现代 C++ 教程：高速上手 C++11/14/17/20

欧长坤 (hi[at]changkun.de)

最后更新 2024 年 5 月 21 日- af45678

注意

此 PDF 的内容可能过期，请检查本书网站以及 GitHub 仓库以获取最新内容。

版权声明

目录

序言

引言

C++ 是一个用户群体相当大的语言。从 C++98 的出现到 C++11 的正式定稿经历了长达十年多之久的积累。C++14/17 则是作为对 C++11 的重要补充和优化，C++20 则将这门语言领进了现代化的大门，所有这些新标准中扩充的特性，给 C++ 这门语言注入了新的活力。那些还在坚持使用**传统 C++**（本书把 C++98 及其之前的 C++ 特性均称之为传统 C++）而未接触过现代 C++ 的 C++ 程序员在见到诸如 Lambda 表达式这类全新特性时，甚至会流露出『学的不是同一门语言』的惊叹之情。

现代 C++ (本书中均指 C++11/14/17/20) 为传统 C++ 注入的大量特性使得整个 C++ 变得更加像一门现代化的语言。现代 C++ 不仅仅增强了 C++ 语言自身的可用性，`auto` 关键字语义的修改使得我们更加有信心来操控极度复杂的模板类型。同时还对语言运行期进行了大量的强化，Lambda 表达式的出现让 C++ 具有了『匿名函数』的『闭包』特性，而这一特性几乎在现代的编程语言（诸如 Python/Swift/...）中已经司空见惯，右值引用的出现解决了 C++ 长期以来被人诟病的临时对象效率问题等等。

C++17 则是近三年依赖 C++ 社区一致推进的方向，也指出了 **现代 C++** 编程的一个重要发展方向。尽管它的出现并不如 C++11 的分量之重，但它包含了大量小而美的语言与特性（例如结构化绑定），这些特性的出现再一次修正了我们在 C++ 中的编程范式。

现代 C++ 还为自身的标准库增加了非常多的工具和方法，诸如在语言自身标准的层面上制定了 `std::thread`，从而支持了并发编程，在不同平台上不再依赖于系统底层的 API，实现了语言层面的跨平台支持；`std::regex` 提供了完整的正则表达式支持等等。C++98 已经被实践证明了是一种非常成功的『范型』，而现代 C++ 的出现，则进一步推动这种范型，让 C++ 成为系统程序设计和库开发更好的语言。Concept 提供了对模板参数编译期的检查，进一步增强了语言整体的可用性。

总而言之，我们作为 C++ 的拥护与实践者，始终保持接纳新事物的开放心态，才能更快的推进 C++ 的发展，使得这门古老而又新颖的语言更加充满活力。

目标读者

1. 本书假定读者已经熟悉了传统 C++ ，至少在阅读传统 C++ 代码上不具备任何困难。换句话说，那些长期使用传统 C++ 进行编码的人、渴望在短时间内迅速了解**现代 C++** 特性的人非常适合阅读本书；
2. 本书一定程度上介绍了一些现代 C++ 的**黑魔法**，但这些魔法毕竟有限，不适合希望进阶学习现代 C++ 的读者，本书的定位系**现代 C++ 的快速上手**。当然，希望进阶学习的读者可以使用本书来回顾并检验自己对 **现代 C++** 的熟悉度。

本书目的

本书号称『高速上手』，从内容上对二十一世纪二十年代之前产生 C++ 的相关特性做了非常相对全面的介绍，读者可以自行根据下面的目录选取感兴趣的内容进行学习，快速熟悉需要了解的内容。这些特性并不需要全部掌握，只需针对自己的使用需求和特定的应用场景，学习、查阅最适合自己的新特性

即可。

同时，本书在介绍这些特性的过程中，尽可能简单明了的介绍了这些特性产生的历史背景和技术需求，这为理解这些特性、运用这些特性提供了很大的帮助。

此外，笔者希望读者在阅读本书后，能够努力在新项目中直接使用 C++17，并努力将旧项目逐步迁移到 C++17。也算是笔者为推进现代 C++ 的普及贡献了一些绵薄之力。

相关代码

本书每章中都出现了大量的代码，如果你在跟随本书介绍特性的思路编写自己的代码遇到问题时，不妨读一读随书附上的源码，你可以在这里中找到书中介绍过的全部的源码，所有代码按章节组织，文件夹名称为章节序号。

随书习题

本书每章最后还加入了少量难度极小的习题，仅用于检验你是否能混合运用当前章节中的知识点。你可以在这里找到习题的答案，文件夹名称为章节序号。

第 1 章迈向现代 C++

编译环境：本书将使用 `clang++` 作为唯一使用的编译器，同时总是在代码中使用 `-std=c++2a` 编译标志。

```
> clang++ -v
Apple LLVM version 10.0.1 (clang-1001.0.46.4)
Target: x86_64-apple-darwin18.6.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

1.1 被弃用的特性

在学习现代 C++ 之前，我们先了解一下从 C++11 开始，被弃用的主要特性：

> **注意**：弃用并非彻底不能用，只是用于暗示程序员这些特性将从未来的标准中消失，应该尽量避免使用。但是，已弃用的特性依然是标准库的一部分，并且出于兼容性的考虑，大部分特性其实会『永久』保留。

- **不再允许字符串字面值常量赋值给一个 `char *`。如果需要用字符串字面值常量赋值和初始化一个 `char *`，应该使用 `const char *` 或者 `auto`。**

  ```
  char *str = "hello world!"; // 将出现弃用警告
  ```

- **C++98 异常说明、`unexpected_handler`、`set_unexpected()` 等相关特性被弃用，应该使用 `noexcept`。**
- **`auto_ptr` 被弃用，应使用 `unique_ptr`。**
- **`register` 关键字被弃用，可以使用但不再具备任何实际含义。**
- **`bool` 类型的 `++` 操作被弃用。**
- **如果一个类有析构函数，为其生成拷贝构造函数和拷贝赋值运算符的特性被弃用了。**
- **C 语言风格的类型转换被弃用（即在变量前使用 `(convert_type)`），应该使用 `static_cast`、`reinterpret_cast`、`const_cast` 来进行类型转换。**
- **特别地，在最新的 C++17 标准中弃用了一些可以使用的 C 标准库，例如 `<ccomplex>`、`<cstdalign>`、`<cstdbool>` 与 `<ctgmath>` 等**
- ……等等

还有一些其他诸如参数绑定（C++11 提供了 `std::bind` 和 `std::function`）、`export` 等特性也均被弃用。前面提到的这些特性**如果你从未使用或者听说过，也请不要尝试去了解他们，应该向新标准靠拢**，直接学习新特性。毕竟，技术是向前发展的。

1.2 与 C 的兼容性

出于一些不可抗力、历史原因，我们不得不在 C++ 中使用一些 C 语言代码（甚至古老的 C 语言代码），例如 Linux 系统调用。在现代 C++ 出现之前，大部分人当谈及『C 与 C++ 的区别是什么』时，普遍除了回答面向对象的类特性、泛型编程的模板特性外，就没有其他的看法了，甚至直接回答『差不多』，也是大有人在。图 1.2 中的韦恩图大致上回答了 C 和 C++ 相关的兼容情况。

从现在开始，你的脑子里应该树立『**C++ 不是 C 的一个超集**』这个观念（而且从一开始就不是，后面的进一步阅读的参考文献中给出了 C++98 和 C99 之间的区别）。在编写 C++ 时，也应该尽可能的避免使用诸如 `void*` 之类的程序风格。而在不得不使用 C 时，应该注意使用 `extern "C"` 这种特性，将 C 语言的代码与 C++ 代码进行分离编译，再统一链接这种做法，例如：

```
// foo.h
#ifdef __cplusplus
extern "C" {
#endif

int add(int x, int y);

#ifdef __cplusplus
}
#endif
```

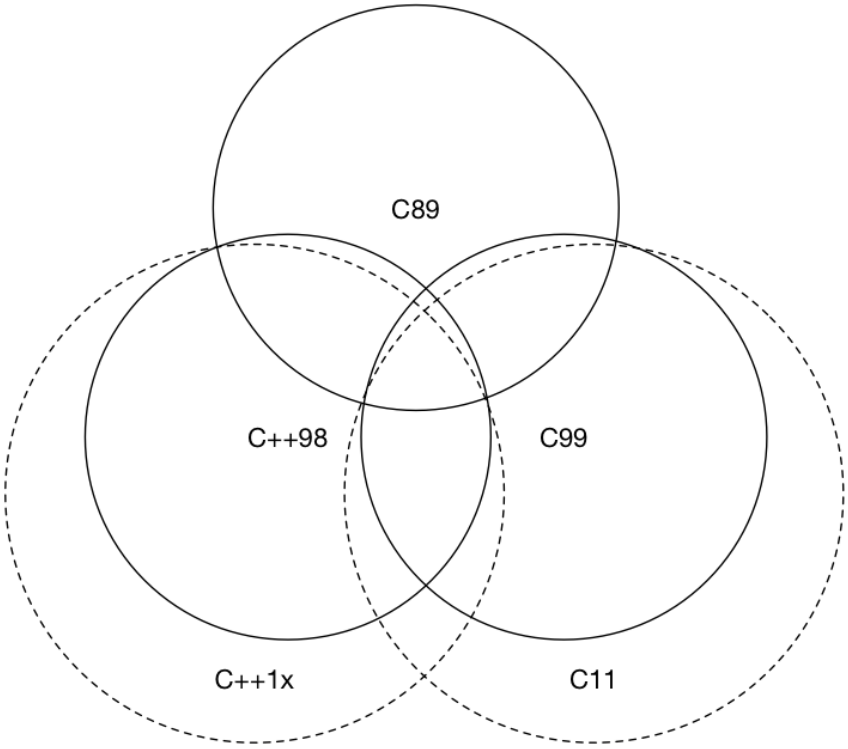


图 1: 图 1.2: C 和 C++ 互相兼容情况

```
// foo.c
int add(int x, int y) {
    return x+y;
}

// 1.1.cpp
#include "foo.h"
#include <iostream>
#include <functional>

int main() {
    [out = std::ref(std::cout << "Result from C code: " << add(1, 2))](){
        out.get() << ".\n";
    }();
    return 0;
}
```

应先使用 gcc 编译 C 语言的代码：

```
gcc -c foo.c
```

编译出 foo.o 文件，再使用 clang++ 将 C++ 代码和 .o 文件链接起来（或者都编译为 .o 再统一链接）：

```
clang++ 1.1.cpp foo.o -std=c++2a -o 1.1
```

当然，你可以使用 Makefile 来编译上面的代码：

```
C = gcc
CXX = clang++

SOURCE_C = foo.c
OBJECTS_C = foo.o

SOURCE_CXX = 1.1.cpp

TARGET = 1.1
LDFLAGS_COMMON = -std=c++2a

all:
    $(C) -c $(SOURCE_C)
    $(CXX) $(SOURCE_CXX) $(OBJECTS_C) $(LDFLAGS_COMMON) -o $(TARGET)
clean:
    rm -rf *.o $(TARGET)
```

> 注意：Makefile 中的缩进是制表符而不是空格符，如果你直接复制这段代码到你的编辑器中，制表符可能会被自动替换掉，请自行确保在 Makefile 中的缩进是由制表符完成的。
>
> 如果你还不知道 Makefile 的使用也没有关系，本教程中不会构建过于复杂的代码，简单的在命令行中使用 clang++ -std=c++2a 也可以阅读本书。

如果你是首次接触现代 C++，那么你很可能还看不懂上面的那一小段代码，即：

```
[out = std::ref(std::cout << "Result from C code: " << add(1, 2))](){
    out.get() << ".\n";
}();
```

不必担心，本书的后续章节将为你介绍这一切。

进一步阅读的参考文献

- C++ 语言导学. Bjarne Stroustrup
- C++ 历史
- C++ 特性在 GCC/Clang 等编译器中的支持情况
- C++98 与 C99 之间的区别

第 2 章语言可用性的强化

当我们声明、定义一个变量或者常量，对代码进行流程控制、面向对象的功能、模板编程等这些都是运行时之前，可能发生在编写代码或编译器编译代码时的行为。为此，我们通常谈及**语言可用性**，是指那些发生在运行时之前的语言行为。

2.1 常量

nullptr

nullptr 出现的目的是为了替代 NULL。C 与 C++ 语言中有**空指针常量**，它们能被隐式转换成任何指针类型的空指针值，或 C++ 中的任何成员指针类型的空成员指针值。NULL 由标准库实现提供，并被定义为实现定义的空指针常量。在 C 中，有些标准库会把 NULL 定义为 ((void*)0) 而有些将它定义为 0。

C++ **不允许**直接将 void * 隐式转换到其他类型，从而 ((void*)0) 不是 NULL 的合法实现。如果标准库尝试把 NULL 定义为 ((void*)0)，那么下面这句代码中会出现编译错误：

```
char *ch = NULL;
```

没有了 void * 隐式转换的 C++ 只好将 NULL 定义为 0。而这依然会产生新的问题，将 NULL 定义成 0 将导致 C++ 中重载特性发生混乱。考虑下面这两个 foo 函数：

