

技术总结文档

技术总结文档

概览

C++语法总结

std::string 用法

1. 字符串赋值
2. 字符串拼接
3. 字符串拷贝
4. char*/char[]与std::string之间转换
5. 数字转std::string
6. std::string转数字

std::array用法

左值引用、右值引用与移动语义

1. 左值、右值概念介绍
2. 左值引用与右值引用概念介绍
3. 左值、右值的转换
4. 左值、右值产生的重载问题
 - 4.1 左值(Lvalue)和右值(Rvalue)的重载规则:
 - 4.2 左值、右值的重载应用于class的构造问题
5. 移动语义的作用
6. 移动构造函数和移动赋值运算符

引用折叠

关键字auto、decltype介绍

1. 关键字auto
2. 关键字decltype(又名复读机)
3. 尾置返回类型 (trailing return type)

枚举类型

1. 简介
2. 指定enum的类型
3. 枚举类型引起的重载问题
4. 枚举类型与int类型的隐式转换

强制类型转换

1. static_cast< type >(expression)
2. const_cast< type >(expression)
3. reinterpret_cast< type >(expression)
4. dynamic_cast< type >(expression)
5. C++Style强制类型转换与CStyle强制类型转换对比

constexpr关键字

enable_if 关键字

STL算法速览

1. 非更易型算法 (Nonmodifying algorithms)
2. 更易型算法 (Modifying algorithms)
3. 移除型算法 (Removing algorithms)
4. 变序型算法 (Mutating algorithms)
5. 排序算法 (Sorting algorithms)
6. 已排序区间算法 (Sorted-range algorithms)
7. 数值算法 (Numeric algorithms)

std::sort

std::stable_sort

std::partial_sort

std::nth_element

STL排序算法比较

std::lock_guard用法

1. 用于指定锁定策略的标签类型

变参宏介绍

宏定义中的#运算符

函数变参介绍

成员函数的引用限定

虚函数介绍

基类与派生类之间的转换规则

1. 从派生类向基类的(隐式)类型转换只对指针和引用类型有效
2. 基类向派生类不存在隐式类型转换
3. 派生类向基类的类型转换可能会由于访问受限而变得不可行

异常处理

1. 抛出异常
 - 1.1 栈展开
 - 1.2 栈展开过程中对象被自动销毁
 - 1.3 析构函数与异常
 - 1.4 异常对象
2. 捕获异常
 - 2.1 查找匹配的处理代码
 - 2.2 重新抛出
 - 2.3 捕获所有异常的处理代码
3. 函数try语句块与构造函数
4. noexcept 异常说明
 - 4.1 违反异常说明
 - 4.2 noexcept运算符
 - 4.3 异常说明与指针、虚函数和拷贝控制

typeid关键字

std::tuple介绍

std::chrono介绍

1. std::chrono::duration_cast

std::underlying_type介绍

格式化字符串示例

预定义宏

多线程编程总结

函数的可重入性判断(线程安全性)

设计模式介绍

编程名词与概念解释

容器类别

1. 序列式容器 (Sequence container)
2. 关联式容器(Associative container)
3. 无序容器 (Unordered (associative) container)

各种容器使用时机

严格弱序 (strict weak ordering)

迭代器种类

经验总结与代码示例

如何求得一个数组的长度

数据库插入时重复主键问题

1. 更新该条数据
2. 不插入数据

3. 删除原有数据后插入新的数据

使用empty()来代替size()检查容器大小是否为0

尽量使用区间操作成员函数代替它们的单元素操作成员函数

使用reserve来避免不必要的重新分配

避免使用vector< bool >

使用using代替typedef

C++中struct和class的区别

虚析构函数问题

获取未知结构体成员信息

MySQL列出不同的值

对称数组实现itoa

产生随机数

读/写csv文件

SQL语句中DROP、TRUNCATE和DELETE的用法

概览

本文档作技术总结交流用。如果你发现文中的内容错误或拼写错误可以随时联系作者予以更正，欢迎各位同行斧正。

- **C++语法总结**：主要介绍C++当中的语法，关键字用法介绍。
- **多线程编程总结**：主要介绍多线程环境下需要注意的东西，以及线程安全的代码介绍。
- **设计模式介绍**：21种设计模式的介绍。（现在只写了两个，后面慢慢补充）
- **编程名词与概念解释**：主要用来介绍编程思想(如RAII、SFINAE)与关于C++编程中一些概念性的解释(如容器类别、迭代器类别等)。
- **经验总结与代码示例**：主要写了一些编程中容易出错的地方，以及一些代码的示例。

C++语法总结

版本号	作者	修改摘要	时间
V1.0	李建聪	添加 std::string 用法小节	2019-08-01 08:04:40
V1.1	李建聪	添加 std::array 用法小节	2019-08-01 10:53:32
V1.2	李建聪	添加 左值引用、右值引用与移动语义 小节	2019-08-01 10:53:37
V1.3	李建聪	添加 引用折叠 小节	2019-08-01 10:53:46
V1.4	李建聪	添加 关键字auto、decltype 介绍小节	2019-08-01 10:53:53
V1.5	李建聪	对 左值引用、右值引用与移动语义 小节增加内容	2019-08-09 14:49:49
V1.6	李建聪	对 std::array 用法小节增添 .at() 成员函数的用法示例	2019-08-14 10:28:15
V1.7	李建聪	增加小节 枚举类型	2019-08-14 15:13:18
V1.8	李建聪	小节 枚举类型 添加内容： 枚举类型与int类型的隐式转换	2019-09-19 11:43:47
V1.9	李建聪	添加 强制类型转换 小节	2019-09-24 13:10:13
V2.0	李建聪	添加 constexpr关键字 小节	2019-09-24 13:59:08
V2.1	李建聪	添加 STL算法速览 小节	2019-10-08 09:11:36
V2.2	李建聪	添加 std::sort 小节	2019-10-08 09:14:17
V2.3	李建聪	添加 std::stable_sort 小节	2019-10-08 09:14:17
V2.4	李建聪	添加 std::partial_sort 小节	2019-10-08 09:14:18
V2.5	李建聪	添加 std::nth_element 小节	2019-10-08 09:14:19
V2.6	李建聪	添加 STL排序算法比较 小节	2019-10-08 09:14:19
V2.7	李建聪	std::string 用法小节添加内容 数字转string、string转数字	2019-10-08 09:25:42
V2.9	李建聪	添加 std::lock_guard 用法小节	2019-10-10 08:24:30
V3.0	李建聪	添加 变参宏 介绍小节	2019-10-15 16:31:37
V3.1	李建聪	添加 宏定义中的#运算符 小节	2019-10-15 16:31:38
V3.2	李建聪	添加 函数变参 介绍小节	2019-10-15 16:31:39
V3.3	李建聪	添加 成员函数的引用限定 小节	2019-10-15 16:55:23
V3.4	李建聪	添加 虚函数 介绍小节	2019-10-31 11:08:56
V3.5	李建聪	添加 基类与派生类之间的转换规则 小节	2019-10-31 15:04:36
V3.6	李建聪	添加 异常处理 小节	2019-11-01 16:39:04
V3.7	李建聪	添加 typeid关键字 小节	2019-11-05 13:37:31
V3.8	李建聪	添加 std::tuple 介绍小节	2019-11-05 13:38:05

版本号	作者	修改摘要	时间
V3.9	李建聪	添加 std::chrono 介绍小节	2019-11-05 13:38:34
V4.0	李建聪	添加 std::underlying_type 介绍小节	2019-11-12 15:35:55
V4.1	李建聪	添加 格式化字符串示例 小节	2019-11-12 15:36:32
V4.2	李建聪	添加 预定义宏 小节	2019-11-15 15:01:37

std::string 用法

1. 字符串赋值

```
#include <string>

/*
    char acSQL[1024] = {0};

    gos_snprintf(acSQL, sizeof(acSQL), "SELECT * FROM dialhistory");
*/

/// 改用string后//
std::string strSQL("SELECT * FROM dialhistory");
或
std::string strSQL = "SELECT * FROM dialhistory";
或
std::string strSQL;
strSQL = "SELECT * FROM dialhistory";
```

2. 字符串拼接

```
#include <string>

/*
    char acStr[50] = "Hello ";
    char acStr2[50] = "world!";

    strcat(acStr, acStr2);
    std::cout << str << std::endl;
    输出: //
    Hello World!
*/

/// 改用string后//
std::string strTest("Hello ");
std::string strTest2("world!");

strTest = strTest+strTest2;
或
strTest += strTest2;
```

3. 字符串拷贝

```
#include <string>
/*
    char acFileName[512] = "DialNumber.txt";
    char acFileNameCopy[512] = {0};

    strcpy(acFileName, acFileNameCopy);
*/

/// 改用string后
std::string strFileName("SELECT * FROM dialhistory");

std::string strFileNameCopy(strFileName);
或
std::string strFileNameCopy = strFileName;
或
std::string strFileNameCopy;
strFileNameCopy = strFileName;
```

4. char*/char[]与std::string之间转换

```
#include <string>

/// char* 转换为std::string
char *szStr = "Hello world!";
std::string strTest(szStr);
/// char[]转换为std::string
char acStr[1024] = "Hello world!";
std::string strTest1(acStr);

/// std::string转换为char*/char[]
std::string strTest2("Hello world!");
char acStr[1024] = {0};
gos_snprintf(acStr, strTest2.c_str());

/// 特别注意!!!
/// 占位符的%s是C语言的char*/char[], std::string类型想要使用%s需要调用c_str()函数
std::string strTest3("Hello world!");
///printf("%s", strTest3); // 错误! 编译能过, 运行会崩溃!
printf("%s", strTest3.c_str()); // 正确
```

5. 数字转std::string

```
std::string strDouble = std::to_string(1.2); // == "1.2"
std::string strInt = std::to_string(123); // == "123"
```

6. std::string转数字

std::string转换为数字的函数有**std::stoi** (int转string)、**std::stol** (long转string)、**std::stoll** (long long转string)、**std::stoul** (unsigned long转string)、**std::stoull** (unsigned long long转string)、**std::stof** (float转string)、**std::stod** (double转string)、**std::stold** (long double转string)。

```
#include <iostream>
#include <string>

int main()
{
    std::string str1 = "45";
    std::string str2 = "3.14159";
    std::string str3 = "31337 with words";
    std::string str4 = "words and 2";

    int myint1 = std::stoi(str1);
    int myint2 = std::stoi(str2);
    int myint3 = std::stoi(str3);
    // 错误: 'std::invalid_argument'
    // int myint4 = std::stoi(str4);

    std::cout << "std::stoi(\"" << str1 << "\") is " << myint1 << '\n';
    std::cout << "std::stoi(\"" << str2 << "\") is " << myint2 << '\n';
    std::cout << "std::stoi(\"" << str3 << "\") is " << myint3 << '\n';
    //std::cout << "std::stoi(\"" << str4 << "\") is " << myint4 << '\n';
}
```

输出:

```
std::stoi("45") is 45
std::stoi("3.14159") is 3
std::stoi("31337 with words") is 31337
```

std::array用法

```
#include <iostream>
#include <string>
#include <array>
#include <cassert>

int main()
{
    /// int a1[3]; a[0] = 1; a[1] = 2; a[2] = 3;
    std::array<int, 3> a1{ 1, 2, 3 };
    assert(a1.at(0) == 1);
    assert(a1.at(1) == 2);
    assert(a1.at(2) == 3);

    /// int a2[3]; a[0] = 5; a[1] = 4; a[2] = 6;
    std::array<int, 3> a2 = { 5, 4, 6 };
    assert(a2.at(0) == 5);
    assert(a2.at(1) == 4);
}
```

```

assert(a2.at(2) == 6);

/// 原生数组不支持以下操作
std::array<std::string, 2> a3 = { std::string("a"), "b" };

// 支持容器操作
std::sort(a2.begin(), a2.end());

// 支持范围循环(遍历数组) (Range-based for loops)
for (const auto& temp : a2)
{
    std::cout << temp << std::endl;
}
/*
    输出为:
    4
    5
    6
*/
}

```

左值引用、右值引用与移动语义

1. 左值、右值概念介绍

左值（赋值操作符“=”的左侧，通常是一个变量）与**右值**（赋值操作符“=”的右侧，通常是一个常数、表达式、函数调用）。

左值可被绑定到**非const引用**或**const引用**(左值可以被修改也可以变成不可修改)。

右值只能被绑定到**const引用**(右值不可被修改)。

```

include <iostream>

void incr(int& a) { ++a; }

int main()
{
    int i = 0;
    incr(i);    // 正确i变为1
    /**
        incr(0);
        错误: 0不是一个左值
        0作为常量是右值, 不可被修改。所以不能绑定到左值引用上。
    */
}

```

2. 左值引用与右值引用概念介绍

通常我们使用&来定义左值引用，使用&&定义右值引用

```
int y = 8;
int& x = y;      // x为y的左值引用，类型为int&。x == y == 8
int&& i = 0;     // i为常量`0`的右值引用
```

3. 左值、右值的转换

```
#include <iostream>

struct X { int i; };

int main()
{
    /// 左值转换为右值
    X a{ 99 };
    X&& b = static_cast<X&&>(a);    // 使用强制类型转换生成一个a的右值引用的临时变量赋给b
    b.i++;                          // a.i == b.i == 100
    X&& c = std::move(a);           // 使用std::move生成一个a的右值引用变量的临时变量赋给c
    /// 在使用过std::move(a)后对象a的值是未定义的（可能被析构，可能不变），
    /// 所以禁止在使用过std::move后再次使用a。
    c.i++;                          // b.i == c.i == 101
    system("pause");
}
```

4. 左值、右值产生的重载问题

4.1 左值(Lvalue)和右值(Rvalue)的重载规则:

1. 如果你只实现了: **void foo(X&);** 而没有实现void foo(X&&), foo()函数**只能被左值入参调用**, 不能被右值入参调用。
2. 如果你实现了: **void foo(const X&);** 而没有实现void foo(X&&), foo()函数**可以被左值入参调用**, 也可以被右值入参调用。
3. 如果你实现了: **void foo(X&); void foo(X&&);** 或 **void foo(X&); void foo(const X&);** 则可以区分左值、右值的**重载**, 即传入左值入参调用void foo(X&), 传入右值入参调用void foo(X&&);或void foo(const X&);
4. 如果你实现了:
void foo(X&&); 但既没有实现void foo(X&);也没有实现void foo(const X&);, 那么该函数**只能被右值入参调用**, 使用左值入参将会引起编译错误。

```

struct X {};
//左值版本
void f(const X& param1){...}
//右值版本
void f(X&& param2){...}

X a;
//调用左值版本
f(a);
//调用右值版本 (使用了struct X的默认构造函数, 构造了一个没有名字的struct X(右值)作为函数f的入参)
f(X());
//调用右值版本
f(std::move(a));

```

4.2 左值、右值的重载应用于class的构造问题

如果**class XInstance**未提供move语义(没有提供移动构造函数和移动赋值符), 只提供惯常的copy构造函数和拷贝赋值符(copy assignment操作符), 右值引用(rvalue reference)可以调用他们。因此, `std::move(XInstance)`意味着"调用move语义(若有提供的话), 否则调用copy语义(即拷贝一次)".

```

/// 你不需要也不应该返回std::move()返回值。
X foo()
{
    X xinstance;
    ...
    /// 不用使用return std::move(xinstance);
    return xinstance;
}

```

上面示例代码编译器会执行以下判断:

1. 如果X由一个可用的拷贝构造函数或移动构造函数, 编译器可以选择略去其中的copy版本, 这就是返回值优化
2. 否则, 如果X有一个move构造函数, X就执行move构造。
3. 否则, 如果X有一个copy构造函数, X就被从拷贝构造。
4. 否则, 报出一个编译器错误。

也请注意, 如果返回的是一个函数内定义的非static(local non-static)对象, 那么返回其右值引用rvalue reference是不对的。

```

X&& foo()
{
    X xInstance;
    ...
    return x; // 错误: 返回一个不存在对象的引用。
}

```

5. 移动语义的作用

右值引用的概念是为了引入移动语义, 来抵消拷贝操作的时间。

/// 通常我们要交换两个元素的值会写成以下函数

```
template<class T>
void swap(T& a, T& b)
{
    T tmp = a;    // 拷贝一次a
    a = b;        // 拷贝一次b
    b = tmp;      // 再拷贝一次a
}
```

一次交换操作需要拷贝三次，如果模板对象T是一个很大对象，比如一个几K的std::string那该函数的速度会慢到不可接受。所以我们引入移动语义。

```
template <class T>
void swap(T& a, T& b)
{
    T tmp = std::move(a);    // 变量a现在失效（内部数据被move到tmp中了）
    a = std::move(b);        // 变量b现在失效（内部数据被move到a中了，变量a现在“满血复活”了）
    b = std::move(tmp);      // 变量tmp现在失效（内部数据被move到b中了，变量b现在“满血复活”了）
}
```

std::move的作用其实就是给变量换了个名字，**被移动对象被移动后会失效或者析构(不可使用)**。

6. 移动构造函数和移动赋值运算符

```
struct X
{
    X();                                //默认构造函数
    X(const X& that);                   //拷贝构造函数
    X(X&& that);                        //移动构造函数
    X& operator=(const X& that);        //拷贝赋值运算符
    X& operator=(X&& that);            //移动赋值运算符
    /// 注意：上面五个函数如果不手动定义，编译器会自动生成。
};

X a;                                    //调用默认构造函数

X b = a;                                //调用拷贝构造函数
X c = std::move(b);                     //移动构造函数（b被析构，不可使用）
b = a;                                  //拷贝赋值运算符（使用a来调用拷贝构造符号，构造一个x对象赋给b）
c = std::move(b);                       //移动赋值运算符（b再次被析构，不可使用）
```

