

Modern C++ 选讲

鸟x

Modern C++ 选讲

常量

`nullptr`

`constexpr`

变量及其初始化

`if/switch` 变量声明强化

列表初始化 ★

结构化绑定

类型推导 ★

`auto`

`decltype`

尾返回类型推导

`decltype(auto)`

控制流

基于范围的for循环 ★

面向对象

委托构造

继承构造

显式虚函数重载

`override`

`final`

显式禁用默认函数

强类型枚举

Lambda 表达式 ★

基础

值捕获

引用捕获

隐式捕获

表达式捕获

泛型Lambda

函数对象包装器

`std::function` ★

`std::bind` 和 `std::placeholder`

右值引用 ★

左值、右值的纯右值、亡值、右值

右值引用和左值引用

移动语义

完美转发

智能指针与内存管理

RAII 与引用计数

`std::unique_ptr`

`std::shared_ptr`

`std::weak_ptr`

并行与并发

并发基础

互斥量与临界区

期值future

条件变量

杂项

`noexcept`

字面量

原始字符串字面量

自定义字面量

Reference

鸣谢

常量

nullptr

替代NULL。C++11 引入了 `nullptr` 关键字，专门用来区分空指针、`0`。而 `nullptr` 的类型为 `nullptr_t`，能够隐式的转换为任何指针或成员指针的类型。

```
foo(0);           // 调用 foo(int)
// foo(NULL);     // 该行不能通过编译
foo(nullptr);     // 调用 foo(char*)
```

constexpr

显式声明常量表达式。const修饰的变量**只在一定情况下**是常量表达式，这有时可能带来困扰。C++11 提供了 `constexpr` 让用户显式的声明函数或对象构造函数**在编译期**会成为常量表达式。从 C++14 开始，`constexpr` 函数可以在内部使用局部变量、循环和分支等简单语句。

```
const int len_1 = 5; // 常量表达式
const int len_2 = len_1 + 1; // 常量表达式
constexpr int len_2_constexpr = 1 + 2 + 3; // 显式声明的常量表达式

int len = 5;
const int len_3 = len + 1; // 非常量表达式

// 使用了static_assert, 当其第一个参数为常量表达式时才不会报错
static_assert(len_1, "");
static_assert(len_2, "");
static_assert(len_2_constexpr, "");
static_assert(len_3, ""); // 报错, 说明len_3不是常量表达式
```

变量及其初始化

if/switch 变量声明强化

C++17 使得我们可以在 `if`（或 `switch`）中声明一个临时的变量：

```
// 将临时变量放到 if 语句内
if (const std::vector<int>::iterator itr = std::find(vec.begin(), vec.end(), 3);
    itr != vec.end()) {
    *itr = 4;
}
```

列表初始化

C++11 引入了列表初始化，提供了统一的初始化方式。从此，我们可以使用brace-init-list（花括号）方便地进行初始化。

```
#include <vector>

class Foo {
public:
    Foo(int, int){}
```

```
};

int main(){
    int a[] {1, 2, 3}; // 数组
    int b{1}; // 变量
    std::vector<int> v {1, 2}; // stl容器
    Foo foo {1, 2}; // 对象
    return 0;
}
```

此外，C++11引入了 `std::initializer_list`，以支持对类的对象进行列表初始化。其允许构造函数或其他函数像参数一样使用初始化列表，还能将其作为普通函数的形参。在某些情况下，brace-init-list可以被自动推导为 `std::initializer_list`。

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

class Foo {
public:
    Foo(std::initializer_list<int> list) {
        x = 0;
        for (std::initializer_list<int>::iterator it = list.begin();
             it != list.end(); ++it) x+=*it;
        std::cout<<x<<std::endl;
    }
    void foo(std::initializer_list<int> list) {
        for (std::initializer_list<int>::iterator it = list.begin();
             it != list.end(); ++it) std::cout<<*it<<std::endl;
    }
private:
    int x;
};

int main(){
    std::vector<Foo> v;
    v.emplace_back(std::initializer_list<int>{1, 2, 3});
    //v.emplace_back({3, 2, 1}); // 报错, braced-init-list 未能推导为 std::initializer_list
    v[0].foo({1,2,3}); // braced-init-list 推导为 std::initializer_list
    return 0;
}
```

结构化绑定

C++17给出的结构化绑定提供一种简单的方法直接从元组中拿到并定义元组中的元素。此外，还可以绑定数组、结构体。

```
#include <tuple>
#include <array>
#include <string>

struct A{
    int i = 0;
    double j = 4.0;
};

int main(){
    auto [x, y, z] = std::make_tuple(1, 2.3, "456"); // 绑定元组
```

```
std::array<int, 3> arr = {1, 2, 3};
auto [a, b, c] = arr; // 绑定数组

A s;
auto [m, n] = s; // 绑定结构体
return 0;
}
```

类型推导 ★

auto

从 C++11 起, 使用 auto 关键字进行类型推导。

