Module04-07
C++ 标准库：I/O 流

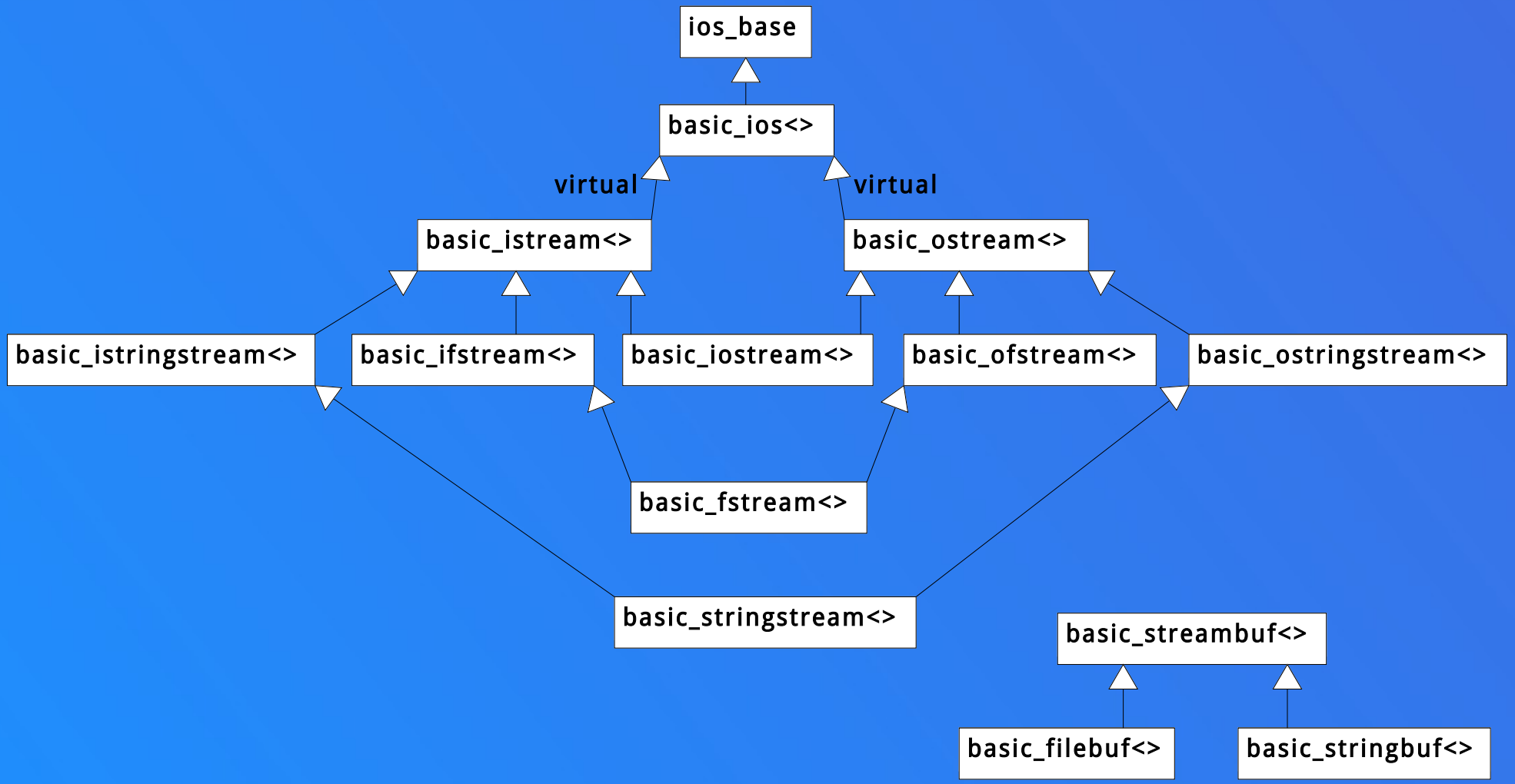
- 数据结构简介
- 标准容器
- 常用算法简介
- 标准算法与函数对象
- 迭代器
- 字符串
- → I/O 流
- 数值

