

07 | 迭代器和好用的新for循环

2019-12-11 吴咏炜

现代C++实战30讲

[进入课程 >](#)



讲述：吴咏炜

时长 11:42 大小 8.05M



你好，我是吴咏炜。

我们已经讲过了容器。在使用容器的过程中，你也应该对迭代器（iterator）或多或少有了些了解。今天，我们就来系统地讲一下迭代器。

什么是迭代器？

迭代器是一个很通用的概念，并不是一个特定的类型。它实际上是一组对类型的要求（[\[1\]](#)）。它的最基本要求就是从一端点出发，下一步、下一步地到达另一端点。按照一般的中文习惯，也许“遍历”是比“迭代”更好的用词。我们可以遍历一个字符串的字符，遍历一个文件的内容，遍历目录里的所有文件，等等。这些都可以用迭代器来表达。

我在用 `output_container.h` 输出容器内容的时候，实际上就对容器的 `begin` 和 `end` 成员函数返回的对象类型提出了要求。假设前者返回的类型是 `I`，后者返回的类型是 `S`，这些要求是：

`I` 对象支持 `*` 操作，解引用取得容器内的某个对象。

`I` 对象支持 `++`，指向下一个对象。

`I` 对象可以和 `I` 或 `S` 对象进行相等比较，判断是否遍历到了特定位置（在 `S` 的情况下是否结束了遍历）。

注意在 C++17 之前，`begin` 和 `end` 返回的类型 `I` 和 `S` 必须是相同的。从 C++17 开始，`I` 和 `S` 可以是不同的类型。这带来了更大的灵活性和更多的优化可能性。

上面的类型 `I`，多多少少就是一个满足输入迭代器（input iterator）的类型了。不过，`output_container.h` 只使用了前置 `++`，但输入迭代器要求前置和后置 `++` 都得到支持。

输入迭代器不要求对同一迭代器可以多次使用 `*` 运算符，也不要求可以保存迭代器来重新遍历对象，换句话说，只要求可以单次访问。如果取消这些限制、允许多次访问的话，那迭代器同时满足了前向迭代器（forward iterator）。

一个前向迭代器的类型，如果同时支持 `--`（前置及后置），回到前一个对象，那它就是个双向迭代器（bidirectional iterator）。也就是说，可以正向遍历，也可以反向遍历。

一个双向迭代器，如果额外支持在整数类型上的 `+`、`-`、`+=`、`-=`，跳跃式地移动迭代器；支持 `[]`，数组式的下标访问；支持迭代器的大小比较（之前只要求相等比较）；那它就是个随机访问迭代器（random-access iterator）。

一个随机访问迭代器 `i` 和一个整数 `n`，在 `*i` 可解引用且 `i + n` 是合法迭代器的前提下，如果额外还满足 `*(addressof(*i) + n)` 等价于 `*(i + n)`，即保证迭代器指向的对象在内存里是连续存放的，那它（在 C++20 里）就是个连续迭代器（contiguous iterator）。

