
量

1.13.12 虚基类及其派生类构造函数

- 虚基类的成员是由最远派生类的构造函数通过调用虚基类的构造函数进行初始化的。
- 在整个继承结构中，直接或间接继承虚基类的所有派生类，都必须在构造函数的成员初始化表中给出对虚基类的构造函数的调用。如果未列出，则表示调用该虚基类的默认构造函数。
然后选择最远的派生类生效
- 在建立对象时，只有最派生类的构造函数调用虚基类的构造函数，该派生类的其他基类对虚基类构造函数的调用被忽略。

1.13.13 虚继承对C++对象内存模型造成的影响

- 遵循结构体的内存对齐原则
- 类的大小与数据成员有关，与成员函数无关
- 类的大小与静态数据成员无关
- 虚继承对类的大小的影响
- 虚函数对类的大小的影响



```

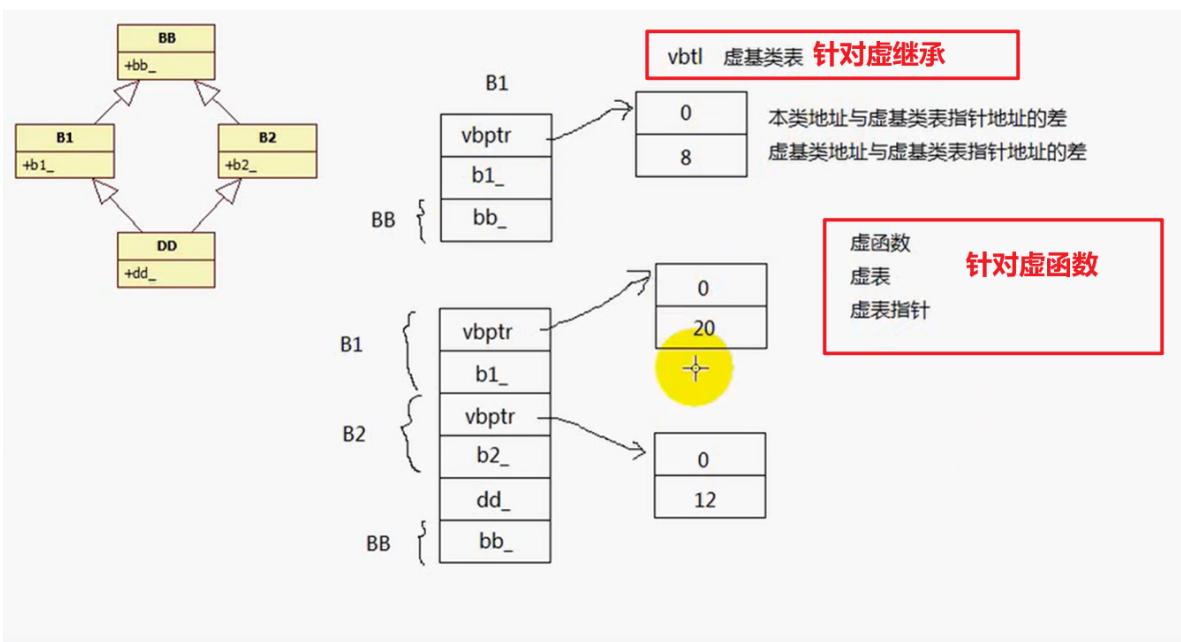
1 #include <iostream>
2
3 using namespace std;
4
5 class BB
6 {
7 public:
8     BB(int b) : bb_(b){}
9
10 public:
11     int bb_;
12 };
13
14 class B1 : virtual public BB
15 {
16 public:
17     B1(int b1, int b) : b1_(b1), BB(b){}
18     int b1_;
19 };
20
21 class B2 : virtual public BB
22 {
23 public:
24     B2(int b2, int b) : b2_(b2), BB(b){}
25     int b2_;
26 };
27
28 class DD : public B1, public B2
29 {
30 public:
31     DD(int d, int b1, int b2, int b) : dd_(d), B1(b1,
b), B2(b2, b), BB(b)

```

```

32     {}
33     int dd_;
34 };
35 // 钻石继承的方式
36
37 int main(void)
38 {
39     cout << sizeof(BB) << endl; // 4
40     cout << sizeof(B1) << endl; // 12
41     cout << sizeof(DD) << endl; // 24
42
43     return 0;
44 }

```



通过指针访问虚基类的成员，是间接访问

1.13.14 多态

多态的定义：调用同名的函数产生不同的行为

多态的实现：

- 函数重载
- 运算符重载
- 模板
- 虚函数（动态多态）：通过virtual函数实现

前面三种都是静态多态，编译期间确定，在编译器就已确定要调用的函数。

1.13.15 静态绑定与动态绑定

动态绑定的实现：

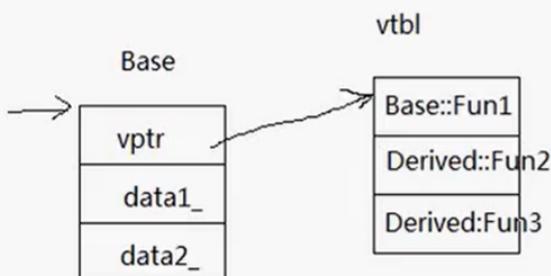
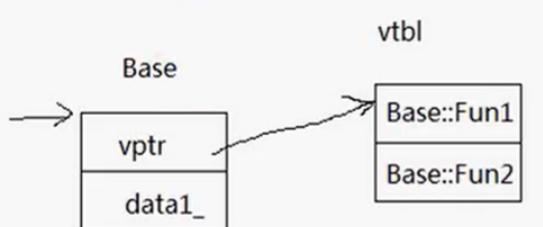
虚函数的动态绑定是通过虚表来实现的。包含虚函数的类头4个字节存放指向虚表的指针

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Base
6 {
7 public:
8     virtual void Func1()
9     {
10         cout << "Base::Func1() ..." << endl;
11     }
12     virtual void Func2()
13     {
14         cout << "Base::Func2() ..." << endl;
15     }
16     int data1_;
17 };
18
19 class Derived : public Base
20 {
21 public:
22     virtual void Func2()
23     {
24         cout << "Derived::Func2() ..." << endl;
25     }
26
27     virtual void Func3()
28     {
29         cout << "Derived::Func3() ..." << endl;
30     }
31     int data2_;
32 };
33
34 typedef void (*FUNC)(); // 定义一个函数指针，名称为FUNC
35 int main(void)
```

```

36 {
37     Base b;
38     cout << sizeof(Base) << endl;
39     cout << sizeof(Derived) << endl;
40
41     long** p = (long**)&b; // 指向base的虚表指针所指向的虚表
42     FUNC fun = (FUNC)p[0][0]; // 将p[0][0]强制转换为FUNC
43     fun();
44     fun = (FUNC)p[0][1]; // Func2
45     fun();
46
47     Derived d;
48     p = (long**)&d;
49
50     fun = (FUNC)p[0][0]; // Base::Func1
51     fun();
52     fun = (FUNC)p[0][1]; // Derived::Func2
53     fun();
54     fun = (FUNC)p[0][2]; // Derived::Func3
55     fun();
56
57     return 0;
58 }

```



1.13.16 虚函数

- 虚函数的概念：在基类中冠以关键字 `virtual` 的成员函数
- 虚函数的定义：
 - `virtual` 函数类型 函数名称(参数列表);
 - 如果一个函数在基类中被声明为虚函数，则他在所有派生类中都是虚函数
- 只有通过基类指针或引用调用虚函数才能引发动态绑定
- 虚函数不能声明为静态



- 不能将虚函数声明为静态函数或者友元函数，因为他们都没有 `this` 指针，无法通过虚表指针进行动态绑定
- 构造函数不能声明为虚函数

因为，在构造函数没有调用完之前，是没办法分配内存，无法得到 `vptr`，进而也无法进行动态绑定构造函数的入口地址

- 而析构函数应该声明为虚函数

1.13.17 虚析构函数

如果一个类要作为多态基类，那么就应该将其析构函数定义成虚函数。这样才能够在基类指针释放的时候，才能够正确的释放基类指针指向的派生类对象的内存。（调用派生类的内存）

- 何时需要虚析构函数？
- 当你可能通过基类指针删除派生类对象时
- 如果你打算允许其他人通过基类指针调用对象的析构函数（通过 `delete` 这样做是正常的），并且被析构的对象是有重要的析构函数的派生类的对象，就需要让基类的析构函数作为虚函数。

纯虚析构函数：如果 `base` 类没有任何成员函数，但是又希望作为基类。那就应该将其析构函数作为 纯虚析构函数

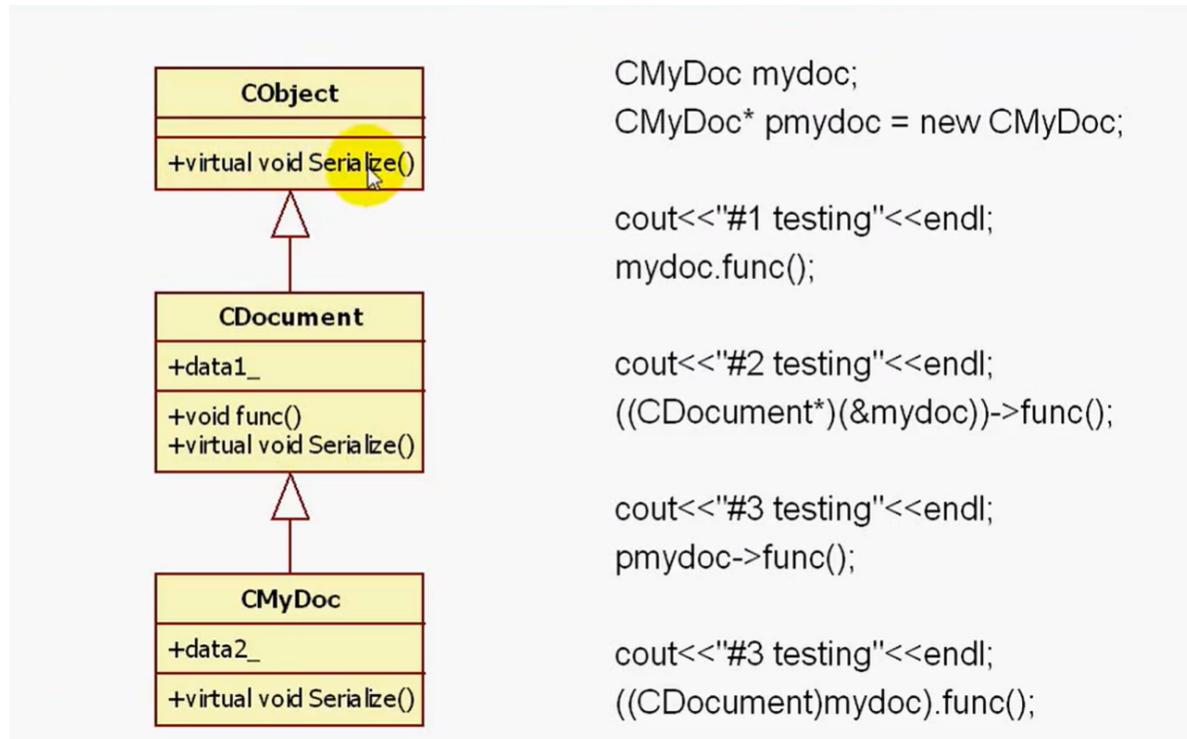
通常情况下，在基类中纯虚函数不需要实现。例外是纯虚析构函数要给出实现（空实现）。

为什么：因为派生类对象在释放的时候，无法调用基类的析构函数。

1.13.18 object slicing与虚函数

对象切割：派生类向上转型的时候，会发生类型切割

完完全全将派生类对象转为了基类对象，包括虚函数表。会发生一次拷贝构造



1.13.19 纯虚函数

虚函数：通过将基类指针指向派生类对象，这时候通过基类指针调用的虚函数实际上是调用的派生类对象的相应实现

- 当基类的接口没办法提供具体实现，或者说只需要提供一个接口而不提供默认实现的时候。应该将基类对应的函数声明为纯虚函数
- 拥有纯虚函数的类是一个抽象类，并且抽象类不能够实例化。**但是可以声明抽象类的指针和引用**
- 派生类中必须实现基类中的纯虚函数，否则仍将它看成一个抽象类

□ 在基类中不能给出有意义的虚函数定义，这时可以把它说明成纯虚函数，把它的定义留给派生类来做

□ 定义纯虚函数：

```
class 类名{  
    virtual 返回值类型 函数名(参数表) = 0;  
};
```

□ 纯虚函数不需要实现

1.13.20 对象的动态创建

反射技术：动态获取类型信息（方法和属性）

动态创建对象：

- 动态调用对象的方法
- 动态调用对象的属性
- 对原有的类不做任何更改，只需要增加一个宏就能够实现动态创建

需要给每个类添加元数据

能够适配新创建的类，并且避免使用if-else语句。结合配置文件使用

组件编程思想：

1.13.21 runtime type information (RTTI)：都不如虚函数的虚函数表效率来得高

运行时类型信息，主要通过

- dynamic_cast运算符：基类指针向下转型的时候，是安全的。
 - 要支持 `dynamic_cast` 需要编译器支持运行时类型识别。
 - 要具有多态类型的继承体系（即基类有虚函数）
- typeid运算符：用于运行时类型识别，返回值类型为type_info
- type_info

```
cout<<typeid(*p).name()<<endl;
cout<<typeid(Circle).name()<<endl;
if (typeid(Circle).name() == typeid(*p).name())
{
    cout<<"p is point to a Circle object"<<endl;
    ((Circle*)p)->Draw();
}
else if (typeid(Circle).name() == typeid(*p).name())
{
    cout<<"p is point to a Circle object"<<endl;
    ((Circle*)p)->Draw();
}
```

typeid和typeinfo的作用

```
class type_info {
public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    int before(const type_info& rhs) const;
    const char* name() const;
    const char* raw_name() const;
private:
    void *_m_data;
    char _m_d_name[1];
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
    static const char _Name_base(const type_info * __type_info_node*
        __ptype_info_node);
};
```

不能将typeinfo赋值给其他对象，因为它的拷贝构造函数是私有的

1.14 异常

1.14.1 C语言错误处理方法

- 返回值 (if-else语句判断返回值)：比较繁琐，每次函数调用都要判断
- 失败时返回-1
- goto语句：当函数内部发生错误时，跳转到局部的错误处理
- setjmp/longjmp

goto语句，局部跳转 只能在函数内部跳转

