MODERN C++ DESIGN PATTERNS

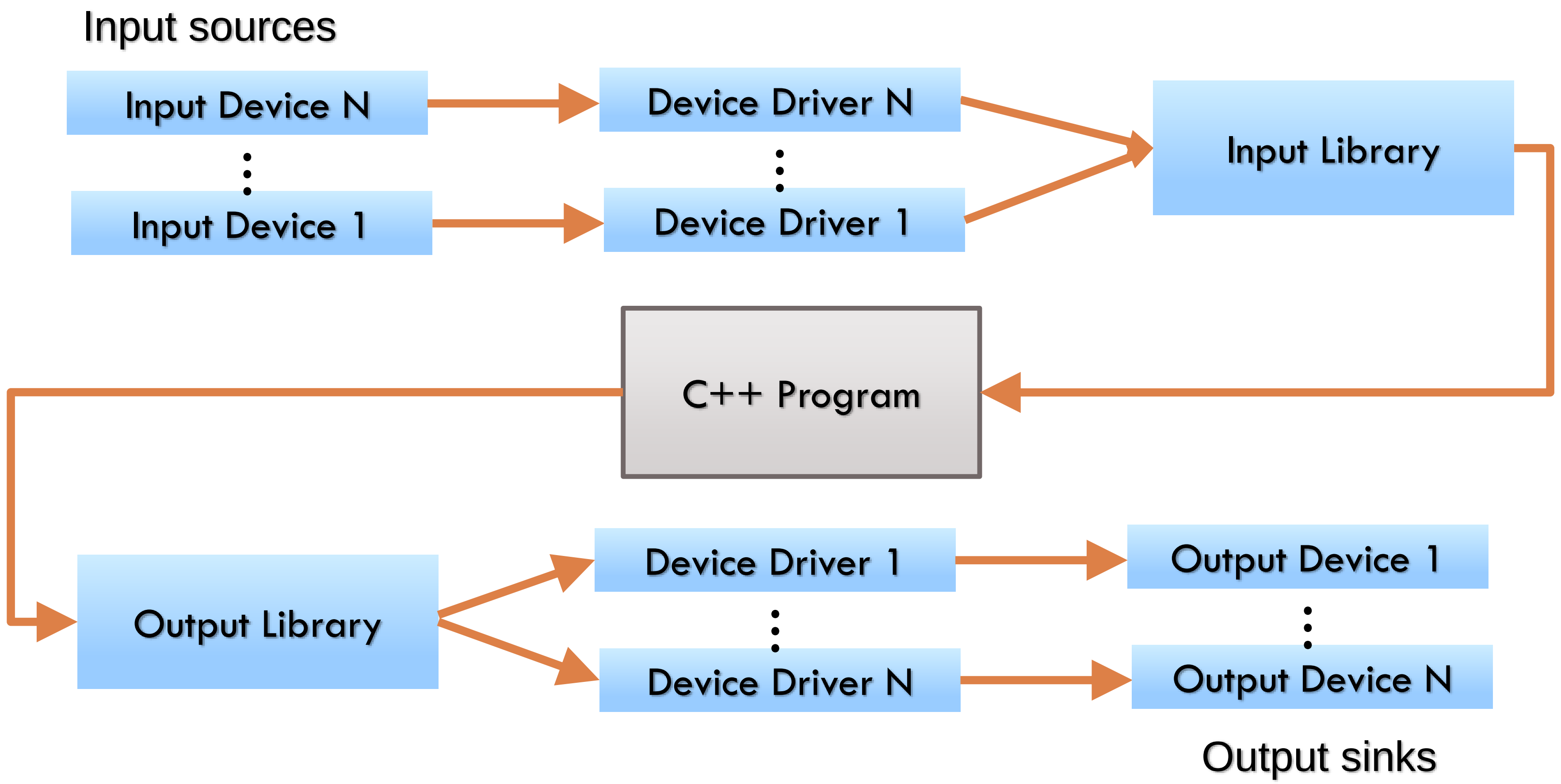
Binary & Unformatted File I/O

by Prasanna Ghali

Topics

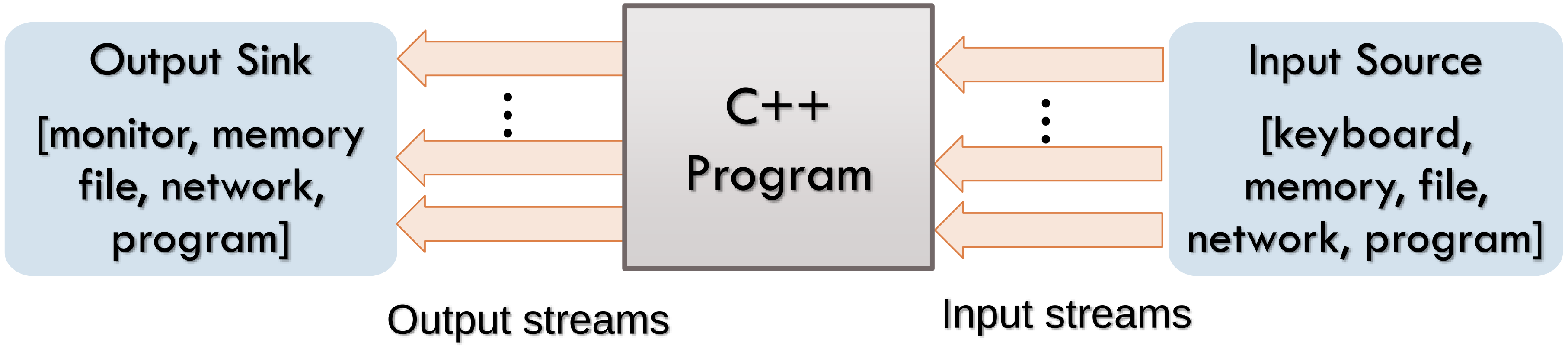
- Binary I/O
- Unformatted (character) I/O
- Stringstreams

Input and Output



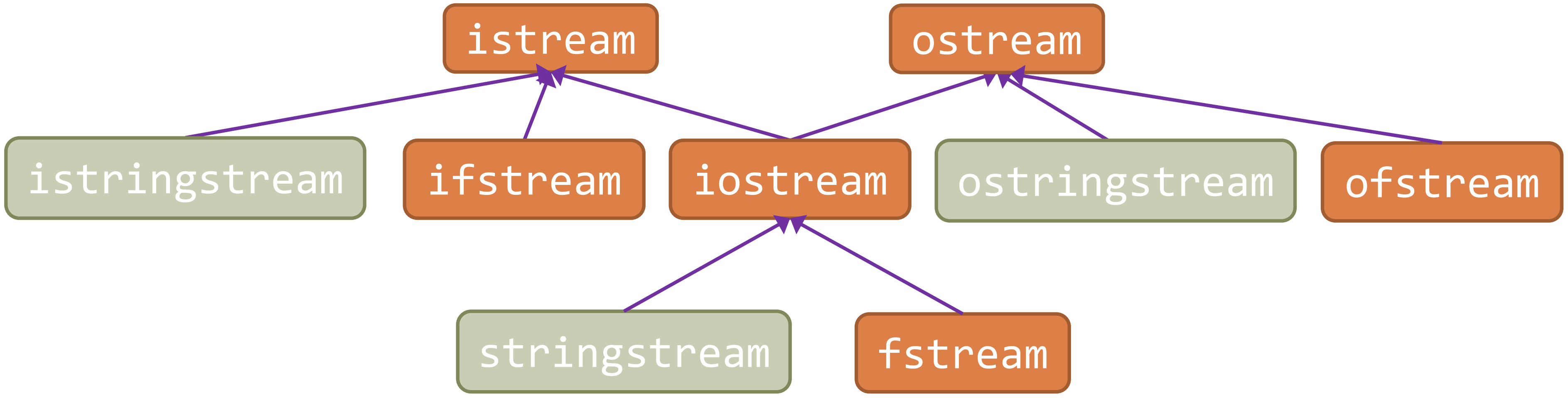
Stream Model

- *Stream* is abstraction for sequence of bytes consumed by program as input and generated by program as output


Output Sink
[monitor, memory file, network, program]
C++ Program
Input Source
[keyboard, memory, file, network, program]
Output streams
Input streams