以上这些迭代器只考虑了读取。如果一个类型像输入迭代器，但 `*i` 只能作为左值来写而不能读，那它就是个输出迭代器（output iterator）。

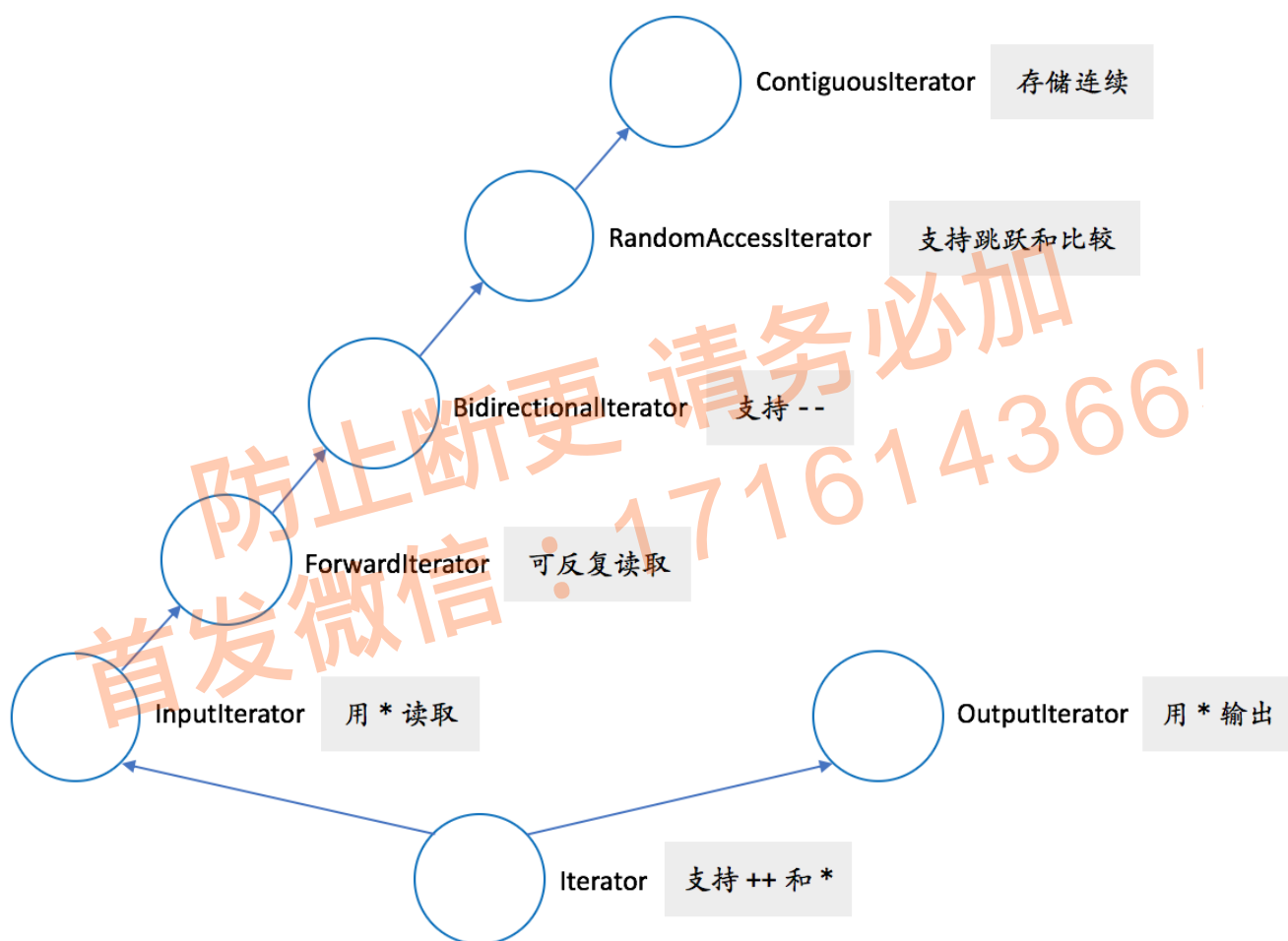
而比输入迭代器和输出迭代器更底层的概念，就是迭代器了。基本要求是：

对象可以被拷贝构造、拷贝赋值和析构。

对象支持 `*` 运算符。

对象支持前置 `++` 运算符。

迭代器类型的关系可从下图中全部看到：



迭代器通常是对象。但需要注意的是，指针可以满足上面所有的迭代器要求，因而也是迭代器。这应该并不让人惊讶，因为本来迭代器就是根据指针的特性，对其进行抽象的结果。事实上，`vector` 的迭代器，在很多实现里就直接是使用指针的。

常用迭代器

最常用的迭代器就是容器的 `iterator` 类型了。以我们学过的顺序容器为例，它们都定义了嵌套的 `iterator` 类型和 `const_iterator` 类型。一般而言，`iterator` 可写入，

`const_iterator` 类型不可写入，但这些迭代器都被定义为输入迭代器或其派生类型：

`vector::iterator` 和 `array::iterator` 可以满足到连续迭代器。

`deque::iterator` 可以满足到随机访问迭代器（记得它的内存只有部分连续）。

`list::iterator` 可以满足到双向迭代器（链表不能快速跳转）。

`forward_list::iterator` 可以满足到前向迭代器（单向链表不能反向遍历）。


很常见的一个输出迭代器是 `back_inserter` 返回的类型 `back_inserter_iterator` 了；用它我们可以很方便地在容器的尾部进行插入操作。另外一个常见的输出迭代器是 `ostream_iterator`，方便我们把容器内容“拷贝”到一个输出流。示例如下：