引用折叠

1. T& & 变为 T& // 左值引用的左值引用是**左值引用**
2. T& && 变为 T& // 左值引用的右值引用是**左值引用**
3. T&& & 变为 T& // 右值引用的左值引用是**左值引用**
4. T&& && 变为 T&& // 右值引用的右值引用是**右值引用**

规则为：如果任一引用为左值引用，则结果为左值引用。否则（即两个都是右值引用），结果为右值引用。

```
int x = 0;           // x 是类型为int的左值
auto&& aLvalue = x; // auto&&被一个左值初始化, &&被视为引用的引用折叠为引用。aLvalue。
auto&& aRvalue = 0; // auto&&被一个右值(常量`0`)初始化, &&被直接视为右值引用。所以aRvalue是右值。
```

关键字auto、decltype介绍

1. 关键字auto

```
auto dNum = 3.14;           // double
auto iNum = 1;              // int
auto& c = iNum;             // int&
auto d = { 0 };            // std::initializer_list<int>
const auto h = 1;          // const int
auto intptr = new auto(123); // int*

auto&& e = 1;                // int&&(常量`1` (右值) 初始化)
auto&& f = iNum;            // int&(引用折叠)

auto i = 1, j = 2, k = 3;   // int, int, int
auto l = 1, m = true, n = 1.61; // 错误, 不同类型的变量不能同时初始化。

auto o;                    // 错误 auto定义的变量必须要初始化

/// 下面两个循环都可以达到遍历容器的效果
std::vector<int> viTemp = {1,2,3,4,5};
for (const auto& Element : viTemp)
{
    std::cout << Element << std::endl;
}

for (const auto cit = viTemp.cbegin(); cit != viTemp.cend(); ++cit)
{
    std::cout << *cit << std::endl;
}

/** 返回值自动推导 */
template <typename T>
auto f(T& t)
{
    return t;
}

f(4)           // 返回值为int
f(0.01)        // 返回值为double
f("Hello world!") // 返回值为std::string
```

2. 关键字decltype(又名复读机)

decltype(T)会复述出括号内T的数据类型。

```
int a = 1;
decltype(a) b = a;      // `decltype(a)` 类型为 `int`

const int& c = a;
decltype(c) d = a;      // `decltype(c)` 类型为 `const int&`
decltype(123) e = 123;  // `decltype(123)` 类型为 `int`

int&& f = 1;
decltype(f) g = 1;      // `decltype(f)` 类型为 `int&&`
decltype((a)) h = g;    // `decltype((a))` 类型为 `int&` // (这句话我(李建聪)不太理解)
```

3. 尾置返回类型 (trailing return type)

```
template <typename X, typename Y>
auto add(X x, Y y) -> decltype(x + y)
{
    return x + y;
}
add(1, 2);      // 返回值类型为int
add(1, 2.0);    // 返回值类型为double(int和double相加的时候int被隐式提升为double)
add(1.5, 1.5);  // 返回值类型为double
```

枚举类型

1. 简介

枚举类型分为：限定作用域枚举类型(enum)和不限定作用域枚举类型(enum class)。

```
enum color { red, yellow, green };      // 不限定作用域枚举类型
enum ErrorColor { red, yellow, green }; // 错误，重复定义了枚举成员
enum class CorrectColor {red, yellow, green}; // 正确：因为这是限定作用域枚举类型

color temp = 0;      // 错误 (0不属于枚举类型color)

color temp = red;    // 正确
color temp2 = color::yellow; // 正确

CorrectColor temp3 = red;      // 错误
CorrectColor temp3 = CorrectColor::red; // 正确 (限定作用域的枚举类型)

int i = color::red;      // 正确：不限定作用域的枚举类型可隐式转换为int类型
int j = CorrectColor::red; // 错误：限定作用域的枚举类型不会进行隐式转换
```

2. 指定enum的类型

尽管每个enum都定义了唯一的类型，但实际上enum是由**某种整数类型**表示的。同时我们也可以显式的指定enum中使用的类型。

```
enum EVENT_DIAL_MSG_E : unsigned long long
{
    FIRST_MSG = 0,
    SECOND_MSG = 18446744073709551615ULL
};
```

1. 如果我们没有指定enum的潜在类型，则默认情况下**限定作用域的**enum成员类型是**int**;不限定作用于的enum成员类型不存在默认类型，我们只知道成员的潜在类型足够大，肯定能够容纳枚举值。
2. 如果我们指定了枚举成员的潜在类型（包括对限定作用域的enum的隐式指定），则一旦某个枚举成员的值超出了该类型所能容纳的范围，将引发程序错误。

3. 枚举类型引起的重载问题

```
#include <iostream>

enum Tokens { INLINE = 128, VIRTUAL = 129};
void ff(Tokens x){}
void ff(int y){}

void newf(unsigned char foo){}
void newf(int bar){}

int main()
{
    Tokens curTok = INLINE;
    ff(128);    // 精确匹配ff(int y)
    ff(INLINE); // 精确匹配ff(Tokens x)
    ff(curTok); // 精确匹配ff(Tokens x)

    unsigned char uc = VIRTUAL;
    newf(VIRTUAL); // 调用newf(int bar): VIRTUAL隐式转换为int了
    newf(uc);      // 调用newf(unsigned char foo)
}
```

4. 枚举类型与int类型的隐式转换

enum类型可以隐式转换为int类型，反之则编译错误。需要进行显示类型转换。

```
#include <iostream>

enum TEST_E : int
{
    TEST_FIRST = 1,
    TEST_SECONED = 2,
};

enum class CLASS_TEST_E : int
{
    CLASS_TEST_FIRST = 1,
    CLASS_TEST_SECONED = 2,
};
```

```

int main()
{
    /// 不具名enum类型(不带class的)可以隐式转换为int类型//
    int i = TEST_FIRST;

    /// int类型不能隐式转换为不具名enum类型//
    /// TEST_E Error = i;

    /// 正确用法//
    TEST_E Correct = static_cast<TEST_E>(i);

    /// 具名enum类型(带class的)不可以隐式转换为int类型//
    /// int i = CLASS_TEST_E::CLASS_TEST_FIRST;

    /// int类型不能隐式转换为具名enum类型//
    /// CLASS_TEST_E Error = i;

    /// 正确用法//
    int x = static_cast<int>(CLASS_TEST_E::CLASS_TEST_FIRST);
    CLASS_TEST_E Error = static_cast<CLASS_TEST_E>(i);

    return 0;
}

```

强制类型转换

C语言中只有一种类型转换的语法，形式为小括号里放入要转换为的类型，后面紧跟需要转换的变量，如(int)Var;

而C++提供了多个专门用来进行类型转换的关键字，**static_cast**(expression)、**const_cast**(expression)、**reinterpret_cast**(expression)和**dynamic_cast**(expression)。他们因为函数重载而设计出来，所以在C++编程中需要使用这三种类型转换的形式。接下来我将分别介绍这三种关键字的用途和他们优于C语言的类型强制转换的地方。

1. static_cast< type >(expression)

我们可以认为static_cast(expression)等价于C语言中的强制类型转换。

```

#include <vector>
#include <iostream>

struct B
{
    int m = 0;
    void hello() const
    {
        std::cout << "Hello world, this is B!\n";
    }
};

struct D : B
{
    void hello() const
    {

```

```

        std::cout << "Hello world, this is D!\n";
    }
};

enum class E { ONE = 1, TWO, THREE};
enum EU { ONE = 1, TWO, THREE, FOUR};

int main ()
{
    // 1: 初始转换//
    int n = static_cast<int>(3.14);
    std::cout << "n = " << n << '\n';           // 输出: n = 3
    std::vector<int> v = static_cast<std::vector<int>>(10);
    std::cout << "v.size() = " << v.size() << '\n';       // 输出: v.size() = 10

    // 2: 静态向下转型//
    D d;
    // 通过隐式转换向上转型(派生类转基类引用)//
    B& br = d;
    br.hello();           // 输出: Hello world, this is B!
    // 向下转型 (基类引用强转派生类引用) //
    D& another_d = static_cast<D&>(br);
    another_d.hello();     // 输出: Hello world, this is D!

    // 3: 左值到右值//
    std::vector<int> v2 = static_cast<std::vector<int>&&>(v);
    std::cout << "after move, v.size() = " << v.size() << '\n';
    // 输出: after move, v.size() = 0

    // 4: 弃值表达式 (作用为: 限制编译器警告没有使用的变量) //
    static_cast<void>(v2.size());

    // 5. 隐式转换的逆//
    void* nv = &n;
    int* ni = static_cast<int*>(nv);

    // 6. 数组到指针后向上转型//
    D a[10];
    B* dp = static_cast<B*>(a);

    // 7. 有作用域枚举到int或float
    E e = E::ONE;
    int one = static_cast<int>(e);
    std::cout << one << '\n';           // 输出: 1

    // 8. int到枚举值, 枚举到另一个枚举//
    E e2 = static_cast<E>(one);
    EU eu = static_cast<EU>(e2);

    // 9. 指向成员指针向上转型//
    int D::* pm = &D::m;
    std::cout << br.*static_cast<int B::*>(pm) << '\n';       // 输出: 0
}

```



```
// 10. void* 到任何类型//
void* voidp = &e;
std::vector<int>* p = static_cast<std::vector<int>*>(voidp);
system("pause");
}
```

2. const_cast< type >(expression)

const_cast提供了C语言没有的特性，即在一个const 变量让其变为可以被赋值，或一个普通变量让其变为拥有const 属性。例如：

```
const int i = 9;    // 不可变//
const_cast<int&>(i) = 10;
/// i = 11; 仍为const变量//

int y = 12;
const_cast<const int&>(y);    // 给y变量加上const属性，使其不可被赋值
// const_cast<const int&>(y) = 13;  // 不可被赋值

// 注意
// const_cast<type>()中的type只能是引用或者指针类型
// 不能写作以下形式：const_cast<int>(i)
```

3. reinterpret_cast< type >(expression)

reinterpret_cast常用于指针指向的数据的转换，如：

```
//以前//
GET_DIAL_NUMBER_RSP_T    *pstRsp = (GET_DIAL_NUMBER_RSP_T*)pVMsg;
//现在//
GET_DIAL_NUMBER_RSP_T    *pstRsp = reinterpret_cast<GET_DIAL_NUMBER_RSP_T*>(pVMsg);
```

下面为官网给的示例。

```
#include <cstdint>
#include <cassert>
#include <iostream>

int f() { return 42; }

int main()
{
    int i = 7;

    // 指针到整数并转回//
    std::uintptr_t v1 = reinterpret_cast<std::uintptr_t>(&i);    // static_cast 为错误//
    std::cout << "The value of &i is 0x" << std::hex << v1 << '\n';
    int* p1 = reinterpret_cast<int*>(v1);
    assert(p1 == &i);

    // 到另一函数指针并转回//
```

```

void(*fp1)() = reinterpret_cast<void(*)>(f);
// fp1(); 未定义行为。//
int (*fp2)() = reinterpret_cast<int(*)>(f);
std::cout << std::dec << fp2() << '\n';    // 安全//

// 通过指针的类型别名使用//
char* p2 = reinterpret_cast<char*>(&i);
if (p2[0] == '\x7')
    std::cout << "This system is little-endian\n";
else
    std::cout << "This system is big-endian\n";

// 通过引用类型别名使用//
reinterpret_cast<unsigned int &>(i) = 42;
std::cout << i << '\n';

const int& const_iref = i;
// int &iref = reinterpret_cast<int&>(const_iref); // 编译错误--不能取出const//
// 必须用const_cast 代替: int &iref = const_cast<int&>(const_iref);
}

```

4. dynamic_cast< type >(expression)

`dynamic_cast<type>(expression)` 常用于基类和派生类的指针的指针转换，如工厂模式中的转换。但起作用更多的是动态类型的检查而不是类型转换。就比如类型检查成功则正确赋值，不成功返回空指针或抛出异常(`bad_cast`)。

```

#include <iostream>

struct V
{
    virtual void f() {};    // 必须为多态使用运行时检查的dynamic_cast
};

struct A : virtual V {};
struct B : virtual V
{
    B(V* v, A* a)
    {
        // 构造中转型(见后述D的构造函数中的调用)
        dynamic_cast<B*>(v);    // 良好定义: v有类型V*, B的V基类, 产生B*
        dynamic_cast<B*>(a);    // 未定义行为: a有类型A*, A非B的基类
    }
};

struct D : A, B
{
    D() : B((A*)this, this) {}
};

struct Base
{
    virtual ~Base() {}
}

```

```
};

struct Derived : Base
{
    virtual void name() {}
};

int main()
{
    D d;    // 最终派生类
    A& a = d;    // 向上转型, 可以用dynamic_cast, 但不是必须
    D& new_d = dynamic_cast<D&>(a); // 向下转型
    B& new_b = dynamic_cast<B&>(a); // 侧向转型

    Base* b1 = new Base;
    if (Derived * d1 = dynamic_cast<Derived*>(b1))
    {
        /// 因为赋值不成功, 所以这句话不会被打印
        std::cout << "base class can't convert to Derived class\n";
    }

    // 只有派生类实体才能转换为派生类指针
    // 由于派生类都拥有基类的共同部分, 所以派生类可以直接使用基类指针访问基类共有部分
    Base* b2 = new Derived;
    if (Derived * d2 = dynamic_cast<Derived*>(b2))
    {
        std::cout << "downcast from b2 to d successful\n";
        d2->name(); // 调用安全
    }

    delete b1;
    delete b2;
}
```

5. C++Style强制类型转换与CStyle强制类型转换对比

概念:

C++Style强制类型转换: 即使用了**static_cast**(expression)、**const_cast**(expression)、**reinterpret_cast**(expression)和**dynamic_cast** (expression)这四个关键字来进行强制类型转换。

CStyle强制类型转换: 即使用老式的小括号类型进行的强制类型转换 (如: **(type)expression;**) 。

首先强调, 尽量**不要使用强制类型转换**。

1. C++强制类型转换的关键字提供了CStyle的强制转换不能提供的功能 (如: **const_cast**提供了增加删除**const**属性的类型转换、**dynamic_cast**提供了适用于基类与派生类的转换)
2. 在排查强制转换的问题时, C++Style的关键字更容易搜索 (如: 直接搜索**static_cast**就可以找到, 而CStyle强制类型转换很难通过搜索特定字符串在代码中定位)
3. C++强制类型转换的关键字更加复杂, 更加难以记忆与书写从源头上提示了我们尽量不要使用强制类型转换。
4. 最重要的一点是: **CStyle强制类型转换是专门为C语言设计, 在.cpp文件中使用可能会出错**。见下面例子:

```

#include <iostream>
#include <stdio.h>
#include <string.h>
#include <conio.h>

using namespace std;
int main()
{
    float a = 1.0f;
    cout << (int&)a << endl;    // 输出: 1065353216
    cout << boolalpha << ((int)a == (int&)a) << endl;    // 输出false

    float b = 0.0f;
    cout << (int&)b << endl;    // 输出: 0
    cout << boolalpha << ((int)b == (int&)b) << endl;    // 输出: true
    system("pause");
    return 0;
}

```

在上面的例子中，因为float的底层存储有几位用来存储小数点的位置信息，跟int型的底层略有差别，所以导致在使用CStyle强制类型转换时，会不给任何提示的输出错误的值，而使用C++Style强制类型转换时，编译器会提示错误。

constexpr关键字

constexpr 指定符声明可以在编译时求得函数或变量的值。

constexpr 变量必须满足下列要求：

1. 其类型必须是字面类型 (LiteralType)
2. 它必须被**立即初始化**
3. 其初始化的完整表达式，包括所有隐式转换、构造函数调用等，都必须是常量表达式

constexpr 函数必须满足下列要求：

1. 它必须不是虚函数 (C++20前)
2. 其**返回类型**必须是字面类型 (LiteralType)
3. 其**每个参数**都必须是字面类型 (LiteralType)
4. 函数体必须不含：asm 声明、goto 语句、goto的标签、try块(C++20前)、非字面行常量的定义、不进行初始化的定义。

constexpr 构造函数必须满足下列要求：

1. 其每个参数都必须是字面类型 (LiteralType)
2. 该类必须无虚基类
3. 该构造函数必须无函数try块(C++20前)
4. 构造函数体必须满足 constexpr 函数体的制约
5. 对于class或struct的构造函数，每个子对象和每个非变体非static数据成员必须被初始化。若类联合体类，对于其每个非空匿名联合体成员，必须恰好有一个变体成员被初始化。
6. 对于非空union的构造函数，恰好有一个非static数据成员被初始化。
7. 每个被选座初始化非static成员和基类的构造函数必须时constexpr构造函数
8. 对于 constexpr 函数模板和类模板的 constexpr 函数成员，必须至少有一个特化满足上述要求。其他特化仍被认为是 constexpr，尽管常量表达式中不能出现这种函数的调用。（这句话我不太理解(李建聪)）

```

#include <iostream>
#include <stdexcept>

// C++11 constexpr函数使用递归而非迭代
// (C++14 constexpr函数可使用局部变量和循环)
constexpr int factorial(int n)
{
    return n <= 1 ? 1 : (n * factorial(n - 1));
}

// 字面类
class conststr
{
    const char* p;
    std::size_t sz;
public:
    template<std::size_t N>
    constexpr conststr(const char(&a)[N]) : p(a), sz(N - 1) {}

    // constexpr 函数通过抛异常来提示错误
    // C++11中, 他们必须用条件运算符?: 这么做
    constexpr char operator[](std::size_t n) const
    {
        return n < sz ? p[n] : throw std::out_of_range("");
    }
    constexpr std::size_t size() const { return sz; }
};

// C++11 constexpr 函数必须把一旦放在单条return语句中
// (C++14无该要求)
constexpr std::size_t countlower(conststr s, std::size_t n = 0, std::size_t c = 0)
{
    return n == s.size() ?
        c :
        'a' <= s[n] && s[n] <= 'z' ? countlower(s, n+1, c+1) : countlower(s, n+1, c);
}

// 输出函数要求编译时常量, 以测试
template<int n>
struct constN
{
    constN() { std::cout << n << '\n'; }
};

int main()
{
    std::cout << "4!=";
    constN<factorial(4)> out1; // 在编译时计算

    volatile int k = 8; // 不允许使用volatile者优化
    std::cout << k << "!=" << factorial(k) << '\n'; // 运行时计算

    std::cout << "the number of lowercast letters in \"Hello, world!\" is ";
}

