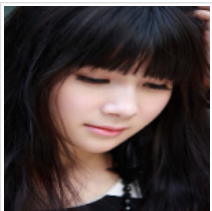


JoeBlackZQQ的专栏

目录视图 摘要视图 RSS 订阅

个人资料



JoeBlackzqq



访问： 771017次
积分： 11588
等级： BLOG 7
排名： 第505名

原创： 289篇
转载： 661篇
译文： 0篇
评论： 143条

文章搜索

文章分类

- ASP.NET (13)
- C (100)
- Java (11)
- Linux_C (139)
- Linux (163)
- Linux_Shell (43)
- Linux_C++ (86)
- VC (122)
- C# (26)
- Network (18)
- Major (59)
- Web Develop (10)
- Win_C++ (78)
- Mac (42)
- Game (11)
- JS (3)
- Python (89)
- Windows (77)
- Algorithm (12)
- RegularExp (9)
- Embedded (22)
- Media (30)

[程序员必须要学会算法吗](#) [博客专家庄晓立：我为什么要选择Rust?](#) [从零练就iOS高手实战班震撼来袭](#) [新型数据库利弊谈](#)

C++ stringstream介绍，使用方法与例子

分类： Win_C++ Linux_C++ 2011-12-01 22:57 24859人阅读 评论(5) 收藏 举报

[c++](#) [iostream](#) [string](#) [stream](#) [date](#)

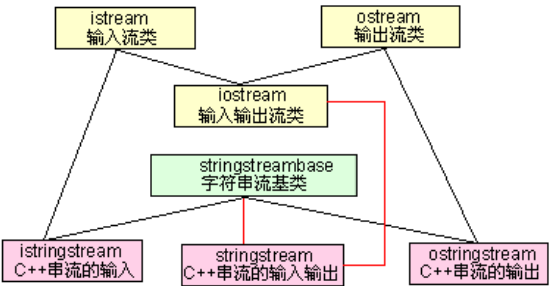
From: <http://www.usidcbbs.com/read-htm-tid-1898.html>

C++引入了ostreamstream、istreamstream、stringstream这三个类，要使用他们创建对象就必须包含sstream.h头文件。

istreamstream类用于执行C++风格的串流的输入操作。
ostreamstream类用于执行C风格的串流的输出操作。
strstream类同时可以支持C风格的串流的输入输出操作。

istreamstream类是从istream和stringstreambase派生而来，ostreamstream是从ostream和 stringstreambase派生而来， stringstream则是从iostream类和stringstreambase派生而来。

他们的继承关系如下图所示：



istreamstream是由一个string对象构造而来，istreamstream类从一个string对象读取字符。
istreamstream的构造函数原形如下：
istreamstream::istreamstream(string str);

```
[cpp]
01. #include <iostream>
02. #include <sstream>
03.
04. using namespace std;
05.
06. int main()
07. {
08.     istreamstream istr;
09.     istr.str("1 56.7");
10.     //上述两个过程可以简单写成 istreamstream istr("1 56.7");
11.     cout << istr.str() << endl;
```

Protocol (23)
OpenSource (21)
QT (10)
Graph_Image (6)
SoftEngineering (5)
PhoneDevelop (2)
Dos_Vbs (1)
Go (1)

文章存档
2015年06月 (9)
2015年05月 (18)
2015年04月 (10)
2015年03月 (9)
2015年02月 (21)
展开

阅读排行
C++ stringstream介绍, 1 (24853)
VS 2010 OpenGL 配置与 (10108)
DataGridView添加一行数据 (9323)
向DataGridView中添加新 (7566)
国内常用NTP服务器地址 (7189)
vsftpd默认用户名/密码 (6982)
将clover安装到硬盘EFI分 (6746)
[bash]删除文件中含特定: (6296)
vim匹配特定的行并删除 (5553)
python获取命令行输出结 (5149)

评论排行
VS 2010 OpenGL 配置与 (15)
【EWSA无线路由密码破 (7)
[bash]删除文件中含特定: (6)
基于海思开发板的屏幕截 (6)
vsftpd默认用户名/密码 (6)
C++ stringstream介绍, 1 (5)
自己实现的一个字符串编 (4)
DataGridView添加一行数据 (4)
在Windows下编译zlib1.2 (4)
MAC OS X10.9.5下成功! (4)

推荐文章
*Unity使用JsonFX插件进行序列化
*PullScrollView详解: PullScrollView实现
*NoSQL数据库概览及其与SQL语法的比较
*Git workflow
*以操作系统的角度述说线程与进程
*Android的Fragment中onActivityResult不被调用的解决方案

最新评论
sockaddr和sockaddr_in的区别 yibao0104: 感谢分享, 学习了
基于海思开发板的屏幕截图程序(habo2015: 详细
在mac上配置cocos2d-x开发环境

```
12.     int a;
13.     float b;
14.     istr >> a;
15.     cout << a << endl;
16.     istr >> b;
17.     cout << b << endl;
18.     return 0;
19. }
```

上例中, 构造字符串流的时候, 空格会成为字符串参数的内部分界, 例子中对a,b对象的输入"赋值"操作证明了这一点, 字符串的空格成为了整型数据与浮点型数据的分解点, 利用分界获取的方法我们事实上完成了字符串到整型对象与浮点型对象的拆分转换过程。

str()成员函数的使用可以让istringstream对象返回一个string字符串 (例如本例中的输出操作(cout<<istr.str();))。

ostringstream同样是由一个string对象构造而来, ostringstream类向一个string插入字符。ostringstream的构造函数原形如下:

```
ostringstream::ostringstream(string str);
```

示例代码如下:

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;
int main()
{
    ostringstream ostr;

    //ostr.str("abc");//如果构造的时候设置了字符串参数,那么增长操作的时候不会从结尾开始增加,而是修改原有数据,超出的部分增长
    ostr.put('d');
    ostr.put('e');
    ostr<<"fg";

    string gstr = ostr.str();
    cout<<gstr;
    system("pause");
}
```

在上例代码中, 我们通过put()或者左移操作符可以不断向ostr插入单个字符或者是字符串, 通过str()函数返回增长过后的完整字符串数据, 但值得注意的一点是, 当构造的时候对象内已经存在字符串数据的时候, 那么增长操作的时候不会从结尾开始增加,而是修改原有数据,超出的部分增长。

```
[ basic_stringbuf::str :
Sets or gets the text in a string buffer without changing the write position. ]
```

对于stringstream来说, 不用我多说, 大家也已经知道它是用于C++风格的字符串的输入输出的。stringstream的构造函数原形如下:

```
stringstream::stringstream(string str);
```

示例代码如下:

```
#include <iostream>
#include <sstream>
```

CoolGaming: 补充下，下了个cocos2d-x-2.2.6，发现里面根本没有install-templates-x...

【EWSA无线路由密码破解工具】baidu_28640363: 试试看

VC按最小化、关闭按钮、Esc都...李长春: 多谢分享，帮大忙了！！

VS2010删除所有断点时不弹出提...李长春: 明白了，原来还有这个。

按ESC关闭当前窗口李长春: 学习了。

COM、COM+和DCOM的定义和李长春: 征用这个。

COM、COM+和DCOM的定义和李长春: 没太懂。

xcode-select: error: tool 'xcodebu...花花猪: 解决了我碰到的问题，点赞，哈哈，我现在了6.3，把所有的Xcode都重新命名了一次，哈哈

Link Collection

DataGrid自定义分页

一个实现自定义event的文章

利用DataGrid编辑、修改、删除记录

一个可以下载自定义控件的网站

C#.NET FUN(有各种可供学习的分类)

C++面试题目

JoeBlackZQQ--163.blog

C++ stringstream介绍，使用方法与例子 - JoeBlackZQQ的专栏 - 博客频道 - CSDN.NET

```
#include <string>

using namespace std;

int main()
{
    stringstream ostr("ccc");
    ostr.put('d');
    ostr.put('e');
    ostr<<"fg";
    string gstr = ostr.str();
    cout<<gstr<<endl;

    char a;
    ostr>>a;
    cout<<a
```

```
system("pause");
}
```

除此而外，stringstream类的对象我们还常用它进行string与各种内置类型数据之间的转换。

示例代码如下：

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main()
{
    stringstream sstr;
    //-----int转string-----
    int a=100;
    string str;
    sstr<<a;
    sstr>>str;
    cout<<str<<endl;
    //-----string转char[]-----
    sstr.clear();//如果你想通过使用同一stringstream对象实现多种类型的转换，请注意在每一次转换之后都必须调用clear()成员函数。
    string name = "colinguan";
    char cname[200];
    sstr<<name;
    sstr>>cname;
    cout<<cname;
    system("pause");
}
```

使用stringstream对象简化类型转换

C++标准库中的<sstream>提供了比ANSI C的<stdio.h>更高级的一些功能，即单纯性、类型安全和可扩展性。在本节中，我们将使用这些库来实现安全和自动的类型转换。

为什么要学习

如果你已习惯了<stdio.h>风格的转换, 也许你首先会问: 为什么要花额外的精力来学习基于<sstream>的类型转换呢? 简单的例子的回顾能够说服你。假设你想用sprintf()函数将一个变量从int类型转换到字符串类型。为了正确地完成这个转换, 缓冲区必须有足够大空间以容纳转换完的字符串。此外, 还必须使用正确的格式化符。如果使用了不正确的格式化符, 后果。下面是一个例子:

```
int n=10000;

chars[10];

sprintf(s,"%d",n);// s中的内容为"10000"
```

到目前为止看起来还不错。但是, 对上面代码的一个微小的改变就会使程序崩溃:

```
int n=10000;

char s[10];

sprintf(s,"%f",n);// 看! 错误的格式化符
```

在这种情况下, 程序员错误地使用了%f格式化符来替代了%d。因此, s在调用完sprintf()后包含了一个不确定的字符。这导致程序崩溃。出正确的类型, 那不是更好吗?

进入stringstream

由于n和s的类型在编译期就确定了, 所以编译器拥有足够的信息来判断需要哪些转换。<sstream>库中声明的标准库函数会自动选择所必需的转换。而且, 转换结果保存在stringstream对象的内部缓冲中。你不必担心缓冲区溢出, 因为这些对象会自动管理存储空间。

你的编译器支持<sstream>吗?

<sstream>库是最近才被列入C++标准的。(不要把<sstream>与标准发布前被删掉的<strstream>弄混了。)因此, 如果你使用的是GCC 2.95, 并不支持它。如果你恰好正在使用这样的编译器而又想使用<sstream>的话, 就要先对它进行升级更新。

<sstream>库定义了三种类型: istream、ostream和stringstream, 分别用来进行流的输入、输出和输入输出。每个流都有一个对应的宽字符版本。简单起见, 我主要以stringstream为中心, 因为每个转换都要涉及到输入和输出操作。

注意, <sstream>使用string对象来代替字符数组。这样可以避免缓冲区溢出的危险。而且, 传入参数和目标对象的类型必须匹配。如果类型不匹配, 即使使用了不正确的格式化符也没有危险。

string到int的转换

```
string result="10000";
int n=0;
stringstream ss(result);
n=ss.get();//n等于10000
```

重复利用stringstream对象

如果你打算在多次转换中使用同一个stringstream对象, 记住再每次转换前要使用clear()方法;

在多次转换中重复使用同一个stringstream (而不是每次都创建一个新的对象) 对象最大的好处在于效率。stringstream对象的clear()函数通常是非常耗费CPU时间的。

在类型转换中使用模板

你可以轻松地定义函数模板来将一个任意的类型转换到特定的目标类型。例如, 需要将各种数字值, 如int、long、double等, 转换为字符串, 要使用以一个string类型和一个任意值t为参数的to_string()函数。to_string()函数将t转换为字符串并写入result中。以下代码获取流内部缓冲的一份拷贝:

```
template<class T>
```

```
void to_string(string & result,const T& t)
```

```
{
```

```
    stringstream oss;//创建一个流
```

```
    oss<<t;//把值传递如流中
```

```
    result=oss.str();//获取转换后的字符串并将其写入result
```

```
}
```

这样，你就可以轻松地将多种数值转换成字符串了：

```
to_string(s1,10.5);//double到string
```

```
to_string(s2,123);//int到string
```

```
to_string(s3,true);//bool到string
```

可以更进一步定义一个通用的转换模板，用于任意类型之间的转换。函数模板convert()含有两个模板参数out_type和in_value值转换成out_type类型：

```
template<class out_type,class in_value>
```

```
out_type convert(const in_value & t)
```

```
{
```

```
    stringstream stream;
```

```
    stream<<t;//向流中传值
```

```
    out_type result;//这里存储转换结果
```

```
    stream>>result;//向result中写入值
```

```
    return result;
```

```
}
```

这样使用convert()：

```
double d;
```

```
string salary;
```

```
string s="12.56";
```

```
d=convert<double>(s);//d等于12.56
```

```
salary=convert<string>(9000.0);//salary等于"9000"
```

结论

在过去留下来的程序代码和纯粹的C程序中，传统的<stdio.h>形式的转换伴随了我们很长的一段时间。但是，如文中am的转换拥有类型安全和不会溢出这样抢眼的特性，使我们有充足得理由抛弃<stdio.h>而使用<sstream>。<sstream>的一个特性——可扩展性。你可以通过重载来支持自定义类型间的转换。

一些实例：

stringstream通常是用来做数据转换的。

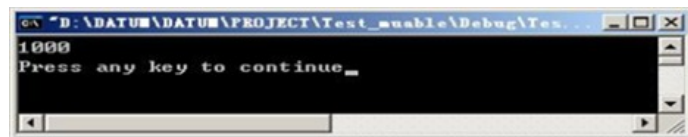
相比c库的转换, 它更加安全, 自动和直接。

例子一: 基本数据类型转换例子 int转string

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

int main()
{
    std::stringstream stream;
    std::string result;
    int i = 1000;
    stream << i; //将int输入流
    stream >> result; //从stream中抽取前面插入的int值
    std::cout << result << std::endl; // print the string "1000"
}
```

运行结果:



例子二: 除了基本类型的转换, 也支持char *的转换。

```
#include <sstream>
#include <iostream>

int main()
{
    std::stringstream stream;
    char result[8];
    stream << 8888; //向stream中插入8888
    stream >> result; //抽取stream中的值到result
    std::cout << result << std::endl; // 屏幕显示 "8888"
}
```



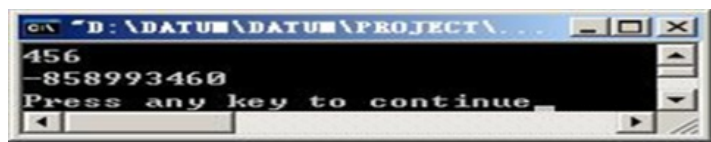
例子三：再进行多次转换的时候，必须调用stringstream的成员函数clear()。

```
#include <sstream>
#include <iostream>
int main()
{
    std::stringstream stream;
    int first, second;
    stream<< "456"; //插入字符串
    stream >> first; //转换成int
    std::cout << first << std::endl;
    stream.clear(); //在进行多次转换前，必须清除stream
    stream << true; //插入bool值
    stream >> second; //提取出int
    std::cout << second << std::endl;
}
```

运行clear的结果



没有运行clear的结果



iostream 的用途与局限

本文主要考虑 x86 Linux 平台，不考虑跨平台的可移植性，也不考虑国际化(i18n)，但是要考虑 32-bit 和 64-bit 的兼容性。本文以 stdio 指代 C 语言的 scanf/printf 系列格式化输入输出函数。本文注意区分“编程初学者”和“C++初学者”，二者含义不同。

摘要：C++ iostream 的主要作用是让初学者有一个方便的命令行输入输出试验环境，在真实的项目中很少用到 iostream，因此不必把精力花在深究 iostream 的格式化与 manipulator。iostream 的设计初衷是提供一个可扩展的类型安全的 IO 机制，但是后来莫名其妙地加入了 locale 和 facet 等累赘。其整个设计复杂不堪，多重+虚拟继承的结构也很巴洛克，性能方面几无亮点。iostream 在实际项目中的用处非常有限，为此投入过多学习精力实在不值。

stdio 格式化输入输出的缺点

1. 对编程初学者不友好

看看下面这段简单的输入输出代码。

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

```
int main()
{
    int i;
    short s;
```

```
float f;
double d;
char name[80];

scanf("%d %hd %f %lf %s", &i, &s, &f, &d, name);
printf("%d %d %f %f %s", i, s, f, d, name);
}
```

注意到其中

输入和输出用的格式字符串不一样。输入 `short` 要用 `%hd`，输出用 `%d`；输入 `double` 要用 `%lf`，输出用 `%f`。

输入的参数不统一。对于 `i`、`s`、`f`、`d` 等变量，在传入 `scanf()` 的时候要取地址(&)，而对于 `name`，则不用取地址。

读者可以试一试如何用几句话向刚开始学编程的初学者解释上面两条背后原因（涉及到传递函数不定参数时的类型转换，函数调用栈的内存布局，指针的意义，字符数组退化为字符指针等等），如果一开始解释不清，只好告诉学生“这是规定”。

缓冲区溢出的危险。上面的例子在读入 `name` 的时候没有指定大小，这是用 C 语言编程的安全漏洞的主要来源。应该在一开始就强调正确的做法，避免养成错误的习惯。正确而安全的做法如 Bjarne Stroustrup 在《Learning Standard C++ as a New Language》所示：

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

```
int main()
{
    const int max = 80;
    char name[max];

    char fmt[10];
    sprintf(fmt, "%%%ds", max - 1);
    scanf(fmt, name);
    printf("%s\n", name);
}
```

这个动态构造格式化字符串的做法恐怕更难向初学者解释。

2. 安全性(security)

C 语言的安全性问题近十几年来引起了广泛的注意，C99 增加了 `snprintf()` 等能够指定输出缓冲区大小的函数，输出方面的安全性问题已经得到解决；输入方面似乎没有太大进展，还要靠程序员自己动手。

考虑一个简单的编程任务：从文件或标准输入读入一行字符串，行的长度不确定。我发现没有哪个 C 语言标准库函数能完成这个任务，除非 **roll your own**。

首先，`gets()` 是错误的，因为不能指定缓冲区的长度。

其次，`fgets()` 也有问题。它能指定缓冲区的长度，所以是安全的。但是程序必须预设一个长度的最大值，这不满足题目要求“行的长度不确定”。另外，程序无法判断 `fgets()` 到底读了多少个字节。为什么？考虑一个文件的内容是 9 个字节的字符串 "Chen\000Shuo"，注意中间出现了 '\0' 字符，如果用 `fgets()` 来读取，客户端如何知道 "\000Shuo" 也是输入的一部分？毕竟 `strlen()` 只返回 4，而且整个字符串里没有 '\n' 字符。

最后，可以用 `glibc` 定义的 `getline(3)` 函数来读取不定长的“行”。这个函数能正确处理各种情况，不过它返回的是 `malloc()` 分配的内存，要求调用端自己 `free()`。

3. 类型安全(type-safe)

如果 `printf()` 的整数参数类型是 `int`、`long` 等标准类型，那么 `printf()` 的格式化字符串很容易

写。但是如果参数类型是 `typedef` 的类型呢？

如果你想在程序中用 `printf` 来打印日志，你能一眼看出下面这些类型该用 `"%d"` `"%ld"` `"%lld"` 中的哪一个来输出？你的选择是否同时兼容 32-bit 和 64-bit 平台？

`clock_t`。这是 `clock(3)` 的返回类型

`dev_t`。这是 `mknod(3)` 的参数类型

`in_addr_t`、`in_port_t`。这是 `struct sockaddr_in` 的成员类型

`nfds_t`。这是 `poll(2)` 的参数类型

`off_t`。这是 `lseek(2)` 的参数类型，麻烦的是，这个类型与宏定义 `_FILE_OFFSET_BITS` 有关。

`pid_t`、`uid_t`、`gid_t`。这是 `getpid(2)` `getuid(2)` `getgid(2)` 的返回类型

`ptrdiff_t`。`printf()` 专门定义了 `"t"` 前缀来支持这一类型（即使用 `"%td"` 来打印）。

`size_t`、`ssize_t`。这两个类型到处都在用。`printf()` 为此专门定义了 `"z"` 前缀来支持这两个类型（即使用 `"%zu"` 或 `"%zd"` 来打印）。

`socklen_t`。这是 `bind(2)` 和 `connect(2)` 的参数类型

`time_t`。这是 `time(2)` 的返回类型，也是 `gettimeofday(2)` 和 `clock_gettime(2)` 的输出结构体的成员类型

如果在 C 程序里要正确打印以上类型的整数，恐怕要费一番脑筋。《The Linux Programming Interface》的作者建议（3.6.2节）先统一转换为 `long` 类型再用 `"%ld"` 来打印；对于某些类型仍然需要特殊处理，比如 `off_t` 的类型可能是 `long long`。

还有，`int64_t` 在 32-bit 和 64-bit 平台上是不同的类型，为此，如果程序要打印 `int64_t` 变量，需要包含 `<inttypes.h>` 头文件，并且使用 `PRId64` 宏：

```
#include <stdio.h>
#define __STDC_FORMAT_MACROS
#include <inttypes.h>
```

```
int main()
{
    int64_t x = 100;
    printf("%" PRId64 "\n", x);
    printf("%06" PRId64 "\n", x);
}
```

muduo 的 `Timestamp` 使用了 `PRId64` <http://code.google.com/p/muduo/source/browse/trunk/muduo/base/Timestamp.cc#25>

Google C++ 编码规范也提到了 64-bit 兼容性：http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#64-bit_Portability

这些问题在 C++ 里都不存在，在这方面 `iostream` 是个进步。

C `stdio` 在类型安全方面原本还有一个缺点，即格式化字符串与参数类型不匹配会造成难以发现的 bug，不过现在的编译器已经能够检测很多这种错误：

```
int main()
{
    double d = 100.0;
    // warning: format '%d' expects type 'int', but argument 2 has type 'double'
    printf("%d\n", d);

    short s;
    // warning: format '%d' expects type 'int*', but argument 2 has type 'short int*'
    scanf("%d", &s);

    size_t sz = 1;
    // no warning
    printf("%zd\n", sz);
```

```
}
```

4. 不可扩展?

C stdio 的另外一个缺点是无法支持自定义的类型, 比如我写了一个 Date class, 我无法像打印 int 那样用 printf 来直接打印 Date 对象。

```
struct Date
{
    int year, month, day;
};
```

```
Date date;
printf("%D", &date); // WRONG
```

Glibc 放宽了这个限制, 允许用户调用 register_printf_function(3) 注册自己的类型, 当然, 前提是与现有的格式字符不冲突 (这其实大大限制了这个功能的用处, 现实中也几乎没有人真的去用它)。 <http://www.gnu.org/s/hello/manual/libc/Printf-Extension-Example.html> http://en.wikipedia.org/wiki/Printf#Custom_format_placeholders

5. 性能

C stdio 的性能方面有两个弱点。

使用一种 little language (现在流行叫 DSL) 来配置格式。固然有利于紧凑性和灵活性, 但损失了一点点效率。每次打印一个整数都要先解析 "%d" 字符串, 大多数情况下不是问题, 某些场合需要自己写整数到字符串的转换。

C locale 的负担。locale 指的是不同语种对“什么是空白”、“什么是字母”, “什么是小数点”有不同的定义 (德语里边小数点是逗号, 不是句点)。C 语言的 printf()、scanf()、isspace()、isalpha()、ispunct()、strtod() 等等函数都和 locale 有关, 而且可以在运行时动态更改。就算是程序只使用默认的 "C" locale, 任然要为这个灵活性付出代价。

iostream 的设计初衷

iostream 的设计初衷包括克服 C stdio 的缺点, 提供一个高效的可扩展的类型安全的 IO 机制。“可扩展”有两层意思, 一是可以扩展到用户自定义类型, 而是通过继承 iostream 来定义自己的 stream, 本文把前一种称为“类型可扩展”后一种称为“功能可扩展”。

“类型可扩展”和“类型安全”都是通过函数重载来实现的。

iostream 对初学者很友好, 用 iostream 重写与前面同样功能的代码:

```
#include <iostream>
#include <string>
using namespace std;
```

```
int main()
{
    int i;
    short s;
    float f;
    double d;
    string name;
```

```
    cin >> i >> s >> f >> d >> name;
    cout << i << " " << s << " " << f << " " << d << " " << name << endl;
}
```

这段代码恐怕比 scanf/printf 版本容易解释得多, 而且没有安全性(security)方面的问题。我们自己的类型也可以融入 iostream, 使用起来与 built-in 类型没有区别。这主要得力于

C++ 可以定义 non-member functions/operators。

```
#include <ostream> // 是不是太重量级了?
```

```
class Date
{
public:
    Date(int year, int month, int day)
        : year_(year), month_(month), day_(day)
    {
    }

    void writeTo(std::ostream& os) const
    {
        os << year_ << '-' << month_ << '-' << day_;
    }

private:
    int year_, month_, day_;
};

std::ostream& operator<<(std::ostream& os, const Date& date)
{
    date.writeTo(os);
    return os;
}

int main()
{
    Date date(2011, 4, 3);
    std::cout << date << std::endl;
    // 输出 2011-4-3
}
```

iostream 凭借这两点（类型安全和类型可扩展），基本克服了 stdio 在使用上的不便与不安全。如果 iostream 止步于此，那它将是一个非常便利的库，可惜它前进了另外一步。

iostream 与标准库其他组件的交互

不同于标准库其他 class 的“值语意”，iostream 是“对象语意”，即 iostream 是 non-copyable。这是正确的，因为如果 fstream 代表一个文件的话，拷贝一个 fstream 对象意味着什么呢？表示打开了两个文件吗？如果销毁一个 fstream 对象，它会关闭文件句柄，那么另一个 fstream copy 对象会因此受影响吗？

C++ 同时支持“数据抽象”和“面向对象编程”，其实主要就是“值语意”与“对象语意”的区别，我发现不是每个人都清楚这一点，这里多说几句。标准库里的 complex<>、pair<>、vector<>、string 等等都是值语意，拷贝之后就与原对象脱离关系，就跟拷贝一个 int 一样。而我们自己写的 Employee class、TcpConnection class 通常是对象语意，拷贝一个 Employee 对象是没有意义的，一个雇员不会变成两个雇员，他也不会领两份薪水。拷贝 TcpConnection 对象也没有意义，系统里边只有一个 TCP 连接，拷贝 TcpConnection 对象不会让我们拥有两个连接。因此如果在 C++ 里做面向对象编程，写的 class 通常应该禁用 copy constructor 和 assignment operator，比如可以继承 boost::noncopyable。对象语意的类型不能直接作为标准容器库的成员。另一方面，如果要写一个图形程序，其中用到三维空间的向量，那么我们可以写 Vector3D class，它应该是值语意的，允许拷贝，并且可以用作标准容器库的成员，例如

`vector<Vector3D>` 表示一条三维的折线。

C stdio 的另外一个缺点是 `FILE*` 可以随意拷贝, 但是只要关闭其中一个 `copy`, 其他 `copies` 也都失效了, 跟空悬指针一般。这其实不光是 C stdio 的缺点, 整个 C 语言对待资源 (malloc 得到的内存, `open()` 打开的文件, `socket()` 打开的连接) 都是这样, 用整数或指针来代表 (即“句柄”)。而整数和指针类型的“句柄”是可以随意拷贝的, 很容易就造成重复释放、遗漏释放、使用已经释放的资源等等常见错误。这是因为 C 语言错误地让“对象语言”的东西变成了值语义。

`iostream` 禁止拷贝, 利用对象的生命期来明确管理资源 (如文件), 很自然地就避免了 C 语言易犯的错误。这就是 `RAII`, 一种重要且独特的 C++ 编程手法。

`std::string`

`iostream` 可以与 `string` 配合得很好。但是有一个问题: 谁依赖谁?

`std::string` 的 `operator <<` 和 `operator >>` 是如何声明的? “string” 头文件在声明这两个 `operators` 的时候要不要 `include "iostream"`?

`iostream` 和 `string` 都可以单独 `include` 来使用, 显然 `iostream` 头文件里不会定义 `string` 的 `<<` 和 `>>` 操作。但是, 如果“string”要 `include "iostream"`, 岂不是让 `string` 的用户被迫也用了 `iostream`? 编译 `iostream` 头文件可是相当的慢啊 (因为 `iostream` 是 `template`, 其实现代码都放到了头文件中)。

标准库的解决办法是定义 `iosfwd` 头文件, 其中包含 `istream` 和 `ostream` 等的前向声明 (forward declarations), 这样 “string” 头文件在定义输入输出操作符时就可以不必包含 “`iostream`”, 只需要包含简短得多的 “`iosfwd`”。我们自己写程序也可借此学习如何支持可选的功能。值得注意的是, `istream::getline()` 成员函数的参数类型是 `char*`, 因为 “`istream`” 没有包含 “`string`”, 而我们常用的 `std::getline()` 函数是个 `non-member function`, 定义在 “`string`” 里边。

`std::complex`

标准库的复数类 `complex` 的情况比较复杂。使用 `complex` 会自动包含 `sstream`, 后者会包含 `istream` 和 `ostream`, 这是个不小的负担。问题是, 为什么?

它的 `operator >>` 操作比 `string` 复杂得多, 如何应对格式不正确的情况? 输入字符串不会遇到格式不正确, 但是输入一个复数可能遇到各种问题, 比如数字的格式不对等。我怀疑有谁会真的在产品项目里用 `operator >>` 来读入字符方式表示的复数, 这样的代码的健壮性如何保证。基于同样的理由, 我认为产品代码中应该避免用 `istream` 来读取带格式的内容, 后面也不再谈 `istream` 的缺点, 它已经被秒杀。

它的 `operator <<` 也很奇怪, 它不是直接使用参数 `ostream& os` 对象来输出, 而是先构造 `ostringstream`, 输出到该 `string stream`, 再把结果字符串输出到 `ostream`。简化后的代码如下:

```
template<typename T>
std::ostream& operator<<(std::ostream& os, const std::complex<T>& x)
{
    std::ostringstream s;
    s << '(' << x.real() << ',' << x.imag() << ')';
    return os << s.str();
}
```

注意到 `ostringstream` 动态分配内存, 也就是说, 每输出一个 `complex` 对象就会分配释放一次内存, 效率

根据以上分析, 我认为 `string` 和 `complex` 配合得不好, 但是它们耦合得更紧密 (与 `string/iostream` 相比) 不得已的技术限制吧 (`complex` 是 `template`, 其 `operator <<` 必须在头文件中定义, 而这个定义又用到了 `ostringstream`, 不得已包含了 `iostream` 的实现)。

如果程序要对 `complex` 做大量输出, 从效率和健壮性方面考虑, 建议不要使用 `iostream`。

`iostream` 在使用方面的缺点

在简单使用 `iostream` 的时候, 它确实比 `stdio` 方便, 但是深入一点就会发现, 二者可说各擅胜

场。下面谈一谈 `iostream` 在使用方面的缺点。

1. 格式化输出很繁琐

`iostream` 采用 `manipulator` 来格式化, 如果我想按照 2010-04-03 的格式输出前面定义的 `Date` class, 那么代码要改成:

```
--- 02-02.cc  2011-07-16 16:40:05.000000000 +0800
+++ 04-01.cc  2011-07-16 17:10:27.000000000 +0800
@@ -1,4 +1,5 @@
#include <iostream>
#include <iomanip>
```

```
class Date
{
@@ -10,7 +11,9 @@
```

```
    void writeTo(std::ostream& os) const
    {
-   os << year_ << '-' << month_ << '-' << day_;
+   os << year_ << '-'
+   << std::setw(2) << std::setfill('0') << month_ << '-'
+   << std::setw(2) << std::setfill('0') << day_;
    }
```

private:

假如用 `stdio`, 会简短得多, 因为 `printf` 采用了一种表达能力较强的小语言来描述输出格式。

```
--- 04-01.cc  2011-07-16 17:03:22.000000000 +0800
+++ 04-02.cc  2011-07-16 17:04:21.000000000 +0800
@@ -1,5 +1,5 @@
#include <iostream>
#include <iomanip>
#include <stdio.h>
```

```
class Date
{
@@ -11,9 +11,9 @@
```

```
    void writeTo(std::ostream& os) const
    {
-   os << year_ << '-' << month_ << '-' << day_;
+   char buf[32];
+   snprintf(buf, sizeof buf,
+   "%d-%02d-%02d"
+   , year_, month_, day_);
+   os << buf;
    }
```

private:

使用小语言来描述格式还带来另外一个好处：外部可配置。

2. 外部可配置性

比方说，我想用一个外部的配置文件来定义日期的格式。C `stdio` 很好办，把格式字符串 `"%d-%02d-%02d"` 保存到配置里就行。但是 `iostream` 呢？它的格式是写死在代码里的，灵活性大打折扣。

再举一个例子，程序的 `message` 的多语言化。

```
const char* name = "Shuo Chen";
int age = 29;
printf("My name is %1$s, I am %2$d years old.\n", name, age);
cout << "My name is " << name << ", I am " << age << " years old." << endl;
```

对于 `stdio`，要让这段程序支持中文的话，把代码中的 `"My name is %1$s, I am %2$d years old.\n"`,

替换为 `"我叫%1$s, 今年%2$d岁.\n"` 即可。也可以把这段提示语做成资源文件，在运行时读入。而对于 `iostream`，恐怕没有这么方便，因为代码是支离破碎的。

C `stdio` 的格式化字符串体现了重要的“数据就是代码”的思想，这种“数据”与“代码”之间的相互转换是程序灵活性的根源，远比 OO 更为灵活。

3. `stream` 的状态

如果我想用 16 进制方式输出一个整数 `x`，那么可以用 `hex` 操控符，但是这会改变 `ostream` 的状态。比如说

```
int x = 8888;
cout << hex << showbase <<
x
<< endl; // forgot to reset state
```

```
cout << 123 << endl;
```

这段代码会把 123 也按照 16 进制方式输出，这恐怕不是我们想要的。

再举一个例子，`setprecision()` 也会造成持续影响：

```
double d = 123.45;
printf("%.3f\n", d);
cout << d << endl;
cout << setw(8) << fixed << setprecision(3) << d << endl;
cout << d << endl;
```

输出是：

```
$ ./a.out
123.450
123.45 # default cout format
123.450 # our format
123.450 # side effects
```

可见代码中的 `setprecision()` 影响了后续输出的精度。注意 `setw()` 不会造成影响，它只对下一个输出有效。

这说明，如果使用 `manipulator` 来控制格式，需要时刻小心防止影响了后续代码。而使用 C `stdio` 就没有这个问题，它是“上下文无关的”。

4. 知识的通用性

在 C 语言之外，有其他很多语言也支持 `printf()` 风格的格式化，例如 Java、Perl、Ruby 等等 (http://en.wikipedia.org/wiki/Printf#Programming_languages_with_printf)。学会 `printf()` 的格式化方法，这个知识还可以用到其他语言中。但是 C++ `iostream` 只此一家别无分店，反正都是格式化输出，`stdio` 的投资回报率更高。

基于这点考虑，我认为不必深究 `iostream` 的格式化方法，只需要用好它最基本的类型安全输出即可。在真的需要格式化的场合，可以考虑 `snprintf()` 打印到栈上缓冲，再用 `ostream` 输出。

5. 线程安全与原子性

`iostream` 的另外一个问题是线程安全性。`stdio` 的函数是线程安全的, 而且 C 语言还提供了 `flockfile(3)/funlockfile(3)` 之类的函数来明确控制 `FILE*` 的加锁与解锁。

`iostream` 在线程安全方面没有保证, 就算单个 `operator<<` 是线程安全的, 也不能保证原子性。因为 `cout << a << b;` 是两次函数调用, 相当于 `cout.operator<<(a).operator<<(b)`。两次调用中间可能会被打断进行上下文切换, 造成输出内容不连续, 插入了其他线程打印的字符。

而 `fprintf(stdout, "%s %d", a, b);` 是一次函数调用, 而且是线程安全的, 打印的内容不会受其他线程影响。

因此, `iostream` 并不适合在多线程程序中做 logging。

`iostream` 的局限

根据以上分析, 我们可以归纳 `iostream` 的局限:

输入方面, `istream` 不适合输入带格式的数据, 因为“纠错”能力不强, 进一步的分析请见孟岩写的《契约思想的一个反面案例》, 孟岩说“复杂的设计必然带来复杂的使用规则, 而面对复杂的使用规则, 用户是可以投票的, 那就是你做你的, 我不用!”可谓鞭辟入里。如果要用 `istream`, 我推荐的做法是用 `getline()` 读入一行数据, 然后用正则表达式来判断内容正误, 并做分组, 然后用 `strtod/strtoul` 之类的函数做类型转换。这样似乎更容易写出健壮的程序。

输出方面, `ostream` 的格式化输出非常繁琐, 而且写死在代码里, 不如 `stdio` 的小语言那么灵活通用。建议只用作简单的无格式输出。

log 方面, 由于 `ostream` 没有办法在多线程程序中保证一行输出的完整性, 建议不要直接用它来写 log。如果是简单的单线程程序, 输出数据量较少的情况下可以酌情使用。当然, 产品代码应该用成熟的 logging 库, 而不要用其它东西来凑合。

in-memory 格式化方面, 由于 `ostringstream` 会动态分配内存, 它不适合性能要求较高的场合。

文件 IO 方面, 如果用作文本文件的输入或输出, `(i|o)fstream` 有上述的缺点; 如果用作二进制数据输入输出, 那么自己简单封装一个 `File class` 似乎更好用, 也不必为用不到的功能付出代价(后文还有具体例子)。`ifstream` 的一个用处是在程序启动时读入简单的文本配置文件。如果配置文件是其他文本格式(XML 或 JSON), 那么用相应的库来读, 也用不到 `ifstream`。

性能方面, `iostream` 没有兑现“高效性”诺言。`iostream` 在某些场合比 `stdio` 快, 在某些场合比 `stdio` 慢, 对于性能要求较高的场合, 我们应该自己实现字符串转换(见后文的代码与测试)。`iostream` 性能方面的一个注脚: 在线 ACM/ICPC 判题网站上, 如果一个简单的题目发生超时错误, 那么把其中 `iostream` 的输入输出换成 `stdio`, 有时就能过关。

既然有这么多局限, `iostream` 在实际项目中的应用就大为受限了, 在这上面投入太多的精力实在不值得。说实话, 我没有见过哪个 C++ 产品代码使用 `iostream` 来作为输入输出设施。<http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml#Streams>

`iostream` 在设计方面的缺点

`iostream` 的设计有相当多的 WTFs, stackoverflow 有人吐槽说“If you had to judge by today's software engineering standards, would C++'s IOStreams still be considered well-designed?”<http://stackoverflow.com/questions/2753060/who-architected-designed-cs-iostreams-and-would-it-still-be-considered-well>。

面向对象的设计

`iostream` 是个面向对象的 IO 类库, 本节简单介绍它的继承体系。

对 `iostream` 略有了解的人 would 知道它用了多重继承和虚拟继承, 简单地画个类图如下, 是典型的菱形继承:

如果加深一点了解, 会发现 `iostream` 现在是模板化的, 同时支持窄字符和宽字符。下图是现在的继承体系, 同时画出了 `fstreams` 和 `stringstreams`。图中方框的第二行是模板的具现化类

型，也就是我们代码里常用的具体类型（通过 `typedef` 定义）。

这个继承体系糅合了面向对象与泛型编程，但可惜它两方面都不讨好。

再进一步加深了解，发现还有一个平行的 `streambuf` 继承体系，`fstream` 和 `stringstream` 的不同之处主要就在于它们使用了不同的 `streambuf` 具体类型。

再把这两个继承体系画到一幅图里：

注意到 `basic_ios` 持有了 `streambuf` 的指针；而 `fstreams` 和 `stringstreams` 则分别包含 `filebuf` 和 `stringbuf` 的对象。看上去有点像 Bridge 模式。

看了这样巴洛克的设计，有没有人还打算在自己的项目中想通过继承 `iostream` 来实现自己的 `stream`，以实现功能扩展么？

面向对象方面的设计缺陷

本节我们分析一下 `iostream` 的设计违反了哪些 OO 准则。

我们知道，面向对象中的 `public` 继承需要满足 Liskov 替换原则。（见《Effective C++ 第3版》条款32：确保你的 `public` 继承模塑出 is-a 关系。《C++ 编程规范》条款 37： `public` 继承意味可替换性。继承非为复用，乃为被复用。）

在程序里需要用到 `ostream` 的地方（例如 `operator<<`），我传入 `ofstream` 或 `ostream` 都应该能按预期工作，这就是 OO 继承强调的“可替换性”，派生类的对象可以替换基类对象，从而被 `operator<<` 复用。

`iostream` 的继承体系多次违反了 Liskov 原则，这些地方继承的目的是为了复用基类的代码，下图中我把违规的继承关系用红线标出。

在现有的继承体系中，合理的有：

`ifstream` is-a `istream`

`istringstream` is-a `istream`

`ofstream` is-a `ostream`

`ostringstream` is-a `ostream`

`fstream` is-a `iostream`

`stringstream` is-a `iostream`

我认为不怎么合理的有：

`ios` 继承 `ios_base`，有没有哪种情况下程序代码期待 `ios_base` 对象，但是客户可以传入一个 `ios` 对象替代之？如果没有，这里用 `public` 继承是不是违反 OO 原则？

`istream` 继承 `ios`，有没有哪种情况下程序代码期待 `ios` 对象，但是客户可以传入一个 `istream` 对象替代之？如果没有，这里用 `public` 继承是不是违反 OO 原则？

`ostream` 继承 `ios`，有没有哪种情况下程序代码期待 `ios` 对象，但是客户可以传入一个 `ostream` 对象替代之？如果没有，这里用 `public` 继承是不是违反 OO 原则？

`iostream` 多重继承 `istream` 和 `ostream`。为什么 `iostream` 要同时继承两个 non-interface class？这是接口继承还是实现继承？是不是可以用组合(composition)来替代？（见《Effective C++ 第3版》条款38：通过组合模塑出 has-a 或“以某物实现”。《C++ 编程规范》条款 34：尽可能以组合代替继承。）

用组合替换继承之后的体系：

注意到在新的设计中，只有真正的 is-a 关系采用了 `public` 继承，其他均以组合来代替，组合关系以红线表示。新的设计没有用的虚拟继承或多重继承。

其中 `iostream` 的新实现值得一提，代码结构如下：

```
class istream;
```



```
class ostream;
```

```
class istream
{
public:
    istream& get_istream();
    ostream& get_ostream();
    virtual ~istream();
};
```

这样一来, 在需要 `istream` 对象表现得像 `istream` 的地方, 调用 `get_istream()` 函数返回一个 `istream` 的引用; 在需要 `istream` 对象表现得像 `ostream` 的地方, 调用 `get_ostream()` 函数返回一个 `ostream` 的引用。功能不受影响, 而且代码更清晰。(虽然我非常怀疑 `istream` 的真正价值, 一个东西既可读又可写, 说明是个 `sophisticated IO` 对象, 为什么还用这么厚的 OO 封装?)

阳春的 `locale`

`istream` 的故事还不止这些, 它还包含一套阳春的 `locale/facet` 实现, 这套实践中没人用的东西进一步增加了 `istream` 的复杂度, 而且不可避免地影响其性能。Nathan Myers 正是始作俑者<http://www.cantrip.org/locale.html>。

`ostream` 自身定义的针对整数和浮点数的 `operator<<` 成员函数的函数体是:

```
bool failed =
    use_facet<num_put>(getloc()).put(
        ostreambuf_iterator(*this), *this, fill(), val).failed();
```

它会转而调用 `num_put::put()`, 后者会调用 `num_put::do_put()`, 而 `do_put()` 是个虚函数, 没办法 `inline`。`istream` 在性能方面的不足恐怕部分来自于此。这个虚函数白白浪费了把 `template` 的实现放到头文件应得的好处, 编译和运行速度都快不起来。

我没有深入挖掘其中的细节, 感兴趣的同学可以移步观看 `facet` 的继承体系: <http://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a00431.html>

据此分析, 我不认为以 `istream` 为基础的上层程序库 (比方说那些克服 `istream` 格式化方面的缺点的库) 有多大的实用价值。

臆造抽象

孟岩评价 “`istream` 最大的缺点是臆造抽象”, 我非常赞同他老人家的观点。

这个评价同样适用于 Java 那一套叠床架屋的 `InputStream/OutputStream/Reader/Writer` 继承体系, .NET 也搞了这么一套繁文缛节。

乍看之下, 用 `input stream` 表示一个可以“读”的数据流, 用 `output stream` 表示一个可以“写”的数据流, 屏蔽底层细节, 面向接口编程, “符合面向对象原则”, 似乎是一件美妙的事情。但是, 真实的世界要残酷得多。

IO 是个极度复杂的东西, 就拿最常见的 `memory stream`、`file stream`、`socket stream` 来说, 它们之间的差异极大:

是单向 IO 还是双向 IO。只读或者只写? 还是既可读又可写?

顺序访问还是随机访问。可不可以 `seek`? 可不可以退回 `n` 字节?

文本数据还是二进制数据。格式有误怎么办? 如何编写健壮的处理输入的代码?

有无缓冲。`write 500` 字节是否能保证完全写入? 有没有可能只写入了 `300` 字节? 余下 `200` 字节怎么办?

是否阻塞。会不会返回 `EWouldBlock` 错误?

有哪些出错的情况。这是最难的, `memory stream` 几乎不可能出错, `file stream` 和 `socket stream` 的出错情况完全不同。`socket stream` 可能遇到对方断开连接, `file stream` 可能遇到超出磁盘配额。

根据以上列举的初步分析, 我不认为有办法设计一个公共的基类把各方面的情况都考虑周全。各种 IO 设施之间共性太小, 差异太大, 例外太多。如果硬要用面向对象来建模, 基类要么太

瘦（只放共性，这个基类包含的 interface functions 没多大用），要么太肥（把各种 IO 设施的特性都包含进来，这个基类包含的 interface functions 很多，但是不是每一个都能调用）。C 语言对此的解决办法是用一个 int 表示 IO 对象（file 或 PIPE 或 socket），然后配以 read()/write()/lseek()/fcntl() 等一系列全局函数，程序员自己搭配组合。这个做法我认为比面向对象的方案要简洁高效。

iostream 在性能方面没有比 stdio 高多少，在健壮性方面多半不如 stdio，在灵活性方面受制于本身的复杂设计而难以让使用者自行扩展。目前看起来只适合一些简单的要求不高的应用，但是又不得不为它的复杂设计付出运行时代价，总之其定位有点不上不下。

在实际的项目中，我们可以提炼出一些简单高效的 strip-down 版本，在获得便利性的同时避免付出不必要的代价。

一个 300 行的 memory buffer output stream

我认为以 operator<< 来输出数据非常适合 logging，因此写了一个简单的 LogStream。代码不到 300行，完全独立于 iostream。

接口 <https://github.com/chenshuo/recipes/blob/master/logging/LogStream.h>

实现 <https://github.com/chenshuo/recipes/blob/master/logging/LogStream.cc>

单元测试 https://github.com/chenshuo/recipes/blob/master/logging/LogStream_test.cc

性能测试 https://github.com/chenshuo/recipes/blob/master/logging/LogStream_benchmark.cc

这个 LogStream 做到了类型安全和类型可扩展。它不支持定制格式化、不支持 locale/facet、没有继承、buffer 也没有继承与虚函数、没有动态分配内存、buffer 大小固定。简单地说，适合 logging 以及简单的字符串转换。

LogStream 的接口定义是

```
class LogStream : boost::noncopyable
{
    typedef LogStream self;
public:
    typedef detail::FixedBuffer Buffer;
    LogStream();

    self& operator<<(bool);

    self& operator<<(short);
    self& operator<<(unsigned short);
    self& operator<<(int);
    self& operator<<(unsigned int);
    self& operator<<(long);
    self& operator<<(unsigned long);
    self& operator<<(long long);
    self& operator<<(unsigned long long);

    self& operator<<(const void*);

    self& operator<<(float);
    self& operator<<(double);
    // self& operator<<(long double);

    self& operator<<(char);
    // self& operator<<(signed char);
```

```
// self& operator<<(unsigned char);
```

```
self& operator<<(const char*);
self& operator<<(const string&);
```

```
const Buffer& buffer() const { return buffer_; }
void resetBuffer() { buffer_.reset(); }
```

```
private:
    Buffer buffer_;
};
```

LogStream 本身不是线程安全的，它不适合做全局对象。正确的方式是每条 log 消息构造一个 LogStream，用完就扔。LogStream 的成本极低，这么做不会有什么性能损失。

目前这个 logging 库还在开发之中，只完成了 LogStream 这一部分。将来可能改用动态分配的 buffer，这样方便在线程之间传递数据。

整数到字符串的高效转换

muduo::LogStream 的整数转换是自己写的，用的是 Matthew Wilson 的算法，见 <http://blog.csdn.net/solstice/article/details/5139302>。这个算法比 stdio 和 ostream 都要快。

浮点数到字符串的高效转换

目前 muduo::LogStream 的浮点数格式化采用的是 snprintf() 所以从性能上与 stdio 持平，比 ostream 快一些。

浮点数到字符串的转换是个复杂的话题，这个领域 20 年以来没有什么进展（目前的实现大都基于 David M. Gay 在 1990 年的工作《Correctly Rounded Binary-Decimal and Decimal-Binary Conversions》，代码<http://netlib.org/fp/>），直到 2010 年才有突破。

Florian Loitsch 发明了新的更快的算法 Grisu3，他的论文《Printing floating-point numbers quickly and accurately with integers》发表在 PLDI 2010，代码见 Google V8 引擎，还有这里<http://code.google.com/p/double-conversion/>。有兴趣的同学可以阅读这篇博客 <http://www.serpentine.com/blog/2011/06/29/here-be-dragons-advances-in-problems-you-didnt-even-know-you-had/>。

将来 muduo::LogStream 可能会改用 Grisu3 算法实现浮点数转换。

性能对比

由于 muduo::LogStream 抛掉了很多负担，可以预见它的性能好于 ostream 和 stdio。我做了一个简单的性能测试，结果如下。

从上表看出，ostream 有时候比 snprintf 快，有时候比它慢，muduo::LogStream 比它们两个都快得多（double 类型除外）。

泛型编程

其他程序库如何使用 LogStream 作为输出呢？办法很简单，用模板。

前面我们定义了 Date class 针对 std::ostream 的 operator<<，只要稍作修改就能同时适用于 std::ostream 和 LogStream。而且 Date 的头文件不再需要 include <ostream>，降低了耦合。

```
class Date
```

```

{
public:
    Date(int year, int month, int day)
        : year_(year), month_(month), day_(day)
    {
    }

- void writeTo(std::ostream& os) const
+ template<typename OStream>
+ void writeTo(OStream& os) const
    {
        char buf[32];
        snprintf(buf, sizeof buf, "%d-%02d-%02d", year_, month_, day_);
        os << buf;
    }

private:
    int year_, month_, day_;
};

-std::ostream& operator<<(std::ostream& os, const Date& date)
+template<typename OStream>
+OStream& operator<<(OStream& os, const Date& date)
{
    date.writeTo(os);
    return os;
}

```

现实的 C++ 程序如何做文件 IO

举两个例子, Kyoto Cabinet 和 Google leveldb。

Google leveldb

Google leveldb 是一个高效的持久化 key-value db。

它定义了三个精简的 interface:

SequentialFile <http://code.google.com/p/leveldb/source/browse/trunk/include/leveldb/env.h#154>

RandomAccessFile <http://code.google.com/p/leveldb/source/browse/trunk/include/leveldb/env.h#178>

WritableFile <http://code.google.com/p/leveldb/source/browse/trunk/include/leveldb/env.h#197>

接口函数如下

```

struct Slice {
    const char* data_;
    size_t size_;
};

```

// A file abstraction for reading sequentially through a file

```

class SequentialFile {
public:
    SequentialFile() { }

```

```

virtual ~SequentialFile();

virtual Status Read(size_t n, Slice* result, char* scratch) = 0;
virtual Status Skip(uint64_t n) = 0;
};

// A file abstraction for randomly reading the contents of a file.
class RandomAccessFile {
public:
    RandomAccessFile() { }
    virtual ~RandomAccessFile();

    virtual Status Read(uint64_t offset, size_t n, Slice* result,
                        char* scratch) const = 0;
};

// A file abstraction for sequential writing. The implementation
// must provide buffering since callers may append small fragments
// at a time to the file.
class WritableFile {
public:
    WritableFile() { }
    virtual ~WritableFile();

    virtual Status Append(const Slice& data) = 0;
    virtual Status Close() = 0;
    virtual Status Flush() = 0;
    virtual Status Sync() = 0;
};

```

leveldb 明确区分 input 和 output, 进一步它又把 input 分为 sequential 和 random access, 然后提炼出了三个简单的接口, 每个接口只有屈指可数的几个函数。这几个接口在各个平台下的实现也非常简单明了 (http://code.google.com/p/leveldb/source/browse/trunk/util/env_posix.cc#35 http://code.google.com/p/leveldb/source/browse/trunk/util/env_chromium.cc#176), 一看就懂。

注意这三个接口使用了虚函数, 我认为这是正当的, 因为一次 IO 往往伴随着 context switch, 虚函数的开销比起 context switch 来可以忽略不计。相反, iostream 每次 operator<<() 就调用虚函数, 我认为不太明智。

Kyoto Cabinet

Kyoto Cabinet 也是一个 key-value db, 是前几年流行的 Tokyo Cabinet 的升级版。它采用了与 leveldb 不同的文件抽象。

KC 定义了一个 File class, 同时包含了读写操作, 这是个 fat interface。 http://fallabs.com/kyotocabinet/api/classkyotocabinet_1_1File.html

在具体实现方面, 它没有使用虚函数, 而是采用 #ifdef 来区分不同的平台 (见 <http://code.google.com/p/read-taobao-code/source/browse/trunk/tair/src/storage/kdb/kyotocabinet/kcfile.cc>), 等于把两份独立的代码写到了同一个文件里边。

相比之下, Google leveldb 的做法更高明一些。

小结

在 C++ 项目里边自己写个 File class，把项目用到的文件 IO 功能简单封装一下（以 RAII 手法封装 FILE* 或者 file descriptor 都可以，视情况而定），通常就能满足需要。记得把拷贝构造和赋值操作符禁用，在析构函数里释放资源，避免泄露内部的 handle，这样就能自动避免很多 C 语言文件操作的常见错误。

如果要用 stream 方式做 logging，可以抛开繁重的 iostream 自己写一个简单的 LogStream，重载几个 operator<<，用起来一样方便；而且可以用 stack buffer，轻松做到线程安全。

上一篇 [awk按分隔符的不同取出不同的列](#)

下一篇 [嵌入式学习路线](#)


主题推荐 [c](#)


猜你在找

- | | |
|--------------------------------|--|
| iOS8-Swift开发教程 | string的size和length |
| C语言及程序设计提高 | C++中WriteFile和ReadFile使用 |
| C++语言基础 | 一些大牛的博客推荐排名不分先后 |
| J2SE轻松入门第一季 | 抄袭事件判决书 |
| 微信公众平台开发入门 | 客户端产生CLOSE_WAIT状态的解决方案 |

准备好了么? [跳](#) 吧! 更多职位尽在 [CSDN JOB](#)

windows C/C++开发工程师	我要跳槽	C++开发工程师	我要跳槽
北京博睿宏远科技发展有限公司	6-10K/月	上海晶赞科技发展有限公司	12-24K/月
美国C++软件工程师	我要跳槽	linux C/C++开发	我要跳槽
北京玛赫西计算机教育咨询有限公司	40-80K/月	中国科学院通用芯片与基础软件研究...	15-20K/月






[PRE-ORDER NOW](#)

查看评论

4楼 [heathyhuhu](#) 2014-03-20 13:29发表


 so good!

3楼 [sun_prayer](#) 2013-12-29 02:47发表


 istr.str("1 56.7"); -> istr.str("1 56.7");

请修改

Re: [JoeBlackzqq](#) 2013-12-30 09:13发表

 回复[sun_prayer](#): thanks

2楼 [龙啸九天520](#) 2013-10-17 11:56发表

 太详细了 必须收藏啊

1楼 [hxq](#) 2012-09-03 19:13发表

 好得很

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场