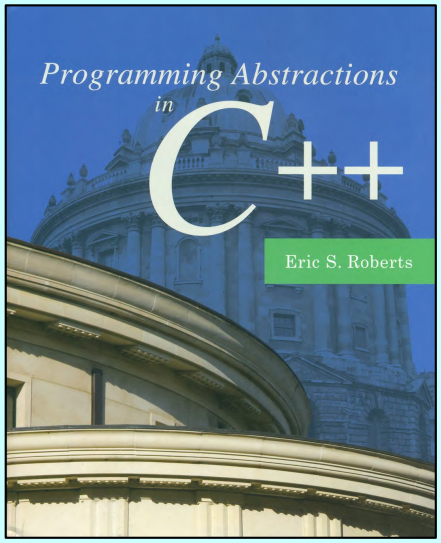
CHAPTER 2

Functions and Libraries

Your library is your paradise.
—Desiderius Erasmus, *Fisher's Study at Rotterdam,* 1524

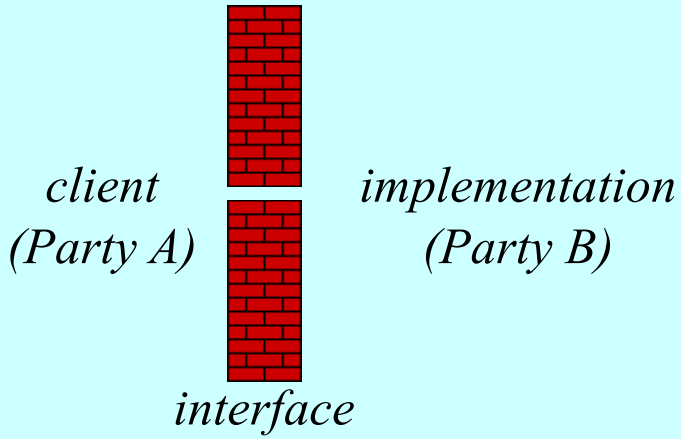


Libraries

- Modern programming depends on the use of ***libraries***. When you create a typical application, you write only a tiny fraction of the code.
- In computer science, a ***library*** is a collection of non-volatile resources used by computer programs, often to develop software.
- A library is also a collection of ***implementations*** of behavior, written in terms of a language, that has a well-defined ***interface*** by which the behavior is invoked.
- If you want to become an effective programmer, you need to spend at least as much time learning about the libraries as you do learning the language itself.