```
void foo(char*);
void foo(int);
```

那么 foo(NULL); 这个语句将会去调用 foo(int)，从而导致代码违反直觉。

为了解决这个问题，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，也能和他们进行相等或者不等的比较。

你可以尝试使用 clang++ 编译下面的代码：

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

void foo(char *);
void foo(int);

int main() {
    if (std::is_same<decltype(NULL), decltype(0)>::value)
        std::cout << "NULL == 0" << std::endl;
    if (std::is_same<decltype(NULL), decltype((void*)0)>::value)
        std::cout << "NULL == (void *)0" << std::endl;
```

```
    if (std::is_same<decltype(NULL), std::nullptr_t>::value)
        std::cout << "NULL == nullptr" << std::endl;

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

void foo(char *) {
    std::cout << "foo(char*) is called" << std::endl;
}
void foo(int i) {
    std::cout << "foo(int) is called" << std::endl;
}
```

将输出：

```
foo(int) is called
foo(char*) is called
```

从输出中我们可以看出，NULL 不同于 0 与 nullptr。所以，请养成直接使用 nullptr 的习惯。

此外，在上面的代码中，我们使用了 decltype 和 std::is_same 这两个属于现代 C++ 的语法，简单来说，decltype 用于类型推导，而 std::is_same 用于比较两个类型是否相同，我们会在后面decltype一节中详细讨论。

constexpr

C++ 本身已经具备了常量表达式的概念，比如 1+2, 3*4 这种表达式总是会产生相同的结果并且没有任何副作用。如果编译器能够在编译时就把这些表达式直接优化并植入到程序运行时，将能增加程序的性能。一个非常明显的例子就是在数组的定义阶段：

```
#include <iostream>
#define LEN 10

int len_foo() {
    int i = 2;
    return i;
}
constexpr int len_foo_constexpr() {
    return 5;
}
```

```
constexpr int fibonacci(const int n) {
    return n == 1 || n == 2 ? 1 : fibonacci(n-1)+fibonacci(n-2);
}

int main() {
    char arr_1[10];                     // 合法
    char arr_2[LEN];                    // 合法

    int len = 10;
    // char arr_3[len];                 // 非法

    const int len_2 = len + 1;
    constexpr int len_2_constexpr = 1 + 2 + 3;
    // char arr_4[len_2];               // 非法
    char arr_4[len_2_constexpr];        // 合法

    // char arr_5[len_foo()+5];         // 非法
    char arr_6[len_foo_constexpr() + 1]; // 合法

    std::cout << fibonacci(10) << std::endl;
    // 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
    std::cout << fibonacci(10) << std::endl;
    return 0;
}
```

上面的例子中，char arr_4[len_2] 可能比较令人困惑，因为 len_2 已经被定义为了常量。为什么 char arr_4[len_2] 仍然是非法的呢？这是因为 C++ 标准中数组的长度必须是一个常量表达式，而对于 len_2 而言，这是一个 const 常数，而不是一个常量表达式，因此（即便这种行为在大部分编译器中都支持，但是）它是一个非法的行为，我们需要使用接下来即将介绍的 C++11 引入的 constexpr 特性来解决这个问题；而对于 arr_5 来说，C++98 之前的编译器无法得知 len_foo() 在运行期实际上是返回一个常数，这也就导致了非法的产生。

> 注意，现在大部分编译器其实都带有自身编译优化，很多非法行为在编译器优化的加持下会变得合法，若需重现编译报错的现象需要使用老版本的编译器。

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，这个关键字明确的告诉编译器应该去验证 len_foo 在编译期就应该是一个常量表达式。

此外，constexpr 修饰的函数可以使用递归：

```
constexpr int fibonacci(const int n) {
    return n == 1 || n == 2 ? 1 : fibonacci(n-1)+fibonacci(n-2);
```

```
}
```

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，例如下面的代码在 C++11 的标准下是不能够通过编译的：

```
constexpr int fibonacci(const int n) {
    if(n == 1) return 1;
    if(n == 2) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

为此，我们可以写出下面这类简化的版本来使得函数从 C++11 开始即可用：

```
constexpr int fibonacci(const int n) {
    return n == 1 || n == 2 ? 1 : fibonacci(n-1) + fibonacci(n-2);
}
```

2.2 变量及其初始化

if/switch 变量声明强化

在传统 C++ 中，变量的声明虽然能够位于任何位置，甚至于 for 语句内能够声明一个临时变量 int，但始终没有办法在 if 和 switch 语句中声明一个临时的变量。例如：

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4};

    // 在 c++17 之前
    const std::vector<int>::iterator itr = std::find(vec.begin(), vec.end(), 2);
    if (itr != vec.end()) {
        *itr = 3;
    }

    // 需要重新定义一个新的变量
    const std::vector<int>::iterator itr2 = std::find(vec.begin(), vec.end(), 3);
    if (itr2 != vec.end()) {
        *itr2 = 4;
    }
```

```
    // 将输出 1, 4, 3, 4
    for (std::vector<int>::iterator element = vec.begin(); element != vec.end();
        ++element)
        std::cout << *element << std::endl;
}
```

在上面的代码中，我们可以看到 itr 这一变量是定义在整个 main() 的作用域内的，这导致当我们需要再次遍历整个 std::vector 时，需要重新命名另一个变量。C++17 消除了这一限制，使得我们可以在 if（或 switch）中完成这一操作：

```
// 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;
}
```

怎么样，是不是和 Go 语言很像？

初始化列表

初始化是一个非常重要的语言特性，最常见的就是在对象进行初始化时进行使用。在传统 C++ 中，不同的对象有着不同的初始化方法，例如普通数组、POD （**P**lain **O**ld **D**ata，即没有构造、析构和虚函数的类或结构体）类型都可以使用 {} 进行初始化，也就是我们所说的初始化列表。而对于类对象的初始化，要么需要通过拷贝构造、要么就需要使用 () 进行。这些不同方法都针对各自对象，不能通用。例如：

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

class Foo {
public:
    int value_a;
    int value_b;
    Foo(int a, int b) : value_a(a), value_b(b) {}
};

int main() {
    // before C++11
    int arr[3] = {1, 2, 3};
    Foo foo(1, 2);
    std::vector<int> vec = {1, 2, 3, 4, 5};
```

```
    std::cout << "arr[0]: " << arr[0] << std::endl;
    std::cout << "foo:" << foo.value_a << ", " << foo.value_b << std::endl;
    for (std::vector<int>::iterator it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << std::endl;
    }
    return 0;
}
```

为解决这个问题，C++11 首先把初始化列表的概念绑定到类型上，称其为 `std::initializer_list`，允许构造函数或其他函数像参数一样使用初始化列表，这就为类对象的初始化与普通数组和 POD 的初始化方法提供了统一的桥梁，例如：

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

class MagicFoo {
public:
    std::vector<int> vec;
    MagicFoo(std::initializer_list<int> list) {
        for (std::initializer_list<int>::iterator it = list.begin();
             it != list.end(); ++it)
            vec.push_back(*it);
    }
};
int main() {
    // after C++11
    MagicFoo magicFoo = {1, 2, 3, 4, 5};

    std::cout << "magicFoo: ";
    for (std::vector<int>::iterator it = magicFoo.vec.begin();
        it != magicFoo.vec.end(); ++it)
        std::cout << *it << std::endl;
}
```

这种构造函数被叫做初始化列表构造函数，具有这种构造函数的类型将在初始化时被特殊关照。

初始化列表除了用在对象构造上，还能将其作为普通函数的形参，例如：

```
public:
    void foo(std::initializer_list<int> list) {
        for (std::initializer_list<int>::iterator it = list.begin();
            it != list.end(); ++it) vec.push_back(*it);
```

```
    }

magicFoo.foo({6,7,8,9});
```

其次，C++11 还提供了统一的语法来初始化任意的对象，例如：

```
Foo foo2 {3, 4};
```

结构化绑定

结构化绑定提供了类似其他语言中提供的多返回值的功能。在容器一章中，我们会学到 C++11 新增了 `std::tuple` 容器用于构造一个元组，进而囊括多个返回值。但缺陷是，C++11/14 并没有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，尽管我们可以使用 `std::tie` 对元组进行拆包，但我们依然必须非常清楚这个元组包含多少个对象，各个对象是什么类型，非常麻烦。

C++17 完善了这一设定，给出的结构化绑定可以让我们写出这样的代码：

```
#include <iostream>
#include <tuple>

std::tuple<int, double, std::string> f() {
    return std::make_tuple(1, 2.3, "456");
}

int main() {
    auto [x, y, z] = f();
    std::cout << x << ", " << y << ", " << z << std::endl;
    return 0;
}
```

关于 `auto` 类型推导会在auto 类型推导一节中进行介绍。

2.3 类型推导

在传统 C 和 C++ 中，参数的类型都必须明确定义，这其实对我们快速进行编码没有任何帮助，尤其是当我们面对一大堆复杂的模板类型时，必须明确的指出变量的类型才能进行后续的编码，这不仅拖慢我们的开发效率，也让代码变得又臭又长。

C++11 引入了 `auto` 和 `decltype` 这两个关键字实现了类型推导，让编译器来操心变量的类型。这使得 C++ 也具有了和其他现代编程语言一样，某种意义上提供了无需操心变量类型的使用习惯。

auto

auto 在很早以前就已经进入了 C++，但是他始终作为一个存储类型的指示符存在，与 register 并存。在传统 C++ 中，如果一个变量没有声明为 register 变量，将自动被视为一个 auto 变量。而随着 register 被弃用（在 C++17 中作为保留关键字，以后使用，目前不具备实际意义），对 auto 的语义变更也就非常自然了。

使用 auto 进行类型推导的一个最为常见而且显著的例子就是迭代器。你应该在前面的小节里看到了传统 C++ 中冗长的迭代写法：

```
// 在 C++11 之前
// 由于 cbegin() 将返回 vector<int>::const_iterator
// 所以 it 也应该是 vector<int>::const_iterator 类型
for(vector<int>::const_iterator it = vec.cbegin(); it != vec.cend(); ++it)
```

而有了 auto 之后可以：

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

class MagicFoo {
public:
    std::vector<int> vec;
    MagicFoo(std::initializer_list<int> list) {
        // 从 C++11 起，使用 auto 关键字进行类型推导
        for (auto it = list.begin(); it != list.end(); ++it) {
            vec.push_back(*it);
        }
    }
};
int main() {
    MagicFoo magicFoo = {1, 2, 3, 4, 5};
    std::cout << "magicFoo: ";
    for (auto it = magicFoo.vec.begin(); it != magicFoo.vec.end(); ++it) {
        std::cout << *it << ", ";
    }
    std::cout << std::endl;
    return 0;
}
```

一些其他的常见用法：

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

从 C++ 14 起，auto 能用于 lambda 表达式中的函数传参，而 C++ 20 起该功能推广到了一般的函数。考虑下面的例子：

```
auto add14 = [](auto x, auto y) -> int {
    return x+y;
}

int add20(auto x, auto y) {
    return x+y;
}

auto i = 5; // type int
auto j = 6; // type int
std::cout << add14(i, j) << std::endl;
std::cout << add20(i, j) << std::endl;
```

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
```

```
2.6.auto.cpp:30:19: error: 'auto_arr2' declared as array of 'auto'
    auto auto_arr2[10] = {arr};
```

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。它的用法和 typeof 很相似：

```
decltype(表达式)
```

有时候，我们可能需要计算某个表达式的类型，例如：

```
auto x = 1;
auto y = 2;
decltype(x+y) z;
```

你已经在前面的例子中看到 decltype 用于推断类型的用法，下面这个例子就是判断上面的变量 x，y，z 是否是同一类型：

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

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

```
type x == int
type z == type x
```

尾返回类型推导

你可能会思考，auto 能不能用于推导函数的返回类型呢？还是考虑一个加法函数的例子，在传统 C++ 中我们必须这么写：

```
template<typename R, typename T, typename U>
R add(T x, U y) {
    return x+y;
}
```

> 注意：typename 和 class 在模板参数列表中没有区别，在 typename 这个关键字出现之前，都是使用 class 来定义模板参数的。但在模板中定义有嵌套依赖类型的变量时，需要用 typename 消除歧义

这样的代码其实变得很丑陋，因为程序员在使用这个模板函数的时候，必须明确指出返回类型。但事实上我们并不知道 add() 这个函数会做什么样的操作，以及获得一个什么样的返回类型。

在 C++11 中这个问题得到解决。虽然你可能马上会反应出来使用 decltype 推导 x+y 的类型，写出这样的代码：

```
decltype(x+y) add(T x, U y)
```

但事实上这样的写法并不能通过编译。这是因为在编译器读到 decltype(x+y) 时，x 和 y 尚未被定义。为了解决这个问题，C++11 还引入了一个叫做尾返回类型（trailing return type），利用 auto 关键字将返回类型后置：

