X

Exercises

You cannot learn bicycling
from a correspondence course.
– anon

Exercises for Bjarne Stroustrup: *The C++ Programming Language (4th Edition)*. Addison-Wesley ISBN 978-0321563842.

Corrections, suggested improvements, and more exercises are welcome.

This version of the exercises is dated May 12, 2013.

X.1 Introduction

You can argue that programming is an art, a craft, a science, or even a branch of mathematics. In any case, it inovolves some practical skills that cannot be learned simply by reading a book. The skills have to be learned by trying to apply the principles and techniques learned from books, articles, lectures, etc.

This ‘‘appendix’’ contains exercises for the readers of TC++PL. They are sorted by chapter and within a chapter roughly by difficulty. The exercises are mainly of the write-a-program variety. Always write enough code for a solution to be compiled and run with at least a few test cases.

The exercises vary considerably in difficulty, so they are marked with an estimate of their difficulty. The scale is exponential so that if a (∗1) exercise takes you ten minutes, a (∗2) might take an hour, and a (∗3) might take a day. The time needed to write and test a program depends more on your experience than on the exercise itself. A (∗1) exercise might take a day if you first have to get acquainted with a new computer system in order to run it. On the other hand, a (∗5) exercise might be done in an hour by someone who happens to have the right collection of programs handy. Many of the exercises marked (∗1) and (∗2) can be thought of as similar to the drills that musicians, athletes, and learners of a new natural language do to prevent unfamiliarity with simple subtasks from impeding the performance of more challenging tasks. They are not intellectual challenges in their own right.

Any book on programming in C can be used as a source of extra exercises for Part II (*The basics*; Chapters 6-15). Any book on data structures and algorithms can be used as a source of exercises for Parts III (*Abstraction Mechanisms*; Chapters 16-29) and IV (*The Standard Library*; Chapters 30-44).

I place this collection of exercises on the web because

- I do want to add another 80 pages to an already thick book.
- I want to add more exercises over the years.
- I hope readers will suggest improvements and new exercises so that eventually the set of exercises will be massive – more extensive and useful than I could make it on my own.

I cannot promise the numbering of exercises to be stable as I add exercises. I plan for relative stability after an initial peiod of major changes, but for now expect details of numbering to change.

X.2 The Structure of This Book

The exercises for this chapter focus on the history and philosophy of C++. They mostly aim at an undestanding of the rationale behind C++ facilities. Do not attempt these until you have a basic understanding of C++.

[1] (∗1) What does (∗2.5) mean for an exercise?
[2] (∗2) Briefly describe the design aims of C++ and comment on the extent to which C++ meets those.
[3] (∗4) Write an essay: What can a good programming language do for you and what can't you expect it to help with?
[4] (∗1) What are the main programming styles supported by C++?
[5] (∗2.5) List five language features offered by the 1985 version of C++, five features added by C++98, and finally five new features added by C++11. In each case, order the features in order of importance and for each feature write a sentence describing its role in programming.
[6] (∗3) Describe the difference between dynamic (run-time) and static (compile-time) type checking and outline the strengths and weaknesses of each.
[7] (∗1.5) List the major components of the C++ standard library.
[8] (∗1.5) List five libraries that you would have liked to be part of the standard.
[9] (∗1) List three (or more) advantages from having a library as part of the standard.
[10] (∗3) List 20 major real-world C++ applications.
[11] (∗2) From §1.3 pick five suggestions that to you looks most likely to help improve your programming style.
[12] (∗2) Make a ‘‘top-ten list’’ of helpful design and programming rules. Hint: §X.2.

X.3 A Tour of C++: The Basics

[1] When first reading this chapter, keep a record of information that was new or surprising to you. Later, use that list to focus your further studies.
[2] (∗1) What does a compiler do? What does a linker do?
[3] (∗2) Get the ‘‘Hello, world!’’ program (§2.2.1) to run. This is not an exercise in programming. It is an exercise to test your use of your edit-compile-link-execute tool chain.

[4] (∗1) List three (or more) C++ compilers.
[5] (∗1) Write out a **bool**, a **char**, an **int**, a **double**, and a **string**.
[6] (∗1) Read in a **bool**, a **char**, an **int**, a **double**, and a **string**.
[7] (∗2) What is an invariant and what good might it do?

X.4 A Tour of C++: Abstraction Mechanisms

[1] When first reading this chapter, keep a record of information that was new or surprising to you. Later, use that list to focus your further studies.
[2] (∗2) Give five examples of concrete types that are built-in types in C++. Give five examples of concrete types that are not built-in types in C++.

X.5 A Tour of C++: Containers and Algorithms

[1] When first reading this chapter, keep a record of information that was new or surprising to you. Later, use that list to focus your further studies.
[2] (∗1) List five standard-library containers.
[3] (∗1) List five standard-library algorithms.
[4] (∗1) List five standard-library headers.
[5] (∗1.5) Write a program that reads a name (a **string**) and an age (an **int**) from the standard input stream **cin**. Then output a message including the name and age to the standard output stream **cout**.
[6] (∗1.5) Redo §X.5[5], storing several (name,age) pairs in a class. Doing the reading and writing using your own **>>** and **<<** operators.
[7] (∗2) Initialize a **vector<int>** with the elements **5**, **9**, **−1**, **200**, and **0**. Print it. Sort is, and print it again.
[8] (∗1) Repeat §X.5[7] with a **vector<string>** initialized with **"Kant"**, **"Plato"**, **"Aristotle"**, **"Kierkegard"**, and **"Hume"**.
[9] (∗1) Open a file for writing (as an **ofstream**) and write a few hundred integers to it.
[10] (∗1) Open the file of integers from §X.5[9] for reading (as an **ifstream**) and read it.

X.6 A Tour of C++: Concurrency and Utilities

[1] When first reading this chapter, keep a record of information that was new or surprising to you. Later, use that list to focus your further studies.
[2] (∗1.5) Write a program with two **tread**s: one that writes **hello** every second and one that writes **world!** every second.
[3] (∗2) Time a loop. Write out the time in milliseconds. Do this for the default setting of your compiler and for a setting using an optimizer (e.g., **−O2** or "release"). Be careful not to have the optimizer eliminate your whole loop as dead code because you did not use a result.
[4] (∗2) Repeat the histogram drawing example from §5.6.3 for a **normal_distribution** and **30** rows.

[5] (∗1.5) Use a regex to find all decimal numbers in a file.

X.7 Types and Declarations

[1] (∗2) Get the "Hello, world!" program (§2.2.1) to run. This is not an exercise in programming. It is an exercise to test your use of your edit-compile-link-execute tool chain.

[2] (∗1) Write a program that prints signed if plain chars are signed on your implementation and unsigned otherwise.

[3] (∗1.5) Find 5 different C++ constructs for which the meaning is undefined (§6.1). (∗1.5) Find 5 different C++ constructs for which the meaning is implementation-defined (§6.1).

[4] (∗1) Find 10 different examples of nonportable C++ code.

[5] (∗1) For each declaration in §6.3, do the following: If the declaration is not a definition, write a definition for it. If the declaration is a definition, write a declaration for it that is not also a definition.

[6] (∗1.5) Write a program that prints the sizes of the fundamental types, a few pointer types, and a few enumerations of your choice. Use the sizeof operator.

[7] (∗1.5) Write a program that prints out the letters 'a'..'z' and the digits '0'..'9' and their integer values. Do the same for other printable characters. Do the same again but use hexadecimal notation.

[8] (∗2) What, on your system, are the largest and the smallest values of the following types: bool, char, short, int, long, long long, float, double, long double, unsigned and unsigned long.

[9] (∗1) What are the sizes (in number of chars) of the types mentioned in §X.7[8]?

[10] (∗1.5) What are the alignments (in number of chars) of the types mentioned in §X.7[8]?

[11] (∗2) Draw a graph of the integer and fundamental types where a type points to another type if all values of the first can be represented as values of the second on every standards-conforming implementation. Draw the same graph for the types on your favorite implementation.

[12] (∗1) What is the longest local name you can use in a C++ program on your system? What is the longest external name you can use in a C++ program on your system? Are there any restrictions on the characters you can use in a name?

[13] (∗1.5) Write a loop that prints out the values 4, 5, 9, 17, 12 without using an array or a vector.

X.8 Pointers, Arrays, and References

[1] (∗1) Write declarations for the following: a pointer to a character, an array of 10 integers, a reference to an array of 10 integers, a pointer to an array of character strings, a pointer to a pointer to a character, a constant integer, a pointer to a constant integer, and a constant pointer to an integer. Initialize each one.

[2] (∗1.5) What, on your system, are the restrictions on the pointer types char∗, int∗, and void∗? For example, may an int∗ have an odd value? Hint: alignment.

[3] (∗1) Use an alias (using) to define the types unsigned char, const unsigned char, pointer to integer, pointer to pointer to char, pointer to array of char, array of 7 pointers to int, pointer to an array of 7 pointers to int, and array of 8 arrays of 7 pointers to int.

[4] (∗1) Given two **char∗**s pointing into an array, find and output the number of characters between the two pointed-to characters (zero if they point to the same element).

[5] (∗1) Given two **int∗**s pointing into an array, find and output the number of **int**s between the two pointed-to **int**s (zero if they point to the same element).

[6] (∗2) What happens when you read and write beyond the bounds of an array. Do a few experiments involving a global array of **int**s , a local array of **int**s, an array of **int**s allocated by **new**, and a member array of **int**s. Try reading and writing just beyond the end and far beyond the end. Try the same for just before and far before the beginning. See what happens for different optimizer levels. Then try hard never to do out-of-range access by mistake.

