COMP 345

Advanced Program Design with C++

Fall 2017

Lecture Notes

Peter Grogono • René Witte

Department of Computer Science
and Software Engineering

UNIVERSITÉ
Concordia
UNIVERSITY

Lecture notes originally developed by Peter Grogono for COMP 446

Contents

1 Getting Started 1
1.1 Hello, world! 2
1.2 Compiling C++ 6
1.2.1 The compilation process 6
1.2.2 The Preprocessor 8
1.2.3 Compiling in practice 8
1.3 Hello, world! – The Details 9
1.4 Namespaces 11
1.5 Strings 12
1.6 A Pretty Frame 14

2 Testing and Looping 19
2.1 Conditions 19
2.2 Conditional Expressions 22
2.3 Loops 23
2.4 Example: Computing the Frame 24
2.5 Counting 27
2.6 Loop Design 29
2.6.1 Counting Lines 30
2.6.2 Finding Roots by Bisection 33
2.7 Assertions 34
2.8 Order Notation 37
2.9 Example: Maximum Subsequence 39

3 Batches of Data and the STL 43
3.1 Lvalues and Rvalues 43
3.2 Functions 44
3.3 Working with files 47
3.3.1 Writing to a file 49
3.3.2 Stream States 49
3.3.3 Summary of file streams 51
3.4 End-of-file Handling 52
3.5 Storing Data: STL Containers 56
3.6 Summary of the Standard Template Library (STL) 59

4 Application: Grading a class 63
4.1 Problem Statement 63
4.2 First version 63
4.3 Program Structure 72
4.3.1 Header Files 72
4.3.2 Implementation Files 74
4.4 Structuring the grading program 75
4.4.1 Translation unit Student 75
4.4.2 Translation unit median 75
4.4.3 Translation unit for the main program 76
4.5 Dependencies 78

4.6 Strings and characters 81

5 Pointers, Iterators, and Memory Management 83
5.1 Pointers 83
5.1.1 Pointer Arithmetic 84
5.1.2 Pointer types vs. reference types 85
5.1.3 A note on null pointers 86
5.1.4 Null pointers in C++11 87
5.2 Dynamic Memory Allocation 87
5.2.1 Stack Allocation 87
5.2.2 Heap Allocation 89
5.3 Iterators 90
5.3.1 Kinds of iterator 91
5.3.2 Using iterators 93
5.3.3 Range Functions 96

6 Designing classes 99
6.1 Constructors 102
6.2 Destructors 107
6.3 Operators 108
6.3.1 Assignment (`operator=`) 110
6.3.2 Arithmetic 111
6.3.3 Comparison 114
6.3.4 Input and output 115
6.4 Conversions 117
6.5 Accessors 119
6.6 Mutators 119
6.7 Odds and ends 121
6.7.1 Indexing (`operator[]`) 121
6.7.2 Calling (`operator()`) 121
6.7.3 Explicit constructors 122
6.7.4 Friends 125
6.8 Summary 126

7 Templates and Generic Programming 131
7.1 Template Functions 131
7.1.1 Type Parameters 131
7.1.2 Missing Functions 135
7.1.3 Conversion Failure 136
7.1.4 Non-type Parameters 137
7.2 Template Classes 138
7.2.1 A Template Class for Coordinates 139
7.2.2 `class` or `typename`? 141
7.3 Compiling Template Code 142
7.4 Template Return Types 143
7.5 Template Specialization 144
7.6 Template Metaprogramming 146
7.7 Default Arguments 147

8 Inheritance and Polymorphism **149**

8.1 What is inheritance? 149

8.1.1 Constructors and destructors 151

8.1.2 Slicing 152

8.2 Designing a base class 154

8.2.1 `virtual` functions 154

8.2.2 Don't redefine non-virtual functions 157

8.2.3 Virtual methods in C++11: `override` and `final` 159

8.2.4 `protected` attributes 159

8.2.5 Example: A base class for bankers 160

8.3 Designing a derived class 161

8.3.1 Example: A derived class for bankers 163

8.3.2 Additional base class functions 168

8.3.3 Dominance 168

8.4 Conversion and Casting 169

8.4.1 Implicit Conversion 169

8.4.2 C-style Casting 169

8.4.3 Modern C++ Casting 170

8.5 Advanced inheritance topics 174

8.5.1 Private Inheritance 174

8.5.2 Multiple Inheritance 175

9 Template Programming **177**

9.1 Designing a template class 177

9.1.1 Iterators 179

9.1.2 Expanding Vectors 181

9.1.3 Refactoring 183

9.2 A Note on Iterators 184

9.3 Templates vs. Inheritance: Tree traversal 187

9.3.1 Organizing the code 195

9.4 The Curiously Recurring Template Pattern 196

10 When things go wrong **201**

10.1 Simple Mechanisms 202

10.1.1 Error messages 202

10.1.2 Error codes 202

10.1.3 Special return values 203

10.2 Exceptions 204

10.2.1 Exception handling syntax 206

10.2.2 Things to know about exceptions 206

10.2.3 You didn't expect that 209

10.2.4 How exceptions work 210

10.3 Exception-safe Coding 211

10.4 Resource Acquisition Is Initialization (RAII) 213

10.4.1 Autopointers 213

10.4.2 Autopointers in C++11 218

10.5 Coding Policies 218

10.5.1 Cowboy coding 219

10.5.2 Pessimistic coding . . . 219
10.5.3 Defensive coding . . . 219
10.5.4 Contractual coding . . . 219
10.6 Static assertions . . . 220
10.7 Summary . . . 222