```
int test()
{
    char* p1 = (char*)malloc(10);
    if (p1 != NULL)
    {
        ...
    }
    else
    {
        goto POS1;
    }
    ...
    char *p2 = (char*)malloc(10);
    if (p2 != NULL)
    {
        I
    }
    else
    {
        goto POS2;
    }
    ...
    ...
}

POS1:
    exit(1);
POS2:
    free(p1);
    exit(1);
}
```

```
double Divide(double a, double b)      C++异常处理的雏形
{
    if (b == 0.0)
    {
        longjmp(buf, 1);           // throw
    }
    else          跳转到buf的setjmp的位置，并返回第二个参数
        return a / b;            设定的值
}

int main(void)
{
    int ret;
    ret = setjmp(buf);
    if (ret == 0)           // try
    {
        printf("division ... \n");
        printf("%f\n", Divide(5.0, 0.0));
    }
    else if (ret == 1)       // catch 根据不同的ret值进行处理
    {
        printf("divisiong by zero\n");
    }
    return 0;
}
```

相比于goto的优势：可以跨函数跳转，适用于错误发生点离调用点远的情形

```
double Divide(double a, double b)
{
    if (b == 0.0)
    {
        throw 1; // Throw
    }
    else
        return a / b;
}

int main(void)
{
    try // try
    {
        cout<<"division ... "<<endl;
        cout<<Divide(5.0, 0.0)<<endl;
    }
    catch (int) // catch
    {
        cout<<"divisiong by zero"<<endl;
    }
    return 0;
}
```

C++
教程网

C++异常处理优点

- 错误处理代码的编写不再冗长乏味，并且不再与“正常”代码混在一起。程序员可以将注意力集中于正常流程，然后在某个区域里编写异常处理代码。如果多次调用同一个函数，只需在一个地方编写一次错误处理代码。**catch**
- 错误不能被忽略。

如果不对抛出的异常进行处理，那么系统会对该错误进行默认的处理。而C语言中如果没有对某个异常进行捕获处理，那么将被忽略

1.14.2 程序错误

□ 编译错误，即语法错误。程序就无法被生成运行代码。

□ 运行时错误

□ 不可预料的逻辑错误

□ 可以预料的运行异常

□ 例如：

□ 动态分配空间时可能不会成功

□ 打开文件可能会失败

□ 除法运算时分母可能为0

□ 整数相乘可能溢出

□ 数组越界.....

1.14.3 异常的语法

```
1 try
2 {
3     // try语句块
4 }
5 catch (类型1 参数1)
6 {
7     // 针对类型1的异常处理
8 }
9 catch (类型2 参数2)
10 {
11     // 针对类型2的异常处理
12 }
13 ...
```

1.14.4 异常抛出：

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 class MyException
7 {
8 public:
```

```
9     MyException(const char* message) :
10    message_(message)
11    {
12        cout << "MyException..." << endl;
13    }
14    MyException(const MyException& other) :
15    message_(other.message_)
16    {
17        cout << "MyException..." << endl;
18    }
19    ~MyException()
20    {
21        cout << "~MyException" << endl;
22    }
23    string what() const
24    {
25        return message_;
26    }
27
28    double Divide(double a, double b)
29    {
30        if (b == 0.0d)
31        {
32            MyException e("division by zeros");
33            throw e;
34            // 如果是 throw MyException("division by
35            // zeros");
36            // 那么会减少一次临时对象的构造工作
37            // throw 1; // 不会被catch(double)捕获，不做类型转
38            // 换
39    }
40
41    int main(void)
42    {
43        try
44        {
45            cout << Divide(5.0, 0.0d) << endl;
46        }
47        catch(MyException& e)
```

```
46     {
47         cout << e.what() << endl;
48     }
49     catch(...) // 捕获任意类型的异常
50     {
51         cout << "catch exception ..." << endl;
52     }
53     return 0;
54 }
```

1.14.5 异常的捕获

- 一个异常处理器一般只能捕捉一种类型的异常
- 异常处理器的参数类型和抛出异常的类型相同
- ...表示可以捕获任何异常

1.14.6 异常的传播

- try块可以嵌套
- 程序按顺序寻找匹配的异常处理器，抛出的异常将被第一个类型符合的异常处理器捕获
- 如果内层try块后面没有找到合适的异常处理器，该异常向外传播，到外层try块后面的catch块中寻找
- 没有被捕获的异常将调用**terminate**函数，**terminate**函数默认调用**abort**终止程序的执行
- 可以使用**set_terminate**函数指定**terminate**函数将调用的函数

1.14.7 栈展开

□ 沿着嵌套调用链接向上查找，直至为异常找到一个catch子句。这个过程称之为栈展开。

□ 为局部对象调用析构函数

□ 析构函数应该从不抛出异常

□ 栈展开期间会执行析构函数，在执行析构函数的时候，已经引发的异常但还没处理，如果这个过程中析构函数又抛出新的异常，将会调用标准库的**terminate**函数。**abort**

□ 异常与构造函数

□ 构造函数中可以抛出异常。如果在构造函数函数中抛出异常，则可能该对象只是部分被构造。即使对象只是被部分构造，也要保证销毁已构造的成员。

如何保证已构造的成员：当成员是指针的时候

1.14.8 异常与继承

1. 如果异常类型是一个类，他的基类也是一个异常类型。那么应该将派生类的错误处理放在前面，而基类的错误处理放在后面。MyException被一个类继承

1.15 I/O流类库

- 标准I/O流
- 文件流
- 字符串流

□ 数据的输入和输出（input/output简写为I/O）

□ 对标准输入设备和标准输出设备的输入输出简称为标准I/O

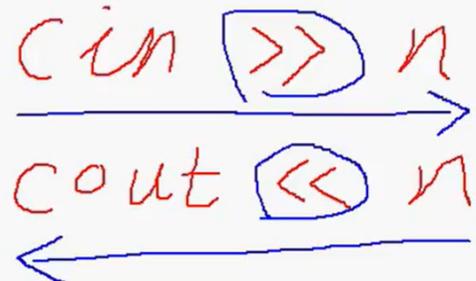
□ 对在外存磁盘上文件的输入输出简称为文件I/O

□ 对内存中指定的字符串存储空间的输入输出简称为串I/O

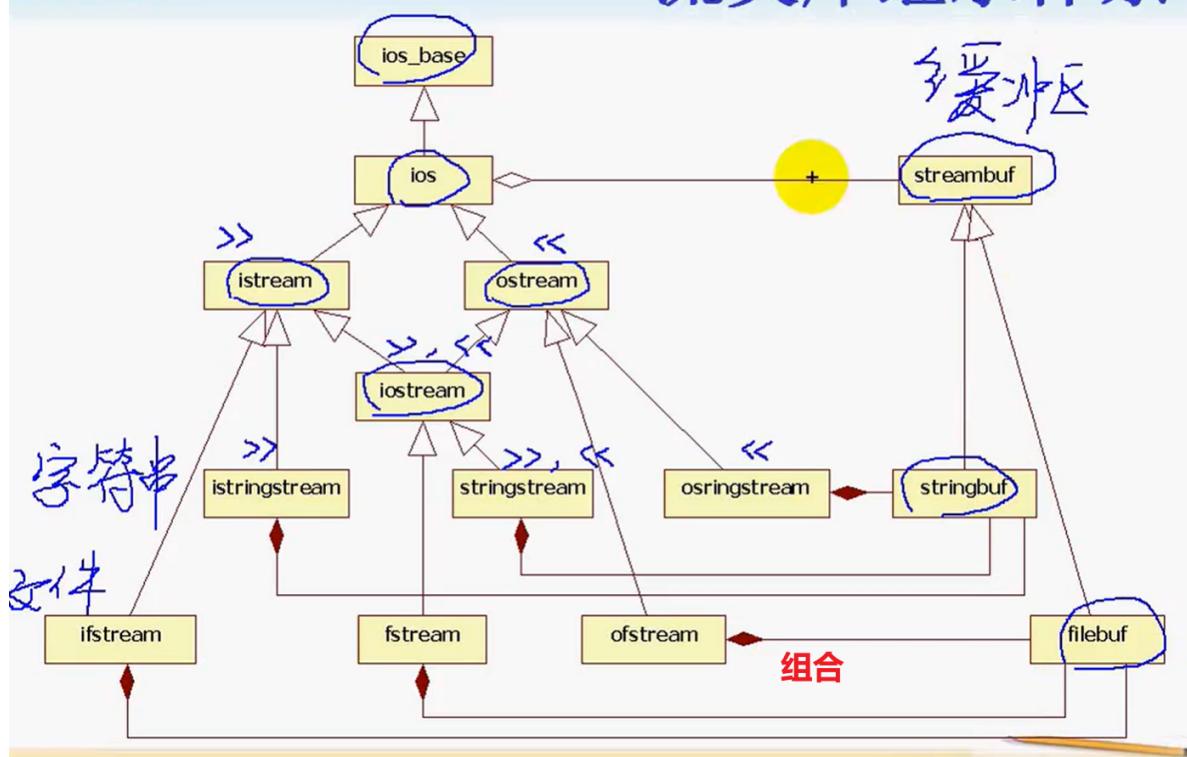
- 数据输入输出的过程,可以形象地看成流 >>
- 从流中获取数据的操作称为“提取”(输入)操作
- 向流中添加数据的操作称为“插入”(输出)操作 <<
- 标准输入输出流

- 文件流
- 字符串流

+



流类库继承体系



- 流库具有两个平行的基类：**streambuf** 和 **ios** 类，所有流类均以两者之一作为基类
- **streambuf** 类提供对缓冲区的低级操作：设置缓冲区、对缓冲区指针操作、向缓冲区存/取字符
- **ios_base**、**ios** 类记录流状态，支持对**streambuf** 的缓冲区输入/输出的格式化或非格式化转换
- **strstreambuf**: 使用串保存字符序列。扩展 **streambuf** 在缓冲区提取和插入的管理
- **filebuf**: 使用文件保存字符序列。包括打开文件；读/写、查找字符

2. STL

C++
教程网

什么是STL

可扩展性

- STL（Standard Template Library），即标准模板库，是一个高效的C++程序库。
- 包含了诸多在计算机科学领域里常用的基本数据结构和基本算法。为广大C++程序员们提供了一个可扩展的应用框架，高度体现了软件的可复用性

新算法

容器

- 从逻辑层次来看，在STL中体现了泛型化程序设计的思想 (generic programming)
 - 在这种思想里，大部分基本算法被抽象，被泛化，独立于与之对应的数据结构，用于以相同或相近的方式处理各种不同情形。
- 从实现层次看，整个STL是以一种类型参数化 (type parameterized) 的方式实现的
 - 基于模板(template)

基于模板的
类型参数化的
在编译阶段实例化

2.1 模板

<https://blog.csdn.net/leizardfu/article/details/56852043>

模板的定义：

- 模板也是一种静态多态的实现方式。将程序所处理的对象的类型参数化。
- 采用模板编程，可以为各种逻辑功能相同，而数据类型不同的程序提供一种代码共享的机制

模板的引入初衷：

- 考虑求两数较大值函数 max(a,b)
- 对于 a, b 的不同类型，都有相同的处理形式：

```
return a < b ? b : a;
```
- 用已有方法解决：
 - (1) 宏替换 #define max(a,b) ((a)<(b) ? (b) : (a))
问题 避开类型检查
 - (2) 重载
问题 需要许多重载版本 可扩展性差
 - (3) 使用函数模板 ✓

- 宏替换：不做类型检查，
- 重载：为每个类型提供一个重载版本，程序自己来维护这些重载的版本（可拓展性差）

模板的特性：为具有相同代码逻辑的代码提供一个模板，并将**类型**作为参数传递。从而实例化出对应数据类型的版本。不同的版本由编译器来维护。（会做安全的类型检查）

- 函数模板
- 类模板（muduo的thread local）
- STL的容器模板

2.1.1 函数模板

函数模板声明形式：

□ 函数模板的一般说明形式如下：

template < 模板形参表 >

返回值类型 函数名(模板函数形参表){

//函数定义体

}

□ 函数模板的定义以**关键字template**开头

□ template之后<>中是函数模板的参数列表

□ 函数模板的参数是类型参数，其类型为**class**或

typename

因为class具有特殊的意义

□ template<class T>

□ template<class T1, class T2>

函数模板的使用：

□ 函数模板为所有的函数提供唯一的一段函数代码，增强了函数设计的通用性

□ 使用函数模板的方法是先说明函数模板，然后实例化成相应的模板函数进行调用执行

□ 函数模板不是函数，不能被执行

□ 置换代码中的类型参数得到**模板函数** —— 实例化

□ 实例化后的模板函数是真正的函数，可以被执行

编译器

函数模板

编译期间完成实例化 → 静态多态

实例化

模板函数

- 模板被编译了两次
 - 实例化之前，先检查模板代码本身，查看语法是否正确；在这里会发现语法错误，如果遗漏分号等。
 - 实例化期间，检查模板代码，查看是否所有的调用都有效。在这里会发现无效的调用，如该实例化类型不支持某些函数调用等。未重载operator等
- 普通函数只需要声明，即可顺利编译，而模板的编译需要查看模板的定义

因此，模板不建议使用分离式编译。而应该将实现也放在 .h 文件中

2.1.1.1 函数模板特化

问题：为什么需要模板特化？

特化的模板，本质上也是模板。只是为了适配特定的数据类型

读完STL源码剖析看是否能找到答案

特化形式：

```
template<>
const char* const& max<const char*>(const char*
    const& a, const char* const& b)
{
    //return a < b ? b : a;
    return strcmp(a, b) < 0 ? b : a;
}
```

2.1.2 类模板

类模板：将类定义中的数据类型参数化

可以用相同的类模板，来组建任意类型的容器组合

2.1.2.1 类模板定义

类的成员函数就变成了函数模板

```

template <typename T>
class Stack
{
public:
    Stack(int maxSize);
    ~Stack(); I
private:
    T* elems_;
    int maxSize_;
};

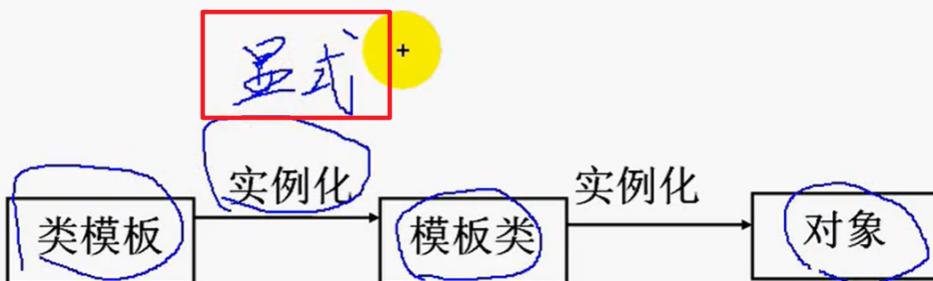
template <typename T>
Stack<T>::Stack(int maxSize) : maxSize_(maxSize)
{
    elems_ = new T[maxSize_];
}

template <typename T>
Stack<T>::~Stack()
{
    delete[] elems_;
}

