# **绪论：**

参考书籍：《深入应用 C++11 ：代码优化与工程级应用》

参考网站 ：[CSDN](https://blog.csdn.net/luoyayun361/category_7650423.html)、博客园

# **第一章：让程序更简洁**

## **1.1 类型推导**

* auto可以在声明变量的时候根据变量初始值的类型自动为此变量选择匹配的类型，类似的关键字还有decltype。
  + auto的作用就是为了简化变量初始化，如果这个变量有一个很长很长的初始化类型，就可以用auto代替。

int a = 10;  
 auto au\_a = a;//自动类型推断，au\_a为int类型  
 cout << typeid(au\_a).name() << endl;  
// 输出 int

* 注意点
  + 用auto声明的变量必须初始化
  + 函数和模板参数不能被声明为auto
  + 因为auto是一个[占位符](https://so.csdn.net/so/search?q=%E5%8D%A0%E4%BD%8D%E7%AC%A6&spm=1001.2101.3001.7020)，并不是一个他自己的类型，因此不能用于类型转换或其他一些操作，如sizeof和typeid
  + 定义在一个auto序列的变量必须始终推导成同一类型

【参考】

1.[【CSDN auto】](https://blog.csdn.net/lwgkzl/article/details/82110068?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522165347777416781818719547%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=165347777416781818719547&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-82110068-null-null.142^v10^pc_search_result_control_group,157^v12^control&utm_term=auto&spm=1018.2226.3001.4187) 2. [【博客园 auto类型推导】](https://www.cnblogs.com/QG-whz/p/4951177.html)

## **1.2 模板的细节改进**

* 模板的右尖括号
  + C++11之前是不允许两个右尖括号出现的，会被认为是右移操作符，所以需要中间加个空格进行分割，避免发生编译错误。

int main() {  
 std::vector<std::vector<int>> a; // error  
 std::vector<std::vector<int> > b; // ok  
}  
// 但是现在可以了

* 模板的别名
  + C++11引入了using，可以轻松的定义别名，而不是使用繁琐的typedef。

typedef std::vector<std::vector<int>> vvi; // before c++11  
using vvi = std::vector<std::vector<int>>; // c++11  
  
typedef void (\*func)(int, int); //   
using func = void (\*)(int, int); //

* 函数模板的默认模板参数
  + C++11之前只有类模板支持默认模板参数，函数模板是不支持默认模板参数的，C++11后都支持。
  + 类模板的默认模板参数必须从右往左定义，而函数模板则没有这个限制。

template <typename R, typename U=int>  
R func1(U val) {  
 return val;  
}  
   
   
template <typename R=int, typename U>  
R func2(U val) {  
 return val;  
}  
   
   
int main() {  
 cout << func1<int, double>(99.9) << endl; // 99  
 cout << func1<double, double>(99.9) << endl; // 99.9  
 cout << func1<double>(99.9) << endl; // 99.9  
 cout << func1<int>(99.9) << endl; // 99  
 cout << func2<int, double>(99.9) << endl; // 99  
 cout << func1<double, double>(99.9) << endl; // 99.9  
 cout << func2<double>(99.9) << endl; // 99.9  
 cout << func2<int>(99.9) << endl; // 99  
 return 0;  
}

## **1.3 列表初始化**

* 统一的初始化方法
  + 在C++11中初始化列表被适用性被放大，可以作用于任何类型对象的初始化。

class Foo  
{  
public:  
Foo(int) {}  
private:  
Foo(const Foo &);  
};  
   
int \_tmain(int argc, \_TCHAR\* argv[])  
{  
Foo a1(123); //调用Foo(int)构造函数初始化  
Foo a2 = 123; //error Foo的拷贝构造函数声明为私有的，该处的初始化方式是隐式调用Foo(int)构造函数生成一个临时的匿名对象，再调用拷贝构造函数完成初始化  
   
Foo a3 = { 123 }; //列表初始化  
Foo a4 { 123 }; //列表初始化  
   
int a5 = { 3 };  
int a6 { 3 };  
return 0;  
}  
  
//===================  
int\* a = new int { 3 };  
double b = double{ 12.12 };  
int \* arr = new int[] {1, 2, 3};

* + 在C++11中，列表初始化不仅能完成对普通类型的初始化，还能完成对类的列表初始化，需要注意的是a3 a4都是列表初始化，私有的拷贝并不影响它，仅调用类的构造函数而不需要拷贝构造函数，a4,a6的写法是C++98/03所不具备的，是C++11新增的写法。
  + 列表初始化方法也适用于用new操作等圆括号进行初始化的地方
  + 可以使用列表初始化方法对堆中分配的内存的数组进行初始化
* 列表初始化的一些细节
  + 如何区分一个类(class struct union)是否可以使用列表初始化来完成初始化工作呢？关键问题看这个类是否是一个聚合体（aggregate)，
    - 无用户自定义构造函数。
    - 无私有或者受保护的非静态数据成员
    - 无基类
    - 无虚函数
    - 无{}和=直接初始化的非静态数据成员。下面我们逐个对上述进行分析。
* 初始化列表
  + 略
* 列表初始化防止类型收窄
  + C++11的列表初始化还有一个额外的功能就是可以防止类型收窄，也就是隐式类型转换，将范围大的转换为范围小的表示。
  + 在C++11中，使用列表初始化的类型收窄编译将会报错

int a = 1.1; //OK  
int b{ 1.1 }; //error  
   
float f1 = 1e40; //OK  
float f2{ 1e40 }; //error  
   
const int x = 1024, y = 1;  
char c = x; //OK  
char d{ x };//error  
char e = y;//error  
char f{ y };//error

## **1.4 基于范围的for循环**

for (auto val : numbers)  
{  
 cout << val << endl;  
}

* 略

## **1.5 std::function & bind绑定器**

* 可调用对象
  + 函数指针
  + 具有operator()成员函数的类对象（仿函数）
  + 可被转换为函数指针的类对象
  + 类成员（函数）指针
  + C++11通过提供std::function和std::bind统一了可调用对象的各种操作。
* 可调用对象包装器：std::function
  + 头文件：<functional>
  + std::function是可调用对象的包装器。它是一个类模板，可以容纳除了类成员（函数）指针之外的所有可调用对象。
  + 通过指定它的模板参数，它可以用统一的方式处理函数，函数对象，函数指针，并允许保存和延迟执行它们。

#include <iostream>  
#include <functional>  
using namespace std;  
   
void func(void)   
{  
cout << \_\_FUNCTION\_\_ << "(" << ") ->: " << endl;  
}  
   
class Foo  
{  
public:  
static int foo\_func(int a) {  
cout << \_\_FUNCTION\_\_ << "(" << a << ") ->: ";  
return a;  
}  
};  
   
class Bar  
{  
public:  
int operator()(int a) {  
cout << \_\_FUNCTION\_\_ << "(" << a << ") ->: ";  
return a;  
}  
};  
   
int main() {  
//绑定一个普通函数  
std::function<void(void)> fr1 = func;  
fr1();  
   
//绑定一个类的静态成员函数  
std::function<int(int)> fr2 = Foo::foo\_func;  
cout << fr2(123) << endl;  
   
