5 Standard Library – Using I/O streams

242

Introduction

- The Standard Library organizes all kinds of I/O operations through a standard class '**iostream**'

- We've already used class iostream several types through the objects cin and cout

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

int main() {
  double x;

  // Read x from standard input
  cin >> x ;

  // Write x to standard output
  cout << "x = " << x << endl ;

  return 0 ;
}
```
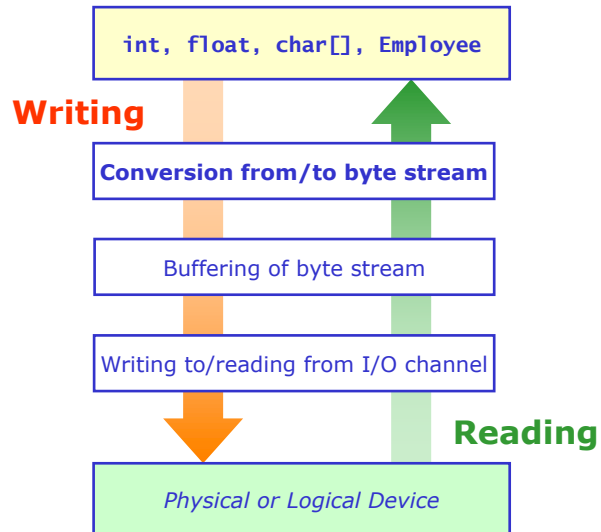
- We will now take a better look at how streams work

243

A look behind the scenes

- I/O in C++ involves three distinct steps

int, float, char[], Employee

Writing

Conversion from/to byte stream

Buffering of byte stream

Writing to/reading from I/O channel

Reading

Physical or Logical Device

244

I/O classes and operators in C++

- Operators<<(), >>() do step 1, classes istream, ostream do step 2

int, float, char[], Employee

operator<<()

Conversion from/to byte stream

operator>>()

Buffering of byte stream

class ostream

Writing to/reading from I/O channel

class istream

Physical or Logical Device

245

2
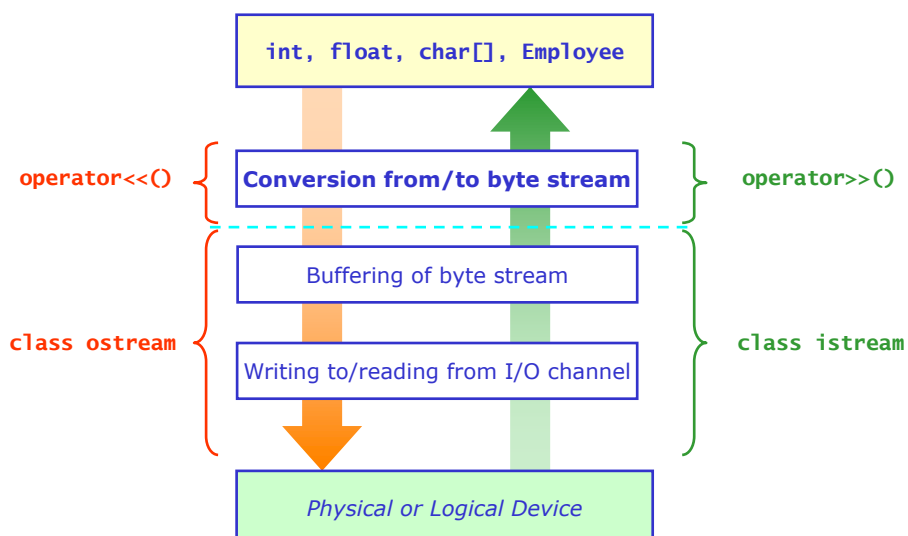
Stream object in the Standard Library

- Stream classes in Standard Library

Include file	Logical or physical device	Direction of byte stream		
		Input	Output	Both
`<iostream>`	Generic (e.g.terminal)	**istream**	**ostream**	**iostream**
`<fstream>`	File	**ifstream**	**ofstream**	**fstream**
`<sstream>`	std::string	**istringstream**	**ostringstream**	**stringstream**

- Standard Library stream classes also implement all operators to convert built-in types to byte streams
  - Implemented as member operators of stream class
  - Example: `ostream::operator<<(int) ;`

- Standard Library also provides three global stream objects for 'standard I/O'
  - `istream` object `cin` for 'standard input'
  - `ostream` objects `cout`,`cerr` for 'standard output','standard error'

246

Using streams without operators >>(),<<()

- Streams provide several basic functions to read and write bytes
  - Block operations

```
char buf[100] ;
int count(99) ;

// read 'count' bytes from input stream
cin.read(buf, count) ;