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

令人欣慰的是从 C++14 开始是可以直接让普通函数具备返回值推导，因此下面的写法变得合法：

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

可以检查一下类型推导是否正确：

```
// after c++11
auto w = add2<int, double>(1, 2.0);
if (std::is_same<decltype(w), double>::value) {
    std::cout << "w is double: ";
}
std::cout << w << std::endl;

// after c++14
auto q = add3<double, int>(1.0, 2);
std::cout << "q: " << q << std::endl;
```

decltype(auto)

decltype(auto) 是 C++14 开始提供的一个略微复杂的用法。

> 要理解它你需要知道 C++ 中参数转发的概念，我们会在语言运行时强化一章中详细介绍，你可以到时再回来看这一小节的内容。

简单来说，decltype(auto) 主要用于对转发函数或封装的返回类型进行推导，它使我们无需显式的指定 decltype 的参数表达式。考虑看下面的例子，当我们需要对下面两个函数进行封装时：

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

在 C++11 中，封装实现是如下形式：

```
std::string look_up_a_string_1() {
    return lookup1();
}
std::string& look_up_a_string_2() {
    return lookup2();
}
```

而有了 decltype(auto)，我们可以让编译器完成这一件烦人的参数转发：

```
decltype(auto) look_up_a_string_1() {
    return lookup1();
}
decltype(auto) look_up_a_string_2() {
    return lookup2();
}
```

2.4 控制流

if constexpr

正如本章开头出，我们知道了 C++11 引入了 constexpr 关键字，它将表达式或函数编译为常量结果。一个很自然的想法是，如果我们把这一特性引入到条件判断中去，让代码在编译时就完成分支判断，岂不是能让程序效率更高？C++17 将 constexpr 这个关键字引入到 if 语句中，允许在代码中声明常量表达式的判断条件，考虑下面的代码：

```
#include <iostream>

template<typename T>
auto print_type_info(const T& t) {
    if constexpr (std::is_integral<T>::value) {
        return t + 1;
    } else {
        return t + 0.001;
    }
}
int main() {
    std::cout << print_type_info(5) << std::endl;
    std::cout << print_type_info(3.14) << std::endl;
}
```

在编译时，实际代码就会表现为如下：

```
int print_type_info(const int& t) {
    return t + 1;
}
double print_type_info(const double& t) {
    return t + 0.001;
}
int main() {
    std::cout << print_type_info(5) << std::endl;
```

```
    std::cout << print_type_info(3.14) << std::endl;
}
```

区间 for 迭代

终于，C++11 引入了基于范围的迭代写法，我们拥有了能够写出像 Python 一样简洁的循环语句，我们可以进一步简化前面的例子：

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

int main() {
    std::vector<int> vec = {1, 2, 3, 4};
    if (auto itr = std::find(vec.begin(), vec.end(), 3); itr != vec.end()) *itr = 4;
    for (auto element : vec)
        std::cout << element << std::endl; // read only
    for (auto &element : vec) {
        element += 1;                      // writeable
    }
    for (auto element : vec)
        std::cout << element << std::endl; // read only
}
```

2.5 模板

C++ 的模板一直是这门语言的一种特殊的艺术，模板甚至可以独立作为一门新的语言来进行使用。模板的哲学在于将一切能够在编译期处理的问题丢到编译期进行处理，仅在运行时处理那些最核心的动态服务，进而大幅优化运行期的性能。因此模板也被很多人视作 C++ 的黑魔法之一。

外部模板

传统 C++ 中，模板只有在使用时才会被编译器实例化。换句话说，只要在每个编译单元（文件）中编译的代码中遇到了被完整定义的模板，都会实例化。这就产生了重复实例化而导致的编译时间的增加。并且，我们没有办法通知编译器不要触发模板的实例化。

为此，C++11 引入了外部模板，扩充了原来的强制编译器在特定位置实例化模板的语法，使我们能够显式的通知编译器何时进行模板的实例化：

```
template class std::vector<bool>;          // 强行实例化
extern template class std::vector<double>; // 不在该当前编译文件中实例化模板
```

尖括号“>”

在传统 C++ 的编译器中，>> 一律被当做右移运算符来进行处理。但实际上我们很容易就写出了嵌套模板的代码：

```
std::vector<std::vector<int>> matrix;
```

这在传统 C++ 编译器下是不能够被编译的，而 C++11 开始，连续的右尖括号将变得合法，并且能够顺利通过编译。甚至于像下面这种写法都能够通过编译：

```
template<bool T>
class MagicType {
    bool magic = T;
};

// in main function:
std::vector<MagicType<(1>2)>> magic; // 合法，但不建议写出这样的代码
```

类型别名模板

在了解类型别名模板之前，需要理解『模板』和『类型』之间的不同。仔细体会这句话：**模板是用来产生类型的**。在传统 C++ 中，typedef 可以为类型定义一个新的名称，但是却没有办法为模板定义一个新的名称。因为，模板不是类型。例如：

```
template<typename T, typename U>
class MagicType {
public:
    T dark;
    U magic;
};

// 不合法
template<typename T>
typedef MagicType<std::vector<T>, std::string> FakeDarkMagic;
```

C++11 使用 using 引入了下面这种形式的写法，并且同时支持对传统 typedef 相同的功效：

> 通常我们使用 typedef 定义别名的语法是：typedef 原名称 新名称;，但是对函数指针等别名的定义语法却不相同，这通常给直接阅读造成了一定程度的困难。

```
typedef int (*process)(void *);
using NewProcess = int(*)(void *);
```

```
template<typename T>
using TrueDarkMagic = MagicType<std::vector<T>, std::string>;

int main() {
    TrueDarkMagic<bool> you;
}
```

变长参数模板

模板一直是 C++ 所独有的**黑魔法**（一起念：**Dark Magic**）之一。在 C++11 之前，无论是类模板还是函数模板，都只能按其指定的样子，接受一组固定数量的模板参数；而 C++11 加入了新的表示方法，允许任意个数、任意类别的模板参数，同时也不需要在定义时将参数的个数固定。

```
template<typename... Ts> class Magic;
```

模板类 Magic 的对象，能够接受不受限制个数的 typename 作为模板的形式参数，例如下面的定义：

```
class Magic<int,
            std::vector<int>,
            std::map<std::string,
            std::vector<int>>> darkMagic;
```

既然是任意形式，所以个数为 0 的模板参数也是可以的：`class Magic<> nothing;`。

如果不希望产生的模板参数个数为 0，可以手动的定义至少一个模板参数：

```
template<typename Require, typename... Args> class Magic;
```

变长参数模板也能被直接调整到到模板函数上。传统 C 中的 `printf` 函数，虽然也能达成不定个数的形参的调用，但其并非类别安全。而 C++11 除了能定义类别安全的变长参数函数外，还可以使类似 `printf` 的函数能自然地处理非自带类别的对象。除了在模板参数中能使用 `...` 表示不定长模板参数外，函数参数也使用同样的表示法代表不定长参数，这也就为我们简单编写变长参数函数提供了便捷的手段，例如：

```
template<typename... Args> void printf(const std::string &str, Args... args);
```

那么我们定义了变长的模板参数，如何对参数进行解包呢?

首先，我们可以使用 `sizeof...` 来计算参数的个数，：

```
template<typename... Ts>
void magic(Ts... args) {
    std::cout << sizeof...(args) << std::endl;
}
```

我们可以传递任意个参数给 magic 函数：

```
magic(); // 输出 0
magic(1); // 输出 1
magic(1, ""); // 输出 2
```

其次，对参数进行解包，到目前为止还没有一种简单的方法能够处理参数包，但有两种经典的处理手法：

1. 递归模板函数

递归是非常容易想到的一种手段，也是最经典的处理方法。这种方法不断递归地向函数传递模板参数，进而达到递归遍历所有模板参数的目的：

```
#include <iostream>
template<typename T0>
void printf1(T0 value) {
    std::cout << value << std::endl;
}
template<typename T, typename... Ts>
void printf1(T value, Ts... args) {
    std::cout << value << std::endl;
    printf1(args...);
}
int main() {
    printf1(1, 2, "123", 1.1);
    return 0;
}
```

2. 变参模板展开

你应该感受到了这很繁琐，在 C++17 中增加了变参模板展开的支持，于是你可以在一个函数中完成 printf 的编写：

```
template<typename T0, typename... T>
void printf2(T0 t0, T... t) {
    std::cout << t0 << std::endl;
    if constexpr (sizeof...(t) > 0) printf2(t...);
}
```

事实上，有时候我们虽然使用了变参模板，却不一定需要对参数做逐个遍历，我们可以利用 std::bind 及完美转发等特性实现对函数和参数的绑定，从而达到成功调用的目的。

3. 初始化列表展开

递归模板函数是一种标准的做法，但缺点显而易见的在于必须定义一个终止递归的函数。

这里介绍一种使用初始化列表展开的黑魔法：

```
template<typename T, typename... Ts>
auto printf3(T value, Ts... args) {
    std::cout << value << std::endl;
    (void) std::initializer_list<T>{([&args] {
        std::cout << args << std::endl;
    }(), value)...};
}
```

在这个代码中，额外使用了 C++11 中提供的初始化列表以及 Lambda 表达式的特性（下一节中将提到）。

通过初始化列表，`(lambda 表达式, value)...` 将会被展开。由于逗号表达式的出现，首先会执行前面的 lambda 表达式，完成参数的输出。为了避免编译器警告，我们可以将 `std::initializer_list` 显式的转为 `void`。

折叠表达式

C++ 17 中将变长参数这种特性进一步带给了表达式，考虑下面这个例子：

```
#include <iostream>
template<typename ... T>
auto sum(T ... t) {
    return (t + ...);
}
int main() {
    std::cout << sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) << std::endl;
}
```

非类型模板参数推导

前面我们主要提及的是模板参数的一种形式：类型模板参数。

```
template <typename T, typename U>
auto add(T t, U u) {
    return t+u;
}
```

其中模板的参数 `T` 和 `U` 为具体的类型。但还有一种常见模板参数形式可以让不同字面量成为模板参数，即非类型模板参数：

```
template <typename T, int BufSize>
class buffer_t {
public:
    T& alloc();
    void free(T& item);
private:
    T data[BufSize];
}

buffer_t<int, 100> buf; // 100 作为模板参数
```

在这种模板参数形式下，我们可以将 100 作为模板的参数进行传递。在 C++11 引入了类型推导这一特性后，我们会很自然的问，既然此处的模板参数以具体的字面量进行传递，能否让编译器辅助我们进行类型推导，通过使用占位符 auto 从而不再需要明确指明类型？幸运的是，C++17 引入了这一特性，我们的确可以 auto 关键字，让编译器辅助完成具体类型的推导，例如：

```
template <auto value> void foo() {
    std::cout << value << std::endl;
    return;
}

int main() {
    foo<10>();  // value 被推导为 int 类型
}
```

2.6 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，从而达到简化代码的目的：

```
#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;
```

```
    }
};

int main() {
    Base b(2);
    std::cout << b.value1 << std::endl;
    std::cout << b.value2 << std::endl;
}
```

继承构造

在传统 C++ 中，构造函数如果需要继承是需要将参数一一传递的，这将导致效率低下。C++11 利用关键字 using 引入了继承构造函数的概念：

```
#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
};
int main() {
    Subclass s(3);
    std::cout << s.value1 << std::endl;
    std::cout << s.value2 << std::endl;
}
```

显式虚函数重载

在传统 C++ 中，经常容易发生意外重载虚函数的事情。例如：

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

```
};
struct SubClass: Base {
    void foo();
};
```

SubClass::foo 可能并不是程序员尝试重载虚函数，只是恰好加入了一个具有相同名字的函数。另一个可能的情形是，当基类的虚函数被删除后，子类拥有旧的函数就不再重载该虚拟函数并摇身一变成为了一个普通的类方法，这将造成灾难性的后果。

C++11 引入了 override 和 final 这两个关键字来防止上述情形的发生。

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其函数签名一致的虚函数，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

在传统 C++ 中，如果程序员没有提供，编译器会默认为对象生成默认构造函数、复制构造、赋值算符以及析构函数。另外，C++ 也为所有类定义了诸如 new delete 这样的运算符。当程序员有需要时，可以重载这部分函数。

这就引发了一些需求：无法精确控制默认函数的生成行为。例如禁止类的拷贝时，必须将复制构造函数与赋值算符声明为 private。尝试使用这些未定义的函数将导致编译或链接错误，则是一种非常不优雅的方式。

并且，编译器产生的默认构造函数与用户定义的构造函数无法同时存在。若用户定义了任何构造函数，编译器将不再生成默认构造函数，但有时候我们却希望同时拥有这两种构造函数，这就造成了尴尬。

C++11 提供了上述需求的解决方案，允许显式的声明采用或拒绝编译器自带的函数。例如：

