c++11新特性，所有知识点都在这了！

0

c++程序员面试过程中基本上都会被问到c++11新特性吧，你是怎么回答的呢？

本文基本上涵盖了c++11的所有新特性，并有详细代码介绍其用法，对关键知识点做了深入分析，对重要的知识点我单独写了相关文章并附上了相关链接，我整理了完备的c++新特性脑图（由于图片太大，我没有放在文章里，同学可以在后台回复消息“新特性”，即可下载完整图片）。

**auto & decltype**

关于C++11新特性，最先提到的肯定是类型推导，C++11引入了auto和decltype关键字，使用他们可以在编译期就推导出变量或者表达式的类型，方便开发者编码也简化了代码。

* auto：让编译器在编译器就推导出变量的类型，可以通过=右边的类型推导出变量的类型。

auto a = 10; // 10是int型，可以自动推导出a是int

复制

* decltype：相对于auto用于推导变量类型，而decltype则用于推导表达式类型，这里只用于编译器分析表达式的类型，表达式实际不会进行运算。

cont int &i = 1;

int a = 2;

decltype(i) b = 2; // b是const int&

复制

关于auto和decltype的详细介绍请看：[一文吃透C++11中auto和decltype知识点](http://mp.weixin.qq.com/s?__biz=MzI3NjA1OTEzMg==&mid=2247483911&idx=1&sn=9808fea93560af81c594e7d7c0f81ed9&chksm=eb7a0494dc0d8d82abfc0db427d921c93eb077e9f9162088b13651e81618bb231cb6f22f96a2&scene=21#wechat_redirect)

**左值右值**

众所周知C++11新增了右值引用，这里涉及到很多概念：

* 左值：可以取地址并且有名字的东西就是左值。
* 右值：不能取地址的没有名字的东西就是右值。
* 纯右值：运算表达式产生的临时变量、不和对象关联的原始字面量、非引用返回的临时变量、lambda表达式等都是纯右值。
* 将亡值：可以理解为即将要销毁的值。
* 左值引用：对左值进行引用的类型。
* 右值引用：对右值进行引用的类型。
* 移动语义：转移资源所有权，类似于转让或者资源窃取的意思，对于那块资源，转为自己所拥有，别人不再拥有也不会再使用。
* 完美转发：可以写一个接受任意实参的函数模板，并转发到其它函数，目标函数会收到与转发函数完全相同的实参。
* 返回值优化：当函数需要返回一个对象实例时候，就会创建一个临时对象并通过复制构造函数将目标对象复制到临时对象，这里有复制构造函数和析构函数会被多余的调用到，有代价，而通过返回值优化，C++标准允许省略调用这些复制构造函数。

这里的详细介绍请看：[左值引用、右值引用、移动语义、完美转发，你知道的不知道的都在这里](http://mp.weixin.qq.com/s?__biz=MzI3NjA1OTEzMg==&mid=2247483921&idx=1&sn=5f5a14ac4db1e12092434bd9c20e44a2&chksm=eb7a0482dc0d8d94df49a59c1b9dce1458e9b385e97a6c53ecb4bea06b24bd47187a1c47cc4c&scene=21#wechat_redirect)

**列表初始化**

在C++11中可以直接在变量名后面加上初始化列表来进行对象的初始化，详细介绍一定要看这篇文章：[学会C++11列表初始化](http://mp.weixin.qq.com/s?__biz=MzI3NjA1OTEzMg==&mid=2247483934&idx=1&sn=26025c7e7d530df870059be967908c6d&chksm=eb7a048ddc0d8d9b42e8954645cf7f60b9f3ba520f0d03adc1863a8456c89f2f0ad4c2645124&scene=21#wechat_redirect)

**std::function & std::bind & lambda表达式**

c++11新增了std::function、std::bind、lambda表达式等封装使函数调用更加方便，详细介绍请看：[搞定c++11新特性std::function和lambda表达式](http://mp.weixin.qq.com/s?__biz=MzI3NjA1OTEzMg==&mid=2247483968&idx=1&sn=3d1e0efb0a0cdc66bbf646fc5d64fcbe&chksm=eb7a04d3dc0d8dc56030fed11c83534c105d71f402f56bbb8ad75cb4b7ea4c5e25a6640c078b&scene=21#wechat_redirect)