// write 'count' bytes to output stream
cout.write(buf, count) ;
```

247

3

Using streams without operators >>(),<<()

- Streams provide several basic functions to read and write bytes
  - Line oriented operations

```
// read line from stdin up to and including the newline char
cin.get(buf,100) ;

// read line from std up to newline char
cin.getline(buf,100) ;

// read line up to and including ':'
cin.get(buf,100,':') ;

// read single character
cin.get(c) ;

// write buffer up to terminating null byte
cout.write(buf,strlen(buf)) ;

// write single character
cout.put(c) ;
```

248

How is the stream doing?

- Member functions give insight into the *state* of the stream

Function	Meaning
bool good()	Next operation *might* succeed
bool eof()	End of input seen
bool fail()	Next operation will fail
bool bad()	Stream is corrupted

- Example – reading lines from a file till the end of the file

```
ifstream ifs("file.txt") ;
char buf[100] ;

// Loop as long as stream is OK
while(!ifs.fail()) {
    ifs.getline(buf,100) ;

    // Stop here if we have reached end of file
    if (ifs.eof()) break ;

    cout << "just read '" << buf << "'" << endl ;
}
```

249

Some handy abbreviations

- Streams overload operator `void*()` to return `!fail()`
  - Can shorten preceding example to

    ```
    while(ifs) { // expanded to while(ifs.operator void*())
        ifs.getline(buf,100) ;
        if (ifs.eof()) break ;
        cout << "just read '" << buf << "'" << endl ;
    }
    ```

- Also return value of `getline()` provides similar information
  - Returns true if stream is good() and stream is not at eof() after operation

    ```
    while(ifs.getline(buf,100)) {
        cout << "just read '" << buf << "'" << endl ;
    }
    ```

250

Using stream operators

- The next step is to use the streaming operators instead of the 'raw' IO routines
  - Encapsulation, abstraction → let objects deal with their own streaming
- Solution: use `operator>>()` instead of `getline()`

  shoesize.txt

  ```
  Bjarne 42
  Leif 47
  Thor 52
  ```

  ```
  ifstream ifs("shoesize.txt") ;
  string name ;
  int size ;

  while(ifs >> name >> size) {
      cout << name << " has shoe size " << size << endl ;
  }
  ```

251

5

Using stream operators

- Remember: syntax of stream operators is like that of any other operator

```
string name ;
int size ;
cin >> name >> size ;
```

```
string name ;
int size ;
cin.operator>>(   cin.operator>>(name),   size) ;
```

- For all built-in types
  - `operator<<(ostream,TYPE)` and `operator>>(istream,TYPE)` are implemented as member functions of the streams
  - Special case: `operator<<(const char*)` and `operator>>(char*)` read and write `char[]` strings

252

Parsing input – some fine points

- Delimiters
  - How does the text line

    `Bjarne Stroustrup 42`

    map on to the statement

    `cin >> firstName >> lastName >> shoeSize ;`

  - Because each `operator>>()` stops reading when it encounters 'white space'
  - White space is 'space','tab','vertical tab','form feed' and 'newline'
  - White space between tokens is automatically 'eaten' by the stream

- Reading string tokens
  - Be careful using `char[]` to read in strings: `operator>>(const char*)` does not know your buffer size and it can overrun!
  - Better to use `class string`

253

Formatting output of built-in types

- For built-in types streams have several functions that control formatting
    - Example: manipulating the base of integer output

    ```
    cout.setf(ios_base::oct,ios_base::basefield) ; // set octal
    cout << 1234 << endl ; // shows '02322'

    cout.setf(ios_base::hex,ios_base::basefield) ; // set hex
    cout << 1234 << endl ; // shows '0x4d2'
    ```

    - But it is often inconvenient to use this as calling formatting function interrupt chained output commands
- To accomplish formatting more conveniently streams have 'manipulators'
    - Manipulators are 'pseudo-objects' that change the state of the stream on the fly:

    ```
    cout << oct << 1234 << endl << hex << 1234 << endl ;
    // shows '2322' '4d2'
    ```

© 2006 Wouter Verkerke, NIKHEF

254

Overview of manipulators

- So manipulators are the easiest way to modify the formatting of built-in types
- What manipulators exist?
    - Integer formatting

Manipulator	Stream type	Description
dec	iostream	decimal base for integer
hex	iostream	hexadecimal base for integer
oct	iostream	octal base for integer
[no]showpos	iostream	show '+' for positive integers
setbase(int n)	iostream	base n for integer

    - Floating point formatting

Manipulator	Stream type	Description
setprecision(int n)	iostream	show n places after decimal point
[no]showpoint	iostream	[don't]show trailing decimal point
scientific	iostream	scientific format x.xxexx
uppercase	iostream	print 0XFF, nnExx
fixed	iostream	format xxxx.xx

255

7

Manipulators – continued

– Alignment & general formatting

Manipulator	Stream type	Description
left	iostream	align left
right	iostream	align right
internal	iostream	use internal alignment for each type
setw(int n)	iostream	next field width is n positions
setfill(char c)	iostream	set field fill character to c

– Miscellaneous

Manipulator	Stream type	Description
endl	ostream	put '\n' and flush
ends	ostream	put '\0' and flush
flush	ostream	flush stream buffers
ws	istream	eat white space
setfill(char c)	iostream	set field fill character to c

– Include `<iomanip>` for most manipulator definitions

256

Formatting output with manipulators

- Very clever, but how do manipulators work?
  - A manipulator is a 'pseudo-object' that modifies the state of the stream
  - More precisely: a manipulator is a *static member function* of the stream that takes a stream as argument, for example