```
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构造
    Magic(int magic_number);
}
```

强类型枚举

在传统 C++ 中，枚举类型并非类型安全，枚举类型会被视作整数，则会让两种完全不同的枚举类型可以进行直接的比较（虽然编译器给出了检查，但并非所有），**甚至同一个命名空间中的不同枚举类型的枚举值名字不能相同**，这通常不是我们希望看到的结果。

C++11 引入了枚举类（enumeration class），并使用 enum class 的语法进行声明：

```
enum class new_enum : unsigned int {
    value1,
    value2,
    value3 = 100,
    value4 = 100
};
```

这样定义的枚举实现了类型安全，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。但相同枚举值之间如果指定的值相同，那么可以进行比较：

```
if (new_enum::value3 == new_enum::value4) {
    // 会输出
    std::cout << "new_enum::value3 == new_enum::value4" << std::endl;
}
```

在这个语法中，枚举类型后面使用了冒号及类型关键字来指定枚举中枚举值的类型，这使得我们能够为枚举赋值（未指定时将默认使用 int）。

而我们希望获得枚举值的值时，将必须显式的进行类型转换，不过我们可以通过重载 << 这个算符来进行输出，可以收藏下面这个代码段：

```
#include <iostream>
template<typename T>
std::ostream& operator<<(
    typename std::enable_if<std::is_enum<T>::value,
        std::ostream>::type& stream, const T& e)
{
    return stream << static_cast<typename std::underlying_type<T>::type>(e);
}
```

这时，下面的代码将能够被编译：

```
std::cout << new_enum::value3 << std::endl
```

总结

本节介绍了现代 C++ 中对语言可用性的增强，其中笔者认为最为重要的几个特性是几乎所有人都需要了解并熟练使用的：

1. auto 类型推导
2. 范围 for 迭代
3. 初始化列表
4. 变参模板

习题

1. 使用结构化绑定，仅用一行函数内代码实现如下函数：

```
template <typename Key, typename Value, typename F>
void update(std::map<Key, Value>& m, F foo) {
    // TODO:
}
int main() {
    std::map<std::string, long long int> m {
        {"a", 1},
        {"b", 2},
        {"c", 3}
    };
    update(m, [](std::string key) {
        return std::hash<std::string>{}(key);
    });
    for (auto&& [key, value] : m)
        std::cout << key << ":" << value << std::endl;
}
```

2. 尝试用折叠表达式实现用于计算均值的函数，传入允许任意参数。

参考答案见此。

第 3 章语言运行期的强化

3.1 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。这样的场景其实有很多很多，所以匿名函数几乎是现代编程语言的标配。

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
}
```

上面的语法规则除了 [捕获列表] 内的东西外，其他部分都很好理解，只是一般函数的函数名被略去，返回值使用了一个 -> 的形式进行（我们在上一节前面的尾返回类型已经提到过这种写法了）。

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：

1. 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：

```
void lambda_value_capture() {
    int value = 1;
    auto copy_value = [value] {
        return value;
    };
    value = 100;
    auto stored_value = copy_value();
    std::cout << "stored_value = " << stored_value << std::endl;
    // 这时, stored_value == 1, 而 value == 100.
    // 因为 copy_value 在创建时就保存了一份 value 的拷贝
}
```

2. 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。

```
void lambda_reference_capture() {
    int value = 1;
    auto copy_value = [&value] {
        return value;
    };
    value = 100;
    auto stored_value = copy_value();
    std::cout << "stored_value = " << stored_value << std::endl;
    // 这时, stored_value == 100, value == 100.
    // 因为 copy_value 保存的是引用
}
```

3. 隐式捕获 手动书写捕获列表有时候是非常复杂的，这种机械性的工作可以交给编译器来处理，这时候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.

总结一下，捕获提供了 Lambda 表达式对外部值进行使用的功能，捕获列表的最常用的四种形式可以是:

- [] 空捕获列表
- [name1, name2, ...] 捕获一系列变量
- [&] 引用捕获, 从函数体内的使用确定引用捕获列表
- [=] 值捕获, 从函数体内的使用确定值捕获列表

4. 表达式捕获

这部分内容需要了解后面马上要提到的右值引用以及智能指针

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 给与了我们方便，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 <iostream>
#include <memory>  // std::make_unique
#include <utility> // std::move

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;
}
```

在上面的代码中，important 是一个独占指针，是不能够被 “=” 值捕获到，这时候我们可以将其转移为右值，在表达式中初始化。

泛型 Lambda

上一节中我们提到了 auto 关键字不能够用在参数表里，这是因为这样的写法会与模板的功能产生冲突。但是 Lambda 表达式并不是普通函数，所以在没有明确指明参数表类型的情况下，Lambda 表达式并不能够模板化。幸运的是，这种麻烦只存在于 C++11 中，从 C++14 开始，Lambda 函数的形式参数可以使用 auto 关键字来产生意义上的泛型：

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

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

3.2 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，这部分内容也相当重要，所以放到这里来进行介绍。

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，例如：

```
#include <iostream>

using foo = void(int); // 定义函数类型，using 的使用见上一节中的别名语法
void functional(foo f) { // 参数列表中定义的函数类型 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
}

int main() {
    auto f = [](int value) {
        std::cout << value << std::endl;
    };
```

```
    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
    return 0;
}
```

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。而这种类型，便是通过 std::function 引入的。

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，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。例如：

```
#include <functional>
#include <iostream>

int foo(int para) {
    return para;
}

int main() {
    // 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;
    };
    std::cout << func(10) << std::endl;
    std::cout << func2(10) << std::endl;
}
```

std::bind 和 std::placeholder

而 std::bind 则是用来绑定函数调用的参数的，它解决的需求是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，通过这个函数，我们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。例如：

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

```
    // 将参数 1,2 绑定到函数 foo 上，
    // 但使用 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);
}
```

提示： 注意 auto 关键字的妙用。有时候我们可能不太熟悉一个函数的返回值类型，但是我们却可以通过 auto 的使用来规避这一问题的出现。

3.3 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、将亡值、右值

要弄明白右值引用到底是怎么一回事，必须要对左值和右值做一个明确的理解。

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。

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。

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；要么是求值结果相当于字面量或匿名临时对象，例如 1+2。非引用返回的临时变量、运算表达式产生的临时变量、原始字面量、Lambda 表达式都属于纯右值。

需要注意的是，字面量除了字符串字面量以外，均为纯右值。而字符串字面量是一个左值，类型为 const char 数组。例如：

```
#include <type_traits>

int main() {
    // 正确，"01234" 类型为 const char [6]，因此是左值
    const char (&left)[6] = "01234";

    // 断言正确，确实是 const char [6] 类型，注意 decltype(expr) 在 expr 是左值
    // 且非无括号包裹的 id 表达式与类成员表达式时，会返回左值引用
    static_assert(std::is_same<decltype("01234"), const char(&)[6]>::value, "");
```

```
    // 错误，"01234" 是左值，不可被右值引用
    // const char (&&right)[6] = "01234";
}
```

但是注意，数组可以被隐式转换成相对应的指针类型，而转换表达式的结果（如果不是左值引用）则一定是个右值（右值引用为将亡值，否则为纯右值）。例如：

```
const char*   p   = "01234";  // 正确，"01234" 被隐式转换为 const char*
const char*&& pr  = "01234";  // 正确，"01234" 被隐式转换为 const char*，该转换的结果是纯右值
// const char*& pl = "01234"; // 错误，此处不存在 const char* 类型的左值
```

将亡值 (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，纯右值和右值是同一个概念），也就是即将被销毁、却能够被移动的值。

将亡值可能稍有些难以理解，我们来看这样的代码：

```
std::vector<int> foo() {
    std::vector<int> temp = {1, 2, 3, 4};
    return temp;
}

std::vector<int> v = foo();
```

在这样的代码中，就传统的理解而言，函数 foo 的返回值 temp 在内部创建然后被赋值给 v，然而 v 获得这个对象时，会将整个 temp 拷贝一份，然后把 temp 销毁，如果这个 temp 非常大，这将造成大量额外的开销（这也就是传统 C++ 一直被诟病的问题）。在最后一行中，v 是左值、foo() 返回的值就是右值（也是纯右值）。但是，v 可以被别的变量捕获到，而 foo() 产生的那个返回值作为一个临时值，一旦被 v 复制后，将立即被销毁，无法获取、也不能修改。而将亡值就定义了这样一种行为：临时的值能够被识别、同时又能够被移动。

在 C++11 之后，编译器为我们做了一些工作，此处的左值 temp 会被进行此隐式右值转换，等价于 static_cast<std::vector<int> &&>(temp)，进而此处的 v 会将 foo 局部返回的值进行移动。也就是后面我们将会提到的移动语义。

右值引用和左值引用

要拿到一个将亡值，就需要用到右值引用：T &&，其中 T 是类型。右值引用的声明让这个临时值的生命周期得以延长、只要变量还活着，那么将亡值将继续存活。

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临时对象，例如：

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

```
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 lv1 = "string,"; // lv1 是一个左值
    // std::string&& r1 = lv1; // 非法，右值引用不能引用左值
    std::string&& rv1 = std::move(lv1); // 合法，std::move 可以将左值转移为右值
    std::cout << rv1 << std::endl; // string,

    const std::string& lv2 = lv1 + lv1; // 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::cout << lv2 << std::endl; // string,string,

    std::string&& rv2 = lv1 + 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
    std::cout << rv2 << std::endl; // string,string,string,Test

    reference(rv2); // 输出左值

    return 0;
}
```

rv2 虽然引用了一个右值，但由于它是一个引用，所以 rv2 依然是一个左值。

注意，这里有一个很有趣的历史遗留问题，我们先看下面的代码：

```
#include <iostream>

int main() {
    // int &a = std::move(1);     // 不合法，非常量左引用无法引用右值
    const int &b = std::move(1); // 合法，常量左引用允许引用右值

    std::cout << a << b << std::endl;
}
```

第一个问题，为什么不允许非常量引用绑定到非左值？这是因为这种做法存在逻辑错误：

```
void increase(int & v) {
    v++;
}
void foo() {
    double s = 1;
    increase(s);
}
```

由于 int& 不能引用 double 类型的参数，因此必须产生一个临时值来保存 s 的值，从而当 increase() 修改这个临时值时，调用完成后 s 本身并没有被修改。

第二个问题，为什么常量引用允许绑定到非左值？原因很简单，因为 Fortran 需要。

移动语义

传统 C++ 通过拷贝构造函数和赋值操作符为类对象设计了拷贝/复制的概念，但为了实现对资源的移动操作，调用者必须使用先复制、再析构的方式，否则就需要自己实现移动对象的接口。试想，搬家的时候是把家里的东西直接搬到新家去，而不是将所有东西复制一份（重买）再放到新家、再把原来的东西全部扔掉（销毁），这是非常反人类的一件事情。

传统的 C++ 没有区分『移动』和『拷贝』的概念，造成了大量的数据拷贝，浪费时间和空间。右值引用的出现恰好就解决了这两个概念的混淆问题，例如：

```
#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;
    }
};
// 防止编译器优化
```

```
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);
    std::cout << "obj:" << std::endl;
    std::cout << obj.pointer << std::endl;
    std::cout << *obj.pointer << std::endl;
    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，产生一个将亡值，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，而将亡值的指针被设置为 nullptr，防止了这块内存区域被销毁。

从而避免了无意义的拷贝构造，加强了性能。再来看看涉及标准库的例子：

```
#include <iostream> // std::cout
#include <utility> // std::move
#include <vector> // std::vector
#include <string> // std::string

int main() {

    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;

    return 0;
}
```

完美转发

前面我们提到了，一个声明的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) {
    std::cout << " 普通传参:";
    reference(v); // 始终调用 reference(int&)
}
int main() {
    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

    return 0;
}
```

对于 `pass(1)` 来说，虽然传递的是右值，但由于 `v` 是一个引用，所以同时也是左值。因此 `reference(v)` 会调用 `reference(int&)`，输出『左值』。而对于 `pass(l)` 而言，`l` 是一个左值，为什么会成功传递给 `pass(T&&)` 呢？

这是基于**引用坍缩规则**的：在传统 C++ 中，我们不能够对一个引用类型继续进行引用，但 C++ 由于右值引用的出现而放宽了这一做法，从而产生了引用坍缩规则，允许我们对引用进行引用，既能左引用，又能右引用。但是却遵循如下规则：

函数形参类型	实参参数类型	推导后函数形参类型
T&	左引用	T&
T&	右引用	T&

函数形参类型	实参参数类型	推导后函数形参类型
T&&	左引用	T&
T&&	右引用	T&&

因此，模板函数中使用 T&& 不一定能进行右值引用，当传入左值时，此函数的引用将被推导为左值。更准确的讲，**无论模板参数是什么类型的引用，当且仅当实参类型为右引用时，模板参数才能被推导为右引用类型**。这才使得 v 作为左值的成功传递。

完美转发就是基于上述规律产生的。所谓完美转发，就是为了让我们在传递参数的时候，保持原来的参数类型（左引用保持左引用，右引用保持右引用）。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) {
    std::cout << "               普通传参: ";
    reference(v);
    std::cout << "       std::move 传参: ";
    reference(std::move(v));
    std::cout << "    std::forward 传参: ";
    reference(std::forward<T>(v));
    std::cout << "static_cast<T&&> 传参: ";
    reference(static_cast<T&&>(v));
}
int main() {
    std::cout << " 传递右值:" << std::endl;
    pass(1);

    std::cout << " 传递左值:" << std::endl;
    int v = 1;
    pass(v);

    return 0;
}
```