```

2.1.2.2 类模板的使用

- 类模板的实例化：用具体的数据类型替换模板的参数以得到具体的类（模板类）
- 模板类也可以实例化为对象
- 用下列方式创建类模板的实例：
类名 <类型实参表> 对象名称；



2.1.3 非类型模板参数

对于函数模板与类模板，模板参数并不局限于类型。普通值也可以做为模板参数

```

1 template <typename T, int MAXSIZE>
2 class Stack
3 {
4 public:

```

```
5     Stack();
6     ~Stack();
7
8     void Push(const T& elem);
9     void Pop();
10    T& Top();
11    const T& top() const;
12 private:
13     T* elems_;
14     int top_;
15 };
16
17 template <typename T, int MAXSIZE>
18 Stack<T, MAXSIZE>::Stack() : top_(-1)
19 {
20     elems_ = new T[MAXSIZE];
21 }
22
23 template <typename T, int MAXSIZE>
24 void Stack<T, MAXSIZE>::Push(const T& elem)
25 {
26     if (top_ + 1 >= MAXSIZE)
27         throw out_of_range("out of range");
28     elems_[++top_] = elem;
29 }
```

2.1.4 缺省模板参数

```
template <typename T, typename CONT =
    std::vector<T> >
class Stack_
```

```
{
...
}
```

相当于用vector<T>充当stack的底层实现容器

(T*)

动态数组

链表
队列

将数据结构类型也传递进来

2.1.5 成员模板

1. 为什么需要成员模板

因为d.Assign(i)会出错，因此需要将成员函数Assign也定义成模板

2. 需要注意的问题：

因为MyClass<X>和MyClass<T>可能是不一样的类型。所以不能直接访问私有成员。应该暴露一个接口

```
template <typename T>
class MyClass
{
private:
    T value;
public:
    template <class X>
    void Assign(const MyClass<X> & x)
    {
        value = x.GetValue();
    }
    T GetValue() const { return value; }
};

MyClass<double> d;
MyClass<int> i;
d.Assign(d); // OK
d.Assign(i); // OK
```

X.Value

应用场景：auto_ptr的拷贝构造函数就是一个成员模板

```
template<class _Other>
auto_ptr<auto_ptr<_Other>& _Right) _THROW0()
: _Myptr(_Right.release())
{ // construct by assuming pointer from _Right
}

auto_ptr<_Ty>& operator=(auto_ptr<_Ty>& _Right) _THROW0()
{ // assign compatible _Right (assume pointer)
reset(_Right.release());
return (*this);
}
```

2.1.6 关键字typename

C++
教程网

关键字typename

```
template <typename T>
class MyClass
{
private:
    typename T::SubType *ptr;
};
```

如果没有typename
那么编译器会将这段代码
解析成：T的subtype*ptr;

有typename会将ptr看成一个指针

静态数据成员

没有typename就会将subtype解析成：T的静态数据成员

有typename则，将subtype解析成一个T的子类型

```

#include <iostream>
using namespace std;

template <typename T>
class MyClass
{
private:
    typename T::SubType* ptr_;
};

class Test
{
public:
    typedef int SubType;
};

int main(void)
{
    MyClass<Test> mc;
    return 0;
}

```

传入参数类型Test必须有子类型subType

2.1.7 派生类和模板

- 为了运行的效率，类模板是相互独立的，即独立设计，没有使用继承的思想。对类模板的扩展是采用适配器（adapter）来完成的。通用性是模板库的设计出发点之一，这是由泛型算法和函数对象等手段达到的。
- 派生的目标之一也是代码的复用和程序的通用性，最典型的就是MFC，派生类的优点是可以由简到繁，逐步深入，程序编制过程中可以充分利用前面的工作，一步步完成一个复杂的任务。
- 模板追求的是运行效率，而派生追求的是编程的效率。

模板编程：将类型当做参数传递，实现适配

2.1.8 面向对象与泛型

面向对象编程和泛型编程都依赖于多种多态：

1. 动态多态：函数入口地址在运行的时候才确定，效率不如静态多态高
 - 忽略基类和派生类之间的类型差异
 - 只要使用基类指针或引用基类类型对象、派生类类型对象就可以共享相同的代码
2. 静态多态（泛型编程）：模板实现（在编译器决定模板实例化）
 - 将编写的类和函数能够多态地用于编译时不相关的类型。
 - 一个类或一个函数可以用来操纵多种类型的对象

2.1.9 模板的应用

2.1.9.1 用模板实现单例模式

```
1 #ifndef _STUDYCPP_SINGLETON_H_
2 #define _STUDYCPP_SINGLETON_H_
3
4 #include <iostream>
5 #include <cstdlib>
6
7 using std::cout;
8 using std::endl;
9
10 template <typename T>
11 class Singleton
12 {
13 public:
14     static T &GetInstance()
15     {
16         Init();
17
18         return *instance_;
19     }
20
21 private:
22
23     static void Init()
24     {
25         if (instance_ == 0)
26         {
```

```

27         instance_ = new T;
28         atexit(Destroy);
29     }
30
31 }
32
33 static void Destroy()
34 {
35     delete instance_;
36 }
37
38 Singleton()
39 {
40     cout << "Singleton()..." << endl;
41 }
42 Singleton(const Singleton & other) = default;
43 Singleton& operator= (const Singleton &other) =
44 default;
45 ~Singleton()
46 {
47     cout << "~Singleton()..." << endl;
48 }
49
50 static T* instance_; // 智能指针实现对象释放
51 };
52 // 类外初始化
53 template <typename T>
54 T* Singleton<T>::instance_ = 0;
55 #endif // _STUDYCPP_SINGLETON_H_

```

存在的问题:

1. 非线程安全的
 - 普通锁: double check lock (存在CPU动态指令优化的问题)
 - linux下, 通过 `pthread_once` 实现

2.1.10 小结

1. 编译器的函数匹配原则
 - 全局的非模板函数

- 模板函数推导：如何进行推导：根据传入参数类型自动推导模板参数类型。
而不需要传递类型

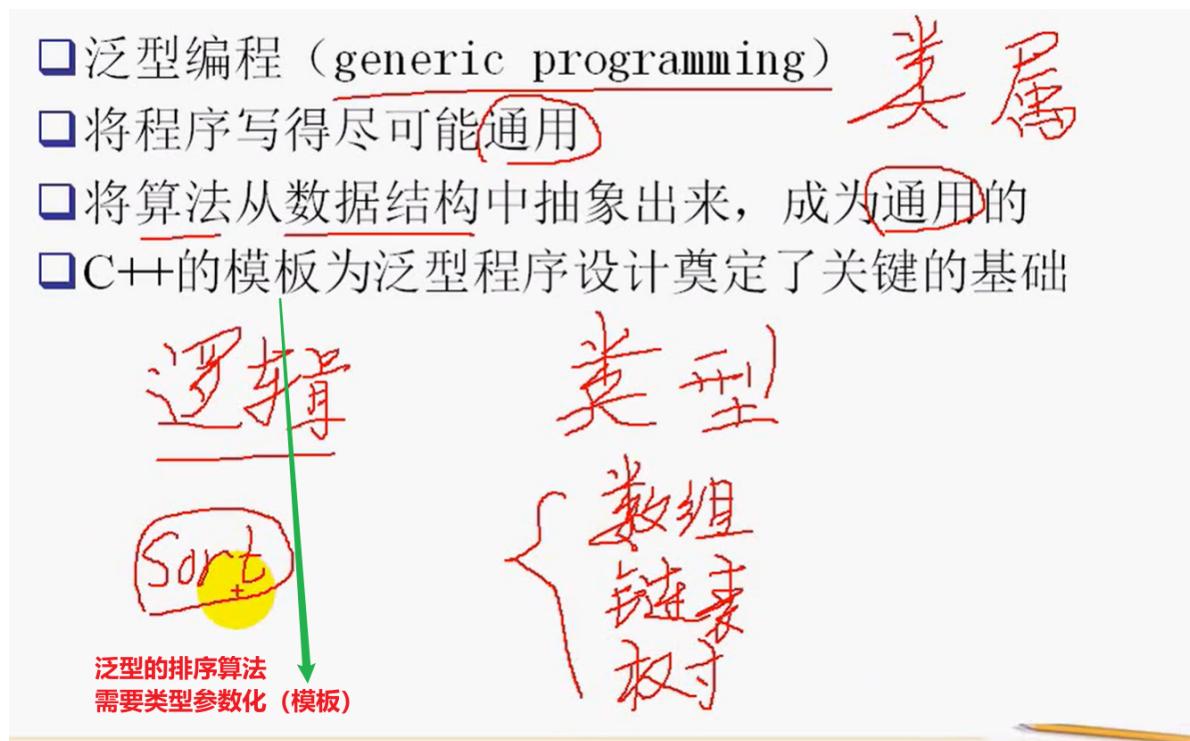
- 默认
- 或者显示指定

2. 类模板必须显示实例化为模板类，才可以实例化对象

- 模板也可以传递非类型参数 `template <typename T, int MAXSIZE>`

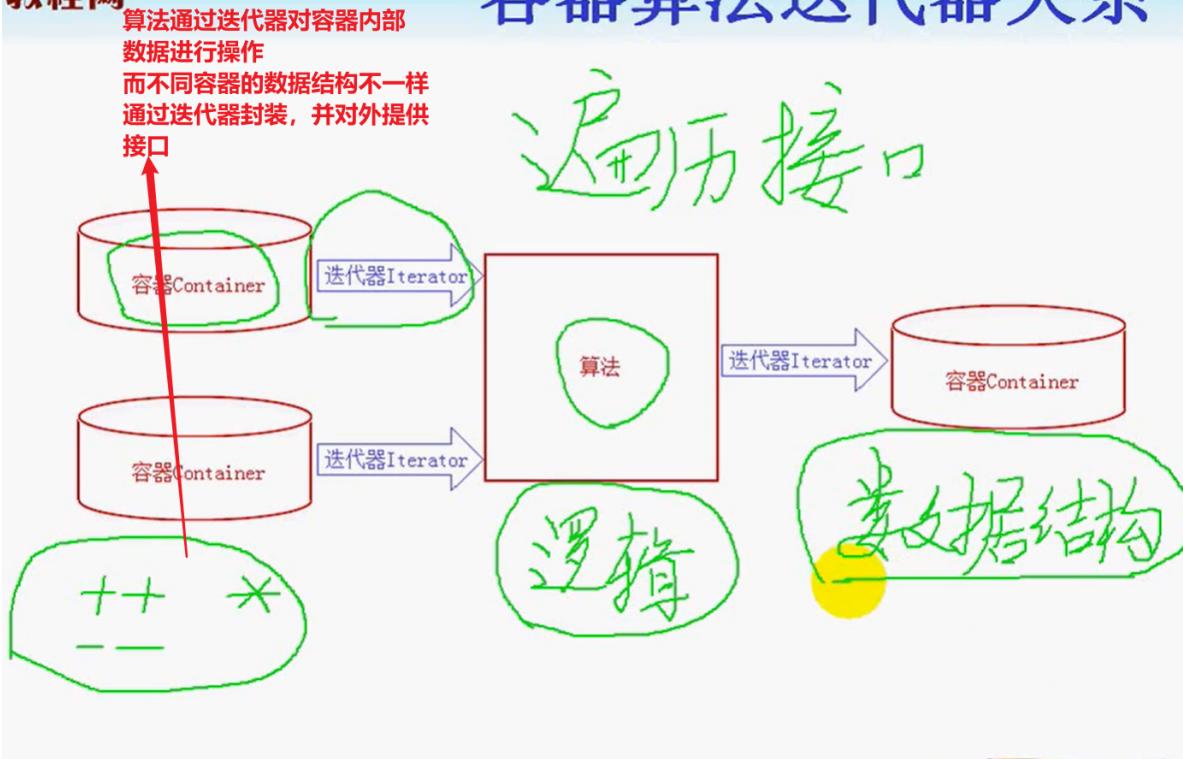
2.2 泛型程序设计 (generic programming)

1. 将程序写得尽可能通用
2. 将算法从数据结构中抽象出来，称为通用的。
3. C++的模板为泛型编程设计奠定了关键的基础



2.3 STL的六大组件

容器算法迭代器关系



2.3.1 容器

提供各种基本数据结构

容器

- 容器类是容纳、包含一组元素或元素集合的对象
- 七种基本容器：
 □ 向量（vector）、双端队列（deque）、列表（list）、
 集合（set）、多重集合（multiset）、映射（map）和
 多重映射（multimap） *双向链表
key, value*
- 标准容器的成员绝大部分都具有共同的名称

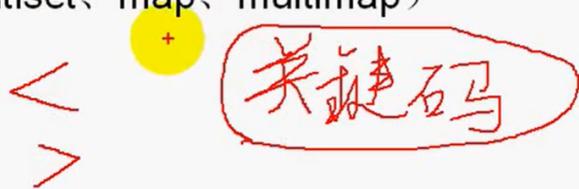
动态数组
连续
片

□ 序列式容器

- 序列式容器 **Sequence containers**, 其中每个元素均有固定位置——取决于插入时机和地点, 和元素值无关。
(vector、deque、list)

□ 关联式容器

- 关联式容器 **Associative containers**, 元素位置取决于特定的排序准则以及元素值, 和插入次序无关。(set、multiset、map、multimap)



2.3.2 适配器

代码复用的方式, 不是通过继承。而是适配。

可以改变容器、迭代器、函数对象接口的一种组件

□ 适配器是一种 接口类

- 为已有的类提供新的接口
- 目的是简化、约束、使之安全、隐藏或者改变被修改类提供的服务集合

stack queue

□ 三种类型的适配器:

- 容器适配器: 用来扩展7种基本容器, 它们和顺序容器相结合构成栈、队列和优先队列容器
- 迭代器适配器 (反向迭代器、插入迭代器、IO流迭代器)
- 函数适配器 (函数对象适配器、成员函数适配器、普通函数适配器)

bind

2.3.3 函数对象

以类模板的形式提供: 使得类使用起来像一个函数 (需要重载 **operator()**)

2.3.4 分配器allocator

2.3.4.1 内存池设计实现

2.3.5 算法

提供了各种基本的算法

- sort
- search

2.3.6 迭代器

用于连接 `containers` 和 `algorithms`

The screenshot shows a slide from a C++ iterator tutorial. The title '迭代器' (Iterator) is at the top right. On the left, there's a logo for 'C++ 教程网' (C++ Tutorial Network). The main content is a list of bullet points explaining what iterators are and how they work.