```
class ostream {
  static ostream& oct(ostream& os) {
    os.setf(ios::oct,ios::basefield) ;
  }
} ;
```

  - The manipulator applies its namesake modification to the stream argument
  - You put manipulators in your print statement because class ostream also defines

```
operator<<(ostream&(*f)(ostream&)) {
  return f(*this) ;
}
```

  This operator processes any function that takes a single ostream& as argument and returns an ostream. The operator calls the function with itself as argument, which then causes the wanted operation to be executed on itself

257

8

Random access streams

- Streams tied to files and to strings also allow random access

  - Can move 'current' position for reading and writing to arbitrary location in file or string

member function	stream	Description
streampos tellg()	input	return current location of 'get()' position
seekg(streampos)	input	set location of 'get()' position
streampos tellp()	output	return current location of 'put()' position
seekp(streampos)	output	set location of 'put()' position

  - Streams open for both input and output (fstream, stringstream) have all four methods, where put() and get() pointer can be in different positions
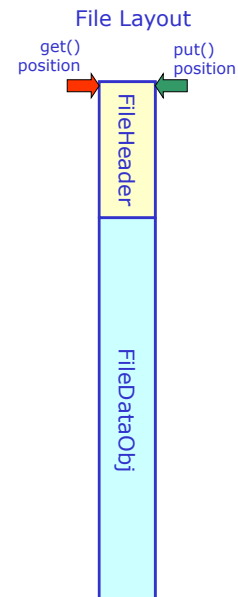
258

Random access streams

- Example use of tell(),seek()

```
#include <fstream>

// Open file for reading and writing
fstream iofile("file.dat",ios::in|ios::out) ;
```

File Layout

get()
position

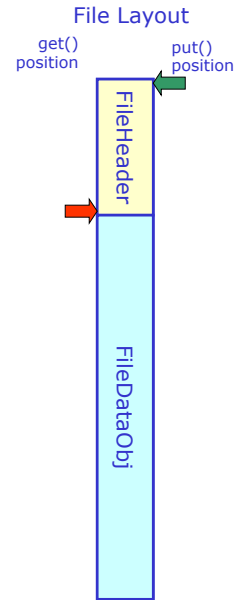put()
position

FileHeader

FileDataObj

259

Random access streams

- Example use of `tell()`,`seek()`

```
#include <fstream>

// Open file for reading and writing
fstream iofile("file.dat",ios::in|ios::out) ;

// Read in (fictitious) file header
FileHeader hdr ;
iofile  >> hdr ;
```

File Layout

get()
position

put()
position

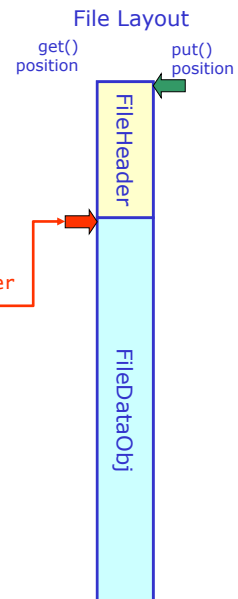FileHeader

FileDataObj

Random access streams

- Example use of `tell()`,`seek()`

```
#include <fstream>

// Open file for reading and writing
fstream iofile("file.dat",ios::in|ios::out) ;

// Read in (fictitious) file header
FileHeader hdr ;
iofile  >> hdr ;

// Store current location of stream 'get()' pointer
streampos marker = ifs.tellg() ;
```

File Layout

get()
position

put()
position

FileHeader

FileDataObj

Random access streams
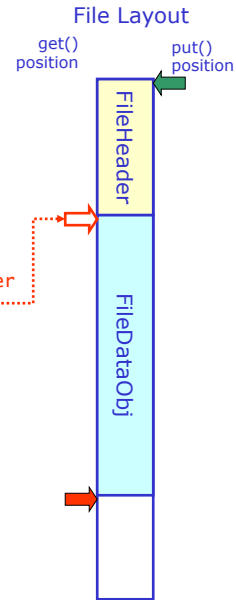
- Example use of `tell()`,`seek()`