输出结果为:

```
传递右值:
              普通传参: 左值引用
       std::move 传参: 右值引用
    std::forward 传参: 右值引用
static_cast<T&&> 传参: 右值引用
传递左值:
              普通传参: 左值引用
       std::move 传参: 右值引用
    std::forward 传参: 左值引用
static_cast<T&&> 传参: 左值引用
```

无论传递参数为左值还是右值,普通传参都会将参数作为左值进行转发;由于类似的原因,std::move 总会接受到一个左值,从而转发调用了 reference(int&&) 输出右值引用。

唯独 std::forward 即没有造成任何多余的拷贝,同时**完美转发** (传递) 了函数的实参给了内部调用的其他函数。

std::forward 和 std::move 一样,没有做任何事情,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。

读者可能会好奇,为何一条语句能够针对两种类型的返回对应的值,我们再简单看一看 std::forward 的具体实现机制,std::forward 包含两个重载:

```
template<typename _Tp>
constexpr _Tp&& forward(typename std::remove_reference<_Tp>::type& __t) noexcept
{ return static_cast<_Tp&&>(__t); }

template<typename _Tp>
constexpr _Tp&& forward(typename std::remove_reference<_Tp>::type&& __t) noexcept
{
    static_assert(!std::is_lvalue_reference<_Tp>::value, "template argument"
        " substituting _Tp is an lvalue reference type");
    return static_cast<_Tp&&>(__t);
}
```

在这份实现中,std::remove_reference 的功能是消除类型中的引用,std::is_lvalue_reference 则用于检查类型推导是否正确,在 std::forward 的第二个实现中检查了接收到的值确实是一个左值,进而体现了坍缩规则。

当 std::forward 接受左值时,_Tp 被推导为左值,所以返回值为左值;而当其接受右值时,_Tp 被推导为右值引用,则基于坍缩规则,返回值便成为了 && + && 的右值。可见 std::forward 的原理在于巧妙的利用了模板类型推导中产生的差异。

这时我们能回答这样一个问题：为什么在使用循环语句的过程中，auto&& 是最安全的方式？因为当 auto 被推导为不同的左右引用时，与 && 的坍缩组合是完美转发。

总结

本章介绍了现代 C++ 中最为重要的几个语言运行时的增强，其中笔者认为本节中提到的所有特性都是值得掌握的：

1. Lambda 表达式
2. 函数对象容器 std::function
3. 右值引用

进一步阅读的参考文献

- Bjarne Stroustrup, C++ 语言的设计与演化

第 4 章容器

4.1 线性容器

std::array

看到这个容器的时候肯定会出现这样的问题：

1. 为什么要引入 std::array 而不是直接使用 std::vector？
2. 已经有了传统数组，为什么要用 std::array？

先回答第一个问题，与 std::vector 不同，std::array 对象的大小是固定的，如果容器大小是固定的，那么可以优先考虑使用 std::array 容器。另外由于 std::vector 是自动扩容的，当存入大量的数据后，并且对容器进行了删除操作，容器并不会自动归还被删除元素相应的内存，这时候就需要手动运行 shrink_to_fit() 释放这部分内存。

```
std::vector<int> v;
std::cout << "size:" << v.size() << std::endl;         // 输出 0
std::cout << "capacity:" << v.capacity() << std::endl; // 输出 0

// 如下可看出 std::vector 的存储是自动管理的，按需自动扩张
// 但是如果空间不足，需要重新分配更多内存，而重分配内存通常是性能上有开销的操作
v.push_back(1);
v.push_back(2);
v.push_back(3);
```

```
std::cout << "size:" << v.size() << std::endl;          // 输出 3
std::cout << "capacity:" << v.capacity() << std::endl; // 输出 4

// 这里的自动扩张逻辑与 Golang 的 slice 很像
v.push_back(4);
v.push_back(5);
std::cout << "size:" << v.size() << std::endl;          // 输出 5
std::cout << "capacity:" << v.capacity() << std::endl; // 输出 8

// 如下可看出容器虽然清空了元素，但是被清空元素的内存并没有归还
v.clear();
std::cout << "size:" << v.size() << std::endl;          // 输出 0
std::cout << "capacity:" << v.capacity() << std::endl; // 输出 8

// 额外内存可通过 shrink_to_fit() 调用返回给系统
v.shrink_to_fit();
std::cout << "size:" << v.size() << std::endl;          // 输出 0
std::cout << "capacity:" << v.capacity() << std::endl; // 输出 0
```

而第二个问题就更加简单，使用 std::array 能够让代码变得更加“现代化”，而且封装了一些操作函数，比如获取数组大小以及检查是否非空，同时还能够友好的使用标准库中的容器算法，比如 std::sort。

使用 std::array 很简单，只需指定其类型和大小即可：

```
std::array<int, 4> arr = {1, 2, 3, 4};

arr.empty(); // 检查容器是否为空
arr.size();  // 返回容纳的元素数

// 迭代器支持
for (auto &i : arr)
{
    // ...
}

// 用 lambda 表达式排序
std::sort(arr.begin(), arr.end(), [](int a, int b) {
    return b < a;
});

// 数组大小参数必须是常量表达式
constexpr int len = 4;
std::array<int, len> arr = {1, 2, 3, 4};
```

```
// 非法，不同于 C 风格数组，std::array 不会自动退化成 T*
// int *arr_p = arr;
```

当我们开始用上了 std::array 时，难免会遇到要将其兼容 C 风格的接口，这里有三种做法：

```
void foo(int *p, int len) {
    return;
}

std::array<int, 4> arr = {1,2,3,4};

// C 风格接口传参
// foo(arr, arr.size()); // 非法，无法隐式转换
foo(&arr[0], arr.size());
foo(arr.data(), arr.size());

// 使用 `std::sort`
std::sort(arr.begin(), arr.end());
```

std::forward_list

std::forward_list 是一个列表容器，使用方法和 std::list 基本类似，因此我们就不花费篇幅进行介绍了。

需要知道的是，和 std::list 的双向链表的实现不同，std::forward_list 使用单向链表进行实现，提供了 O(1) 复杂度的元素插入，不支持快速随机访问（这也是链表的特点），也是标准库容器中唯一一个不提供 size() 方法的容器。当不需要双向迭代时，具有比 std::list 更高的空间利用率。

4.2 无序容器

我们已经熟知了传统 C++ 中的有序容器 std::map/std::set，这些元素内部通过红黑树进行实现，插入和搜索的平均复杂度均为 O(log(size))。在插入元素时候，会根据 < 操作符比较元素大小并判断元素是否相同，并选择合适的位置插入到容器中。当对这个容器中的元素进行遍历时，输出结果会按照 < 操作符的顺序来逐个遍历。

而无序容器中的元素是不进行排序的，内部通过 Hash 表实现，插入和搜索元素的平均复杂度为 O(constant)，在不关心容器内部元素顺序时，能够获得显著的性能提升。

C++11 引入了的两组无序容器分别是：std::unordered_map/std::unordered_multimap 和 std::unordered_set/std::unordered_multiset。

它们的用法和原有的 std::map/std::multimap/std::set/set::multiset 基本类似，由于这些容器我们已经很熟悉了，便不一一举例，我们直接来比较一下 std::map 和 std::unordered_map：

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
#include <map>

int main() {
    // 两组结构按同样的顺序初始化
    std::unordered_map<int, std::string> u = {
        {1, "1"},
        {3, "3"},
        {2, "2"}
    };
    std::map<int, std::string> v = {
        {1, "1"},
        {3, "3"},
        {2, "2"}
    };

    // 分别对两组结构进行遍历
    std::cout << "std::unordered_map" << std::endl;
    for( const auto & n : u)
        std::cout << "Key:[" << n.first << "] Value:[" << n.second << "]\n";

    std::cout << std::endl;
    std::cout << "std::map" << std::endl;
    for( const auto & n : v)
        std::cout << "Key:[" << n.first << "] Value:[" << n.second << "]\n";
}
```

最终的输出结果为:

```
std::unordered_map
Key:[2] Value:[2]
Key:[3] Value:[3]
Key:[1] Value:[1]

std::map
Key:[1] Value:[1]
Key:[2] Value:[2]
Key:[3] Value:[3]
```

4.3 元组

了解过 Python 的程序员应该知道元组的概念，纵观传统 C++ 中的容器，除了 std::pair 外，似乎没有现成的结构能够用来存放不同类型的数据（通常我们会自己定义结构）。但 std::pair 的缺陷是显而易见的，只能保存两个元素。

元组基本操作

关于元组的使用有三个核心的函数：

1. std::make_tuple: 构造元组
2. std::get: 获得元组某个位置的值
3. std::tie: 元组拆包

```
#include <tuple>
#include <iostream>

auto get_student(int id)
{
    // 返回类型被推断为 std::tuple<double, char, std::string>

    if (id == 0)
        return std::make_tuple(3.8, 'A', " 张三");
    if (id == 1)
        return std::make_tuple(2.9, 'C', " 李四");
    if (id == 2)
        return std::make_tuple(1.7, 'D', " 王五");
    return std::make_tuple(0.0, 'D', "null");
    // 如果只写 0 会出现推断错误，编译失败
}

int main()
{
    auto student = get_student(0);
    std::cout << "ID: 0, "
    << "GPA: " << std::get<0>(student) << ", "
    << " 成绩: " << std::get<1>(student) << ", "
    << " 姓名: " << std::get<2>(student) << '\n';

    double gpa;
    char grade;
    std::string name;
```

```
    // 元组进行拆包
    std::tie(gpa, grade, name) = get_student(1);
    std::cout << "ID: 1, "
    << "GPA: " << gpa << ", "
    << " 成绩: " << grade << ", "
    << " 姓名: " << name << '\n';
}
```

std::get 除了使用常量获取元组对象外，C++14 增加了使用类型来获取元组中的对象：

```
std::tuple<std::string, double, double, int> t("123", 4.5, 6.7, 8);
std::cout << std::get<std::string>(t) << std::endl;
std::cout << std::get<double>(t) << std::endl; // 非法，引发编译期错误
std::cout << std::get<3>(t) << std::endl;
```

运行期索引

如果你仔细思考一下可能就会发现上面代码的问题，std::get<> 依赖一个编译期的常量，所以下面的方式是不合法的：

```
int index = 1;
std::get<index>(t);
```

那么要怎么处理？答案是，使用 std::variant<>（C++ 17 引入），提供给 variant<> 的类型模板参数可以让一个 variant<> 从而容纳提供的几种类型的变量（在其他语言，例如 Python/JavaScript 等，表现为动态类型）：

```
#include <variant>
template <size_t n, typename... T>
constexpr std::variant<T...> _tuple_index(const std::tuple<T...>& tpl, size_t i) {
    if constexpr (n >= sizeof...(T))
        throw std::out_of_range(" 越界.");
    if (i == n)
        return std::variant<T...>{ std::in_place_index<n>, std::get<n>(tpl) };
    return _tuple_index<(n < sizeof...(T)-1 ? n+1 : 0)>(tpl, i);
}
template <typename... T>
constexpr std::variant<T...> tuple_index(const std::tuple<T...>& tpl, size_t i) {
    return _tuple_index<0>(tpl, i);
}
template <typename T0, typename ... Ts>
```

```
std::ostream & operator<< (std::ostream & s, std::variant<T0, Ts...> const & v) {
    std::visit([&](auto && x){ s << x;}, v);
    return s;
}
```

这样我们就能：

```
int i = 1;
std::cout << tuple_index(t, i) << std::endl;
```

元组合并与遍历

还有一个常见的需求就是合并两个元组，这可以通过 std::tuple_cat 来实现：

```
auto new_tuple = std::tuple_cat(get_student(1), std::move(t));
```

马上就能够发现，应该如何快速遍历一个元组？但是我们刚才介绍了如何在运行期通过非常数索引一个 tuple 那么遍历就变得简单了，首先我们需要知道一个元组的长度，可以：

```
template <typename T>
auto tuple_len(T &tpl) {
    return std::tuple_size<T>::value;
}
```

这样就能够对元组进行迭代了：