[7] (∗1) Write a function that swaps (exchanges the values of) two integers. Use **int∗** as the argument type. Write another swap function using **int&** as the argument type.

[8] (∗1.5) What is the size of the array **str** in the following example:

```
char str[] = "a short string";
```

What is the length of the string **"a short string"**?

[9] (∗1) Define functions **f(char)**, **g(char&)**, and **h(const char&)**. Call them with the arguments **'a'**, **49**, **3300**, **c**, **uc**, and **sc**, where **c** is a **char**, **uc** is an **unsigned char**, and **sc** is a **signed char**. Which calls are legal? Which calls cause the compiler to introduce a temporary variable?

[10] (∗1) Define an array of strings in which the strings contain the names of the months. Print those strings. Pass the array to a function that prints those strings.

[11] (∗2) Read a sequence of words from input. Use **Quit** as a word that terminates the input. Print the words in the order they were entered. Don't print a word twice. Modify the program to sort the words before printing them.

[12] (∗2) Write a function that counts the number of occurrences of a pair of letters in a **string** and another that does the same in a zero-terminated array of **char** (a C-style string). For example, the pair **"ab"** appears twice in **"xabaacbaxabb"**.

[13] (∗2) Run some tests to see if your compiler really generates equivalent code for iteration using pointers and iteration using indexing (§7.4.1). If different degrees of optimization can be requested, see if and how that affects the quality of the generated code.

X.9 Structures, Unions, and Enumerations

[1] (∗1) Define a **struct** with a member of each of the types **bool**, **char**, **int**, **long**, **double**, and **long double**. Order the members so as to get the largest size of the **struct** and the smallest size of the **struct**.

[2] (∗1.5) Define a table of the names of months of the year and the number of days in each month. Write out that table. Do this twice; once using an array of **char** for the names and an array for the number of days and once using an array of structures, with each structure holding the name of a month and the number of days in it.

[3] (∗1.5) Find an example where it would make sense to use a name in its own initializer.

[4] (∗1.5) Define a **struct Date** to keep track of dates. Provide functions that read **Date**s from input, write **Date**s to output, and initialize a **Date** with a date.

[5] (∗2) Implement an enum called Season with enumerators spring, summer, autumn, and winter. Define operators ++ and −− for Season. Define input (>>) and output (<<) operations for Season, providing string values. Provide a way to control the mapping between Season values and their string representations. For example, I might want the strings to reflect the Danish names for the seasons. For further study, see Chapter 39.

X.10 Statements

[1] (∗1) Rewrite the following for-statement as an equivalent while-statement:

```
for (i=0; i!=max_length; i++)
        if (input_line[i] == '?')
        quest_count++;
```

Rewrite it to use a pointer as the controlled variable, that is, so that the test is of the form ∗p=='?'. Rewrite it to use a range-for.

[2] (∗1) See how your compiler reacts to these errors:

```
void f(int a, int b)
{
        if (a = 3) // ...
        if (a&077 == 0) // ...
        a := b+1;
}
```

Devise more simple errors and see how the compiler reacts.

[3] (∗1.5) What does the following example do?

```
void send(int∗ to, int∗ from, int count)
        // Duff’s device. Helpful comment deliberately deleted.
{
        int n = (count+7)/8;
        switch (count%8) {
        case 0:    do {  ∗to++ = ∗from++;
        case 7:          ∗to++ = ∗from++;
        case 6:          ∗to++ = ∗from++;
        case 5:          ∗to++ = ∗from++;
        case 4:          ∗to++ = ∗from++;
        case 3:          ∗to++ = ∗from++;
        case 2:          ∗to++ = ∗from++;
        case 1:          ∗to++ = ∗from++;
                } while (−−n>0);
        }
}
```

Why would anyone write something like that? No, this is not recommended as good style.

[4] (∗2) Write a function atoi(const char∗) that takes a C-style string containing digits and returns the corresponding int. For example, atoi("123") is 123. Modify atoi() to handle C++ octal and hexadecimal notation in addition to plain decimal numbers. Modify atoi() to handle the C++ character constant notation.

[5] (*2) Write a function `itoa(int i, char b[])` that creates a string representation of `i` in `b` and returns `b`.

[6] Modify `iota()` from the previous exercise to take an extra "string length" argument to make overflow less likely.

[7] (*2.5) Write a program that strips comments out of a C++ program. That is, read from `cin`, remove both `//` comments and `/* */` comments, and write the result to `cout`. Do not worry about making the layout of the output look nice (that would be another, and much harder, exercise). Do not worry about incorrect programs. Beware of `//`, `/*`, and `*/` in comments, strings, and character constants.

[8] (*2) Look at some programs to get an idea of the variety of indentation, naming, and commenting styles actually used.

X.11 Expressions

[1] (*1) Fully parenthesize the following expressions:

```
a = b + c * d << 2 & 8
a & 077 != 3
a == b || a == c && c < 5
c = x != 0
0 <= i < 7
f(1,2)+3
a = - 1 + + b -- - 5
a = b == c ++
a = b = c = 0
a[4][2] *= * b ? c : * d * 2
a-b,c=d
```

[2] (*2) Read a sequence of possibly whitespace-separated (name,value) pairs, where the name is a single whitespace-separated word and the value is an integer or a floating-point value. Compute and print the sum and mean for each name and the sum and mean for all names. Hint: §10.2.8.

[3] (*1) Write a table of values for the bitwise logical operations (§11.1.1) for all possible combinations of `0` and `1` operands.

[4] (*2) Write 5 expressions for which the order of evaluation is undefined. Execute them to see what one or – preferably – more implementations do with them.

[5] (*1.5) What happens if you divide by zero on your system? What happens in case of overflow and underflow?

[6] (*1) Fully parenthesize the following expressions:

```
*p++
*--p
++a--
(int*)p->m
*p.m
*a[i]
```

[7] (∗2) Implement and test these functions: **strlen()**, which returns the length of a C-style string; **strcpy()**, which copies a C-style string into another; and **strcmp()**, which compares two C-style strings. Consider what the argument types and return types ought to be. Then compare your functions with the standard library versions as declared in **<cstring>** (**<string.h>**) and as specified in §43.4.

[8] (∗2) Modify the program from §X.11[3] to also compute the median.

[9] (∗2) Write a function **cat()** that takes two C-style string arguments and returns a string that is the concatenation of the arguments. Use **new** to find store for the result.

[10] (∗2) Write a function **rev()** that takes a C-style string argument and reverses the characters in it. That is, after **rev(p)** the last character of **p** will be the first, etc.

[11] (∗2) Type in the calculator example and get it to work. Do not ‘‘save time’’ by using an already entered text. You’ll learn most from finding and correcting ‘‘little silly errors.’’

[12] (∗2) Modify the calculator to report line numbers for errors.

[13] (∗3) Allow a user to define functions in the calculator. Hint: Define a function as a sequence of operations just as a user would have typed them. Such a sequence can be stored either as a character string or as a list of tokens. Then read and execute those operations when the function is called. If you want a user-defined function to take arguments, you will have to invent a notation for that.

[14] (∗1.5) Convert the desk calculator to use a **symbol** structure instead of using the static variables **number_value** and **string_value**.

[1] (∗1) Copy all even non-zero elements of an **int[]** into a **vector<int>**. Use a pointer and **++** for the traversal.

X.12 Select Operations

[2] (∗2) Allocate so much memory using **new** that **bad_alloc** is thrown. Report how much memory was allocated and how much time it took. Do this twice: once not writing to the allocated memory and once writing to each element.

[3] Write a simple loop calculating a sum of elements (like **std::accumulate()**). Write it in a dozen or more ways using **for**-statements, range-**for** statements, the **for_each()** algorithm, using indices, pointers, and iterators, using ‘‘plain code’’, function objects, and lambdas, and using different element types. See if you can find any performance differences between the different versions.

[4] (∗2.5) Define an **apply(v,f)** that applies a function **f** to each element of **v** assumed to be a **vector<Shape∗>**. Test **apply()** with a a variety of functions, function objects, and lambdas. Note that by capturing variables from a lambda or storing values in a function object, you can call **Shape** functions that takes arguments without having to have **f()** take explicit arguments. Hint: there is a variety of ways of specifying **apply()**’s argument types; experiment.

[5] (∗4) Find a program of a few thousand lines of code, preferably a program used for a real-world task, rather than an exercise solution. Count the number of casts (of all kinds). If not already done, classify all casts by turning them into named casts. Eliminate as many **static_cast**s, **reinterpre_cast**s, and **const_cast**s as possible. This will most likely involve adding specific type conversion functions, templates, etc.

X.13 Functions

[1] (∗1) Write declarations for the following: a function taking arguments of type pointer to character and reference to integer and returning no value; a pointer to such a function; a function taking such a pointer as an argument; and a function returning such a pointer. Write the definition of a function that takes such a pointer as an argument and returns its argument as the return value. Hint: Use a type alias (**using**).

[2] (∗1) What does the following mean? What would it be good for?

```
using riffi = int (&) (int, int);
```

[3] (∗1.5) Write a program like ‘‘Hello, world!’’ that takes a name as a command-line argument and writes ‘‘Hello, *name* **!**’’. Modify this program to take any number of names as arguments and to say hello to each.

[4] (∗1.5) Write a program that reads an arbitrary number (possibly limited to some maximum number) of files whose names are given as command-line arguments and writes them one after another on **cout**. Because this program concatenates its arguments to produce its output, you might call it **cat**.