```
class MagicFoo {
public:
    std::vector<int> vec;
    MagicFoo(std::initializer_list<int> list) {
        // 不用再写冗长的迭代器类型名了
        for (auto it = list.begin(); it != list.end(); ++it) {
            vec.push_back(*it);
        }
    }
};

auto i = 5; // i 被推导为 int
auto arr = new auto(10); // arr 被推导为 int *
```

从 C++ 20 起, auto 甚至能用于函数传参。

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

auto i = 5; // 被推导为 int
auto j = 6; // 被推导为 int
```

注意: auto 还不能用于推导数组类型:

```
auto auto_arr2[10] = {arr}; // 错误, 无法推导数组元素类型
```

decltype

decltype 关键字是为了解决 auto 关键字只能对变量进行类型推导的缺陷而出现的, 可推导表达式的类型。

```
auto x = 1;
auto y = 2;
decltype(x+y) z; // z的类型是int
```

std::is_same<T, U> 用于判断 T 和 U 这两个类型是否相等。

```
if (std::is_same<decltype(x), int>::value) // 为真
    std::cout << "type x == int" << std::endl;
if (std::is_same<decltype(x), float>::value) // 为假
    std::cout << "type x == float" << std::endl;
```

尾返回类型推导

C++11 引入了一个尾返回类型（trailing return type），利用 `auto` 关键字将返回类型后置。C++14 开始可以直接让普通函数具备返回值推导。

```
// after c++11
template<typename T, typename U>
auto add2(T x, U y) -> decltype(x+y){
    return x + y;
}

// after c++14
template<typename T, typename U>
auto add3(T x, U y){
    return x + y;
}
```

decltype(auto)

C++14 引入的 `decltype(auto)` 主要用于对转发函数或封装的返回类型进行推导，它使我们无需显式指定 `decltype` 的参数表达式。

```
std::string lookup1();
std::string& lookup2();

// C++11
std::string look_up_a_string_1() {
    return lookup1();
}
std::string& look_up_a_string_2() {
    return lookup2();
}

// after C++14
decltype(auto) look_up_a_string_1() {
    return lookup1();
}
decltype(auto) look_up_a_string_2() {
    return lookup2();
}
```

控制流

基于范围的for循环 ★

C++11 引入了基于范围的循环写法。

```
std::vector<int> vec = {1, 2, 3, 4};
for (auto element : vec)
    std::cout << element << std::endl; // 不会改变vec的元素，用于读
for (auto &element : vec)
    element += 1; // 可以改变vec的元素，用于写
```

面向对象

委托构造

C++11 引入了委托构造的概念，这使得构造函数可以在同一个类中一个构造函数调用另一个构造函数。

```
#include <iostream>

class Base {
public:
    int value1;
    int value2;
    Base() {
        value1 = 1;
    }
    Base(int value) : Base() { // 委托 Base() 构造函数
        value2 = value;
    }
};

Base b(2);
```

继承构造

C++11 利用关键字 `using` 引入了继承构造函数。

```
#include <iostream>

class Base {
public:
    int value1;
    int value2;
    Base() {
        value1 = 1;
    }
    Base(int value) : Base() { // 委托 Base() 构造函数
        value2 = value;
    }
};

class Subclass : public Base {
public:
    using Base::Base; // 继承构造
};
```

显式虚函数重载

C++11 引入了 `override` 和 `final` 这两个关键字来防止**意外重载虚函数**和基类的虚函数被删除后**子类的对应函数变为普通方法**的情况发生。

override

当重载虚函数时，引入 `override` 关键字将显式的告知编译器进行重载，编译器将检查基函数是否存在这样的虚函数，否则将无法通过编译：

```
struct Base {
    virtual void foo(int);
};

struct Subclass: Base {
    virtual void foo(int) override; // 合法
    virtual void foo(float) override; // 非法，父类没有此虚函数
};
```

final

`final` 则是为了防止类被继续继承以及终止虚函数继续重载引入的。

```
struct Base {
    virtual void foo() final;
};
struct SubClass1 final: Base {
}; // 合法

struct SubClass2 : SubClass1 {
}; // 非法, SubClass1 已 final

struct SubClass3: Base {
    void foo(); // 非法, foo 已 final
};
```

显式禁用默认函数

C++11允许显式的声明采用或拒绝编译器默认生成的函数。

```
class Magic {
public:
    Magic() = default; // 显式声明使用编译器生成的构造
    Magic& operator=(const Magic&) = delete; // 显式声明拒绝编译器生成默认赋值函数
    Magic(int magic_number);
}
```