 复制代码

```
1 #include <algorithm> // std::copy
2 #include <iterator>  // std::back_inserter
3 #include <vector>     // std::vector
4 using namespace std;
```

 复制代码

```
1 vector<int> v1{1, 2, 3, 4, 5};
2 vector<int> v2;
3 copy(v1.begin(), v1.end(),
4      back_inserter(v2));
```

 复制代码

```
1 v2
```

```
{ 1, 2, 3, 4, 5 }
```

 复制代码

```
1 #include <iostream> // std::cout
2 copy(v2.begin(), v2.end(),
3      ostream_iterator<int>(cout, " "));
```

使用输入行迭代器

下面我们来看下一个我写的输入迭代器。它的功能本身很简单，就是把一个输入流（`istream`）的内容一行行读进来。配上 C++11 引入的基于范围的 `for` 循环的语法，我们可以把遍历输入流的代码以一种自然、非过程式的方式写出来，如下所示：

[复制代码](#)

```
1 for (const string& line :  
2     istream_line_reader(is)) {  
3     // 示例循环体中仅进行简单输出  
4     cout << line << endl;  
5 }
```

我们可以对比一下以传统的方式写的 C++ 代码，其中需要照顾不少细节：

[复制代码](#)

```
1 string line;  
2 for (;;) {  
3     getline(is, line);  
4     if (!is) {  
5         break;  
6     }  
7     cout << line << endl;  
8 }
```

从 `is` 读入输入行的逻辑，在前面的代码里一个语句就全部搞定了，在这儿用了 5 个语句.....

我们后面会分析一下这个输入迭代器。在此之前，我先解说一下基于范围的 `for` 循环这个语法。虽然这可以说是个语法糖，但它对提高代码的可读性真的非常重要。如果不用这个语法糖的话，简洁性上的优势就小多了。我们直接把这个循环改写成等价的普通 `for` 循环的样子。

[复制代码](#)

```
1 {  
2     auto&& r = istream_line_reader(is);
```

```
3   auto it = r.begin();
4   auto end = r.end();
5   for (; it != end; ++it) {
6       const string& line = *it;
7       cout << line << endl;
8   }
9 }
```

可以看到，它做的事情也不复杂，就是：

获取冒号后边的范围表达式的结果，并隐式产生一个引用，在整个循环期间都有效。注意根据生命期延长规则，表达式结果如果是临时对象的话，这个对象要在循环结束后才被销毁。

自动生成遍历这个范围的迭代器。

循环内自动生成根据冒号左边的声明和 `*it` 来进行初始化的语句。

下面就是完全正常的循环体。

生成迭代器这一步有可能是——但不一定是——调用 `r` 的 `begin` 和 `end` 成员函数。具体规则是：

对于 C 数组（必须是没有退化为指针的情况），编译器会自动生成指向数组头尾的指针（相当于自动应用可用于数组的 `std::begin` 和 `std::end` 函数）。

对于有 `begin` 和 `end` 成员的对象，编译器会调用其 `begin` 和 `end` 成员函数（我们目前的情况）。

否则，编译器会尝试在 `r` 对象所在的名空间寻找可以用于 `r` 的 `begin` 和 `end` 函数，并调用 `begin(r)` 和 `end(r)`；找不到的话则失败报错。

定义输入行迭代器

下面我们看一下，要实现这个输入行迭代器，需要做些什么工作。

C++ 里有些固定的类型要求规范。对于一个迭代器，我们需要定义下面的类型：


```

1  class istream_line_reader {
2  public:
3      class iterator { // 实现 InputIterator
4      public:
5          typedef ptrdiff_t difference_type;
6          typedef string value_type;
7          typedef const value_type* pointer;
8          typedef const value_type& reference;
9          typedef input_iterator_tag
10             iterator_category;
11         ...
12     };
13     ...
14 };

```

仿照一般的容器，我们把迭代器定义为 `istream_line_reader` 的嵌套类。它里面的这五个类型是必须定义的（其他泛型 C++ 代码可能会用到这五个类型；之前标准库定义了一个可以继承的类模板 `std::iterator` 来产生这些类型定义，但这个类目前已经被废弃 [2]）。其中：

