

Streams

Object-Oriented Programming with C++

Why streams?

- Original C I/O used printf, scanf
- Streams introduced in C++
 - C I/O libraries still work
- Advantages of streams
 - Better type safety
 - Extensible
 - More object-oriented

Why streams?

- Disadvantages
 - More verbose (`std::format` comes back in C++20)
 - Might be slower
 - Turn off synchronization:
`std::ios::sync_with_stdio(false);`

C vs. C++

- C stdio operations work
 - No *object-oriented* features
 - No overloadable operators
- C++
 - Can overload *inserters* and *extractors*
- Moral
 - When converting C to C++, leave the I/O intact

What is a stream?

- Common logical interface to a device
 - One-dimension, unidirectional
 - Random access on file, but not on `std::cin/cout`

Stream naming conventions

	Input	Output	Header
Generic	istream	ostream	<iostream>
File	ifstream	ofstream	<fstream>
C string	istrstream	ostrstream	<strstream>
C++ string	istringstream	ostringstream	<sstream>

Stream operations

- Extractors
 - Read a value from the stream
 - Overload the `operator>>`
- Inserters
 - Insert a value into a stream
 - Overload the `operator<<`
- Manipulators
 - Change the stream state

Kinds of streams

- Text streams
 - Deal with ASCII text
 - Perform some characters translation
 - e.g., newline => actual OS file representation
 - Include
 - files, character buffers
- Binary streams
 - Binary data
 - No translation

Predefined streams

- `cin`
 - standard input
- `cout`
 - standard output
- `cerr`
 - unbuffered error (debugging) output
- `clog`
 - buffered error (debugging) output

Examples

```
#include <iostream>
int i; float f; char c;
char buffer[80];

// Read the next character
cin >> c;
// Read an integer, skips whitespace
cin >> i;
// Read a float and a string separated by whitespace
cin >> f >> buffer;
```

Predefined extractors

- `istream >> lvalue`

Expression type	input format	C I/O
<i>char</i>	Character	<code>%c</code>
<i>short, int</i>	Integer	<code>%d</code>
<i>long</i>	Long decimal integer	<code>%ld</code>
<i>float</i>	Floating point	<code>%g</code>
<i>double</i>	Double precision flp.	<code>%lg</code>
<i>char[], char*</i>	String	<code>%s</code>
<i>void*</i>	Pointer	<code>%p</code>

- Extractors skip leading whitespace, in general

Defining a stream extractor

- Has to be a 2-argument free function
 - First argument is an `istream&`
 - Second argument is a *reference* to a value

```
istream& operator>>(istream& is, T& obj) {  
    // specific code to read obj  
    return is;  
}
```

- Return an `istream&` for chaining

```
cin >> a >> b >> c;    // ((cin >> a) >> b) >> c;
```

Other input operators

- `int get()`
 - Returns the next character in the stream
 - Returns EOF if no characters left
 - Example: copy input to output

```
int ch;  
while ((ch = cin.get()) != EOF)  
    cout.put(ch);
```

More input operators

- `get(char *buf, int limit, char delim='\n')`
 - Read up to `limit` characters, or to `delim`
 - Appends a NULL character to `buf`
 - *Does not* consume the delimiter
- `getline(char *b, int l, char d='\n')`
 - Similar to above
 - *Does* consume the delimiter

More input operators

- `int gcount()`
 - Returns number of characters just read
- `void putback(char c)`
 - Pushes a single character back into the stream
- `char peek()`
 - Examines the next character without consuming it

Predefined inserters

- `ostream << expression`

Expression type	input format	C I/O
<i>char</i>	Character	<code>%c</code>
<i>short, int</i>	Integer	<code>%d</code>
<i>long</i>	Long decimal integer	<code>%ld</code>
<i>float</i>	Floating point	<code>%g</code>
<i>double</i>	Double precision flp.	<code>%lg</code>
<i>char[], char*</i>	String	<code>%s</code>
<i>void*</i>	Pointer	<code>%p</code>

Creating a stream inserter

- Has to be a 2-argument free function
 - First argument is an `ostream&`
 - Second argument is any value

```
ostream& operator<<(ostream& os, const T& obj) {  
    // specific code to write obj  
    return os;  
}
```

- Return an `ostream&` for chaining

```
cout << a << b << c;    // ((cout << a) << b) << c;
```

Other output operators

- `put(char)`
 - Prints a single character
- `flush()`
 - Force output of stream contents

```
cout << "Enter a number";  
cout.flush();
```

Formatting using manipulators

- Manipulators modify the state of the stream
 - `#include <iomanip>`
 - Effects hold (usually)
- Example

```
int n;  
cout << "enter number in hexadecimal" << endl;  
cin >> hex >> n;
```

Example

- A simple printing program

```
#include <iostream>
#include <iomanip>
int main() {
    cout << setprecision(2) << 1230.243 << endl;
    cout << setw(20) << "OK!";
    return 0;
}
```

1.2e+03

OK!

Manipulators

Manipulator	Effect	Type
<i>dec, hex, oct</i>	Set numeric conversion	I,O
<i>endl</i>	Insert newline and flush	O
<i>flush</i>	Flush stream	O
<i>setw(int)</i>	Set field width	I,O
<i>setfill(ch)</i>	Change fill character	I,O
<i>setbase(int)</i>	Set number base	O
<i>ws</i>	Skip whitespace	I
<i>setprecision(int)</i>	Set floating point precision	O
<i>setiosflags(long)</i>	Turn on specified flags	I,O
<i>resetiosflags(long)</i>	Turn off specified flags	I,O

Creating manipulators

- You can define your own manipulators!

```
// skeleton for an output stream manipulator
ostream& manip(ostream& out) {
    ...
    return out;
}

ostream& tab(ostream& out) {
    return out << '\t';
}

cout << "Hello" << tab << "World!" << endl;
```

Stream flags control formatting

Flag	Purpose (when set)
<i>ios::skipws</i>	Skip leading white space
<i>ios::left, ios::right</i>	Justification
<i>ios::internal</i>	Pad between sign and value
<i>ios::dec, ios::oct, ios::hex</i>	Format for numbers
<i>ios::showbase</i>	Show base of number
<i>ios::showpoint</i>	Always show decimal point
<i>ios::uppercase</i>	Put base in uppercase
<i>ios::showpos</i>	Display + on positive numbers
<i>ios::scientific, ios::fixed</i>	Floating point format
<i>ios::unitbuf</i>	Flush on every write

Setting flags

- Using manipulators
 - `setiosflags(flags);`
 - `resetiosflags(flags);`
- Using stream member functions
 - `setf(flags);`
 - `unsetf(flags);`