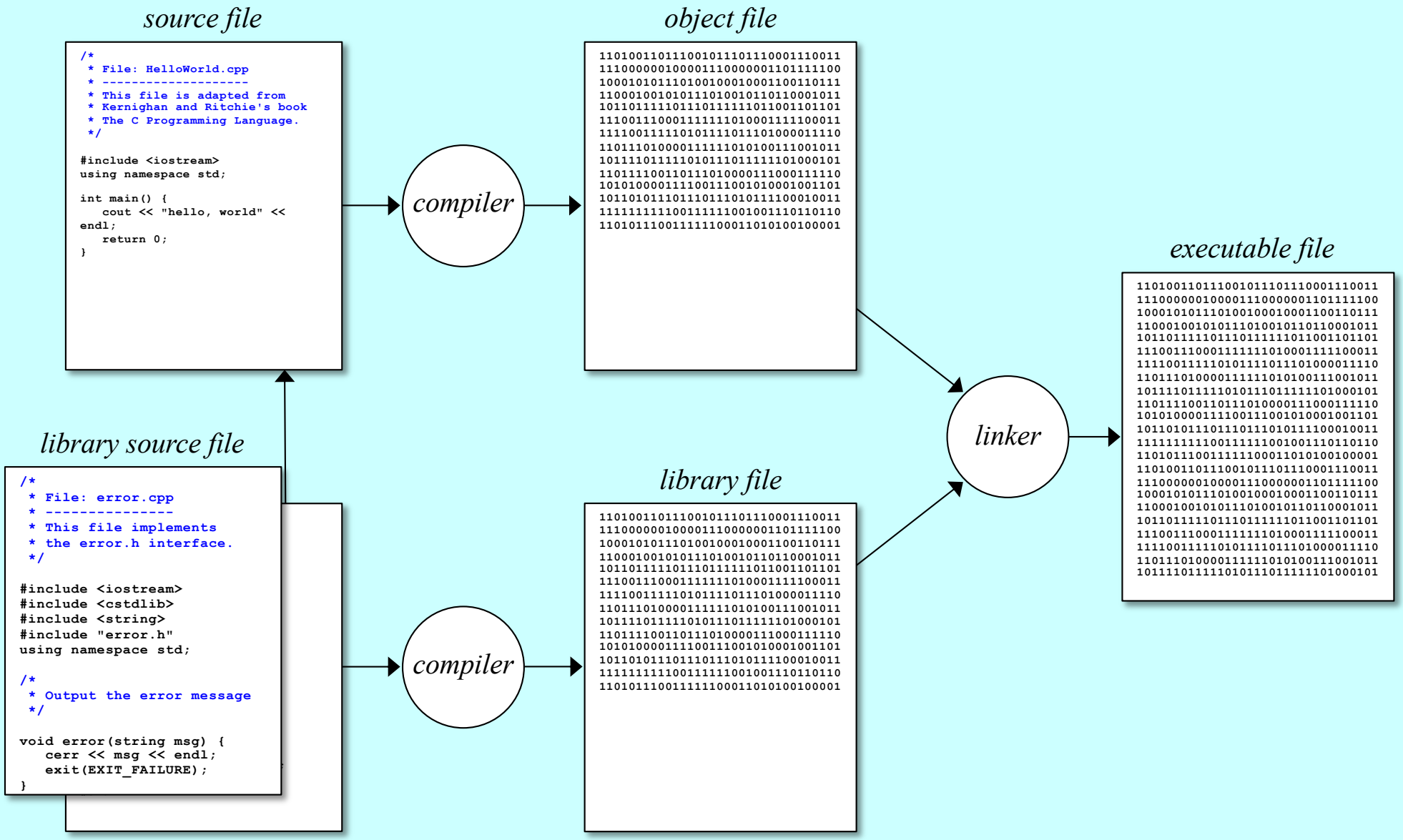
Libraries

- Libraries can be viewed from two perspectives. Code that uses a library is called a ***client***. The code for the library itself is called the ***implementation***.
- The point at which the client and the implementation meet is called the ***interface***, which serves as both a barrier and a communication channel:
  - Sharing and reuse
  - Abstraction and hiding



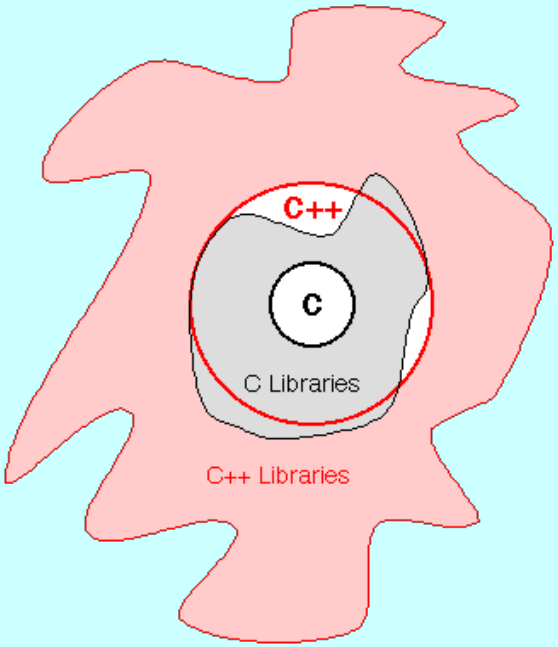
Pyramus and Thisbe

The Compilation Process



Relationship Between C and C++

- One of the fundamental design principles of C++ was that it would contain C as a subset, which is also part of the reason behind the success of C++. This design strategy makes it possible to convert applications from C to C++ incrementally.



- The downside of this strategy is that the design of C++ is less consistent and integrated than it might otherwise be.

Introduction to the C++ Standard Libraries

- A collection of ***classes*** and ***functions***, which are written in the core language and part of the C++ ISO Standard itself. Features of the C++ Standard Library are declared within the **`std`** ***namespace***
  - Containers: vector, queue, stack, map, set, etc.
  - General: algorithm, functional, iterator, memory, etc.
  - Strings
  - Streams and Input/Output: iostream, fstream, sstream, etc.
  - Localization
  - Language support
  - Thread support library
  - Numerics library
  - C standard library: cmath, cctype, cstring, cstdio, cstdlib, etc.