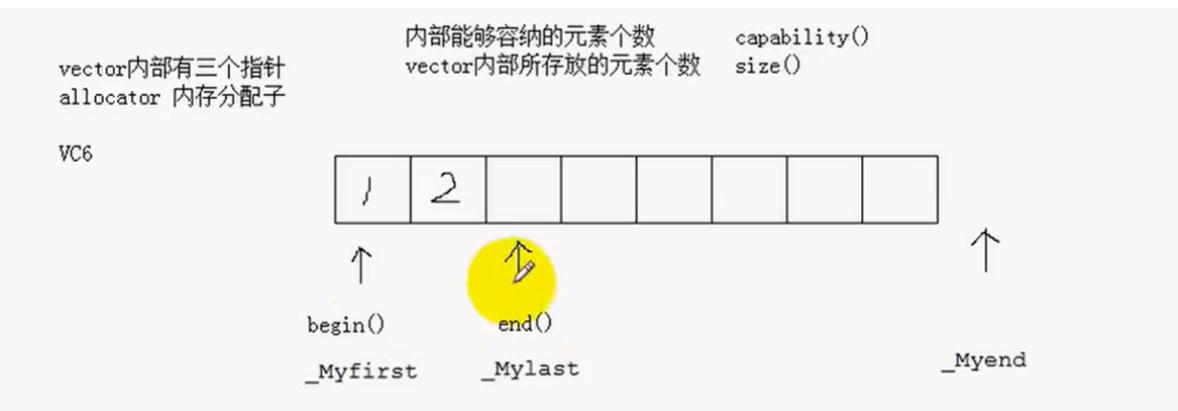
- 迭代器 Iterators, 用来在一个对象群集 (collection of objects) 的元素上进行遍历。这个对象群集或许是个容器, 或许是容器的一部分。迭代器的主要好处是, 为所有容器提供了一组很小的公共接口。迭代器以`++`进行累进, 以`*`进行提领, 因而它类似于指针, 我们可以把它视为一种 smart pointer
- 比如`++`操作可以遍历至群集内的下一个元素。至于如何做到, 取决于容器内部的数据组织形式。
- 每种容器都提供了自己的迭代器, 而这些迭代器能够了解容器内部的数据结构。

2.4 源码分析

2.4.1 vector源码分析

vector内部有三个指针

1. `_Myfirst`
2. `_Mylast`
3. `_Myend`



vector内部能够容纳的元素个数: `capacity`

vector内部所存放的元素个数: `size`

```

1 size_type size() const
2 { // return length of sequence
3     return (_Mylast - _Myfirst);
4 }
5
6 size_type capacity() const
7 { // return length of sequence
8     return (_Myfirst == 0 ? 0 : _Myend - _Myfirst);
9 }
10
11 size_type max_size() const
12 {
13
14 }
```

1. 通常情况下, `capacity`大于等于 `size`

为的是预留与部分空间来准备给新插入的元素。 (避免每次插入都动态扩张)

每个vector有个 `max_size()`

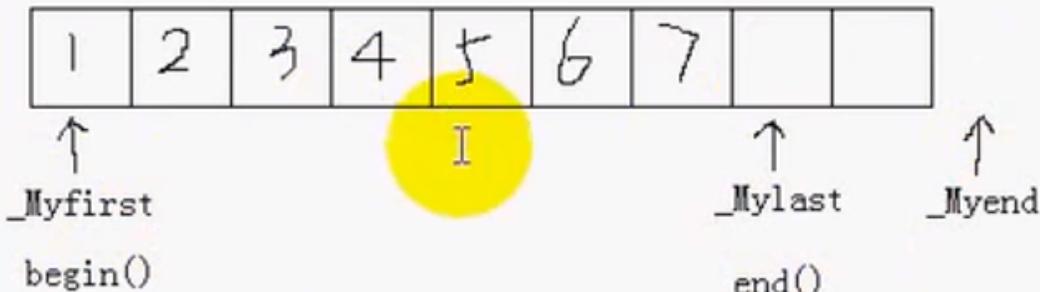
2.4.1.1 vector的push_back所做的工作

1. 判断 `size()`是否小于 `capacity()`
2. 如果小于等于, 就直接 `insert(end(), val)`
3. 否则动态扩容, 然后 `insert`

2.4.1.2 vector的capacity和size

1. capacity(): 容量，当前向量能够容纳的元素的个数
2. size(): 当前向量实际存储的元素的个数
3. capacity() >= size(): 向量通常缓存了一部分内存空间，用来容纳更多的元素。
这样，在下次插入新元素的时候，就不必重新分配内存。提高了插入速度

```
vector<int> v
```



2.4.1.3 vector元素的删除erase和remove

```
1 // 源码示例
2 // sequence (1)
3 string& erase (size_t pos = 0, size_t len = npos);
4 // character (2)
5 iterator erase (const_iterator p);
6 // range (3)
7 iterator erase (const_iterator first, const_iterator
last);
8
9 // remove的工作
10 template <class ForwardIterator, class T>
11 ForwardIterator remove (ForwardIterator first,
ForwardIterator last,
12                         const T& val)
13 {
14     ForwardIterator result = first;
15     while (first!=last) {
16         if (!(*first == val)) {
17             if (result!=first)
18                 *result = move(*first);
19                 ++result;
20         }
21         ++first;
22     }
```

```
23 |     return result; // 返回移除后的尾部迭代器。之前的元素都不是  
24 |     val  
25 | } // 这时候就可以直接[result, end)删除值为val的所有元素
```

1. 删除某一个位置的元素
2. 删除某个区间的元素
3. 删除某个值的元素：
 - remove
 - erase

```
1 | v.erase(remove(v.begin(), v.end(), target), v.end());
```

2.4.1.4 小结

1. vector的动态扩容机制

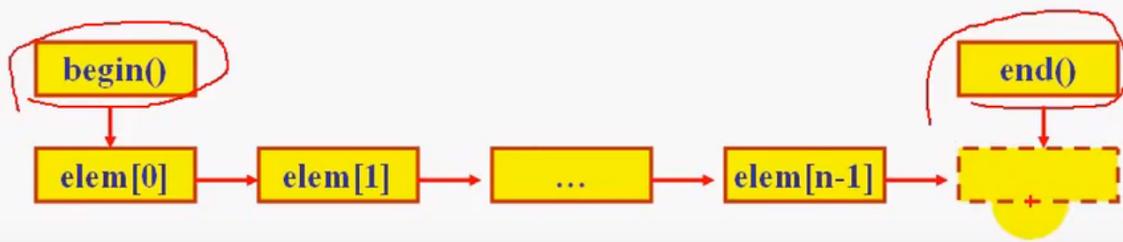
```
1 | // 在vs2008中实现  
2 | _Capacity = max_size() - Capacity / 2 < _Capacity ? 0  
| : _Capacity + _Capacity / 2; // try to grow by 50%  
3 |  
4 | // vc6中动态扩容机制：成倍增长  
5 | size_type _N = size() + (_M < size() ? size() : _M);  
| // 因为当前的size()后面  
6 | iterator _S = allocator.allocate(_N, (void*)0); // 分  
配空间
```

拷贝移动机制如何实现？

2. 如何实现动态连续空间？
 - 如果原有向量的容量capacity不够使用，那就需要扩容
 - 尾部插入，在新开辟的内存中，直接在尾部位置插入元素，然后将原向量拷贝
 - 中间插入，在新开辟的内存位置头部插入元素，然后将原向量拷贝到插入元素后

2.4.2 迭代器

- 迭代器是泛型指针 $(\ast \rightarrow ++--)$
- 普通指针可以指向内存中的一个地址
- 迭代器可以指向容器中的一个位置
- STL的每一个容器类模版中，都定义了一组对应的迭代器类。使用迭代器，算法函数可以访问容器中指定位置的元素，而无需关心元素的具体类型。



2.4.2.1 迭代器类型

- 输入迭代器
 - 可以用来从序列中读取数据
- 输出迭代器
 - 允许向序列中写入数据
- 前向迭代器
 - 既是输入迭代器又是输出迭代器，并且可以对序列进行单向的遍历
- 双向迭代器
 - 与前向迭代器相似，但是在两个方向上都可以对数据遍历
- 随机访问迭代器
 - 也是双向迭代器，但能够在序列中的任意两个位置之间进行跳转

$\ast p$ ++
 $\ast p =$ ++
 ++

++ --

迭代器操作和分类					
种类:	Output	input	forward	bidirectional	random-access
缩写:	Out	In	For	Bi	Ran
Read:		=*p	=*p	=*p	=*p
Access:		->	->	->	-> []
Write:	*p=		*p=	*p=	*p=
Iteration:	++	++	++	++ --	++ -- + - += -=
Comparison:		== !=	== !=	== !=	== != < > >= <=

注意: 这些迭代器都重载了
== 和 !=, 因此在迭代器遍历的时候, 建议使用

2.4.3 算法

C++
教程网

算法

- 算法是以函数模板的形式实现的。常用的算法涉及到比较、交换、查找、搜索、复制、修改、移除、反转、排序、合并等等。
- 算法并非容器类型的成员函数, 而是一些全局函数, 要与迭代器一起搭配使用。
- 算法的优势在于只需实作一份, 可以适应所有的容器, 不必为每一种容器量订制。也可以与用户定义的容器搭配。

- 非变动性算法既不改变元素次序，也不改变元素值。
- 变动性算法，要么直接改变元素值，要么就是在复制到另一个区间的过程中改变元素值。如果是第二种情况，原区间不会发生变化
- 移除性算法是一种特殊的变动性算法。移除性算法是在一区间内移除某些元素，这些算法并不能改变元素的数量，它们只是以逻辑上的思考，将原本置于后面的“不需要移除元素”向前移动，覆盖那些被移除元素而已。它们都返回新区间的逻辑终点。移除性算法也可以在复制的过程中执行移除。注意，目标区间不能是关联式容器。
- 变序性算法改变元素次序，但不改变元素值。这些算法不能用于关联式容器，因为关联式容器中，元素有固定的次序。
- 排序算法，排序算法是一种特殊的变序算法。但比一般的变序性算法更复杂，花费更多的时间
- 已序区间算法，一般来说这些算法的结果，仍然是已序的。
- 用来处理数值的算法，需要加上头文件 #include<numeric>

2.4.3.1 非变动性算法

非变动性算法

非变动性算法	
for_each()	对每个元素执行某操作
count()	返回元素个数
count_if()	返回满足某一条件的元素个数
min_element()	返回最小值元素
max_element()	返回最大值元素
find()	搜索等于某个值的第一个元素
find_if()	搜索满足某个条件的第一个元素
search_n()	搜索具有某特性的第一段n个连续元素
search()	搜索某个子区间第一次出现位置
find_end()	搜索某个子区间最后一次出现位置
find_first_of()	搜索等于“某数个值之一”的第一元素
adjacent_find()	搜索连续两个相等的元素
equal()	判断两区间是否相等
mismatch()	返回两个序列的各组对应元素中，第一对不相等元素
5. lexicographical_compare()	判断某一序列在“字典顺序”下是www.cppcourse.co

2.4.3.2 变动性算法

变动性算法	
for_each()	对每个元素执行某操作
copy()	从第一个元素开始，复制某段区间
copy_backward()	从最后一个元素开始，复制某段区间
transform()	变动(并复制)元素，将两个区间的元素合并
merge()	合并两个区间
swap_ranges()	交换两区间内的元素
fill()	以给定值替换每一个元素
fill_n()	以给定值替换n个元素
generate()	以某项操作的结果替换每一个元素
generate_n()	以某项操作的结果替换n个元素
replace()	将具有某特定值的元素替换为另一个值
replace_if()	将符合某准则的元素替换为另一个值
replace_copy()	复制整个区间，同时并将具有某特定值的元素替换为另一个值
replace_copy_if()	复制整个区间，同时并将符合某个条件的元素替换为另一个值

2.4.3.3 已序区间算法

已序区间算法	
binary_search()	判断某区间内是否包含某个元素
includes()	判断某区间内的每一个元素是否都涵盖于另一区间中
lower_bound()	搜索第一个"大于等于给定值"的元素
upper_bound()	搜索第一个"大于给定值"的元素
equal_range()	返回"等于给定值"的所有元素构成的区间
merge()	将两个区间合并
set_union()	求两个区间的并集
set_intersection()	求两个区间的交集
set_difference()	求位于第一区间但不位于第二区间的所有元素，形成一个已序区间
set_symmetric_difference()	找出只出现于两区间之一的所有元素，形成一个已序区间
inplace_merge()	将两个连续的已序区间合并

2.4.3.4 数值算法

数值算法

accumulate()	组合所有元素(求总和, 求乘积...)
inner_product()	组合两区间内的所有元素
adjacent_difference()	将每个元素和其前一元素组合
partial_sum()	将每个元素和其先前的所有元素组合

2.4.3.5 算法尾词

□_if

- 比如find (按某个值来查找), find_if (按某个条件来查
找) ③ 逐对象

□_copy

- 这个尾词用来表示在算法中, 元素不光被操作, 还会
被复制到目标区间。比如reverse、reverse_copy

2.5 函数对象

2.5.1 函数对象

- 函数对象 (function object) 也称为仿函数 (functor)
- 一个行为类似函数的对象，它可以没有参数，也可以带有若干参数。
- 任何重载了调用运算符operator() 的类的对像都满足函数对象的特征
- 函数对象可以把它称之为smart function。状态
- STL中也定义了一些标准的函数对象，如果以功能划分，可以分为算术运算、关系运算、逻辑运算三大类。为了调用这些标准函数对象，需要包含头文件 <functional>。

函数对象：一个重载了operator() 的类对象。使用起来像一个函数

```
#include <iostream>
using namespace std;
class CFunObj{
public:
    void operator()()
    {
        cout<<"hello,function object!"<<endl;
    }
};

int main(){
    CFunObj fo;
    fo();
    CFunObj()();
    return 0;
}
```

2.5.2 函数对象与容器

map容器和函数对象的使用

```
1 // 如何定义map的<key, val, pred>
2 map<int, string, less<int> > mp1;
3 map<int, string, greater<int> > mp2;
4
5 // 自定义结构体
6 struct MyGreater
7 {
```

```
8     bool operator()(int left, int right)
9     {
10         return left > right;
11     }
12 }
13
14 map<int, string, MyGreater> mp3;
```