[5] (∗2) Convert a small C program to C++. Modify the header files to declare all functions called and to declare the type of every argument. Where possible, replace **#define**s with **enum**, **const**, **constexpr**, or **inline**. Remove **extern** declarations from **.cpp** files and if necessary convert all function definitions to C++ function definition syntax. Replace calls of **malloc()** and **free()** with **new** and **delete**. Remove unnecessary casts.

[6] (∗2) Modify the result of §X.13[5] by eliminating arrays and ‘‘naked’’ **new**s and **delete**s. Hint: **vector** and **array**.

[7] (∗2) Implement **ssort()** (§12.5) using a more efficient sorting algorithm. Hint: **sort()** and **qsort()**.

[8] (∗2.5) Consider:

```
struct Tnode {
        string word;
        int count;
        Tnode* left;
        Tnode* right;
};
```

Write a function for entering new words into a tree of **Tnode**s. Write a function to write out a tree of **Tnode**s. Write a function to write out a tree of **Tnode**s with the words in alphabetical order. Modify **Tnode** so that it stores (only) a pointer to an arbitrarily long word stored as an array of characters on free store using **new**. Modify the functions to use the new definition of **Tnode**.

[9] (∗2.5) Write a function to invert a two-dimensional array. Hint: §7.4.2.

[10] (∗2) Write an encryption program that reads from **cin** and writes the encoded characters to **cout**. You might use this simple encryption scheme: the encrypted form of a character **c** is **c^key[i]**, where **key** is a string passed as a command-line argument. The program uses the characters in **key** in a cyclic manner until all the input has been read. Re-encrypting encoded text with the same key produces the original text. If no key (or a null string) is passed, then

no encryption is done.

[11] (∗3.5) Write a program to help decipher messages encrypted with the method described in §X.13[9] without knowing the key. Hint: See David Kahn: *The Codebreakers*, Macmillan, 1967, New York, pp. 207-213.

[12] (∗2) Without using copy and paste, implement and test TEA (the Tiny Encryption Algorithm). D.J. Wheeler and R.M. Needham: *TEA, a tiny encryption algorithm*. Lecture Notes in Computer Science 1008: 363366. http://143.53.36.235:8080/tea.htm.

[13] (∗1) How would you choose names for pointer to function types defined using a type alias?

[14] (∗2) Look at some programs to get an idea of the diversity of styles of names actually used. How are uppercase letters used? How is the underscore used? When are short names such as `i` and `x` used?

[15] (∗1) What is wrong with these macro definitions?

```
#define PI = 3.141593;
#define MAX(a,b) a>b?a:b
#define fac(a) (a)∗fac((a)−1)
```

[16] (∗3) Write a macro processor that defines and expands simple macros (like the C preprocessor does). Read from `cin` and write to `cout`. At first, don't try to handle macros with arguments. Hint: The desk calculator (§10.2) contains a symbol table and a lexical analyzer that you could modify.

[17] (∗3) Write an `error` function that takes a `printf`-style format string containing `%s`, `%c`, and `%d` directives and an arbitrary number of arguments. Don't use `printf()`. Look at §43.3 if you don't know the meaning of `%s`, `%c`, and `%d`. Use `<cstdarg>`.

[18] (∗2) Implement a useful subset of `print()` from §12.2.5.

[19] (∗2) Add functions such as `sqrt()`, `log()`, and `sin()` to the desk calculator from §10.2. Hint: Predefine the names and call the functions through an array of pointers to functions. Don't forget to check the arguments in a function call.

[20] (∗1) Write a factorial function that does not use recursion.

[21] (∗2) Write functions to add one day, one month, and one year to a `Date` as defined in §X.9[4]. Write a function that gives the day of the week for a given `Date`. Write a function that gives the `Date` of the first Monday following a given `Date`.

X.14 Exception Handling

[1] (∗3) Write a `Checked_ptr<T>` that uses exceptions to signal run-time errors for a pointer supposed to point to an element of an array (or one-beyond-the-end-of the array).

[2] (∗3) Write a function that searches a binary tree of nodes based on a `char∗` field for a match. If a node containing `hello` is found, `find("hello")` will return a pointer to that node. Use an exception to indicate "not found."

[3] (∗3) Define a class `Int` that acts exactly like the built-in type `int`, except that it throws exceptions rather than overflowing or underflowing.

[4] (∗2.5) Take the basic operations for opening, closing, reading, and writing from the C interface to your operating system and provide equivalent C++ functions that call the C functions but throw exceptions in case of errors.

[5] (∗2.5) Write a complete `Vector` template with `Range` and `Size` exceptions.

[6] (∗1) Write a loop that computes the sum of a `Vector` as defined in §X.14[5] without examining the size of the `Vector`. Why is this a bad idea?

[7] (∗2.5) Consider using a class `Exception` as the base of all classes used as exceptions. What should it look like? How should it be used? What good might it do? What disadvantages might result from a requirement to use such a class?

[8] (∗1) Given:

```
int main() { /* ... */ }
```

Change it so that it catches all exceptions thrown from the `...`, turns them into error messages, and `abort()`s.

[9] (∗2) Write a class or template suitable for implementing callbacks.

[10] (∗2) Write a program consisting of functions calling each other to a calling depth of 10. Give each function an argument that determines at which level an exception is thrown. Have `main()` catch these exceptions and print out which exception is caught. Don't forget the case in which an exception is caught in the function that throws it.

[11] (∗2) Modify the program from §X.14[10] to measure if there is a difference in the cost of catching exceptions depending on where in the function call stack the exception is thrown. Add a string object to each function and measure again.

[12] (∗2) Write a function that either returns a value or that throws that value based on an argument. Measure the difference in run-time between the two ways.

[13] (∗2) Modify the calculator version from §X.15[2] to use exceptions. Keep a record of the mistakes you make. Suggest ways of avoiding such mistakes in the future.

[14] (∗2.5) Write `plus()`, `minus()`, `multiply()`, and `divide()` functions that check for possible overflow and underflow and that throw exceptions if such errors happen.

[15] (∗2) Modify the calculator to use the functions from §X.14[14].

X.15 Namespaces

[1] (∗2) Take some not-too-large program that uses at least one library that does not use namespaces and modify it to use a namespace for that library.

[2] (∗2.5) Modify the desk calculator program into a module with a simple interface specifying calls and potential errors. Don't expose implemetation details to users. Don't make it easy for users to supply bad data. Don't use any global *using-directive*s. Keep a record of the mistakes you made. Suggest ways of avoiding such mistakes in the future.

X.16 Files

[1] (∗2) Find where the standard library headers are kept on your system. List their names. Are any nonstandard headers kept together with the standard ones? Can any nonstandard headers be `#include`d using the `<>` notation?

[2] (∗2) Where are the headers for nonstandard ''foundation'' libraries kept?

[3] (∗2.5) Write a program that reads a source file and writes out the names of files `#include`d. Indent file names to show files `#included` by included files. Try this program on some real source files (to get an idea of the amount of information included).

[4] (∗3) Modify the program from the previous exercise to print the number of comment lines, the number of non-comment lines, and the number of non-comment, whitespace-separated words for each file `#include`d.

[5] (∗2.5) An external include guard is a construct that tests outside the file it is guarding and `include`s only once per compilation. Define such a construct, devise a way of testing it, and discuss its advantages and disadvantages compared to the include guards described in §15.3.3. Is there any significant run-time advantage to external include guards on your system?

[6] (∗3) How is dynamic linking achieved on your system? What restrictions are placed on dynamically linked code? What requirements are placed on code for it to be dynamically linked?

[7] (∗3) Open and read 100 files containing 1500 characters each. Open and read one file containing 150,000 characters. Is there a performance difference? What is the highest number of files that can be simultaneously open on your system? Consider these questions in relation to the use of `#include` files.

[8] (∗2) Modify the desk calculator so that it can be invoked from `main()` or from other functions as a simple function call.

[9] (∗2) Draw the ‘‘module dependency diagrams’’ (§15.3.2) for the version of the calculator that used `error()` instead of exceptions (§14.2.2).

X.17 Classes

[1] (∗1) Find the error in `Date::add_year()` in §16.2.3. Then find two additional errors in the version in §16.2.10.

[2] (∗2.5) Complete and test `Date`. Reimplement it with ‘‘number of days after 1/1/1970’’ representation.

[3] (∗2) Find a `Date` class that is in commercial use. Critique the facilities it offers. If possible, then discuss that `Date` with a real user.

[4] (∗2) Define a class `Histogram` that keeps count of numbers in some intervals specified as arguments to `Histogram`’s constructor. Provide functions to print out the histogram. Handle out-of-range values.

[5] (∗2.5) Complete class `Table` to hold (name,value) pairs. Then modify the desk calculator program from §10.2 to use class `Table` instead of `map`. Compare and contrast the two versions.

[6] (∗2) Rewrite `Tnode` from §X.13[8] as a class with constructors, destructors, etc. Define a tree of `Tnode`s as a class with constructors, destructors, etc.

[7] (∗3) Define, implement, and test a set of integers, class `Intset`. Provide union, intersection, and symmetric difference operations.

[8] (∗1.5) Modify class `Intset` into a set of nodes, where `Node` is a structure you define.