`difference_type` 是代表迭代器之间距离的类型，定义为 `ptrdiff_t` 只是种标准做法（指针间差值的类型），对这个类型没什么特别作用。

`value_type` 是迭代器指向的对象的值类型，我们使用 `string`，表示迭代器指向的是字符串。

`pointer` 是迭代器指向的对象的指针类型，这儿就平淡无奇地定义为 `value_type` 的常指针了（我们可不希望别人来更改指针指向的内容）。

类似的，`reference` 是 `value_type` 的常引用。

`iterator_category` 被定义为 `input_iterator_tag`，标识这个迭代器的类型是 `input iterator`（输入迭代器）。

作为一个真的只能读一次的输入迭代器，有个特殊的麻烦（前向迭代器或其衍生类型没有）：到底应该让 `*` 负责读取还是 `++` 负责读取。我们这儿采用常见、也较为简单的做法，让 `++` 负责读取，`*` 负责返回读取的内容（这个做法会有些副作用，但按我们目前的用法则没有问题）。这样的话，这个 `iterator` 类需要有一个数据成员指向输入流，一个数据成员来存放读取的结果。根据这个思路，我们定义这个类的基本成员函数和数据成员：

```


1  class istream_line_reader {
2  public:
3      class iterator {
4          ...
5          iterator() noexcept
6              : stream_(nullptr) {}
7          explicit iterator(istream& is)
8              : stream_(&is)
9          {
10             ++*this;
11         }
12
13         reference operator*() const noexcept
14         {
15             return line_;
16         }
17         pointer operator->() const noexcept
18         {
19             return &line_;
20         }
21         iterator& operator++()
22         {
23             getline(*stream_, line_);
24             if (!*stream_) {
25                 stream_ = nullptr;
26             }
27             return *this;
28         }
29         iterator operator++(int)
30         {
31             iterator temp(*this);
32             ++*this;
33             return temp;
34         }
35
36     private:
37         istream* stream_;
38         string line_;
39     };
40     ...
41 };

```

我们定义了默认构造函数，将 `stream_` 清空；相应的，在带参数的构造函数里，我们根据传入的输入流来设置 `stream_`。我们也定义了 `*` 和 `->` 运算符来取得迭代器指向的文本行的引用和指针，并用 `++` 来读取输入流的内容（后置 `++` 则以惯常方式使用前置 `++` 和拷贝构造来实现）。唯一“特别”点的地方，是我们在构造函数里调用了 `++`，确保在构造后调

用 * 运算符时可以读取内容，符合日常先使用 *、再使用 ++ 的习惯。一旦文件读取到尾部（或出错），则 stream_ 被清空，回到默认构造的情况。

对于迭代器之间的比较，我们则主要考虑文件有没有读到尾部的情况，简单定义为：

 复制代码

```
1     bool operator==(const iterator& rhs)
2         const noexcept
3     {
4         return stream_ == rhs.stream_;
5     }
6     bool operator!=(const iterator& rhs)
7         const noexcept
8     {
9         return !operator==(rhs);
10    }
```

有了这个 iterator 的定义后，istream_line_reader 的定义就简单得很了：

 复制代码

```
1 class istream_line_reader {
2 public:
3     class iterator {...};
4     istream_line_reader() noexcept
5         : stream_(nullptr) {}
6     explicit istream_line_reader(
7         istream& is) noexcept
8         : stream_(&is) {}
9     iterator begin()
10    {
11        return iterator(*stream_);
12    }
13    iterator end() const noexcept
14    {
15        return iterator();
16    }
17
18 private:
19     istream* stream_;
20 };
```

也就是说，构造函数只是简单地把输入流的指针赋给 `stream_` 成员变量。`begin` 成员函数则负责构造一个真正有意义的迭代器；`end` 成员函数则只是返回一个默认构造的迭代器而已。

以上就是一个完整的基于输入流的行迭代器了。这个行输入模板的设计动机和性能测试结果可参见参考资料 [3] 和 [4]；完整的工程可用代码，请参见参考资料 [5]。该项目中还提供了利用 C 文件接口的 `file_line_reader` 和基于内存映射文件的 `mmap_line_reader`。

内容小结

今天我们介绍了所有的迭代器类型，并介绍了基于范围的 `for` 循环。随后，我们介绍了一个实际的输入迭代器工具，并用它来简化从输入流中读入文本行这一常见操作。最后，我们展示了这个输入迭代器的定义。

课后思考

请思考一下：

1. 目前这个输入行迭代器的行为，在什么情况下可能导致意料之外的后果？
2. 请尝试一下改进这个输入行迭代器，看看能不能消除这种意外。如果可以，该怎么做？
如果不可以，为什么？

欢迎留言和我交流你的看法。

参考资料

[1] [cppreference.com](https://en.cppreference.com/w/cpp/iterator), “Iterator library” .