**模板的改进**

C++11关于模板有一些细节的改进：

* 模板的右尖括号
* 模板的别名
* 函数模板的默认模板参数

详细介绍请看：[C++11的模板改进](http://mp.weixin.qq.com/s?__biz=MzI3NjA1OTEzMg==&mid=2247483929&idx=1&sn=7a2fac8daf54d12705ae6f36c57771ef&chksm=eb7a048adc0d8d9cd6ba448c6423ae167deca839965cf82ecec2a92d51b9cb325217689b0764&scene=21#wechat_redirect)

**并发**

c++11关于并发引入了好多好东西，有：

* std::thread相关
* std::mutex相关
* std::lock相关
* std::atomic相关
* std::call\_once相关
* volatile相关
* std::condition\_variable相关
* std::future相关
* async相关

详细介绍请看：[c++11新特性之线程相关所有知识点](http://mp.weixin.qq.com/s?__biz=MzI3NjA1OTEzMg==&mid=2247483972&idx=1&sn=de4a41c7c7920241c35e76d25e092350&chksm=eb7a04d7dc0d8dc102d8400cb081f0a23ab231e36c7adff8c24bfc5cbcc73ac7389614167c95&scene=21#wechat_redirect)

这里也使用c++11来实现的线程池和定时器，可以看：

[C++线程池的实现之格式修订版](http://mp.weixin.qq.com/s?__biz=MzI3NjA1OTEzMg==&mid=2247483823&idx=1&sn=732303e36464f0c9fb4bd75fa10ba4db&chksm=eb7a073cdc0d8e2a6dca8babec94ce063ccab0c7832b9d85539357fe2da692fc6ad6cd5b938b&scene=21#wechat_redirect)

[C++定时器的实现之格式修订版](http://mp.weixin.qq.com/s?__biz=MzI3NjA1OTEzMg==&mid=2247483823&idx=2&sn=9d39ed7ef1d3bb6c2984db28aa013a9f&chksm=eb7a073cdc0d8e2a2e24291dc9c71d6c48bd8dd8e9149e91f10b44a75b10034005908de7b78d&scene=21#wechat_redirect)

**智能指针**

很多人谈到c++，说它特别难，可能有一部分就是因为c++的内存管理吧，不像java那样有虚拟机动态的管理内存，在程序运行过程中可能就会出现内存泄漏，然而这种问题其实都可以通过c++11引入的智能指针来解决，相反我还认为这种内存管理还是c++语言的优势，因为尽在掌握。

c++11引入了三种智能指针：

* std::shared\_ptr
* std::weak\_ptr
* std::unique\_ptr

详细介绍请看：[c++11新特性之智能指针](http://mp.weixin.qq.com/s?__biz=MzI3NjA1OTEzMg==&mid=2247483979&idx=1&sn=c815b28d8ffcb49adb7538bf76378f6d&chksm=eb7a04d8dc0d8dce96f262e0b14a74c0bc3a7bf335b202b4f404899489e7e1a325bc0b5b9068&scene=21#wechat_redirect)

**基于范围的for循环**

直接看代码

vector<int> vec;

for (auto iter = vec.begin(); iter != vec.end(); iter++) { // before c++11

cout << \*iter << endl;

}

for (int i : vec) { // c++11基于范围的for循环

cout << "i" << endl;

}

复制

**委托构造函数**

委托构造函数允许在同一个类中一个构造函数调用另外一个构造函数，可以在变量初始化时简化操作，通过代码来感受下委托构造函数的妙处吧：

不使用委托构造函数：

struct A {

A(){}

A(int a) { a\_ = a; }

A(int a, int b) { // 好麻烦

a\_ = a;

b\_ = b;

}

A(int a, int b, int c) { // 好麻烦

a\_ = a;

b\_ = b;

c\_ = c;

}

int a\_;

int b\_;

int c\_;

};

复制

使用委托构造函数：

struct A {

A(){}

A(int a) { a\_ = a; }

A(int a, int b) : A(a) { b\_ = b; }

A(int a, int b, int c) : A(a, b) { c\_ = c; }

int a\_;

int b\_;

int c\_;

};

复制

初始化变量是不是方便了许多。

**继承构造函数**

继承构造函数可以让派生类直接使用基类的构造函数，如果有一个派生类，我们希望派生类采用和基类一样的构造方式，可以直接使用基类的构造函数，而不是再重新写一遍构造函数，老规矩，看代码：