标准库 - I/O 流

- I/O 流 (I/O Stream)
  - C++ I/O 流简介
  - 输出流 (Output Stream)
  - 输入流 (Input Stream)
  - 格式化与操控符 (Format & Manipulators)
  - 文件流与字符串流 (File Stream& String Stream)
  - 流与缓冲区 (Streams & Buffers)
  - 本地化 (Locale)
  - C 的输入 / 输出简介 (C I/O API)

I/O 流 - 简介

- C++ I/O 流类层次结构



I/O 流 - 输出流

- 关于输出流
  - ostream 的实际类型：

```
// in <ostream>
template<typename CharT, typename Traits>
class basic_ostream: virtual public basic_ios<CharT,
Traits> {
    // ...
};

// in <iosfwd>
namespace std {
// ...
typedef basic_ostream<char> ostream;
// ...
typedef basic_ostream<wchar_t> wostream; // 宽字符输出流
// ...
}
```

I/O 流 - 输出流

- 预定义的输出流对象

```
// in <iostream>
namespace std {
// ...
ostream cout;    // 标准输出
ostream cerr;    // 标准错误(无缓冲)
ostream clog;    // 标准错误(有缓冲)
wostream wcout; // 宽字符标准输出
wostream wcerr; // 宽字符标准错误(无缓冲)
wostream wclog; // 宽字符标准错误(有缓冲)
// ...
}
```

I/O 流 - 输出流

- 基本类型的输出
  - 成员函数

```
template<typename CharT, typename Traits>
class basic_ostream: virtual public basic_ios<CharT, Traits> {
    // ...
    basic_ostream& operator<<(int n);
    basic_ostream& operator<<(bool n);
    basic_ostream& operator<<(double f);
    // ...
    // output pointers
    basic_ostream& operator<<(const void* p);
    // output single character
    basic_ostream& put(char_type c);
    basic_ostream& write(const char_type* s, streamsize n);
    // ...
};
```

注意：并没有应用于 char 类型的 operator<<() 成员函数

- 基本类型的输出（续）
  - 非成员函数：对于 char 、 const char* 以及各自的 unsigned 类型的输出，采用非成员函数：

```
namespace std {
//...
// 特化
template<class Traits>
inline basic_ostream<char, Traits>&
operator<<(basic_ostream<char, Traits>& out, char c);
// ...

// 特化
template<class Traits>
inline basic_ostream<char, Traits>&
operator<<(basic_ostream<char, Traits>& out, const char* p);
//...
}
```

- 自定义类型的输出
  - 自定义类型的输出通过对操作符 << 进行重载：

```
class Employee {
    string name;
    short department;
    // ...
public:
    // ...
    friend ostream& operator<<(ostream& os, const Employee&
e);
};

ostream& operator<<(ostream& os, const Employee& e) {
    os << "Employee: name[" << e.name
       << "], department[" << e.department << "]";
    return os;
}
```

可不可以不声明成 friend 函数？

- 自定义类型的输出
  - 虚输出函数？
    - 考虑到性能， ostream 的输出函数不是 virtual 函数
    - 但希望强制一个类层次结构中所有 class 都必须提供输出函数，怎么办？

```
struct B {
    // ...
    virtual ostream& output(ostream& os) const = 0;
};
ostream& operator<<(ostream& os, const B& obj) {
    return obj.output(os);
}
struct D: B {
    virtual ostream& output(ostream& os) const {
        // ...
        return os;
    }
};
```

I/O 流 - 输入流

- 关于输入流
  - 输入流的实际类型

```
namespace std {
// in <istream>
template<typename CharT, typename Traits>
class basic_istream: virtual public basic_ios<CharT, Traits> {
    // ...
};

// in <iosfwd>
typedef basic_istream<char> istream;
typedef basic_istream<wchar_t> wistream;

// in <iostream>
extern istream cin;     // 标准输入(如键盘)
extern wistream wcin;    // 宽字符标准输入(如键盘)
}
```

