深蓝学院
shenlanxueyuan.com

C++ 基础

第 7 章：深入 IO

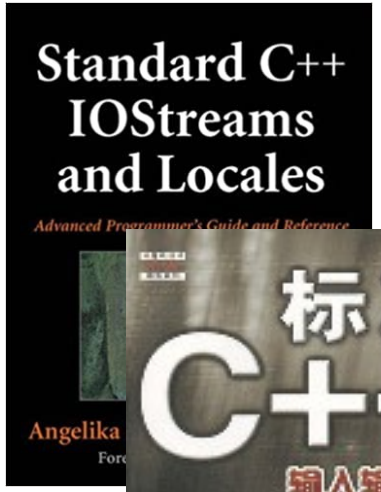
主讲人 李伟

微软高级工程师

《C++ 模板元编程实战》作者

目录

推荐看这个英文版的，非常好



不推荐这个中文版的。翻译的不好

标准 C++ IOStreams 与 Locales

Angelika Langer & Klaus Kreft 著
李伟译 (V2.0)

2015 年 6 月 20 日

IOStream 概述

- IOStream 采用流式 I/O 而非记录 I/O ，但可以在此基础上引入结构信息
- 所处理的两个主要问题
  - 表示形式的变化：使用格式化 / 解析在数据的内部表示与字符序列间转换
  - 与外部设备的通信：针对不同的外部设备（终端、文件、内存）引入不同的处理逻辑
- 所涉及到的操作
  - 格式化 / 解析
  - 缓存
  - 编码转换
  - 传输
- 采用模板来封装字符特性，采用继承来封装设备特性
  - 常用的类型实际上是类模板实例化的结果

```
ifstream实际上是 std::basic_ifstream<char, std::char_traits<char>>的别名
```

输入与输出

```
int x;
std::cin.read(reinterpret_cast<char*>(&x). sizeof(x));
std::cout << x << std::endl;
// 输入100 输出 170930225 （因为没有格式化）
// 使用非格式化的例子：我们希望无论float的值发生什么改变，我们希望float在输入的文件中只占某个特定的字节数
float y = 3.1415; //为了保证输出是特定的字符数，就可以使用read
```

- 输入与输出分为格式化与非格式化两类

非格式化使用的比较少,因为不是人类易于理解的形式

- 非格式化 I/O ： 不涉及数据表示形式的变化

get:读取单个字符,read: 读取多个字符，getline: 读取一行 gcount:返回输入元素的个数

  - 常用输入函数： get / read / getline / gcount

put:写一个字符，write: 写多个字符

  - 常用输出函数： put / write

- 格式化 I/O ： 使用移位操作符来进行的输入 (>>) 与输出 (<<)

```
char a = '0';
int x = 48;
std::cout.seff(std::ios_base::showpos);
std::cout << x << std::endl; // +48

std::cout.width(10);
std::cout << a << std::endl; //         0;

std::cout.fill('.');
std::cout << a << std::endl; // .........0;
```

  - C++ 通过操作符重载以支持内建数据类型的格式化 I/O
  - 可以通过重载操作符以支持自定义类型的格式化 I/O

- 格式控制

show positive 展示正值

  - 可接收位掩码类型（ showpos ）、字符类型（ fill ）与取值相对随意（ width ）的格式化参数
  - 注意 width 方法的特殊性：触发后被重置

输入与输出（续）

```
char a = '0'; int x = static_cast<char>(a);
std::cout << std::showpos << << std::setw(10)
<< std::setfill('.') << a << "\n" << x << '\n';
```

- 操纵符
  - 简化格式化参数的设置
  - 触发实际的插入与提取操作

- 提取会放松对格式的限制

- 提取 C 风格字符串时要小心内存越界

```
char x[5];
std::cin >> x; //可能会崩溃
std::cout << x << std::endl;
// 改进方法：
std::cin >> std::setw(5) >> x;
std::cout << x << std::endl;
```

表 1.4: 操纵符

操纵符	影响	用途	等价表达式	参考
flush	o	刷新流缓冲区	o.flush()	0
endl	o	插入新行并刷新流缓冲区	o.put(o.widen('\n')); o.flush()	0
ends	o	插入字符串结束符	o.put(o.widen('\0'))	0
ws	i	提取空白字符		I
boolalpha	io	设置标记，用字母形式表示bool值	io.setf (ios_base::boolalpha)	B
noboolalpha	io	复位上个标记	io.unsetf (ios_base::boolalpha)	B
showbase	o	设置标记，输出表示整数进制的前缀	o.setf (ios_base::showbase)	B
noshowbase	o	复位上个标记	io.unsetf (ios_base::showbase)	B
showpoint	o	设置标记，总是显示浮点数小数点	o.setf (ios_base::showpoint)	B
noshowpoint	o	复位上个标记	o.unsetf (ios_base::showpoint)	B
showpos	o	设置标记，输出非负数的同时输出+	o.setf (ios_base::showpos)	B
noshowpos	o	复位上个标记	o.unsetf (ios_base::showpos)	B
skipws	i	设置标记，跳过前面的空白字符	i.setf (ios_base::skipws)	B
noskipws	i	复位上个标记	i.unsetf	B
uppercase	o	设置标记，使数值格式化时产生的字母为大写字母	o.setf (ios_base::uppercase)	B
nouppercase	o	复位上个标记	o.unsetf (ios_base::uppercase)	B
unitbuf	o	设置标记，每次格式化操作后刷新缓存	o.setf (ios_base::unitbuf)	B
nounitbuf	o	复位上个标记	o.unsetf (ios_base::unitbuf)	B
resetioflags (ios_base:: fmtflags mask)	io	根据mask复位标记	io.setf ((ios_base::fmtflags)0, mask)	M
setbase (int base)	io	设置整数表示的进制 (base=8,10,16)。	io.setf(base== 8 ? ios_base::oct : base==10 ? ios_base::dec : base==16 ? ios_base::hex : ios_base::fmtflags(0), ios_base::basefield)	M
setfill (charT c)	io	设置填充字符	io.fill(c)	M
setprecision (int n)	o	设置浮点数的精度	o.precision(n)	M
setw(int n)	io[17]	设置最小字段宽度	io.width(n)	M
internal	o	设置标记，在指定的内部位置填充字符	o.setf (ios_base::internal, ios_base::adjustfield)	B
left	o	设置标记，填充字符以确保左对齐	o.setf(ios_base::left, ios_base::adjustfield)	B
right	o	设置标记，填充字符以确保右对齐	o.setf(ios_base::right, ios_base::adjustfield)	B
dec	io	设置标记，使用十进制转换整数	io.setf(ios_base::dec, ios_base::basefield)	B
hex	io	设置标记，使用十六进制转换整数	io.setf(ios_base::hex, ios_base::basefield)	B
oct	io	设置标记，使用八进制转换整数	io.setf(ios_base::oct,	B

文件与内存操作

```
std::ofstream outFile("my_file");
outFile << "Hello";
std::ifstream inFile;
inFile.open("my_file");
std::cout << inFile.is_open() << std::endl;
inFile.close();
//文件在输出的时候会存到缓存中。关闭之后会
把缓存中的内容强制刷新
```

- 文件操作
  - basic_ifstream / basic_ofstream / basic_fstream
  - 文件流可以处于打开 / 关闭两种状态，处于打开状态时无法再次打开，只有打开时才能 I/O
- 文件流的打开模式（图引自 C++ IOStream 一书）
  - 每种文件流都有缺省的打开方式
  - 注意 ate 与 app 的异同
  - binary 能禁止系统特定的转换

    禁止系统\n\r和\n之间的转化
  - 避免意义不明确的流使用方式（如 ifstream + out ）

标记名	作用
in	打开以供读取
out	打开以供写入
ate	表示起始位置位于文件末尾
app	附加文件，即总是向文件尾写入
trunc	截断文件，即删除文件中的内容
binary	二进制模式

文件与内存操作

- 合法的打开方式组合（引自 C++ IOStream 一书）

打开方式	效果	加结尾模式标记 ate	加二进制模式标记
in ifstream	只读方式打开文本文件	初始文件位置位于文件末尾	禁止系统转换
out|trunc out ofstream	如果文件存在，将长度截断为 0； 否则建立文件供写入	初始文件位置位于文件末尾	禁止系统转换
out|app ofstream	附加；打开或建立文本文件， 仅供文件末尾写入	初始文件位置位于文件末尾	禁止系统转换
in|out fstream	打开文本文件供更新使用（支持读写）	初始文件位置位于文件末尾	禁止系统转换
in|out|trunc fstream	如果文件存在，将长度截断为 0； 否则建立文件供更新使用	初始文件位置位于文件末尾	禁止系统转换

文件与内存操作

```
std::ostringstream obj1;
obj1 << 1234; // 1234会被格式化成相应的字符串
std::string res = obj1.str();

std::istringstream obj2(res);
int x;
obj2 >> x;
```

- 内存流： basic_istringstream / basic_ostringstream / basic_stringstream
- 也会受打开模式： in / out / ate / app 的影响

```
std::ostringstream buf2("test", std::ios_base::ate);
buf2 << "1";
std::cout << buf2.str() << "\n"; // test1
```

- 使用 str() 方法获取底层所对应的字符串
  - 小心避免使用 str().c_str() 的形式获取 C 风格字符串

```
auto res = buf2.str();
auto c_res = res.c_str();
// 不要下列写法
auto res = buf2.str().c_str();
// str()返回的是一个临时对象，右值
// 当这条语句执行完成之后，就会被销毁。
// res就会指向一个被销毁的内存
```

- 基于字符串流的字符串拼接优化操作

```
std::string x;
x += "Hello";
x += " world";
// 这个方法不好，对string进行添加操作。string会在内部维护一段内存。在进行添加操作后，
会判断，我当前维护的内存是不是可以存放这些东西。如果不能，那么就开辟一块新的内存，存放新的内容。
然后再释放原来的内存。这样程序就在不停的开辟新的内存，和释放旧的内存。（内存的开辟和释放非常耗资源）
// 好的方法：字符串流
std::ostringstream ostr;
ostr << "hello";
ostr << " world";
std::string y = ostr.str(); // 流都会维持一个固定大小的缓冲区（不会太小）。缓冲区填满了，才会一次性
输出到终端。
```

流的状态

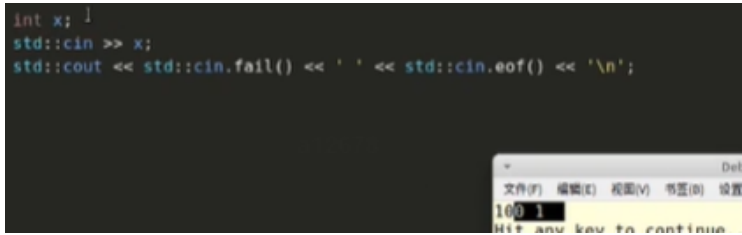
```
int x;
std::cin >> x;
std::cout << std::cin.good() << " "
          << std::cin.fail() << " "
          << std::cin.bad() << " "
          << std::cin.eof() << " "
          << static_cast<bool>(std::cin()) << "\n";
```

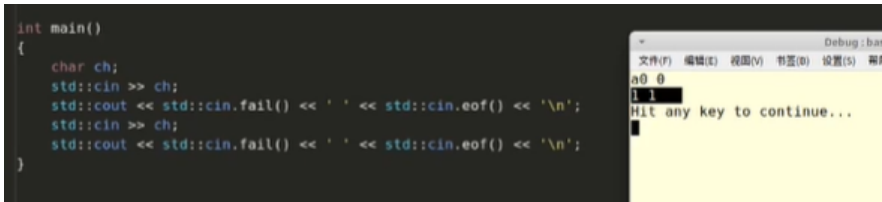
ios_base::iostate 标志			basic_ios 访问器					
eofbit	failbit	badbit	good()	fail()	bad()	eof()	**operator bool**	operator!
false	false	false	true	false	false	false	true	false
false	false	true	false	true	true	false	false	true
false	true	false	false	true	false	false	false	true
false	true	true	false	true	true	false	false	true
true	false	false	false	false	false	true	true	false
true	false	true	false	true	true	true	false	true
true	true	false	false	true	false	true	false	true
true	true	true	false	true	true	true	false	true

- iostate
  - failbit / badbit / eofbit / goodbit
- 检测流的状态
  - good( ) / fail() / bad() / eof() 方法
  - 转换为 bool 值（参考cppreference）
- 注意
  - 转换为 bool 值时不会考虑 eof
  - fail 与 eof 可能会被同时设置，但二者含意不同
- 通常来说，只要流处于某种错误状态时，插入 / 提取操作就不会生效

fail本质是可恢复性错误，例：输入输出操作过程中，插入和提取会造成错误，特别是提取。比如，我们希望得到一个整数值，开始输入确实字符串，这个字符串不能转化为整数值。

```
if(std::cin>>x) // 括号内的操作结束后会返回std::cin,cin会被隐式的转化为bool
```





流的状态（续）

- 设置流状态
  - clear： 设置流的状态为具体的值（缺省为 goodbit ）
  - setstate： 将某个状态附加到现有的流状态上
- 捕获流异常：exceptions方法F

流的定位

```
std::ostringstream s;
std::cout << s.tellp() << '\n'; // 0
s << 'h';
std::cout << s.tellp() << '\n'; // 1
```

- 获取流位置
  - g:get p:put
  - tellg() / tellp() 可以用于获取输入 / 输出流位置 (pos_type 类型)
  - 两个方法可能会失败，此时返回 pos_type(-1)
- 设置流位置
  - 例子见cppreference
  - seekg() / seekp() 用于设置输入 / 输出流的位置 会被覆盖，而不是插入
  - 这两个方法分别有两个重载版本：
    - 设置绝对位置：传入 pos_type 进行设置
    - 设置相对位置：通过偏移量（字符个数 ios_base::beg ） + 流位置符号的方式设置
      - ios_base::beg
      - ios_base::cur
      - ios_base::end

```
std::string str = "Hello, world";
std::istringstream in(str);
std::string word1, word2;
in >> word1; // Hello,
in.seekg(0);
in >> word2; // Hello,
```

流的同步

流都会维持一个自己的缓冲区，通常来说，缓冲区填满后才会一次性的做操作。
优点：最大性能的利用系统带宽，减少和终端的访问，提升系统的性能。

- 基于 flush() / sync() / unitbuf 的同步
  - flush() 用于输出流同步，刷新缓冲区 需要将数据显示到终端上，或者需要把它保存到文件里。
  - sync() 用于输入流同步，其实现逻辑是编译器所定义的 打开了一个文件想要读取。
  - 输出流可以通过设置 unitbuf 来保证每次输出后自动同步
- 基于绑定 (tie) 的同步
  - 流可以绑定到一个输出流上，这样在每次输入 / 输出前可以刷新输出流的缓冲区
  - 比如： cin 绑定到了 cout 上
- 与 C 语言标准 IO 库的同步
  - 缺省情况下， C++ 的输入输出操作会与 C 的输入输出函数同步
  - 可以通过 sync_with_stdio 关闭该同步

深蓝学院
shenlanxueyuan.com

感谢聆听！

Thanks for Listening