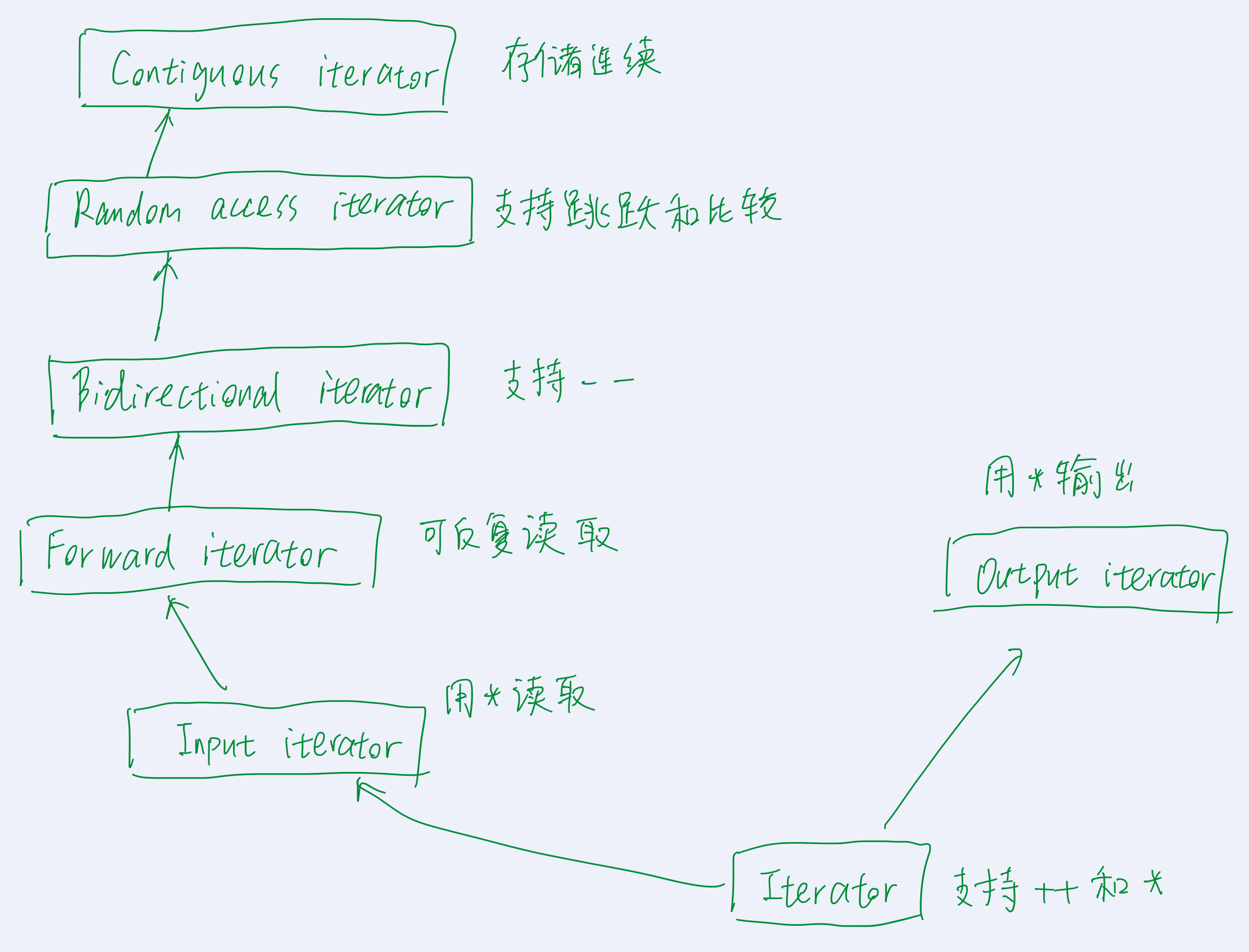
Modern cpp 30 lectures — for loop

Saturday, December 19, 2020 8:16 AM

迭代器：

- 组对类型的要求，从一个端点出发，下一步，下一步地到达另一个端点，
- 前向迭代器 forward iterator 允许多次访问
- 一个前向迭代器，如果同时支持 -- (前置及后置)，回到前一个对象，就是个双向迭代器 bidirectional iterator，可以正向遍历，也可以反向遍历。
- 双向迭代器，如果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)，它就是个随机访问迭代器 random_access iterator.
- 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 *(addressdof(*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，那它就是个连续迭代器 Contiguous iterator
- 如果一个类型像输入迭代器，但 *i 只能做为左值来写而不能读，那它就是个输出迭代器 output iterator.
- 比输入迭代器和输出迭代器更底层的概念，就是迭代器了，
  - 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.



常用迭代器.

最常用的是 iterator 类型. 顺序容器都定义了

- iterator 类型，可写入.
- const_iterator 不可写入.
- vector::iterator 和 array::iterator 可满足连续迭代器.
- deque::iterator 可满足随机访问迭代器.
- list::iterator 可满足双向迭代器.
- forward_list::iterator 可满足前向迭代器.

很常见的一个输出迭代器是 back_inserter 返回的类型 back_inserter_iterator

用它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：

```
#include <algorithm> // std::copy
#include <iterator> // std::back_inserter
#include <vector> // std::vector

using namespace std;

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

#include <iostream> // std::cout
copy(v2.begin(), v2.end(), ostream_iterator<int>(cout, " "));
Output: 1 2 3 4 5
```

使用输出迭代器.

把一个输出流 istream 的内容一行行读来

```
for(const string& line: istream_line_reader(is)) {
    cout << line << endl;
}
```

对比传统方式，需要照顾不少细节

```
string line;
for(;;) {
   getline(is, line);
   if(!is) {
      break;
   }
   cout << line << endl;
}
```

使用传统的 for 循环，也可等价写为

```
{
  auto&& r = istream_line_reader(is);
  auto it = r.begin();
  auto end = r.end();
  for(; it != end; ++it) {
     const string& line = *it;
     cout << line << endl;
  }
}
```

隐式产生一个引用在整个循环区间都有效.

定义输入行迭代器

```
class istream_line_reader {
public:
   class iterator { // 实现 input iterator
   public:
       typedef ptrdiff_t difference_type;
       typedef string value_type;
       typedef const value_type* pointer;
       typedef const value_type& reference;
       typedef input_iterator_tag iterator_category;
       ...
   };
   ...
};
```

仿照一般容器，把迭代器定义为 istream_line_reader 的嵌套类，

它里面的5个类型是必须定义的，

- 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)
- value_type 是迭代器指向的对象的值类型，这里是 string，表示迭代器指向的是 string.
- pointer 是迭代器指向的对象的指针类型，这里定义为 value_type 的常指针，因为不希望别人更改指针指向的内容
- reference 是 value_type 的常引用.
- iterator_category 被定义为 input_iterator_tag，标识这个迭代器是 input iterator.

让 ++ 负责读取，* 返回读取内容，iterator 类需要有一个数据成员指向输入流，一个数据成员存放读取结果.

```
class istream_line_reader {
public:
  class iterator {
    ...
    iterator() noexcept
      : stream_(nullptr) {}
    explicit iterator(istream& is)
      : stream_(&is)
    {
      ++*this;
    }

    reference operator*() const noexcept
    {
      return line_;
    }

    pointer operator->() const noexcept
    {
      return &line_;
    }

     iterator& operator++()
     {
       getline(*stream_, line_);
       if(!*stream_) {
         stream_ = nullptr;
       }
       return *this;
     }

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

  private:
     istream* stream_;
     string line_;
  };
  ...
};
```

默认构造函数，将 stream_ 清空

在带参数的构造函数里，根据传入的输入流设置 stream_.

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，符合日常先用*，再用++的习惯.

定义*和→运算符来取得迭代器指向的文本行的引用和指针，

用++读取输入流的内容(后置++则以惯常方式使用前置++和拷贝构造实现)

一旦文件读取到尾部，则 stream_ 被清空，回到默认构造的情况.

对于迭代器之间的比较，主要考虑文件有没有读到尾部的情况，

```
bool operator==(const iterator& rhs)
   const noexcept
{
   return stream_ == rhs.stream_;
}

bool operator!=(const iterator& rhs)
  const noexcept
{
  return !operator==(rhs);
}
```

有了 iterator 的定义后，istream_line_reader 的定义就很简单了：

```
class istream_line_reader {
public:
  class iterator {...};
  istream_line_reader() noexcept
   : stream_(nullptr) {}
  explicit istream_line_reader(
     istream& is) noexcept
     : stream_(&is) {}

  iterator begin()
  {
    return iterator(*stream_);
  }
  iterator end() const noexcept
  {
    return iterator();
  }

private:
  istream* stream_;
};
```

构造函数只是简单地把输入流的指针赋给 stream_ 成员变量.

begin 成员函数负责构造一个真正有意义的迭代器.

end 成员函数只是返回一个默认构造的迭代器.

目前这个迭代器的行为，在什么情况下可能导致意料之外的后果？

```
#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';
  }
}
```

因为 begin 多调用了一次，输出就少了一行.

Tips:

- cout << *it; 就是输出迭代器
- *it = 42; 就是输入迭代器.