[9] (∗3) Define a class for analyzing, storing, evaluating, and printing simple arithmetic expressions consisting of integer constants and the operators `+`, `−`, `∗`, and `/`. The public interface

should look like this:

```
class Expr {
        // ...
public:
        Expr(const char*);
        int eval();
        void print();
};
```

The string argument for the constructor Expr::Expr() is the expression. The function Expr::eval() returns the value of the expression, and Expr::print() prints a representation of the expression on cout. A program might look like this:

```
Expr x("123/4+123*4−3");
cout << "x = " << x.eval() << "\n";
x.print();
```

Define class Expr twice: once using a linked list of nodes as the representation and once using a character string as the representation. Experiment with different ways of printing the expression: fully parenthesized, postfix notation, assembly code, etc.

[10] (∗2) Define a class Char_queue so that the public interface does not depend on the representation. Implement Char_queue (a) as a linked list and (b) as a vector. Do not worry about concurrency.

[11] (∗3) Design a symbol table class and a symbol table entry class for some language. Have a look at a compiler for that language to see what the symbol table really looks like.

[12] (∗2) Modify the expression class from §X.17[11] to handle variables and the assignment operator =. Use the symbol table class from §X.17[11].

[13] (∗1) Given this program:

```
#include <iostream>

int main()
{
        std::cout << "Hello, world!\n";
}
```

modify it to produce this output:

```
Initialize
Hello, world!
Clean up
```

Do not change main() in any way.

[14] (∗2) Define two classes, each with a static member, so that the construction of each static member involves a reference to the other. Where might such constructs appear in real code? How can these classes be modified to eliminate the order dependence in the constructors?

[15] (∗2.5) Compare class `Date` (§16.3) with your solution to §X.9[4] and §X.13[21]. Discuss errors found and likely differences in maintenance of the two solutions.

[16] (∗3) Write a function that, given an `istream` and a `vector<string>`, produces a `map<string,vector<int> >` holding each string and the numbers of the lines on which the string appears. Run the program on a text-file with no fewer than 1,000 lines looking for no fewer than 10 words.

X.18 Construction, Cleanup, Copy, and Move

[1] (∗2) Implement the functions of the ‘‘cannonical complete class’’ `X` from §17.1 to do noting but print out their name and the address of their object (and of their argument where applicable). Now test `X` by defining variables, members, objects on the free store, passing arguments, etc.

[2] (∗2) Define a resource handle, a ‘‘smart pointer’’ that holds a pointer to an object on the free store passed as an argument to a constructor. ‘‘Forget’’ to define copy operations. Now test the handle by defining variables, members, objects on the free store, passing arguments, etc., and see where leaks occur.

[3] (∗1.5) When do you need a constructor? When do memberwise construction suffice? When is memberwise construction preferable?

[4] (∗1) Design, implement and test a `Vector` class with an initializer-list constructor and no constructor that takes a integer as a size. Instead, give it a constructo that takes a `Count` type, so that you can’t get ambiguities between and integer number of elements and an integer element value.

[5] (∗3) Write an extremely simple document editor with only three commands: ‘‘Insert a `string` as line number n,’’ ‘‘delete line n,’’ and ‘‘undo the last (non-undo) operation.’’ In addition, provide a cursor to a character in the document (or one beyond the last character) that can be moved forwards and backwards. Do not let any command lead to an invalid cursor. Hint: §17.5.1.3.

[6] (∗3) Implement a simple `Matrix` along the lines of the one in §17.5.2. Measure the performance of an addition of 1000-by-1000 matrices with and without move operations. If you did not have the option of adding move operations, what would you do?

X.19 Overloading

[1] (∗2) In the following program, which conversions are used in each expression?

```
struct X {
      int i;
      X(int);
      X operator+(int);
};
```

```
struct Y {
        int i;
        Y(X);
        Y operator+(X);
        operator int();
};

extern X operator*(X, Y);
extern int f(X);

X x = 1;
Y y = x;
int i = 2;

int main()
{
        i + 10;
        y + 10;
        y + 10 * y;

        x + y + i;
        x * x + i;

        f(7);
        f(y);

        y + y;
        106 + y;
}
```

Modify the program so that it will run and print the values of each legal expression.

[2] (∗2) Complete and test class **String** from §19.3.

[3] (∗2) Define a class **INT** that behaves exactly like an **int**. Hint: Define **INT::operator int()**.

[4] (∗1) Define a class **RINT** that behaves like an **int** except that the only operations allowed are **+** (unary and binary), **−** (unary and binary), **∗**, **/**, and **%**. Hint: Do not define **RINT::operator int()**.

[5] (∗3) Define a class **LINT** that behaves like a **RINT**, except that it has at least 64 bits of precision.

[6] (∗4) Define a class implementing arbitrary precision arithmetic. Test it by calculating the factorial of **1000**. Hint: You will need to manage storage in a way similar to what was done for class **String**.

[7] (∗2) Write a program that has been rendered unreadable through use of operator overloading and macros. An idea: Define **+** to mean **−** and vice versa for **INT**s. Then, use a macro to define **int** to mean **INT**. Redefine popular functions using reference type arguments. Writing a few misleading comments can also create great confusion.

[8] (∗3) Swap the result of §X.19[7] with a friend. Without running it, figure out what your friend's program does. When you have completed this exercise, you'll know what to avoid.

[9] (∗2) Define a type **Vec4** as a vector of four **float**s. Define **operator[]** for **Vec4**. Define operators **+**, **−**, **∗**, **/**, **=**, **+=**, **−=**, **∗=**, and **/=** for combinations of vectors and floating-point numbers.

[10] (∗3) Define a class **Mat4** as a vector of four **Vec4**s. Define **operator[]** to return a **Vec4** for **Mat4**. Define the usual matrix operations for this type. Define a function doing Gaussian elimination for a **Mat4**.

[11] (∗2) Define a class **Vector** similar to **Vec4** but with the size given as an argument to the constructor **Vector::Vector(int)**.

[12] (∗3) Define a class **Matrix** similar to **Mat4** but with the dimensions given as arguments to the constructor **Matrix::Matrix(int,int)**.

[13] (∗1) Given two structures:

```
struct S { int x, y; };
struct T { char∗ p; char∗ q; };
```

write a class **C** that allows the use of **x** and **p** from some **S** and **T**, much as if **x** and **p** had been members of **C**. Hint: pointer to member.

[14] (∗1.5) Define a class **Index** to hold the index for an exponentiation function **mypow(double,Index)**. Find a way to have **2∗∗I** call **mypow(2,I)**.

[15] (∗2) Define a class **Imaginary** to represent imaginary numbers. Define class **Complex** based on that. Implement the fundamental arithmetic operators. Define **i** as a user-defined literal suffix meaning “imaginary.”

X.20 Special Operations

[1] (∗2) Complete class **Ptr** from §19.2.4 and test it. To be complete, **Ptr** must have at least the operators **∗**, **−>**, **=**, **++**, and **−−** defined. Do not cause a run-time error until a wild pointer is actually dereferenced.

[2] (∗2) Define an external iterator for class **String**:

```
class String_iter {
        // refer to string and string element
public:
        String_iter(String& s);          // iterator for s
        char& next();                    // reference to next element

        // more operations of your choice
};
```

Compare this in utility, programming style, and efficiency to having an internal iterator for **String** (that is, a notion of a current element for the **String** and operations relating to that element).

[3] (∗1.5) Provide a substring operator for a string class by overloading **()**. What other operations would you like to be able to do on a string?

[4] (∗3) Design class **String** so that the substring operator can be used on the left-hand side of an assignment. First, write a version in which a string can be assigned to a substring of the same length. Then, write a version in which the lengths may differ.

[5] (∗2) Define an operation for `String` that produces a C-string representation of its value. Discuss the pros and cons of having that operation as a conversion operator. Discuss alternatives for allocating the memory for that C-string representation.

[6] (∗2.5) Define and implement a simple regular expression pattern match facility for class `String`.

[7] (∗1.5) Modify the pattern match facility from §X.20[6] to work on the standard library `string`. Note that you cannot modify the definition of `string`.

X.21 Derived Classes

[1] (∗1) Define

```
class Base {
public:
        virtual void iam() { cout << "Base\n"; }
};
```

Derive two classes from `Base`, and for each define `iam()` to write out the name of the class. Create objects of these classes and call `iam()` for them. Assign pointers to objects of the derived classes to `Base*` pointers and call `iam()` through those pointers.

[2] (∗3.5) Implement a simple graphics system using whatever graphics facilities are available on your system (if you don't have a good graphics system or have no experience with one, you might consider a simple ''huge bit ASCII implementation'' where a point is a character position and you write by placing a suitable character, such as `*` in a position): `Window(n,m)` creates an area of size `n` times `m` on the screen. Points on the screen are addressed using (x,y) coordinates (Cartesian). A `Window w` has a current position `w.current()`. Initially, `current` is `Point(0,0)`. The current position can be set by `w.current(p)` where `p` is a `Point`. A `Point` is specified by a coordinate pair: `Point(x,y)`. A `Line` is specified by a pair of `Point`s: `Line(w.current(),p2);` class `Shape` is the common interface to `Dot`s, `Line`s, `Rectangle`s, `Circle`s, etc. A `Point` is not a `Shape`. A `Dot`, `Dot(p)` can be used to represent a `Point p` on the screen. A `Shape` is invisible unless `draw()`n. For example: `w.draw(Circle(w.current(),10))`. Every `Shape` has 9 contact points: `e` (east), `w` (west), `n` (north), `s` (south), `ne`, `nw`, `se`, `sw`, and `c` (center). For example, `Line(x.c(),y.nw())` creates a line from `x`'s center to `y`'s top left corner. After `draw()`ing a `Shape` the current position is the `Shape`'s `se().` A `Rectangle` is specified by its bottom left and top right corner: `Rectangle(w.current(),Point(10,10))`. As a simple test, display a simple ''child's drawing of a house'' with a roof, two windows, and a door.