I/O 流 - 输入流

- 基本类型的输入
  - 成员函数

```
template<typename CharT, typename Traits>
class basic_ostream: virtual public basic_ios<CharT, Traits> {
    // ...
    basic_istream& operator>>(int& n);
    basic_istream& operator>>(bool& n);
    basic_istream& operator>>(double& f);
    // ...
    // pointers
    basic_istream& operator>>(void*& p);
    // ...
};
```

- 基本类型的输入（续）
  - 非成员函数

```
namespace std {
template<typename CharT, typename Traits>
basic_istream<CharT, Traits>&
operator>>(basic_istream<CharT, Traits>& in, CharT& c);
//...

// 不断从输入流读入字符放入数组中，直到遇到空白字符或文件结束符停止
template<typename CharT, typename Traits>
basic_istream<CharT, Traits>&
operator>>(basic_istream<CharT, Traits>& in, CharT* s);
//...
}
```

输入操作符，默认会跳过前导的空白字符。

I/O 流 - 输入流

- 输入过程中的问题 (DEMO)
  - 输入类型不匹配，导致输入失败
  - 读入 C-Style 字符串，存在数组越界的隐患

```
    int k;
    cin >> k;    // input: cc

    char ca[5];
//    cin.width(5);
    cin >> ca;  // input: abcdefghijk
    cout << ca << endl;
```

I/O 流 - 输入流

- 流状态操作
  - 检查 C++ 流的状态的操作定义于基类 basic_ios 中 :

```
template<typename CharT, typename Traits>
class basic_ios: public ios_base {
public:
    // ...
    bool good() const;
    bool eof() const;
    bool fail() const;
    bool bad() const;

    iostate rdstate() const;
    void clear(iostate f = goodbit);
    void setstate(iostate f) { clear(rdstate() | f); }

    operator void*() const;
    bool operator!() const { return fail(); }
    // ...
};
```

- 流状态标志
  - C++ 流的状态标志定义于顶级基类 ios_base 中 :

```
class ios_base {
public:
    //...
    typedef _Ios_Iostate iostate;

    // 流已经破坏
    static const iostate badbit = _S_badbit;

    // 遇到文件结束
    static const iostate eofbit = _S_eofbit;

    // 下一个操作将失败
    static const iostate failbit = _S_failbit;

    // 下一次操作可能成功 , goodbit == 0
    static const iostate goodbit = _S_goodbit;
    //...
};
```

I/O 流 - 输入流

- 流状态的应用
  - 由于 operator void*() 和 operator !() 的重载，可以直接将流状态的检查用于条件判断的依据：

```
void f() {
    char c;
    while (cin >> c)
        cout << c;
}
```

I/O 流 - 输入流

- 字符的输入
  - 除了 operator>>() 一族函数用作格式化输入外，输入流还有一族函数用于非格式化输入

```
template<typename CharT, typename Traits>
class basic_istream: virtual public basic_ios<CharT, Traits>
{
    // ...
    streamsize gcount() const; // 最近一次非格式化输入的字符数
    int_type get();
    basic_istream& get(char_type& c);
    // 读入n-1个字符，不会丢弃结束符
    basic_istream& get(char_type* s, streamsize n, char_type
delim);
    basic_istream& get(char_type* s, streamsize n);
    // 读入n-1个字符，丢弃结束符
    basic_istream& getline(char_type* s, streamsize n,
char_type delim);
    basic_istream& getline(char_type* s, streamsize n); //以换
行符作为结束符
```

I/O 流 - 输入流

- 字符的输入（续）
  - 除了 operator>>() 一族函数用作格式化输入外，输入流还有一族函数用于非格式化输入

```
    basic_istream& ignore(); // 简单忽略一个字符
    basic_istream& ignore(streamsize n); // 忽略n个字符
    // 忽略n个字符或遇到delim停止
    basic_istream& ignore(streamsize n, int_type delim);

    // 读入n个字符
    basic_istream& read(char_type* s, streamsize n);
    //...

    // 将字符c放回流中
    basic_istream& putback(char_type c);
};
```