2.5.3 函数对象与算法

函数对象通过类或者结构体定义的时候，可以有成员变量

```
1 class AddObj
2 {
3     AddObj(int number) : number_(number) {}
4
5     void operator()(int &n)
6     {
7         n += number_;
8     }
9 private:
10    number_;
11 }
12
13 vector<int> vec = {1,2,3,4,5};
14 for_each(v.begin(), v.end(), AddObj(3)); // 实现vec每个元素+3
```

2.5.4 STL中内置的函数对象

断言函数对象		
equal_to	Binary	arg1==arg2
not_equal_to	Binary	arg1!=arg2
greater	Binary	arg1>arg2
less	Binary	arg1<arg2
greater_equal	Binary	arg1>=arg2
less_equal	Binary	arg1<=arg2
logic_and	Binary	arg1&&arg2
logic_or	Binary	arg1 arg2
logic_not	Unary	!arg

算术操作函数对象		
plus	Binary	arg1+arg2
minus	Binary	arg1-arg2
multiplies	Binary	arg1*arg2
divides	Binary	arg1/arg2
modulus	Binary	arg1%arg2
negate	Unary	-arg

可以自己编写一个函数作为算法的参数

2.6 适配器

□ 三种类型的适配器：

- 容器适配器：用来扩展7种基本容器，利用基本容器扩展形成了栈、队列和优先级队列
- 迭代器适配器（反向迭代器、插入迭代器、IO流迭代器）
- 函数适配器

2.6.1 容器适配器

STL中由容器适配器适配而来的容器有：

1. stack
2. queue
3. priority_queue

基本容器：vector, list, deque, set, multiset, map, multimap

栈stack的默认实现：由deque适配而来。借用deque的接口，来封装实现

- top

- push
- pop
- empty

优先队列: `priority_queue` 实现

1. 需要实现的接口

- empty
- size
- top
- push
- pop
- emplace

2. 声明使用:

```
1 priority_queue<int> pq; // 默认大根堆less<int>
2 priority_queue<int, vector<int>, greater<int> > pq2;
// 小根堆
```

3. 构造函数类型

- 默认构造函数
- 拷贝构造

4. 对一个数组进行堆化: `make_heap` 实现一个最大堆 (二叉堆)

5. 堆排序的使用

```
1 void testQP()
2 {
3     int a[] = {5, 1, 2, 4, 3};
4     make_heap(a, a + 5, less<int>()); // make_heap是一个
// 大根堆
5     cout << "after make_heap" << endl;
6     copy(a, a+5, ostream_iterator<int>(cout, " "));
7     cout << endl;
8
9     cout << "after sort_heap" << endl;
10    sort_heap(a, a+5, less<int>()); // 对于大根堆, 只能实现
// 从小到大递增排序
11    copy(a, a+5, ostream_iterator<int>(cout, " "));
12    cout << endl;
13 }
```

要点: 排序时的谓词要和 `make_heap` 的谓词相匹配

2.6.2 函数适配器：用于将函数适配成函数对象，用于algorithm

函数适配器：能够将仿函数和另一个仿函数（或某个值、某个一般函数）结合起来

C++ 教程网

函数适配器示例

```
//计算奇数元素的个数
// 这里的bind2nd将二元函数对象modulus转换为一元函数对象。
//bind2nd(op, value) (param)相当于op(param, value)
cout << count_if(v.begin(), v.end(),
    bind2nd(modulus<int>(),2)) << endl;
```

op是二元函数对象
value是默认绑定的第二个参数
此时，适配成了一个一元函数对象
可接受一个参数value

函数适配器bind2nd的底层实现：

- 一个模板类，然后将传入的第二个参数赋值给second_argument
- 进而返回一个构造的binder2nd对象。
 - 在binder2nd类模板中，构造函数进行初始化
 - 并且重载了operator()，使得函数调用发生二元函数的调用。但是第二个函数参数默认

函数适配器bind1st的底层实现：可以绑定一个二元函数对象，绑定的是第一个参数

```
1 vector<int> vec = {1, 2, 3, 4, 5};
2
3 cout << count_if(vec.begin(), vec.end(),
4     bind2nd(modulus<int>(), 2)) << endl;
5 cout << count_if(vec.begin(), vec.end(),
6                     bind1st(less<int>(), 4)) << endl; // 找
7 到vec中大于4的元素个数
8
9 // 因为less是一个二元谓词，绑定第一个参数表明，return 4 < 传入的
参数
```

2.6.3 针对成员函数的适配器 mem_fun

1. mem_fun_ref（适用于成员函数）：一元函数适配器，实现将一个不带参数的成员函数，适配成一元函数对象
2. mem_fun（适用于成员函数指针）

```

1 template<class _Result,
2         class _Ty> inline
3     const_mem_fun_ref_t<_Result, _Ty>
4             mem_fun_ref(_Result(_Ty::*_Pm)() const)
// 接受一个类成员函数，不带参数
5     {
6         return (std::const_mem_fn_ref_t<_Result,
7             _Ty>(_Pm));
8     }

```

```

template<class _Result,
class _Ty>
class const_mem_fn_ref_t
    : public unary_function<_Ty, _Result>
{ // functor adapter (*left.*pfunc)(), const *pfunc
public:
    explicit const_mem_fn_ref_t(_Result (_Ty::*_Pm)() const)
        : _Pmemfun(_Pm)
    { // construct from pointer
    }

    _Result operator()(const _Ty& _Left) const
    { // call function
        return ((_Left.*_Pmemfun)());
    }

private:
    _Result (_Ty::*_Pmemfun)() const; // the member function pointer
};

```

也是重载了operator()

也可以将一元成员函数转换成二元函数对象。

2.6.4 针对一般函数的函数适配器 `ptr_fun`

```

1 int main(void)
2 {
3     char* a[] = {"", "BBB", "CCC"};
4     vector<char*> v(a, a+2);
5     vector<char*>::iterator it;
6
7     it = find_if(v.begin(), v.end(),
bind2nd(ptr_fun(strcmp), ""));
8     // 查找第一个空字符串
9     if (it != v.end())
10         cout << *it << endl;
11
12     return 0;
13 }

```

2.6.5 not1

```
using namespace std;
bool check(int elem)
{
    return elem < 3;
}

int main(void)
{
    int a[] = {1, 2, 3, 4, 5};
    vector<int> v(a, a+5);

    vector<int>::iterator it;
    it = find_if(v.begin(), v.end(), not1(ptr_fun(check)));
    if (it != v.end())
        cout<<*it<<endl;
    return 0;                                查找第一个大于等于3的元素
}
```

2.6.6 反向迭代器适配器rbegin, rend

2.6.7 插入迭代器

1. back_inserter_iterator实现：重载运算符`*`, `=`

- operator=的操作是：push_back

```
1 template <class Container>
2     class back_insert_iterator :
3         public
4             iterator<output_iterator_tag,void,void,void,void>
5     {
6     protected:
7         Container* container;
8
9     public:
10        typedef Container container_type;
11        explicit back_insert_iterator (Container& x) :
12            container(&x) {}
13        back_insert_iterator<Container>& operator= (const
14            typename Container::value_type& value)
15        { container->push_back(value); return *this; }
16        back_insert_iterator<Container>& operator=
17            (typename Container::value_type&& value)
18        { container->push_back(std::move(value)); return
19            *this; }
20        back_insert_iterator<Container>& operator* ()
```

```

16     { return *this; }
17     back_insert_iterator<Container>& operator++ ()
18     { return *this; }
19     back_insert_iterator<Container> operator++ (int)
20     { return *this; }
21 };
22
23 // 快速获取back_inserter
24 template <class Container>
25     back_insert_iterator<Container> back_inserter
26     (Container& x);
27 // 函数模板，可以根据传入参数类型自动推导模板参数类型。而不需要传递类型

```

2. inserter

3. front_inserter: 要求容器的迭代器是双向迭代器, 才能发生适配

2.6.8 IO流迭代器

1. 输出流迭代器: ostream_iterator

```

1 // CLASS TEMPLATE ostream_iterator
2 template <class _Ty, class _Elem = char, class
3     _Traits = char_traits<_Elem>>
4 class ostream_iterator { // wrap _Ty inserts to
5     output stream as output iterator
6 public:
7     using iterator_category = output_iterator_tag;
8     using value_type = void;
9     using difference_type = void;
10    using pointer = void;
11    using reference = void;
12
13    using char_type = _Elem;
14    using traits_type = _Traits;
15    using ostream_type = basic_ostream<_Elem,
16    _Traits>;
17
18    ostream_iterator(ostream_type& _Ostr, const
19    _Elem* const _Delim = nullptr)
20        : _Mydelim(_Delim), _Myostr(_STD
21        addressof(_Ostr)) {}

```

```

17     ostream_iterator& operator=(const _Ty& _val) {
18         // insert value into output stream, followed by
19         // delimiter
20         *_Myostr << _val;
21         if (_Mydelim) {
22             *_Myostr << _Mydelim;
23         }
24         return *this;
25     }
26
27     _NODISCARD ostream_iterator& operator*() { // pretend to return designated value
28         return *this;
29     }
30
31     ostream_iterator& operator++() { // pretend to preincrement
32         return *this;
33     }
34
35     ostream_iterator& operator++(int) { // pretend to postincrement
36         return *this;
37     }
38
39 protected:
40     const _Elem* _Mydelim; // pointer to delimiter
41     string (NB: not freed)
42     ostream_type* _Myostr; // pointer to output stream
43 };

```

2. 输入流迭代器: istream_iterator

重载的运算符operator

- =
- *
- ->

- ++
- ==
- !=

使用：

1 |

2.7 STL中适配器的实现

1. 适配的过程中，重载原有容器、迭代器、函数的函数运算符
2. 容器适配器的实现：借用标准容器的功能，内部含有一个标准容器对象。进而，在封装适配的容器中，调用相关的容器对象接口。

2.8 小结

2.8.1 容器的对比

1. 序列式容器对比

- vector在头部与中间插入删除效率较低，在尾部插入删除效率很高
- vector能以O (1) 时间复杂度访问元素
- list在中间插入和删除效率很高。但是遍历访问某一个元素的效率很低（线性复杂度）
- deque容器在头部和尾部插入与删除效率较高。常数级别访问，不如vector高
- vector在内存上是连续的，而list是不连续的。deque分片连续

2. 序列容器的选择

- list：当需要频繁的在中间插入删除元素的时候，同时不需要过多的进行长距离跳转的情况
- deque：频繁在头尾插入、删除元素
- vector：快速访问、不频繁的向中间插入删除元素

value_type	元素类型
allocator_type	内存管理器类型
size_type	下标, 元素计数类型
difference_type	循环子之间差的类型
iterator	循环子, 相当于 value_type*
const_iterator	常循环子, 相当于 const value_type*
reverse_iterator	逆序循环子, 相当于 value_type*
const_reverse_iterator	常逆序循环子, 相当于 const value_type*
reference	元素引用, 相当于 value_type&
const_reference	元素常引用, 相当于 const value_type&
key_type	关键字类型 (只用于关联容器)
mapped_type	映射值类型 (只用于关联容器)
key_compare	比较标准类型 (只用于关联容器)

关联式容器

循环子操作	
begin()	指向第一个元素
end()	指向最后一个元素的后一个位置
rbegin()	指向逆序的第一个元素
rend()	指向逆序的最后一个元素的后一个位置
访问元素操作	
front()	访问第一个元素
back()	访问最后一个元素
[]	无测试的下标访问 (不用于 list) <i>map</i>
at()	有测试的下标访问 (只用于 vector 和 deque)

堆栈和队列操作	
push_back()	将新元素加入到尾部
pop_back()	移出最后一个元素
push_front()	将新元素加入头部 (只用于 list 和 deque)
pop_front()	移出第一个元素 (只用于 list 和 deque)
表操作	
insert(p,x)	将元素x加入到 p 之前
insert(p,n,x)	在 p 之前加入 n 个 x 的拷贝
insert(p,first,last)	在 p 之前加入 区间 [first:last) 中的序列
erase(p)	删除p处的元素
erase(first,last)	删除区间[first, last)中的序列
clear()	清除所有的元素

其他操作	
size()	获取元素个数
empty()	测试包容器是否为空
max_size()	最大可能的包容器的大小
capacity()	为向量包容器分配的空间 (只用于 vector)
reserve()	为扩充向量包容器保留空间 (只用于 vector)
resize()	改变包容器的大小 (只用于 vector, list 和 deque)
swap()	交换两个包容器中的元素
get_allocator()	获取包容器的内存管理器的副本
==	测试两个包容器的内容是否相等
!=	测试两个包容器的内容是否不同
<	测试一个包容器是否在另一个包容器字典序之前

2.8.2 迭代器

反向迭代器实现：

1. 迭代器适配： `std::reverse_iterator<iterator>`
2. 反向迭代器的基类拥有一个正向迭代器成员。

```

1 reverse_iterator rbegin()
2 {
3     return (reverse_iterator(end()));

```

```

4 }
5
6 const_reverse_iterator rbegin() const
7 {
8     return (const_reverse_iterator(end()));
9 }
10
11 _NODISCARD _CONSTEXPR17 reference operator*() const
12 {
13     _BIdIt _Tmp = current;
14     return *--_Tmp;
15 }
```

3. 关键在于`*`, `++` 运算符重载

```

1 reference __CLR_OR_THIS_CALL operator*() const
2 {
3     _RanIt _Tmp = current;
4     return (*--_Tmp);
5 }
```