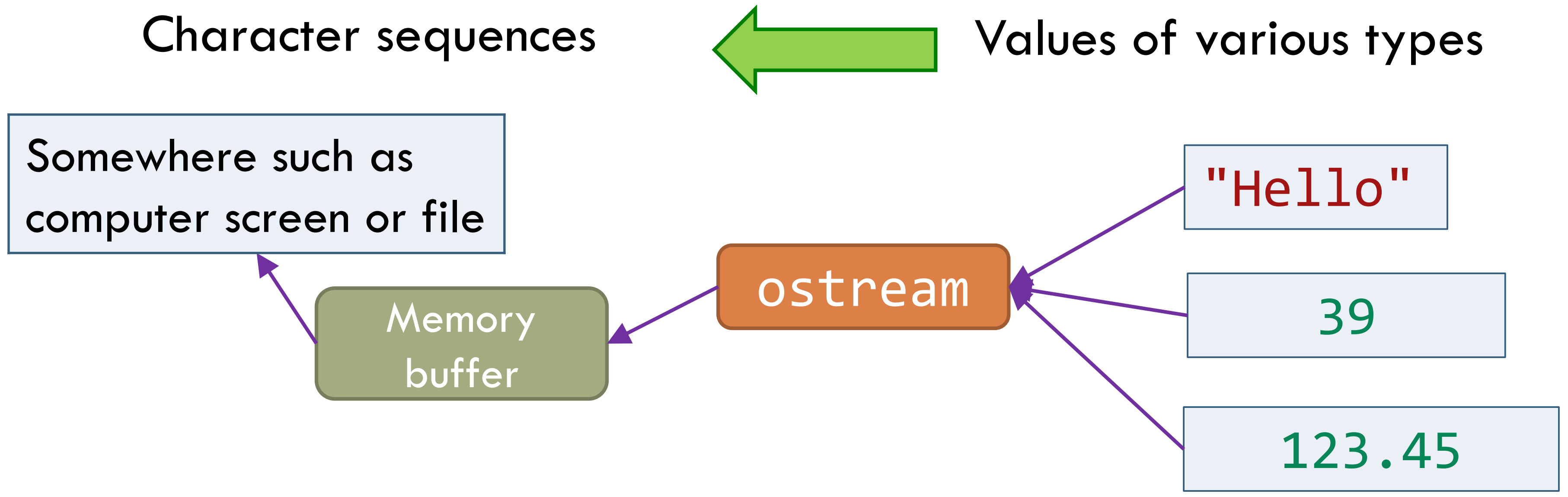
C++ I/O Streams Hierarchy

- Our job then becomes:
  - To set up I/O streams to appropriate data sources and destinations
  - To read from and write to those streams



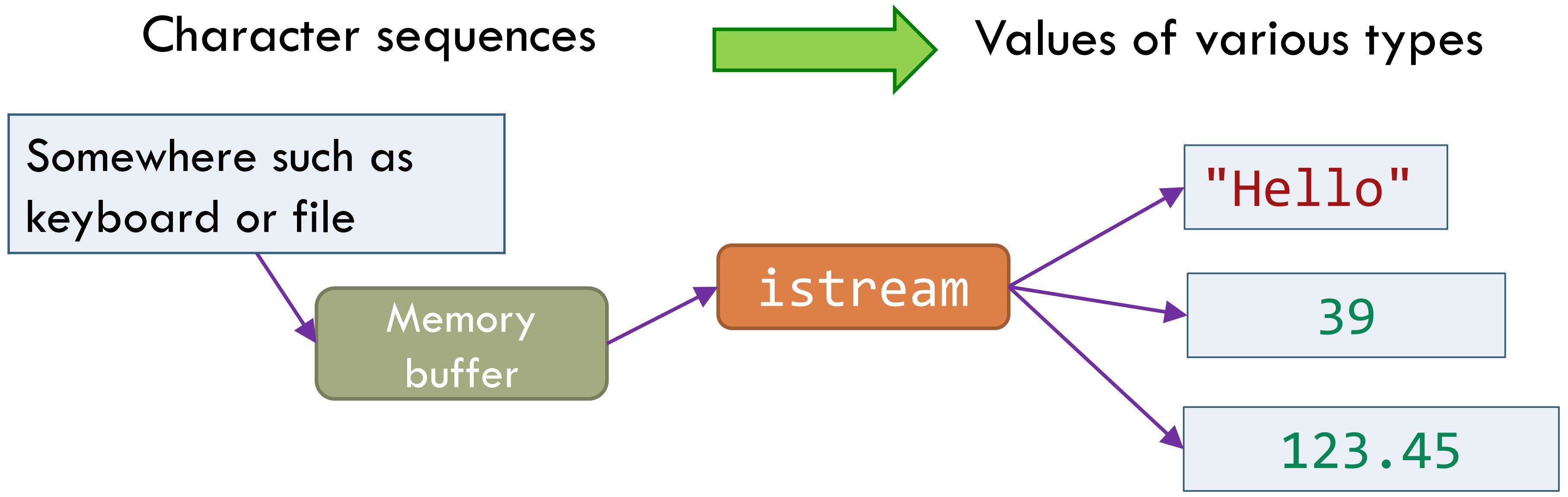
std::ostream

- std::ostream [defined in <ostream>] is type that converts objects into stream [i.e., sequence] of characters [i.e., bytes]
- std::cout [defined in <iostream>] is global variable of type std::ostream that exclusively writes to output stream stdout



std::istream

- std::istream [defined in <istream>] is type that converts stream [i.e., sequence] of characters [i.e., bytes] to typed objects
- std::cin [defined in <iostream>] is global variable of type std::istream that exclusively reads from input stream stdin



Formatted I/O

- By *default*, `iostream`s deal with characters and perform *formatted I/O* to convert collections of characters into values of specific types
  - `istream` reads sequence of characters and turns it into object of desired type
  - `ostream` takes object of specified type and transforms it into sequence of characters which it writes out
- This is I/O that we're familiar with

Formatted Output

Class `std::ostream` provides member function *overloads* of binary left shift operator for built-in types [`int`, `long`, `float`, `double`, …]. Equivalent to: `(std::cout).operator<<(3+7);`

```
#include <iostream>

int main() {
   std::cout << 3+7;
   std::cout << "Hello World\n";
}
```