I/O 流 - 输入流

- 自定义类型的输入
  - 同自定义类型的输出类似

```
struct S {
    int k;
    double d;
};

istream& operator>>(istream& is, S& s) {
    is >> s.k >> s.d;
    return is;
}

int main() {
    S s;
    cin >> s;
    cout << "s.k: " << s.k << ", s.d: " << s.d << endl;
}
```

I/O 流 - 输入流

- std::string 的输入

```
// in <basic_string.h>
namespace std {
// ...
template<typename CharT, typename Traits, typename Alloc>
inline basic_istream<CharT, Traits>&
getline(basic_istream<CharT, Traits>& is,
        basic_string<CharT, Traits, Alloc>& str) {
    return getline(is, str, is.widen('\n'));
}

template<>
basic_istream<char>&
getline(basic_istream<char>& in, basic_string<char>& str,
char delim);
//...
}
```

- 流的格式化状态
  - 流的格式化状态定义于顶级基类 ios_base 中

```
class ios_base {
public:
    // ...
    typedef _Ios_Fmtflags fmtflags;
    static const fmtflags
        boolalpha, // 以字符 true/false表示bool值

        right, // 域调整，在值前填充
        internal, // 域调整，在符号和值之间填充
        left, // 域调整，在值后填充

        dec, // 以10进制形式显示整数
        hex, // 以16进制形式显示整数
        oct, // 以8进制形式显示整数

        fixed, // 浮点数格式显示浮点数: dddd.dd
        scientific, // 科学计数法显示浮点数: d.ddddddEdd
```

- 流的格式化状态（续）

```
        showbase, // 显示前缀，按8进制为0，16进制为0x
        showpoint, // 打印浮点数的后面的0，输出为8.000000
        showpos, // 强制显示正数前的+号
        uppercase, // 用大写E和X来显示，如0X3FFF、1.20E-9

        skipws, // 输入时跳过前导的空白字符
        unitbuf, // 在每次输出操作后刷新缓冲区

        adjustfield, // 域调整相关的标志组
        basefield, // 进制相关的标志组
        floatfield; // 浮点数显示相关的标志组

    fmtflags flags() const; // 读取标志
    fmtflags flags(fmtflags fmtfl); // 设置标志
    fmtflags setf(fmtflags fmtfl); // 添加标志
    fmtflags setf(fmtflags fmtfl, fmtflags mask); //在掩码中设
置标志
    void unsetf(fmtflags mask); //清除标志
    // ...
};
```

- 流的格式化状态（续）
  - 示例：

```
void f() {
    ios::fmtflags oldfmt = cout.flags(); // 保存当前的flags

    ios::fmtflags newfmt = ios::showbase | ios::left |
ios::hex | ios::scientific;
    cout.flags(newfmt); // 设置新的flags
    cout << 255 << ' ' << 12.09 << endl;

    cout.flags(oldfmt); // 恢复到原先的格式状态
    cout << 255 << ' ' << 12.09 << endl;

    cout.flags(cout.flags() | ios::scientific);
    cout << 12.09 << endl;

    cout.setf(ios::boolalpha);
    cout << true << endl;
}
```

I/O 流 - 格式化

- 流的格式化状态的复制
  - basic_ios::copyfmt()

```
basic_ios& cpoyfmt(const basic_ios& f);
```

I/O 流 - 格式化

- 整数的输出
  - 常用的格式设置（进制和显示基数）：
    - `cout.setf(ios::oct, ios::basefield) //  八进制`
    - `cout.setf(iso::dec, ios::basefield) //  十进制`
    - `cout.setf(ios::hex, ios::basefield) //  十六进制`
    - `cout.setf(ios::showbase) //` 显示基数
  - 上述各个设置将影响到随后所有的输出格式，直到重新设置为止

I/O 流 - 格式化

- 浮点数的输出
  - 常用的格式设置（浮点格式和精度）：
    - `cout.setf(ios::scientific, ios::floatfield)` // 科学计数法
    - `cout.setf(iso::fixed, ios::floatfield)` // 定点格式
    - `cout.setf(ios::fmtflags(0), ios::floatfield)` // 一般格式
    - `cout.precision() const` // 查看精度
    - `cout.precision(3)` // 小数精度设置为 3，默认为 6
  - 上述各个设置将影响到随后所有的输出格式，直到重新设置为止