强类型枚举

C++11引入了枚举类 (enumeration class)，并使用 `enum class` 的语法进行声明。枚举类实现了类型安全，首先他不能够被隐式的转换为整数，同时也不能够将其与整数数字进行比较，更不可能对不同的枚举类型的枚举值进行比较。希望获得枚举值的值时，必须**显式**的进行类型转换。

```
#include <iostream>
enum class new_enum : unsigned int {
    value1,
    value2,
    value3 = 100,
    value4 = 100
};

int main(){
    if (new_enum::value3 == new_enum::value4) {
        std::cout << "new_enum::value3 == new_enum::value4" << std::endl;
    }
    std::cout<<(int)new_enum::value4;
    return 0;
}
```

Lambda 表达式

Lambda 表达式是现代 C++ 中最重要的特性之一，而 Lambda 表达式，实际上就是提供了一个类似匿名函数的特性，而匿名函数则是在需要一个函数，但是又不想费力去命名一个函数的情况下去使用的。

基础

Lambda 表达式的基本语法如下：

```
[捕获列表] (参数列表) mutable(可选) 异常属性 -> 返回类型 {  
    // 函数体  
}
```

所谓捕获列表，其实可以理解为参数的一种类型，Lambda 表达式内部函数体在默认情况下是不能够使用函数体外部的变量的，这时候捕获列表可以起到**传递外部数据**的作用。根据传递的行为，捕获列表也分为以下几种。

值捕获

与参数传值类似，值捕获的前提是变量可以拷贝，不同之处则在于，被捕获的变量在 Lambda 表达式被创建时拷贝，而非调用时才拷贝。

```
int value = 1;  
auto copy_value = [value] {  
    return value;  
};  
value = 100;  
auto stored_value = copy_value();  
// stored_value == 1, 而 value == 100.  
// 因为 copy_value 在创建时就保存了一份 value 的拷贝
```

引用捕获

与引用传参类似，引用捕获保存的是引用，值会发生变化。

```
int value = 1;  
auto copy_value = [&value] {  
    return value;  
};  
value = 100;  
auto stored_value = copy_value();  
// 这时, stored_value == 100, value == 100.  
// 因为 copy_value 保存的是引用
```

隐式捕获

可以在捕获列表中写一个 `&` 或 `=` 向编译器声明采用引用捕获或者值捕获。

```
int value = 1;  
auto copy_value = [&] {  
    return value;  
};
```

捕获列表常用的四种形式：

- `[]` 空捕获列表
- `[name1, name2, ...]` 捕获一系列变量
- `[&]` 引用捕获, 让编译器自行推导引用列表
- `[=]` 值捕获, 让编译器自行推导值捕获列表

表达式捕获

上面提到的值捕获、引用捕获都是已经在外层作用域声明的变量，因此这些捕获方式捕获的均为左值，而不能捕获右值。C++14 允许捕获的成员用任意的表达式进行初始化，这就允许了右值的捕获，被声明的捕获变量类型会根据表达式进行判断，判断方式与使用 `auto` 本质上是相同的。


```
#include <memory> // std::make_unique
#include <utility> // std::move, 将important转换为右值

void lambda_expression_capture() {
    auto important = std::make_unique<int>(1);
    auto add = [v1 = 1, v2 = std::move(important)](int x, int y) -> int {
        return x+y+v1+(*v2);
    };
    std::cout << add(3,4) << std::endl;
}
```

泛型Lambda


从 C++14 开始，Lambda 函数的形式参数可以使用 `auto` 关键字来自动推导参数类型。

```
auto add = [](auto x, auto y) {
    return x+y;
};

add(1, 2);
add(1.1, 2.2);
```

函数对象包装器

这部分内容虽然属于标准库的一部分，但是从本质上来看，它却增强了 C++ 语言运行时的能力。

`std::function` 

Lambda 表达式的本质是一个和函数对象类型相似的类类型（称为闭包类型）的对象（称为闭包对象）。当 Lambda 表达式的捕获列表为空时，闭包对象还能够转换为函数指针值进行传递：

```
using foo = void(int); // 定义函数类型
void functional(foo f) { // 定义在参数列表中的函数类型 foo 被视为退化后的函数指针类型 foo*
    f(1); // 通过函数指针调用函数
}

auto f = [](int value) {
    std::cout << value << std::endl;
}; // f 是闭包对象
functional(f); // 传递闭包对象，隐式转换为 foo* 类型的函数指针值
f(1); // lambda 表达式调用
```

上面的代码给出了两种不同的调用形式，一种是将 Lambda 作为函数类型传递进行调用，而另一种则是直接调用 Lambda 表达式，在 C++11 中，统一了这些概念，将能够被调用的对象的类型，统一称之为可调用类型。

C++11 `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，可以理解为**函数的容器**。

```
#include <functional>

int foo(int para) {
    return para;
}

// std::function 包装了一个返回值为 int，参数为 int 的函数
std::function<int(int)> func = foo;

int important = 10;
```

```
std::function<int(int)> func2 = [&](int value) -> int {
    return 1+value+important;
};
func(10);
func2(10);
```