Class `std::ostream` provides non-member function *overloads* of binary left shift operator for inserting characters [`char`, `unsigned char`, `char const*`, …]. Equivalent to: `std::operator<<(std::cout, "Hello World\n");`

Formatted Input (1/2)

Global variable of type std::istream instantiated at program startup to write characters to standard stream stdin

```
#include <iostream>

int main() {
  std::cout << "Enter your first name: ";
  char name[81];
  std::cin >> name;
  std::cout << "Hello " << name << '\n';
}
```

Class std::istream provides non-member function *overloads* of binary right shift operator for extracting characters [char, unsigned char, char const*, …] that is equivalent to:

std::operator>>(std::cin, name);

Formatted Input (2/2)

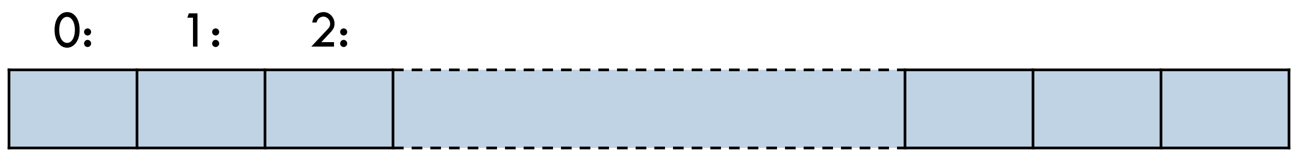
```
#include <iostream>

int main() {
  std::cout << "Enter your first name and age: ";
  char name[81];
  std::cin >> name;
  int age;
  std::cin >> age;
  std::cout << "Hello " << name << " age " << age << "\n";
}
```

Class `std::istream` provides member function overloads of binary right shift operator for built-in types [`int`, `long`, `float`, `double`, …] that is equivalent to: `(std::cin).operator>>(age);`

Files

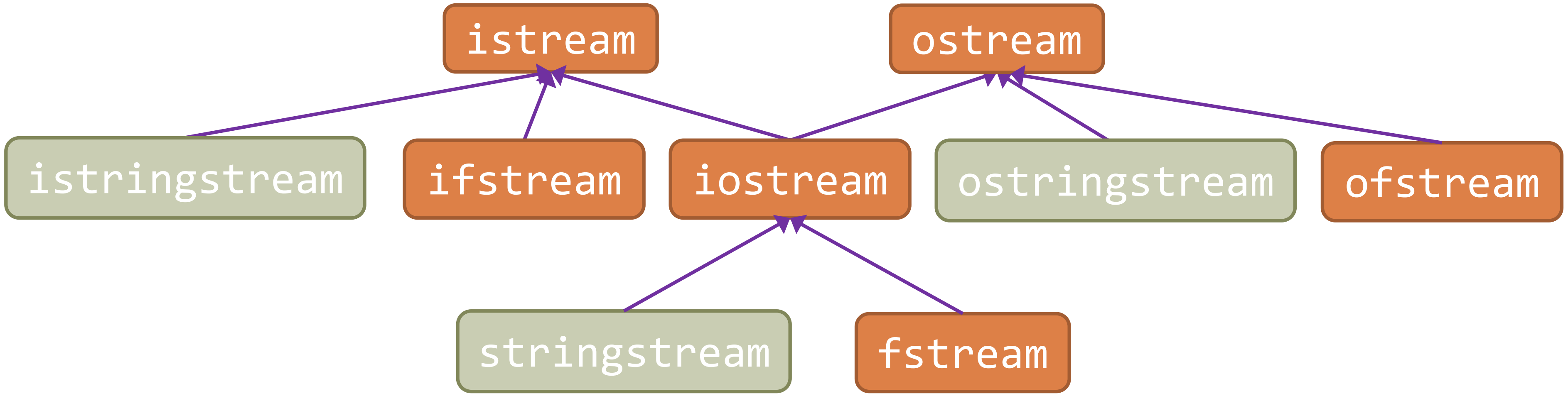
- At most basic level, file is simply sequence of bytes numbered from 0 upward



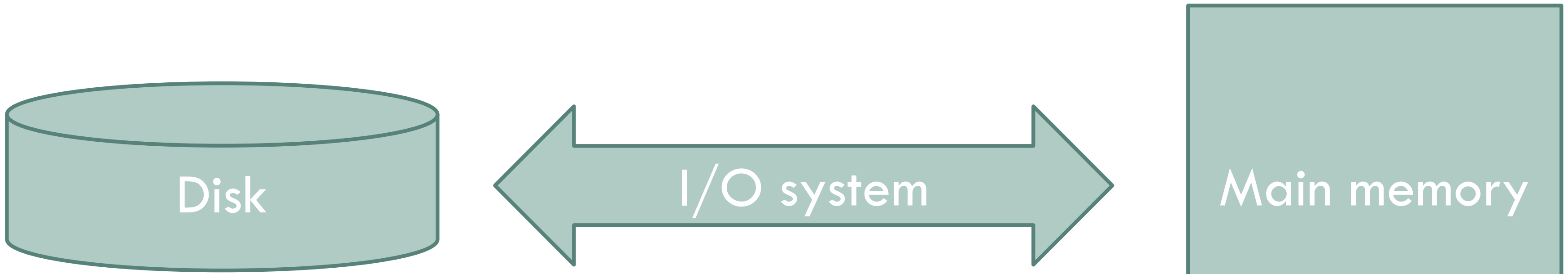
- Every file has a *format* [text or binary]
  - File's format serves same role as types serve for objects in memory