[3] (∗2) Important aspects of a `Shape` appear on the screen as a set of line segments. Implement operations to vary the appearance of these segments: `s.thickness(n)` sets the line thickness to `0,` `1`, `2`, or `3`, where `2` is the default and `0` means invisible. In addition, a line segment can be `solid`, `dashed`, or `dotted`. This is set by the function `Shape::outline()`.

[4] (∗2.5) Provide a function `Line::arrowhead()` that adds arrow heads to an end of a line. A line has two ends and an arrowhead can point in two directions relative to the line, so the argument or arguments to `arrowhead()` must be able to express at least four alternatives.

[5] (∗3.5) Make sure that points and line segments that fall outside the `Window` do not appear on the screen. This is often called ''clipping.'' As an exercise only, do not rely on the

implementation graphics system for this.

[6] (∗2.5) Add a Text type to the graphics system. A Text is a rectangular Shape displaying characters. By default, a character takes up one coordinate unit along each coordinate axis.

[7] (∗2) Define a function that draws a line connecting two shapes by finding the two closest ‘‘contact points’’ and connecting them.

[8] (∗3) Add a notion of color to the simple graphics system. Three things can be colored: the background, the inside of a closed shape, and the outlines of shapes.

[9] (∗2) Consider:

```
class Char_vec {
        int sz;
        char element[1];
public:
        static Char_vec∗ new_char_vec(int s);
        char& operator[](int i) { return element[i]; }
        // ...
};
```

Define new_char_vec() to allocate contiguous memory for a Char_vec object so that the elements can be indexed through element as shown. Under what circumstances does this trick cause serious problems?

[10] (∗2.5) Given classes Circle, Square, and Triangle derived from a class Shape, define a function intersect() that takes two Shape∗ arguments and calls suitable functions to determine if the two shapes overlap. It will be necessary to add suitable (virtual) functions to the classes to achieve this. Don’t bother to write the code that checks for overlap; just make sure the right functions are called. This is commonly referred to as *double dispatch* or a *multi-method*.

[11] (∗5) Design and implement a library for writing event-driven simulations. Hint: <task.h>. However, that is an old program, and you can do better. There should be a class task. An object of class task should be able to save its state and to have that state restored (you might define task::save() and task::restore()) so that it can operate as a coroutine. Specific tasks can be defined as objects of classes derived from class task. The program to be executed by a task might be specified as a virtual function. It should be possible to pass arguments to a new task as arguments to its constructor(s). There should be a scheduler implementing a concept of virtual time. Provide a function task::delay(long) that ‘‘consumes’’ virtual time. Whether the scheduler is part of class task or separate will be one of the major design decisions. The tasks will need to communicate. Design a class queue for that. Devise a way for a task to wait for input from several queues. Handle run-time errors in a uniform way. How would you debug programs written using such a library?

[12] (∗2) Define interfaces for Warrior, Monster, and Object (that is a thing you can pick up, drop, use, etc.) classes for an adventure-style game.

[13] (∗1.5) Why is there both a Point and a Dot class in §X.21[2]? Under which circumstances would it be a good idea to augment the Shape classes with concrete versions of key classes such as Line?

[14] (∗3) Outline a different implementation strategy for the Ival_box example (§21.2) based on the idea that every class seen by an application is an interface containing a single pointer to the implementation. Thus, each "interface class" will be a handle to an "implementation

class," and there will be an interface hierarchy and an implementation hierarchy. Write code fragments that are detailed enough to illustrate possible problems with type conversion. Consider ease of use, ease of programming, ease of reusing implementations and interfaces when adding a new concept to the hierarchy, ease of making changes to interfaces and implementations, and need for recompilation after change in the implementation.

[15] (∗2) Write a version of the **clone()** operation from §20.3.6 that can place its cloned object in an **Arena** (see §11.2.4) passed as an argument. Implement a simple **Arena** as a class derived from **Arena**.

X.22 Class Hierarchies

[16] (∗3.5) Implement a version of a Reversi/Othello board game. Each player can be either a human or the computer. Focus on getting the program correct and (then) getting the computer player ‘‘smart’’ enough to be worth playing against.

[17] (∗3) Improve the user interface of the game from §X.22[16].

[18] (∗3) Define a graphical object class with a plausible set of operations to serve as a common base class for a library of graphical objects; look at a graphics library to see what operations were supplied there. Define a database object class with a plausible set of operations to serve as a common base class for objects stored as sequences of fields in a database; look at a database library to see what operations were supplied there. Define a graphical database object with and without the use of multiple inheritance and discuss the relative merits of the two solutions.

[19] (∗2.5) Draw a plausible memory layout for a **Radio** as defined in §21.3.4. Explain how a virtual function call could be implemented.

[20] (∗2.5) Draw a plausible memory layout for a **Radio** as defined in §22.2.2. Explain how a virtual function call could be implemented.

[21] (∗2) Assume that the type-checking rules for arguments were relaxed in a way similar to the relaxation for return types so that a function taking a **Derived∗** could override a function taking a **Base∗**. Then write a program that would corrupt an object of class **Derived** without using a cast. Describe a safe relaxation of the overriding rules for argument types.

X.23 Run-Time Type Information

[1] (∗1) Write a template **ptr_cast** that works like **dynamic_cast**, except that it throws **bad_cast** rather than returning **0**.

[2] (∗2) Write a program that illustrates the sequence of constructor calls at the state of an object relative to RTTI during construction. Similarly illustrate destruction.

[3] (∗3) Consider how **dynamic_cast** might be implemented. Define and implement a **dcast** template that behaves like **dynamic_cast** but relies on functions and data you define only. Make sure that you can add new classes to the system without having to change the definitions of **dcast** or previously-written classes.

[4] (∗3) What information would you like to have associated with a class to be accessible at run time? Implement and association scheme as outlined in §22.5.1 as an

unordered_map<type_index,My_info*> (§35.5.4). First make a simple version where **My_info** just holds a string associated with the class (e.g., a comment about its purpose). Second, implement a version that does what you want (e.g. holds a table of member function names and pointers to member functions).

X.24 Templates

[1] (*2) Fix the errors in the definition of **List** from §23.3.2 and write out C++ code equivalent to what the compiler must generate for the definition of **List** and the function **f()**. Run a small test case using your hand-generated code and the code generated by the compiler from the template version. If possible on your system given your knowledge, compare the generated code.

[2] (*3) Write a singly-linked list class template that accepts elements of any type derived from a class **Link** that holds the information necessary to link elements. This is called an *intrusive list*. Using this list, write a singly-linked list that accepts elements of any type (a non-intrusive list). Compare the performance of the two list classes and discuss the tradeoffs between them.

[3] (*2.5) Write intrusive and non-intrusive doubly-linked lists. What operations should be provided in addition to the ones you found necessary to supply for a singly-linked list?

[4] (*2) Complete the **String** template from §23.2 based on the **String** class from §19.3.

[5] (*2) Define a **sort()** that takes its comparison criterion as a template argument. Define a class **Record** with two data members **count** and **price**. Sort a **vector<Record>** on each data member.

[6] (*2) Implement a **qsort()** template.

[7] (*2) Write a program that reads **(key,value)** pairs and prints out the sum of the **value**s corresponding to each distinct **key**. Specify what is required for a type to be a **key** and a **value**.

[8] (*2.5) Implement a simple **Map** class based on the **Assoc** class from §19.2.1. Make sure **Map** works correctly using both C-style strings and **string**s as keys. Make sure **Map** works correctly for types with and without default constructors. Provide a way of iterating over the elements of a **Map**.

[9] (*3) Compare the performance of the word count program from §19.2.1 against a program not using an associative array. Use the same style of I/O in both cases.

[10] (*3) Re-implement **Map** from §X.24[8] using a more suitable data structure (e.g., a red-black tree or a Splay tree).

[11] (*2.5) Use **Map** to implement a topological sort function. Topological sort is described in [Knuth,1968] vol. 1 (second edition), pg 262.

[12] (*1.5) Make the sum program from §X.24[7] work correctly for names containing spaces; for example, “thumb tack.”

[13] (*2) Write **readline()** templates for different kinds of lines. For example (item,count,price).

[14] (*1.5) Construct an example that demonstrates at least three differences between a function template and a macro (not counting the differences in definition syntax).

[15] (*2) Devise a scheme that ensures that the compiler tests general constraints on the template arguments for every template for which an object is constructed. It is not sufficient just to test constraints of the form “the argument **T** must be a class derived from **My_base**.”

X.25 Generic Programming

[1] (∗3) Write a function that writes an array of C-stryle strings to stdout . Write a function that writes a **list<string>** to **cout**. Lift this algorithm so that it can write any sequence or any element type using any I/O system.

X.26 Specialization

[1] ???

X.27 Instantiation

[1] ???

X.28 Templates and Hierarchies

[1] ???

X.29 Metaprogramming

[1] ???

X.30 A Matrix Design

[1] ???

X.31 Library Overview

[1] (∗1) How many standard-library header files are there? If you could have only ten, which ten would you choose?
[2] (∗1) Which operator is missing from **std::initializer_list**?