不使用继承构造函数：

struct Base {

Base() {}

Base(int a) { a\_ = a; }

Base(int a, int b) : Base(a) { b\_ = b; }

Base(int a, int b, int c) : Base(a, b) { c\_ = c; }

int a\_;

int b\_;

int c\_;

};

struct Derived : Base {

Derived() {}

Derived(int a) : Base(a) {} // 好麻烦

Derived(int a, int b) : Base(a, b) {} // 好麻烦

Derived(int a, int b, int c) : Base(a, b, c) {} // 好麻烦

};

int main() {

Derived a(1, 2, 3);

return 0;

}

复制

使用继承构造函数：

struct Base {

Base() {}

Base(int a) { a\_ = a; }

Base(int a, int b) : Base(a) { b\_ = b; }

Base(int a, int b, int c) : Base(a, b) { c\_ = c; }

int a\_;

int b\_;

int c\_;

};

struct Derived : Base {

using Base::Base;

};

int main() {

Derived a(1, 2, 3);

return 0;

}

复制

只需要使用using Base::Base继承构造函数，就免去了很多重写代码的麻烦。

**nullptr**

nullptr是c++11用来表示空指针新引入的常量值，在c++中如果表示空指针语义时建议使用nullptr而不要使用NULL，因为NULL本质上是个int型的0，其实不是个指针。举例：

void func(void \*ptr) {

cout << "func ptr" << endl;

}

void func(int i) {

cout << "func i" << endl;

}

int main() {

func(NULL); // 编译失败，会产生二义性

func(nullptr); // 输出func ptr

return 0;

}

复制

**final & override**

c++11关于继承新增了两个关键字，final用于修饰一个类，表示禁止该类进一步派生和虚函数的进一步重载，override用于修饰派生类中的成员函数，标明该函数重写了基类函数，如果一个函数声明了override但父类却没有这个虚函数，编译报错，使用override关键字可以避免开发者在重写基类函数时无意产生的错误。

示例代码1：

struct Base {

virtual void func() {

cout << "base" << endl;

}

};

struct Derived : public Base{

void func() override { // 确保func被重写

cout << "derived" << endl;

}

void fu() override { // error，基类没有fu()，不可以被重写

}

};

复制

示例代码2：

struct Base final {

virtual void func() {

cout << "base" << endl;

}

};

struct Derived : public Base{ // 编译失败，final修饰的类不可以被继承

void func() override {

cout << "derived" << endl;

}

};

复制

**default**

c++11引入default特性，多数时候用于声明构造函数为默认构造函数，如果类中有了自定义的构造函数，编译器就不会隐式生成默认构造函数，如下代码：

struct A {

int a;

A(int i) { a = i; }

};

int main() {

A a; // 编译出错

return 0;

}

复制

上面代码编译出错，因为没有匹配的构造函数，因为编译器没有生成默认构造函数，而通过default，程序员只需在函数声明后加上“=default;”，就可将该函数声明为 defaulted 函数，编译器将为显式声明的 defaulted 函数自动生成函数体，如下：

struct A {

A() = default;

int a;

A(int i) { a = i; }

};

int main() {

A a;

return 0;

}

复制

编译通过。

**delete**

c++中，如果开发人员没有定义特殊成员函数，那么编译器在需要特殊成员函数时候会隐式自动生成一个默认的特殊成员函数，例如拷贝构造函数或者拷贝赋值操作符，如下代码：

struct A {

A() = default;

int a;

A(int i) { a = i; }

};

int main() {

A a1;

A a2 = a1; // 正确，调用编译器隐式生成的默认拷贝构造函数

A a3;

a3 = a1; // 正确，调用编译器隐式生成的默认拷贝赋值操作符

}

复制

而我们有时候想禁止对象的拷贝与赋值，可以使用delete修饰，如下：

struct A {

A() = default;

A(const A&) = delete;

A& operator=(const A&) = delete;

int a;

A(int i) { a = i; }

};

int main() {

A a1;

A a2 = a1; // 错误，拷贝构造函数被禁用

A a3;

a3 = a1; // 错误，拷贝赋值操作符被禁用

}

复制

delele函数在c++11中很常用，std::unique\_ptr就是通过delete修饰来禁止对象的拷贝的。

**explicit**

explicit专用于修饰构造函数，表示只能显式构造，不可以被隐式转换，根据代码看explicit的作用：

不用explicit：

struct A {

A(int value) { // 没有explicit关键字

cout << "value" << endl;