🔗 <https://en.cppreference.com/w/cpp/iterator>

[1a] [cppreference.com](https://en.cppreference.com/w/cpp/iterator), “迭代器库” . 🔗 <https://zh.cppreference.com/w/cpp/iterator>

[2] Jonathan Boccara, “std::iterator is deprecated: why, what it was, and what to use instead” . 🔗 <https://www.fluentcpp.com/2018/05/08/std-iterator-deprecated/>

[3] 吴咏炜, "Python `yield` and C++ coroutines" .

<https://yongweiwu.wordpress.com/2016/08/16/python-yield-and-cplusplus-coroutines/>

[4] 吴咏炜, "Performance of my line readers" .

<https://yongweiwu.wordpress.com/2016/11/12/performance-of-my-line-readers/>

[5] 吴咏炜, nvwa. <https://github.com/adah1972/nvwa/>


点击查看 

打卡学习 C++ 拒绝从入门到放弃



PC 端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 06 | 异常：用还是不用，这是个问题

下一篇 08 | 易用性改进 I：自动类型推断和初始化

精选留言 (19)

 写留言



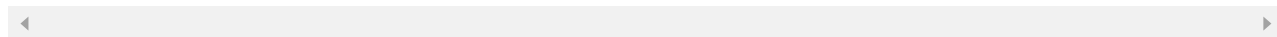
nelson

2019-12-12

如果stream_是nullptr会怎么样？

展开 ▾

作者回复: 得到一个空的不能遍历的迭代器。跟任何 `end()` 相等比较返回真, 因而你不可以对它做 `++` 操作。如果你要硬来, 它就死给你看。



💬 1

👍 1



小一日一

2019-12-11

看了老师的代码再看自己学的代码, 感觉我的C++是小学生水平。

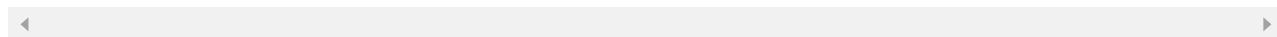
以为自己看过几遍C++ PRIMER 5th, 看过并理解effective++, more effective c++, inside the c++ object model, 能应付平时的开发需要, 也能看懂公司别人的代码, 就觉得自己C++不错了, 看了老师github的代码后我是彻底服了, 感叹C++太博大精深, 永远...

展开 ▾

作者回复: 肯定还有更好的 C++ 代码的。学习无止境!

认真学习, 应该不用那么久 (我还没有极客时间专栏来帮助我学习呢 😊)。

反过来, 说明老程序员还有点价值么。😏



💬

👍 1



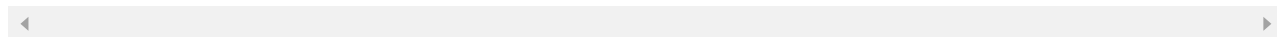
晚风·和煦

2019-12-11

从 C++17 开始, `l` 和 `s` 可以是不同的类型。这带来了更大的灵活性和更多的优化可能性。没太理解这句话 😊

展开 ▾

作者回复: 现在 `r.begin()` 和 `r.end()` 可以是不同类型了。



💬

👍 1



干鲤湖

2019-12-18

过来看看老师问的那两个问题, 好奇中。。。

展开 ▾

作者回复: 公布第 1 个问题的答案吧:

```
#include <fstream>
#include <iostream>
#include "istream_line_reader.h"

using namespace std;

int main()
{
    ifstream ifs{"test.cpp"};
    istream_line_reader reader{ifs};
    auto begin = reader.begin();
    for (auto it = reader.begin();
         it != reader.end(); ++it) {
        cout << *it << '\n';
    }
}
```

以上代码, 因为 begin 多调用了一次, 输出就少了一行.....