```
#include <fstream>

// Open file for reading and writing
fstream iofile("file.dat",ios::in|ios::out) ;

// Read in (fictitious) file header
FileHeader hdr ;
iofile  >> hdr ;

// Store current location of stream 'get()' pointer
streampos marker = ifs.tellg() ;

// Read (fictitious) file data object
FileDataObj fdo ;
iofile >> fdo ;
```

File Layout

get()
position

put()
position

FileHeader

FileDataObj

262

Random access streams

- Example use of `tell()`,`seek()`

```
#include <fstream>

// Open file for reading and writing
fstream iofile("file.dat",ios::in|ios::out) ;

// Read in (fictitious) file header
FileHeader hdr ;
iofile  >> hdr ;

// Store current location of stream 'get()' pointer
streampos marker = ifs.tellg() ;

// Read (fictitious) file data object
FileDataObj fdo ;
iofile >> fdo ;

// modify file data object

// Move current' location of stream
// 'put '()' pointer to marked position
ifs.tellp(marker) ;
```
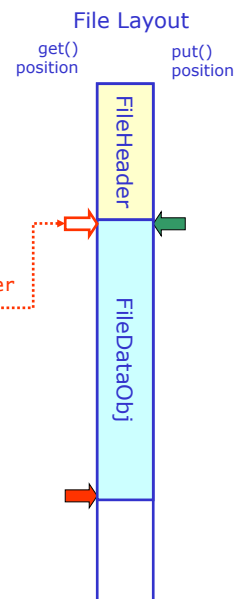
File Layout

get()
position

put()
position

FileHeader

FileDataObj

263

11

Random access streams

- Example use of `tell()`,`seek()`

```
#include <fstream>

// Open file for reading and writing
fstream iofile("file.dat",ios::in|ios::out) ;

// Read in (fictitious) file header
FileHeader hdr ;
iofile  >> hdr ;

// Store current location of stream 'get()' pointer
streampos marker = ifs.tellg() ;

// Read (fictitious) file data object
FileDataObj fdo ;
iofile >> fdo ;

// modify file data object

// Move current' location of stream
// 'put '()' pointer to marked position
ifs.tellp(marker) ;

// Write modified object over old location in file
iofile << fdo ;
```
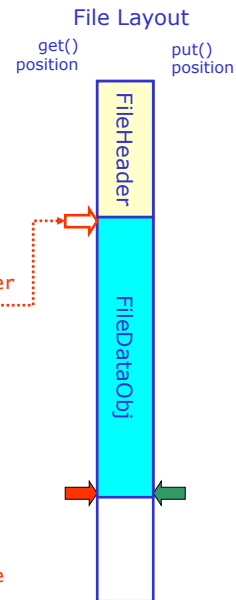
File Layout

get() position | put() position

FileHeader

FileDataObj

264

Streaming custom classes

- You can stream custom classes by defining your matching `operator<<()`,`operator>>()` for those classes
  - Standard Library stream classes implement operators <<, >> as member functions for streaming of all basic types basic types
  - This is not an option for you as you can't modify the Standard Library classes

  - But in general, binary operators can be
    1. member of class `ostream(cout)`,
    2. member of your class, or
    3. be a global function.

  - Option 1) already ruled out
  - Option 2) doesn't work because class being read/written needs to be *right*most argument of operator, while as a member function it is by construction the *left* argument of the operator
  - Option 3) works: implement `operator<<` as global operator

265

Streaming custom classes

- For types that can be printed on a single line, overloading the operator<<, operator>> is sensible
  - Class string obvious example

```
String s("Hello") ;
cout << string << " World" ;
```

```
String s("Hello") ;
cout.operator<<(operator<<(cout,string),"World") ;
```

- For classes that read/write multi-line output, consider a separate function
  - operator>>,<< syntax for such cases potentially confusing: processing white space etc traditionally handled by stream not by operator
  - Example names: readFromStream(),writeToStream()

266

Implementing your own <<,>> operators

- Important: operators <<,>> need to return a reference to the input ostream, istream respectively
  - Essential for ability to chain << operations
    ```
    cin >> a >> b ;
    cout << a << b << c ;
    ```
- Example implementation for class string

```
ostream& operator<<(ostream& os, const String& s) {
   os << s._s ;
   return os ;
}

istream& operator>>(istream& is, String& s) {
   const int bufmax = 256 ;
   static char buf[256] ;
   is >> buf ;
   s = buf ;
   return is ;
}
```

Note: no const here
as String is modified

267