//绑定一个仿函数  
Bar bar;  
fr2 = bar;  
cout << fr2(123) << endl;  
   
return 0;  
}

* + std::function的使用方法：我们给std::function填入合适的函数签名（即一个函数类型，只需要包括返回值和参数表）之后，它就变成了一个可以容纳所有这一类调用方式的“函数包装器”。
  + std::function可以取代函数指针的作用。因为它可以保存函数延迟执行，所以比较适合作为回调函数，也可以把它看做类似于C#中特殊的委托（只有一个成员的委托）
  + std::function还可以作为函数入参，这样可以在函数外部控制函数的内部行为了，让我们的函数变得更加灵活。
  + std::function 是一种通用、多态的函数封装，它的实例可以对任何可以调用的目标实体进行存储、复制和调用操作，它也是对 C++中现有的可调用实体的一种类型安全的包裹（相对来说，函数指针的调用不是类型安全的），换句话说，就是函数的容器。当我们有了函数的容器之后便能够更加方便的将函数、函数指针作为对象进行处理。

#include <functional>  
#include <iostream>  
   
int add(int a,int b) {  
return a + b;  
}  
   
//使用函数对象包装器作为参数,改变模板的默认行为  
template<typename T, typename F>  
T calc(T a, T b, F f)  
{  
return f(a, b);  
}  
   
int main() {  
// std::function 包装了一个返回值为 int, 参数为 int 的普通函数add  
std::function<int(int,int)> func = add;  
  
  
// std::function 包装了一个返回值为 int, 参数为 int 的lambda函数  
std::function<int(int,int)> func2 = [&](int a,int b) -> int {  
return a - b;  
};  
   
//使用例子：  
//1.直接使用函数包装器  
std::cout << func(10,10) << std::endl;  
std::cout << func2(10,10) << std::endl;  
   
//2.将函数包装器作为参数使用  
std::cout << calc(10, 10,func2) << std::endl;  
return 0;  
}

* std::bind 绑定器
  + std::bind用来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进行保存，并延迟调用到任何我们需要的时候。通俗来讲，它主要有两大作用：
    - 将可调用对象与其参数一起绑定成一个仿函数。
    - 将多元（参数个数为n,n>1)可调用对象转成一元或者（n-1)元可调用对象，即只绑定部分参数。
  + function模板类和bind模板函数，都可以实现类似函数指针的功能，但却却比函数指针更加灵活，特别是函数指向类的非静态成员函数时。
    - std::function可以绑定到全局函数/类静态成员函数
    - 绑定到类的非静态成员函数，则需要使用std::bind

//使用组合bind函数，找出集合中大于5小于10的元素个数  
 #include <iostream>  
 #include <functional>  
   
 using namespace std;  
   
 auto f = std::bind(std::logical\_and<bool>(),std::bind(std::greater<int>(),\_1,5),std::bind(std::less\_equal<int>(), \_1, 10));  
   
 int main(){  
 set<int> se={1,2,3,4,5,6,7,8,9};  
 int count = std::count\_if(se.begin(), se.end(), f);  
 cout << count <<endl;  
   
 return 0;  
 }

* + std::bind需要注意的一些事项
    - std::bind预先绑定的参数需要传具体的变量或值进去，对于预先绑定的参数，是pass-by-value的
    - 对于不事先绑定的参数，需要传std::placeholders进去，从\_1开始，依次递增。placeholder是pass-by-reference的
    - bind的返回值是可调用实体，可以直接赋给std::function对象
    - 对于绑定的指针、引用类型的参数，使用者需要保证在可调用实体调用之前，这些参数是可用的
    - 类的this可以通过对象或者指针来绑定

## **1.6 lambda表达式**

* 特点:
  + 就地匿名定义目标函数或函数对象，不需要额外写一个命名函数或者函数对象。
  + 就地封装短小的功能闭包，可以机器方便第表达出我们希望执行的具体操作。
* 概念：定义了一个匿名函数，并且可以捕获一定范围内的变量。
  + {capture } { params } opt -> ret { body ; }
  + 其中：capture 是捕获列表， params 是参数表 ， opt 是函数选项 ， ret 是返回值 ， body 是函数体
  + lambda表达式的返回值由返回值后置语法来定义
  + 举例：

auto f = [](int a) -> int {return a+1; };   
std::cout << f(1) <<std::endl;

* 可省略：
  1. 没有参数列表时，省略参数列表
  2. 可省略 opt
  3. 可省略 ->
  4. 可省略 ret
  5. 举例：

auto f1 = []() { return 1; };  
auto f2 = [] { return 1 ; };

* 关于lambda表达式可以捕获的变量
  1. []不捕获
  2. [&] 按引用捕获 外部作用域中所有变量，并在函数体中使用 （可以修改）
  3. [=] 按值捕获 外部作用域中所有变量， 并在函数体中使用 （复制一份）
  4. [=,&foo] 按值捕获 外部作用域中所有变量，并 按引用捕获 foo变量
  5. [bar] 按值捕获 bar变量，同时不捕获其他变量
  6. [this] 捕获当前类的this指针，让lambda表达式拥有和当前类成员函数同样的访问权限。
  7. 显性指明lambda表达式为mutable， 可以修改按值捕获的外部变量 (此时参数列表不可省略)

auto f1 = [=]() mutable { return a++;};

* 没有捕获变量的lambda表达式可以直接转换为函数指针

【参考】

1. [【csdn：lambda】](https://blog.csdn.net/toby54king/article/details/105127327?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522165301690016782391817484%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=165301690016782391817484&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-1-105127327-null-null.142^v10^control,157^v4^control&utm_term=c%2B%2B+lambda%E8%A1%A8%E8%BE%BE%E5%BC%8F&spm=1018.2226.3001.4187)

## **1.7 tupe元组**

* 目前代码中还没见过，暂不展开

## **1.8 总结**

# **第二章：改进程序性能**

## **2.1 右值引用**

* 右值引用只不过是一种新的 C++ 语法，真正理解起来有难度的是基于右值引用引申出的 2 种 C++ 编程技巧，分别为移动语义和完美转发
  + 右值引用可以从字面意思上理解，指的是以引用传递（而非值传递）的方式使用 C++ 右值。
  + 左值的英文简写为“lvalue”，lvalue 是“loactor value”的缩写，可意为存储在内存中、有明确存储地址（可寻址）的数据
  + 右值的英文简写为“rvalue”，译为 “read value”，指的是那些可以提供数据值的数据（不一定可以寻址，例如存储于寄存器中的数据）
  + 其实 C++98/03 标准中就有引用，使用 “&” 表示。但此种引用方式有一个缺陷，即正常情况下只能操作 C++ 中的左值，无法对右值添加引用。
  + 判断方法：
    - 可位于赋值号（=）左侧的表达式就是左值；反之，只能位于赋值号右侧的表达式就是右值
    - 有名称的、可以获取到存储地址的表达式即为左值；反之则是右值

int a = 5;  
5 = a; //错误，5 不能为左值  
int b = 10; // b 是一个左值  
a = b; // a、b 都是左值，只不过将 b 可以当做右值使用  
//左值引用  
int num = 10;  
int &b = num; //正确  
int &c = 10; //错误  
//常量左值引用既可以操作左值，也可以操作右值  
int num = 10;  
const int &b = num;  
const int &c = 10;

* 右值引用
  + 实际开发中我们可能需要对右值进行修改（实现移动语义时就需要），显然左值引用的方式是行不通的。因此C++11 标准新引入右值引用，用 “&&” 表示。
  + 和声明左值引用一样，右值引用也必须立即进行初始化操作，且只能使用右值进行初始化
  + 和常量左值引用不同的是，右值引用还可以对右值进行修改
  + C++ 语法上是支持定义常量右值引用的，但这种定义出来的右值引用并无实际用处。