C++ I/O Streams Hierarchy

- `std::istream` connects [source] input device, file, or `std::string` to [destination] program
- `std::ostream` connects [source] program to [destination] output device, file, or `std::string`



Files and C++ I/O Streams



Files
(sequences of bytes)

`iostream`s

Objects
(of various types)

Formatted File I/O

- Just like `iostreams`, by *default*, file streams perform *formatted I/O* to convert collections of characters into values of specific types
  - `ofstream` takes object of specified type and transforms it into sequence of characters which it writes out to file
  - `ifstream` reads sequence of characters from file and turns it into object of desired type

Opening a File Stream (1/2)

- File stream can be opened either by constructor or by an `open()` call:

Opening files with file stream	
`std::fstream fs;`	Make a file stream variable for opening later
`fs.open(s, m);`	Open a file called `s` [C-style string] with mode `m` and have variable [defined in previous row] `fs` refer to it
`std::fstream fs(s, m);`	Open a file called `s` [C-style string] with mode `m` and make a file stream `fs` refer to it
`fs.is_open();`	Is file referenced by file stream `fs` open?
`fs.close();`	Close file referenced by file stream `fs`

Opening a File Stream (1/2)

- You can open a file in one of several modes:

Opening files with file stream	
`std::ios_base::in`	Open file for reading
`std::ios_base::out`	Open file for writing
`std::ios_base::app`	Open file for appending [i.e., add from end of the file]
`std::ios_base::binary`	Open file so that operations are performed in binary [as opposed to text]
`std::ios_base::ate`	"at end [of file]" [open and seek to the end]
`std::ios_base::trunc`	Truncate file to zero length

Formatted File I/O

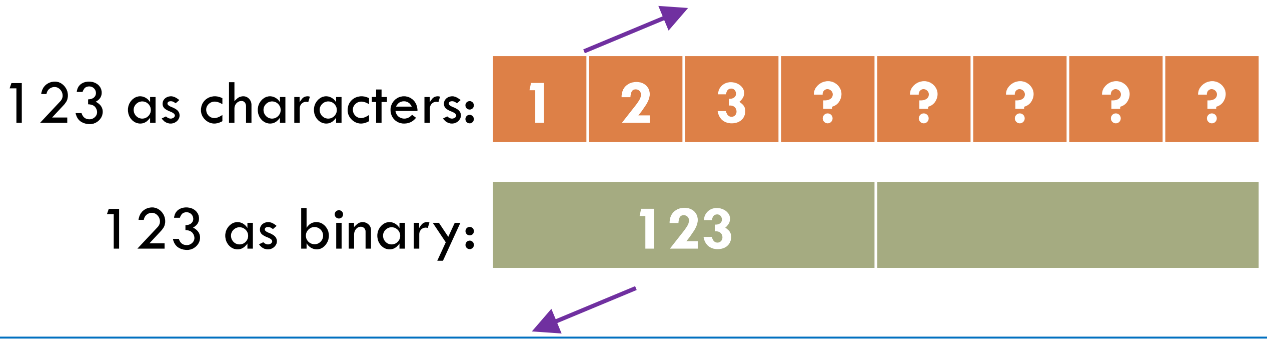
- Because of streams inheritance hierarchy, anything you could do to output stream `stdout` and input stream `stdin`, you can do to files too …
- See *formatted-fileio.cpp*

Representation of Values in Memory (1/3)

- In memory, number 123 can be represented as string value or an integer value

```
std::string s{"123"};
int n = 123;
```

