1.endl 的本质

自从 C 语言教科书利用 Hello world 作为示例程序之后，很多程序设计语言的教科书都沿用了这个做法。我们写过的第一个 C++ 程序可能是这样的。

#include <iostream>

using namespace std;

int main() {

cout << "Hello world" << endl;

}

1

2

3

4

5

6

学习过 C 语言的程序猿自然会把输出语句与 C 语言中的输出语句联系起来，也就是说：

cout<<”Hello world”<<endl;相当于printf(“Hello world\n”);由于 endl 会导致输出的文字换行，自然而然地我们会想到 endl 可能就是换行符’\n’。但是，如果我们定义char c=endl;会得到一个编译错误，这说明 endl 并不是一个字符，所以应该到系统头文件中去查找 endl 的定义。通过 VS2017 转到定义，找到了 endl 的定义如下：

template<class \_Elem,class \_Traits> inline basic\_ostream<\_Elem, \_Traits>&

\_\_CLRCALL\_OR\_CDECL endl(basic\_ostream<\_Elem, \_Traits>& \_Ostr) {

// insert newline and flush stream

\_Ostr.put(\_Ostr.widen('\n'));

\_Ostr.flush();

return (\_Ostr);

}

1

2

3

4

5

6

7

从定义中看出，endl 是一个函数模板，它实例化之后变成一个模板函数，其作用如这个函数模板的注释所示，插入换行符并刷新输出流。其中刷新输出流指的是将缓冲区的数据全部传递到输出设备并将输出缓冲区清空。

2.cout 使用 endl

endl 是一个函数模板，再被使用时会实例化为模板函数。但是函数调用应该使用一对圆括号，也就是写成 endl() 的形式，而在语句cout<<”Hello world”<<endl;中并没有这样，原因何在？

在头文件 iostream 中，有这样一条申明语句：extern ostream& cout；这说明cout是一个ostream类对象。而<<原本是用于移位运算的操作符，在这里用于输出，说明它是一个经过重载的操作符函数。如果把endl当做一个模板函数，那么cout<<endl可以解释成cout.operator<<(endl);由于一个函数名代表一个函数的入口地址，所以在 cout 的所属类 ostream 中应该有一个operator<<()函数的重载形式接受一个函数指针做参数。

查找 ostream 类的定义，发现其实是另一个类模板实例化之后生成的模板类，即：

typedef basic\_ostream<char, char\_traits<char>> ostream;

1

所以，实际上应该在类模板basic\_ostream中查找operator<<()的重载版本。在头文件ostream中查找basic\_ostream的定义，发现其中operator<<作为成员函数被重载了17次，其中的一种：

typedef basic\_ostream<\_Elem, \_Traits> \_Myt;

\_Myt& \_\_CLR\_OR\_THIS\_CALL operator<<(\_Myt& (\_\_cdecl \*\_Pfn)(\_Myt&)) {

// call basic\_ostream manipulator

\_DEBUG\_POINTER(\_Pfn);

return ((\*\_Pfn)(\*this));

}

1

2

3

4

5

6

7

在ostream类中，operator<<作为成员函数重载方式如下：

ostream& ostream::operator<<(ostream& (\*op)(ostream&)) {

return (\*op)(\*this);

}

1

2

3

这个重载正好与endl函数的声明相匹配，所以<<后面是可以跟着endl 。也就是说，cout对象的<<操作符接收到endl函数的地址后会在重载的操作符函数内部调用endl函数，而endl函数会结束当前行并刷新输出缓冲区。

为了证明endl是一个函数模板，或者说endl是一个经过隐式实例化之后的模板函数，我们把程序改造如下：

#include <iostream>

using namespace std;

int main() {

cout<<"Hello world"<<&endl;

}

1

2

3

4

5

6

这个程序可以正常运行，并且结果完全同上一个程序。原因是对于一个函数而言，函数名本身就代表函数的入口地址，而函数名前加&也代表函数的入口地址。

3.endl 其实是 IO 操纵符

实际上，endl 被称为 IO 操纵符，也可翻译成 IO 算子。IO 操作符的本质是自由函数，他们并不封装在某个类的内部，使用时不采用显示的函数调用的形式。在<iostream>头文件中定义的操纵符有：

endl:输出时插入换行符并刷新流

ends：输出时插入NULL字符，通常用来结束一个字符串

flush：刷新缓冲区，把流从缓冲区输出到目标设备，并清空缓冲区

ws：输入时略去空白字符

dec：令IO数据按十进制格式输入或输出

hex:令IO数据按十六进制格式输入或输出

oct：令IO数据按八进制格式输入或输出

1

2

3

4

5

6

7

在<iomanip>头文件中定义的操作符有：

setbase(int)

resetiosflags(long)

setiosflags(long)

setfill(char)

setprecision(int)

setw(int)

1

2

3

4

5

6

这些格式控制符大致可以替代ios的格式函数成员的功能，且使用比较方便。例如，为了把整数345按16进制输出，可以采用两种方式：

int i=345;

cout.setf(ios::hex,ios::basefield);

cout<<i<<endl;

1

2

3

或者

cout<<hex<<i<<endl;

1

可以看出采用格式操纵符比较方便，二者的区别主要在于：格式成员函数是标准输出对象cout的成员函数，因此在使用时必须和cout同时出现，而操纵符是自由函数，可以独立出现，使用格式成员函数要显示采用函数调用的形式，不能用IO运算符”<<”和”>>”形成链式操作。

4.自定义格式操纵符

除了利用系统预定义的操纵符来进行IO格式的控制外，用户还可以自定义操纵符来合并程序中频繁使用的IO读写操作。定义形式如下：

输出流自定义操纵符：

ostream &操纵符名(ostream &s) {

　　// 自定义代码

　　return s;

}

1

2

3

4

输入流自定义操纵符：

istream &操纵符名(istream &s) {

　　// 自定义代码

　　return s;

}

1

2

3

4

示例代码如下：

#include <iostream>

#include <iomanip>

using namespace std;

// 编号格式如：0000001

std::ostream& OutputNo(std::ostream& s) {

s << std::setw(7) << std::setfill('0') << std::setiosflags(std::ios::right);

return s;

}

// 要求输入的数为十六进制数

std::istream& InputHex (std::istream& s) {

s>>std::hex;

return s;

}

int main() {

std::cout<<OutputNo<<8<<std::endl;

int a;

std::cout<<"请输入十六进制的数:";

std::cin>> InputHex >>a;

std::cout<<"转化为十进制数："<<a<<std::endl;

return 0;

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

程序运行结果：

0000008

请输入十六进制的数:ff

转化为十进制数:255

1

2

3

程序中OutputNo和InputHex都是用户自定义的格式操纵符，操作符的函数原型必须满足cout对象的成员函数operator<<()的重载形式：

ostream& ostream::operator<<(ostream& (\*op)(ostream&));

1

所以只要编写一个返回值为 std::ostream&，接收一个类型为 std::ostream& 参数的函数，就可以把函数的入口地址传递给cout.operator<<()，完成格式操纵符的功能。

————————————————

版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。

原文链接：https://blog.csdn.net/k346k346/article/details/49981695