`std::bind` 和 `std::placeholder`

我们有时候可能并不一定能够一次性获得调用某个函数的全部参数，通过 `std::bind` 可以将部分调用参数提前绑定到函数身上成为一个新的对象，然后在参数齐全后，完成调用。

```
int foo(int a, int b, int c) {
    return a+b+c;
}

// 将参数1,2绑定到函数 foo 上, 但是使用 std::placeholders::_1 来对第一个参数进行占位
auto bindFoo = std::bind(foo, std::placeholders::_1, 1, 2);
// 这时调用 bindFoo 时, 只需要提供第一个参数即可
bindFoo(1);
```

C++14之后，lambda表达式可以完全替代 `std::bind`，详见<https://stackoverflow.com/questions/17363003/why-use-stdbind-over-lambdas-in-c14>，此处不再多叙。

右值引用 ★

右值引用是 C++11 引入的与 Lambda 表达式齐名的重要特性之一。它的引入解决了 C++ 中大量的历史遗留问题，消除了诸如 `std::vector`、`std::string` 之类的额外开销，也才使得函数对象容器 `std::function` 成为了可能。

左值、右值的纯右值、亡值、右值

左值(lvalue, left value)，顾名思义就是赋值符号左边的值。准确来说，左值是表达式（不一定是赋值表达式）后依然存在的持久对象。（注意：字符串字面量为左值，如"hello"）

右值(rvalue, right value)，右边的值，是指表达式结束后就不再存在的临时对象（或引用）。

而 C++11 中为了引入强大的右值引用，将右值的概念进行了进一步的划分，分为：纯右值、将亡值。

纯右值(prvalue, pure rvalue)，纯粹的右值，要么是纯粹的字面量，例如 `10`，`true`；要么是临时对象，例如 `-1`，`1+2`，`a++`的结果。非引用返回的临时变量、运算表达式产生的临时变量、Lambda 表达式都属于纯右值。

亡值(xvalue, expiring value)，是 C++11 为了引入右值引用而提出的概念（因此在传统 C++ 中，只有右值这一个概念），也就是即将被销毁、却能够被移动（move）的值。

右值引用和左值引用

顾名思义，右值引用可以引用一切右值（包括纯右值和亡值）：`T &&`，其中 `T` 为非左值引用类型（若 `T` 为左值引用类型，则结果仍是左值引用）。**右值引用的声明让这个临时对象的生命周期得以延长，只要右值引用生命期还未结束，那么这个临时对象的生命期也不会结束。**

C++11 提供了 `std::move` 这个方法将左值参数无条件的转换为右值，有了它我们就能够方便的获得一个右值引用（匿名右值引用）。

```
#include <iostream>
#include <utility> // std::move
void reference(std::string& str) {
    std::cout << "左值" << std::endl;
}
void reference(std::string&& str) {
    std::cout << "右值" << std::endl;
}
int main(){
```

```

std::string lv0 = "string,"; // lv0 是一个左值

std::string& lv1 = lv0; // lv1 是左值引用

// std::string&& r1 = lv0; // 非法，右值引用不能引用左值
std::string&& rv1 = std::move(lv0); // 合法，std::move可以将左值转换为右值

const std::string& lv2 = lv0 + lv0; // 合法，常量左值引用能够延长临时变量的生命周期
// lv2 += "Test"; // 非法，常量引用无法被修改

std::string&& rv2 = lv0 + lv2; // 合法，右值引用延长临时对象生命周期
rv2 += "Test"; // 合法，非常量引用能够修改临时变量

reference(lv1); // lv1是左值引用，为左值
reference(rv2); // rv2是一个具名右值引用，为左值，见完美转发
reference(std::move(lv0)); // 匿名右值引用，为右值
return 0;
}

```

移动语义

右值引用的出现恰好解决了传统C++没有区分『移动』和『拷贝』的概念的问题。

```

#include <iostream>
class A {
public:
    int *pointer;
    A():pointer(new int(1)) {
        std::cout << "构造" << pointer << std::endl;
    }
    A(A& a):pointer(new int(*a.pointer)) {
        std::cout << "拷贝" << pointer << std::endl;
    } // 无意义的对象拷贝
    A(A&& a):pointer(a.pointer) {
        a.pointer = nullptr;
        std::cout << "移动" << pointer << std::endl;
    }
    ~A(){
        std::cout << "析构" << pointer << std::endl;
        delete pointer;
    }
};

A return_rvalue(bool test) {
    A a,b;
    if(test) return a; // 等价于 static_cast<A&&>(a);
    else return b;     // 等价于 static_cast<A&&>(b);
}

int main(){
    A obj = return_rvalue(false);
    // 输出:
    // 构造0xa9eeb0
    // 构造0xa9fee0
    // 移动0xa9fee0
    // 析构0
    // 析构0xa9eeb0
    // 析构0xa9fee0
    return 0;
}