String 123 represented with individual characters in ASCII: `0x310x320x33`

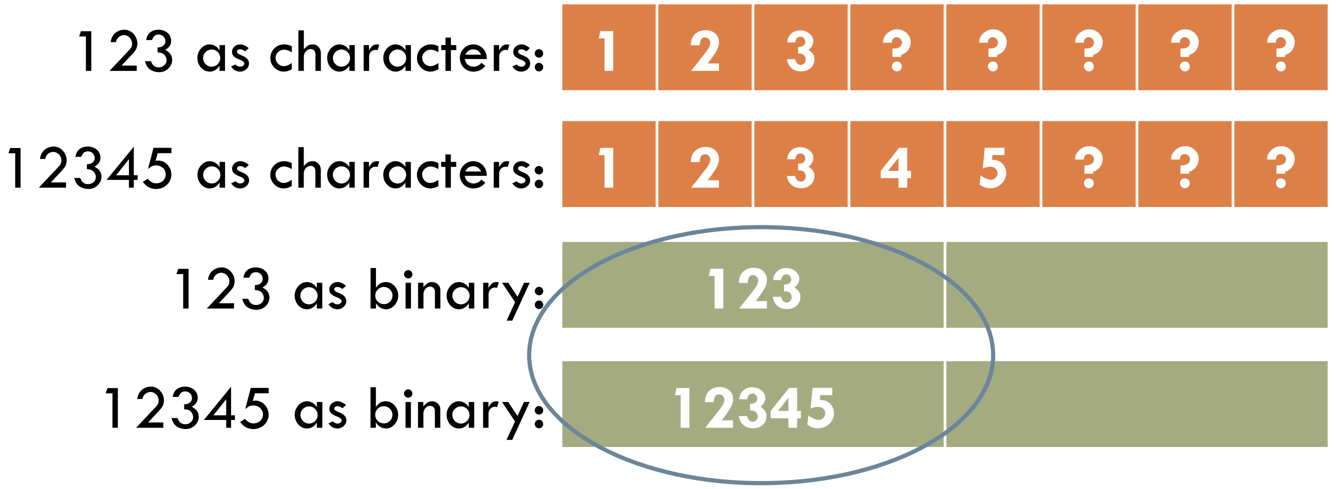


Number 123 represented in two's complementary form as `0x0000007b`

Representation of Values in Memory (2/3)

- What about number 12345?

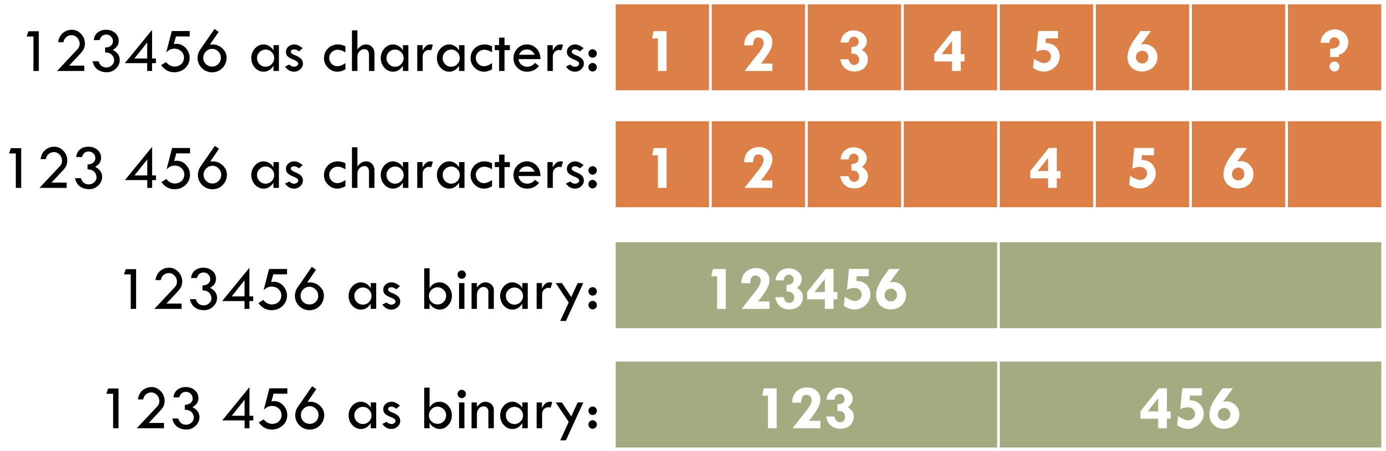
```
std::string s{"12345"};
int n = 12345;
```



Unlike numbers represented as strings, all `int` values stored in memory with 4 bytes

Representation of Values in Memory (3/3)

- Since `int`s are fixed-sized, numbers represented as `int`s don’t need to be separated
- On other hand, numbers represented as strings need to be separated by whitespace character



Binary File I/O (1/2)

- Possible to request `istream` and `ostream` to simply copy bytes to and from files by opening files with mode `ios::binary`
  - We use `binary` mode to tell stream not to try anything clever with bytes
- See *binary-fileio.cpp* ...