2.8.3 算法示例

1. 非变动性算法

for_each

```

Fn1 for_each(_InIt _First, _InIt _Last, _Fn1 _Func)
{
    // perform function for each element
    _DEBUG_RANGE(_First, _Last);
    _DEBUG_POINTER(_Func);
    _CHECKED_BASE_TYPE(_InIt) _ChkFirst(_CHECKED_BASE(_First));
    _CHECKED_BASE_TYPE(_InIt) _ChkLast(_CHECKED_BASE(_Last));
    for (; _ChkFirst != _ChkLast; ++_ChkFirst)
        _Func(*_ChkFirst);
    return (_Func); | 1
}
```

min_element

max_element

find

find_if

```
1 // 一元谓词: pred, 输入参数类型为value_type
2 template<class InputIterator, class
3 UnaryPredicate>
4     InputIterator find_if (InputIterator first,
5 InputIterator last, UnaryPredicate pred)
6 {
7     while (first!=last) {
8         if (pred(*first)) return first;
9         ++first;
10    }
11 }
```

search: 查找第一次出现的位置

```
1 // version1: equality (1)
2 template <class ForwardIterator1, class
3 ForwardIterator2>
4 ForwardIterator1 search (ForwardIterator1 first1,
5 ForwardIterator1 last1, ForwardIterator2 first2,
6 ForwardIterator2 last2);
7 // version2: 自定义比较规则, predicate (2)
8 template <class ForwardIterator1, class
9 ForwardIterator2,
10           class BinaryPredicate>
11 ForwardIterator1 search (ForwardIterator1 first1,
12 ForwardIterator1 last1,
13                         ForwardIterator2 first2,
14                         ForwardIterator2 last2,
15                         BinaryPredicate pred);
16 // 具体的逻辑
17 template<class ForwardIterator1, class
18 ForwardIterator2>
19 ForwardIterator1 search ( ForwardIterator1
20 first1,
21                         ForwardIterator1 last1,
22                         ForwardIterator2 first2,
23                         ForwardIterator2 last2)
24 {
25     if (first2==last2) return first1; // specified in C++11
```

```
18
19     while (first1!=last1)
20     {
21         ForwardIterator1 it1 = first1;
22         ForwardIterator2 it2 = first2;
23         while (*it1==*it2) {
24             // or: while (pred(*it1,*it2)) for
25             ++it1; ++it2;
26             if (it2==last2) return first1;
27             if (it1==last1) return last1;
28             //
29         }
30         ++first1;
31     }
32     return last1;
33 }
```

copy

```
1 template<class InputIterator, class
2 OutputIterator>
3     OutputIterator copy (InputIterator first,
4     InputIterator last, OutputIterator result)
5     {
6         while (first!=last) {
7             *result = *first;
8             ++result; ++first;
9         }
10        return result;
11    }
```

copy_backward

```

1 template<class BidirectionalIterator1, class
2 BidirectionalIterator2>
3 BidirectionalIterator2 copy_backward (
4     BidirectionalIterator1 first,
5
6     BidirectionalIterator1 last,
7
8     BidirectionalIterator2 result )
9
10 // 代码示例:
11 copy_backward(v1.begin(), v1.end(), v2.end());

```

transform

```

1 // unary operation(1)
2 template <class InputIterator, class
3 OutputIterator, class UnaryOperation>
4 OutputIterator transform (InputIterator first1,
5 InputIterator last1,
6
7             OutputIterator
8 result, UnaryOperation op);
9 // binary operation(2)
10 template <class InputIterator1, class
11 InputIterator2,
12
13             class OutputIterator, class
14 BinaryOperation>
15 OutputIterator transform (InputIterator1 first1,
16 InputIterator1 last1,
17
18             InputIterator2 first2,
19             OutputIterator result,
20
21             BinaryOperation
22 binary_op);

```

- remove: 所做的操作，有一个 `remove_copy` 的操作。**并不会直接删除元素，而是移到v.begin() + k, v.end()区间**
- 首先查找给定值第一个位置，然后遍历后面的元素。将非移除元素拷贝到前面。覆盖前面的元素（最后k个元素为k个remove的值）

```

1 template <class ForwardIterator, class T>
2 ForwardIterator remove (ForwardIterator first,
3 ForwardIterator last, const T& val)
4 {
5     ForwardIterator result = first;
6     while (first!=last) {
7         if (!(*first == val)) {
8             if (result!=first)
9                 *result = move(*first);
10            ++result;
11        }
12        ++first;
13    }
14    return result;
15 }
```

rotate使用

```

int main(void)
{
    int a[] = { 1, 2, 3, 4, 5, 6 };
    vector<int> v(a, a+6);

    for_each(v.begin(), v.end(), print_element);
    cout<<endl;

    rotate(v.begin(), v.begin()+2, v.end()-1);
    for_each(v.begin(), v.end(), print_element);
    cout<<endl;

    return 0;
}
```

lower_bound和upper_bound

lower_bound()	搜索第一个“大于等于给定值”的元素 如果要插入给定值，保持区间有序性，返回第一个可插入的位置
upper_bound()	搜索第一个“大于给定值”的元素 如果要插入给定值，保持区间有序性，返回最后一个可插入的位置

数值算法：accumulate

```

1 // sum (1)
2 template <class InputIterator, class T>
3 T accumulate (InputIterator first, InputIterator
4 last, T init);
5 // custom (2)
6 template <class InputIterator, class T, class
BinaryOperation>
7 T accumulate (InputIterator first, InputIterator
8 last, T init,
```

```

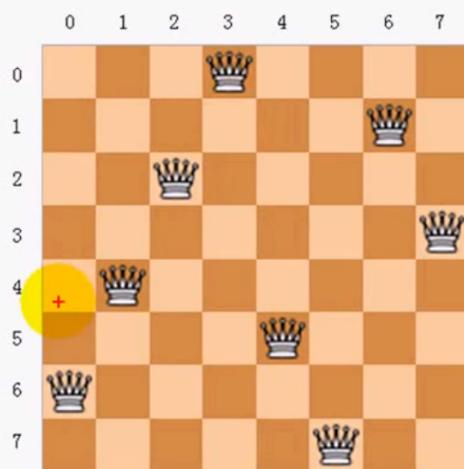
7             BinaryOperation binary_op);
8
9 template <class InputIterator, class T>
10 T accumulate (InputIterator first, InputIterator
11 last, T init)
12 {
13     while (first!=last) {
14         init = init + *first;
15         // or: init=binary_op(init,*first) for the
16         // binary_op version
17         ++first;
18     }
19
20 // 第二个版本的作用：可以实现累乘，累除。需要提供函数对象
21 int multi(int a, int b)
22 {
23     return a*b;
24 }
25
26 accumulate(v.begin(), v.end(), val, multi);

```

2.8.3.1 用STL算法解决八皇后问题

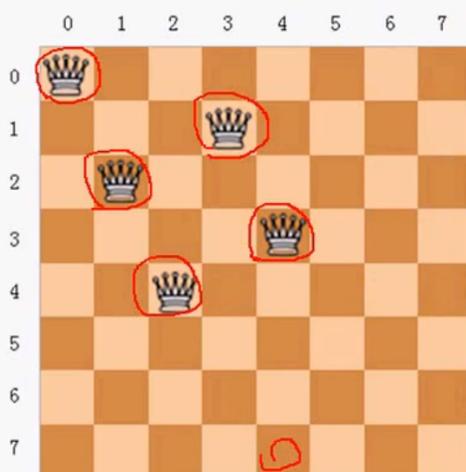
C++
教程网

八皇后问题



八皇后问题是一个以国际象棋为背景的问题：如何能够在 8×8 的国际象棋棋盘上放置八个皇后，使得任何一个皇后都无法直接吃掉其他的皇后？为了达到此目的，任两个皇后都不能处于同一条横行、纵行或斜线上。

(y, x)



N皇后问题

- 1) 从第一列开始，为皇后找到安全位置 $|y_1 - y_2| \neq |x_1 - x_2|$ ，然后跳到下一列
- 2) 如果在第n列出现死胡同，如果该列为第一列，棋局失败，否则后退到上一列，在进行回溯
- 3) 如果在第8列上找到了安全位置，则棋局成功。

1. 用stl的算法：`next_permutation`得到下一个排列

2. 对于生成的每一个排列，用(y, x)来记录坐标。其中y为行号，x为列号

- 对于 $[0, 1, 2, 3, 4, 7, 6, 5]$ ，坐标表示为
- $[[0,0], [1,1], [2,2], [3,3], [4,4], [5,7], [6,6], [7,5]]$
- 这个坐标已经保证了N个皇后不在同一行或者同一列。
- 当 $|y_1 - y_2| = |x_1 - x_2|$ ，表明他们在同一条斜线上。斜率为1。

2.8.4 inserter、back_inserter和front_inserter

1. front_inserter

A *front-insert iterator* is a special type of *output iterator* designed to allow *algorithms* that usually overwrite elements (such as *copy*) to instead insert new elements automatically at the beginning of the container.

2. back_inserter

A *back-insert iterator* is a special type of *output iterator* designed to allow *algorithms* that usually overwrite elements (such as *copy*) to instead insert new elements automatically at the end of the container.

3. inserter

Constructs an *insert iterator* that inserts new elements into x in successive locations starting at the position pointed by it.

```
1 template <class Container>
2     insert_iterator<Container> inserter (Container& x,
3         typename Container::iterator it);
```

2.8.5 函数适配器分类

STL中可被适配的函数必须是 `unary_function` 或者 `binary_function`

```
1 template <class _Arg, class _Result>
2 struct unary_function { // base class for unary
3     functions
4     using argument_type = _Arg;
5     using result_type = _Result;
6 };
7
8 // binary_function
9 template <class _Arg1, class _Arg2, class _Result>
10 struct binary_function { // base class for binary
11     functions
12     using first_argument_type = _Arg1;
13     using second_argument_type = _Arg2;
14     using result_type = _Result;
15 };
```

1. bind2nd: 绑定二元函数对象的第二个参数

```
1 // FUNCTION TEMPLATE bind2nd
2 template <class _Fn, class _Ty>
3 _NODISCARD binder2nd<_Fn> bind2nd(const _Fn& _Func,
4 const _Ty& _Right) {
5     typename _Fn::second_argument_type _val(_Right);
6     // 在这里相当于检查是否是一个binary_function
7     return binder2nd<_Fn>(_Func, _val);
8     // 调用binder2nd模板，生成一个一元函数对象
9 }
10 template <class _Fn> // 继承自unary_function，表明这也
11 是一个unary_function对象
12 class binder2nd : public unary_function<typename
13 _Fn::first_argument_type,
```

```

12         typename _Fn::result_type> {
13             // functor adapter _Func(left, stored)
14         public:
15             using _Base = unary_function<typename
16                 _Fn::first_argument_type, typename
17                 _Fn::result_type>;
18             using argument_type = typename
19                 _Base::argument_type;
20             using result_type = typename
21                 _Base::result_type;
22
23             binder2nd(const _Fn& _Func, const typename
24                 _Fn::second_argument_type& _Right) : op(_Func),
25                 value(_Right) {} // 构造函数
26
27             result_type operator()(const argument_type&
28                 _Left) const {
29                 return op(_Left, value); // 重载operator(),
使得是一元函数调用行为
30             }
31
32             result_type operator()(argument_type& _Left)
33             const {
34                 return op(_Left, value);
35             }
36
37         protected:
38             _Fn op;
39             typename _Fn::second_argument_type value; // the
40             right operand
41     };

```

2. bind1st: 绑定二元函数的第一个参数

要求被绑定的函数对象有 `second_argument_type`

对于一元函数 `unary_function` 的适配行为有哪些?

1. not1: 对原本的谓词调用逻辑取反

```

1 template <class _Fn>
2 class _CXX17_DEPRECATED_NEGATORS unary_negate {
3     public:

```

```

4     using argument_type = typename
5         _Fn::argument_type;
6
7     constexpr explicit unary_negate(const _Fn&
8         _Func) : _Functor(_Func) {}
9
10    constexpr bool operator()(const argument_type&
11        _Left) const {
12        return !_Functor(_Left);
13    }
14
15 private:
16     _Fn _Functor;
17 };

```

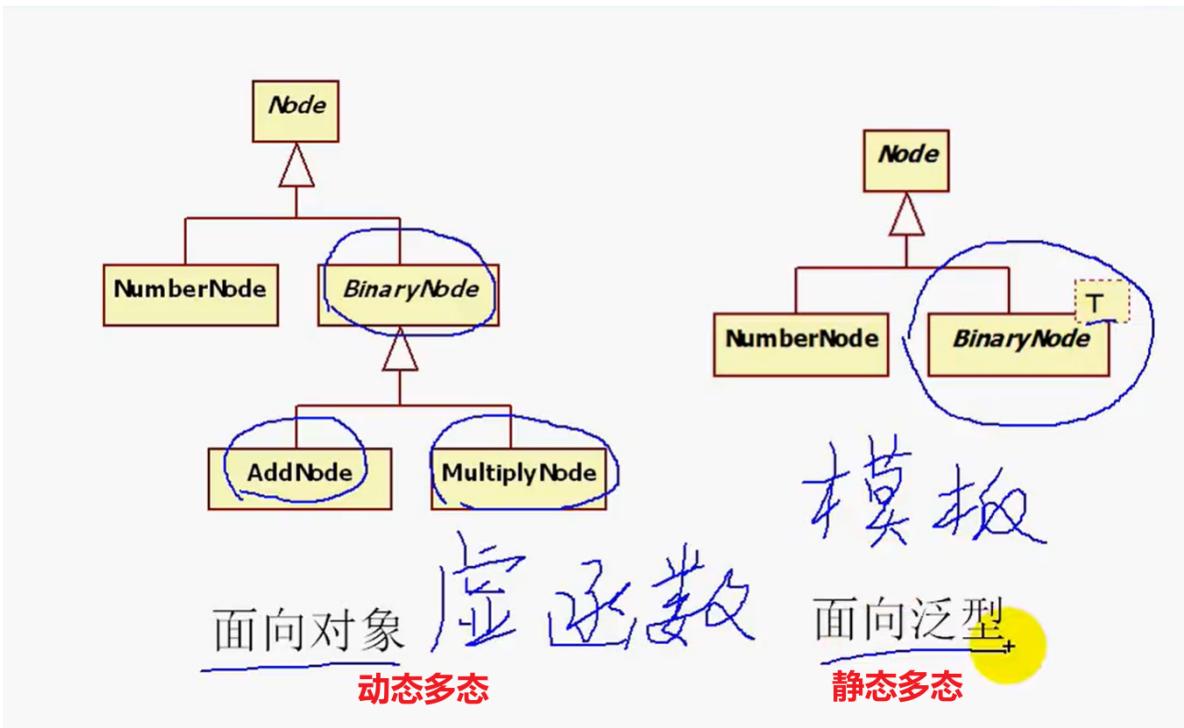
当要适配的函数，是普通函数或者成员函数的时候。应该先将他们提升为
unary_function 或者 **binary_function**

- 对于函数参数为0或1个的成员函数或者普通函数，会进行函数模板自动推导（如何实现？）
- 对于普通函数：使用ptr_fun
- 对于成员函数使用
 - mem_fun_ref（适用于成员函数）：一元函数适配器，**实现将一个不带参数的成员函数，适配成一元函数对象**
 - mem_fun（适用于成员函数指针）：也就是适配后的函数对象传入的参数是对象指针。
- 底层会调用 **const_mem_fun1_t** 类模板生成相应的对象。并且 **重载 operator()**