X.32 STL Containers

The solutions to several exercises for this chapter can be found by looking at the source text of an implementation of the standard library. Do yourself a favor: try to find your own solutions before looking to see how your library implementer approached the problems.
[1] (∗1.5) Create a **vector<char>** containing the letters of the alphabet in order. Print the elements of that vector in order and in reverse order.
[2] (∗1.5) Create a **vector<string>** and read a list of names of fruits from **cin** into it. Sort the list and print it.

[3] (∗1.5) Using the vector from §X.32[2], write a loop to print the names of all fruits with the initial letter a.

[4] (∗1) Using the vector from §X.32[2], write a loop to delete all fruits with the initial letter a.

[5] (∗1) Using the vector from §X.32[2], write a loop to delete all citrus fruits.

[6] (∗1.5) Using the vector from §X.32[2], write a loop to delete all fruits that you don't like.

[7] (∗3) Design a Container with an interface consisiting of an abstract class as outlined in §3.2.2. Derive a doubly-linked List, a singly-linked Slist, and a Vector, from it. The elements are of some type Elem. Use these containers for a few simple tests. Write a critique of the interface provided by Container from the point of view of a user. Don't forget to discuss error handling.

[8] (∗2.5) Define three independent classes: a doubly-linked List, a singly-linked Slist, and a Vector. For simple traversals, define

```
class Itor {
public:
        virtual Elem∗ first();
virtual Elem∗ next();
};
```

The members return a pointer to an element or the nullptr if there is none. Implement iterators for List, Slist, and Vector as classes derived from Itor. Compare this design to the abstract Container design in §X.32[7].

[9] (∗1) Make a version of §X.32[7] where the element type is a template argument.

[10] (∗2.5) Given an Itor class like the one in §X.32[8], consider how to provide iterators for forwards iteration, backwards iteration, iteration over a container that might change during an iteration, and iteration over an immutable container. Organize this set of iterators so that a user can interchangeably use iterators that provide sufficient functionality for an algorithm. Minimize replication of effort in the implementation of the containers. What other kinds of iterators might a user need? List the strengths and weaknesses of your approach.

[11] (∗1) Make a version of §X.32[8] where the element type is a template argument.

[12] (∗3) Design a "truly object-oriented" container holding Object∗s as elements A Container (along the lines of §X.32[7]) is itself derived (directly or indirectly) from Object. Implement a List, Slist, and Vector as outlined in §X.32[7] and test them as in §X.32[7].

[13] (∗2.5) Generate 10,000 uniformly distributed random numbers in the range 0 to 1,023 and store them in

(a) an standard library vector<int>,

(b) a Vector<int> from §X.32[7],

(c) a Vector<int> from §X.32[8].

(d) a Vector from §X.32[12].

In each case, calculate the arithmetic mean of the elements of the vector (as if you didn't know it already). Time the resulting loops. Estimate, measure, and compare the memory consumption for the three styles of vectors.

[14] (∗2) Write a template that implements a container with the same member functions and member types as the standard vector for an existing (nonstandard, non-student-exercise) container type. Do not modify the (pre)existing container type. How would you deal with

functionality offered by the nonstandard vector but not by the standard vector?

[15] (∗1.5) What is seriously wrong with this function?:

```
void duplicate(vector<string& v)      // not the way to do it
{
        for (auto p = v.begin(); p!=v.end(); ++v)
                v.insert(p,∗p);
}
```

Outline the possible behavior of duplicate_elements() for a vector<string> with the three elements don't do this.

[16] (∗2.5) Understand the ‘‘Big-O’’ notation (§31.3). Do some measurements of operations on standard containers to determine the constant factors involved.

[17] (∗2) Many phone numbers don't fit into a long. Write a phone_number type and a class that provides a set of useful operations on a container of phone_numbers.

[18] (∗2) Write a program that lists the distinct words in a file in alphabetical order. Make two versions: one in which a word is simply a whitespace-separated sequence of characters and one in which a word is a sequence of letters separated by any sequence of non-letters.

[19] (∗2.5) Implement a simple solitaire card game.

[20] (∗1.5) Implement a simple test of whether a word is a palindrome (that is, if its representation is symmetric; examples are ada, otto, and tut). Implement a simple test of whether an integer is a palindrome. Implement a simple test of whether a sentence is a palindrome. Generalize.

[21] (∗1.5) Define a queue using (only) two stacks.

[22] (∗1.5) Define a stack similar to stack (§31.3.6), except that it doesn't copy its underlying container and that it allows iteration over its elements.

[23] (∗3) Your computer will have support for concurrent activities through the concept of a thread, task, or process. Figure out how that is done. The concurrency mechanism will have a concept of locking to prevent two tasks accessing the same memory simultaneously. Use the machine's locking mechanism to implement a lock class.

[24] (∗2.5) Read a sequence of dates such as Dec85, Dec50, Jan76, etc., from input and then output them so that later dates come first. The format of a date is a three-letter month followed by a two-digit year. Assume that all the years are from the same century.

[25] (∗2.5) Generalize the input format for dates to allow dates such as Dec1985, 12/3/1990, (Dec,30,1950), 3/6/2001, etc. Modify exercise §X.32[24] to cope with the new formats.

[26] (∗1.5) Use a bitset to print the binary values of some numbers, including 0, 1, −1, 18, −18, and the largest positive int.

[27] (∗1.5) Use bitset to represent which students in a class were present on a given day. Read the bitsets for a series of 12 days and determine who was present every day. Determine which students were present at least 8 days.

[28] (∗2) Implement and test a doubly-linked list so that an empty list takes up only the space for a Link∗.

[29] (∗1.5) Write a List of pointers that deletes the objects pointed to when it itself is destroyed or if the element is removed from the List.

[30] (∗1.5) Given a stack object, print its elements in order (without changing the value of the stack).

[31] (∗2.5) Implement and test a list in the style of the standard **list**.

[32] (∗2) Sometimes, the space overhead of a **list** can be a problem. Write and test a singly-linked list in the style of a standard container.

[33] (∗2.5) Implement a list that is like a standard **list**, except that it supports subscripting. Compare the cost of subscripting for a variety of lists to the cost of subscripting a **vector** of the same length.

[34] (∗2) Implement a template function that merges two containers.

[35] (∗1.5) Given a C-style string, determine whether it is a palindrome. Determine whether an initial sequence of at least three words in the string is a palindrome.

[36] (∗2) Read a sequence of **(name,value)** pairs and produce a sorted list of **(name,total,mean,median)** 4-tuples. Print that list.

[37] (∗2.5) Determine the space overhead of each of the standard containers on your implementation.

[38] (∗3.5) Consider what would be a reasonable implementation strategy for a **hash_map** that needed to use minimal space. Consider what would be a reasonable implementation strategy for a **hash_map** that needed to use minimal lookup time. In each case, consider what operations you might omit so as to get closer to the ideal (no space overhead and no lookup overhead, respectively). Hint: There is an enormous literature on hash tables.

[39] (∗2) Devise a strategy for dealing with overflow in **hash_map** (different values hashing to the same hash value) that makes **equal_range()** trivial to implement.

[40] (∗2.5) Estimate the space overhead of a **hash_map** and then measure it. Compare the estimate to the measurements. Compare the space overhead of your **hash_map** and your implementation's **map**.

[41] (∗2.5) Profile your **hash_map** to see where the time is spent. Do the same for your implementation's **map** and a widely-distributed **hash_map**.

[42] (∗2.5) Implement a **hash_map** based on a **vector<map<K,V>∗>** so that each **map** holds all keys that have the same hash value.

[43] (∗3) Implement a **hash_map** using Splay trees (see D. Sleator and R. E. Tarjan: *Self-Adjusting Binary Search Trees*, JACM, Vol. 32. 1985).

[44] (∗2) Given a data structure describing a string-like entity:

```
struct St {
        int size;
        char type_indicator;
        char* buf;              // point to size characters
        St(const char* p);      // allocate and fill buf
};
```

Create 1000 **St**s and use them as keys for a **hash_map**. Devise a program to measure the performance of the **hash_map**. Write a hash function (a **hash<>**; §31.4.3.4) specifically for **St** keys.

[45] (∗2) Give at least four different ways of removing the **erased** elements from a **hash_map**. You should use a standard library algorithm (§3.4.2, Chapter 32) to avoid an explicit loop.

[46] (∗3) Implement a **hash_map** that erases elements immediately.

[47] (∗2) Give an example of when it might be wise to ignore part of a key and write a hash function that computes its value based only on the part of a key considered relevant.

[48] (∗3) Given some implementation of **hash_map**, implement **hash_multimap**, **hash_set**, and **hash_multiset**.

[49] (∗2.5) Write a hash function intended to map uniformly distributed **int** values into hash values intended for a table size of about 1024. Given that function, devise a set of 1024 key values, all of which map to the same value.

X.33 STL Algorithms

The solutions to several exercises for this chapter can be found by looking at the source text of an implementation of the standard library. Do yourself a favor: try to find your own solutions before looking to see how your library implementer approached the problems.

[1] (∗2) Understand the Big-**O** notation. Give a realistic example in which an **O(N∗N)** algorithm is faster than an **O(N)** algorithm for some **N>10**.

[2] (∗1) Write an algorithm **match()** that is like **mismatch()**, except that it returns iterators to the first corresponding pair that matches the predicate.

[3] (∗1.5) Implement and test **Print_name** from §32.4.1.

[4] (∗1) Sort a **list** using only standard library algorithms.

[5] (∗2.5) Define a class **Iseq** so that **Iseq(p,q)** can be used for a pair of iterators (**p**, **q**) that represent a input sequence. Test is for built-in arrays, **istream**s, and **vector**s . Define a suitable set of overloads for the nonmodifying standard algorithms (§32.4) for **Iseq**s. Discuss how best to avoid ambiguities and an explosion in the number of template functions.