```
// 迭代
for(int i = 0; i != tuple_len(new_tuple); ++i)
    // 运行期索引
    std::cout << tuple_index(new_tuple, i) << std::endl;
```

总结

本章简单介绍了现代 C++ 中新增的容器，它们的用法和传统 C++ 中已有的容器类似，相对简单，可以根据实际场景丰富的选择需要使用的容器，从而获得更好的性能。

std::tuple 虽然有效，但是标准库提供的功能有限，没办法满足运行期索引和迭代的需求，好在我们还有其他的方法可以自行实现。

第 5 章智能指针与内存管理

5.1 RAII 与引用计数

了解 Objective-C/Swift 的程序员应该知道引用计数的概念。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，每当增加一次对同一个对象的引用，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。

凡事都有例外，我们总会有需要将对象在自由存储上分配的需求，在传统 C++ 里我们只好使用 new 和 delete 去『记得』对资源进行释放。而 C++11 引入了智能指针的概念，使用了引用计数的想法，让程序员不再需要关心手动释放内存。这些智能指针包括 std::shared_ptr/std::unique_ptr/std::weak_ptr，使用它们需要包含头文件 <memory>。

> 注意：引用计数不是垃圾回收，引用计数能够尽快收回不再被使用的对象，同时在回收的过程中也不会造成长时间的等待，更能够清晰明确的表明资源的生命周期。

5.2 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，所以 std::make_shared 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。例如：

```
#include <iostream>
#include <memory>
void foo(std::shared_ptr<int> i) {
    (*i)++;
}
int main() {
    // auto pointer = new int(10); // illegal, no direct assignment
    // 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。例如：

```
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;           // pointer2 已 reset; 0
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;           // pointer3 已 reset; 0
```

5.3 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)... ) );
}
```

至于为什么没有提供，C++ 标准委员会主席 Herb Sutter 在他的博客中提到原因是因为『被他们忘记了』。

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，例如：

```
#include <iostream>
#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
}
```

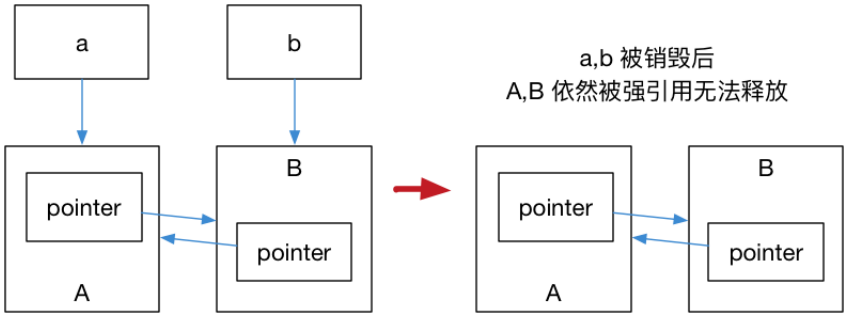


图 2: 图 5.1

5.4 std::weak_ptr

如果你仔细思考 std::shared_ptr 就会发现依然存在着资源无法释放的问题。看下面这个例子:

```
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;
}
```

运行结果是 A, B 都不会被销毁，这是因为 a,b 内部的 pointer 同时又引用了 a,b，这使得 a,b 的引用计数均变为了 2，而离开作用域时，a,b 智能指针被析构，却只能造成这块区域的引用计数减一，这样就导致了 a,b 对象指向的内存区域引用计数不为零，而外部已经没有办法找到这块区域了，也就造成了内存泄露，如图 5.1:

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，当换用弱引用时候，最终的

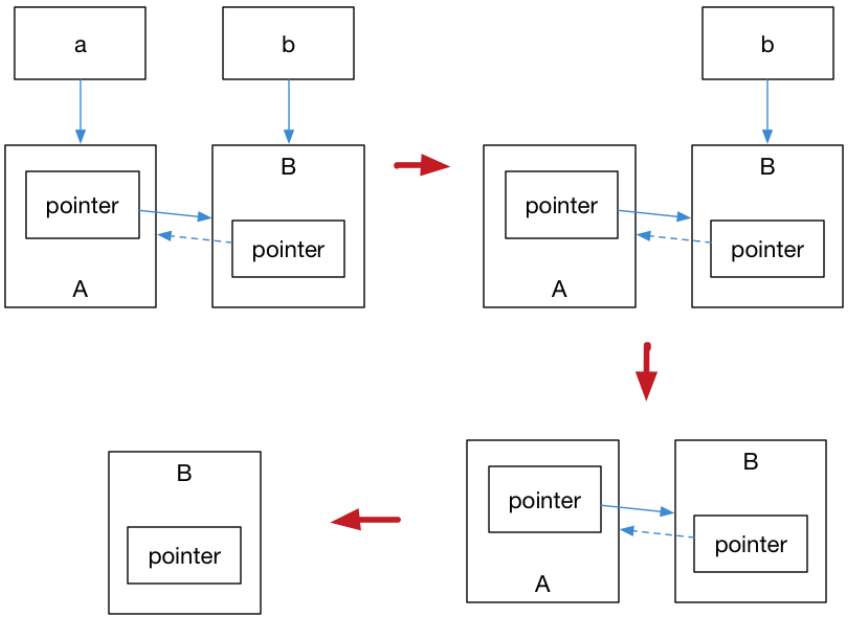


图 3: 图 5.2

释放流程如图 5.2 所示：

在上图中，最后一步只剩下 B，而 B 并没有任何智能指针引用它，因此这块内存资源也会被释放。

`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 `nullptr`。

总结

智能指针这种技术并不新奇，在很多语言中都是一种常见的技术，现代 C++ 将这项技术引进，在一定程度上消除了 `new/delete` 的滥用，是一种更加成熟的编程范式。

进一步阅读的参考资料

1. stackoverflow 上关于『C++11 为什么没有 make_unique』的讨论

第 6 章正则表达式

6.1 正则表达式简介

正则表达式不是 C++ 语言的一部分，这里仅做简单的介绍。

正则表达式描述了一种字符串匹配的模式。一般使用正则表达式主要是实现下面三个需求：

1. 检查一个串是否包含某种形式的子串；
2. 将匹配的子串替换；
3. 从某个串中取出符合条件的子串。

正则表达式是由普通字符（例如 a 到 z）以及特殊字符组成的文字模式。模式描述在搜索文本时要匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。

普通字符

普通字符包括没有显式指定为元字符的所有可打印和不可打印字符。这包括所有大写和小写字母、所有数字、所有标点符号和一些其他符号。

特殊字符

特殊字符是正则表达式里有特殊含义的字符，也是正则表达式的核心匹配语法。参见下表:

特别字符	描述
$	匹配输入字符串的结尾位置。
(,)	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用。
*	匹配前面的子表达式零次或多次。
+	匹配前面的子表达式一次或多次。
.	匹配除换行符 \n 之外的任何单字符。
[	标记一个中括号表达式的开始。
?	匹配前面的子表达式零次或一次，或指明一个非贪婪限定符。
\	将下一个字符标记为或特殊字符、或原义字符、或向后引用、或八进制转义符。例如，n 匹配字符 n。\n 匹配换行符。序列 \\ 匹配 '\' 字符，而 \(则匹配 '(' 字符。
^	匹配输入字符串的开始位置，除非在方括号表达式中使用，此时它表示不接受该字符集合。
{	标记限定符表达式的开始。
|	指明两项之间的一个选择。

限定符

限定符用来指定正则表达式的一个给定的组件必须要出现多少次才能满足匹配。见下表:

字符	描述
*	匹配前面的子表达式零次或多次。例如，foo* 能匹配 fo 以及 foooo。* 等价于 {0,}。
+	匹配前面的子表达式一次或多次。例如，foo+ 能匹配 foo 以及 foooo，但不能匹配 fo。+ 等价于 {1,}。

字符	描述
?	匹配前面的子表达式零次或一次。例如，Your(s)? 可以匹配 Your 或 Yours 中的 Your 。? 等价于 {0,1}。
{n}	n 是一个非负整数。匹配确定的 n 次。例如，o{2} 不能匹配 for 中的 o，但是能匹配 foo 中的两个 o。
{n,}	n 是一个非负整数。至少匹配 n 次。例如，o{2,} 不能匹配 for 中的 o，但能匹配 fooooo 中的所有 o。o{1,} 等价于 o+。o{0,} 则等价于 o*。
{n,m}	m 和 n 均为非负整数，其中 n 小于等于 m。最少匹配 n 次且最多匹配 m 次。例如，o{1,3} 将匹配 fooooo 中的前三个 o。o{0,1} 等价于 o?。注意，在逗号和两个数之间不能有空格。

有了这两张表，我们通常就能够读懂几乎所有的正则表达式了。

6.2 std::regex 及其相关

对字符串内容进行匹配的最常见手段就是使用正则表达式。可惜在传统 C++ 中正则表达式一直没有得到语言层面的支持，没有纳入标准库，而 C++ 作为一门高性能语言，在后台服务的开发中，对 URL 资源链接进行判断时，使用正则表达式也是工业界最为成熟的普遍做法。

一般的解决方案就是使用 boost 的正则表达式库。而 C++11 正式将正则表达式的的处理方法纳入标准库的行列，从语言级上提供了标准的支持，不再依赖第三方。

C++11 提供的正则表达式库操作 std::string 对象，模式 std::regex (本质是 std::basic_regex) 进行初始化，通过 std::regex_match 进行匹配，从而产生 std::smatch （本质是 std::match_results 对象）。

我们通过一个简单的例子来简单介绍这个库的使用。考虑下面的正则表达式:

- [a-z]+\.txt: 在这个正则表达式中，[a-z] 表示匹配一个小写字母，+ 可以使前面的表达式匹配多次，因此 [a-z]+ 能够匹配一个小写字母组成的字符串。在正则表达式中一个 . 表示匹配任意字符，而 \. 则表示匹配字符 .，最后的 txt 表示严格匹配 txt 则三个字母。因此这个正则表达式的所要匹配的内容就是由纯小写字母组成的文本文件。

std::regex_match 用于匹配字符串和正则表达式，有很多不同的重载形式。最简单的一个形式就是传入 std::string 以及一个 std::regex 进行匹配，当匹配成功时，会返回 true，否则返回 false。例如:

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

int main() {
```

```
    std::string fnames[] = {"foo.txt", "bar.txt", "test", "a0.txt", "AAA.txt"};
    // 在 C++ 中 \ 会被作为字符串内的转义符，
    // 为使 \. 作为正则表达式传递进去生效，需要对 \ 进行二次转义，从而有 \\.
    std::regex txt_regex("[a-z]+\\.txt");
    for (const auto &fname: fnames)
        std::cout << fname << ": " << std::regex_match(fname, txt_regex) << std::endl;
}
```

另一种常用的形式就是依次传入 std::string/std::smatch/std::regex 三个参数，其中 std::smatch 的本质其实是 std::match_results。故而在标准库的实现中，std::smatch 被定义为了 std::match_results<std::string::const_iterator>，也就是一个子串迭代器类型的 match_results。使用 std::smatch 可以方便的对匹配的结果进行获取，例如：

```
std::regex base_regex("([a-z]+)\\.txt");
std::smatch base_match;
for(const auto &fname: fnames) {
    if (std::regex_match(fname, base_match, base_regex)) {
        // std::smatch 的第一个元素匹配整个字符串
        // std::smatch 的第二个元素匹配了第一个括号表达式
        if (base_match.size() == 2) {
            std::string base = base_match[1].str();
            std::cout << "sub-match[0]: " << base_match[0].str() << std::endl;
            std::cout << fname << " sub-match[1]: " << base << std::endl;
        }
    }
}
```

以上两个代码段的输出结果为：

```
foo.txt: 1
bar.txt: 1
test: 0
a0.txt: 0
AAA.txt: 0
sub-match[0]: foo.txt
foo.txt sub-match[1]: foo
sub-match[0]: bar.txt
bar.txt sub-match[1]: bar
```

总结

本节简单介绍了正则表达式本身，然后根据使用正则表达式的主要需求，通过一个实际的例子介绍了正则表达式库的使用。

习题

1. 在 Web 服务器开发中，我们通常希望服务某些满足某个条件的路由。正则表达式便是完成这一目标的工具之一。

   给定如下请求结构：

```
struct Request {
    // request method, POST, GET; path; HTTP version
    std::string method, path, http_version;
    // use smart pointer for reference counting of content
    std::shared_ptr<std::istream> content;
    // hash container, key-value dict
    std::unordered_map<std::string, std::string> header;
    // use regular expression for path match
    std::smatch path_match;
};
```

   请求的资源类型：

```
typedef std::map<
    std::string, std::unordered_map<
        std::string,std::function<void(std::ostream&, Request&)>>> resource_type;
```

   以及服务端模板：

```
template <typename socket_type>
class ServerBase {
public:
    resource_type resource;
    resource_type default_resource;

    void start() {
        // TODO
    }
protected:
    Request parse_request(std::istream& stream) const {
        // TODO
    }
}
```

   请实现成员函数 `start()` 与 `parse_request`。使得服务器模板使用者可以如下指定路由：

```
template<typename SERVER_TYPE>
void start_server(SERVER_TYPE &server) {

    // process GET request for /match/[digit+numbers],
    // e.g. GET request is /match/abc123, will return abc123
    server.resource["fill_your_reg_ex"]["GET"] =
        [](ostream& response, Request& request)
    {
        string number=request.path_match[1];
        response << "HTTP/1.1 200 OK\r\nContent-Length: "
                 << number.length() << "\r\n\r\n" << number;
    };

    // peocess default GET request;
    // anonymous function will be called
    // if no other matches response files in folder web/
    // default: index.html
    server.default_resource["fill_your_reg_ex"]["GET"] =
        [](ostream& response, Request& request)
    {
        string filename = "www/";

        string path = request.path_match[1];

        // forbidden use '..' access content outside folder web/
        size_t last_pos = path.rfind(".");
        size_t current_pos = 0;
        size_t pos;
        while((pos=path.find('.', current_pos)) != string::npos && pos != last_pos) {
            current_pos = pos;
            path.erase(pos, 1);
            last_pos--;
        }

        // (...)
    };

    server.start();
}
```