Useful Functions in the `<cmath>` Library

`abs``(`*x*`)`	Returns the absolute value of *x*.
`sqrt``(`*x*`)`	Returns the square root of *x*.
`floor``(`*x*`)`	Returns the largest integer less than or equal to *x*.
`ceil``(`*x*`)`	Returns the smallest integer greater than or equal to *x*.
`exp``(`*x*`)`	Returns the exponential function of *x* (e^x).
`log``(`*x*`)`	Returns the natural logarithm (base e) of *x*.
`log10``(`*x*`)`	Returns the common logarithm (base 10) of *x*.
`pow``(`*x*`, `*y*`)`	Returns x^y.
`cos``(`*theta*`)`	Returns the trigonometric cosine of the radian angle *theta*.
`sin``(`*theta*`)`	Returns the sine of the radian angle *theta*.
`tan``(`*theta*`)`	Returns the tangent of the radian angle *theta*.
`atan``(`*x*`)`	Returns the principal arctangent of *x*.
`atan2``(`*y*`, `*x*`)`	Returns the arctangent of *y* divided by *x*.

http://www.cplusplus.com/reference/

http://www.cppreference.com/

Introduction to the Stanford Libraries

- *Programming Abstractions in C++* comes with a variety of library packages developed at Stanford to make it easier for students to learn the underlying concepts. These libraries include, but are not limited to, the following:

`console.h`	Redirects standard input and output to a console window.
`error.h`	Supports error reporting and recovery.
`gwindow.h`	Implements a simple, portable, object-oriented graphical model.
`simpio.h`	Suoports improved error-checking for input operations.
`strlib.h`	Extends the set of string operations.

- You can learn more about these libraries from the tables of functions in the textbook, from the web-based documentation, or by reading the interfaces.
- The latest code and documentation are available here: https://web.stanford.edu/class/cs106b/library/documentation/

From Client to Implementer

- Libraries can be viewed from two perspectives. Code that uses a library is called a *client*. The code for the library itself is called the *implementation*.
- Consequently, a programmer who calls functions provided by a library is also called a *client* of that library. A programmer who implements a library is called an *implementer*.
- As we go through the chapters in this book, you will have a chance to look at several libraries (e.g., the collection classes) from both of these perspectives, first as a client and later as an implementer.
- We will now switch from a client's perspective to an implementer's.

Creating Library Interfaces

- In C++, libraries are made available to clients through the use of an interface file that has the suffix **`.h`**, which designates a ***header file***, as illustrated by the following **`error.h`** file:

```
/*
 * File: error.h
 * -------------
 * This file defines a simple function for reporting errors.
 */

#ifndef _error_h
#define _error_h
// #include <string>
// using namespace std;

/*
 * Function: error
 * Usage: error(msg);
 * ------------------
 * Writes the string msg to the cerr stream and then exits the program
 * with a standard status code indicating failure.
 */

void error(std::string msg);

#endif
```

Interfaces require standardized preprocessor directive definitions (called boilerplate) to ensure that the interface file is read only once during a compilation.

`#include <string>` *was omitted in the textbook, but clearly string is used here. Why? Because error.h is included in error.cpp which includes* **`<string>`**. *When error.cpp is compiled, string will be available for use.*

`using namespace` *should generally NOT be used in a header file, because it will probably be included by many other source files. Therefore all references to standard libraries in header files must include the* **`std::`** *marker.*

Implementing Library Interfaces

- In C++, the header file contains only the prototypes of the exported functions. The implementations of those functions appear in the corresponding **.cpp** file:

```
/*
 * File: error.cpp
 * ---------------
 * This file implements the error.h interface.
 */

#include <iostream>          // cerr, endl
#include <cstdlib>           // exit, EXIT_FAILURE
#include <string>            // string
#include "error.h"
using namespace std;

/*
 * This function writes out the error message to the cerr stream and
 * then exits the program. The EXIT_FAILURE constant is defined in
 * <cstdlib> to represent a standard failure code.
 */

void error(string msg) {
   cerr << msg << endl;
   exit(EXIT_FAILURE);
}
```

Implementation files typically include their own interface. That is also why error.h does not include **<string>** *but still works.*

using namespace *should always come after the inclusion of all the libraries and header files, because it affects all the code after it.*