Binary File I/O (2/2)

- Binary I/O is messy, somewhat complicated, and error prone
- However, sometimes, we must use binary I/O simply because many information types don't have reasonable character representations: image files, audio files, …

Character vs. Binary Streams

Character streams	Binary streams
Characters represented as bytes	Binary stream is anything that is not character stream: groups of bytes might represent other types of data, such as integers and floating-point numbers
Sequence of characters divided into lines	
Each line consists of zero or more characters followed by newline character	Non-portable between platforms because of little- and big-endianness of processors
Newline character Windows: `'\x0d''\x0a'` UNIX & Mac OS: `'\x0a'`	

Positioning in Files (1/2)

- Easiest and least error-prone way is to just read and write files from beginning to end
- However, if you must, you can use positioning to select specific place in file for reading or writing
- File open for reading has “read/get position”
- File open for writing has “write/put position”
- See *file-position.cpp* …

Positioning in Files (2/2)

Class	Member Functions	Meaning
`basic_istream<>`	`tellg()`	Returns read position
	`seekg(pos)`	Sets read position as absolute value
	`seekg(offset,pos)`	Sets read position as relative value
`basic_ostream<>`	`tellp()`	Returns write position
	`seekp(pos)`	Sets write position as absolute value
	`seekp(offset,pos)`	Sets write position as relative value

Constant	Meaning
`ios::beg`	Position is relative to the beginning [“beginning”]
`ios::cur`	Position is relative to the current position [“current”]
`ios::end`	Position is relative to the end [“end”]

Reading Raw Characters (1/3)

- Input and output operators [<< and >>] format data they read or write
  - Input operator ignores whitespace characters
  - Output operators can apply precision, padding, …
- Sometimes we need to read individual characters including whitespace characters …

```
1 + 4 * x<=y / z* 5
```

- Instead, we could write:

Reading Raw Characters (2/3)

```
// read tokens ...
for (char ch; std::cin.get(ch); ) {
  if (std::isspace(ch)) { // ch is whitespace
    // do nothing [i.e., skip whitespace]
  }
  if (std::isdigit(ch)) {
    // read this digit and subsequent ones as number
  } else if (std::isalpha(ch)) {
    // read this Latin character and subsequent ones as identifier
  } else {
    // deal with operators
  }
}
```

```
// copies all characters including whitespace from
// standard input stream to standard output stream
char ch;
while (std::cin.get(ch)) {
  std::cout.put(ch);
}
```

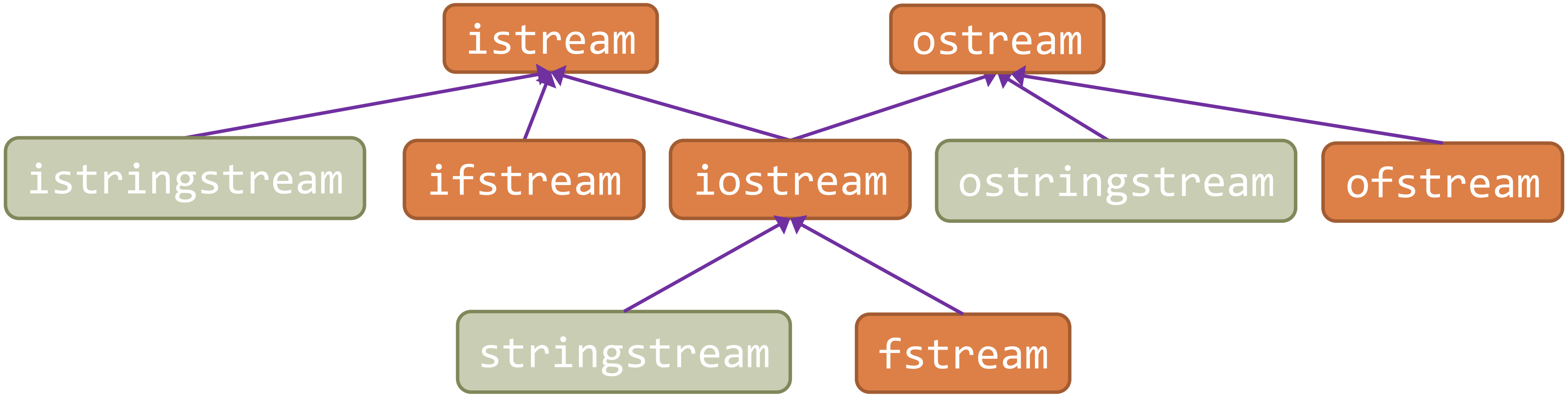
Reading Raw Characters (3/3)

- The library provides set of low-level operations that support unformatted I/O allowing us to deal with a stream as sequence of uninterpreted bytes

Single-Byte Low-Level I/O Operations	
`is.get(ch)`	Put next byte from `istream` `is` in character `ch`; returns `is`
`os.put(ch)`	Put character `ch` onto `ostream` `os`; returns `os`.
`is.get()`	Returns next byte from `is` as an `int`
`is.putback(ch)`	Put character `ch` back on `is`; returns `is`
`is.unget()`	Move `is` back one byte; returns `is`
`is.peek()`	Returns next byte as an `int` but doesn't remove it

ofstream *is-a* ostream (1/2)

- Anywhere you can use `ofstream` you can use `ostream`



ofstream *is-a* ostream (2/2)