```

```
constN<countlower("Hello, world!")> out2; // 字面型常量隐式转换到conststr
}
```

enable_if 关键字

```
#include <type_traits>
#include <iostream>
#include <string>

namespace detail { struct inplace_t {}; }
void* operator new(std::size_t, void* p, detail::inplace_t)
{
    return p;
}

// #1, 通过返回类型使用
template<class T, class... Args>
typename std::enable_if<std::is_trivially_constructible<T, Args&&...>::value>::type
construct(T* t, Args&& ... args)
{
    std::cout << "constructing trivially constructible T\n";
}

// #2
template<class T, class... Args>
std::enable_if_t<!std::is_trivially_constructible<T, Args&&...>::value> // 使用帮助类型
construct(T* t, Args&& ... args)
{
    std::cout << "constructing non-trivially constructible T\n";
    new(t, detail::inplace_t{}) T(args...);
}

// #3, 通过形参启用
template<class T>
void destroy(T* t,
    typename std::enable_if<std::is_trivially_destructible<T>::value>::type* = 0)
{
    std::cout << "destroying trivially destructible T\n";
}

// #4, 通过模板形参启用
template<class T, typename std::enable_if <
    !std::is_trivially_destructible<T>{} &&
    (std::is_class<T>{} || std::is_union<T>{}),
    int>::type = 0>
void destroy(T * t)
{
    std::cout << "destroying non-trivially destructible T\n";
    t->~T();
}

// #5, 通过模板形参启用
```

```

template<class T, typename = std::enable_if_t<std::is_array<T>::value>>
void destroy(T * t) // 注意, 不修改函数签名
{
    for (std::size_t i = 0; i < std::extent<T>::value; ++i)
    {
        destroy((*t)[i]);
    }
}

/*
template<class T,
        typename = std::enable_if_t<std::is_void<T>::value> >
void destroy(T* t){} // 错误: 与 #5 拥有相同签名
*/

// A 的部分特化通过模板形参启用
template<class T, class Enable = void>
class A {};

template<class T>
class A<T, typename std::enable_if<std::is_floating_point<T>::value>::type>
{}; // 为浮点类型特化

int main()
{
    std::aligned_union_t<0, int, std::string> u;

    construct(reinterpret_cast<int*>(&u));
    destroy(reinterpret_cast<int*>(&u));

    construct(reinterpret_cast<std::string*>(&u), "Hello");
    destroy(reinterpret_cast<std::string*>(&u));

    A<int> a1; // OK, 匹配初等模板
    A<double> a2; // OK, 匹配部分特化

    system("pause");
}

```

STL算法速览

注意: 如某容器的成员函数有对应的功能函数, 应**优先选用成员函数**而非以下STL算法函数。因为特定容器的成员函数为该容器的实现提供了更好的算法, 相比较普适性较强的STL算法函数拥有更好的时间复杂度和空间复杂度。

STL算法:

- 非更易型算法 (Nonmodifying algorithms)
- 更易型算法 (Modifying algorithms)
- 移除型算法 (Removing algorithms)
- 变序型算法 (Mutating algorithms)
- 排序算法 (Sorting algorithms)
- 已排序区间算法 (Sorted-range algorithms)
- 数值算法 (Numeric algorithms)

1. 非更易型算法 (Nonmodifying algorithms)

名称	效果
for_each()	对每个元素执行某操作
count()	返回元素个数
count_if()	返回满足某一准则(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
minmax_element()	返回最小值和最大值元素
find()	查找“与被传入值相等”的第一个元素
find_if()	查找“满足某个准则”的第一个元素
find_if_not()	查找“不满足某个准则”的第一个元素
search_n()	查找“具备某特性”之前n个连续元素
search()	查找某个子区间的第一次出现位置
find_end()	查找某个子区间的最后一次出现的位置
find_first_of()	查找“数个可能元素中的第一个出现者”
adjacent_find()	查找连续两个相等（或者说符合特定准则）的元素
equal()	判断两区间是否相等
is_permutation()	连个不定序区间是否含有相等元素
mismatch()	返回两序列的各组对应元素中的第一对不相等元素
lexicographical_compare()	判断在“字典顺序”(lexicographically)下某序列是否小于另一序列
is_sorted()	返回“是否区间内的元素已排序”
is_sorted_until()	返回“区间内的元素是否基于某准则被分割为两组”
partition_point()	返回区间内的一个分割元素， 它把元素切割为两组， 其中一组满足某个 predicate， 另一组则不然
is_heap()	返回“是否区间内的元素形成一个heap”
is_heap_until()	返回“是否所有元素都吻合某准则的元素”
all_of()	返回“是否所有元素都吻合某准则”
any_of()	返回“是否至少一个元素吻合某准则”
none_of()	返回“是否无任何元素吻合某准则”

2. 更易型算法 (Modifying algorithms)

名称	效果
for_each()	针对每个元素执行某项操作
copy()	从某个元素开始，复制某个区间
copy_if()	复制那些“符合某个给定准则”的元素
copy_n()	复制n个元素
copy_backward()	从最后一个元素开始，复制某个区间
move()	从第一个元素开始，搬移某个区间
move_backward()	从最后一个元素开始，搬移某个区间
transform()	改动（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
iota()	将所有元素以一系列的递增值取代
replace()	将具有某特定值的元素替换为另一个值
replace_copy()	复制整个区间，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，并将符合某准则的元素替换为另一个值

3. 移除型算法 (Removing algorithms)

名称	效果
remove()	将“等于某特定值”的元素全部移除
remove_if()	将“满足某准则”的元素全部移除
remove_copy()	将“不等于某特定值”的元素全部复制到他处
remove_copy_if()	将“不满足某准则”的元素全部复制到他处
unique()	一处毗邻的重复元素
unique_copy()	一处毗邻的重复元素，并复制到他处

4. 变序型算法 (Mutating algorithms)

名称	效果
reverse()	将元素的次序逆转
reverse_copy()	复制的同时，逆转元素顺序
rotate()	旋转元素顺序
rotate_copy()	复制的同时，旋转元素次序
next_permutation()	得到元素的下一个排列次序
prev_permutation()	得到元素的上一个排列次序
shuffle()	将元素的次序随机打乱
random_shuffle()	将元素的次序随机打乱
partition()	改变元素次序，是“符合某准则”者移到前面
stable_partition()	和partition()类似，但保持“与准则相符”和“与准则不符”之各个元素之间的相对位置
partition_copy()	改变元素次序，使“符合某准则”者移到前面，过程中会复制元素

5. 排序算法 (Sorting algorithms)

名称	效果
sort()	对所有元素排序
stable_sort()	对于所有元素排序，并保持相等元素之间的相对次序
partial_sort()	排序，知道前n个元素就位
partial_sort_copy()	排序，直到前n个元素九尾；将结果复制于他处
nth_element()	根据第n个位置进行排序
partition()	改变元素次序，是“符合某准则”者一道前面，过程中还会复制元素
make_heap()	将某个区间转换成一个heap
push_heap()	将元素加入一个heap
pop_heap()	从heap移除一个元素
sort_heap()	对heap进行排序（完成后就不再是个heap了）

检验是否排序算法

名称	效果
is_sort()	检验区间内的元素是否都已排序
is_sorted_until()	返回区间内第一个“破坏排序状态”的元素
is_partitioned()	检验区间内的元素是否根据某个准则被分为两组
partition_point()	返回区间内的分割点，他把区间分割为“满足”和“不满足”某predicate的两组
is_heap()	检验区间内的元素是否都排序成为一个heap
is_heap_until()	返回区间内第一个“破坏heap排序状态”的元素

6. 已排序区间算法 (Sorted-range algorithms)

名称	效果
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个区间中
lower_bound()	查找第一个“大于等于某给定值”的元素
upper_bound()	查找第一个“大于某给定值”的元素
equal_range()	返回“等于某给定值”的所有元素构成的区间
merge()	将两个区间的元素合并
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，形成一个已排序区间
set_symmetric_difference()	找出“只出现两区间之一”的所有元素，形成一个已排序区间
inplace_merge()	将两个连贯的已排序区间合并
partition_point()	用一个判断式分割区间，返回分割元素

7. 数值算法 (Numeric algorithms)

名称	效果
accumulate()	结合所有元素（求总和、求乘积）
inner_product()	结合两区间内的所有元素
adjacent_difference()	将每个元素和其前以元素结合
partial_sum()	将每个元素和其之前的所有元素结合

std::sort

以升序排序范围 `[first, last)` 中的元素。不保证维持相等元素的顺序。

复杂度: $O(N \log(N))$ 次比较, 其中 $N = \text{std::distance}(\text{first}, \text{last})$ (C++11起)

```
#include <algorithm>
#include <functional>
#include <array>
#include <iostream>

int main()
{
    std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

    // 用默认的 operator< 排序
    std::sort(s.begin(), s.end());
    for (auto a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';

    // 用标准库比较函数对象排序
    std::sort(s.begin(), s.end(), std::greater<int>());
    for (auto a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';

    // 用自定义函数对象排序
    struct {
        bool operator()(int a, int b) const
        {
            return a < b;
        }
    } customLess;
    std::sort(s.begin(), s.end(), customLess);
    for (auto a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';

    // 用 lambda 表达式排序
    std::sort(s.begin(), s.end(), [](int a, int b) {
        return b < a;
    });
    for (auto a : s) {
        std::cout << a << " ";
    }
    std::cout << '\n';
}
```

输出:

```
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

std::stable_sort

以升序排序范围 `[first, last)` 中的元素。保证保持等价元素的顺序。

复杂度：

应用 `cmp` $O(N \cdot \log(N)^2)$ 次，其中 $N = \text{std::distance}(first, last)$ 。若额外内存可用，则复杂度为 $O(N \cdot \log(N))$ 。

注意：

此函数试图分配等于待排序序列长度的临时缓冲区。若分配失败，则选择较低效的算法。

```
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>

struct Employee {
    int age;
    std::string name; // 不参与比较
};

bool operator<(const Employee &lhs, const Employee &rhs) {
    return lhs.age < rhs.age;
}

int main()
{
    std::vector<Employee> v = {
        {108, "Zaphod"},
        {32, "Arthur"},
        {108, "Ford"},
    };

    std::stable_sort(v.begin(), v.end());

    for (const auto &e : v) {
        std::cout << e.age << ", " << e.name << '\n';
    }
}
```

输出：

```
32, Arthur
108, Zaphod
108, Ford
```

std::partial_sort

重排元素，使得范围 `[first, middle)` 含有范围 `[first, last)` 中已排序的 `middle - first` 个最小元素。不保证保持相等的元素顺序。范围 `[middle, last)` 中剩余的元素顺序未指定。

复杂度：

约 $(last - first) \log(middle - first)$ 次应用 `cmp`。

```
#include <algorithm>
#include <functional>
#include <array>
#include <iostream>

int main()
{
    std::array<int, 10> s{5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

    std::partial_sort(s.begin(), s.begin() + 3, s.end());
    for (int a : s) {
        std::cout << a << " ";
    }
}
```

可能的输出：

```
0 1 2 7 8 6 5 9 4 3
```

std::nth_element

`nth_element` 是部分排序算法，它重排 `[first, last)` 中元素，使得：

- `nth` 所指向的元素被更改为假如 `[first, last)` 已排序则该位置会出现的元素。
- 这个新的 `nth` 元素前的所有元素小于或等于新的 `nth` 元素后的所有元素。

更正式而言，`nth_element` 以升序部分排序范围 `[first, last)`，使得对于任何范围 `[first, nth)` 中的 `i` 和任何范围 `[nth, last)` 中的 `j`，都满足条件 $!(a[j] < a[i])$ 。置于 `nth` 位置的元素则准确地是假如完全排序范围则应出现于此位置的元素。

简单的来说就是给定一个数值，比这个数值大的放该元素后面，比这个数值小的放前面。

复杂度：

平均与 `std::distance(first, last)` 成线性。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

int main()
{
    std::vector<int> v{5, 6, 4, 3, 2, 6, 7, 9, 3};
```

```

std::nth_element(v.begin(), v.begin() + v.size()/2, v.end());
std::cout << "The median is " << v[v.size()/2] << '\n';

std::nth_element(v.begin(), v.begin()+1, v.end(), std::greater<int>());
std::cout << "The second largest element is " << v[1] << '\n';
}

```

输出：

```

The median is 5
The second largest element is 7

```

STL排序算法比较

注意：不要尝试写出比std::sort()更快的排序算法(如：introsort、pdqsort实际上都没有std::sort块)如无特殊需求应优先选用STL封装好的排序算法，当前最快的为std::sort()。

std::sort()传统上是采用quicksort算法。因此保证了很好的平均效能，复杂度为 $n \times \log(n)$ ，但最差情况下可能为 n^2 。如果“避免最差情况”对你是一件重要的事，你应该采用其他算法，如partial_sort()或stable_sort()。

std::partial_sort()传统上采用heapsort算法。因此它在任何情况下保证 $n \times \log(n)$ 复杂度。

虽然partial_sort()拥有较好的复杂度，但sort()在多数情况下却拥有较好的运行期效能。partial_sort()的优点是它在任何时候都保证 $n \times \log(n)$ 复杂度，绝不会变成二次复杂度。

partial_sort()还有一种特殊能力：如果你只需前n个元素排序，它可在完成任务后立刻停止。

std::stable_sort()传统上采用mergesort。它对所有元素进行排序。然而只有在内存充足的前提下它才有 $n \times \log(n)$ 复杂度，否则其复杂度为 $n \times \log(n) \times \log(n)$ 。stable_sort()的有点是会保持先庚元素之间的相对次序。

如果你只需要前n个排序元素，或只需要令最先或最后的n个元素（未排序）就位，可以使用nth_element()。你可以利用nth_element()将元素按照某排序准则分割成两个子集，也可以利用partition()或stable_partition()达到相同效果。三者区别如下：

- 对于nth_element(), 在第一子集中指出元素个数(当然也就确定了第二子集的元素个数)例如：

```

// move the four lowes elements to the front
std::nth_element(coll.begin(), // beginning of range
                 coll.begin()+3 // position between first and second part
                 coll.end()     // end of range);

```

然而调用后你并不精确知道第一子集和第二子集之间有什么不同。两部分都可能包含“与第n个元素相等”的元素。

- 对于partition(), 你必须传入“将第一子集和第二子集区别开”的精确排序准则：

```

// move all elements less than seven to the front
vector<int>::iterator pos;
pos = partition(coll1.begin(), coll1.end(),
               [](int elem) { return elem<7;});

```

调用之后你并不知道第一和第二子集内各有多少元素。返回的pos用来指出第二子集七点。第二子集的所有元素都不满足上述（被指出的）准则。

- `stable_partition()` 的行为类似`partition()`，不过更具额外能力，保证两子集内的元素的相对次序维持不变。

std::lock_guard用法

创建 `lock_guard` 对象时，它试图接收给定互斥(锁)的所有权。控制离开创建 `lock_guard` 对象的作用域时，销毁 `lock_guard` 并释放互斥(锁)。`lock_guard` 类不可复制。

使用 `lock_guard` 可以避免加锁后忘记解锁，另一方面在一个函数具有多个return分支的时候，重复调用`unlock()`函数会显得代码很冗余，而 `lock_guard` 只用在加锁的时候调用该函数，该 `lock_guard` 变量出作用域时自动解锁。

```
#include <thread>
#include <mutex>
#include <iostream>

int g_i = 0;
std::mutex g_i_mutex; // 保护 g_i

void safe_increment()
{
    std::lock_guard<std::mutex> lock(g_i_mutex);
    ++g_i;

    std::cout << std::this_thread::get_id() << ": " << g_i << '\n';

    // g_i_mutex 在锁离开作用域时自动释放
}

int main()
{
    std::cout << "main: " << g_i << '\n';

    std::thread t1(safe_increment);
    std::thread t2(safe_increment);

    t1.join();
    t2.join();

    std::cout << "main: " << g_i << '\n';
}
```

1. 用于指定锁定策略的标签类型

`std::defer_lock`、`std::try_to_lock` 和 `std::adopt_lock` 分别是空结构体标签类型 `std::defer_lock_t`、`std::try_to_lock_t` 和 `std::adopt_lock_t` 的实例。