```

在上面的代码中：

1. 首先会在 `return_rvalue` 内部构造两个 `A` 对象，于是获得两个构造函数的输出；
2. 函数返回后，产生一个亡值，被 `A` 的移动构造（`A(A&&)`）引用，从而延长生命周期，并将这个右值中的指针拿到，保存到了 `obj` 中，而亡值的指针被设置为 `nullptr`。

下面是涉及标准库的例子，使用右值引用避免无意义拷贝以提升性能。

```
std::string str = "Hello world.";
std::vector<std::string> v;

// 将使用 push_back(const T&), 即产生拷贝行为
v.push_back(str);
// 将输出 "str: Hello world."
std::cout << "str: " << str << std::endl;

// 将使用 push_back(const T&&), 不会出现拷贝行为
// 而整个字符串会被移动到 vector 中, 所以有时候 std::move 会用来减少拷贝出现的开销
// 这步操作后, str 中的值会变为空
v.push_back(std::move(str));
// 将输出 "str: "
std::cout << "str: " << str << std::endl;
```

完美转发

一个具名右值引用其实是一个左值。这就为我们进行参数转发（传递）造成了问题：

```
void reference(int& v) {
    std::cout << "左值" << std::endl;
}

void reference(int&& v) {
    std::cout << "右值" << std::endl;
}

template <typename T>
void pass(T&& v) {
    reference(v); // v是具名右值引用, 为左值, 故始终调用 reference(int&)
}

std::cout << "传递右值:" << std::endl;
pass(1); // 1是右值, 但输出是左值

std::cout << "传递左值:" << std::endl;
int l = 1;
pass(l); // l 是左值, 输出左值
```

为了解决这个问题，我们应该使用 `std::forward` 来进行参数的转发（传递）：

```
#include <iostream>
#include <utility>

void reference(int& v) {
    std::cout << "左值引用" << std::endl;
}

void reference(int&& v) {
    std::cout << "右值引用" << std::endl;
}

template <typename T>
void pass(T&& v) { // v是左值
    reference(v); // 普通传参, 永远输出左值引用
    reference(std::move(v)); // std::move 传参, 永远输出右值引用
}
```

```

reference(std::forward<T>(v)); // std::forward 传参
reference(static_cast<T&&>(v)); // static_cast<T&&> 传参
}
int main(){
    int a = 1;
    pass(1);
    pass(a);
    return 0;
}

```

`std::forward` 不会造成任何多余的拷贝，同时**完美转发**函数的实参给内部调用的其他函数：

当实参是右值（int、或者int的引用），T被推导为int，T&&是int&&；

当实参是左值（int、或者int的引用），T被推导为int&，T&&是int&。

`std::forward` 和 `std::move` 一样，只是类型转换，`std::move` 单纯的将左值转化为右值，`std::forward` 也只是单纯的将参数做了一个类型的转换，从现象上来看，`std::forward<T>(v)` 和 `static_cast<T&&>(v)` 是完全一样的。

智能指针与内存管理

RAII 与引用计数

引用计数这种计数是为了防止内存泄露而产生的。基本想法是对于动态分配的对象，进行引用计数，每当增加一次对一个对象的引用，那么引用对象的引用计数就会增加一次，每删除一次引用，引用计数就会减一，当一个对象的引用计数减为零时，就自动删除指向的堆内存。

在传统 C++ 中，『记得』手动释放资源，总不是最佳实践。因为我们很有可能就忘记了去释放资源而导致泄露。所以通常的做法是对于一个对象而言，我们在构造函数的时候申请空间，而在析构函数（在离开作用域时调用）的时候释放空间，也就是我们常说的 RAII 资源获取即初始化技术。

C++11 引入智能指针的概念，让程序员不再需要关心手动释放内存。使用它们需要包含头文件 `<memory>`。

`std::unique_ptr`

`std::unique_ptr` 是一种独占的智能指针，它禁止其他智能指针与其共享同一个对象，从而保证代码的安全。

```

std::unique_ptr<int> pointer = std::make_unique<int>(10); // make_unique 从 C++14 引入
std::unique_ptr<int> pointer2 = pointer; // 非法

```

`make_unique` 并不复杂，C++11 没有提供 `std::make_unique`，可以自行实现：

```

template<typename T, typename ...Args>
std::unique_ptr<T> make_unique( Args&& ...args ) {
    return std::unique_ptr<T>( new T( std::forward<Args>(args)... ) );
}

```

既然是独占，换句话说就是不可复制。但是，我们可以利用 `std::move` 将其转移给其他的 `unique_ptr`。

```

#include <memory>

struct Foo {
    Foo() { std::cout << "Foo::Foo" << std::endl; }
    ~Foo() { std::cout << "Foo::~~Foo" << std::endl; }
    void foo() { std::cout << "Foo::foo" << std::endl; }
};

void f(const Foo &) {
    std::cout << "f(const Foo&)" << std::endl;
}

```

```

int main() {
    std::unique_ptr<Foo> p1(std::make_unique<Foo>());
    // p1 不空, 输出
    if (p1) p1->foo();
    {
        std::unique_ptr<Foo> p2(std::move(p1));
        // p2 不空, 输出
        f(*p2);
        // p2 不空, 输出
        if(p2) p2->foo();
        // p1 为空, 无输出
        if(p1) p1->foo();
        p1 = std::move(p2);
        // p2 为空, 无输出
        if(p2) p2->foo();
        std::cout << "p2 被销毁" << std::endl;
    }
    // p1 不空, 输出
    if (p1) p1->foo();
    // Foo 的实例会在离开作用域时被销毁
}

```

此外, 由于独占, `std::unique_ptr` 不会有引用计数的开销, 因此常常是首选。

`std::shared_ptr`

`std::shared_ptr` 是一种智能指针, 它能够记录多少个 `shared_ptr` 共同指向一个对象, 从而消除显式的调用 `delete`, 当引用计数变为零的时候就会将对象自动删除。

但还不够, 因为使用 `std::shared_ptr` 仍然需要使用 `new` 来调用, 这使得代码出现了某种程度上的不对称。

`std::make_shared` 就能够用来消除显式的使用 `new`, 会分配创建传入参数中的对象, 并返回这个对象类型的 `std::shared_ptr` 指针。

```

#include <iostream>
#include <memory>
void foo(std::shared_ptr<int> i) {
    (*i)++;
}
int main() {
    // Constructed a std::shared_ptr
    auto pointer = std::make_shared<int>(10);
    foo(pointer);
    std::cout << *pointer << std::endl; // 11
    // The shared_ptr will be destructed before leaving the scope
    return 0;
}

```

`std::shared_ptr` 可以通过 `get()` 方法来获取原始指针, 通过 `reset()` 来减少一个引用计数, 并通过 `use_count()` 来查看一个对象的引用计数。

```

auto pointer = std::make_shared<int>(10);
auto pointer2 = pointer; // 引用计数+1
auto pointer3 = pointer; // 引用计数+1
int *p = pointer.get(); // 这样不会增加引用计数
std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl; // 3
std::cout << "pointer2.use_count() = " << pointer2.use_count() << std::endl; // 3
std::cout << "pointer3.use_count() = " << pointer3.use_count() << std::endl; // 3

```

```

pointer2.reset();
std::cout << "reset pointer2:" << std::endl;
std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl; // 2
std::cout << "pointer2.use_count() = " << pointer2.use_count() << std::endl; // 0, pointer2 已
reset
std::cout << "pointer3.use_count() = " << pointer3.use_count() << std::endl; // 2
pointer3.reset();
std::cout << "reset pointer3:" << std::endl;
std::cout << "pointer.use_count() = " << pointer.use_count() << std::endl; // 1
std::cout << "pointer2.use_count() = " << pointer2.use_count() << std::endl; // 0
std::cout << "pointer3.use_count() = " << pointer3.use_count() << std::endl; // 0, pointer3 已
reset

```

`std::weak_ptr`

`std::shared_ptr` 引入了引用成环的问题。

```

struct A;
struct B;

struct A {
    std::shared_ptr<B> pointer;
    ~A() {
        std::cout << "A 被销毁" << std::endl;
    }
};

struct B {
    std::shared_ptr<A> pointer;
    ~B() {
        std::cout << "B 被销毁" << std::endl;
    }
};

int main() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->pointer = b;
    b->pointer = a;
    // 结果是A和B依然不能被销毁，因为二者间形成了一个环，导致两个shared_ptr引用计数都为1
}

```

解决这个问题的办法就是使用弱引用指针 `std::weak_ptr`，`std::weak_ptr` 是一种弱引用（相比较而言 `std::shared_ptr` 就是一种强引用）。弱引用不会引起引用计数增加。

对于上面的代码，将A或B中的任意一个 `std::shared_ptr` 改为 `std::weak_ptr` 即可解决问题。

```

struct A {
    std::shared_ptr<B> pointer;
    ~A() {
        std::cout << "A 被销毁" << std::endl;
    }
};

struct B {
    std::weak_ptr<A> pointer;
    ~B() {
        std::cout << "B 被销毁" << std::endl;
    }
};

// 将B的智能指针改为weak_ptr，这样程序结束时A的引用计数就会变成0而销毁，B的引用计数随之变为0而销毁

```

`std::weak_ptr` 没有 `*` 运算符和 `->` 运算符，所以不能够对资源进行操作，它可以用于检查 `std::shared_ptr` 是否存在，其 `expired()` 方法能在资源未被释放时，会返回 `false`，否则返回 `true`；除此之外，它也可以用于获取指向原始对象的 `std::shared_ptr` 指针，其 `lock()` 方法在原始对象未被释放时，返回一个指向原始对象的 `std::shared_ptr` 指针，进而访问原始对象的资源，否则返回默认构造的 `std::shared_ptr`（即未托管任何指针）。

```
std::weak_ptr<int> b;
{
    auto a = std::make_shared<int>(1);
    b = a;
    if(auto c = b.lock()) { // 输出
        std::cout << *c << std::endl;
    }
}
if(auto c = b.expired()) { // 输出
    std::cout << "b is expired" << std::endl;
}
```

并行与并发

并发基础

`std::thread` 用于创建一个执行的线程实例，所以它是一切并发编程的基础，使用时需要包含 `<thread>` 头文件，它提供了很多基本的线程操作，例如 `get_id()` 来获取所创建线程的线程 ID，使用 `join()` 来加入一个线程等等。

```
#include <thread>

int main() {
    std::thread t([](){
        std::cout << "hello world." << std::endl;
    });
    t.join();
    return 0;
}
```

互斥量与临界区

互斥量 `mutex` 是并发技术中的核心之一。C++11 引入了 `mutex` 相关的类，其所有相关的函数都放在 `<mutex>` 头文件中。

`std::mutex` 是 C++11 中最基本的 `mutex` 类，通过实例化 `std::mutex` 可以创建互斥量，而通过其成员函数 `lock()` 可以进行上锁，`unlock()` 可以进行解锁。但是在实际编写代码的过程中，最好不去直接调用成员函数，因为调用成员函数就需要在每个临界区的出口处调用 `unlock()`，当然，还包括异常。这时候 C++11 还为互斥量提供了一个 RAII 语法的模板类 `std::lock_guard`。RAII 在不失代码简洁性的同时，很好的保证了代码的异常安全性。

在 RAII 用法下，对于临界区的互斥量的创建只需要在作用域的开始部分。


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

int v = 1;

void critical_section(int change_v) {
    static std::mutex mtx;
    std::lock_guard<std::mutex> lock(mtx);
    // 执行竞争操作
    v = change_v;
    // 离开此作用域后 mtx 会被释放
}
```

这样的代码也是异常安全的。无论临界区正常返回、还是在中途抛出异常，都会自动调用 `unlock()`。

而 `std::unique_lock` 则相对于 `std::lock_guard` 出现的，`std::unique_lock` 更加灵活，`std::unique_lock` 的对象会以独占所有权（没有其他的 `unique_lock` 对象同时拥有某个 `mutex` 对象的所有权）的方式管理 `mutex` 对象上的上锁和解锁的操作。所以在并发编程中，推荐使用 `std::unique_lock`。

`std::lock_guard` 不能显式的调用 `lock` 和 `unlock`，而 `std::unique_lock` 可以在声明后的任意位置调用，可以缩小锁的作用范围，提供更高的并发度。

如果你用到了条件变量 `std::condition_variable::wait` 则必须使用 `std::unique_lock` 作为参数。

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

int v = 1;

void critical_section(int change_v) {
    static std::mutex mtx;
    std::unique_lock<std::mutex> lock(mtx);
    // 执行竞争操作
    v = change_v;
    // 将锁进行释放
    lock.unlock();

    // 在此期间，任何人都可以抢夺 v 的持有权

    // 开始另一组竞争操作，再次加锁
    lock.lock();
    v += 1;
    // 离开此作用域后 mtx 会被释放
}
```

期值future

如果我们的主线程 A 希望新开辟一个线程 B 去执行某个我们预期的任务，并返回一个结果。而这时候，线程 A 可能正在忙其他的事情，无暇顾及 B 的结果，所以我们会很自然的希望能够在某个特定的时间获得线程 B 的结果。

在 C++11 的 `std::future` 被引入之前，通常的做法是：创建一个线程 A，在线程 A 里启动任务 B，当准备完毕后发送一个事件，并将结果保存在全局变量中。而主函数线程 A 里正在做其他的事情，当需要结果的时候，调用一个线程等待函数来获得执行的结果。

而 C++11 提供的 `std::future` 简化了这个流程，可以用来获取异步任务的结果。自然地，我们很容易能够想象到把它作为一种简单的线程同步手段，即屏障（barrier）。

标准库中的 `std::async` 可在其中调用的函数执行完成前就返回其 `std::future` 对象，对该对象使用 `get()` 即可阻塞程序到函数执行完成时取得返回值：

```

#include <iostream>
#include <future>

int main(){
    auto result = std::async([](){return 7;});
    std::cout << "waiting..." << std::endl;
    std::cout << "future result is " << result.get() << std::endl;
    return 0;
}