参考答案见此。

进一步阅读的参考资料

1. 知乎『如何评价 GCC 的 C++11 正则表达式?』中原库作者 Tim Shen 的回答
2. 正则表达式库文档

第 7 章并行与并发

7.1 并行基础

`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()` 来等待一个线程结束（与该线程汇合）等等，例如：

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

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

7.2 互斥量与临界区

我们在操作系统、亦或是数据库的相关知识中已经了解过了有关并发技术的基本知识，`mutex` 就是其中的核心之一。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，例如：

```
#include <iostream>
#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
}

int main() {
    std::thread t1(critical_section, 2), t2(critical_section, 3);
    t1.join();
    t2.join();

    std::cout << v << std::endl;
    return 0;
}
```

由于 C++ 保证了所有栈对象在生命周期结束时会被销毁，所以这样的代码也是异常安全的。无论 critical_section() 正常返回、还是在中途抛出异常，都会引发栈回溯，也就自动调用了 unlock()。

没有捕获抛出的异常（此时由实现定义是否进行栈回溯）。

而 std::unique_lock 则是相对于 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。

例如：

```
#include <iostream>
#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;
    std::cout << v << std::endl;
    // 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;
    std::cout << v << std::endl;
}

int main() {
    std::thread t1(critical_section, 2), t2(critical_section, 3);
    t1.join();
    t2.join();
    return 0;
}
```

7.3 期物

期物（Future）表现为 `std::future`，它提供了一个访问异步操作结果的途径，这句话很不好理解。为了理解这个特性，我们需要先理解一下在 C++11 之前的多线程行为。

试想，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，并返回我一个结果。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）。

为了看一个例子，我们这里额外使用 `std::packaged_task`，它可以用来封装任何可以调用的目标，从而用于实现异步的调用。举例来说：

```
#include <iostream>
```

```
#include <future>
#include <thread>

int main() {
    // 将一个返回值为 7 的 lambda 表达式封装到 task 中
    // std::packaged_task 的模板参数为要封装函数的类型
    std::packaged_task<int()> task([](){return 7;});
    // 获得 task 的期物
    std::future<int> result = task.get_future(); // 在一个线程中执行 task
    std::thread(std::move(task)).detach();
    std::cout << "waiting...";
    result.wait(); // 在此设置屏障，阻塞到期物的完成
    // 输出执行结果
    std::cout << "done!" << std:: endl << "future result is "
              << result.get() << std::endl;
    return 0;
}
```

在封装好要调用的目标后，可以使用 get_future() 来获得一个 std::future 对象，以便之后实施线程同步。

7.4 条件变量

条件变量 std::condition_variable 是为了解决死锁而生，当互斥操作不够用而引入的。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;
    bool notified = false;  // 通知信号
```

```
// 生产者
auto producer = [&]() {
    for (int i = 0; ; i++) {
        std::this_thread::sleep_for(std::chrono::milliseconds(900));
        std::unique_lock<std::mutex> lock(mtx);
        std::cout << "producing " << i << std::endl;
        produced_nums.push(i);
        notified = true;
        cv.notify_all(); // 此处也可以使用 notify_one
    }
};
// 消费者
auto consumer = [&]() {
    while (true) {
        std::unique_lock<std::mutex> lock(mtx);
        while (!notified) {  // 避免虚假唤醒
            cv.wait(lock);
        }
        // 短暂取消锁，使得生产者有机会在消费者消费空前继续生产
        lock.unlock();
        // 消费者慢于生产者
        std::this_thread::sleep_for(std::chrono::milliseconds(1000));
        lock.lock();
        while (!produced_nums.empty()) {
            std::cout << "consuming " << produced_nums.front() << std::endl;
            produced_nums.pop();
        }
        notified = false;
    }
};

// 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;
}
```

值得一提的是，在生产者中我们虽然可以使用 notify_one()，但实际上并不建议在此处使用，因为在多消费者的情况下，我们的消费者实现中简单放弃了锁的持有，这使得可能让其他消费者争夺此锁，从而更好的利用多个消费者之间的并发。话虽如此，但实际上因为 std::mutex 的排他性，我们根本无法期待多个消费者能真正意义上的并行消费队列的中生产的内容，我们仍需要粒度更细的手段。

7.5 原子操作与内存模型

细心的读者可能会对前一小节中生产者消费者模型的例子可能存在编译器优化导致程序出错的情况产生疑惑。例如，编译器可能对变量 notified 存在优化，例如将其作为一个寄存器的值，从而导致消费者线程永远无法观察到此值的变化。这是一个好问题，为了解释清楚这个问题，我们需要进一步讨论从 C++ 11 起引入的内存模型这一概念。我们首先来看一个问题，下面这段代码输出结果是多少?

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

int main() {
    int a = 0;
    volatile int flag = 0;

    std::thread t1([&]() {
        while (flag != 1);

        int b = a;
        std::cout << "b = " << b << std::endl;
    });

    std::thread t2([&]() {
        a = 5;
        flag = 1;
    });

    t1.join();
    t2.join();
    return 0;
}
```

从直观上看，t2 中 a = 5; 这一条语句似乎总在 flag = 1; 之前得到执行，而 t1 中 while (flag != 1) 似乎保证了 std::cout << "b = " << b << std::endl; 不会再标记被改变前执行。从逻辑上看，似乎 b 的值应该等于 5。但实际情况远比此复杂得多，或者说这段代码本身属于未定义的行为，因

为对于 a 和 flag 而言，他们在两个并行的线程中被读写，出现了竞争。除此之外，即便我们忽略竞争读写，仍然可能受 CPU 的乱序执行，编译器对指令的重排的影响，导致 a = 5 发生在 flag = 1 之后。从而 b 可能输出 0。

原子操作

std::mutex 可以解决上面出现的并发读写的问题，但互斥锁是操作系统级的功能，这是因为一个互斥锁的实现通常包含两条基本原理：

1. 提供线程间自动的状态转换，即『锁住』这个状态
2. 保障在互斥锁操作期间，所操作变量的内存与临界区外进行隔离

这是一组非常强的同步条件，换句话说当最终编译为 CPU 指令时会表现为非常多的指令（我们之后再来看如何实现一个简单的互斥锁）。这对于一个仅需原子级操作（没有中间态）的变量，似乎太苛刻了。

关于同步条件的研究有着非常久远的历史，我们在这里不进行赘述。读者应该明白，现代 CPU 体系结构提供了 CPU 指令级的原子操作，因此在 C++11 中多线程下共享变量的读写这一问题上，还引入了 std::atomic 模板，使得我们实例化一个原子类型，将一个原子类型读写操作从一组指令，最小化到单个 CPU 指令。例如：

```
std::atomic<int> counter;
```

并为整数或浮点数的原子类型提供了基本的数值成员函数，举例来说，包括 fetch_add, fetch_sub 等，同时通过重载方便的提供了对应的 +，- 版本。比如下面的例子：

```
#include <atomic>
#include <thread>
#include <iostream>

std::atomic<int> count = {0};

int main() {
    std::thread t1([](){
        count.fetch_add(1);
    });
    std::thread t2([](){
        count++;        // 等价于 fetch_add
        count += 1;     // 等价于 fetch_add
    });
    t1.join();
    t2.join();
    std::cout << count << std::endl;
```

```
    return 0;
}
```

当然，并非所有的类型都能提供原子操作，这是因为原子操作的可行性取决于具体的 CPU 架构，以及所实例化的类型结构是否能够满足该 CPU 架构对内存对齐条件的要求，因而我们总是可以通过 `std::atomic<T>::is_lock_free` 来检查该原子类型是否需支持原子操作，例如：

```
#include <atomic>
#include <iostream>

struct A {
    float x;
    int y;
    long long z;
};

int main() {
    std::atomic<A> a;
    std::cout << std::boolalpha << a.is_lock_free() << std::endl;
    return 0;
}
```

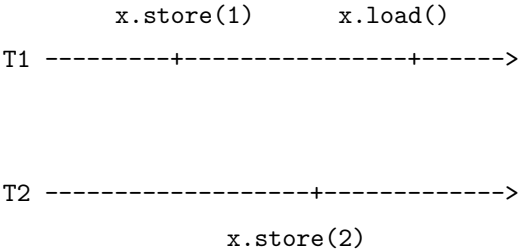
一致性模型

并行执行的多个线程，从某种宏观层面上讨论，可以粗略的视为一种分布式系统。在分布式系统中，任何通信乃至本地操作都需要消耗一定时间，甚至出现不可靠的通信。

如果我们强行将一个变量 v 在多个线程之间的操作设为原子操作，即任何一个线程在操作完 v 后，其他线程均能**同步**感知到 v 的变化，则对于变量 v 而言，表现为顺序执行的程序，它并没有由于引入多线程而得到任何效率上的收益。对此有什么办法能够适当的加速呢？答案便是削弱原子操作的在进程间的同步条件。

从原理上看，每个线程可以对应为一个集群节点，而线程间的通信也几乎等价于集群节点间的通信。削弱进程间的同步条件，通常我们会考虑四种不同的一致性模型：

1. 线性一致性：又称强一致性或原子一致性。它要求任何一次读操作都能读到某个数据的最近一次写的数据，并且所有线程的操作顺序与全局时钟下的顺序是一致的。



在这种情况下线程 T1, T2 对 x 的两次写操作是原子的,且 x.store(1) 是严格的发生在 x.store(2) 之前，x.store(2) 严格的发生在 x.load() 之前。值得一提的是，线性一致性对全局时钟的要求是难以实现的，这也是人们不断研究比这个一致性更弱条件下其他一致性的算法的原因。

2. 顺序一致性：同样要求任何一次读操作都能读到数据最近一次写入的数据，但未要求与全局时钟的顺序一致。

```
        x.store(1)  x.store(3)   x.load()
T1 ---------+-----------+----------+----->



T2 ---------------+---------------------->
              x.store(2)
```

或者

```
        x.store(1)  x.store(3)   x.load()
T1 ---------+-----------+----------+----->



T2 ------+------------------------------->
      x.store(2)
```

在顺序一致性的要求下,x.load() 必须读到最近一次写入的数据,因此 x.store(2) 与 x.store(1) 并无任何先后保障，即只要 T2 的 x.store(2) 发生在 x.store(3) 之前即可。

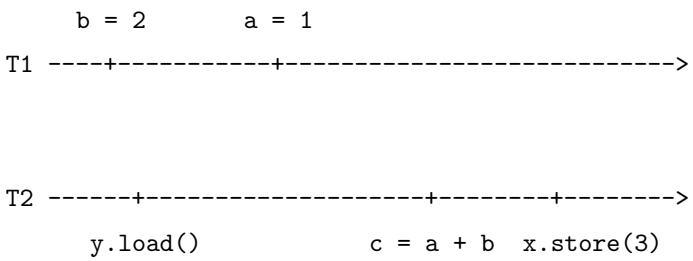
3. 因果一致性：它的要求进一步降低，只需要有因果关系的操作顺序得到保障，而非因果关系的操作顺序则不做要求。

```
      a = 1      b = 2
T1 ----+-----------+---------------------------->



T2 ------+--------------------+--------+-------->
      x.store(3)         c = a + b    y.load()
```

或者

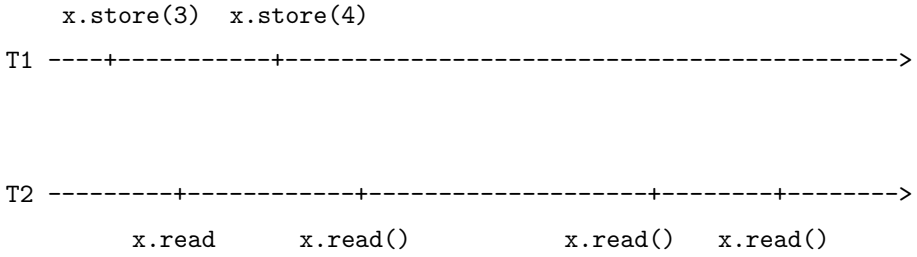
```
      a = 1      b = 2
T1 ----+-----------+---------------------------->



T2 ------+--------------------+--------+-------->
      x.store(3)          y.load()   c = a + b
```

亦或者



上面给出的三种例子都是属于因果一致的，因为整个过程中，只有 c 对 a 和 b 产生依赖，而 x 和 y 在此例子中表现为没有关系（但实际情况中我们需要更详细的信息才能确定 x 与 y 确实无关）