Exporting types

- Let's create an interface that exports one of the ***enumerated types***, such as the Direction type used to encode the four standard compass points:

```
/*
 * File: direction.h
 * -----------------
 * This interface exports an enumerated type called Direction whose
 * elements are the four compass points: NORTH, EAST, SOUTH, and WEST.
 */

#ifndef _direction_h
#define _direction_h

#include <string>

/*
 * Type: Direction
 * ---------------
 * This enumerated type is used to represent the four compass directions.
 */

enum Direction { NORTH, EAST, SOUTH, WEST };
```

```
/*
 * Function: leftFrom
 * Usage: Direction newdir = leftFrom(dir);
 * ----------------------------------------
 * Returns the direction that is to the left of the argument.
 * For example, leftFrom(NORTH) returns WEST.
 */

Direction leftFrom(Direction dir);

/*
 * Function: rightFrom
 * Usage: Direction newdir = rightFrom(dir);
 * -----------------------------------------
 * Returns the direction that is to the right of the argument.
 * For example, rightFrom(NORTH) returns EAST.
 */

Direction rightFrom(Direction dir);

/*
 * Function: directionToString
 * Usage: string str = directionToString(dir);
 * -------------------------------------------
 * Returns the name of the direction as a string.
 */

std::string directionToString(Direction dir);
```

```
/*
 * Operator: <<
 * Usage: cout << dir;
 * -------------------
 * Overloads the << operator so that it is able to display Direction values.
 */

std::ostream & operator<<(std::ostream & os, Direction dir);

/*
 * Operator: ++
 * Usage: ++dir
 * ------------
 * Overloads the prefix version of the ++ operator to work with Direction
 * values.
 */

Direction operator++(Direction & dir);

/*
 * Operator: ++
 * Usage: dir++
 * ------------
 * Overloads the suffix version of the ++ operator to work with Direction
 * values, to support, e.g., the idiom
 *
 *    for (Direction dir = NORTH; dir <= WEST; dir++) . . .
 */

Direction operator++(Direction & dir, int);

#endif
```

Exporting types

- Implementation of the Direction library

```
/*
 * File: direction.cpp
 * -------------------
 * This file implements the direction.h interface.
 */

#include <string>
#include "direction.h"
using namespace std;

/*
 * Implementation notes: leftFrom, rightFrom
 * -----------------------------------------
 * These functions use the remainder operator to cycle through the
 * internal values of the enumeration type.  Note that the leftFrom
 * function cannot subtract 1 from the direction because the result
 * might then be negative; adding 3 achieves the same effect but
 * ensures that the values remain positive.
 */

Direction leftFrom(Direction dir) {
   return Direction((dir + 3) % 4);
}
Direction rightFrom(Direction dir) {
   return Direction((dir + 1) % 4);
}
```

```
/*
 * Implementation notes: directionToString
 * ---------------------------------------
 * Most C++ compilers require the default clause to make sure that this
 * function always returns a string, even if the direction is not one
 * of the legal values.
 */

string directionToString(Direction dir) {
   switch (dir) {
    case NORTH: return "NORTH";
    case EAST: return "EAST";
    case SOUTH: return "SOUTH";
    case WEST: return "WEST";
    default: return "???";
   }
}

/*
 * Implementation notes: <<
 * ------------------------
 * This operator must return the stream by reference after printing
 * the value.  The operator << returns this stream, so the function
 * can be implemented as a single line.
 */

std::ostream & operator<<(std::ostream & os, Direction dir) {
   return os << directionToString(dir);
}
```

```
/*
 * Implementation notes: ++
 * ------------------------
 * The int parameter in the signature for this operator is a marker used
 * by the C++ compiler to identify the suffix form of the operator.  Note
 * that the value after incrementing a variable containing WEST will be
 * out of the Direction range.  That fact will not cause a problem if
 * this operator is used only in the for loop idiom for which it is defined.
 */

Direction operator++(Direction & dir) {
   dir = Direction(dir + 1);
   return dir;
}

Direction operator++(Direction & dir, int) {
   Direction old = dir;
   dir = Direction(dir + 1);
   return old;
}
```

If you want **dir** *to go back to* **NORTH** *after* **WEST**, *how should you modify the code?*

Exporting Constants

```
/*
 * File: gmath.h
 * -------------
 * This file exports the constant PI along with a few degree-based
 * trigonometric functions, which are typically easier to use.
 */

#ifndef _gmath_h
#define _gmath_h

/* Constants */

extern const double PI;                    /* stant pi */

/*
 * Function: sinDegrees
 * Usage: double sine = sinDegrees(angle);
 * ---------------------------------------
 * Returns the trigonometric sine of angle expressed in degrees.
 */
```

In C++, constants written in this form are private to the source file that contains them and cannot be exported through an interface. To export the constant **PI**, *you need to add the keyword* **extern** *to both its definition and the prototype declaration in the interface.*

```
double sinDegrees(double angle);

/*
 * Function: cosDegrees
 * Usage: double cosine = cosDegrees(angle);
 * -----------------------------------------
 * Returns the trigonometric cosine of angle expressed in degrees.
 */

double cosDegrees(double angle);

/*
 * Function: toDegrees
 * Usage: double degrees = toDegrees(radians);
 * -------------------------------------------
 * Converts an angle from radians to degrees.
 */

double toDegrees(double radians);

/*
 * Function: toRadians
 * Usage: double radians = toRadians(degrees);
 * -------------------------------------------
 * Converts an angle from degrees to radians.
 */

double toRadians(double degrees);

#endif
```

Exporting Constants

```
/*
 * File: gmath.cpp
 * ---------------
 * This file implements the gmath.h interface.  In all cases, the
 * implementation for each function requires only one line of code,
 * which makes detailed documentation unnecessary.
 */

#include <cmath>
#include "gmath.h"

extern const double PI = 3.14159265358979323846;

double sinDegrees(double angle) {
   return sin(toRadians(angle));
}

double cosDegrees(double angle) {
   return cos(toRadians(angle));
}

double toDegrees(double radians) {
   return radians * 180 / PI;
}

double toRadians(double degrees) {
   return degrees * PI / 180;
}
```

In C++, constants written in this form are private to the source file that contains them and cannot be exported through an interface. To export the constant **`PI`**, *you need to add the keyword* **`extern`** *to both its definition and the prototype declaration in the interface.*

Principles of Interface Design

- ***Unified***. Every library should define a consistent abstraction with a clear unifying theme. If a function does not fit within that theme, it should not be part of the interface.
- ***Simple***. The interface design should simplify things for the client. To the extent that the underlying implementation is itself complex, the interface must seek to hide that complexity.
- ***Sufficient***. For clients to adopt a library, it must provide functions that meet their needs. If some critical operation is missing, clients may decide to abandon it and develop their own tools.
- ***General***. A well-designed library should be general enough to meet the needs of many different clients. A library that offers narrowly defined operations for one client is not nearly as useful as one that can be used in many different situations.
- ***Stable***. The functions defined in a class exported by a library should maintain precisely the same structure and effect, even as the library evolves. Making changes in the behavior of a library forces clients to change their programs, which reduces its utility.

Designing a Random Number Library

- Nondeterministic behavior turns out to be difficult to achieve on a computer. A computer executes its instructions in a precise, predictable way. If you give a computer program the same inputs, it will generate the same outputs every time, which is not what you want in a nondeterministic program.
- Given that true nondeterminism is so difficult to achieve in a computer, libraries such as the **`random.h`** interface described in this chapter must instead *simulate* rando… out a deterministic process that satisfies t…

  1. The values generated by that process … for human observers to predict.
  2. Those values should appear to be random, in the sense that they should pass statistical tests for randomness.

- Because the process is not truly random, the values generated by the **`random.h`** interface are said to be ***pseudorandom***.

Since C++11, there is a **`<random>`** *library in the C++ standard. But here we are implementing a simplified version using only* **`<cstdlib>`**.

Designing a Random Number Library

- The function **`rand`** from **`<cstdlib>`** takes no arguments and returns an integer between 0 and **`RAND_MAX`** randomly. It is not ***sufficient*** for the clients.
- Choosing the right set of functions
  - ✓ *Selecting a random integer in a specified range*
  - ✓ *Choosing a random real number in a specified range*
  - ✓ *Simulating a random event with a specific probability*
- The designers of the C++ library (and the earlier C libraries) decided that **`rand`** should return the same random sequence each time a program is run, so that it is possible to use **`rand`** in a deterministic way in order to support debugging.
  - ✓ *Set the initial value (**seed**) of the random number generating process, e.g., the time when the program is run*

The `random.h` Interface

```
/*
 * File: random.h
 * --------------
 * This file exports functions for generating pseudorandom numbers.
 */

#ifndef _random_h
#define _random_h

/*
 * Function: randomInteger
 * Usage: int n = randomInteger(low, high);
 * ----------------------------------------
 * Returns a random integer in the range low to high, inclusive.
 */

int randomInteger(int low, int high);
```

The `random.h` Interface

```
/*
 * Function: randomReal
 * Usage: double d = randomReal(low, high);
 * ----------------------------------------
 * Returns a random number in the half-open interval [low, high).
 * A half-open interval includes the first endpoint but not the
 * second.
 */

double randomReal(double low, double high);

/*
 * Function: randomChance
 * Usage: if (randomChance(p)) ...
 * -------------------------------
 * Returns true with the probability indicated by p.  The
 * argument p must be a floating-point number between 0 (never)
 * and 1 (always).
 */

bool randomChance(double p);
```

The `random.h` Interface

```
/*
 * Function: randomBool
 * Usage: if (randomBool()) ...
 * ----------------------------
 * Returns <code>true</code> with 50% probability.
 */
bool randomBool();

/*
 * Function: setRandomSeed
 * Usage: setRandomSeed(seed);
 * ---------------------------
 * Sets the internal random number seed to the specified value.
 * You can use this function to set a specific starting point
 * for the pseudorandom sequence or to ensure that program
 * behavior is repeatable during the debugging phase.
 */

void setRandomSeed(int seed);

#endif
```

random.cpp Implementation

```
/*
 * File: random.cpp
 * ----------------
 * This file implements the random.h interface.
 */

#include <cstdlib>
#include <cmath>
#include <ctime>
#include "random.h"
using namespace std;

/* Private function prototype */

void initRandomSeed();
```

The `random.cpp` Implementation

```
/*
 * Implementation notes: randomInteger
 * -----------------------------------
 * The code for randomInteger produces the number in four steps:
 *
 * 1. Generate a random real number d in the range [0 .. 1).
 * 2. Scale the number to the range [0 .. N).
 * 3. Translate the number so that the range starts at low.
 * 4. Truncate the result to the next lower integer.
 *
 * The implementation is complicated by the fact that both the
 * expression RAND_MAX + 1 and the expression high - low + 1 can
 * overflow the integer range.
 */

int randomInteger(int low, int high) {
   initRandomSeed();
   double d = rand() / (double(RAND_MAX) + 1);
   double s = d * (double(high) - low + 1);
   return int(floor(low + s));
}
```

The `random.cpp` Implementation

```
/*
 * Implementation notes: randomReal
 * --------------------------------
 * The code for randomReal is similar to that for randomInteger,
 * without the final conversion step.
 */

double randomReal(double low, double high) {
   initRandomSeed();
   double d = rand() / (double(RAND_MAX) + 1);
   double s = d * (high - low);
   return low + s;
}
```

The `random.cpp` Implementation

```
/*
 * Implementation notes: randomChance
 * ----------------------------------
 * The code for randomChance calls randomReal(0, 1) and then checks
 * whether the result is less than the requested probability.
 */

bool randomChance(double p) {
   initRandomSeed();
   return randomReal(0, 1) < p;
}

bool randomBool() {
    return randomChance(0.5);
}
```

The `random.cpp` Implementation

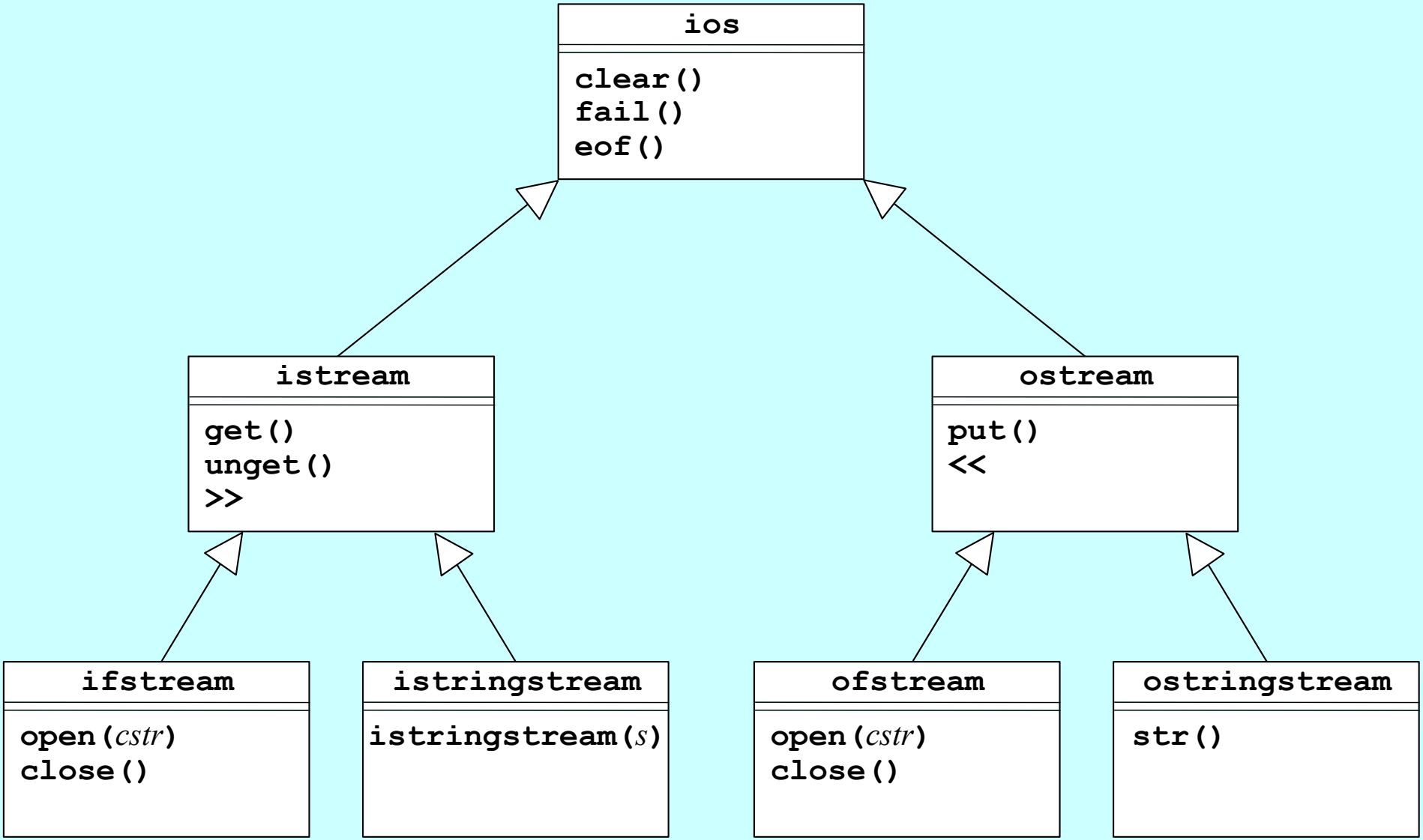
```
/*
 * Implementation notes: setRandomSeed
 * -----------------------------------
 * The setRandomSeed function simply forwards its argument to
 * srand.  The call to initRandomSeed is required to set the
 * initialized flag.
 */

void setRandomSeed(int seed) {
   initRandomSeed();
   srand(seed);
}
```

The `random.cpp` Implementation

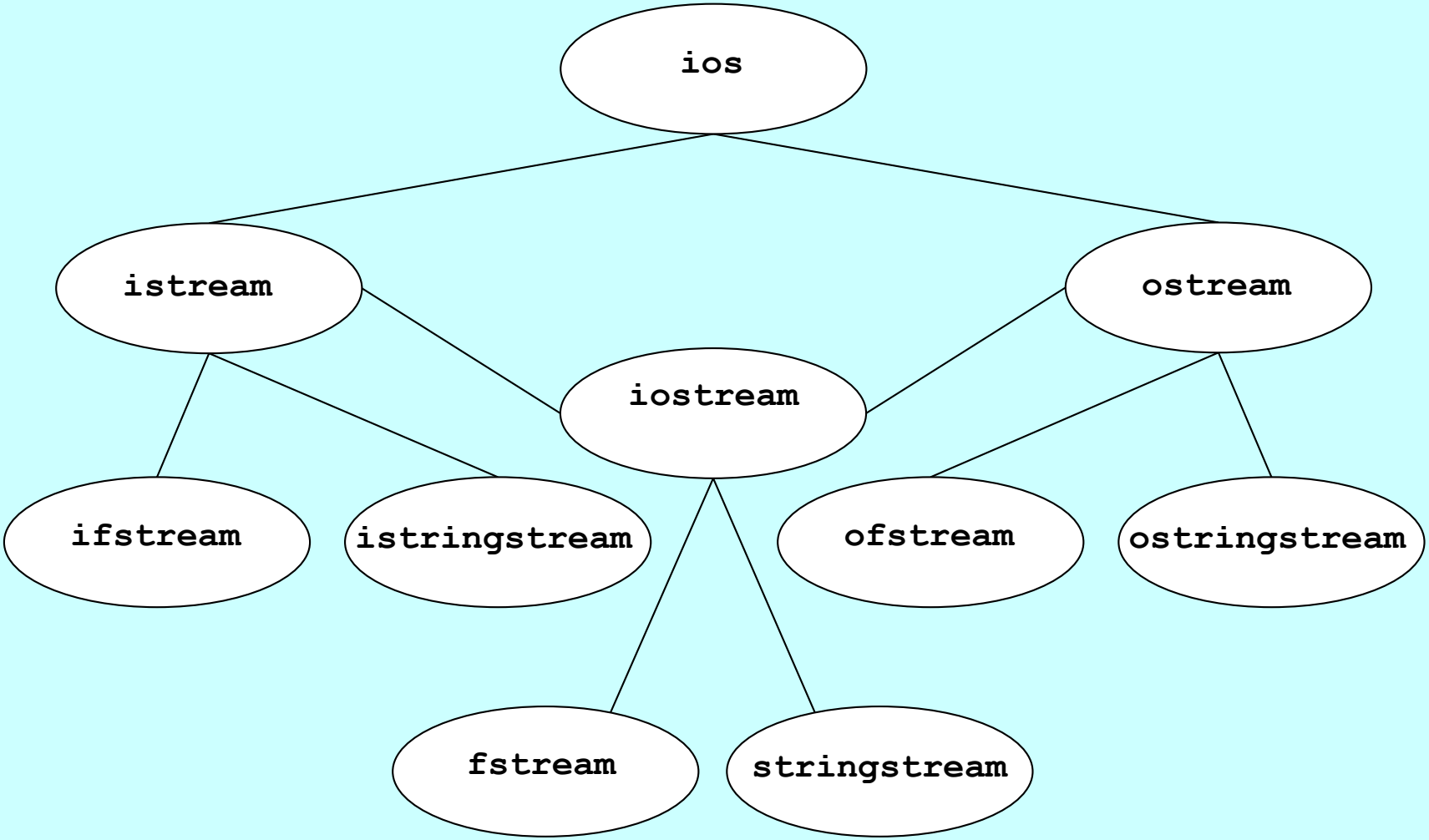
```
/*
 * Implementation notes: initRandomSeed
 * ------------------------------------
 * The initRandomSeed function declares a static variable that
 * keeps track of whether the seed has been initialized.  The
 * first time initRandomSeed is called, initialized is false,
 * so the seed is set to the current time.
 */

void initRandomSeed() {
   static bool initialized = false;
   if (!initialized) {
      srand(time(NULL));
      initialized = true;
   }
}
```

The lifetime of `static` *variables begins the first time the program flow encounters the declaration and it ends at program termination. The compiler allocates only one copy of* `initialized`*, which is initialized exactly once, and then shared by all calls to* `initRandomSeed`*. This ensures that the initialization step must be performed once and only once.*

Simplified View of the Stream Hierarchy

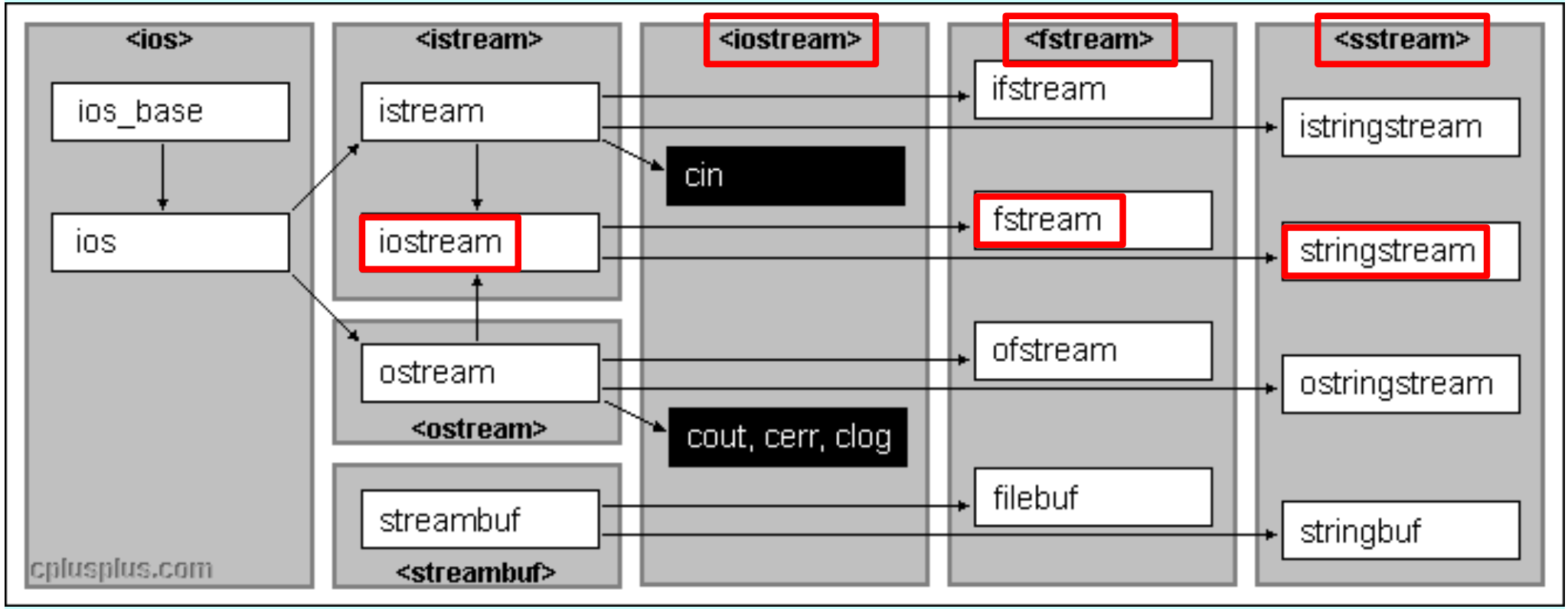


Selected Classes in the Stream Hierarchy



Library vs. Class Hierarchy

- Library vs. Class Hierarchy: Interestingly, the C++ stream libraries are not organized based on the class hierarchies.



- Organize libraries for the users' convenience, and design classes for the data integrity and implementation efficiency.

The End