- 输出域
  - 填充域跟输出宽度和填充的字符相关：

```
class ios_base {
public:
    // ...
    streamsize width() const; // 查看宽度
    streamsize width(streamsize wide); // 设置宽度, 0表示按输出内容
的宽度为准
    // ...
};

template<typename CharT, typename Traits>
class basic_ios: public ios_base {
public:
    // ...
    char_type fill() const; // 查看填充字符
    char_type fill(char_type c); // 设置填充字符
    // ...
};
```

I/O 流 - 格式化

- 输出域（续）
  - width(n) 只会影响随后的一次输出操作
  - 示例：

```
void f2() {
    cout.width(5);
    cout.fill('*');
    cout << "xy" << endl; // ***xy
}
```

I/O 流 - 格式化

- 域的调整
  - 即输出的内容按左对齐、右对齐、两端对齐等方式
    - `cout.setf(ios::left, ios::adjustfield)  // 左`
    - `cout.setf(ios::right, ios::adjustfield) // 右`
    - `cout.setf(ios::internal, ios::adjustfield)// 中`
  - 与 `width(n)` 配合使用

- 域的调整（续）
  - 示例：

```
void f2() {
    cout.fill('#');
    cout.width(5);
    cout << -88 << endl; // ##-88 , ios::right default

    cout.width(5);
    cout.setf(ios::left, ios::adjustfield);
    cout << -88 << endl; // -88##

    cout.width(5);
    cout.setf(ios::internal, ios::adjustfield);
    cout << -88 << endl; // -##88
}
```

I/O 流 - 操控符

- 操控符 (Manipulators)
  - 操控符是如何适用于格式化输出、输入的:

```
template<typename CharT, typename Traits>
class basic_ostream: virtual public basic_ios<CharT, Traits> {
public:
    basic_ostream& operator<<(basic_ostream& (*f)
(basic_ostream&));
    basic_ostream& operator<<(basic_ios& (*f)(basic_ios &));
    basic_ostream& operator<<(ios_base& (*f)(ios_base&));
};

template<typename CharT, typename Traits>
class basic_istream: virtual public basic_ios<CharT, Traits> {
public:
    basic_istream& operator>>(basic_istream& (*f)
(basic_istream&));
    basic_istream& operator>>(basic_ios& (*f)(basic_ios&));
    basic_istream& operator>>(ios_base& (*f)(ios_base&));
};
```

I/O 流 - 操控符

- 操控符的实现细节
  - 查看 <iomanip>
- 比较流格式设置与操控符

```
// 流格式设置
cout.width(5);
cout.fill('#');
cout << 9;

// 等效的操控符
cout << setw(5) << setfill('#') << 9;
```

I/O 流 - 操控符

- 标准 I/O 操控符

```
// in ios_base
ios_base& boolalpha(ios_base& base);
ios_base& noboolalpha(ios_base& base);
ios_base& showbase(ios_base& base);
ios_base& noshowbase(ios_base& base);
ios_base& showpoint(ios_base& base);
ios_base& noshowpoint(ios_base& base);
ios_base& showpos(ios_base& base);
ios_base& noshowpos(ios_base& base);
ios_base& skipws(ios_base& base);
ios_base& noskipws(ios_base& base);
ios_base& uppercase(ios_base& base);
ios_base& nouppercase(ios_base& base);
ios_base& unitbuf(ios_base& base);
ios_base& nounitbuf(ios_base& base);
ios_base& internal(ios_base& base);
ios_base& left(ios_base& base);
ios_base& right(ios_base& base);
ios_base& dec(ios_base& base);
ios_base& hex(ios_base& base);
ios_base& oct(ios_base& base);
ios_base& fixed(ios_base& base);
ios_base& scientific(ios_base& base);
```