3. 小项目

3.1 面向对象计算器设计【未】

3.2 面向泛型版的计算器实现



3.3 银行储蓄系统

3.4 MFC框架

4. C++新特性

C++11还有其他的新特性，比如

- auto关键字类型推导

4.1 智能指针

4.2 右值引用？

4.3 function/bind (基于对象编程)

[函数指针和函数类型](#)

- **函数指针**指向的是函数而非对象。和其他指针类型一样，函数指针指向某种特定类型。
- **函数类型**由它的返回值和参数类型决定，与函数名无关。

```
1 | bool length_compare(const string &, const string &);
```

上述函数类型是: `bool (const string &, const string &);`

pf = length_compare

上述函数指针pf: `bool (*pf)(const string &, const string &);`

1. 将一个函数指针作为一个值使用时，该函数自动转换成一个指针。

```
1 | typedef bool Func(const string &, const string &) //  
Func是函数类型;  
2 | typedef bool (*FuncP)(const string &, const string &) //  
FuncP是函数指针类型;  
3 |  
4 | typedef decltype(length_compare) Func2 // Func2是函数类  
型;  
5 | typedef decltype(length_compare) *Func2P // Func2是函数指  
针类型;
```

4.3.1 作用：解决的问题

C++中**可调用对象**的虽然都有一个比较统一的操作形式，但是定义方法五花八门，这样就导致使用统一的方式**保存可调用对象或者传递可调用对象时，会十分繁琐**。

C++11中提供了**std::function**和**std::bind**统一了可调用对象的各种操作。

可调用对象：个人理解就是函数对象、普通函数、成员函数指针

- 是一个函数指针，参考 [C++ 函数指针和函数类型](#)；
- 是一个具有operator()成员函数的类的对象；（**仿函数**）
- 可被转换成函数指针的类对象；
- 一个类成员函数指针；

不同类型可能具有相同的调用形式；

```
1 | // 普通函数  
2 | int add(int a, int b){return a+b;}  
3 |  
4 | // Lambda表达式  
5 | auto mod = [](int a, int b){ return a % b;};  
6 |  
7 | // 函数对象类  
8 | struct divide{
```

```

9     int operator()(int denominator, int divisor){
10        return denominator/divisor;
11    }
12 }
13
14 // 上述三种可调用对象虽然类型不同，但是共享了一种调用形式
15 int(int ,int)

```

通过 `std::function` 将他们保存起来

```

1 std::function<int(int ,int)> a = add;
2 std::function<int(int ,int)> b = mod ;
3 std::function<int(int ,int)> c = divide();

```

4.3.2 std::function

- `std::function` 是一个可调用对象包装器，是一个类模板，可以容纳除了类成员函数指针之外的所有可调用对象，它可以用统一的方式处理函数、函数对象、函数指针，并允许保存和延迟它们的执行。
- 定义格式：`std::function<函数类型>`。
- `std::function` 可以取代函数指针的作用，因为它可以延迟函数的执行，特别适合作为回调函数使用。它比普通函数指针更加的灵活和便利。

4.3.3 std::bind

可将 `std::bind` 函数看作一个通用的函数适配器，它接受一个可调用对象，生成一个新的可调用对象来“适应”原对象的参数列表。

`std::bind` 将可调用对象与其参数一起进行绑定，绑定后的结果可以使用 `std::function` 保存。
`std::bind` 主要有以下两个作用：

- 将可调用对象和其参数绑定成一个防函数；
- 只绑定部分参数，减少可调用对象传入的参数。

`std::bind` 可以绑定的对象类型

1. 普通函数

```

1 double my_divide (double x, double y) {return x/y;}
2 auto fn_half = std::bind (my_divide,_1,2);
3 std::cout << fn_half(10) << '\n';
4 // 5

```

bind的第一个参数是函数名，普通函数做实参时，会隐式转换成函数指针。因此std::bind (my_divide,_1,2)等价于std::bind (&my_divide,_1,2);_1表示占位符，位于<functional>中，std::placeholders::_1；

2. 成员函数

```
1 struct Foo {  
2     void print_sum(int n1, int n2)  
3     {  
4         std::cout << n1+n2 << '\n';  
5     }  
6     int data = 10;  
7 };  
8 int main()  
9 {  
10    Foo foo;  
11    auto f = std::bind(&Foo::print_sum, &foo, 95,  
12    std::placeholders::_1);  
13    // 适配成了一个通用的std::function, 只接受一个传入参数  
14    f(5); // 100  
15 }
```

- bind绑定类成员函数时，第一个参数表示对象的成员函数的指针，第二个参数表示对象的地址。
- 必须显示的指定&Foo::print_sum，因为编译器不会将对象的成员函数隐式转换成函数指针，所以必须在Foo::print_sum前添加&；
- 使用对象成员函数的指针时，必须要知道该指针属于哪个对象，因此第二个参数为对象的地址 &foo；

3. 指向成员函数的指针:void (Foo::*fun)()

指向成员函数的指针的定义方法：

```
1 #include <iostream>  
2 struct Foo {  
3     int value;  
4     void f() { std::cout << "f(" << this->value <<  
5         ")\\n"; }  
6     void g() { std::cout << "g(" << this->value <<  
7         ")\\n"; }  
8 };  
9 void apply(Foo* foo1, Foo* foo2, void (Foo::*fun)()) {  
10    (foo1->*fun)(); // call fun on the object foo1
```

```

9     (foo2->*fun)(); // call fun on the object foo2
10    }
11    int main() {
12        Foo foo1{1};
13        Foo foo2{2};
14        apply(&foo1, &foo2, &Foo::f);
15        apply(&foo1, &foo2, &Foo::g);
16    }
17
18 // 成员函数指针的定义: void (Foo::*fun)(), 调用是传递的实参:
19 // &Foo::f;
20 // fun为类成员函数指针, 所以调用是要通过解引用的方式获取成员函数
21 *fun, 即(foo1->*fun)();

```

4. 绑定一个引用参数

默认情况下, bind的那些不是占位符的参数被拷贝到bind返回的可调用对象中。但是, 与lambda类似, 有时对有些绑定的参数希望以引用的方式传递, 或是要绑定参数的类型无法拷贝。

4.4 lambda表达式

5. MySQL

《数据库系统概论》

C++ 教程网

数据库基本概念

- **数据库 (DB)**
 - 按照数据结构来组织、存储数据的仓库
- **数据库管理系统 (DBMS)**
 - 数据库管理系统(Database Management System)是一套操纵和管理数据库的软件, 是用于建立、使用和维护数据库
- **数据库系统 (DBS)**
 - 数据库
 - 数据库管理系统 (及其开发工具) + MySQL
 - 应用系统
 - 数据库管理员
 - 用户

MySQL + Database

Bank system

5.1 关系数据库

一个关系对应数据库中的一张表

员工 (编号、姓名、年龄、民族、部门)

主键: 能够唯一标识一条记录, 可以有多个属性构成主键

□ 关系数据库

□ 采用关系模型作为数据组织方式。简单地说数据的逻辑结构是一张二维表, 由行和列组成。表的每一行为一个元组, 每一列为一个属性。 *唯一*

□ 关系的完整性约束

□ 实体完整性

+ □ 主键不为空

□ 参照完整性

□ 或者为空, 或者等于另一个关系的主码值

□ 用户定义的完整性

□ 用于设置某个属性的取值范围

编号	姓名	年龄	民族	部门
1	王涛	33	汉族	人事管理部
2	李梅	27	汉族	人事管理部

冗余存储的改进方式: **外键存储** (外键必须是另一张表格的主键)

6. 拓展总结

6.1 boost智能指针

智能指针是利用RAII(Resource Acquisition Is Initial: 资源获取即初始化)来管理资源。在构造函数中对资源进行初始化, 在析构函数中对资源进行释放

本质思想:

将堆对象的生存期用栈对象(智能指针)来管理, 当new一个堆对象的时候, 立刻用智能指针来接管。具体做法是在构造函数进行初始化(用一个指针指向堆对象), 在析构函数中调用delete来释放堆对象

而智能指针本身是一个**栈对象**, 它的作用域结束的时候, 自动调用析构函数。从而调用了delete释放了堆对象

<u>scoped_ptr<T></u>	当这个指针的作用域结束之后 <u>自动释放</u> ，与 <u>auto_ptr</u> 不同之处在于， <u>所有权不能转移</u> ，但是可以 <u>交换</u>
<u>shared_ptr<T></u>	内部维护一个引用计数器来判断此指针是不是需要被 <u>释放</u> 。 <u>线程安全</u>
<u>intrusive_ptr<T></u>	也维护一个 <u>轻量级</u> 的引用计数器，比 <u>shared_ptr</u> 有更 <u>好的性能</u> ，但是要求T自己提供这个计数器。
<u>weak_ptr<T></u>	弱指针，它不控制对象的生命期，但是它知道对象是否还活着。如果对象还活着，那么它可以提升(promote)为有效的 <u>shared_ptr</u> ；如果对象已经死了， <u>提升会失败</u> ，返回一个空的 <u>shared_ptr</u> 。
<u>scoped_array<T></u>	与 <u>scoped_ptr</u> 相似，用来处理数组
<u>shared_array<T></u>	与 <u>shared_ptr</u> 相似，用来处理数组

6.1.1 scoped_ptr<T>

- scoped_ptr管理的对象既不能够拷贝，也不能够转移。（底层实现：拷贝构造函数，operator=是private）

```

1 #include <boost/scoped_ptr.hpp>
2 #include <iostream>
3
4 class X
5 {
6 public:
7     X()
8     {
9         cout << "construct X" << endl;
10    }
11 ~X()
12 {
13     cout << "destruct X" << endl;
14 }
15 };
16
17 void testScoped_ptr()
18 {
19     boost::scoped_ptr<X> p(new X);
20 }
21
22 int main(void)
23 {
24     cout << "entering main" << endl;
25 }
```

```

26     testScoped_ptr();
27
28     cout << "exiting main" << endl;
29
30     return 0;
31 }

```

6.1.2 shared_ptr<T> (线程安全的)

成员函数:

- `use_count`: 原子性操作
- `reset`
- `shared_ptr<T>`可以放在`vector`当中。而`auto_ptr`不可以。因为在底层实现的时候，`shared_ptr`的接口和`vector`的`push_back`的接口一致，都是`const`引用传递
- 避免使用匿名的临时的`shared_ptr`对象

http://www.boost.org/doc/libs/1_52_0/libs/smart_ptr/shared_ptr.htm

C++ 教程网

```

int main(void)
{
    cout<<"Entering main ..."<<endl;
    boost::shared_ptr<X> p1(new X);
    cout<<p1.use_count()<<endl;
    boost::shared_ptr<X> p2 = p1;
    cout<<p2.use_count()<<endl;
    p1.reset();
    cout<<p2.use_count()<<endl;
    p2.reset();
    cout<<"Exiting main ..."<<endl;
    return 0;
}

```

屏幕录像专家 未注册

shared_ptr<T>

The diagram illustrates the state transitions of shared_ptr<T> objects p1 and p2. It shows four states:

- Initial state: p1 points to a box labeled "X reference = 1".
- After p2 = p1: p1 points to X (reference count 2), and p2 also points to X.
- After p1.reset(): p1 is NULL, and p2 still points to X (reference count 1).
- Final state: Both p1 and p2 are NULL, and the X object is marked as deleted.

```
void f(shared_ptr<int>, int);  
int g();
```

为什么不要使用临时对象测试代码。
原理可以参考effective CPP

```
void ok()  
{  
    shared_ptr<int> p(new int(2));  
    f(p, g());  
}
```



堆对象先构造，但是智能指针还没有接管

```
void bad()  
{  
    f(shared_ptr<int>(new int(2)), g());  
}
```

接着g函数调用，但是抛出异常。
就存在内存泄露的风险

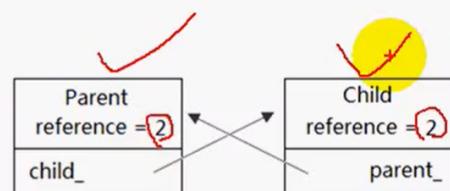
堆
✓
×
抛异常

6.1.3 循环引用

C++
教程网

循环引用

```
class Parent;  
class Child;  
typedef boost::shared_ptr<Parent> parent_ptr;  
typedef boost::shared_ptr<Child> child_ptr;  
class Child  
{  
public:  
    Child() { cout<<"Child ..." << endl; }  
    ~Child() { cout<<"~Child ..." << endl; }  
    parent_ptr parent_;  
};  
class Parent  
{  
public:  
    Parent() { cout<<"Parent ..." << endl; }  
    ~Parent() { cout<<"~Parent ..." << endl; }  
    child_ptr child_;  
};  
int main(void)  
{  
    parent_ptr parent(new Parent);  
    child_ptr child(new Child);  
    parent->child_ = child;  
    child->parent_ = parent;  
    return 0;  
}
```



www.cdcourse.com

解决办法：

- 手动打破循环引用

```
1 | parent->child_.reset();  
2 | child->parent_.reset();
```

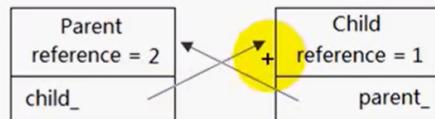
- weak_ptr<T>打破循环引用

```

class Parent;
class Child;
typedef boost::shared_ptr<Parent> parent_ptr;
typedef boost::shared_ptr<Child> child_ptr;
class Child
{
public:
    Child() { cout<<"Child ..."<

```

10



- 强引用，只要有一个引用存在，对象就不能释放
- 弱引用，并不增加对象的引用计数。但它能知道对象是否存在
 - 如果存在，提升为 shared_ptr(强引用) 成功
 - 如果不存在，提升失败
- 通过weak_ptr访问对象的成员的时候，要提升为 shared_ptr

www.cppcourse.com

6.1.4 weak_ptr

```

int main(void)
{
    boost::weak_ptr<X> p;
    {
        boost::shared_ptr<X> p2(new X);
        cout<<p2.use_count()<

```

为什么要提升为shared_ptr?
因为weak_ptr内部没有重载->

→ 提升

6.1.5 scoped_array和shared_array

1 | boost::scoped_array<x> xx(new X[3]);