[6] (∗2) Define a clas **Iseq** to complement **Iseq**. The output sequence given as the argument to **Oseq** should be replaced by the output produced by an algorithm using it. Define a suitable set of overloads for at least three standard algorithms of your choice.

[7] (∗1.5) Produce a **vector** of squares of numbers 1 through 100. Print a table of squares. Take the square root of the elements of that **vector** and print the resulting vector.

[8] (∗2) Write a set of functional objects that do bitwise logical operations on their operands. Test these objects on vectors of **char**, **int**, and **bitset<67>**.

[9] (∗1) Write a **binder3()** that binds the second and third arguments of a three-argument function to produce a unary predicate. Give an example where **binder3()** is a useful function.

[10] (∗1.5) Write a small program that that removes adjacent repeated words from from a file file. Hint: The program should remove a **that**, a **from**, and a **file** from the previous statement.

[11] (∗2.5) Define a format for records of references to papers and books kept in a file. Write a program that can write out records from the file identified by year of publication, name of author, keyword in title, or name of publisher. The user should be able to request that the output be sorted according to similar criteria.

[12] (∗2) Implement a **move()** algorithm in the style of **copy()** in such a way that the input and output sequences can overlap. Be reasonably efficient when given random-access iterators as arguments.

[13] (∗1.5) Produce all anagrams of the word **food**. That is, all four-letter combinations of the letters **f**, **o**, **o**, and **d**. Do not generate duplicates. Generalize this program to take a word as input and produce anagrams of that word.

[14] (∗1.5) Write a program that produces anagrams of sentences; that is, a program that produces all permutations of the words in the sentences (rather than permutations of the letters in the words).

[15] (∗1.5) Implement `find_if()` (§32.4.4) and then implement `find()` using `find_if()`. Find a way of doing this so that the two functions do not need different names.

[16] (∗2) Implement `search()` (§32.4.6). Provide an optimized version for random-access iterators.

[17] (∗2) Take a sort algorithm (such as `sort()` from your standard library or the Shell sort from §23.5) and insert code so that it prints out the sequence being sorted after each swap of elements.

[18] (∗2) There is no `sort()` for bidirectional iterators. The conjecture is that copying to a vector and then sorting is faster than sorting a sequence using bidirectional iterators. Implement a general sort for bidirectional iterators and test the conjecture.

[19] (∗2.5) Imagine that you keep records for a group of sports fishermen. For each catch, keep a record of species, length, weight, date of catch, name of fisherman, etc. Sort and print the records according to a variety of criteria. Hint: `inplace_merge()`.

[20] (∗2) Create lists of students taking Math, English, French, and Biology. Pick about 20 names for each class out of a set of 40 names. List students who take both Math and English. List students who take French but not Biology or Math. List students who do not take a science course. List students who take French and Math but neither English nor Biology.

[21] (∗1.5) Write a `remove()` function that actually removes elements from a container.

X.34 STL Iterators

[1] Write an output iterator, `Sink`, that doesn’t actually write anywhere. When can `Sink` be useful?

[2] (∗2) Implement `reverse_iterator` (§33.2.1).

[3] (∗1.5) Implement `ostream_iterator` (§38.5).

[4] (∗2) Implement `istream_iterator` (§38.5).

[5] (∗2.5) Design, implement, and test a run-time range-checked random-access iterator `Checked_iter`.

[6] (∗2) Design and implement a handle class that can act as a proxy for a container by providing a complete container interface to its users. Its implementation should consist of a pointer to a container plus implementations of container operations that do range checking.

X.35 Memory and Resources

[1] (∗2.5) Complete and test a pool allocator for objects of type `T`:

```
template<typename T>
class Pool_alloc {
        T* allocate(size_t n);              // allocate space for n objects of type T; do not initialize
        void deallocate(T* p, size_t n);        // deallocate space for n ojects of type T starting at p
        // ...
};
```

Make sure that Pool_alloc provides all of the facilities of the standard library allocator (§34.4). Compare the performance of std::allocator and Pool_alloc to see if there is any reason to use a Pool_alloc on your system.

X.36 Utilities

[1] ???

X.37 String

The solutions to several exercises for this chapter can be found by looking at the source text of an implementation of the standard library. Do yourself a favor: try to find your own solutions before looking to see how your library implementer approached the problems.

[1] (∗2) Write a function that takes two strings and returns a string that is the concatenation of the strings with a dot in the middle. For example, given file and write, the function returns file.write. Do the same exercise with C-style strings using only C facilities such as malloc() and strlen(). Compare the two functions. What are reasonable criteria for a comparison?

[2] (∗2) Make a list of differences between vector and basic_string. Which differences are important?

[3] (∗2) The string facilities are not perfectly regular. For example, you can assign a char to a string, but you cannot initialize a string with a char. Make a list of such irregularities. Which could have been eliminated without complicating the use of strings? What other irregularities would this introduce?

[4] (∗1.5) Class basic_string has a lot of members. Which could be made nonmember functions without loss of efficiency or notational convenience?

[5] (∗1.5) Write a version of back_inserter() (§33.2.2) that works for basic_string.

[6] (∗2) Complete Basic_substring from §36.3.8 and integrate it with a String type that overloads () to mean ‘‘substring of’’ and otherwise acts like string.

[7] (∗2.5) Write a find() function that finds the first match for a simple regular expression in a string. Use ? to mean ‘‘any character,’’ * to mean any number of characters not matching the next part of the regular expression, and [abc] to mean any character from the set specified between the square braces (here a, b, and c). Other characters match themselves. For example, find(s,"name:") returns a pointer to the first occurrence of name: in s; find(s,"[nN]ame:") returns a pointer to the first occurrence of name: or Name: in s; and find(s,"[nN]ame(*)") returns a pointer to the first occurence of Name or name followed by a (possibly empty) parenthesized sequence of characters in s.

[8] (∗2.5) What operations do you find missing from the simple regular expression function from §X.37[7]? Specify and add them. Compare the expressiveness of your regular expression matcher to that of a widely distributed one. Compare the performance of your regular expression matcher to that of a widely distributed one.

[9] (∗2.5) Use a regular expression library to implement pattern-matching operations on a **String** class that has an associated **Substring** class.

[10] (∗2.5) Consider writing an ‘‘ideal’’ class for general text processing. Call it **Text**. What facilities should it have? What implementation constraints and overheads are imposed by your set of ‘‘ideal’’ facilities?

[11] (∗1.5) Define a set of overloaded versions for **isalpha()**, **isdigit()**, etc., so that these functions work correctly for **char**, **unsigned char**, and **signed char**.

[12] (∗2.5) Write a **String** class optimized for strings having no more than eight characters. Compare its performance to that of the **String** from §19.3 and your implementation’s version of the standard library **string**.

[13] (∗2) Measure the performance of copying of **string**s. Does your implementation’s implementation of **string** adequately optimize copying?

[14] (∗2.5) Imagine that reading medium-long strings (most are 5 to 25 characters long) from **cin** is the bottleneck in your system. Write an input function that reads such strings as fast as you can think of. You can choose the interface to that function to optimize for speed rather than for convenience. Compare the result to your implementation’s **>>** for **string**s.

[15] (∗1.5) Write a function **itos(int)** that returns a **string** representing its **int** argument.

X.38 Regular Expressions

[1] (∗2) Write a program that finds dates in a file. Write out each line containing a data preceeded by its line number using the format: **number: line**. Start with a simple format, such as **12/24/2024**, and then add patterns to recognize more formats.

[2] (∗2) Modify §X.38[1] to take a pattern as input and write lines that match that pattern. You might call this program ‘‘grep.’’

[3] (∗2) Rewrite the program from §X.38[1] to produce an output file that is identical to the input file exept that all date are printed in the ISO standard format, such as **2024/12/24**.

[4] (∗2) Modify §X.38[3] to validate dates. Write out an error for each line that has a date that is not valid after a set of criteria of your choice.

[5] (∗1.5) Describe a pattern that cannot be expressed as a regular expression.

X.39 I/O Streams

[1] (∗1.5) Read a file of floating-point numbers, make complex numbers out of pairs of numbers read, and write out the complex numbers.

[2] (∗1.5) Define a type **Name_and_address**. Define **<<** and **>>** for it. Copy a stream of **Name_and_address** objects.

[3] (∗2.5) Copy a stream of **Name_and_address** objects in which you have inserted as many errors as you can think of (e.g., format errors and premature end of string). Handle these

errors in a way that ensures that the copy function reads most of the correctly formatted Name_and_addresses, even when the input is completely messed up.

[4] (∗2.5) Redefine the I/O format of Name_and_address to make it more robust in the presence of format errors.

[5] (∗2.5) Design some functions for requesting and reading information of various types. Ideas: integer, floating-point number, file name, mail address, date, personal information, etc. Try to make them foolproof.

[6] (∗1.5) Write a program that prints (a) all lowercase letters, (b) all letters, (c) all letters and digits, (d) all characters that may appear in a C++ identifier on your system, (e) all punctuation characters, (f) the integer value of all control characters, (g) all whitespace characters, (h) the integer value of all whitespace characters, and finally (i) all printing characters.

[7] (∗2) Read a sequence of lines of text into a fixed-sized character buffer. Remove all whitespace characters and replace each alphanumeric character with the next character in the alphabet (replace z by a and 9 by 0). Write out the resulting line.