```

条件变量

条件变量 `std::condition_variable` 是为解决死锁而生。比如，线程可能需要等待某个条件为真才能继续执行，而一个忙等待循环中可能会导致所有其他线程都无法进入临界区使得条件为真时，就会发生死锁。所以，`condition_variable` 实例被创建出现主要就是用于唤醒等待线程从而避免死锁。`std::condition_variable` 的 `notify_one()` 用于唤醒一个线程；`notify_all()` 则是通知所有线程。下面是一个生产者和消费者模型的例子。

```

#include <queue>
#include <chrono>
#include <mutex>
#include <thread>
#include <iostream>
#include <condition_variable>

int main() {
    std::queue<int> produced_nums;
    std::mutex mtx;
    std::condition_variable cv;

    // 生产者
    auto producer = [&]() {
        for (int i = 0; ; i++) {
            std::this_thread::sleep_for(std::chrono::milliseconds(900)); // 生产时间
            {
                std::unique_lock<std::mutex> lock(mtx);
                // 生产1个
                std::cout << "producing " << i << std::endl;
                produced_nums.push(i);

                cv.notify_one();
            }
        }
    };

    // 消费者
    auto consumer = [&]() {
        while (true) {
            std::this_thread::sleep_for(std::chrono::milliseconds(1000)); // 消费慢于生产
            {
                std::unique_lock<std::mutex> lock(mtx);
                while (produced_nums.empty()) { // 避免虚假唤醒
                    cv.wait(lock);
                }
                // 等价写法 cv.wait(lock, [&]{ return !produced_nums.empty(); });
                // 消费1个
                std::cout << "consuming " << produced_nums.front() << std::endl;
                produced_nums.pop();
            }
        }
    };
}

```

```

    }
}

};

// 分别在不同的线程中运行
std::thread p(producer);
std::thread cs[2];
for (int i = 0; i < 2; ++i) {
    cs[i] = std::thread(consumer);
}
p.join();
for (int i = 0; i < 2; ++i) {
    cs[i].join();
}
return 0;
}

```

杂项

noexcept

C++11 将异常的声明简化为以下两种情况：

1. 函数可能抛出任何异常
2. 函数不能抛出任何异常

并使用 `noexcept` 对这两种行为进行限制，例如：

```

void may_throw(); // 可能抛出异常
void no_throw() noexcept; // 不可能抛出异常

```

使用 `noexcept` 修饰过的函数如果抛出异常，编译器会使用 `std::terminate()` 来立即终止程序运行。

`noexcept` 还能够做操作符，用于操作一个表达式，当表达式无异常时，返回 `true`，否则返回 `false`。

```

#include <iostream>
void may_throw() {
    throw true;
}
auto non_block_throw = []{
    may_throw();
};
void no_throw() noexcept {
    return;
}

auto block_throw = []() noexcept {
    no_throw();
};
int main()
{
    std::cout << std::boolalpha
        << "may_throw() noexcept? " << noexcept(may_throw()) << std::endl
        << "no_throw() noexcept? " << noexcept(no_throw()) << std::endl
        << "!may_throw() noexcept? " << noexcept(non_block_throw()) << std::endl
        << "!no_throw() noexcept? " << noexcept(block_throw()) << std::endl;
    return 0;
}
try {

```

```

    may_throw();
} catch (...) {
    std::cout << "捕获异常, 来自 may_throw()" << std::endl;
}
try {
    non_block_throw();
} catch (...) {
    std::cout << "捕获异常, 来自 non_block_throw()" << std::endl;
}
try {
    block_throw();
} catch (...) {
    std::cout << "捕获异常, 来自 block_throw()" << std::endl;
}
// output
// 捕获异常, 来自 may_throw()
// 捕获异常, 来自 non_block_throw()

```

字面量

原始字符串字面量

C++11 提供了原始字符串字面量的写法, 可以在一个字符串前方使用 `R` 来修饰这个字符串, 同时, 将原始字符串使用括号包裹。

```

#include <iostream>
#include <string>

int main() {
    std::string str = R"(C:\File\To\Path)";
    std::cout << str << std::endl;
    return 0;
}

```

自定义字面量

C++11 引进了自定义字面量的能力, 通过重载双引号后缀运算符实现。

```

// 字符串字面量自定义必须设置如下的参数列表
std::string operator"" _wow1(const char *wow1, size_t len) {
    return std::string(wow1)+"woooooooooow, amazing";
}

std::string operator"" _wow2 (unsigned long long i) {
    return std::to_string(i)+"woooooooooow, amazing";
}

int main() {
    auto str = "abc"_wow1;
    auto num = 1_wow2;
    std::cout << str << std::endl;
    std::cout << num << std::endl;
    return 0;
}

```

自定义字面量支持四种字面量:

1. 整型字面量: 重载时必须使用 `unsigned long long`、`const char *`、模板字面量算符参数, 在上面的代码中使用的是前者;

2. 浮点型字面量：重载时必须使用 `long double`、`const char *`、模板字面量算符；
3. 字符串字面量：必须使用 `(const char *, size_t)` 形式的参数表；
4. 字符字面量：参数只能是 `char`，`wchar_t`，`char16_t`，`char32_t` 这几种类型。

Reference

[欧长坤 Modern C++ Tutorial](#)（内含很多错误0.0）（如果欲深入学习建议另寻他道）

鸣谢

刘雪枫学长通读了讲义初版，纠正了其中**大量**错误。