6.2 单例模式和auto_ptr

6.2.1 单例模式

- 保证一个类只有一个实例，并提供一个全局访问点
- 禁止拷贝

原理实现：将构造函数设为private成员，这样就无法在外部任意地创建对象

代码实现

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Singleton{
6 public:
7     static Singleton* GetInstance()
8     {
9         if (instance_ == nullptr)
10        {
11            instance_ = new Singleton; // 调用private构造
12            函数
13        }
14    ~Singleton()
15    {
16
17    }
18    class Garbo
19    {
20 public:
21     ~Garbo()
22    {
23         if (Singleton::instance_ != nullptr)
24        {
25             delete instance_;
26        }
27    }
28 }
29 private:
30     Singleton(const Singleton& other);
```

```
31     Singleton& operator=(const Singleton& other); // 禁止拷贝
32     Singleton()
33     {}
34     static Singleton* instance_; // 引用性声明
35
36     static Garbo garbo_;
37 };
38
39 Singleton::Garbo Singleton::garbo; // 定义garbo_
40 static Singleton* instance_; // 定义性声明
41
42 int main(void)
43 {
44     Singleton* s1 = Singleton::GetInstance();
45     Singleton* s2 = Singleton::GetInstance(); // 返回的总是同一个实例
46
47     return 0;
48 }
```

- instance_是一个static成员变量，所有对象共有。
- `GetInstance()`是一个全局访问点

当前存在的问题：

- 无法调用析构函数，存在资源泄露的可能
- 还没有实现禁止拷贝功能

解决方案：

- 利用对象的确定性析构来实现 嵌套Garbo类
- 局部的静态对象（在运行期初始化）

```
class Singleton
{
public:
    static Singleton* GetInstance()
    {
        static Singleton instance;           // 局部静态对象
        return &instance;
    }

    ~Singleton()
    {
        cout<<"~Singleton ... "<<endl;
    }

private:
    Singleton(const Singleton& other);
    Singleton& operator=(const Singleton& other);
    Singleton()
    {
        cout<<"Singleton ... "<<endl;
    }
};
```

但是这不是线程安全的，因为是在函数内部定义的static局部变量

6.2.2 auto_ptr在单例模式中的应用

代码实现：

```
1 #include <iostream>
2 #include <memory>
3
4 using namespace std;
5
6 class Singleton{
7 public:
8     static Singleton* GetInstance()
9     {
10         if (!instance_.get()) // .get() 返回原生指针
11         {
12             instance_ = auto_ptr<Singleton>(new
13             Singleton);           // 调用private构造函数
14         }
15         return instance_.get(); // 和.release()不同
16     }
17     ~Singleton()
18     {
19         cout << "~Singleton()" << endl;
20     }
}
```

```

21 private:
22     Singleton(const Singleton& other);
23     Singleton& operator=(const Singleton& other); // 禁止拷贝
24     Singleton()
25     {
26         cout << "Singleton()" << endl;
27     }
28     static auto_ptr<Singleton> instance_; // 引用性声明
29
30 };
31
32 auto_ptr<Singleton> Singleton::instance_; // 定义性声明
33
34 int main(void)
35 {
36     Singleton* s1 = Singleton::GetInstance();
37     Singleton* s2 = Singleton::GetInstance(); // 返回的总是同一个实例
38
39     return 0;
40 }
```

6.2.3 muduo中单例模式实现

6.2.3.1 ThreadLocalSingleton模板类实现

```

1 template<typename T>
2 class ThreadLocalSingleton : boost::noncopyable // 默认
private继承
3 {
4     public:
5
6     static T& instance()
7     {
8         if (!t_value_)
9         {
10             t_value_ = new T();
11             deleter_.set(t_value_);
12         }
13         return *t_value_;
14     }
```

```
15     static T* pointer()
16     {
17         return t_value_;
18     }
19 }
20
21 private:
22
23     static void destructor(void* obj)
24     {
25         assert(obj == t_value_);
26         typedef char T_must_be_complete_type[sizeof(T) == 0
27 ? -1 : 1];
28         delete t_value_;
29         t_value_ = 0;
30     }
31
32 // 嵌套类
33 class Deleter // 为了能够回调函数destructor, 使得deleter
34 对象被销毁的时候, t_value也能够被释放
35 {
36     public:
37     Deleter()
38     {
39         pthread_key_create(&pkey_,
40 &ThreadLocalsingleton::destructor);
41     }
42
43     ~Deleter()
44     {
45         pthread_key_delete(pkey_); // 析构函数调用的时候, 会
46         // 删除key, 并且注册了destructor, 那么也会析构与之绑定的newObj, 即
47         // t_value_;
48     }
49
50     void set(T* newObj)
51     {
52         assert(pthread_getspecific(pkey_) == NULL);
53         pthread_setspecific(pkey_, newObj);
54     }
55 }
```

```

51     pthread_key_t pkey_;
52 }
53
54     static __thread T* t_value_; // 加了__thread表示每个线程
都拥有一份
55     static Deleter deleter_; // 销毁指针所指向的对象
56 };
57
58 // 后面是static成员变量的定义
59 template<typename T>
60 __thread T* ThreadLocalsingleton<T>::t_value_ = 0;
61
62 template<typename T>
63 typename ThreadLocalsingleton<T>::Deleter
ThreadLocalsingleton<T>::deleter_;

```

pthread_key相关函数

```

1 #include <pthread.h>
2
3 int pthread_key_create(pthread_key_t *key, void
(*destructor)(void*));
4 // 返回值: 成功时将新创建的key保存在*key, 并且返回0, 否则返回
error
5 pthread_key_delete(pthread_key_t *key);
6
7 pthread_getspecific(pthread_key_t *key);
8 pthread_setspecific(pthread_key_t *key, T* newobj);

```

- pthread_key_create: 一旦一个线程创建了一个key, 那么所有的线程都拥有这个key
- pthread_key_delete: 删除这个key, 但是不删除这个数据, 要删除数据需要在create的时候注册一个回调函数。 (数据是堆上数据)
- pthread_getspecific: 通过key得到线程特有数据
- pthread_setspecific: 指定特定的数据, 线程私有的

6.3 Noncopyable实现

```

1 #include <iostream>
2
3 class Noncopyable

```

```

4 {
5     public:
6         noncopyable(const noncopyable&) = delete; // =delete禁止拷贝构造
7         void operator=(const noncopyable&) = delete;
8
9     protected:
10    noncopyable() = default;
11    ~noncopyable() = default;
12 };
13
14 class Parent : private Noncopyable // 声明为private继承,
15     是因为只需要继承默认实现
16 {
17     public:
18         // 符合规范的拷贝构造函数定义
19         Parent(const Parent& other) : Noncopyable(other) // 见effective c++
20         {
21     }
22 };
23
24 int main(void)
25 {
26     Parent p1;
27     // Parent p2(p1); Error: 无法访问private成员
28     // 要调用parent的拷贝构造函数, parent拷贝构造函数又要调用
29     // Noncopyable的拷贝构造函数

```

6.4 用宏实现sizeof的功能

1. 计算一个变量的大小 `sizeof_v`: 通过指针偏移得到
2. 计算一个类型的大小 `sizeof_t`

```

1 #include <iostream>
2
3 using namespace std;
4 // 两个指针相见, 得到的是相隔几个元素
5 // &x + 1: x的地址向后偏移一个元素地址

```

```

6 #define sizeof_v(x) ((char*)(&x + 1) - (char*)&x)
7 #define sizeof_t(t) static_cast<int>((t*)0 + 1)
8
9 // 对齐宏，要求b是2的整数次方
10 #define ALIGN(v, b) ((v+b-1) & ~(b-1))
11
12 class Empty
13 {
14 };
15 int main(void)
16 {
17     Empty e;
18     int n;
19
20     cout << sizeof_v(e) << endl; // 1
21     cout << sizeof_t(Empty) << endl; // 1
22     cout << sizeof_t(n) << endl; // 4
23     cout << sizeof_v(int) << endl; // 4
24
25     cout << ALIGN(3, 16) << endl; // 将v对齐到b的整数倍（上
26 取整）
27 }

```

3. 实现某个数对齐到整数（2的整数次方）倍

```

#define ALIGN(v, b) ((v+b-1) & ~(b-1))

          0011
          1111
          10010
          0000
10000 = 16

原理、思想
某个数要对齐到16的整数倍      0000      32的整数倍00000
1000

向上对齐 3 + 15
超出的部分我们要把它抹除掉（低4位都置0）

0
ALIGN(0, 16) = 0

```

对齐宏的使用：内存池中，内存块大小是规则的。否则会产生空隙，**内存碎片**

6.5 编程技巧

1. 通过typedef实现编译期间发现错误

```
1 | typedef char T_must_be_complete_type[sizeof(T) == 0  
? -1 : 1];
```

2. 实现功能的时候，先编写测试代码；见 **String** 类实现

6.5.1 工厂模式

```
// 简单工厂模式  
class ShapeFactory  
{  
public:  
    static Shape* CreateShape(const string& name)  
    {  
        Shape* ps = 0;  
        if (name == "Circle")  
        {  
            ps = new Circle;  
        }  
        else if (name == "Square")  
        {  
            ps = new Square;  
        }  
        else if (name == "Rectangle")  
        {  
            ps = new Rectangle;  
        }  
        return ps;  
    }  
};
```

```
int main(void)
{
    //Shape s;          //Error, 不能实例化抽象类
    vector<Shape*> v;
    //Shape* ps;
    //ps = new Circle;
    //v.push_back(ps);
    //ps = new Square;
    //v.push_back(ps);
    //ps = new Rectangle;
    //v.push_back(ps);

    Shape* ps;
    ps = ShapeFactory::CreateShape("Circle");
    v.push_back(ps);
    ps = ShapeFactory::CreateShape("Square");
    v.push_back(ps);
    ps = ShapeFactory::CreateShape("Rectangle");
    v.push_back(ps);

    DrawAllShapes(v);
    DeleteAllShapes(v);

    return 0;
}
```

6.5.2 常量是否可以更改?

将变量声明为常量，目的在于告诉使用者，其值不应该被修改。而不是不能够被修改

```
1 const int n = 100;
2
3 int *pn = &n;
4
5 *pn = 200;
```

6.5.3 使用PIMPL

不使用PIMPL

- 引入更多的头文件，降低编译速度
- 提高模块的耦合度
 - 编译期
 - 运行期
- 降低了接口的稳定程度
 - 对于库的使用，方法不能改变
 - 对于库的编译，动态库的变更，客户程序不用重新编译 **y的实现跟x耦合**

```
// file y.h
#include "x.h"
class Y{
    void Fun();
    X x;
};
// file y.cpp
#include "y.h"
void Y::Fun { return x_.Fun(); }

// file main.cpp
#include "y.h"
int main(void)
{
    Y y;
    y.Fun();
}
```

依赖于x.h的展开
依赖于x.h的展开

y.h不依赖于x.h,只需要一个前向声明

- PIMPL (private implementation 或 pointer to implementation) 也称为 handle/body idiom
- PIML背后的思想是把客户与所有关于类的私有部分的知识隔离开。避免其它类知道其内部结构
- 降低编译依赖、提高重编译速度
- 接口和实现分离
- 降低模块的耦合度
 - 编译期
 - 运行期 **因为是ptrX, 所以还支持多态**
- 提高了接口的稳定程度
 - 对于库的使用，方法不能改变
 - 对于库的编译，动态库的变更，客户程序不用重新编译

```
// file y.h
class X;
class Y{
    Y();
    ~Y();
    void Fun();
    X* px_;
};
// file y.cpp
#include "x.h"
Y::Y() : px_( new X() {} )
Y::~Y() { delete px_; px_ = 0; }
void Y::Fun { return x_->Fun(); }
// file main.cpp
#include "y.h"
int main(void)
{
    Y y;
    y.Fun();
}
```

客户

6.6 知识点辨析

6.6.1 重载 (overload) 、重写 (重定义overwrite) 、覆盖 (override)

- 重载发生在同一个作用域
- 重写发生在基类和派生类之间
- 覆盖：虚函数之间

教程网 **overload、overwrite、override**

- 成员函数被重载的特征：
 - (1) 相同的范围（在同一个类中）；
 - (2) 函数名字相同；
 - (3) 参数不同；
 - (4) virtual关键字可有可无。
- 覆盖是指派生类函数覆盖基类函数，特征是：
 - (1) 不同的范围（分别位于派生类与基类）；
 - (2) 函数名字相同；
 - (3) 参数相同；
 - (4) 基类函数必须有virtual关键字。
- 重定义（派生类与基类）
 - (1) 不同的范围（分别位于派生类与基类）；
 - (2) 函数名与参数都相同，无virtual关键字
 - (3) 函数名相同，参数不同，virtual可有可无

6.6.2 组合和继承

- 继承相当于将基类作为一个成员类对象使用
- 继承是is-a
- 组合是has-a

- 无论是继承与组合本质上都是把子对象放在新类型中，两者都是使用构造函数的初始化列表去构造这些子对象。
- 组合通常是在希望新类内部具有已存在的类的功能时使用，而不是希望已存在类作为它的接口。
组合通过嵌入一个对象以实现新类的功能，而新类用户看到的是新定义的接口，而不是来自老类的接口。**(has-a)**
- 如果希望新类与已存在的类有相同的接口（在这基础上可以增加自己的成员）。这时候需要用继承，也称为子类型化。**(is-a)**

6.6.3 只能在构造函数的参数列表初始化的情况

1. **const成员变量**：只能在初始化列表中初始化。（在构造函数体中就不是初始化了，其实算是赋值）
2. **引用成员**
3. 当**基类没有默认构造函数**的时候，需要在派生类的构造函数初始化列表中调用
 - 派生类的构造函数需要给基类的构造函数传递对应的参数
4. 类成员对象的构造，由该类的构造函数负责。
 - 基类对象有一个**对象成员，并且该对象所属的类没有默认构造函数**。那么该基类应该负责该对象成员的构造和析构工作。

6.6.4 构造函数的调用顺序

1. **派生类对象的构造次序**
 - 基类构造函数
 - 派生类对象成员的构造函数
 - 派生类自身的构造函数
2. **如果基类也有一个对象成员，则该成员的构造函数还要早于基类的构造函数**
 - 基类成员对象的构造函数
 - 基类的构造函数
 - 派生类自身的构造函数
3. 如果类中有多个对象成员，那么这些对象成员的构造函数按照声明的顺序调用

6.6.5 泛型程序设计

1. 模板为泛型程序设计奠定了基础
2. STL是一套C++标准模板库，体现了泛型程序设计思想。（STL是泛型程序设计思想比较成功的一套产品）

6.7 拓展阅读

- C++primer (正在)
- Effective C++ (可以再看)
- C++编码规范
- 敏捷软件开发-原则、模式与实践 (关于面向对象的语言学习)
- 代码大全，第二版