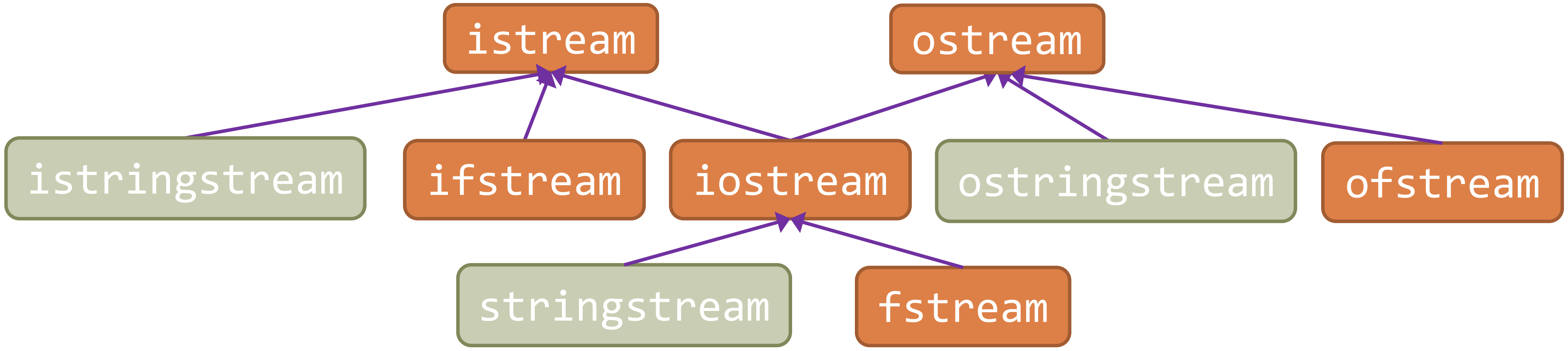
```
template <typename T>
std::ostream& operator<<(std::ostream& os,
                         std::complex<T> const& rhs) {
  os << "(" << rhs.real() << ", " << rhs.imag() << ")";
  return os;
}

std::complex<double> cd{1.1, 2.2};
std::cout << cd << "\n";

std::ofstream ofs{"file.txt"};
ofs << cd << "\n";
ofs.close();
```

string Streams

- You can use string as source of istream or target for ostream
  - istringstream *is-a* istream that reads from string
  - ostringstream *is-a* ostream that stores characters written to it in a string
  - stringstream is *adapter class* that allows you to access strings as streams



string Streams Uses (1/2)

- istringstream is useful for extracting numeric values from string

```
std::string date{"March 22, 2021"};
std::string month = ???
int day = ???
int year = ???
```

string Streams Uses (2/2)

- istringstream is useful for extracting numeric values from string
- Conversely, ostringstream can be useful for formatting output for system that requires string argument

```
std::string month {"March"};
int day {2};
int year {2021};
std::string date = ???;
```

string Streams Example (1/2)

```cpp
struct Date {
  std::string month;
  int day, year;
};

Date str_to_date(std::string const& str) {
  std::istringstream iss{str};
  Date d;
  std::string comma;
  iss >> d.month >> d.day >> comma >> d.year;
  return d;
}

Date today = str_to_date("March 1, 2021");
std::cout << today.month << " " << std::setw(2)
          << std::setfill('0') << today.day
          << ", " << today.year << "\n";
```

string Streams Example (2/2)

```
struct Date {
  std::string month;
  int day, year;
};

std::string date_to_str(Date const& d) {
  std::ostringstream oss;
  oss << d.month << " " << std::setw(2) << std::setfill('0')
      << d.day << ", " << d.year;
  return oss.str();
}

Date today = str_to_date("March 1, 2021");
// write to standard output stream: March 01, 2021
std::cout << date_to_str(today) << "\n";
```

std::strings: Numeric Conversions

Function	Effect
`stoi(str, idx=nullptr, base=10)`	Converts *str* to an int
`stol(str, idx=nullptr, base=10)`	Converts *str* to a long
`stoul(str, idx=nullptr, base=10)`	Converts *str* to an unsigned long
`stoll(str, idx=nullptr, base=10)`	Converts *str* to a long long
`stoull(str, idx=nullptr, base=10)`	Converts *str* to an unsigned long long
`stof(str, idx=nullptr)`	Converts *str* to a float
`stod(str, idx=nullptr)`	Converts *str* to a double
`stold(str, idx=nullptr)`	Converts *str* to a long double
`to_string(val)`	Converts *val* to a std::string

Example: Numeric Conversions (1/2)

Function	Effect
`stoi(str, idx=nullptr, base=10)`	Converts *str* to an `int`

```
int string_to_int(std::string const& s) {
  std::istringstream iss{s};
  int ival;
  iss >> ival;
  return ival;
}
```

Example: Numeric Conversions (2/2)

Function	Effect
`to_string(val)`	Converts *val* to a `std::string`

```
std::string int_to_string(int val) {
  std::ostringstream oss;
  oss << val;
  return oss.str();
}
```