[8] (∗3) Write a ‘‘miniature’’ stream I/O system that provides classes istream, ostream, ifstream, ofstream providing functions such as operator<<() and operator>>() for integers and operations such as open() and close() for files.

[9] (∗4) Implement the C standard I/O library (<stdio.h>) using the C++ standard I/O library (<iostream>).

[10] (∗4) Implement the C++ standard I/O library (<iostream>) using the C standard I/O library (<stdio.h>).

[11] (∗4) Implement the C and C++ libraries so that they can be used simultaneously.

[12] (∗2) Implement a class for which [] is overloaded to implement random reading of characters from a file.

[13] (∗3) Repeat §X.39[12] but make [] useful for both reading and writing. Hint: Make [] return an object of a ‘‘descriptor type’’ for which assignment means ‘‘assign through descriptor to file’’ and implicit conversion to char ‘‘means read from file through descriptor.’’

[14] (∗2) Repeat §X.39[13] but let [] index objects of arbitrary types, not just characters.

[15] (∗3.5) Implement versions of istream and ostream that read and write numbers in their binary form rather than converting them into a character representation. Discuss the advantages and disadvantages of this approach compared to the character-based approach.

[16] (∗3.5) Design and implement a pattern-matching input operation. Use printf-style format strings to specify a pattern. It should be possible to try out several patterns against some input to find the actual format. One might derive a pattern-matching input class from istream.

[17] (∗4) Invent (and implement) a much better kind of pattern for pattern matching. Be specific about what is better about it.

[18] (∗2) Define an output manipulator based that takes two arguments – a base and an int value – and outputs the integer in the representation specified by the base. For example, based(2,9) should print 1001.

[19] (∗2) Write manipulators that turn character echoing on and off.

[20] (∗2) Implement Bound_form from §38.4.5.3 for the usual set of built-in types.

[21] (∗2) Re-implement Bound_form from §38.4.5.3 so that an output operation never overflows its width(). It should be possible for a programmer to ensure that output is never quietly truncated beyond its specified precision.

[22] (∗3) Implement an encrypt(k) manipulator that ensures that output on its ostream is encrypted using the key k. Provide a similar decrypt(k) manipulator for an istream. Provide the means for turning the encryption off for a stream so that further I/O is cleartext.

[23] (∗2) Trace a character's route through your system from the keyboard to the screen for a simple:

```
char c;
cin >> c;
cout << c << endl;
```

[24] (∗2.5) There is a standard way of reading, writing, and representing dates under control of a locale. Find it in the documentation of your implementation and write a small program that reads and writes dates using this mechanism. Hint: struct tm.

[25] (∗2.5) Define an ostream called ostrstream that can be attached to an array of characters (a C-style string) in a way similar to the way ostringstream is attached to a string. However, do not copy the array into or out of the ostrstream. The ostrstream should simply provide a way of writing to its array argument. It might be used for in-memory formatting like this:

```
char buf[message_size];
ostrstream ost(buf,message_size);
do_something(arguments,ost);          // output to buf through ost
cout << buf;                          // ost adds terminating 0
```

An operation such as do_something() can write to the stream ost, pass ost on to its suboperations, etc., using the standard output operations. There is no need to check for overflow because ost knows its size and will go into fail() state when it is full. Finally, a display() operation can write the message to a "real" output stream. This technique can be most useful for coping with cases in which the final display operation involves writing to something more complicated than a traditional line-oriented output device. For example, the text from ost could be placed in a fixed-sized area somewhere on a screen. Similarly, define class istrstream as an input string stream reading from a zero-terminated string of characters. Interpret the terminating zero character as end-of-file. These strstreams were part of the original streams library and can often be found in <strstream.h>.

X.40 Locales

[1] (∗2.5) Define a Season_io (§39.3.2) for a language other than American English.

[2] (∗2) Define a Season_io (§39.3.2) class that takes a set of name strings as a constructor argument so that Season names for different locales can be represented as objects of this class.

[3] (∗3) Write a collate<char>::compare() that gives dictionary order. Preferably, do this for a language, such as German or French, that has more letters in its alphabet than English does.

[4] (∗2) Write a program that reads and writes bools as numbers, as English words, and as words in another language of your choice.

[5] (∗2.5) Define a Time type for representing time of day. Define a Date_and_time type by using Time and a Date type. Discuss the pros and cons of this approach compared to the Date from (§39.4.4). Implement locale-sensitive I/O for Time and Date_and_time.

[6] (∗2.5) Design and implement a postal code (zip code) facet. Implement it for at least two countries with dissimilar conventions for writing addresses. For example: NJ 07932 and CB21QA .

[7] (∗2.5) Design and implement a phone number facet. Implement it for at least two countries with dissimilar conventions for writing phone numbers. For example, (973) 360−8000 and 1223 343000.

[8] (∗2.5) Experiment to find out what input and output formats your implementation uses for date information.

[9] (∗2.5) Define a get_time() that ‘‘guesses’’ about the meaning of ambiguous dates, such as 12/5/1995, but still rejects all or almost all mistakes. Be precise about what ‘‘guesses’’ are accepted, and discuss the likelihood of a mistake.

[10] (∗2) Make a list of the locales supported on your system.

[11] (∗2.5) Figure out where named locales are stored on your system. If you have access to the part of the system where locales are stored, make a new named locale. Be very careful not to break existing locales.

[12] (∗2) Compare the two Season_io implementations (§39.3.2 and §39.4.7.1).

[13] (∗2.5) Implement I/O of Roman numerals (such as XI and MDCLII).

[14] (∗2.5) Implement and test Cvt_to_upper (§39.4.6).

[15] (∗2.5) Use clock() to determine average cost of (1) a function call, (2) a virtual function call, (3) reading a char, (4) reading a 1-digit int, (5) reading a 5-digit int, (6) reading a 5-digit double, (7) a 1-character string, (8) a 5-character string,and (9) a 40-character string. (∗6.5) Learn another natural language.

X.41 Numerics

[1] (∗1.5) Write a function that behaves like apply() from §40.5.3, except that it is a nonmember function and accepts function objects.

[2] (∗1.5) Write a function that behaves like apply() from §40.5.3, except that it is a nonmember function, accepts function objects, and modifies its valarray argument.

[3] (∗2) Complete Slice_iter (§40.5.4). Take special care when defining the destructor.

[4] (∗1.5) Rewrite the functions in §29.3.3 using accumulate().

[5] (∗2) Implement I/O operators << and >> for valarray. Implement a get_array() function that creates a valarray of a size specified as part of the input itself.

[6] (∗2.5) Define and implement a three-dimensional matrix with suitable operations.

[7] (∗2.5) Define and implement an n−dimensional matrix with suitable operations.

[8] (∗2.5) Implement a valarray-like class and implement + and ∗ for it. Compare its performance to the performance of your C++ implementation’s valarray. Hint: Include x=0.5(x+y)+z among your test cases and try it with a variety of sizes for the vectors x, y, and z.

[9] (∗3) Implement a Fortran-style array Fort_array where indices start from 1 rather than 0.

[10] (∗3) Implement Matrix using a valarray member as the representation of the elements (rather than a pointer or a reference to a valarray).

[11] (∗2) Generalize the idea from the program in §40.7 into a function that, given a generator as an argument, prints a simple graphical representation of its distribution that can be used as a

crude visual check of the generator’s correctness.

[12] (∗1) If n is an int, what is the distribution of (double(rand())/RAND_MAX)∗n?

[13] (∗2.5) Plot points in a square output area. The coordinate pairs for the points should be generated by Urand(N), where N is the number of pixels on a side of the output area. What does the output tell you about the distribution of numbers generated by Urand?

[14] (∗2) Implement a Normal distribution generator, Nrand.

X.42 Concurrency

[1] ???

X.43 Threads and Tasks

[1] ???

X.44 The C Standard Library

[1] ???

X.45 Compatibility

[1] (∗2.5) Take a C program and convert it to a C++ program; list the kinds of non-C++ constructs used and determine if they are valid ANSI C constructs. First convert the program to strict ANSI C (adding prototypes, etc.), then to C++. Estimate the time it would take to convert a 100,000 line C program to C++.

[2] (∗2) Without looking in the book, write down as many C++ keywords as you can.

[3] (∗2) Write a standards-conforming C++ program containing a sequence of at least ten different consecutive keywords not separated by identifiers, operators, punctuation characters, etc.

[4] (∗2.5) Write a program to help convert C programs to C++ by renaming variables that are C++ keywords, replacing calls of malloc() by uses of new, etc. Hint: don’t try to do a perfect job.

[5] (∗2) Replace all uses of malloc() in a C-style C++ program (maybe a recently converted C program) to uses of new.

[6] (∗2.5) Minimize the use of macros, global variables, uninitialized variables, and casts in a C-style C++ program (maybe a recently converted C program).

[7] (∗3) Take a C++ program that is the result of a crude conversion from C and critique it as a C++ program considering locality of information, abstraction, readability, extensibility, and potential for reuse of parts. Make one significant change to the program based on that critique.

[8] (∗2) Take a small (say, 500 line) C++ program and convert it to C. Compare the original with the result for size and probable maintainability.

[9] (∗3) Write a small set of test programs to determine whether a C++ implementation has ‘‘the latest’’ standard features. Checking for keywords is easy. For example, is `constexpr` recognized? Are delegating constructors supported? Can you inherit constructors?

[10] (∗2.5) Take a C++ program that use `<X.h>` headers and convert it to using `<X>` and `<cX>` headers. Minimize the use of *using-directive*s.