}

};

int main() {

A a = 1; // 可以隐式转换

return 0;

}

复制

使用explicit:

struct A {

explicit A(int value) {

cout << "value" << endl;

}

};

int main() {

A a = 1; // error，不可以隐式转换

A aa(2); // ok

return 0;

}

复制

**const**

因为要讲后面的constexpr，所以这里简单介绍下const。

const字面意思为只读，可用于定义变量，表示变量是只读的，不可以更改，如果更改，编译期间就会报错。

主要用法如下：

1. 用于定义常量，const的修饰的变量不可更改。

const int value = 5;

复制

1. 指针也可以使用const，这里有个小技巧，从右向左读，即可知道const究竟修饰的是指针还是指针所指向的内容。

char \*const ptr; // 指针本身是常量

const char\* ptr; // 指针指向的变量为常量

复制

1. 在函数参数中使用const，一般会传递类对象时会传递一个const的引用或者指针，这样可以避免对象的拷贝，也可以防止对象被修改。

class A{};

void func(const A& a);

复制

1. const修饰类的成员变量，表示是成员常量，不能被修改，可以在初始化列表中被赋值。

class A {

const int value = 5;

};

class B {

const int value;

B(int v) : value(v){}

};

复制

1. 修饰类成员函数，表示在该函数内不可以修改该类的成员变量。

class A{

void func() const;

};

复制

1. 修饰类对象，类对象只能调用该对象的const成员函数。

class A {

void func() const;

};

const A a;

a.func();

复制

**constexpr**

constexpr是c++11新引入的关键字，用于编译时的常量和常量函数，这里直接介绍constexpr和const的区别：

两者都代表可读，const只表示read only的语义，只保证了运行时不可以被修改，但它修饰的仍然有可能是个动态变量，而constexpr修饰的才是真正的常量，它会在编译期间就会被计算出来，整个运行过程中都不可以被改变，constexpr可以用于修饰函数，这个函数的返回值会尽可能在编译期间被计算出来当作一个常量，但是如果编译期间此函数不能被计算出来，那它就会当作一个普通函数被处理。如下代码：

#include<iostream>

using namespace std;

constexpr int func(int i) {

return i + 1;

}

int main() {

int i = 2;

func(i);// 普通函数

func(2);// 编译期间就会被计算出来

}

复制

**enum class**

c++11新增有作用域的枚举类型，看代码

不带作用域的枚举代码：

enum AColor {

kRed,

kGreen,

kBlue

};

enum BColor {

kWhite,

kBlack,

kYellow

};

int main() {

if (kRed == kWhite) {

cout << "red == white" << endl;

}

return 0;

}

复制

如上代码，不带作用域的枚举类型可以自动转换成整形，且不同的枚举可以相互比较，代码中的红色居然可以和白色比较，这都是潜在的难以调试的bug，而这种完全可以通过有作用域的枚举来规避。

有作用域的枚举代码：

enum class AColor {

kRed,

kGreen,

kBlue

};

enum class BColor {

kWhite,

kBlack,

kYellow

};

int main() {

if (AColor::kRed == BColor::kWhite) { // 编译失败

cout << "red == white" << endl;

}

return 0;

}

复制

使用带有作用域的枚举类型后，对不同的枚举进行比较会导致编译失败，消除潜在bug，同时带作用域的枚举类型可以选择底层类型，默认是int，可以改成char等别的类型。

enum class AColor : char {

kRed,

kGreen,

kBlue

};

复制

我们平时编程过程中使用枚举，一定要使用有作用域的枚举取代传统的枚举。

**非受限联合体**

c++11之前union中数据成员的类型不允许有非POD类型，而这个限制在c++11被取消，允许数据成员类型有非POD类型，看代码：

struct A {

int a;

int \*b;

};

union U {

A a; // 非POD类型 c++11之前不可以这样定义联合体

int b;

};

复制

对于什么是POD类型，大家可以自行查下资料，大体上可以理解为对象可以直接memcpy的类型。

**sizeof**

c++11中sizeof可以用的类的数据成员上，看代码：

c++11前：

struct A {

int data[10];

int a;

};

int main() {

A a;

cout << "size " << sizeof(a.data) << endl;

return 0;

}

复制

c++11后：

struct A {

int data[10];

int a;

};

int main() {

cout << "size " << sizeof(A::data) << endl;

return 0;