总统老唐

2019-12-16

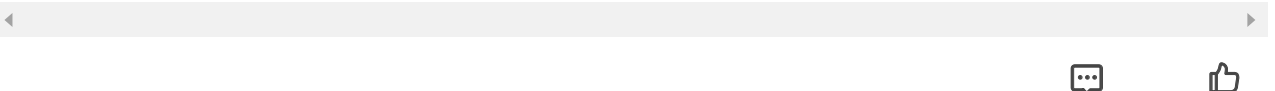
吴老师, 这一课有两个疑问:

- 1, “到底应该让 * 负责读取还是 ++ 负责读取”, 该怎样理解? 如果“读取”指的是在 istream 上读取一行, 放入 line_成员中, 用 ++ 实现这个操作是最常见和直觉的, 同时, 用 * 返回读取的内容也在最容易想到的方式, 反过来, 什么情况下会需要“用 * 来负责读取”?
- 2, 输入迭代器为什么要定义 iterator operator++(int)

展开 ∨

作者回复: 1. 用 ++ 是最合理的, 但也有一个奇怪的地方, 目前还没人说到。

2. 这个就是后置 ++。迭代器要求前置和后置 ++ 都要定义, 虽然我目前只使用了前置版本。



干鲤湖

1. 可能是operator==中，比较时没有获取当前文件流位置，这样的话，无法比较不同istream(同一个文件)创建的iterator?

2 采用ftell获取当前文件流位置

展开 ∨

作者回复: 不是我想的那个.....

这个是个问题，但一般不必解决。要能够比较，对性能影响太大。我线上的版本里是有下面这段注释的：

```
// This implementation basically says, any iterators
// pointing to the same stream are equal. This behaviour
// may seem a little surprising in the beginning, but, in
// reality, it hardly has any consequences, as people
// usually compare an input iterator only to the sentinel
// object. The alternative, using _M_stream->tellg() to
// get the exact position, harms the performance too dearly.
// I do not really have a better choice.
//
// If you do need to compare valid iterators, consider using
// file_line_reader or mmap_line_reader.
```



禾桃

2019-12-15

#1 目前这个输入行迭代器的行为，在什么情况下可能导致意料之外的后果？

```
auto x = istream_line_reader();
```

```
auto xit = x.begin();
```

这个函数会调用istream_line_reader::iterator::operator++() {
getline(*nullptr, _M_line); <---- 死翘翘 }...

展开 ∨

作者回复: #1 对我来讲，这不是意外。就像你对空指针解引用崩溃也不是意外一样。没有有效的istream，你要取这个流的开头，出错很正常。

#2 因为你没有看到我想的问题，所以第二部分也不是我要的回答.....



MT

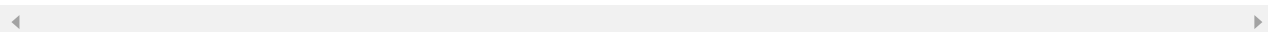
2019-12-14

老师，这次是上次关于那个例子的补充：

1. 在进行迭代的时候，begin()和end()方法，即你所说的，编译器会自动生成指向数组头尾的指针
 2. 在end()方法内返回了struct null_sentinel{} 的一个对象，即 I 和 S 的类型不同
 3. 通过使用 struct null_sentinel{}; 所提供的operator!=() 从而达到对字符串遍历的截至...
- 展开 ∨

作者回复: 前面部分没有问题。后面的失败部分，没看懂你的意思。

我这个例子重点在于，null_sentinel 表示的不是一个位置，而是一个条件。我们可以用迭代器来表示一个条件，这是对它的功能的很大扩展。虽然这种扩展方式性能非常好，但这个功能主要不是优化，而是新的可能性。



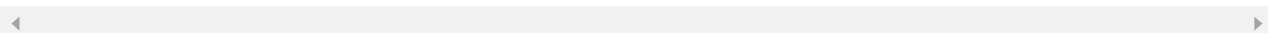
旭东

2019-12-14

老师，您好，iterater中后置++的实现是不是应该返回const; 避免 (i++)++这样的代码通过编译？

作者回复: 1. 不能写 const，因为你修改了自己。

2. 就算能写也防不了，因为你返回的是个全新的对象。



木瓜777

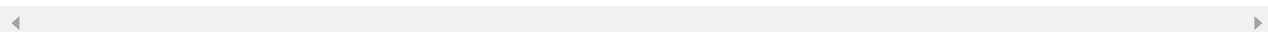
2019-12-13

```
iterator operator++(int) { iterator temp(*this); ++*this; return temp; }
```