4. 最终一致性：是最弱的一致性要求，它只保障某个操作在未来的某个时间节点上会被观察到，但并未要求被观察到的时间。因此我们甚至可以对此条件稍作加强，例如规定某个操作被观察到的时间总是有界的。当然这已经不在我们的讨论范围之内了。



在上面的情况中，如果我们假设 x 的初始值为 0，则 T2 中四次 x.read() 结果可能但不限于以下情况：

```
3 4 4 4 // x 的写操作被很快观察到
0 3 3 4 // x 的写操作被观察到的时间存在一定延迟
0 0 0 4 // 最后一次读操作读到了 x 的最终值，但此前的变化并未观察到
0 0 0 0 // 在当前时间段内 x 的写操作均未被观察到，
        // 但未来某个时间点上一定能观察到 x 为 4 的情况
```

内存顺序

为了追求极致的性能，实现各种强度要求的一致性，C++11 为原子操作定义了六种不同的内存顺序 std::memory_order 的选项，表达了四种多线程间的同步模型：

1. 宽松模型：在此模型下，单个线程内的原子操作都是顺序执行的，不允许指令重排，但不同线程间原子操作的顺序是任意的。类型通过 std::memory_order_relaxed 指定。我们来看一个例子：

```
std::atomic<int> counter = {0};
std::vector<std::thread> vt;
for (int i = 0; i < 100; ++i) {
```

```
        vt.emplace_back([&](){
            counter.fetch_add(1, std::memory_order_relaxed);
        });
    }

    for (auto& t : vt) {
        t.join();
    }
    std::cout << "current counter:" << counter << std::endl;
```

2. 释放/消费模型：在此模型中，我们开始限制进程间的操作顺序，如果某个线程需要修改某个值，但另一个线程会对该值的某次操作产生依赖，即后者依赖前者。具体而言，线程 A 完成了三次对 x 的写操作，线程 B 仅依赖其中第三次 x 的写操作，与 x 的前两次写行为无关，则当 A 主动 x.release() 时候（即使用 std::memory_order_release），选项 std::memory_order_consume 能够确保 B 在调用 x.load() 时候观察到 A 中第三次对 x 的写操作。我们来看一个例子：

```
    // 初始化为 nullptr 防止 consumer 线程从野指针进行读取
    std::atomic<int*> ptr(nullptr);
    int v;
    std::thread producer([&]() {
        int* p = new int(42);
        v = 1024;
        ptr.store(p, std::memory_order_release);
    });
    std::thread consumer([&]() {
        int* p;
        while(!(p = ptr.load(std::memory_order_consume)));

        std::cout << "p: " << *p << std::endl;
        std::cout << "v: " << v << std::endl;
    });
    producer.join();
    consumer.join();
```

3. 释放/获取模型：在此模型下，我们可以进一步加紧对不同线程间原子操作的顺序的限制，在释放 std::memory_order_release 和获取 std::memory_order_acquire 之间规定时序，即发生在释放（release）操作之前的**所有**写操作，对其他线程的任何获取（acquire）操作都是可见的，亦即发生顺序（happens-before）。

   可以看到，std::memory_order_release 确保了它之前的写操作不会发生在释放操作之后，是一个向后的屏障（backward），而 std::memory_order_acquire 确保了它之前的写行为不会发生在该获取操作之后，是一个向前的屏障（forward）。对于选项 std::memory_order_acq_rel 而言，则结合了这两者的特点，唯一确定了一个内存屏障，使得当前线程对内存的读写不会被重排并越过此操作的前后：

我们来看一个例子：

```
std::vector<int> v;
std::atomic<int> flag = {0};
std::thread release([&]() {
    v.push_back(42);
    flag.store(1, std::memory_order_release);
});
std::thread acqrel([&]() {
    int expected = 1; // must before compare_exchange_strong
    while(!flag.compare_exchange_strong(expected, 2, std::memory_order_acq_rel))
        expected = 1; // must after compare_exchange_strong
    // flag has changed to 2
});
std::thread acquire([&]() {
    while(flag.load(std::memory_order_acquire) < 2);

    std::cout << v.at(0) << std::endl; // must be 42
});
release.join();
acqrel.join();
acquire.join();
```

在此例中我们使用了 compare_exchange_strong 比较交换原语（Compare-and-swap primitive），它有一个更弱的版本，即 compare_exchange_weak，它允许即便交换成功，也仍然返回 false 失败。其原因是因为在某些平台上虚假故障导致的，具体而言，当 CPU 进行上下文切换时，另一线程加载同一地址产生的不一致。除此之外，compare_exchange_strong 的性能可能稍差于 compare_exchange_weak，但大部分情况下，鉴于其使用的复杂度而言，compare_exchange_weak 应该被有限考虑。

4. 顺序一致模型：在此模型下，原子操作满足顺序一致性，进而可能对性能产生损耗。可显式的通过 std::memory_order_seq_cst 进行指定。最后来看一个例子：

```
std::atomic<int> counter = {0};
std::vector<std::thread> vt;
for (int i = 0; i < 100; ++i) {
    vt.emplace_back([&](){
        counter.fetch_add(1, std::memory_order_seq_cst);
    });
}

for (auto& t : vt) {
    t.join();
```

```
}
std::cout << "current counter:" << counter << std::endl;
```

这个例子与第一个宽松模型的例子本质上没有区别，仅仅只是将原子操作的内存顺序修改为了 `memory_order_seq_cst`，有兴趣的读者可以自行编写程序测量这两种不同内存顺序导致的性能差异。

总结

C++11 语言层提供了并发编程的相关支持，本节简单的介绍了 `std::thread`，`std::mutex`，`std::future` 这些并发编程中不可回避的重要工具。除此之外，我们还介绍了 C++11 最重要的几个特性之一的『内存模型』，它们为 C++ 在标准化高性能计算中提供了重要的基础。

习题

1. 请编写一个简单的线程池，提供如下功能：

```
ThreadPool p(4); // 指定四个工作线程

// 将任务在池中入队，并返回一个 std::future
auto f = pool.enqueue([](int life) {
    return meaning;
}, 42);

// 从 future 中获得执行结果
std::cout << f.get() << std::endl;
```

2. 请使用 `std::atomic<bool>` 实现一个互斥锁。

进一步阅读的参考资料

- C++ 并发编程 (中文版)
- 线程支持库文档
- Herlihy, M. P., & Wing, J. M. (1990). Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3), 463–492. https://doi.org/10.1145/78969.78972

第 8 章文件系统

文件系统库提供了文件系统、路径、常规文件、目录等等相关组件进行操作的相关功能。和正则表达式库类似，他也是最先由 boost 发起，并最终被合并为 C++ 标准的众多库之一。

8.1 文档与链接

TODO:

8.2 std::filesystem

TODO:

第 9 章其他杂项

9.1 新类型

`long long int`

`long long int` 并不是 C++11 最先引入的，其实早在 C99，`long long int` 就已经被纳入 C 标准中，所以大部分的编译器早已支持。C++11 的工作则是正式把它纳入标准库，规定了一个 `long long int` 类型至少具备 64 位的比特数。

9.2 noexcept 的修饰和操作

C++ 相比于 C 的一大优势就在于 C++ 本身就定义了一套完整的异常处理机制。然而在 C++11 之前，几乎没有人去使用在函数名后书写异常声明表达式，从 C++11 开始，这套机制被弃用，所以我们不去讨论也不去介绍以前这套机制是如何工作如何使用，你更不应该主动去了解它。

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
        << "lmay_throw() noexcept? " << noexcept(non_block_throw()) << std::endl
        << "lno_throw() noexcept? " << noexcept(block_throw()) << std::endl;
    return 0;
}
```

noexcept 修饰完一个函数之后能够起到封锁异常扩散的功效，如果内部产生异常，外部也不会触发。例如：

```
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;
}
```

最终输出为：

```
捕获异常，来自 may_throw()
```

捕获异常，来自 non_block_throw()

9.3 字面量

原始字符串字面量

传统 C++ 里面要编写一个充满特殊字符的字符串其实是非常痛苦的一件事情，比如一个包含 HTML 本体的字符串需要添加大量的转义符，例如一个 Windows 上的文件路径经常会：C:\\File\\To\\Path。

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，例如：

```
#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。

9.4 内存对齐

C++ 11 引入了两个新的关键字 alignof 和 alignas 来支持对内存对齐进行控制。alignof 关键字能够获得一个与平台相关的 std::size_t 类型的值，用于查询该平台的对齐方式。当然我们有时候并不满足于此，甚至希望自定定义结构的对齐方式，同样，C++ 11 还引入了 alignas 来重新修饰某个结构的对齐方式。我们来看两个例子：

```
#include <iostream>

struct Storage {
    char      a;
    int       b;
    double    c;
    long long d;
};

struct alignas(std::max_align_t) AlignasStorage {
    char      a;
    int       b;
    double    c;
    long long d;
};

int main() {
    std::cout << alignof(Storage) << std::endl;
    std::cout << alignof(AlignasStorage) << std::endl;
    return 0;
}
```

其中 std::max_align_t 要求每个标量类型的对齐方式严格一样，因此它几乎是最大标量没有差异，进而大部分平台上得到的结果为 long double，因此我们这里得到的 AlignasStorage 的对齐要求是 8 或 16。

总结

本节介绍的几个特性是从仍未介绍的现代 C++ 新特性里使用频次较靠前的特性了，noexcept 是最为重要的特性，它的一个功能在于能够阻止异常的扩散传播，有效的让编译器最大限度的优化我们的代码。

第 10 章展望：C++20 简介

C++20 如同 C++11 一样，似乎能够成为一个振奋人心的更新。例如，早在 C++11 时期就跃跃欲试呼声极高却最终落选的 Concept，如今已经箭在弦上。C++ 组委会在讨论投票最终确定 C++20 有很多提案，诸如 **Concepts/Module/Coroutine/Ranges/** 等等。本章我们就来一览 C++20 即将引入的那些重要特性。

概念与约束

概念（Concepts）是对 C++ 模板编程的进一步增强扩展。简单来说，概念是一种编译期的特性，它能够让编译器在编译期时对模板参数进行判断，从而大幅度增强我们在 C++ 中模板编程的体验。使用模板进行编程时候我们经常会遇到各种令人发指的错误，这是因为到目前为止我们始终不能够对模板参数进行检查与限制。举例而言，下面简单的两行代码会造成大量的几乎不可读的编译错误：

```
#include <list>
#include <algorithm>
int main() {
    std::list<int> l = {1, 2, 3};
    std::sort(l.begin(), l.end());
    return 0;
}
```

而这段代码出现错误的根本原因在于，std::sort 对排序容器必须提供随机迭代器，否则就不能使用，而我们知道 std::list 是不支持随机访问的。用概念的语言来说就是：std::list 中的迭代器不满足 std::sort 中随机迭代器这个概念的约束（Constraint）。在引入概念后，我们就可以这样对模板参数进行约束：

```
template <typename T>
requires Sortable<T> // Sortable 是一个概念
void sort(T& c);
```

缩写为：

```
template<Sortable T> // T 是一个 Sortable 的类型名
void sort(T& c)
```

甚至于直接将其作为类型来使用：

```
void sort(Sortable& c); // c 是一个 Sortable 类型的对象
```

我们现在来看一个实际的例子。

TODO: https://godbolt.org/z/9liFPD

模块

TODO:

合约

TODO:

范围

TODO:

协程

TODO:

事务内存

TODO:

总结

总的来说，终于在 C++20 中看到 Concepts/Ranges/Modules 这些令人兴奋的特性，这对于一门已经三十多岁『高龄』的编程语言，依然是充满魅力的。

进一步阅读的参考资料

- Why Concepts didn't make C++17?
- C++11/14/17/20 编译器支持情况
- C++ 历史

附录 1：进一步阅读的学习材料

首先，恭喜 你阅读完本书！笔者希望本书有提起你对现代 C++ 的兴趣。

正如本书引言部分提到的，本书只是一本带你快速领略现代 C++ 11/14/17/20 新特性的读物，而非进阶学习实践 C++『黑魔法』的内容。笔者当然也想到了这个需求，只是这样的内容非常艰深，鲜有受众。在此，笔者列出一些能够帮助你在此书基础之上进一步学习现代 C++ 的资料，希望能够祝你一臂之力：

- C++ 参考
- CppCon YouTube 频道
- Ulrich Drepper. 每位程序员都需要知道的内存知识. 2007
- 待补充

附录 2：现代 C++ 的最佳实践

这篇附录我们来简单谈一谈现代 C++ 的最佳实践。总的来说，笔者关于 C++ 的最佳实践相关的思考主要吸收自《Effective Modern C++》和 《C++ 风格指南》。在这篇附录里将简单讨论、并使用实际例子来阐明的方法，介绍一些笔者**个人的**、**不是随处可见的**、**非常识性**的最佳实践，并如何保证代码的整体质量。

常用工具

TODO:

代码风格

TODO:

整体性能

TODO:

代码安全

TODO:

可维护性

TODO:

可移植性

TODO: