Creating Library Interfaces

touching lives

- 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

/*
 * 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 definitions called boilerplate to ensure that the interface file is read only once during a compilation.

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>
#include <cstdlib>
#include <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.

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.
- *Flexible*. 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* randomness by carrying out a deterministic process that satisfies the following criteria:
  1. The values generated by that process should be difficult 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*.

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: 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
```

The library "random.h" and other Stanford C++ libraries used throughout this textbook are freely available as open-source at

http://cs.stanford.edu/people/eroberts/StanfordCPPLib/

The `random.cpp` Implementation

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

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

/* Private function prototype */

static 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;
}
```

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);
}
```

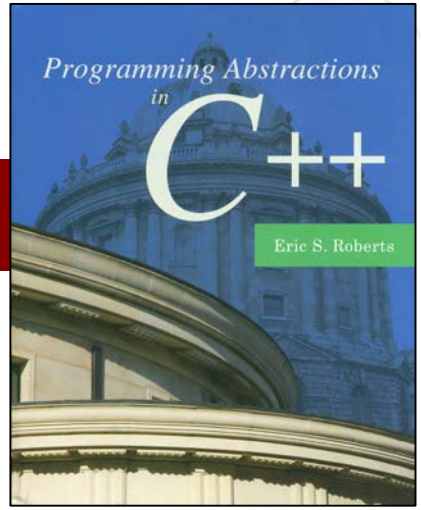
The random.cpp Implementation

```
/*
 * 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.
 */

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

CHAPTER 3

Strings



Whisper music on those strings.
—T. S. Eliot, *The Waste Land,* 1922

Using Strings as Abstract Values

- Ever since the very first program in the text, which displayed the message `"hello, world"` on the screen, you have been using strings to communicate with the user.
- Up to now, you might not have known how C++ represents strings inside the computer or how you might manipulate the characters that make up a string. At the same time, the fact that you don't know those things has not compromised your ability to use strings effectively because you have been able to think of strings holistically as if they were a primitive type.
- For most applications, the abstract view of strings you have held up to now is precisely the right one. On the inside, strings are surprisingly complicated objects whose details are better left hidden.
- C++ supports a high-level view of strings by making `string` a class whose methods hide the underlying complexity.

Calling String Methods

- Because `string` is a class, it is best to think of its methods in terms of sending a message to a particular object. The object to which a message is sent is called the *receiver,* and the general syntax for sending a message looks like this:

> *receiver*`.`*name*`(`*arguments*`);`

- For example, if you want to determine the length of a string `s`, you need to use the assignment statement

```
int len = s.length();
```

- Unlike most languages, C++ allows classes to redefine the meanings of the standard operators. As a result, several string operations, such as `+` for concatenation, are implemented as operators.

Common Methods in the `string` Class

`str.length()` Returns the number of characters in the string `str`.
`str.at(index)` Returns the character at position `index`; most clients use `str[index]` instead.
`str.substr(pos, len)` Returns the substring of `str` starting at `pos` and continuing for `len` characters.
`str.find(ch, pos)` Returns the first index ≥ `pos` containing `ch`, or `string::npos` if not found.
`str.find(text, pos)` Similar to the previous method, but with a string instead of a character.

Note: the first character in the string is at index 0.

Operators on the `string` Class

`str[i]` Returns the i^{th} character of `str`. Assigning to `str[i]` changes that character.
`s1 + s2` Returns a new string consisting of `s1` concatenated with `s2`.
`s1 = s2;` Copies the character string `s2` into `s1`.
`s1 += s2;` Appends `s2` to the end of `s1`.
`s1 == s2` (and similarly for <, <=, >, >=, and !=) Compares two strings lexicographically.

Characters

- Both C and C++ use ASCII (*American Standard Code for Information Interchange*) as their encoding for character representation. The data type `char` therefore fits in a single eight-bit byte.
- With only 256 possible characters, ASCII is inadequate to represent the many alphabets in use throughout the world. In most modern language, ASCII has been superseded by Unicode, which permits a much larger number of characters.
- Even though the weaknesses in the ASCII encoding were clear at the time C++ was designed, changing the definition of `char` was impossible given the decision to keep C as a subset.
- The C++ libraries define the type `wchar_t` to represent "wide characters" that allow for a larger range. The details of the `wchar_t` type are beyond the scope of this text.

Functions in the <cctype> Interface

`bool isdigit(char ch)` Determines if the specified character is a digit.
`bool isalpha(char ch)` Determines if the specified character is a letter.
`bool isalnum(char ch)` Determines if the specified character is a letter or a digit.
`bool islower(char ch)` Determines if the specified character is a lowercase letter.
`bool isupper(char ch)` Determines if the specified character is an uppercase letter.
`bool isspace(char ch)` Determines if the specified character is *whitespace* (spaces and tabs).
`char tolower(char ch)` Converts `ch` to its lowercase equivalent, if any. If not, `ch` is returned unchanged.
`char toupper(char ch)` Converts `ch` to its uppercase equivalent, if any. If not, `ch` is returned unchanged.

Modifying the Contents of a String

- In many languages, including Java, strings are *immutable,* which means that they never change once they are allocated. C++, by contrast, allows clients to change the contents of a string, both by assigning new values to selected characters and by calling string methods such as the following:

`str.insert(pos, text)`	←*Destructively changes* `str`
Inserts the characters from `text` before index position `pos`.	
`str.replace(pos, count, text)`	←*Destructively changes* `str`
Replaces `count` characters from `text` starting at position `pos`.	

- As a tool for writing programs that are easier to debug and maintain, immutable strings have many advantages over their modifiable counterparts in C++. Fortunately, it is easy to secure these advantages in C++ by avoiding the use of destructive operations like `insert`, `replace`, and assignment to individual characters.

The Legacy of C-Style Strings

- One of the fundamental design principles of C++ was that it would contain C as a subset. As a consequence, C++ must retain the older string model it inherits from its predecessor.
- Conceptually, a string is just an array of characters, which is precisely how strings are implemented in the C subset of C++. If you put double quotation marks around a sequence of characters, you get what is called a *C string,* in which the characters are stored in an array of bytes, terminated by a *null byte* whose ASCII value is 0. For example, the characters in the C string `"hello, world"` are arranged like this:

h	e	l	l	o	,		w	o	r	l	d	\0
0	1	2	3	4	5	6	7	8	9	10	11	12

- As in the `string` class model, character positions in a C string are identified by an *index* that begins at 0 and extends up to one less than the length of the string.

Concatenation and C Strings

- As those of you who have studied Python or Java already know, the `+` operator is a convenient shorthand for *concatenation,* which consists of combining two strings end to end with no intervening characters. In Java, this extension to `+` is part of the language. In C++, it is an extension to the `string` class.

- In Java, the `+` operator is often used to combine items as part of a `println` call, as in

```
println("The total is " + total + ".");
```

  In C++, you achieve the same result using the << operator:

```
cout << "The total is " << total << "." << endl;
```

- Although you might imagine otherwise, you can't use the `+` operator in this statement, because the quoted strings are C strings and not `string` objects.

The End