I/O 流 - 操控符

- 标准 I/O 操控符（续）

```
// in class basic_ostream
basic_ostream& endl(basic_ostream&); // 加'\n'且刷新
basic_ostream& ends(basic_ostream&); // 加'\0'
basic_ostream& flush(basic_ostream&); // 刷新流

// in class basic_istream
basic_istream& ws(basic_ostream&); // skip white space
```

```
// in <iomanip>
_Resetiosflags resetiosflags(ios_base::fmtflags mask);
_Setiosflags setiosflags(ios_base::fmtflags mask);
_Setbase setbase(int base);
_Setfill<CharT> setfill(CharT c);
_Setprecision setprecision(int n);
_Setw setw(int n);
```

I/O 流 - 操控符

- 标准 I/O 操控符的使用
  - 不带参数的操控符都不需带括号调用
  - 使用带参数的操控符需包含头文件 <iomanip>

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

int main() {
    // 注意: 操控符 scientific 和 endl 没有括号
    cout << setw(20) << setfill('*')
         << scientific << 12.09 << endl;
}
```

I/O 流 - 文件流和字符串流

- 文件流
  - 文件输入流

```
template<typename CharT, typename Traits>
class basic_ifstream: public basic_istream<CharT, Traits> {
public:
    basic_ifstream();
    explicit basic_ifstream(const char* s, ios::openmode mode
= ios::in);
    ~basic_ifstream();
    basic_filebuf<char_type, traits_type>* rdbuf() const;
    bool is_open();
    bool is_open() const;
    void open(const char* s, ios::openmode mode = ios::in);
    void close();
};
```

- 文件流（续）
  - 文件输出流

```
template<typename CharT, typename Traits>
class basic_ofstream: public basic_ostream<CharT, Traits> {
public:
    basic_ofstream();
    explicit basic_ofstream(const char* s,
        ios::openmode mode = ios::out | ios::trunc);
    ~basic_ofstream();
    basic_filebuf<char_type, traits_type>* rdbuf() const;
    bool is_open();
    bool is_open() const;
    void open(const char* s,
        ios::openmode mode = ios::out | ios::trunc);
    void close();
};
```

I/O 流 - 文件流和字符串流

- 文件流（续）
  - 文件流的打开模式，在 class ios_base 中定义

```
class ios_base {
    // ...
    typedef _Ios_Openmode openmode;
    static const openmode
        app, // 尾部追加
        ate, // 打开文件且定位到末尾(at end)
        binary, //二进制I/O
        in, // 为读而打开
        out, // 为写而打开
        trunc; // 将文件内容清空
    // ...
};
```

I/O 流 - 文件流和字符串流

- 文件流（续）
  - 示例：文件复制 (DEMO)

- 字符串流
  - 字符串输出流

```
// in <sstream>
template<typename CharT, typename Traits, typename Alloc>
class basic_ostringstream: public basic_ostream<CharT,
Traits> {
public:
    // ...
    explicit basic_ostringstream(ios::openmode mode =
ios::out);
    explicit basic_ostringstream(const string_type& str,
ios::openmode mode = ios::out);
    ~basic_ostringstream();
    stringbuf_type* rdbuf() const;
    string_type str() const;
    void str(const string_type& s);
};
```

- 字符串流
  - 字符串输入流

```
// in <sstream>
template<typename CharT, typename Traits, typename Alloc>
class basic_istringstream: public basic_istream<CharT,
Traits> {
public:
    // ...
    explicit basic_istringstream(ios::openmode mode =
ios::in);
    explicit basic_istringstream(const string_type& str,
ios::openmode mode = ios::in);
    ~basic_istringstream();
    stringbuf_type* rdbuf() const;
    string_type str() const;
    void str(const string_type& s);
};
```

- 字符串流
  - 示例