/\*右值初始化\*/  
int num = 10;  
//int && a = num; //右值引用不能初始化为左值  
int && a = 10;  
/\* 右值引用还可以对右值进行修\*/  
int && a = 10;  
a = 100;  
cout << a << endl;  
程序输出结果为 100。

* + 图示

## **2.2 move语义**

* move语义
  + move是将对象的状态或者所有权从一个对象转移到另一个对象
  + move实际上并不能移动任何东西，它只是将一个左值强制转换成一个右值引用，使我们可以通过右值引用使用该值，以用于移动语义，强制转换为右值的目的是为了方便实现移动构造。
  + move语义允许一个对象在特定的情形下，取得其他对象的资源，这也就是经典的右值引用，而被引用的变量被释放。
  + 当一个对象内部有较大的堆内存或者动态数组时，很有必要写move语义的拷贝构造函数和赋值函数，避免无谓的深拷贝，以提高性能。

#include <iostream>  
#include <utility>  
#include <vector>  
#include <string>  
   
int main()  
{  
 std::string str = "Hello";  
 std::vector<std::string> v;  
   
 // 使用 push\_back(const T&) 重载，  
 // 表示我们将带来复制 str 的成本  
 v.push\_back(str);  
 std::cout << "After copy, str is \"" << str << "\"\n";  
   
 // 使用右值引用 push\_back(T&&) 重载，  
 // 表示不复制字符串；而是  
 // str 的内容被移动进 vector  
 // 这个开销比较低，但也意味着 str 现在可能为空。  
 v.push\_back(std::move(str));  
 std::cout << "After move, str is \"" << str << "\"\n";  
   
 std::cout << "The contents of the vector are \"" << v[0]  
 << "\", \"" << v[1] << "\"\n";  
}  
  
---------结果如下--------  
After copy, str is "Hello"  
After move, str is ""  
The contents of the vector are "Hello", "Hello"

## **2.3 forward和完美转发**

* 不完美转发
  + 所谓完美转发，是指在函数模板中，完全按照模板的参数的类型，将[参数传递](https://so.csdn.net/so/search?q=%E5%8F%82%E6%95%B0%E4%BC%A0%E9%80%92&spm=1001.2101.3001.7020)给函数模板中调用的另一个函数
  + 在ImaForwarding的参数中使用了最基本类型进行转发，该方法会导致参数在传给IrunCodeActually之前就产生了一次临时对象拷贝。因此这样的转发只能说是正确的转发，谈不上完美。

template <typename T>  
void IamForwording(T t)  
{  
 IrunCodeActually(t);  
}

* 完美转发
  + 一个右值引用参数作为函数的形参，在函数内部再转发该参数的时候它已经变成一个左值了，并不是原来的类型。如果我们需要按照参数原来的类型转发到另一个函数，这种转发就被成为完美转发。
  + 一旦定义中出现了左值引用，引用折叠总是优先将其折叠为左值引用
  + C++11提供了一个函数用于解决这个问题，通过std::forward来实现，不管参数是T&&这种未定的引用还是明确的左值引用或者右值引用，它会按照参数原本的类型转发。

& + & -> &  
& + && -> &  
&& + & -> &  
&& + && -> &&

* std::forward
  + 转换规则
    - 模板参数为引用类型T、T&&时，返回右值引用
    - 模板参数为引用类型T&时，返回左值引用

using namespace std;  
  
template<typename T>  
void PrintT(int &t)  
{  
 cout << "lvalue" << endl;  
}  
  
template<typename T>  
void PrintT(int &&t)  
{  
 cout << "rvalue" << endl;  
}  
  
template<typename T>  
void TestForward(T && v)  
{  
 PrintT(v);  
 PrintT(std::forward<int>(v));  
 PrintT(std::move(v));  
 cout << endl;  
}  
  
  
int main()  
{  
 TestForward(1);  
 int x = 1;  
 TestForward(x);  
 TestForward(std::forward<int>(x));  
  
 test();  
 return 0;  
}  
===========结果  
lvalue  
rvalue  
rvalue  
  
lvalue  
rvalue  
rvalue  
  
lvalue  
rvalue  
rvalue

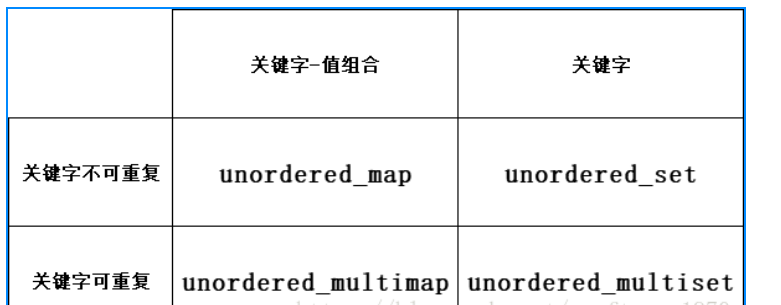
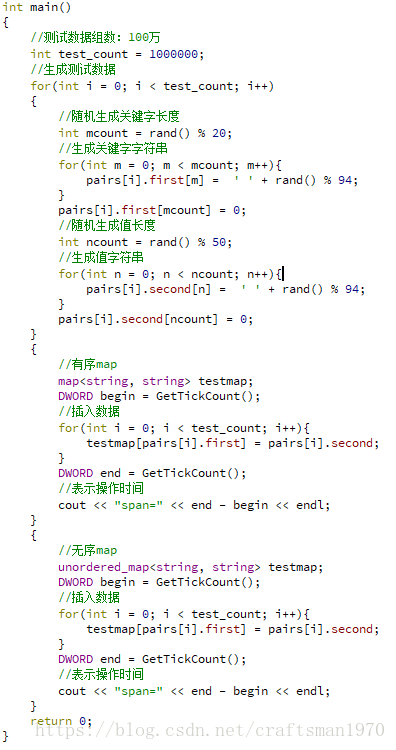
* 扩展
  + 右值引用、完美转发再结合可变模板参数，可以写一个万能的函数包装器，带返回值的、不带返回值的、带参数的或者不带参数的都可以委托这个万能的函数包装器来实现参数的完美转发。

## **2.4 emplace\_back 减少内存拷贝和移动**

* 背景
  + 我们在对STL容器进行插入操作时，常会使用insert或push\_back。C++11提出了更高效的插入方法：emplace
  + 新标准引入了三个新成员：emplace\_front、emplace和emplace\_back，这些操作构造而不是拷贝元素。这些操作分别对应push\_front、insert和push\_back，允许我们将元素放置在容器头部、一个指定位置之前或容器尾部。
* 以vector举例
  + push\_back(右值)：先向容器尾部添加一个临时对象（调用一次构造函数），最后调用一次移动构造函数将这个临时对象放入容器中并释放这个临时对象（调用一次析构函数）。
    - 注：最后调用的不是拷贝构造函数，而是移动构造函数。因为需要释放临时对象，所以通过std::move进行移动构造，可以避免不必要的拷贝操作。
  + push\_back(左值)：调用一次拷贝构造函数。
  + emplace\_back()：在容器尾部添加一个元素，调用构造函数原地构造，不需要触发拷贝构造函数和移动构造函数。因此比push\_back()更加高效。

#include <iostream>  
#include <vector>  
#include <string>  
using namespace std;  
  
struct Student {  
string name;  
int age;  
  
Student(string n, int a)  
:name(n), age(a)  
{  
cout << "ctor" << endl;  
}  
  
Student(const Student& s)  
: name(s.name), age(s.age)  
{  
cout << "copy ctor" << endl;  
}  
  
Student(Student&& s)  
:name(std::move(s.name)), age(s.age)  
{  
cout << "move ctor" << endl;  
}  
Student& operator=(const Student& src) noexcept  
{  
if (this == &src)  
return \*this;  
this->name = src.name;  
this->age = src.age;  
cout << "copy assignment" << endl;  
return \*this;  
}  
Student& operator=(Student&& src) noexcept  
{  
if (this == &src)  
return \*this;  
this->name = move(src.name);//调用string类的move assignment  
this->age = src.age;  
cout << "move assignment" << endl;  
return \*this;  
}  
  
};  
  
int main()  
{  
//基本数据类型的插入  
vector<int> v1;  
v1.push\_back(1);  
v1.emplace\_back(1);  
  
//自定义类型的插入  
vector<Student> v2,v3;  
v2.emplace\_back("xiaohong", 24); //ctor  
v3.push\_back(Student("xiaoming", 23)); //ctor和move ctor和dtor  
  
system("pause");  
return 0;  
}  
  
============结果  
ctor  
ctor  
move ctor

## **2.5 unordered container 无序容器**

* C++11之前的关联容器一共有四种。
  + 两种最基本的类型:map和set。map中保存的是关键字-值的组合，而set中只保存关键字。
    - map和set中的关键字不允许重复，一旦相同的关键字重新写入，则原来的内容被覆盖。如果希望关键字可以重复，可以使用multimap和multiset。
    - 由于这些关联容器的存储位置是根据比较运算符决定的，所以也成为有序关联容器。
* 无序关联容器
  + 这些容器将存储组织为一组桶，根据哈希值将数据映射到桶
  + 
  + 除了哈希管理操作以外，无序容器还提供了与有序容器相同的操作。也就是说有序容器和无序容器可以互换
  + 优势：通常情况下，无序容器通常会有更好的性能。
* 栗子：
  + 
  + 从实际的结果来看，使用无序关联容器时的操作时间不到使用有序关联容器时的一半。
* 详细解释
  + 暂不展开

## **2.6 总结**

# **第三章：提高代码质量 (模板类：待补充)**

## **3.1 type\_traits ———类型萃取**

通过type\_traits可以实现在编译期计算、查询、判断、转换和选择，增强了泛型编程的能力，也增强了程序的弹性，使得我们在编译期就能做到优化改进甚至排错，能进一步提高代码质量。

* 假设我们要实现一个针对16位、32位型或64位整数类型的字节交换的功能，借助于C++泛型编程，我们可以很容易的实现：

template <typename T>  
T byte\_swap( T value ) {  
 unsigned char \*bytes = reinterpret\_cast< unsigned char \* >( &value );  
 for (size\_t i = 0; i < sizeof( T ); i += 2) {  
 // Take the value on the left and switch it   
 // with the value on the right  
 unsigned char v = bytes[ i ];  
 bytes[ i ] = bytes[ i + 1 ];  
 bytes[ i + 1 ] = v;  
 }  
 return value;  
}

* + 如果传入32位的值0x11223344，返回值为0x22114433，如果传入16位的值0x1122，将返回0x2211。看起来这个功能实现的不错。
  + 但是 … 如果传入一个char类型的值呢？代码会因为访问到不属于自己的内存而引起程序崩溃。由于模板的泛型特性，我们也无法阻止用户传递这样类型的值。
  + 为了处理用户传入的double和char类型的值，我们可以加入模板特化进行处理

template <>  
double byte\_swap( double value ) {  
 assert( false && "Illegal to swap doubles" );  
 return value;  
}  
  
template <>  
char byte\_swap( char value ) {  
 assert( false && "Illegal to swap chars" );  
 return value;  
}

* + 这个特化处理可以处理传入的double和char类型的值，问题是，用户还可能传入float、unsigned char等类型。如果我们为每种类型都添加上特化处理，可以想象得到，代码会膨胀成怎样。
* 这个时候Type Traits可以派上用场了。Type Traits是在编译时获取有关作为模板参数传入的类型的信息的一种方式，因此我们可以做出更明智的决定。
  + Type Traits的典型用法如下：
    - 使用一个模板化的结构，通常以的类型特征命名。例如is\_integer，is\_pointer，is\_void等等
    - 结构包含一个静态const bool命名值
    - 我们可以对特征的结构进行特化，并且把它们的布尔值设置为一个合理的状态值
    - 我们可以查询其值来使用类型特征，如：my\_type\_trait :: value
    - 继续上面的例子来说明，我们可以通过定义一个类型特征来决定某个类型的值是否可交换：

template <typename T>  
struct is\_swapable {  
 static const bool value = false;  
};  
  
  
template <>  
struct is\_swapable<unsigned short> {  
 static const bool value = true;  
};  
  
  
template <>  
struct is\_swapable<short> {  
 static const bool value = true;  
};  
  
  
template <>  
struct is\_swapable<unsigned long> {  
 static const bool value = true;  
};  
  
  
template <>  
struct is\_swapable<long> {  
 static const bool value = true;  
};  
  
  
template <>  
struct is\_swapable<unsigned long long> {  
 static const bool value = true;  
};  
  
  
template <>  
struct is\_swapable<long long> {  
 static const bool value = true;  
};

* + - 这样我们只需在byte\_swap函数中增加一行语句

assert( is\_swapable<T>::value && "Cannot swap this type" );

* + - 这就是Type Traits的核心用法。
* 后续内容可扩展的非常多，贴两个参考网址

【参考网址】

1.【[博客园：类型萃取](https://www.cnblogs.com/gtarcoder/p/4807670.html)】 2. 【[csdn:type traits](https://blog.csdn.net/CodeMyDream/article/details/116375341?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_utm_term~default-0-116375341-blog-79264925.pc_relevant_default&spm=1001.2101.3001.4242.1&utm_relevant_index=3)】

## **3.2 可变参数模板**

## **3.3 可变参数模板和type\_taits的综合应用**

## **3.4 总结**

# **第四章：解决内存泄露问题**

## **4.1 shared\_ptr 共享的指针**

* 头文件<memory>
* std::shared\_ptr 使用引用计数，每一个shared\_ptr的拷贝都指向相同的内存。在最后一个shared\_ptr析构的时候，内存才会被释放。
* shared\_ptr的基本用法
  + 初始化：可以使用构造函数、std::make\_shared<T>辅助函数和reset函数方法来初始化shared\_ptr
    - 优先使用make\_shared 来构造一个智能指针
  + 获取原始指针
    - 当需要获取原始指针时，可以通过get方法来返回原始指针。
  + 指定删除器
    - ====2022年5月24日
  + 使用shared\_ptr需要注意的问题
    - 不要用一个原始指针初始化多个shared\_ptr
    - 不要再函数参数中创建shared\_ptr
      * 先创建智能指针，再用
    - 通过shared\_from\_this（）返回this指针，不要将this指针作为shared\_ptr返回出来，因为this指针本质时一个罗指针，因此这样可能会导致重复析构。
    - 要避免循环引用、循环引用会导致内存泄漏

【参考】

1. [【CSDN : 共享指针】](https://blog.csdn.net/shaosunrise/article/details/85228823?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522165329293716782246495366%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=165329293716782246495366&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-85228823-null-null.142^v10^control,157^v5^new_style2&utm_term=shared_ptr&spm=1018.2226.3001.4187) 2.

## **4.2 unique\_ptr 独占的智能指针**

## **4.3 weak\_ptr 弱引用的智能指针**

弱引用指针weak\_ptr是用来检视shared\_ptr的，不会是引用计数+1，他部管理shared\_ptr内部的指针，主要卫视检视shared\_ptr的生命周期，更像是shared\_ptr的一个助手。

* weak\_ptr的基本用法
  + 1
  + 2
  + 3
* weak\_ptr返回this指针
* weak\_ptr解决循环引用的问题

## **4.4 通过智能指针管理第三方库**

## **4.5 总结**

# **第五章：多线程开发**

## **5.1 线程**

* 头文件： <thread>
* 建立线程

std::thread t(func);  
t.join(); //阻塞线程，直到线程函数执行完毕  
t.detach();// 让线程到后台执行，当前线程不会被阻塞。  
// 但分离之后就无法再和线程发送联系了。

* 1. 线程不可复制 ，但可以移动 ： std::thread t1( std::move(t) )
  2. 需要注意线程对象的生命周期：
     1. 应该保证线程对象的生命周期再线程函数执行完时仍存在。
* 线程的基本用法
  + 获取当前信息：
    - t.get\_id() 获取当前线程ID
    - std::thread::hardware\_concurrenct() 获取CPU核数，失败返回0
  + 线程休眠
    - std::this\_thread::sleep\_for(std::chrono::seconds(3));
      * 线程休眠3秒

## **5.2 互斥量**

* 一种线程同步的手段，用来保护多线程同时访问的共享数据。
  + <mutex>
  + ~~<chrono>~~
* 独占互斥量 std::mutex
  + 一般用法是用lock()来阻塞线程，知道获得互斥量的所有权为止。
    - 获得互斥量并且完成任务后，就必须使用unlock()来解除对互斥量的占用
    - 所以lock()和unlock（）必须成对出现。
  + try\_lock() 尝试锁定互斥量，成功返回ture，失败返回false， 是非阻塞的。
  + 使用lock\_guard 可以简化lock/unlock的写法，同时更加安全
    - 因为lock\_guard 构造是会自动锁定互斥量，析构时会自动解锁
    - 可以避免忘记unlock操作
    - std::lock\_guard< std::mutex> > locker<g\_lock>
* 递归的独占互斥量 std::recursive\_mutex
  + 先不写
  + ============2022年5月24日
* 带超时的互斥量 std:::timed\_mutex 和std::recursive\_timed\_mutex
  + 先不写
  + ============2022年5月24日

## **5.3 条件变量**

* 头文件：<condition\_variable>
* 一种用于等待的同步机制，他能阻塞一个或多个线程，知道收到另一个线程发出的通知或者超时，才会唤醒当前阻塞的线程。条件变量需要和互斥量配合起来使用。
* 拥有条件变量的线程获取互斥量
  + 循环检测某个条件，如果条件不满足，则阻塞到条件满足；如果条件满足，则向下执行
  + 某个线程满足条件执行之后条用notify\_one 或notify\_all唤醒一个或者所有的等待线程。
  + 可以使用条件变量来实现一个同步队列，同步队列作为一个线程安全的数据共享区，经常用于线程之间的数据读取，比如半同步半异步线程池的同步队列。

void put(const T& x)  
{  
 std::unique\_lock < std :: mutex > locker(m\_mutex);  
 while( IsFull())  
 {  
 std::cout<<"缓冲区满了，需要等待"<<std::endl;  
 m\_notFull.wait(m\_mutex);  
 }  
 /\* ============简洁写法===========\*/  
 m\_notFull.wait(locker,[this]{return !IsFull()};);  
 /\*================================\*/  
 m\_queue.push\_back(x);  
 m\_notEmpty.notify\_one();  
}  
std::condition\_variable m\_notFull;

* lock\_gurad 只能在析构时才会释放锁，unique\_lock可以随时释放锁。
  + 因此再这里使用unique\_lock 更为准确，虽然前者也不会又问题。
  + lock\_guard >>conditon\_variable\_any
  + unique\_lock >> condition\_variable

## **5.4 原子变量**

* <atomic>

c++11 提供了一个原子类型std::atomic<T> ,可以使用任意类型作为模板参数。

使用原子变量就不需要用互斥量来保护该变量了，用起来更简洁了。

#include<atomic>  
  
struct AtomicCounter{  
 std::atomic<int> value;  
 void increment(){  
 ++value;  
 }  
 void decrement(){  
 --value;  
 }  
 int get(){  
 return value.load();  
 }  
}

## **5.5 call\_once/once\_flag 的使用**

为了保证再多线程环境中某个函数仅被调用一次，比如初始化。

就可以用std::call\_once来保证函数在多线程环境中只被调用一次。

使用std::cal\_\_once时，需要一个once\_flag 作为call\_once的入参，他的用法比较简单。

std::once\_flag flag;  
void do\_once()  
{  
 std::call\_once(flag,[](){std::cout<<"called once"<<std::endl;});  
}  
int main()  
{  
 std::thread t1(do\_once);  
 std::thread t2(do\_once);  
 std::thread t3(do\_once);  
   
 t1.join();  
 t2.join();  
 t3.join();  
  
}  
  
/\* ===========输出结果============\*/  
Called once

## **5.6 异步操作类**

* std:: future
  + thread库提供了future用来访问异步操作的结果，因为一个异步操作的结果不能马上获取，只能在未来的某个时候从某个地方获取，这个异步操作的结果时一个未来的期待值，所以被称为future。future提供了一个获取异步操作结果的通道。
  + 我们可以通过同步等待的方式来获取结果，可以通过查询future的状态来获取异步操作的结果。
  + future\_status 有三个状态
    - Deferred ，异步操作还没开始
    - Ready ，异步操作已经完成
    - Timeout ， 异步操作超时
    - 可以通过不断查询future的状态，知道异步任务的执行情况。
  + 获取future的结果有3种方式
    - get 等待异步操作结束并返回结果
    - wait 等待异步操作完成，没有返回值
    - wait\_for 超时等待返回结果
* std::promise
  + std::promise 将数据和future绑定起来，为获取线程函数中的某个值提供便利，在线程函数中为外面传进来的promise赋值。

std::promise<int> pr;  
std::thread t([](std::promise<int>&p)  
 {  
 p.set\_value\_at\_thread\_exit(9);  
 },std::ref(pr));  
std::future<int> f = pr.get\_future();  
auto r = f.get();

* std::package\_task
  + 包装了一个可调用对象的包装类，将函数和tufure绑定起来，以便于异步调用，他和std::promise在某种程度上有点像。promise保存了一个共享状态的值，而packaged\_task保存的是一个函数。

std::packageg\_task<int()> taks([](){return 7;});  
std::thread t1(std::ref(task));  
std::future<int> f1 = task.get\_future();  
auto r1 = f1.get();

* 三者之间的关系
  + future提供了一个访问异步操作结果的机制。他和线程是一个级别的，属于低层次的对象。在future之上的高一层是packaged\_task 和 promise ，他们内部都有future以便访问异步操作结果。
  + 有时需要获取线程中的某个值，这是就用promise，而有时需要获取一个异步操作的返回值，就用packaged\_task。
  + 懵逼了这里。

## **5.7 线程异步操作函数 async**

* std::async 比 std：：promise、std::packaged\_task 和 std::thread更高一层，他可以用来直接创建异步的task，异步任务返回的结果也保存在future中，当需要获取异步任务的结果时，值需要调用future.get()方法即可，如果不关注异步任务的结果，只是简单地等待任务完成得话，则调用future.wait()方法。
* std::async 的原型 async(std::launch::async | std::launch::deferred,f,args....)
  + 第一个参数时线程的创建策略，有两种策略，默认的测量是立即创建线程
    - std::launch::async : 在调用async时就开始创建线程
    - std::launch::deferred : 延迟加载方式创建线程。调用async时不创建线程，知道调用了future的get或wait时才创建线程
  + 第二个参数是线程函数，
  + 第三个参数是线程函数的参数

## **5.8 总结**

==待补充 2022年5月24日

# **第六章：便利工具**

## **6.1 处理日期和时间的chrono库**

* 原则上不需要自己造轮子
* 后续有时间再展开

## **6.2 数值类型和字符串的相互转换**

* c++11提供了to\_string方法，将各种数值类型转换为字符串类型

// 头文件 <string>  
string to\_string (int val);  
string to\_string (long val);  
string to\_string (long long val);  
string to\_string (unsigned val);  
string to\_string (unsigned long val);  
string to\_string (unsigned long long val);  
string to\_string (float val);  
string to\_string (double val);  
string to\_string (long double val);

* 还提供了stoxxx方法，将string转换为各种类型的数据

// 定义于头文件 <string>  
int stoi( const std::string& str, std::size\_t\* pos = 0, int base = 10 );  
long stol( const std::string& str, std::size\_t\* pos = 0, int base = 10 );  
long long stoll( const std::string& str, std::size\_t\* pos = 0, int base = 10 );  
  
unsigned long stoul( const std::string& str, std::size\_t\* pos = 0, int base = 10 );  
unsigned long long stoull( const std::string& str, std::size\_t\* pos = 0, int base = 10 );  
  
float stof( const std::string& str, std::size\_t\* pos = 0 );  
double stod( const std::string& str, std::size\_t\* pos = 0 );  
long double stold( const std::string& str, std::size\_t\* pos = 0 );

* str：要转换的字符串
* pos：传出参数，该地址用于存储转换之后得到的整数值
* base：若 base 为 0 ，则自动检测数值进制：

若前缀为 0 ，则为八进制，若前缀为 0x 或 0X，则为十六进制，否则为十进制

* + 注意
    - **如果字符串中所有字符都是数值类型，整个字符串会被转换为对应的数值，并通过返回值返回**
    - **如果字符串的前半部分字符是数值类型，后半部不是，那么前半部分会被转换为对应的数值，并通过返回值返回**
    - **如果字符第一个字符不是数值类型转换失败**

## **6.3 宽窄字符转换**

* c++11增加了unicode字面量的支持，可以通过L来定义宽字符
  + std::wstring wide\_str = L"中国人"; //定义了宽字符字符串
  + 将宽字符转换为窄字符需要用到codecvt库中的std::wstring\_convert
  + std::wstring\_convert使std::string和std::wstring相互转换：

#include "stdio.h"  
  
#include<string>  
#include<codecvt>  
#include<iostream>  
using namespace std;  
  
void main()  
{  
 wstring wstr = L"中国人";  
 cout << "unicode编码：" << wstr.c\_str() << endl;  
  
 wstring\_convert<codecvt<wchar\_t, char, mbstate\_t>> converter(new codecvt<wchar\_t, char, mbstate\_t>("CHS"));  
  
 string str = converter.to\_bytes(wstr);  
 cout << "ansi编码：" << str << " " << str.c\_str() << endl;  
  
 wstring wstr1 = converter.from\_bytes(str);  
 wcout.imbue(locale("CHS")); // 初始化cout为中文输出  
 wcout << L"unicode编码：" << wstr1 << endl;  
}

## **6.4 总结**

# **第七章：其他特性**

## **7.1 委托构造函数 和继承构造函数**

* 委托构造函数
  + 如果一个类中重载了多个构造函数，并且在每个构造函数中都需要对某些变量进行初始化，这时候就会出现很多重复的代码。
  + C++11中新增的委托构造函数，就是为了解决这个问题，它允许在同一个类中一个构造函数可以调用另外一个构造函数，从而可以在初始化时简化变量的初始化。
  + 老版本的写法示例：

class ClassA{  
  
public:  
 ClassA(){}  
 ClassA(int max){  
 m\_max = max > 0 ? max : 20;  
 }  
 ClassA(int max,int min){  
 m\_max = max > 0 ? max : 20;  
 m\_min = min > 0 && min < m\_max ? min : 5;  
 }  
 ClassA(int max,int min,int middle){  
 m\_max = max > 0 ? max : 20;  
 m\_min = min > 0 && min < m\_max ? min : 5;  
 m\_middle = middle < max && middle > min ? middle : 10;  
 }  
  
private:  
 int m\_max;  
 int m\_min;  
 int m\_middle;  
};

* + 通过委托构造函数就可以简化这个过程：

class ClassA{  
  
public:  
 ClassA(){}  
 ClassA(int max){  
 m\_max = max > 0 ? max : 20;  
 }  
 ClassA(int max,int min):ClassA(max){  
 m\_min = min > 0 && min < m\_max ? min : 5;  
 }  
 ClassA(int max,int min,int middle):ClassA(max,min){  
 m\_middle = middle < max && middle > min ? middle : 10;  
 }  
  
private:  
 int m\_max;  
 int m\_min;  
 int m\_middle;  
};

* + 需要注意的是，这种链式调用构造函数不能形成一个环，否则将会在运行时抛异常
  + 使用了委托构造函数后就不能用类成员初始化了
* 继承构造函数
  + 如果有一个派生类，希望能和基类一样采取相同的构造方式，其直接派生于基类是不能获取基类构造函数的。因为C++派生类会隐藏基类同名函数。

class Base  
{  
public:  
 Base(int i):x(i),y(0){}  
 Base(int i,double j):x(i),y(j){}  
 Base(int i,double j,const string & str):x(i),y(j),s(str){}  
  
private:  
 int x;  
 double y;  
 string s;  
};  
  
class Derived : Base  
{  
  
};  
  
int main()  
{  
 Derived d(1,3.2,"hello"); //error:没有合适的构造函数  
 return 0;  
}

* + 若希望使用基类的构造函数，可行的方法就是在派生类中也定义这些构造函数并依次调用基类。

class Base  
{  
public:  
 Base(int i):x(i),y(0){}  
 Base(int i,double j):x(i),y(j){}  
 Base(int i,double j,const string & str):x(i),y(j),s(str){}  
  
private:  
 int x;  
 double y;  
 string s;  
};  
  
class Derived : Base  
{  
public:  
 Derived(int i):Base(i){}  
 Derived(int i,double j):Base(i,j){}  
 Derived(int i,double j,const string & str):Base(i,j,str){}  
};  
  
int main()  
{  
 Derived d1(1);  
 Derived d2(1,3.2);  
 Derived d3(1,3.2,"hello");  
 return 0;  
}

* + 以上方法虽然可行，但是代码非常繁琐切重复性高，而C++11中的继承构造函数特征正是用于**解决派生类隐藏基类同名函数**的问题
  + 可以通过 using Base::SomeFunction来表示使用基类的同名函数。

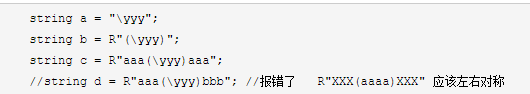
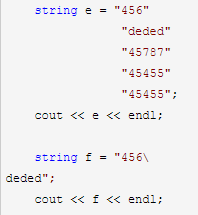
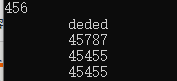
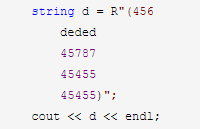
class Base  
{  
public:  
 Base(int i):x(i),y(0){}  
 Base(int i,double j):x(i),y(j){}  
 Base(int i,double j,const string & str):x(i),y(j),s(str){}  
  
private:  
 int x;  
 double y;  
 string s;  
};  
  
class Derived : Base  
{  
public:  
 using Base::Base; //声明使用基类的构造函数  
};  
int main()  
{  
 Derived d1(1);  
 Derived d2(1,3.2);  
 Derived d3(1,3.2,"hello");  
 return 0;  
}

* + 需要注意的是，继承构造函数不会去初始化派生类新定义的数据成员
  + 如果通过 using Base::Base；来声明使用基类的构造函数，但是同时又在派生类中定义相同构造函数。
    - 产生的结果是，会直接调用派生类中的构造函数。
    - 该特性不仅对构造函数有用，对其他同名函数也适用。

## **7.2 原始的字面量**

* 定义
  + 原始字符串字面量的定义为：R “xxx(raw string)xxx”
  + 其中，原始字符串必须用括号（）括起来，括号的前后可以加其他字符串，所加的字符串会被忽略，并且加的字符串必须在括号两边同时出现。

#include <iostream>  
#include <string>  
  
int main()  
{  
 // 一个普通的字符串，'\n'被当作是转义字符，表示一个换行符。  
 std::string normal\_str = "First line.\nSecond line.\nEnd of message.\n";  
  
 // 一个raw string，'\'不会被转义处理。因此，"\n"表示两个字符：字符反斜杠 和 字母n。  
 std::string raw\_str = R"(First line.\nSecond line.\nEnd of message.\n)";  
  
 std::cout << normal\_str << std::endl;  
 std::cout << raw\_str << std::endl;  
 std::cout << R"foo(Hello, world!)foo" << std::endl;  
  
 // raw string可以跨越多行，其中的空白和换行符都属于字符串的一部分。  
 std::cout <<R"(  
 Hello,  
 world!  
 )" << std::endl;  
  
 return 0;  
}

* 举例：
  + 
  + 
* 举例2：
  + 
  + 

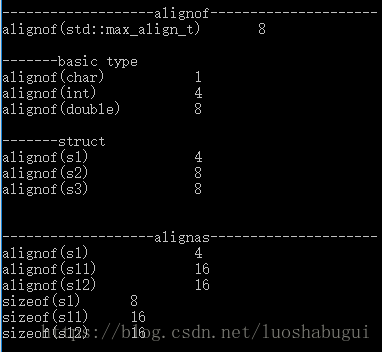
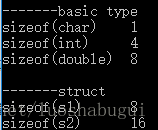
## **7.3 final 和override 标识符**

* final 标识符
  + 只能修饰虚函数
  + 要放在类或者函数的后面
  + 用来限制某个类不能被继承
  + 或者某个虚函数不能被重写
* override标注符
  + 保证派生类中声明的重写函数与基类的虚函数有相同的签名，同时表明将会重写基类的虚函数
  + 防止因失误把本来想重写基类的虚函数声明称重载。

## **7.4 内存对齐  alignof alignas**

* 什么是内存对齐：
  + 数据项仅仅能存储在地址是数据项大小的整数倍的内存位置上
* 为什么要内存对齐
  + 硬件原因，一些硬件平台必须要求内存对齐，否则抛出异常；另外涉及到不同平台的移植问题
  + 性能原因，对齐后访问效率更高
* 实现
  + C++11 新引入**操作符alignof**， **对齐描述符alignas**，**基本对齐值 alignof(std::max\_align\_t)**
  + alignas可以接受常量表达式和类型作为参数，可以修饰变量、类的数据成员等，不能修饰位域和用register申明的变量。一般往大对齐。直接看一段代码

struct s1  
{  
 char s;  
 int i;  
};  
   
struct s2  
{  
 int i;  
 double d;  
};  
   
cout << "-------basic type" << endl;  
cout << "sizeof(char)" << sizeof(char) << endl;  
cout << "sizeof(int)" << sizeof(int) << endl;  
cout << "sizeof(double)" << sizeof(double) << endl;  
   
cout << endl;  
cout << "-------struct" << endl;  
cout << "sizeof(s1)" << sizeof(s1) << endl;  
cout << "sizeof(s2)" << sizeof(s2) << endl;  
struct s3  
{  
 char s;  
 double d;  
 int i;  
};  
   
   
struct s11  
{  
 alignas(16) char s;  
 int i;  
};  
   
struct s12  
{  
 alignas(16) char s;  
 int i;  
};  
   
   
// alignof  
cout << "-------------------alignof---------------------" << endl;  
// 基本对齐值  
cout << "alignof(std::max\_align\_t)" << alignof(std::max\_align\_t) << endl;  
cout << endl;  
cout << "-------basic type" << endl;  
cout << "alignof(char)" << alignof(char) << endl;  
cout << "alignof(int)" << alignof(int) << endl;  
cout << "alignof(double)" << alignof(double) << endl;  
   
cout << endl;  
cout << "-------struct" << endl;  
cout << "alignof(s1)" << alignof(s1) << endl;  
cout << "alignof(s2)" << alignof(s2) << endl;  
cout << "alignof(s3)" << alignof(s3) << endl;  
   
cout << endl;  
cout << endl;  
   
// alignas  
cout << "-------------------alignas---------------------" << endl;  
cout << "alignof(s1)" << alignof(s1) << endl;  
cout << "alignof(s11)" << alignof(s11) << endl;  
cout << "alignof(s12)" << alignof(s12) << endl;  
   
cout << "sizeof(s1) " << sizeof(s1) << endl;  
cout << "sizeof(s11)" << sizeof(s11) << endl;  
cout << "sizeof(s12)" << sizeof(s12) << endl;

* + 结果
* 其他新增的内存对齐函数 （暂不展开）
  + std::alignment\_of
  + std::aligned\_storage
  + std::max\_align\_t
  + std::align

## **7.5 C++ 11新增的便利算法**

* 没见过：先略

## **7.6 总结**

# **工程级代码**

# **第八章 改进模式**

## **8.1 改进单例模式**

## **8.2 改进观察者模式**

## **8.3 改进访问者模式**

## **8.4 改进命令模式**

## **8.5 改进对象池模式**

# **第九章 开发半同步半异步线程池**

## **9.1 半同步半异步线程池介绍**

* 半同步半异步线程池分为三层：同步服务层，排队层，异步服务层
  + 第一层是同步服务层，她处理来自上层的任务请求，上层的请求是可能是并发的，这些请求不是马上就被处理，而是将这些任务放到一个同步排队层中，等待处理。
  + 第二层是同步排队层，来自上层的任务请求都会 加到排队层中等待处理。
  + 第三次是异步服务层，这一层中会有多个线程同时处理排队层中的任务，一部服务层自从给同步服务层中取出任务并行的处理。
* 这些线程是一开始就创建的，不会因为大量的任务到来而创建新的线程，避免拿了频繁创建和销毁线程导致的系统开销。而且通过多和处理大幅提高处理效率。