它们用于为 [std::lock_guard](#)、[std::unique_lock](#) 及 [std::shared_lock](#) 指定锁定策略。

类型	效果
defer_lock_t	不获得互斥的所有权
try_to_lock_t	尝试获得互斥的所有权而不阻塞
adopt_lock_t	假设调用方线程已拥有互斥的所有权

```

#include <mutex>
#include <thread>

struct bank_account {
    explicit bank_account(int balance) : balance(balance) {}
    int balance;
    std::mutex m;
};

void transfer(bank_account &from, bank_account &to, int amount)
{
    // 锁定两个互斥而不死锁.
    std::lock(from.m, to.m);
    // 保证二个已锁定互斥在作用域结尾解锁.
    std::lock_guard<std::mutex> lock1(from.m, std::adopt_lock);
    std::lock_guard<std::mutex> lock2(to.m, std::adopt_lock);

    // 等价方法: .
    //     std::unique_lock<std::mutex> lock1(from.m, std::defer_lock);
    //     std::unique_lock<std::mutex> lock2(to.m, std::defer_lock);
    //     std::lock(lock1, lock2);

    from.balance -= amount;
    to.balance += amount;
}

int main()
{
    bank_account my_account(100);
    bank_account your_account(50);

    std::thread t1(transfer, std::ref(my_account), std::ref(your_account), 10);
    std::thread t2(transfer, std::ref(your_account), std::ref(my_account), 5);

    t1.join();
    t2.join();
}

```

变参宏介绍

```

#include <iostream>

/** 注意宏定义中的三个点`...`就是假定有不定数目(可以为0)个参数放进这个宏定义 */

```

```

/** 其中__VA_ARGS__对应着宏定义中的三个点代表变参放置的位置 */
#define PR(X, ...)  printf(X, __VA_ARGS__)

int main()
{
    /** 这里变参个数为零 */
    PR("This is Test\n");
    /** 这里变参个数为3 */
    PR("int: %d, float: %f, string: %s\n", 111, 222.3333, "Hello world!");
    system("pause");
    return 0;
}

```

宏定义中的#运算符

宏定义的后半部分出现的符号#代表了字符串，符号##代表了实际的代码。

```

#include <iostream>
#include <cassert>

/** 代表了变量x会被实际展开成printf("x") */
#define PRINT(X)    printf(#X)

/** 代表了该代码会被展开成a[index] */
#define VAR_NAME(a, index)  a[##index]

int main()
{
    PRINT(This is a String!\n);    // 被展开为printf("This is a String!")

    int a[2] = { 99, 100 };
    assert(VAR_NAME(a, 0) == a[0]);    // VAR_NAME(a, 0)被展开为a[0]
    assert(VAR_NAME(a, 1) == a[1]);    // VAR_NAME(a, 1)被展开为a[1]
    a[0] = 88;
    assert(VAR_NAME(a, 0) == 88);

    system("pause");
    return 0;
}

```

函数变参介绍

函数变参多用于实现sprintf类的函数，用法如下。

```

#include <iostream>
#include <stdarg.h>

/** smdv4.0的拨测卡驱动日志函数代码 */
void DrvLog(char* szFormat, ...)
{
    /** 使用va_list来代表函数入参的变参列表 */
    va_list      vaLog;

```

```

    /** 注意不可以漏掉调用va_start()和va_end() */
    va_start(vaLog, szFormat);

    char acLog[1024] = { 0 };
    vsnprintf(acLog, sizeof(acLog) - 1, szFormat, vaLog);

    va_end(vaLog);
    std::cout << acLog << std::endl;
}

int main()
{
    DrvLog("This is Test!");
    DrvLog("int: %d, float: %f, string: %s!", 111, 222.3333, "Hello world!");
    system("pause");
}

```

成员函数的引用限定

```

#include <iostream>

struct Test
{
    void workLvalue() &
    {
        std::cout << " 只能被一个左值对象调用\n";
    }

    void workRvalue() &&
    {
        std::cout << " 只能被一个右值对象调用\n";
    }
};

/** 该函数返回一个Test类型的临时变量（右值） */
Test ReturnRValueTest()
{
    Test t;
    return t;
}

int main()
{
    Test LValueObject;
    LValueObject.workLvalue(); // ok
    // LValueObject.workRvalue(); // 左值对象不能调用右值限定的成员函数，编译不通过

    ReturnRValueTest().workRvalue(); // ok
    //ReturnRValueTest().workLvalue(); // 右值对象不能调用左值限定的成员函数，编译不通过
}

```

虚函数介绍

虚函数被设计出来就是为了实现**派生类覆盖基类的函数实现**，来让用户可以在**程序运行期**决定来使用哪种底层实现，最简单的例子就是使用哪个数据库。

假如我们要写一个数据库模块的操作函数，我们通常会直接把MySQL的API抄过来封装一个函数就行了，但这样会导致更换使用的数据库时重写业务逻辑代码，可扩展性变差。

所以这个时候有人想出来了一个方法，就是我写出来MySQL的API，Oracel的API，SQLite的API后，在外层再封装一个统一接口的类。通过一个指针来指向MySQL的API实现或Oracel的API实现。这样**业务层调用数据库操作时，只用调用外层封装的统一接口，而不用在乎底层实现**。使用**哪个数据库让客户随时随地选择**，而**业务层代码不变**，底层实现的选择就交给编译器吧。

那么问题来了，怎么让编译器**动态选择底层的数据库API的实现**？

由于派生类都从基类继承了基类成员，所以派生类除去其再派生类中定义的东西，剩下的都是基类部分。**如果假设派生类不定义任何东西，那么这个派生类实例可以说几乎等价于基类实例**。也就是说我们可以**直接返回一个基类指针或引用指向派生类实例**，我们只调用其基类也拥有的成员函数就没有任何问题。如下：

```
#include <iostream>

class BASE_DB_API
{
public:
    bool Insert() { return true; }
    bool Update() { return true; }
};

class DERIVE_DB_API : public BASE_DB_API
{
    /** 在继承BASE_DB_API时，也自动把Insert()、Update()包含进来了 */
};

int main()
{
    DERIVE_DB_API Instance;
    Instance.Insert(); /** OK, 使用BASE_DB_API中定义的Insert() */
    Instance.Update(); /** OK, 使用BASE_DB_API中定义的Update() */
}
```

到了这一步你可能会问不同派生类实例只能调用其基类一种实现，根本没有选择底层实现的余地。那么我们来引入最重要的一部分：**虚函数**。**让基类的统一接口函数不实现，而采用派生类的不同实现**，只需要加上下面这些东西。

```
#include <iostream>

class BASE_DB_API
{
public:
    /** 在统一的接口处加上virtual，后面加上=0来声明这个函数在基类不实现，让编译器去不同派生类中找 */
    virtual bool Insert() = 0;
    virtual bool Update() = 0;
};
```

```

class MYSQL_DERIVE_DB_API : public BASE_DB_API
{
public:
    /** 与上面代码不同的时，这里只继承了两个接口函数的声明，没有继承其实现 */
    /** 现在来写接口函数的实现 */
    virtual bool Insert() override
    {
        std::cout << "这里是派生类 MYSQL 的 Insert 实现" << std::endl;
        return true;
    }

    /** 注意：在覆写基类虚函数时，需要显式使用override关键字 */
    virtual bool update() override
    {
        std::cout << "这里是派生类 MYSQL 的 Update 实现" << std::endl;
        return true;
    }
};

class ORACEL_DERIVE_DB_API : public BASE_DB_API
{
public:
    /** 另一个派生类，写Oracel的数据库接口实现 */
    virtual bool Insert() override
    {
        std::cout << "这里是派生类 ORACEL 的 Insert 实现" << std::endl;
        return true;
    }

    virtual bool update() override
    {
        std::cout << "这里是派生类 ORACEL 的 Update 实现" << std::endl;
        return true;
    }
};

int main()
{
    MYSQL_DERIVE_DB_API MySQLInstance;
    MySQLInstance.Insert();    /** OK，使用MYSQL_DERIVE_DB_API中定义的Insert() */
    MySQLInstance.Update();    /** OK，使用MYSQL_DERIVE_DB_API中定义的Update() */

    ORACEL_DERIVE_DB_API OracelInstance;
    OracelInstance.Insert();    /** OK，使用ORACEL_DERIVE_DB_API中定义的Insert() */
    OracelInstance.Update();    /** OK，使用ORACEL_DERIVE_DB_API中定义的Update() */
    system("pause");
}

```

到了这一步就差给这两个派生类加一个封装，让用户只用调用接口而不用在乎底层实现的类（工厂类）。

```

/** 延续以上的代码 */

```

```

/** 可以把这个函数叫做工厂 */
BASE_DB_API* CreateDBInstance(int i)
{
    switch (i)
    {
        case 1:
            return new MYSQL_DERIVE_DB_API;

        case 2:
            return new ORACEL_DERIVE_DB_API;
    }
}

/** main函数改为 */
int main()
{
    BASE_DB_API* DB_Instanc = nullptr;
    /** 想用MySQL的数据库API时这样写 */
    DB_Instanc = CreateDBInstance(1);
    DB_Instanc->Insert();
    DB_Instanc->Update();

    /** 想用Oracle的数据库API时入参换为2就行了，其他的一样 */
    DB_Instanc = CreateDBInstance(2);
    DB_Instanc->Insert();
    DB_Instanc->Update();

    system("pause");
    return 0;
}

```

最后完整的代码如下：

```

#include <iostream>

class BASE_DB_API
{
public:
    /** 在统一的接口处加上virtual，后面加上=0来声明这个函数在基类不实现，让编译器去不同派生类中找 */
    virtual bool Insert() = 0;
    virtual bool Update() = 0;
};

class MYSQL_DERIVE_DB_API : public BASE_DB_API
{
public:
    /** 与上面代码不同的时，这里只继承了两个接口函数的声明，没有继承其实现 */
    /** 现在来写接口函数的实现 */
    virtual bool Insert() override
    {
        std::cout << "这里是派生类 MYSQL 的 Insert 实现" << std::endl; return true;
    }
}

```

```

    /** 注意：在覆写基类虚函数时，需要显式使用override关键字 */
    virtual bool update() override
    {
        std::cout << "这里是派生类 MYSQL 的 Update 实现" << std::endl; return true;
    }
};

class ORACEL_DERIVE_DB_API : public BASE_DB_API
{
public:
    /** 另一个派生类，写Oracel的数据库接口实现 */
    virtual bool Insert() override
    {
        std::cout << "这里是派生类 ORACEL 的 Insert 实现" << std::endl;
        return true;
    }

    virtual bool update() override
    {
        std::cout << "这里是派生类 ORACEL 的 Update 实现" << std::endl;
        return true;
    }
};

/** 工厂函数 */
BASE_DB_API* CreateDBInstance(int i)
{
    switch (i)
    {
        {
            case 1: return (new MYSQL_DERIVE_DB_API);
            case 2: return (new ORACEL_DERIVE_DB_API);
        }
    }
}

/** main函数改为 */
int main()
{
    BASE_DB_API* DB_Instan = nullptr;
    if (true) DB_Instan = CreateDBInstance(1); /** 想用MySQL的数据库API时这样写 */
    else      DB_Instan = CreateDBInstance(2); /** 想用Oracel的数据库API时入参换为2就行了 */

    DB_Instan->Insert();
    DB_Instan->Update();

    system("pause");
    return 0;
}

```

基类与派生类之间的转换规则

三条规则：

- 从派生类向基类的(隐式)类型转换只对**指针和引用类型有效**
- **基类向派生类不存在隐式类型转换。**
- 和任何其他类内成员一样，派生类向基类的类型转换也可能会由于**访问受限**而变得不可行

1. 从派生类向基类的(隐式)类型转换只对指针和引用类型有效

首先要明确的时，派生类和基类是两种不同的类型，其之间只能通过指针或引用才能让派生类**假装**是基类。

2. 基类向派生类不存在隐式类型转换

之所以存在派生类向基类的类型转换时因为**每个派生类对象都包含一个基类部分**，而基类的引用或指针可以绑定到该基类部分上。一个基类的对象既可以以独立的形式存在也可以作为派生类对象的一部分存在。如果基类对象不是派生类对象的一部分，则它只含有基类定义的成员，而不含有派生类定义的成员。

存在基类指针(引用)转换为派生类指针(引用)么？

答案是：存在，但基类指针或引用**必须本身就指向要转换为的派生类实例**才行。而且也不能直接转换，需要用到一些技巧。

见下面无法正确转换的例子：

```
#include <iostream>

struct Base {/** ... */};
struct Derived : public Base {/** ... */};

int main()
{
    Derived derived;
    Base* ptr_base = &derived;  /** 基类指针指向派生类 */
    Base& ref_base = derived;   /** 基类引用指向派生类 */

    Derived* derived2 = ptr_base;    ///< 编译报错无法从“Base* ”转换为“Derived* ”.
    Derived& derived3 = ref_base;    ///< 编译器报错：无法从“Base &”转换为“Derived &”.

    system("pause");
}
```

上面例子中**即使基类指针或引用指向了派生类实例**，也不能违反**基类不能转换为派生类**的规则。但如果真的需要这样做的，有三种方法。

- **方法一：**使用关键字 `static_cast<>()`，**优点：**简单就完事。**缺点：**关键字 `static_cast<>()` 屏蔽了编译器的类型检查。少了这个类型检查将会导致**其他派生类**转换为基类而导致未定义行为。

```
Derived* derived4 = static_cast<Derived*>(ptr_base);    ///< 编译通过，但没有执行类型检查
Derived& derived5 = static_cast<Derived&>(ref_base);    ///< 编译通过，但没有执行类型检查
```

- **方法二：**使用专用关键字 `dynamic_cast<>()`，**优点：**这个关键字被设计出来就是为解决动态类型检查的，优点不用赘述。**缺点：****运行期才会报错**，使用麻烦（多写代码），可能会抛出异常。还一定要定义基类的虚析构函数。

```
#include <iostream>
```



```

#include <cassert>

struct Base { virtual ~Base() = default; };
struct Derived : public Base { /** ... */ };
struct OtherDerived : public Base { /** ... */ };

int main()
{
    Derived derived;
    Base* ptr_base = &derived;    /** 基类指针指向派生类 */
    Base& ref_base = derived;     /** 基类引用指向派生类 */

    /** 正确情况下 */
    Derived* derived6 = dynamic_cast<Derived*>(ptr_base);
    if (derived6 != NULL) std::cout << "动态类型检查成功" << std::endl;
    else std::cout << "动态类型检查失败，这句话不会被打印" << std::endl;
    Derived& derived9 = dynamic_cast<Derived&>(ref_base);    ///< 成功继续运行

    /** 以下是错误情况 */
    OtherDerived otherDerived;    /** 使用一个其他的派生类 */
    Base* ptr_Otherbase = &otherDerived;    /** 其他派生类的指针 */
    Base& ref_Otherbase = otherDerived;    /** 其他派生类的引用 */

    /** static_cast<>()关键字无法进行类型检查 */
    Derived* derived10 = static_cast<Derived*>(ptr_Otherbase);    ///< 未定义行为
    Derived& derived11 = static_cast<Derived&>(ref_Otherbase);    ///< 未定义行为

    Derived* derived8 = dynamic_cast<Derived*>(ptr_Otherbase);
    if (derived8 == nullptr) std::cout << "动态类型检查失败" << std::endl;
    Derived& derived9 = dynamic_cast<Derived&>(ref_Otherbase);    ///< 在这里抛出异常

    system("pause");
}

```

- **方法三：**封装函数让类型检查在编译期抛出错误。**优点：**调用方便，**编译期抛出错误**和**assert抛出错误**双重检查。**缺点：**只能对指针进行检查，**无法对引用进行检查**。

```

#include <iostream>
#include <cassert>

struct Base { virtual ~Base() = default; };
struct Derived : public Base { /** ... */ };
struct OtherDerived : public Base { /** ... */ };

template<typename To, typename From>
inline To CheckCast(From const& f) { return f; }

template<typename To, typename From>    // use like this: down_cast<T*>(foo);
inline To down_cast(From* f) // 只接受指针
{
    if (false) CheckCast<From*, To>(0);    /** 编译期检查 */
    assert(f == NULL || dynamic_cast<To>(f) != NULL);
    return static_cast<To>(f);
}

```

```

}

int main()
{
    Derived derived;
    Base* ptr_base = &derived;  /** 基类指针指向派生类 */

    OtherDerived otherDerived;  /** 使用一个其他的派生类 */
    Base* ptr_Otherbase = &otherDerived;  /** 其他的派生类的Base指针 */

    Derived* derived11 = down_cast<Derived*>(ptr_base);  /** 正确执行 */
    Derived* derived12 = down_cast<Derived*>(ptr_Otherbase);  /** assert报错 */

    system("pause");
}

```

3. 派生类向基类的类型转换可能会由于访问受限而变得不可行

派生类向基类转换的可访问性规则：

- 只有当派生类公有地(`public`)继承基类时，用户代码才能使用派生类向基类的转换；如果派生类继承基类的方式时私有的(`private`)或受保护的(`protected`)，则用户代码不能使用该转换。
- 不论派生类以什么方式继承基类，派生类的成员函数和友元都能使用派生类向基类的转换；派生类向其直接基类的类型转换对于派生类的成员和友元来说永远是可访问的。
- 如果派生类继承基类的方式是公有的或者受保护的，则该派生类的派生类成员和友元可以使用派生类向基类的类型转换；反之，如果派生类继承基类的方式是私有的，则不能使用。

引用《C++ primer》英文版：P 614

Tips:

For any given point in your code, if a public member of the base class would be accessible, then the derived-to-base conversion is also accessible, and not otherwise.

对于任意给定代码，假如一个基类的 `public` 的成员可以被访问，那么其派生类转换为基类的指针或引用也是可访问的，反之不行。

有一点需要特别注意的是：`static_cast<type>()` 关键字会屏蔽编译器对基类与派生类的转换时的类型检查。当使用 `static_cast<type>()` 关键字后编译器就无法检查出基类向派生类的转换（未定义行为）。

```

#include <iostream>

struct Top {};

struct Mid1 : public Top { int iMid1; };
struct Mid2 : public Top { int iMid2; };
struct Bottom : public Mid1, public Mid2 {};

void Function(Mid1& A)
{
    std::cout << "This is Mid1!" << A.iMid1 << std::endl;
}

void Function(Mid2& B)

```

```

{
    std::cout << "This is Mid2!" << B.iMid2 << std::endl;
}

int main()
{
    Bottom bottom;
    bottom.iMid1 = 22;
    bottom.iMid2 = 33;

    // Function(bottom);                ///< 两个重载函数都匹配，编译失败

    Function(static_cast<Mid1&>(bottom));    ///< 明确调用Function(Mid1& A)
    Function(static_cast<Mid2&>(bottom));    ///< 明确调用Function(Mid2& B)

    Top top;
    Function(static_cast<Mid1&>(top));    ///< 基类向派生类转换，未定义行为，但编译通过无警告。
    Function(static_cast<Mid2&>(top));    ///< 基类向派生类转换，未定义行为，但编译通过无警告。

    system("pause");
}

```

上面例子中，因为 `Bottom` 继承了两个基类 `Mid1` 和 `Mid2`，所以 `Bottom` 类型既可以隐式转换为 `Mid1` 也可以转换为 `Mid2`，所以导致重载两个函数都匹配，编译错误。

而 `Top` 类型为基类，它不能转换为其派生类使用，但关键字 `static_cast<>()` 屏蔽掉了编译器的检查，这将在运行期产生未定义行为。

为了能够让派生类显式地转换为基类而且还要进行类型检查，使用以下自定义的类型转换即可。

```

template<class T>
struct icast_identity
{
    typedef T type;
};

template <typename T>
inline T implicit_cast (typename icast_identity<T>::type x)
{
    return x;
}

int main()
{
    Bottom bottom;
    bottom.iMid1 = 22;
    bottom.iMid2 = 33;

    Function(implicit_cast<Mid1&>(bottom));    ///< 明确调用Function(Mid1& A)
    Function(implicit_cast<Mid2&>(bottom));    ///< 明确调用Function(Mid2& B)

    Top top;
    // Function(implicit_cast<Mid1&>(top));    ///< 编译失败
}

```

```
// Function(implicit_cast<Mid2&>(top)); ///  
    system("pause");  
}
```

异常处理

异常处理 (exception handling) 机制允许程序独立开发的部分能够在运行时就出现问题**进行通信**并作出**相应的处理**。异常是的是我们能够将问题的检测 and 解决过程分离开来。程序的一部分负责检测问题的出现，然后解决该问题的任务传递给程序的另一部分。检测环节无需知道问题处理模块的所有细节，反之亦然。

1. 抛出异常

在C++语言中，我们通过**抛出**(throwing)一条表达式来**引发**(raised)一个异常。被抛出的表达式的类型以及当前的调用链共同决定了哪段**处理代码**(handler)将被用来处理该异常。被选中的处理代码实在调用链中与抛出对象类型匹配的最近的处理代码。其中，根据抛出对象的类型和内容，程序的异常抛出部分会告知异常处理部分到底发生了什么错误。

当执行一个 `throw` 时，跟在 `throw` 后面的语句将不再被执行。相反，程序的控制权从 `throw` 转移到与之匹配的 `catch` 模块。该 `catch` **可能是同一函数中的局部 catch**，**也可能位于直接或间接调用了发生异常的函数的另一个函数中**。控制权从一处转移到另一处，这有两个重要的含义：

- 沿着调用链的函数可能会**提早推出**。
- 一旦程序开始执行异常处理代码，则沿着调用链创建的对象将被销毁。

因为跟在 `throw` 后面的语句将不再被执行，所以 `throw` 语句的有类似于 `return` 语句：它通常作为条件语句的一部分或者作为某个函数的最后(或者唯一)一条语句。

1.1 栈展开

当**抛出一个异常后**，程序暂停当前函数的执行过程并立即开始寻找与异常匹配的 `catch` 子句。

- 当 `throw` 出现在一个**try语句块**(try block)内时，检查与该 try 块关联的 `catch` 子句。
- 如果找到了匹配的 `catch`，就使用该 `catch` 处理异常。
- 如果这一步**没找到匹配的 catch 且该 try 语句嵌套在其他 try 块中**，则**继续检查与外层 try 匹配的 catch 子句**。
- 如果**还是找不到匹配的 catch**，则**退出当前函数**，在调用当前函数的外层函数中继续寻找。
- 如果对抛出异常的函数的调用语句位于一个 try 语句块内，则检查与该 try 块关联的 `catch` 子句。
- 如果找到了匹配的 `catch`，就使用该 `catch` 处理异常。
- 否则，如果该 try 语句嵌套在其他 try 块中，则继续检查与外层 try 匹配的 `catch` 子句。
- 如果仍然没找到匹配的 `catch`，则退出当前这个主调函数，继续在调用刚刚退出的这个函数的其他函数中寻找，以此类推。

上述过程被称为**栈展开**(stack unwinding)过程。栈展开过程沿着嵌套函数的调用链不断查找，直到找到了与异常匹配的 `catch` 子句为止；或者也可能一致没找到匹配的 `catch`，则退出主函数后过程中止。

假设找到了一个匹配的 `catch` 子句，则程序进入该子句并执行其中代码。当执行完这个 `catch` 子句后，找到与 try 块关联的最后一个 `catch` 子句后的点，并从这里继续执行。

如果没有找到匹配的 catch 子句，程序将退出。因为异常通常被认为妨碍程序正常执行的事件，所以一旦引发了某个异常，就不能对它置之不理。当找到匹配的 `catch` 时，程序将调用标准库函数 `terminate`，顾名思义，`terminate` 负责中止程序的执行过程。

1.2 栈展开过程中对象被自动销毁

在栈展开过程中，位于调用链上的语句块可能会提前退出。如果在**栈展开过程中**退出了某个块，编译器将负责确保在这个块中**创建的对象都能被正确的销毁**。如果某个局部对象的类型是**类类型**，则**该对象的析构函数将被自动调用**。与往常一样，编译器在销毁内置类型的对象时不需要做任何事情。

如果异常发生在构造函数中，则当前的对象可能只构造了一部分。有的成员已经开始初始化了，而另外一些成员在异常发生前也许还没有开始初始化。即使某个对象只构造了一部分，我们也要**确保构造的成员能被正确的销毁**（否则会发生内存泄露）。

类似的，异常也可能发生在**数组或标准库容器的元素初始化过程中**。与之前类似，如果在异常发生前已经构造了一部分元素，则我们应该确保这部分元素被正确的销毁。

1.3 析构函数与异常

析构函数总是会被执行的，但是函数中负责释放资源的代码却可能会被跳过。如果一个块分配了资源，并且在**负责释放这些资源的代码前面发生了异常**，则释放资源的代码将**不会被执行**。另一方面，类对象分配的资源将由类的析构函数负责释放。因此，如果我们使用类来控制资源的分配，就能确保无论函数正常结束还是遭遇异常，资源都能被正确地释放。（RAII的思想，在构造函数中获取资源(i.e `new`)，在析构函数中释放资源(i.e `delete`)。)

所以出于栈展开可能使用析构函数的考虑，析构函数不应该抛出不能被它自身处理的异常。换句话说，**如果析构函数需要执行某个可能抛出异常的操作，则该操作应该被放置在一个try语句块当中，并且在析构函数内部得到处理**（如果不这样做的话，程序会马上被终止）。

注：所有标准库类型都能保证它们的析构函数不会引发异常。

1.4 异常对象

**** 异常对象 (exception object) 是一种特殊的对象，编译器使用异常抛出表达式来对异常对象进行拷贝初始化。因此，`throw` 语句中的表达式必须拥有完整类型。而且如果该表达式是类类型的话，则相应的类必须含有一个可访问的析构函数和一个可访问的拷贝或移动构造函数。如果该表达式是数组类型或函数类型，则表达式将被转换成与之对应的指针类型**。**

异常对象位于有编译器管理的空间中，编译器确保无论调用哪个 `catch` 子句都能访问该空间。异常处理完毕后，异常对象被销毁。

当一个异常被抛出是，沿着调用链的块将依次退出直至找到与异常匹配的处理代码。如果退出某个块，则同时释放块中局部对象使用的内存。因此，抛出一个指向**局部对象的指针**几乎肯定是一种**错误行为**。如果指针所指的对象位于某个块中，而该块在`catch`语句之前就已经退出了，则意味着在执行 `catch` 语句之前局部对象已经被销毁了。

当我们抛出一条表达式时，该表达式的静态编译时类型决定了异常对象的类型。很多情况下程序抛出的表达式类型来自于某个继承体系。如果一条 `throw` 表达式**解引用一个基类指针**，而该指针**实际指向的是派生类对象**，则**抛出的对象将被切掉一部分，只有基类部分被抛出**。

注：抛出指针要求在任何对应处理代码存在的地方，指针所指的对象都必须存在。

2. 捕获异常

`catch` 子句 (catch clause) 中的一场声明 (exception declaration) 看起来像是只包含一个形参的函数形参列表。像在形参列表中一样，如果 `catch` 无须访问抛出的表达式的话，则我们可以忽略捕获形参的名字。

声明的类型决定了处理代码所能捕获的异常类型。这个类型必须是完全类型，它可以是左值引用，不能是右值引用。当进入一个`catch`语句后，入参通过异常对象初始化异常声明中的参数。和函数的参数类似，如果 `catch` 打的参数类型是非引用类型，则该参数是异常对象的一个副本，如果参数是引用类型，则和其他引用参数一样，该参数是异常对象的一个别名。

如果 `catch` 的参数是基类类型，则我们可以使用其派生类类型的异常对象对其进行初始化。此时，如果 `catch` 的参数是非引用类型，则异常对象将被切掉一部分，如果 `catch` 的参数是基类的引用，则该参数将以常规方式绑定到异常对象上。

最后一点需要注意的是，异常声明的静态类型将决定 `catch` 语句所能执行的操作。如果 `catch` 的参数是基类类型，则 `catch` 无法使用派生类特有的任何成员。

Tips: 通常情况下，如果 `catch` 接受的异常与某个继承体系有关，则最好将该 `catch` 的参数定义成引用类型。

2.1 查找匹配的处理代码

在搜寻 `catch` 语句的过程中，我们最终找到的 `catch` 未必是异常的最佳匹配。相反，挑选出来的应该是第一个与异常匹配的 `catch` 语句。因此，越是专门的 `catch` 越应该置于整个 `catch` 列表的前端。

因为 `catch` 语句是按照其出现的顺序逐一匹配的，所以当程序员使用具有继承关系的多个异常时必须对 `catch` 语句的顺序进行组织管理，是的派生类异常的处理代码出现在基类异常的处理代码异常之前。

与实参和形参的匹配规则相比，异常和 `catch` 异常声明的匹配规则受到更多限制。此时，绝大多数类型转换都不被允许，除了一些极细小的差别之外，要求异常的类型和 `catch` 声明的类型时精确匹配的：

- 允许从非常量的类型转换，也就是说一条非常量对象的 `throw` 语句可以匹配一个接受常量引用的 `catch` 语句
- 允许从派生类向基类的类型转换。
- 数组被转换成指向数组（元素）类型的指针，函数被转化成指向该函数类型的指针。

除此之外，包括标准算术类型转换和类类型转换在内，其他所有转换规则都不能在匹配`catch`的过程中使用。

如果在多个`catch`语句的类型之间存在着继承关系，则我们应该把继承链最低端的类（most derived type）放在前面，而将继承链最顶端的类（least derived type）放在后面。

2.2 重新抛出

一个单独的`catch`语句不能完整的处理某个异常。在执行了某些校正操作之后，当前的 `catch` 可能会决定由调用链更上一层的函数接着处理异常。一条`catch`语句通过重新抛出的操作将异常传递给另外一个 `catch` 语句。这里的重新抛出仍然是一条 `throw` 语句，只不过不包含任何表达式: `throw;`

空的 `throw` 语句只能出现在 `catch` 语句或 `catch` 语句直接或间接调用的函数之内。如果在处理代码之外的区域遇到了空 `throw` 语句，编译器将调用 `terminate`。

一个重新抛出语句并不指定新的表达式，而是将当前的异常对象沿着调用链向上传递。

很多时候，`catch`语句会改变其参数内容。如果在改变了参数的内容后`catch`语句重新抛出异常，则只有当`catch`异常声明是引用类型时我们对参数所作的改变才会被保留并继续传播。

2.3 捕获所有异常的处理代码

为了一次性捕获所有异常，我们使用省略号作为异常声明，这样的处理代码称为捕获所有异常的处理代码，形如 `catch(...)`。

`catch(...)` 通常与重新抛出语句一起使用，其中`catch`执行当前局部能完成的工作，随后重新抛出异常。

Tips: 如果 `catch(...)` 与其他几个 `catch` 语句一起出现，则 `catch(...)` 必须在最后的位置。出现在捕获所有异常语句后面的 `catch` 语句将永远不会被匹配。

3. 函数try语句块与构造函数

通常情况下，程序执行的任何时刻都可能发生异常，特别是一场可能发生在处理构造函数初始值的过程中。构造函数在进入其函数体之前首先执行初始值列表。因为在初始值列表抛出异常时构造函数体内的 `try` 语句块还未生效，所以构造函数体内的 `catch` 语句无法处理构造函数初始值列表抛出的异常。

要想处理构造函数初始值抛出的异常，我们必须将构造函数写成**函数try语句块**（function try block）的形式。函数 `try` 语句使得一组 `catch` 语句既能处理构造函数体（或析构造函数体），也能处理构造函数的初始化过程（或析构造函数的析构过程）。

```
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il)
try
    : data(std::make_shared<std::vector<T>>(il))
    {/** ... */}
catch (const std::bad_alloc &e)
{
    handle_out_of_memory(e);
}
```

4. noexcept 异常说明

```
void recoup() noexcept; /** 不会抛出异常 */
void alloc();           /** 可能会抛出异常 */
```

- 对于一个函数来说，`noexcept`说明要么出现在该函数的所有声明语句和定义语句中，要么一次也不出现。该说明应该在函数应该在函数的尾置返回类型之前。
- 我们也可以在函数指针的声明和定义中指定 `noexcept`。
- 在 `typedef` 或类型别名中则不能出现 `noexcept`。
- 在成员函数中，`noexcept` 说明符需要跟在 `const` 及引用限定符之后，而在 `final`、`override` 或虚函数 `=0` 之前。

4.1 违反异常说明

读者需要清楚的一个事实是编译器并不会在编译时检查 `noexcept` 说明。实际上，如果一个函数说明了 `noexcept` 的同时又含有 `throw` 语句或者调用了可能抛出异常的其他函数，编译器将顺利通过，并不会因为这种违反异常说明的情况而报错。

因此可能会出现一种情况：尽管函数说明了它不会抛出异常，但实际上还是抛出了。一旦一个 `noexcept` 函数抛出异常，程序就会调用 `terminate` 以确保遵守不在运行时抛出异常的承诺。

上述过程是执行栈展开未作约定，因此 `noexcept` 可以用在两种情况下：一是我们确认函数不会抛出异常，二是我们根本不知道该如何处理异常。

4.2 noexcept运算符

`noexcept` 说明符接受一个可选实参，该实参必须能转换为 `bool` 类型：如果实参是 `true`，则函数不会抛出异常；如果实参是 `false`，则函数可能抛出异常：

```
void recoup() noexcept(true);    /** 不会抛出异常 */
void alloc() noexcept(false);    /** 可能抛出异常 */
```

`noexcept` 说明符的实参常常与 `noexcept` 运算符混合使用。`noexcept` 运算符是一个一元运算符，它的返回值是一个 `bool` 类型的右值常量表达式，用于表示给定的表达式是否会抛出异常。和 `sizeof` 类似，`noexcept` 也不会求其运算对象的值。

```
noexcept(recoup())    /** 如果recoup不跑出异常则结果为true；否则结果为false */
noexcept(e)           /** 等价于上一句 */
```

我们可以使用 `noexcept` 运算符得到如下的异常说明：

```
void f() noexcept(noexcept(g()));    // f 和 g的异常说明一致
```

如果函数 `g` 承诺了不会抛出异常，则 `f` 也不会抛出异常；如果 `g` 没有异常说明符，或者 `g` 虽然有异常说明符但是允许抛出异常，则 `f` 也可能抛出异常。

`noexcept` 有两层含义：当跟在函数参数列表后面时它是异常说明符；而当作为 `noexcept` 异常说明的 `bool` 实参出现时，它是一个运算符。

4.3 异常说明与指针、虚函数和拷贝控制

函数指针及该指针所指的函数必须具有一致的异常说明。也就是说我们为某个指针做了不抛出异常的声明，则该指针将只能指向不抛出异常的函数。相反，如果我们显式或隐式地说明了指针可能抛出异常，则该指针可以指向任何函数，即使是承诺了不抛出异常的函数也可以。

如果**虚函数**承诺了它**不会抛出异常**，则后续派生出来的**虚函数**也必须做出**同样的承诺**；与之相反如果**基类的虚函数允许抛出异常**，则**派生类的对应函数既可以允许抛出异常，也可以不允许抛出异常**。

当编译器**合成拷贝控制成员**时，同时也**生成一个异常说明**。如果对**所有成员**和**基类的所有操作**都承诺了不会抛出异常，则合成的成员是 `noexcept` 的。如果合成成员调用的**任意一个函数可能抛出异常**，则合成的成员是 `noexcept(false)`。而且如果我们定义了一个析构函数但是没有为它提供异常说明，则编译器将合成一个。合成的异常说明将于假设有编译器为类合成析构函数时所得的异常说明一致。

typeid关键字

作用：查询类型的信息。会返回某个类型的 `std::type_info` 对象。

头文件：< typeinfo >（与其他关键字不同，`typeid` 关键字如不包含该头文件，编译会报错）

用法：`typeid(Type)`

例子：

```
#include <iostream>
#include <string>
#include <typeinfo>

struct Base {}; // 非多态
struct Derived : Base {};

struct Base2 { virtual void foo() {} }; // 多态
```



```

struct Derived2 : Base2 {};

int main() {
    int myint = 50;
    std::string mystr = "string";
    double *mydoubleptr = nullptr;

    std::cout << "myint has type: " << typeid(myint).name() << '\n'
              << "mystr has type: " << typeid(mystr).name() << '\n'
              << "mydoubleptr has type: " << typeid(mydoubleptr).name() << '\n';

    // std::cout << myint 为多态类型的泛左值表达式; 求值它
    const std::type_info& r1 = typeid(std::cout << myint);
    std::cout << '\n' << "std::cout<<myint has type : " << r1.name() << '\n';

    // std::printf() 不是多态类型的泛左值表达式; 不求值
    const std::type_info& r2 = typeid(std::printf("%d\n", myint));
    std::cout << "printf(\"%d\\n\",myint) has type : " << r2.name() << '\n';

    // 非多态左值为静态类型
    Derived d1;
    Base& b1 = d1;
    std::cout << "reference to non-polymorphic base: " << typeid(b1).name() << '\n';

    Derived2 d2;
    Base2& b2 = d2;
    std::cout << "reference to polymorphic base: " << typeid(b2).name() << '\n';

    try {
        // 解引用空指针: 对于非多态表达式 OK
        std::cout << "mydoubleptr points to " << typeid(*mydoubleptr).name() << '\n';
        // 解引用空指针: 对多态左值不 OK
        Derived2* bad_ptr = nullptr;
        std::cout << "bad_ptr points to... ";
        std::cout << typeid(*bad_ptr).name() << '\n';
    } catch (const std::bad_typeid& e) {
        std::cout << " caught " << e.what() << '\n';
    }
}

```

std::tuple介绍

```

#include <tuple>
#include <map>
#include <iostream>
#include <string>
#include <stdexcept>

std::tuple<double, char, std::string> get_student(int id)
{
    static const std::map<int, std::tuple<double, char, std::string>> mapStudentInfo
    {

```

```

        {0, std::make_tuple(3.8, 'A', "Lisa Simpson")},
        {1, std::make_tuple(2.9, 'C', "Milhouse Van Houten")},
        {2, std::make_tuple(1.7, 'D', "Ralph Wiggum")},
    };

    return mapStudentInfo.at(id);
}

int main()
{
    auto student22 = get_student(3);
    auto student0 = get_student(0);
    std::cout << "ID: 0, "
        << "GPA: " << std::get<0>(student0) << ", "
        << "grade: " << std::get<1>(student0) << ", "
        << "name: " << std::get<2>(student0) << '\n';

    double gpa1;
    char grade1;
    std::string name1;
    std::tie(gpa1, grade1, name1) = get_student(1);
    std::cout << "ID: 1, "
        << "GPA: " << gpa1 << ", "
        << "grade: " << grade1 << ", "
        << "name: " << name1 << '\n';
    system("pause");
}

```

std::chrono介绍

1. std::chrono::duration_cast

度量函数的执行时间:

```

#include <iostream>
#include <chrono>
#include <ratio>
#include <thread>

void f()
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

int main()
{
    auto t1 = std::chrono::high_resolution_clock::now();
    f();
    auto t2 = std::chrono::high_resolution_clock::now();

    // 整数时长: 要求 duration_cast
    auto int_ms = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1);
}

```

```
// 小数时长: 不要求 duration_cast
std::chrono::duration<double, std::milli> fp_ms = t2 - t1;

std::cout << "f() took " << fp_ms.count() << " ms, "
          << "or " << int_ms.count() << " whole milliseconds\n";
}
```

std::underlying_type介绍

每个枚举类型都拥有底层类型, 它可以是

1. 显式指定 (有作用域和无作用域枚举均可)
2. 省略, 该情况下对于有作用域枚举是 int, 或 (对于无作用域枚举) 是足以表示枚举所有值的实现定义的整数类型

为了获取枚举类型的底层类型, 我们使用 `std::underlying_type`:

```
#include <iostream>
#include <type_traits>
#include <typeinfo>

enum e1 {};
enum class e2 : int {};

int main()
{
    bool e1_type = std::is_same< unsigned, typename std::underlying_type<e1>::type
>::value;
    bool e2_type = std::is_same< int , typename std::underlying_type<e2>::type >::value;

    std::cout << "underlying type for 'e1' is "
              << (e1_type ? "unsigned" : "non-unsigned") << '\n'
              << "underlying type for 'e2' is "
              << (e2_type ? "int" : "non-int") << '\n';

    std::cout << typeid(std::underlying_type_t<e1>).name() << '\n';
    std::cout << typeid(std::underlying_type_t<e2>).name() << '\n';
    system("pause");
}
```

格式化字符串示例

```
#include <stdio.h>

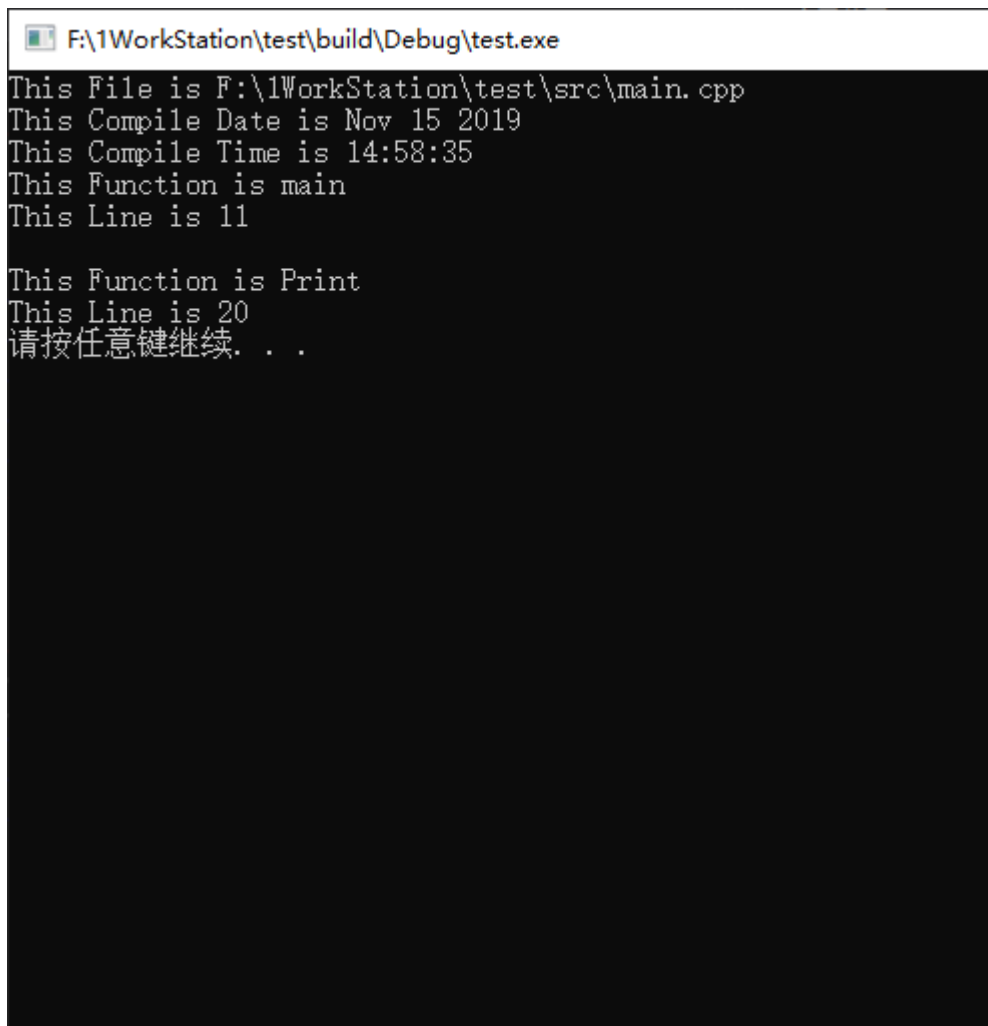
int main(void)
{
    printf("Strings:\n");
    const char* s = "Hello";
    printf("\t.%.10s.\n\t.%.10s.\n\t.*s.\n", s, s, 10, s);

    printf("Characters:\t%c %%\n", 65);
}
```

```
F:\1WorkStation\test\build\Debug\test.exe
Strings:
    .      Hello.
    .Hello
    .      Hello.
Characters:  A %
Integers
Decimal:      1 2 000003 0  +4 4294967295
Hexadecimal:  5 a A 0x6
Octal:  12 012 04
Floating point
Rounding:      1.500000 2 1.300000000000000004440892098500626
Padding:      01.50 1.50  1.50
Scientific:      1.500000E+00 1.500000e+00
Hexadecimal:      0x1.8000000000000p+0 0X1.8000000000000P+0
```

```
{
    std::cout << std::endl;
    std::cout << "This Function is " << __func__ << std::endl;
    std::cout << "This Line is " << __LINE__ << std::endl;
}
```

输出:



```
F:\1WorkStation\test\build\Debug\test.exe
This File is F:\1WorkStation\test\src\main.cpp
This Compile Date is Nov 15 2019
This Compile Time is 14:58:35
This Function is main
This Line is 11

This Function is Print
This Line is 20
请按任意键继续. . .
```

多线程编程总结

版本号	作者	修改摘要	时间
V1.0	李建聪	添加函数的可重入性判断(线程安全性)小节	2019-08-02 10:22:57

函数的可重入性判断(线程安全性)

在多线程编程下，一个函数可能同时被两个线程调用，如果该函数使用了全局变量或者类成员变量或函数内的static变量则会导致问题，所以在新版本中**我们的注释中需要引入可重入性的选项**。现给出如下规则判断一个函数是否可重入：

1. 函数中使用了非函数内定义的变量，如**全局变量**，**类成员变量**，**extern声明的变量**，函数为**不可重入**。
2. 函数中定义了**static变量**，该函数为**不可重入**。

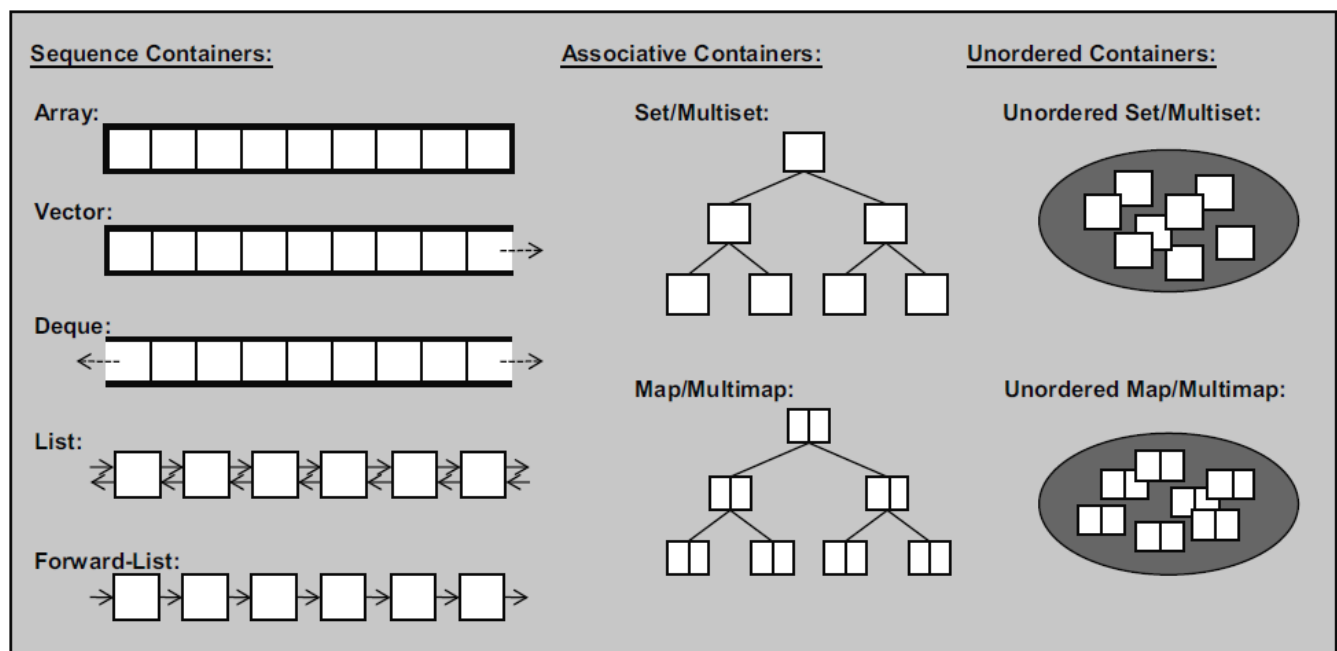
- 3. 调用函数时，入参为**指针**或者**引用**的入参使用**全局变量**，**类成员变量**，**extern声明的变量**，函数为**不可重入**。
- 4. 一个函数调用了**不可重入的函数**，该函数也为**不可重入**。
- 5. **不包含**以上情景的函数，为**可重入函数**(即线程安全)。

设计模式介绍

编程名词与概念解释

版本号	作者	修改摘要	时间
V1.0	李建聪	创建	2019-10-08 08:47:56
V1.1	李建聪	添加 容器类别 小节	2019-10-08 08:48:47
V1.2	李建聪	添加 各种容器使用时机 小节	2019-10-08 08:49:27
V1.3	李建聪	添加 严格弱序 (strict weak ordering) 小节	2019-10-08 08:49:52
V1.4	李建聪	添加 迭代器种类 小节	2019-10-08 08:50:17

容器类别



1. 序列式容器 (Sequence container)

这是一种有序(ordered)集合，其内每个元素均有确凿的位置----取决于插入时机和地点，与元素值无关。如果你以追加方式对一个集和置入6个元素，他们的排列次序将和置入次序一致。STL提供了5个定义好的序列式容器：array、vector、deque、list和forward_list。

2. 关联式容器(Associative container)

这是一种已排序(sorted)集合，元素位置取决于其value(或key---如果元素是个key/value pair)和给定的某个排序准则。如果将六个元素置入这样的集合中，他们的值将决定他们的次序，和插入次序无关。STL提供了4个关联式容器：set、multiset、map和multimap。

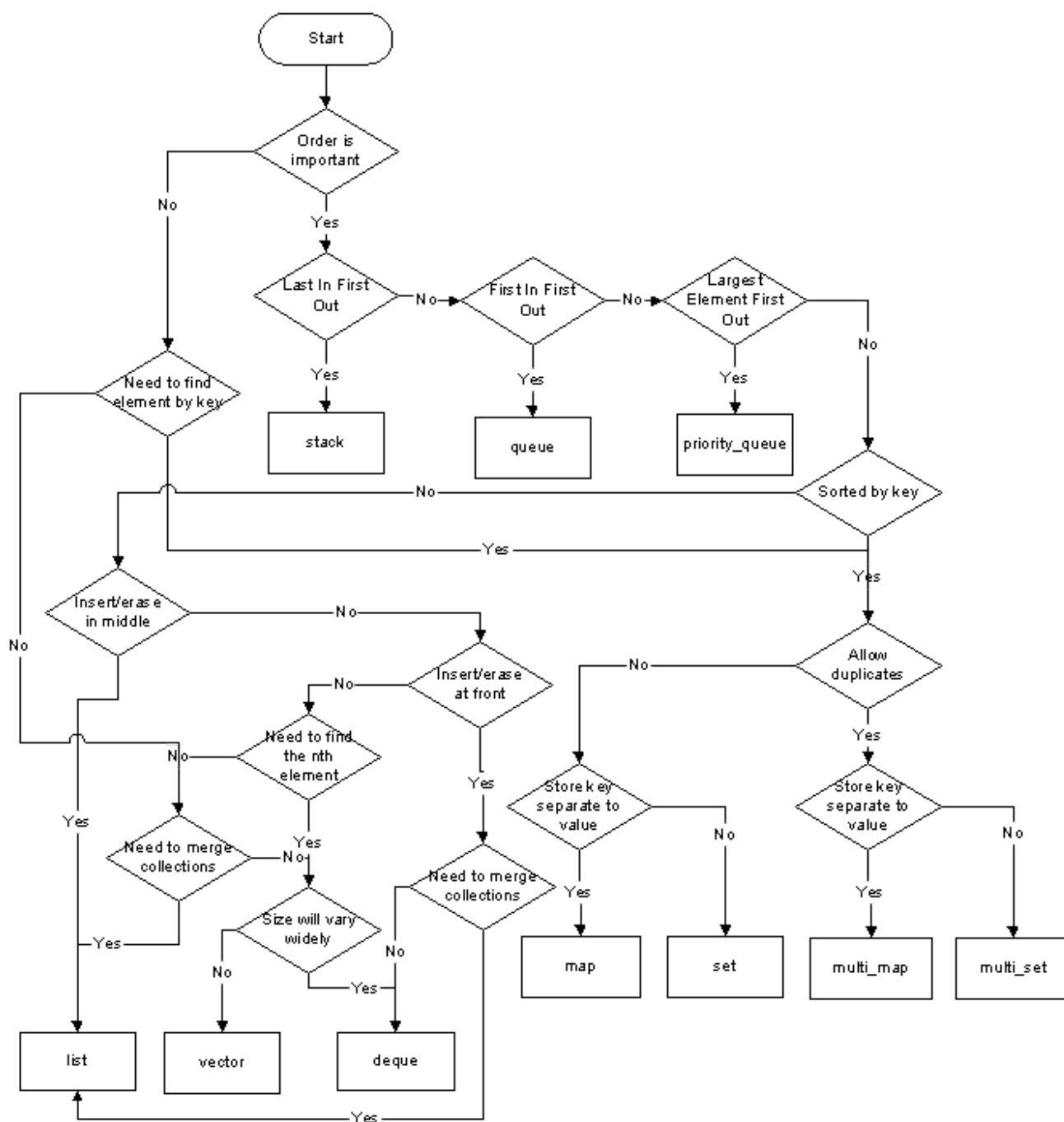
3. 无序容器 (Unordered (associative) container)

这是一种无序集合(unordered collection), 其内每个元素的每个位置无关紧要，唯一重要的是某特定元素是否位于此集合内。元素值或其安插顺序，都不影响元素的位置，而且元素的位置有可能在容器生命周期中被改变。如果你放6个元素到这种集合内，它们的次序不明确，并且可能随时间而改变。STL内含4个预定义的无序容器：

unordered_set、unordered_multiset、unordered_map和unordered_multimap。

- **Sequence**容器通常被实现为array或linked list
- **Associative**容器通常被实现为binary tree
- **Unordered**容器通常被实现为hash table

各种容器使用时机



- 默认情况下应该使用vector。Vector的内部构造最简单，并允许随机访问，所以数据的访问十分方便灵活，数据的处理也够快。
- 如果经常要在序列头部和尾部安插和一处元素，应该采用deque。如果你希望元素被移除时，容器能够自动缩减内部用量，那么也该使用deque。此外，由于vector通常采用一个内存区块来存放元素，而deque采用多个区块，所以后者可内含更多元素。
- 如果需要经常在容器中执行元素安插、移除和移动，可考虑使用list。List提供特殊的成员函数，可在常量时间内将元素从A容器转移到B容器。但由于list不支持随机访问，所以如果只知道list的头部却要造访list的中端元素，效能会大打折扣。和所有“以节点为基础”的容器相似，只要元素仍是容器的一部分，list就不会令只想那些元素的迭代器失效。Vector则不然，一旦超过其容量，它的所有iterator、pointer和reference失效。至于deque，当它的大小改变，所有iterator、pointer和reference都会失效。
- 如果你要的容器对异常处理使得“每次操作若不成功便无任何作用”，那么应该选用list(但是不调用其assignment操作符和sort()，而且如果元素比较过程中会抛出异常，就不要调用merge()、remove()、remove_if()和

unique(), 或选用associative/unordered容器 (但不调用多元素安插动作, 而且如果比较准则的复制/赋值动作可能抛出异常, 就不要调用swap()或erase()))。

- 如果你经常需要根据某个准则查找元素, 应当使用“依据该准则进行hash”的unordered set 或multiset。然而, hash容器内是无序的, 所以如果你必须以来元素的次序(order),应该使用set或multiset, 他们根据查找准则对元素排序。
- 如果想处理key/value pair, 请采用unordered (multi)map。如果元素次序很重要, 可采用(multi)map。
- 如果需要关联式数组(associative array), 应采用unordered map。如果元素次序很重要, 可采用map。
- 如果需要字典结构, 应采用unordered multimap。如果元素次序很重要, 可采用multimap。

	Array	Vector	Deque	List	Forward List	Associative Containers	Unordered Containers
Available since	TR1	C++98	C++98	C++98	C++11	C++98	TR1
Typical internal data structure	Static array	Dynamic array	Array of arrays	Doubly linked list	Singly linked list	Binary tree	Hash table
Element type	Value	Value	Value	Value	Value	Set: value Map: key/value	Set: value Map: key/value
Duplicates allowed	Yes	Yes	Yes	Yes	Yes	Only multiset or multimap	Only multiset or multimap
Iterator category	Random access	Random access	Random access	Bidirectional	Forward	Bidirectional (element/key constant)	Forward (element/key constant)
Growing/shrinking	Never	At one end	At both ends	Everywhere	Everywhere	Everywhere	Everywhere
Random access available	Yes	Yes	Yes	No	No	No	Almost
Search/find elements	Slow	Slow	Slow	Very slow	Very slow	Fast	Very fast
Inserting/removing invalidates iterators	—	On reallocation	Always	Never	Never	Never	On rehashing
Inserting/removing references, pointers	—	On reallocation	Always	Never	Never	Never	Never
Allows memory reservation	—	Yes	No	—	—	—	Yes (buckets)
Frees memory for removed elements	—	Only with shrink_to_fit()	Sometimes	Always	Always	Always	Sometimes
Transaction safe (success or no effect)	No	Push/pop at the end	Push/pop at the beginning and the end	All insertions and all erasures	All insertions and all erasures	Single-element insertions and all erasures if comparing doesn't throw	Single-element insertions and all erasures if hashing and comparing don't throw

严格弱序 (strict weak ordering)

关联式容器 (set、multiset、map和multimap) 的排序准则的定义, 和std::sort的排序准则定义必须遵守严格弱序, 详细描述见官方解释([strict weak ordering.pdf](#))。

严格弱序的定义:

简单的来说就是**a<b**返回true, **a=b**和**a>b**返回false。

详细定义:

1. 必须是**非对称的** (antisymmetric) 。

对 `operator<` 而言, 如果 $x < y$ 为true, 则 $y < x$ 为false。

对判断式(predicate) `op()` 而言, 如果 $op(x, y)$ 为true, 则 $op(y, x)$ 为false。

2. 必须是**可传递的** (transitive) 。

对 `operator<` 而言, 如果 $x < y$ 为true且 $y < z$ 为true, 则 $x < z$ 为true。

对判断式(predicate) `op()` 而言, 如果`op(x, y)`为true且`op(y, z)`为true, 则`op(x, z)`为true。

3. 必须是**非自反的** (irreflexive)

对 `operator<` 而言, `x < x` 永远是false

对判断式(predicate) `op()` 而言, `op(x, x)`永远是false。

4. 必须有**等效传递性** (transitivity of equivalence)

对 `operator<` 而言, 假如 `!(a<b) && !(b<a)` 为true且 `!(b<c) && !(c<b)` 为 true 那么 `!(a<c) && !(c<a)` 也为true.

对判断式(predicate) `op()` 而言, 假如 `op(a,b)`, `op(b,a)`, `op(b,c)`, 和`op(c,b)` 都为 false, 那么`op(a,c)` and `op(c,a)` 也为false.

```
// 一个定义std::set<struct>的例子
#include <set>
#include <iostream>

struct ORDERING_EXAMPLE
{
    int x;
    int y;
    int z;

    /// 重载遵循严格弱序的运算符<
    bool operator < (const ORDERING_EXAMPLE& OtherStruct) const
    {
        if (this->x < OtherStruct.x)
            return true;
        if (OtherStruct.x < this->x)
            return false;

        // x == x则比较y
        if (this->y < OtherStruct.y)
            return true;
        if (OtherStruct.y < this->y)
            return false;

        // y == y则比较z
        if (this->z < OtherStruct.z)
            return true;

        return false;
    }
};

int main()
{
    std::set<ORDERING_EXAMPLE> setOrderingExample;

    ORDERING_EXAMPLE stOrderingExample0 = { 0, 0, 0 };
    ORDERING_EXAMPLE stOrderingExample1 = { 0, 1, 2 };
```

```
ORDERING_EXAMPLE stOrderingExample2 = { 0, 1, 3 };
ORDERING_EXAMPLE stOrderingExample3 = { 0, 1, 3 };

setOrderingExample.insert(stOrderingExample0);
setOrderingExample.insert(stOrderingExample1);
setOrderingExample.insert(stOrderingExample2);
setOrderingExample.insert(stOrderingExample3);

return 0;
}
```

迭代器种类

根据能力的不同，迭代器被划分为物种不同类别。STL预先定义好的所有容器，其迭代器均属于一下三种分类：

1. **前向迭代器 (Forward iterator)** 只能够以累加操作符 (increment operator) 向前迭代。Class `forward_list`的迭代器就属此类。其他容器如`unordered_set`、`unordered_multiset`、`unordered_map`和`unordered_multimap`也都至少是此类别（但标准库其实为他们提供的是双向迭代器）
2. **双向迭代器 (Bidirectional iterator)** 顾名思义它可以双向行进：以递增 (increment) 运算前进或以递减 (decrement)运算后退。`list`、`set`、`multiset`、`map`和`multimap`提供的迭代器都属此类。
3. **随机访问迭代器 (Random-access iterator)** 它不但具备双向迭代器的所有属性，还具备随机访问能力。更明确的说，他们提供了迭代器算数运算的必要操作符（和寻常指针的算数运算完全对应）。你可以对迭代器增加或减少一个偏移量、计算两迭代器间的距离，或使用`<`和`>`之类的relational（相对关系）操作符进行比较。`vector`、`deque`、`array`和`string`提供的迭代器都属此类。

除此之外，STL还定义了两个类别：

- **输入型迭代器 (Input iterator)** 向前迭代时能够读取/处理value。Input stream迭代器就是这样的例子。
- **输出型迭代器 (Output iterator)** 向前迭代时能够涂写value。Inserter和output stream迭代器都属此类。

经验总结与代码示例

版本号	作者	修改摘要	时间
V1.0	李建聪	创建	2019-10-08 10:23:44
V1.1	李建聪	添加小节 如何求得一个数组的长度	2019-10-08 10:44:54
V1.2	李建聪	添加小节 如何使用一条SQL语句， ...	2019-10-12 09:12:35
V1.3	李建聪	修改小节题目到 数据库插入时重复主键问题 ，并增添内容。	2019-10-30 11:31:27
V1.4	李建聪	增加小节 使用empty()来代替size()检查容器大小是否为0	2019-10-30 11:37:47
V1.5	李建聪	增加小节 使用reserve来避免不必要的重新分配	2019-10-30 11:40:13
V1.6	李建聪	增加小节 避免使用vector< bool >	2019-10-30 11:40:14
V1.7	李建聪	增加小节 使用using代替typedef	2019-10-30 11:40:14
V1.8	李建聪	增加小节 C++中struct和class的区别	2019-10-30 11:40:15
V1.9	李建聪	增加小节 虚析构函数问题	2019-10-30 11:40:15
V2.0	李建聪	增加小节 获取未知结构体成员信息	2019-11-05 13:35:32
V2.1	李建聪	增加小节 MySQL列出不同的值	2019-11-05 13:36:05
V2.2	李建聪	增加小节 对称数组实现itoa	2019-11-15 14:38:26
V2.3	李建聪	增加小节 产生随机数	2019-11-15 14:39:24

如何求得一个数组的长度

数组长度概念：即一个原生数组中可包含的元素数量。如int a[5]; 数组a的长度为5.

推荐使用Lee::ArraySize替代ARRAY_SIZE(x)。接下来介绍Lee::ArraySize为什么优于ARRAY_SIZE(x)。

以下是两种数组长度求解的实现。

```

/** 惯常使用的宏定义类型的数组长度 */
#define ARRAY_SIZE(x)    sizeof(x)/sizeof((x)[0])

/** 使用了C++11中类型推导的特性来求得数组长度 */
template <class T, std::size_t N>
constexpr inline std::size_t ArraySize(T (&)[N]) noexcept
{
    return N;
}

```

数组长度的求得我们通常使用宏定义ARRAY_SIZE(x)，但现在更加推荐使用使用类型推导的语法实现的函数ArraySize()。

宏定义ARRAY_SIZE(x)的缺点：宏定义无法在编译期间是被传入的是数组还是指针。尤其是当一个数组被传入到函数中数组参数会自动降级为指针的情况，给程序带来了隐藏的错误。

函数ArraySize()的优点：由于采用了自动类型推导，当一个指针被传入时在编译器期间就会产生错误，从而大大减少了调试和查找错误的难度。相较老式的ARRAY_SIZE(x)更加容易使用且出错概率更小。

函数ArraySize()的缺点：只可以传入原生数组，容器（如std::array, std::vector和std::string）的大小还是使用容器自带的.size()函数。当然ARRAY_SIZE(x)也无法提供该功能，所以仍推荐使用ArraySize()。

完整例子：

```
#include <iostream>
#include <cassert>
#include <vector>

template <class T, std::size_t N>
constexpr inline std::size_t ArraySize(T (&)[N]) noexcept
{
    return N;
}

#define ARRAY_SIZE(x)    sizeof(x)/sizeof((x)[0])

std::size_t ErrorArraySize(int a[])
{
    /// 因为数组传入函数时降级为指针所以这里会固定返回1而不是数组的长度
    return ARRAY_SIZE(a);    ///< 返回错误值
}

std::size_t CorrectArraySize(int a[])
{
    /** 当数组被传入函数中降级为指针时，编译错误 */
    // auto CompileError = ArraySize(a);
    return 0;
}

int main()
{
    int a0[5];
    int a1[5] = {0};
    int a2[5] = {1, 2, 3, 4, 5};
    char a3[5];
    double a4[5];
    int* p = new int(5);

    assert(5 == ArraySize(a0));
    assert(5 == ArraySize(a1));
    assert(5 == ArraySize(a2));
    assert(5 == ArraySize(a3));
    assert(5 == ArraySize(a4));

    /** 编译可以通过，但返回的值是错误的 */
    auto ErrorSize0 = ARRAY_SIZE(p);
    /** 传入指针时编译不通过 */
    // assert(1 == ArraySize(p));
```

```

/** 编译通过, 宏定义返回错误的值 */
assert(1 == ErrorArraySize(a0));
/** 编译无法通过 */
// CorrectArraySize(a0);

std::vector<int> vec = {1, 2, 3, 4, 5, 6, 7};
/** 编译可以通过, 但返回的值是错误的 */
auto ErrorSize1 = ARRAY_SIZE(vec);    // 返回一个错误的值
/** 编译错误 */
// auto CompileError = ArraySize(vec);

return 0;
}

```

数据库插入时重复主键问题

解决办法:

- 方法一: 当数据存在时, 更新该条数据
- 方法二: 当数据存在时, 不插入数据
- 方法三: 当数据存在时, 删除原有数据后插入新的数据

1. 更新该条数据

使用关键字**INSERT ... ON DUPLICATE KEY UPDATE**。(注意这几个单词是连在一起的, 少一个关键字都会报错)

[ON DUPLICATE KEY UPDATE官网解释](#)

用法:

```

# 在TableName表(id为主键)中插入一条id = 3的数据, 当主键重复 (即该条数据已经存在时, 更新该条数据, ID自加)
INSERT INTO TableName(id) VALUES(3) ON DUPLICATE KEY UPDATE id = id + 1;

```

具体例子详见:

smdV4.0/server/DialDao.cpp/DialDao::InitServerAndAgentStatus()

```

/** 当dialagentstatus表中没有该条数据时, 插入一条新数据, 如果有则更新该条数据 */
BOOL DialDao::InitServerAndAgentStatus()
{
    CHAR    acSQL[1024] = { 0 };
    gos_snprintf(acSQL, sizeof(acSQL),
        "INSERT INTO dialagentstatus(AgentID, LinkStateUpdateTime, "
        "LinkState0, LinkState1, AgentState, AgentStateUpdateTime)"
        " VALUES(%d, %d, '%s', '%s', %d, UNIX_TIMESTAMP())"
        " ON DUPLICATE KEY UPDATE AgentState = %d, "
        "AgentStateUpdateTime = UNIX_TIMESTAMP()",
        dialagentstatus::SERVER_AGENT_ID,
        dialagentstatus::INVAIL_UPDATE_TIME,
        dialagentstatus::INVAIL_LINK_STATE.c_str(),
        dialagentstatus::INVAIL_LINK_STATE.c_str(),
        dialagentstatus::ON_LINE,

```

```

        dialagentstatus::ON_LINE);

    INT32 iAffectedRow = m_pDcco->Update(acSQL);
    if (iAffectedRow <= 0)
    {
        return FALSE;
    }
    return TRUE;
}

```

2. 不插入数据

使用关键字: `INSERT IGNORE INTO ...`

以下为语法示例:

```

# 有重复主键时直接跳过, 没有重复主键时直接插入.
INSERT IGNORE INTO dialagentgroup(AgentGroupID, AgentID, NumberGroupID, DialPriority)
VALUES(11111, 222222, 33333333, 4444444);

```

3. 删除原有数据后插入新的数据

使用关键字: `REPLACE INTO`

以下时语法示例:

```

REPLACE INTO dialagentgroup(AgentGroupID, AgentID, NumberGroupID, DialPriority)
VALUES(11111, 222222, 33333333, 4444444);

```

使用empty()来代替size()检查容器大小是否为0

对于任意容器c, 写下

```

if (c.size() == 0)...

```

本质上等价于写下

```

if (c.empty())...

```

这就是例子。你可能会奇怪为什么一个构造会比另一个好，特别是事实上empty的典型实现是一个返回size是否返回0的内联函数。你应该首选empty的构造，而且理由很简单：**对于所有的标准容器，empty是一个常数时间的操作，但对于一些list实现，size花费线性时间。**因为std::list的数据结构为不遍历所有元素不知道整个容器元素个数，所以在使用size()这个成员函数的时间根据list容器的大小而定，时间复杂度为O(n)。而使用empty()则始终为常数时间。

尽量使用区间操作成员函数代替它们的单元操作成员函数

给定两个vector, v1和v2, 使v1的内容和v2的后半部分一样的最简单方式是什么?

```
v1.assign(v2.begin() + v2.size() / 2, v2.end());
```

VS

```
vector<Widget> v1, v2; // 假设v1和v2是Widget的vector
v1.clear();
for (vector<Widget>::const_iterator ci = v2.begin() + v2.size() / 2; ci != v2.end(); ++ci)
    v1.push_back(*ci);
```

写一行代码的错误几率肯定比其他的要小且速度最快。如容器支持区间赋值函数，则应该使用区间赋值函数代替单个赋值函数。

使用reserve来避免不必要的重新分配

关于STL容器只要不超过它们的最大大小，它们就可以自动增长到足以容纳你放进去的数据。（要知道这个最大值，只要调用名叫max_size的成员函数。）对于vector和string，只需要更多空间，这个操作有四个部分：

1. 分配新的内存块，它有容器目前容量的几倍。在大部分实现中，vector和string的容量每次以2为因数增长。也就是说，当容器必须扩展时，它们的容量每次翻倍。
2. 把所有元素从容器的旧内存拷贝到它的新内存。
3. 销毁旧内存中的对象。
4. 回收旧内存。

执行了所有的分配，回收，拷贝和析构，你就应该知道那些步骤都很昂贵。当然，你不会想要比必须的更为频繁地执行它们。如果这没有给你打击，那么也许当你想到每次这些步骤发生时，**所有指向vector或string中的迭代器、指针和引用都会失效**时，它会给你打击的。这意味着简单地把一个元素插入vector或string的动作也**可能因为需要更新其他使用了指向vector或string中的迭代器、指针或引用的数据结构而膨胀**。**reserve成员函数允许你最小化必须进行的重新分配的次数，因而可以避免真分配的开销和迭代器/指针/引用失效**。但在我解释reserve为什么可以那么做之前，让我简要介绍有时候令人困惑的四个相关成员函数。在标准容器中，只有vector和string提供了所有这些函数。

- size(): 告诉你**容器中有多少元素**。它没有告诉你容器为它容纳的元素分配了多少内存。
- capacity(): 告诉你**容器在它已经分配的内存中可以容纳多少元素**。那是容器在那块内存中总共可以容纳多少元素，而不是还可以容纳多少元素。如果你想知道一个vector或string中有多少没有被占用的内存，你必须从capacity()中减去size()。如果size和capacity返回同样的值，容器中就没有剩余空间了，而下次插入（通过insert或push_back等）会引发上面的重新分配步骤。
- resize(Container::size_type n): **强制把容器改为容纳n个元素**。调用resize之后，size将会返回n。**如果n小于当前大小，容器尾部的元素会被销毁。如果n大于当前大小，新默认构造的元素会添加到容器尾部。如果n大于当前容量，在元素加入之前会发生重新分配。**
- reserve(Container::size_type n): **强制容器把它的容量改为至少n，提供的n不小于当前大小**。这一般强迫进行一次重新分配，因为容量需要增加。（如果n小于当前容量，vector忽略它，这个调用什么都不做，string可能把它的容量减少为size()和n中大的数，但string的大小没有改变。

这个简介明确表示了只要有元素需要插入而且容器的容量不足时就会发生重新分配（包括它们维护的原始内存分配和回收，对象的拷贝和析构和迭代器、指针和引用的失效）。所以，避免重新分配的关键是使用reserve尽快把容器的容量设置为足够大，**最好在容器被构造之后立刻进行**。

例如，假定你想建立一个容纳1-1000值的vector。没有使用reserve，你可以像这样做：


```
vector<int> v;  
for (int i = 1; i <= 1000; ++i) v.push_back(i);
```

在大多数STL实现中，这段代码在循环过程中将会导致**2到10次重新分配**。（10这个数没什么奇怪的。记住vector在重新分配发生时一般把容量翻倍，而1000约等于 2^{10} 。）

把代码改为使用reserve，我们得到这个：

```
vector<int> v;  
v.reserve(1000);  
for (int i = 1; i <= 1000; ++i) v.push_back(i);
```

这在循环中不会发生重新分配。在大小和容量之间的关系让我们可以预言什么时候插入将引起vector或string执行重新分配，而且，可以预言什么时候插入会使指向容器中的迭代器、指针和引用失效

避免使用vector< bool >

vector是vector在传入bool值时的一个特化容器，它区别于正常的vector。其成员函数和存储方式都与正常的vector略有不同。如没有把握，应避免使用该容器。

以下引用《C++标准库》中的描述

C++标准库针对元素类型为bool的vector<>专门设计了一个特化版本，目的是获取一个优化的vector，使其耗用空间远小于一般的vector实现出来。一般实现版本会为每个bool元素分配至少1 byte空间，而vector< bool >特化版的内部只使用1 bit存放一个元素，空间节省8倍。不过这里有个小麻烦：c++的最小定址值仍是以byte为单位，所以上述的vector特化版必须对reference和iterator做特殊处理

使用using代替typedef

我们都可以使用using或typedef来定义类型别名。如：

```
/// 下面两句话等价。  
typedef unsigned int UINT32;  
using UINT32 = unsigned int;
```

但使用using关键字可以使格式更**简洁更易于理解**。如：

```
/// 定义FP是一个返回值为void，入参为int和const string& 的函数指针。  
typedef void (*FP)(int, const std::string&);  
using FP = void (*)(int, const std::string&);
```

最重要的是：using关键字相较于typedef多了一个使用**模板**的功能。如：

```
/// typedef中不可以使用模板  
template <typename T>  
typedef std::vector<T> v;    ///< 编译错误。  
  
template <typename T>  
using v = std::vector<T>;    ///< 编译通过。
```

C++中struct和class的区别

C++中 `struct` 和 `class` 几乎完全等价，唯一的区别在于默认成员访问权限和默认派生访问权限不一样。（默认访问权限就是关键字: `public`、`private`）

默认成员访问权限:

如果我们使用 `struct` 关键字，则定义在第一个访问说明符之前的成员是`public`；相反如果我们使用 `class` 关键字,则这些成员是`private`的。

```
struct test
{
    int i; ///< 该成员默认的访问权限为public
};

class test
{
    int i; ///< 该成员默认的访问权限为private
};
```

默认派生访问权限:

默认派生运算符由定义派生类所用的关键字来决定。默认情况下，使用`class`关键字定义的派生类时私有继承的；而使用`struct`关键字定义的派生类时公有继承的：

```
class Base {/** ... */};
struct Derived1 : Base {/** ... */};    ///< 默认public继承
class Derived2 : Base {/** ... */};    ///< 默认private继承
```

注意：关于派生的访问权限最好显式的定义而不是依赖于默认的访问权限，提高代码可读性的同时减少出错概率。

虚析构函数问题

引用标准中原文：一条有用的方针，是任何基类的析构函数必须为公开且虚，或受保护且非虚。

虚析构这个概念被设计出来就是为了解决基类指针指向派生类实例的析构问题，当一个基类指针指向派生类实例然后进行`delete`该指针时，只会执行基类析构函数而派生类的析构函数不会被执行，这将导致派生类构造的资源不会被正确释放，造成内存泄漏。如下示例：

```
#include <iostream>

struct Base
{
    Base() { std::cout << "Base Construct!" << std::endl; }
    ///< 该析构函数为错误示例，严禁这样写。
    ~Base() { std::cout << "Base Deconstruct!" << std::endl; }
};

struct Derived : public Base
{
    Derived() { std::cout << "Derived Construct!" << std::endl; }
```

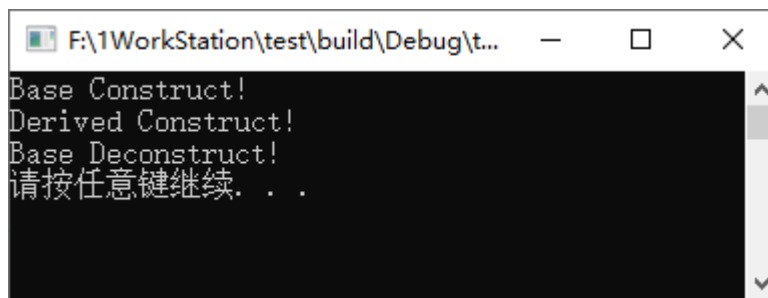
```

    ~Derived() { std::cout << "Derived Deconstruct!" << std::endl; }
};

int main()
{
    {
        /** 使用基类指针指向派生类实例 */
        Base* BasePtr = new Derived;
        delete BasePtr;
    }
    system("pause");
}

```

运行结果：



```

F:\1WorkStation\test\build\Debug\t...
Base Construct!
Derived Construct!
Base Deconstruct!
请按任意键继续. . .

```

可以看到派生类没有被析构，如要解决该问题在基类析构函数处加上**virtual**关键字即可。

```

#include <iostream>

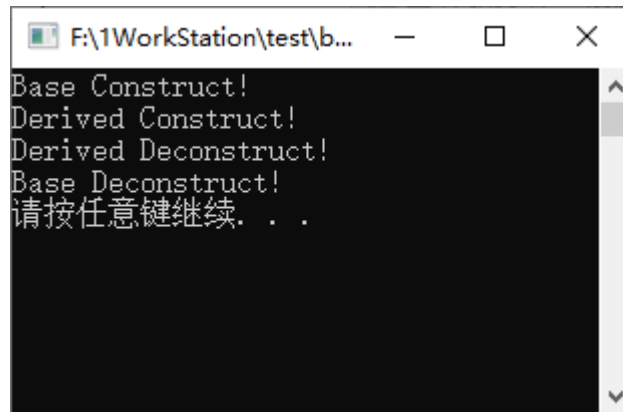
struct Base
{
    Base() { std::cout << "Base Construct!" << std::endl; }
    /** 正确写法： 加上关键字virtual， 后面函数体可写可不写，或者直接使用=default都行。 */
    virtual ~Base() { std::cout << "Base Deconstruct!" << std::endl; }
};

struct Derived : public Base
{
    Derived() { std::cout << "Derived Construct!" << std::endl; }
    ~Derived() { std::cout << "Derived Deconstruct!" << std::endl; }
};

int main()
{
    {
        /** 使用基类指针指向派生类实例 */
        Base* BasePtr = new Derived;
        delete BasePtr;
    }
    system("pause");
}

```

运行结果：



获取未知结构体成员信息

使用定义一个获取成员的模板，使用模板类型推导和offsetof关键字实现。坏处是要对每一个想要获取成员的的结构体，实现一个模板特例化的get_members。

```
#include <cstdint>
#include <typeinfo>
#include <iostream>
#include <vector>

struct S
{
    float a;
    int b;
    const char* c;
};

struct Member
{
    const char* name;
    const std::type_info& type;
    std::size_t offset;
};

template <class T, class M>
M member_type(M T::*) {}

#define MEMBER(T, M) {#M, typeid(member_type(&T::M)), offsetof(T, M) }

template <class T>
const std::vector<Member>& get_members();

template <>
const std::vector<Member>& get_members<S>()
{
    static const std::vector<Member> members
    {
        MEMBER(S, a),
        MEMBER(S, b),
        MEMBER(S, c)
    };
};
```

```

    return members;
}

template <class M, class T> M& get_member_by_index(T& a, size_t index)
{
    return *reinterpret_cast<M*>(reinterpret_cast<char*>(&a) + get_members<T>
().at(index).offset);
}

int main()
{
    for (auto& m : get_members<S>())
        std::cout << m.name << " " << m.type.name() << " " << m.offset << std::endl;

    S s{1.0f, 2, "hello"};
    std::cout << get_member_by_index<float>(s, 0) << std::endl;
    std::cout << get_member_by_index<int>(s, 1) << std::endl;
    std::cout << get_member_by_index<const char*>(s, 2) << std::endl;

    get_member_by_index<float>(s, 0) = 3.0f;
    std::cout << s.a << std::endl;
    system("pause");
}

```

MySQL列出不同的值

关键词: `DISTINCT`

语法:

```
SELECT DISTINCT 列名称 FROM 表名称;
```

对称数组实现itoa

```

#include <iostream>
#include <cassert>
/**
 * @name          itoa
 * @brief         功能等同于标准中的itoa, 用来转换int值到const char*的函数 (支持负数转换)
 *
 * @param         buf      [out]   传出已经转换的字符
 * @param         value    [in]   int值
 *
 * @return
 * @author        Lijiancong, 316, lijiancong@fritt.com.cn
 * @date          2019-11-12 15:44:56
 * @warning       线程不安全
 *
 * @note
 */
template<class BidirIt>
void Itoa_Reverse(BidirIt first, BidirIt last)

```

```

{
    while ((first != last) && (first != --last)) std::iter_swap(first++, last);
}
const char* itoa(char buf[], int value)
{
    /** 对称数组来应对负数情况 */
    static char digits[] = {'9', '8', '7', '6', '5', '4', '3', '2', '1',
                            '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
    static const char* zero = digits + 9; // zero 指向'0'

    int i = value;
    ///< work from INT_MIN ~ INT_MAX
    char* p = buf;
    do
    {
        int lsd = i % 10;
        i /= 10;
        *p++ = zero[lsd];
    } while (i != 0);

    if (value < 0) { *p++ = '-'; }

    *p = '\0';
    Itoa_Reverse(buf, p);
    return p;
}

int main()
{
    char acBuf[16] = { 0 };
    int value = -123456;
    itoa(acBuf, value);
    assert(0 == strcmp(acBuf, "-123456"));

    memset(acBuf, sizeof(acBuf), 0);
    value = 0;
    itoa(acBuf, value);
    assert(0 == strcmp(acBuf, "0"));

    memset(acBuf, sizeof(acBuf), 0);
    value = INT_MAX;
    itoa(acBuf, value);
    assert(0 == strcmp(acBuf, "2147483647"));

    memset(acBuf, sizeof(acBuf), 0);
    value = INT_MIN;
    itoa(acBuf, value);
    assert(0 == strcmp(acBuf, "-2147483648"));

    system("pause");
    return 0;
}

```

产生随机数

```
#include <ctime>
#include <random>
#include <stdlib.h>
#include <cassert>

int main()
{
    srand(time(nullptr));
    /** 获取[50, 100)区间的随机数 */
    for (int i = 0; i < 1000; ++i)
    {
        auto RandNum = (rand() % (100 - 50)) + 50;
        assert(RandNum >= 50 && RandNum < 100);
    }

    /** 获取[50, 100]区间的随机数 */
    for (int i = 0; i < 1000; ++i)
    {
        auto RandNum = (rand() % (100 - 50 + 1)) + 50;
        assert(RandNum >= 50 && RandNum <= 100);
    }

    /** 获取(50, 100]区间的随机数 */
    for (int i = 0; i < 1000; ++i)
    {
        auto RandNum = (rand() % (100 - 50)) + 50 + 1;
        assert(RandNum > 50 && RandNum <= 100);
    }

    /** 获取[0, 1]之间的浮点数 */
    for (int i = 0; i < 1000; ++i)
    {
        auto RandNum = (rand() / static_cast<double>(RAND_MAX));
        assert(RandNum >= 0.00000001 && RandNum <= 1.00000001);
    }

    system("pause");
}
```

读/写csv文件

```
#include <iostream>
#include <string>
#include <vector>
#include <fstream>
#include <sstream>

int main()
{
    std::ofstream outFile("data.csv", std::ios::out);
```

```

outFile << "name" << "," << "age" << "," << "hobby" << std::endl;
outFile << "Mike" << "," << "18" << "," << "paiting" << std::endl;
outFile << "Tom" << "," << "25" << "," << "football" << std::endl;
outFile << "Jack" << "," << "21" << "," << "music" << std::endl;
outFile.close();

// 读文件
std::ifstream inFile("data.csv", std::ios::in);
std::string lineStr;
std::vector<std::vector<std::string>> strArray;
while (getline(inFile, lineStr))
{
    // 打印整行字符串
    std::cout << lineStr << std::endl;
    // 存成二维表的结构
    std::stringstream ss(lineStr);
    std::string str;
    std::vector<std::string> lineArray;
    // 按照逗号分割
    while (getline(ss, str, ','))
        lineArray.push_back(str);
    strArray.push_back(lineArray);
}
system("pause");
return 0;
}

```

SQL语句中DROP、TRUNCATE和DELETE的用法

语法:

```

DROP TABLE 表名称;
TRUNCATE TABLE 表名称;
DELETE FROM 表名称 WHERE 列名称 = 值;

```

DROP、TRUNCATE和DELETE的区别

1. DROP用于删除表：删除内容和定义，释放空间。但跟这张表有关的触发器和存储过程不会被删除。
2. TRUNCATE用于清空表中数据：删除内容、释放空间但保留数据表结构。TRUNCATE不能删除行数据，只能清空表。对于由foreign key约束引用的表，不能使用truncate table，而应使用不带WHERE子句的DELETE语句。由于TRUNCATE TABLE记录在日志中，所以它不能激活触发器。
3. DELETE用于删除表中的特定数据：该操作被记录于事务记录中，便于回滚。

执行速度:

DROP > TRUNCATE > DELETE

DELETE语句是数据库操作语言，这个操作会被记录在事务日志中，事务提交后才会生效，相应的触发器在执行的时候也将会被触发。

TRUNCATE、DROP是数据库定义语言，操作立即生效，无法回滚，也无法触发触发器。