```
ostringstream os("Date: ", ios::ate);
// ostringstream os("Date: "); // 与上行的输出的区别？
os << setw(4) << setfill('0') << 121 << '-' << setw(2);
os << 6 << '-' << setw(2) << 8;
cout << os.str() << endl;
os.str(""); // 清空os的内容

string line("number 0xff");
istringstream is(line);
string s;
is >> s;
cout << "First word is a string: " << s << endl;
int n;
is >> std::hex >> n; // 如果不要 std::hex 操控符 ？
cout << "Second word is a number: " << n << endl;
```

- 输出流与缓冲区 (Buffers)
  - 输出流与缓冲区操作相关的函数：

```
template<typename CharT, typename Traits>
class basic_ostream: virtual public basic_ios<CharT, Traits>
{
public:
    // ...
    explicit basic_ostream(basic_streambuf<CharT, Traits>*
sb);

    basic_ostream& flush(); // 刷新缓冲区，清空缓冲区
    pos_type tellp(); // 返回当前位置
    basic_ostream& seekp(pos_type); // 设置当前位置
    basic_ostream& seekp(off_type, ios::seekdir); //设置当前位置
    basic_ostream& operator<<(basic_streambuf<CharT, Traits>*
sb);
    // ...
};
```

- 输出流与缓冲区 (Buffers)（续）
  - 关于 seekdir：
    - seekp() 函数返回写入的位置，只对文件流有效
    - seekdir 在类 ios_base 中定义

```
class ios_base {
    //...
    typedef _Ios_Seekdir seekdir;
    static const seekdir
        beg, // 当前文件的开始
        cur, // 当前位置
        end; // 当前文件的结束
    //...
};
```

I/O 流 - 流与缓冲区

- 输入流与缓冲区 (Buffers)
  - 输入流与缓冲区操作相关的函数：

```
template<typename CharT, typename Traits>
class basic_istream: virtual public basic_ios<CharT, Traits>
{
public:
    // ...
    explicit basic_istream(basic_streambuf<CharT, Traits>*
sb);

    int_type peek(); // 查看系一个字符但不取出
    streamsize readsome(char_type* s, streamsize n); // 最多读n
个字符
    basic_istream& putback(char_type c); // 将c放回缓冲区
    basic_istream& unget(); // 将最近读取的一个字符放回缓冲区

    int sync(); // 清空缓冲区
```

I/O 流 - 流与缓冲区

- 输入流与缓冲区 (Buffers)
  - 输入流与缓冲区操作相关的函数：

```
    pos_type tellg(); // 返回当前读取的位置
    basic_istream& seekg(pos_type); // 读取的位置
    basic_istream& seekg(off_type, ios_base::seekdir);

    basic_istream& operator>>(basic_streambuf<CharT, Traits>*
sb);
    basic_istream& get(basic_streambuf<CharT, Traits>& sb,
char_type delim);
    basic_istream& get(basic_streambuf<CharT, Traits>& sb) {
        return this->get(sb, this->widen('\n'));
    }
    // ...
};
```

I/O 流 - 本地化

- 本地化 (Locale)
  - 本地化也就是一个控制不同类别字符的划分（如字母、数字等）、字符串的和对顺序、数值在输入输出中的出现形式等的对象

I/O 流 - C 输入 / 输出

- 与 C 的 I/O 操作共享缓冲区
  - 在第一次 I/O 操作前调用 sync_with_stdio()

```
class ios_base {
public:
    //...
    static bool sync_with_stdio(bool sync = true);
    //...
};
```

I/O 流 - C 输入 / 输出

- 常用的 C 输入输出函数

```
// in <stdio.h>
// 输出操作
int fprintf (FILE* stream, const char* format, ...);
int printf (const char* format, ...);
int sprintf (char* s, const char* format, ...);
// 输入操作
int fscanf (FILE* stream, const char* format, ...);
int scanf (const char* format, ...);
int sscanf (const char* s, const char* format, ...);
```

I/O 流 - Bjarne's Advices

- Bjarne's Advices
  - 尽量少用 get() 、 read() 等函数
  - 在使用 get() 、 getline() 、 read() 时要注意终止准则
  - 在控制 I/O 格式时，尽量采用操控符，少用状态标志
  - 在使用标准操控符时记住 #include <iomanip>
  - 用字符串流做内存里的格式化