这个拷贝构造，是否会出问题？ 如果失败，this继续读取下一行，但temp是异常的。

展开 ∨

作者回复: 拷贝构造失败的话，直接抛异常了，当然不会继续读取下一行。





我叫bug谁找我

2019-12-13

遍历一遍后，第二次调用begin会崩溃，stream_指针已经为空

作者回复: 作为input iterator，本来你就不应该遍历第二次的。这个不是问题。



MT

2019-12-12

老师，可以讲以下为什么可以将I 和 S 设置成不同的类型吗？具体使用在那些方面？

作者回复: 给个例子你仔细研读一下吧。功能是遍历字符串，直到遇到字符串结尾（事先不知道字符串长度）。

```
#include <stdio.h>
```

```
struct null_sentinel {};
```

```
bool operator!=(const char* ptr, null_sentinel)
{
    return *ptr != 0;
}
```

```
// operator!=(null_sentinel, const char* ptr), operator==, ...
```

```
struct c_string_view {
    c_string_view(const char* str) : str_(str) {}
    const char* begin() const { return str_; }
    null_sentinel end() const { return null_sentinel{}; }
    const char* str_;
};
```

```
int main()
{
    c_string_view msg{"Hello world!"};
    for (char ch : msg) {
        putchar(ch);
    }
}
```

```
    putchar('\n');  
}
```



Scott

2019-12-12

我的理解是istream_line_reader的iterator在到达end时，再++会直接crash，这个和STL里面主流容器的行为是不一致的。

可以在get_line之前，判断一下stream_是否为nullptr，不是才调用，对end的iterator反复进行++都一直返回自己本身。

展开 ▾

作者回复: 你对容器的 end() 解引用，同样可能崩溃（取决于实现）。你不被允许这么做。这么做，你就进入了 undefined behavior 的领域，系统是死还是出 bug 都正常。



禾桃

2019-12-11

输入迭代器和输出迭代器，
这个入和出是相对于什么而言的？
感觉有点绕。

谢谢！

展开 ▾

作者回复: cout << *it 就是读；
*it = 42 就是写。



tt

2019-12-11

意料之外的后果，是不是主要就是资源发生了不可控或不可知的泄露或状态改变？

这里的资源我觉得一是string对象，一个是istream对象，那么在这两个对象的内存管理上会引起问题？

...

展开 ▾

作者回复: 你说的情况会出问题, 但这个是需要调用者保证的, 我做不了什么事情。

再想想。🤔



小一日一

2019-12-11

1. 我能想到的一点是, `istream_line_reader`在构造时没有对输入流的状态做检测, 如果在输入流处于错误状态时调用`getline()`, 会抛 `ios_base::failure`异常。

2. 我把带输入流检查的构造贴一下:

```
istream_line_reader() noexcept : stream_(nullptr) {}  
explicit istream_line_reader( istream& is) noexcept { ...
```

展开 ▾

作者回复: 抛异常是很正常的呀, 不是问题。不过, IO streams 缺省应该是不抛异常的。



廖熊猫

2019-12-11

我认为是因为stream操作有副作用吧, 在使用++还是*读取的时候有提到, 每次读取的话都会受到影响, 如果我在使用迭代器之前操作了stream, 这个迭代器的操作范围就不是我预期的范围了。迭代器玩法很多啊, 有一本《Functional C++》讲了很多基于范围的操作, 不过水平不够, 没学到什么精髓。

展开 ▾

作者回复: 你说的用法我觉得还是不会让人惊讶的.....

你是说 Ivan Čukić 写的 Functional Programming in C++ 吗? 我是那本书的技术校对.....嗯, 我当然推荐它的。Range 下面会单独有一讲。

学语言, 还是要多读多写多练。



Geek_71d4ac

2019-12-11

在构造函数中使用this是否安全？万一构造中途失败了呢？

作者回复: 本身没有任何问题。如何保证行为安全（如异常安全）是个独立问题，跟是否在构造函数里没啥关系。尽量不使用裸指针非常重要，用了的话就需要照顾很多细节了.....



未来、尽在我手

2019-12-11

老师，可以讲讲auto？

我一直很期待新的for，可是没看到在哪？

这儿就详细介绍了迭代器及各种不同的迭代器。

展开 ▾

作者回复: auto 正是下一讲的主要内容。第二个问题，在正文里搜“基于范围的 for 循环”。代码例子没细看么？ 😊

1