}

复制

想知道类中数据成员的大小在c++11中是不是方便了许多，而不需要定义一个对象，在计算对象的成员大小。

**assertion**

static\_assert(true/false, message);

复制

c++11引入static\_assert声明，用于在编译期间检查，如果第一个参数值为false，则打印message，编译失败。

**自定义字面量**

c++11可以自定义字面量，我们平时c++中都或多或少使用过chrono中的时间，例如：

std::this\_thread::sleep\_for(std::chrono::milliseconds(100)); // 100ms

std::this\_thread::sleep\_for(std::chrono::seconds(100)); // 100s

复制

其实没必要这么麻烦，也可以这么写：

std::this\_thread::sleep\_for(100ms); // c++14里可以这么使用，这里只是举个自定义字面量使用的例子

std::this\_thread::sleep\_for(100s);

复制

这就是自定义字面量的使用，示例如下：

struct mytype {

unsigned long long value;

};

constexpr mytype operator"" \_mytype ( unsigned long long n ) {

return mytype{n};

}

mytype mm = 123\_mytype;

cout << mm.value << endl;

复制

关于自定义字面量，可以看下chrono的源代码，相信大家会有很大收获，需要源码分析chrono的话，可以留言给我。

**内存对齐**

**什么是内存对齐**

理论上计算机对于任何变量的访问都可以从任意位置开始，然而实际上系统会对这些变量的存放地址有限制，通常将变量首地址设为某个数N的倍数，这就是内存对齐。

**为什么要内存对齐**

1. 硬件平台限制，内存以字节为单位，不同硬件平台不一定支持任何内存地址的存取，一般可能以双字节、4字节等为单位存取内存，为了保证处理器正确存取数据，需要进行内存对齐。
2. 提高CPU内存访问速度，一般处理器的内存存取粒度都是N的整数倍，假如访问N大小的数据，没有进行内存对齐，有可能就需要两次访问才可以读取出数据，而进行内存对齐可以一次性把数据全部读取出来，提高效率。

在c++11之前如果想创建内存对齐需要：

void align\_cpp11\_before()

{

static char data[sizeof(void \*) + sizeof(A)];

const uintptr\_t kAlign = sizeof(void \*) - 1;

char \*align\_ptr =

reinterpret\_cast<char \*>(reinterpret\_cast<uintptr\_t>(data + kAlign) & ~kAlign);

A \*attr = new (align\_ptr) A;

}

复制

c++11关于内存对齐新增了一些函数：

void align\_cpp11\_after()

{

static std::aligned\_storage<sizeof(A),

alignof(A)>::type data;

A \*attr = new (&data) A;

}

复制

还有：alignof()、std::alignment\_of()、alignas()，关于内存对齐详情可以看这篇文章：[内存对齐之格式修订版](http://mp.weixin.qq.com/s?__biz=MzI3NjA1OTEzMg==&mid=2247483981&idx=1&sn=cfda6e84fd77084c75e5048187a1ab28&chksm=eb7a04dedc0d8dc8cb2a26e92f254e23a4538cd2756a1d7f0e8a2aaf99c6e18a42391208276f&scene=21#wechat_redirect)

**thread\_local**

c++11引入thread\_local，用thread\_local修饰的变量具有thread周期，每一个线程都拥有并只拥有一个该变量的独立实例，一般用于需要保证线程安全的函数中。

#include <iostream>

#include <thread>

class A {

public:

A() {}

~A() {}

void test(const std::string &name) {

thread\_local int count = 0;

++count;

std::cout << name << ": " << count << std::endl;

}

};

void func(const std::string &name) {

A a1;

a1.test(name);

a1.test(name);

A a2;

a2.test(name);

a2.test(name);

}

int main() {

std::thread(func, "thread1").join();

std::thread(func, "thread2").join();

return 0;

}

复制

输出：

thread1: 1

thread1: 2

thread1: 3

thread1: 4

thread2: 1

thread2: 2

thread2: 3

thread2: 4

复制

验证上述说法，对于一个线程私有变量，一个线程拥有且只拥有一个该实例，类似于static。

**基础数值类型**

c++11新增了几种数据类型：long long、char16\_t、char32\_t等

**随机数功能**

c++11关于随机数功能则较之前丰富了很多，典型的可以选择概率分布类型，先看如下代码：

#include <time.h>

#include <iostream>

#include <random>

using namespace std;

int main() {

std::default\_random\_engine random(time(nullptr));

std::uniform\_int\_distribution<int> int\_dis(0, 100); // 整数均匀分布

std::uniform\_real\_distribution<float> real\_dis(0.0, 1.0); // 浮点数均匀分布

for (int i = 0; i < 10; ++i) {

cout << int\_dis(random) << ' ';

}

cout << endl;

for (int i = 0; i < 10; ++i) {

cout << real\_dis(random) << ' ';

}

cout << endl;

return 0;

}

复制

输出：

38 100 93 7 66 0 68 99 41 7

0.232202 0.617716 0.959241 0.970859 0.230406 0.430682 0.477359 0.971858 0.0171148 0.64863

复制

代码中举例的是整数均匀分布和浮点数均匀分布，c++11提供的概率分布类型还有好多，例如伯努利分布、正态分布等，具体可以见最后的参考资料。

**正则表达式**

c++11引入了regex库更好的支持正则表达式，见代码：

#include <iostream>

#include <iterator>

#include <regex>

#include <string>

int main() {

std::string s = "I know, I'll use2 regular expressions.";

// 忽略大小写

std::regex self\_regex("REGULAR EXPRESSIONS", std::regex\_constants::icase);

if (std::regex\_search(s, self\_regex)) {

std::cout << "Text contains the phrase 'regular expressions'\n";

}

std::regex word\_regex("(\\w+)"); // 匹配字母数字等字符

auto words\_begin = std::sregex\_iterator(s.begin(), s.end(), word\_regex);

auto words\_end = std::sregex\_iterator();

std::cout << "Found " << std::distance(words\_begin, words\_end) << " words\n";

const int N = 6;

std::cout << "Words longer than " << N << " characters:\n";

for (std::sregex\_iterator i = words\_begin; i != words\_end; ++i) {

std::smatch match = \*i;

std::string match\_str = match.str();

if (match\_str.size() > N) {

std::cout << " " << match\_str << '\n';

}

}

std::regex long\_word\_regex("(\\w{7,})");

// 超过7个字符的单词用[]包围

std::string new\_s = std::regex\_replace(s, long\_word\_regex, "[$&]");

std::cout << new\_s << '\n';

}

复制

**chrono**

c++11关于时间引入了chrono库，源于boost，功能强大，chrono主要有三个点：

* duration
* time\_point
* clocks

**duration**

std::chrono::duration表示一段时间，常见的单位有s、ms等，示例代码：

// 拿休眠一段时间举例，这里表示休眠100ms

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

复制

sleep\_for里面其实就是std::chrono::duration，表示一段时间，实际是这样：

typedef duration<int64\_t, milli> milliseconds;

typedef duration<int64\_t> seconds;

复制

duration具体模板如下：

1 template <class Rep, class Period = ratio<1> > class duration;

复制

Rep表示一种数值类型，用来表示Period的数量，比如int、float、double，Period是ratio类型，用来表示【用秒表示的时间单位】比如second，常用的duration<Rep, Period>已经定义好了，在std::chrono::duration下：

* ratio<3600, 1>：hours
* ratio<60, 1>：minutes
* ratio<1, 1>：seconds
* ratio<1, 1000>：microseconds
* ratio<1, 1000000>：microseconds
* ratio<1, 1000000000>：nanosecons

ratio的具体模板如下：

template <intmax\_t N, intmax\_t D = 1> class ratio;

复制

N代表分子，D代表分母，所以ratio表示一个分数，我们可以自定义Period，比如ratio<2, 1>表示单位时间是2秒。

**time\_point**

表示一个具体时间点，如2020年5月10日10点10分10秒，拿获取当前时间举例：

std::chrono::time\_point<std::chrono::high\_resolution\_clock> Now() {

return std::chrono::high\_resolution\_clock::now();

}

// std::chrono::high\_resolution\_clock为高精度时钟，下面会提到

复制

**clocks**

时钟，chrono里面提供了三种时钟：

* steady\_clock
* system\_clock
* high\_resolution\_clock

**steady\_clock**

稳定的时间间隔，表示相对时间，相对于系统开机启动的时间，无论系统时间如何被更改，后一次调用now()肯定比前一次调用now()的数值大，可用于计时。

**system\_clock**

表示当前的系统时钟，可以用于获取当前时间：

int main() {

using std::chrono::system\_clock;

system\_clock::time\_point today = system\_clock::now();

std::time\_t tt = system\_clock::to\_time\_t(today);

std::cout << "today is: " << ctime(&tt);

return 0;

}

// today is: Sun May 10 09:48:36 2020

复制

**high\_resolution\_clock**

high\_resolution\_clock表示系统可用的最高精度的时钟，实际上就是system\_clock或者steady\_clock其中一种的定义，官方没有说明具体是哪个，不同系统可能不一样，我之前看gcc chrono源码中high\_resolution\_clock是steady\_clock的typedef。

更多关于chrono的介绍可以看下我之前的文章：[RAII妙用之计算函数耗时](http://mp.weixin.qq.com/s?__biz=MzI3NjA1OTEzMg==&mid=2247483852&idx=1&sn=34dde853d5304f8a165e82effe287f3c&chksm=eb7a075fdc0d8e4971f55dc3c4ad4388a3c7466c6df5a47bcfa81510de230cf22065298fee26&scene=21#wechat_redirect)

**新增数据结构**

* std::forward\_list：单向链表，只可以前进，在特定场景下使用，相比于std::list节省了内存，提高了性能

std::forward\_list<int> fl = {1, 2, 3, 4, 5};

for (const auto &elem : fl) {

cout << elem;

}

复制

* std::unordered\_set：基于hash表实现的set，内部不会排序，使用方法和set类似
* std::unordered\_map：基于hash表实现的map，内部不会排序，使用方法和set类似
* std::array：数组，在越界访问时抛出异常，建议使用std::array替代普通的数组
* std::tuple：元组类型，类似pair，但比pair扩展性好

typedef std::tuple<int, double, int, double> Mytuple;

Mytuple t(0, 1, 2, 3);

std::cout << "0 " << std::get<0>(t);

std::cout << "1 " << std::get<1>(t);

std::cout << "2 " << std::get<2>(t);

std::cout << "3 " << std::get<3>(t);

复制

**新增算法**

* all\_of：检测表达式是否对范围[first, last)中所有元素都返回true，如果都满足，则返回true

std::vector<int> v(10, 2);

if (std::all\_of(v.cbegin(), v.cend(), [](int i) { return i % 2 == 0; })) {

std::cout << "All numbers are even\n";

}

复制

* any\_of：检测表达式是否对范围[first, last)中至少一个元素返回true，如果满足，则返回true，否则返回false，用法和上面一样
* none\_of：检测表达式是否对范围[first, last)中所有元素都不返回true，如果都不满足，则返回true，否则返回false，用法和上面一样
* find\_if\_not：找到第一个不符合要求的元素迭代器，和find\_if相反
* copy\_if：复制满足条件的元素
* itoa：对[容器](https://cloud.tencent.com/product/tke?from=10680)内的元素按序递增

std::vector<int> l(10);

std::iota(l.begin(), l.end(), 19); // 19为初始值

for (auto n : l) std::cout << n << ' ';

// 19 20 21 22 23 24 25 26 27 28

复制

* minmax\_element：返回容器内最大元素和最小元素位置

int main() {

std::vector<int> v = {3, 9, 1, 4, 2, 5, 9};

auto result = std::minmax\_element(v.begin(), v.end());

std::cout << "min element at: " << \*(result.first) << '\n';

std::cout << "max element at: " << \*(result.second) << '\n';

return 0;

}

// min element at: 1

// max element at: 9

复制

* is\_sorted、is\_sorted\_until：返回容器内元素是否已经排好序。

关于c++11的新特性基本上就是这些，相信各位看完一定会有所收获。

**参考资料**

https://zh.cppreference.com/w/cpp/language/range-for

https://juejin.im/post/5dcaa857e51d457f7675360b

https://zhuanlan.zhihu.com/p/21930436

https://zh.wikipedia.org/wiki/Nullptr

https://zh.wikipedia.org/wiki/Constexpr

https://zh.cppreference.com/w/cpp/language/enum

https://kheresy.wordpress.com/2019/03/27/using-enum-class/

https://zh.cppreference.com/w/cpp/language/union

http://c.biancheng.net/view/7165.html

https://zhuanlan.zhihu.com/p/77585472

http://www.cplusplus.com/reference/random/

https://zh.cppreference.com/w/cpp/regex

https://www.cnblogs.com/jwk000/p/3560086.html

https://zh.cppreference.com/w/cpp/algorithm/all\_any\_none\_of