## **9.2 线程池实现的关键技术介绍**

线程池有两个活动过程，一个是往同步队列中添加任务的过程，另一个是从同步队列中取任务的过程。

## **9.3 同步队列**

同步队列 是中间那一层，主要作用是

* 1. 保证队列中共享数据线程安全
  2. 为上一次同步服务层提供添加新任务的接口
  3. 为下一次一部服务层提供取任务的接口。

#include<list>  
#include<mutex>  
#include<thread>  
#include<condition\_variable>  
#include<iostream>  
template<typename T>  
class SyncQueue  
{  
 public:  
 SyncQueue(int maxSize):m\_maxSize(maxSize),m\_needStop(false)  
 {  
 }  
 void Put(const T&x)  
 {  
 Add(x);  
 }  
   
 void Put(T&x)  
 {  
 Add(std::forward<T>(x));  
 }  
 void Take(std::list<T>& list )  
 {  
 std::unique\_lock<std::mutes> locker(m\_mutex);  
 m\_notEmpty.wait(locker,[this]{return m\_needStop||NotEmpty();});  
   
 if(m\_needStop)  
 return ;  
 list = std::move(m\_queue);  
 m\_notFull.notify\_one();  
 }  
 void Stop()  
 {  
 {  
 std::lock\_guard<std::mutex> locker(m\_mutex);  
 m\_needStop = true;  
 }  
 m\_notFull.notify\_all();  
 m\_notEmpty.notify\_all();  
 }  
   
 bool Empty()  
 {  
 std::lock\_guard<std::mutex> locker(m\_mutex);  
 return m\_queue.empty();  
 }  
 bool Full()  
 {  
 std::lock\_guard<std::mutex> locker(m\_mutex);  
 return m\_queue.size() == m\_maxSize;  
 }  
   
 size\_t Size()  
 {  
 std::lock\_guard<std::mutex> locker(m\_mutex);  
 return m\_queue.size();  
 }  
 int Count()  
 {  
 return m\_queue.size();  
 }  
 private:  
   
 bool NotFull() const  
 {  
 bool full = m\_queue.size() >+ m\_maxSize;  
 if(full)  
 std::cout<<"缓冲区满了，需要等待..."<<std::endl;  
 return !full;  
 }  
   
 boolNotEmpty()  
 {  
 bool empty = m\_queue.empty();  
 if(empty)  
 std::cout<<"缓冲区满了，需要等待....，异步层的线程ID:"<<this\_thread:get\_id()<<std::endl;  
 return !empty;  
 }  
   
 template<typename F>  
 void Add(F& x)  
 {  
 std::unique\_lock<std::mutex> locker(m\_mutex);  
 m\_notFull.wait(locker,[this]{return m\_needStop|||NotFull();});  
 if(m\_needStop)  
 return ;  
 m\_queue.push\_back(std::forward<F>(X));  
 m\_notEmpty.notify\_one();  
 }  
   
 private:  
 std::list<T> m\_queue;//缓冲区  
 std::mutex m\_mutex;//互斥量和条件变量结合起来使用  
 std::condition\_variable m\_notEmpty;//不为空的条件变量  
 std::condition\_variable m\_notFull；//没有满的条件变量  
 int m\_maxSize; //同步队列最大的size  
 bool m\_needStop; //停止的标志  
   
   
}

## **9.4 线程池**

#include<list>  
#include<thread>  
#include<functional>  
##include<memory>  
#include<atomic>  
#include"SyncQueue.hpp>  
  
const int MaxRaskCount = 100;  
class ThreadPool  
{  
 using Task = std::function<void()>;  
 TreadPool(int numThread = std::thread""hardware\_concurrency()):m\_queue(MaxTaskCount)  
 {  
 Start(numThreads);  
 }  
 ~ThreadPool(void)  
 {  
 //如果没有停止时则主动停止线程池  
 Stop();  
}  
 void Stop()  
 {  
 //保证多线程情况下值调用一次stopthreadgroup  
 std::call\_once(m\_flag,[this]{StopThreadGroup();});  
 }  
 void AddTask(Task&&task)  
 {  
m\_queue.Put(std::forward<Task>(task));  
 }  
 void AddTask(const Task& task)  
 {  
 m\_queue.Put(task);  
 }  
 private:  
 void Start(int numThreads)  
 {  
 m\_running = true;  
 //创建线程组  
 for(int i=0;i<numThreads;i++)  
 {  
 m\_threadgroup.push\_back(std::make\_shared<std::thread>(&ThreadPool::RunInThread,this));  
 }  
 }  
 void RumInThread()  
 {  
 while(m\_running)  
 {  
 //取任务分别执行  
 std::list<Task> list;  
 m\_queue.Task(task);  
   
 for(auto& task : list)  
 {  
 if(!m\_running)  
 return ;  
 task();  
 }  
 }  
 }  
 void StopThradGroup()  
 {  
 m\_queue.Stop();//让同步队列中的线程停止  
 m\_running = false;//置为flase，让内部线程跳出循环并退出  
   
 for(auto thread : m\_threadgroup) //等待线程结束  
 {  
 if(thrad)  
 thrad -> join();  
   
 }  
 m\_threadgroup.clear();  
 }  
  
 std::list<std::shared\_ptr<std::thread>> m\_threadgroup; //处理任务的线程组  
 SyncQueue<Task> m\_queue;//同步队列  
 atomic\_bool m\_running;//是否停止的标志  
 std::once\_flag m\_flag;  
};

## **9.5 应用实例**

在这个例子中，线程池将初始化创建两个线程，然后外部线程将不断地想线程中添加新任务，线程池内部的线程将会并行处理同步队列的任务。

void TestThdPool()  
{  
 ThreadPool pool;  
   
 std::thread thd1([&pool]{  
 for(int i = 0 ; i<10;i++)  
 {  
 auto thdID = this\_thread::get\_id();  
 pool.AddTasj([thdId]{  
 std::cout<<"同步层线程1的线程ID："<<thdId<<std::endl;  
 });  
 }  
 });  
   
 std::thread thd2([&pool]{  
 for(int i = 0 ; i<10;i++)  
 {  
 auto thdID = this\_thread::get\_id();  
 pool.AddTasj([thdId]{  
 std::cout<<"同步层线程2的线程ID："<<thdId<<std::endl;  
 });  
 }  
 });  
   
 this\_thread::sleep\_for(std::chrono::second(2));  
 getchar();  
 pool.Stop();  
 thd1.join();  
 thd2.join();  
}

# **第10章 开发一个轻量级的AOP库 （后续暂时不想看）**

# **第11章 开发一个轻量级的IoC容器**

# **第12章 开发一个对象的消息总线库**

# **第13章 封装sqlite库**

# **第14章 开发一个linq to objects库**

# **第15章 开发一个轻量级的并行task库**

# **第16章 开发一个简单的通信程序**