11 Parallel and Distributed Programming 223
11.1 Foundations . . . 223
11.1.1 Thread-Safe Programming . . . 224
11.2 The POSIX Thread Library . . . 225
11.2.1 Encapsulating Threads . . . 227
11.2.2 A Better `Lock` . . . 232
11.3 Multithreading in C++11 . . . 232
11.3.1 Hello, C++11 Thread! . . . 232
11.3.2 Working with threads . . . 234
11.3.3 Sharing Data between Threads . . . 236
11.3.4 Communication between Threads . . . 238
11.3.5 The Keyword `volatile` . . . 243
11.3.6 Constant and Mutable . . . 244
11.3.7 Promising the Future . . . 246
11.4 Distributed Computing . . . 247
11.4.1 MapReduce . . . 248
11.4.2 MapReduce Concept . . . 248
11.4.3 MapReduce Implementations . . . 249
11.4.4 Hello, MapReduce! . . . 251

12 System Design 253
12.1 Logical and physical design . . . 254
12.2 Linkage . . . 255
12.3 Namespaces . . . 256
12.4 Cohesion . . . 256
12.5 Coupling . . . 258
12.5.1 Encapsulation . . . 258
12.5.2 Hidden coupling . . . 261
12.5.3 Compilation dependencies . . . 261
12.5.4 Cyclic Dependencies . . . 265
12.5.5 Inlining . . . 266
12.6 The Pimpl Idiom . . . 267
12.6.1 Introducing Pimpl . . . 269
12.6.2 Combining Pimpl and autopointers . . . 272
12.7 Refactoring . . . 276
12.7.1 Simplify Calling Patterns . . . 277
12.7.2 Introduce Design Patterns . . . 278
12.7.3 Encapsulate . . . 279
12.7.4 Move Common Code to the Root . . . 279
12.7.5 Global Names . . . 279
12.8 Miscellaneous Techniques . . . 279
12.8.1 CRC Cards . . . 280

12.8.2 DRY 280
12.8.3 YAGNI/KISS 280

13 Using Design Patterns 283
13.1 Singleton 283
13.2 Composite 285
13.3 Visitor 289
13.4 State 293
13.5 Bridge 298
13.6 Command 300
13.7 Observer 300
13.8 Pattern Languages 307

14 But wait, there's more! 309
14.1 Policies 309
14.1.1 Hello, Policy! 309
14.1.2 Policies in LOKI 311
14.1.3 Policies in the STL 313
14.2 Traits 313
14.2.1 Traits in the STL 313
14.2.2 Traits in Boost 315
14.3 Metaprogramming 317
14.3.1 MPL 317
14.3.2 Blitz++ 319
14.4 Functional programming 320
14.4.1 Lambdas 321
14.4.2 Closures 322
14.5 Envoi 323

A Hic Sunt Dracones 327

B Coding Standards 329
B.1 Rules 330
B.2 Layout 331
B.2.1 Line Length 332
B.2.2 Brace Wars 332
B.2.3 Tabs and Blanks 334
B.2.4 Automatic Formatting 335
B.3 Comments 335
B.3.1 Disabling blocks of code 336
B.3.2 Warnings 338
B.4 Names 338
B.4.1 Hungarian Notation 339
B.4.2 Enumerations 340
B.5 Constants and Literals 341
B.6 `const` 341
B.6.1 The keyword `mutable` 342
B.6.2 `const` with pointers 343

B.7 Variables 344
B.8 Expressions 344
B.9 Exceptions 344
B.10 Functions 345
B.10.1 Parameters 345
B.10.2 Operators 345
B.11 Classes 347
B.11.1 Organization 347
B.11.2 Required Methods 348
B.11.3 Initialization 349
B.11.4 Composition and Inheritance 349
B.11.5 Abstract Base Classes 349
B.12 Libraries 349
B.13 Files 350
B.13.1 Documentation 350
B.13.2 Header Files 350
B.13.3 Implementation Files 351
B.13.4 Directory Structure 351
B.14 Development 351
B.14.1 Portability 351
B.14.2 Change management 351
B.14.3 Automation 352
B.15 Style 352

C Glossary 354

References 362

List of Figures

1 Hello.1 3
2 Parsing with Boost/Spirit 6
3 Hello.2 12
4 Hello.3 12
5 Greeting.1 13
6 Greeting.2 15
7 Greeting.3 25
8 A dialogue with Greeting.3 26
9 Fencepost numbering 28
10 There should be a better way. 29
11 Starting from 1 32
12 The cause of the Zune crash 31.12.2008 32
13 Finding zeroes by bisection 33
14 Finding zeroes by bisection 35
15 Part of the complexity hierarchy 39
16 Solving equations 39
17 Maximum subvector: first attempt 40

18 Maximum subvector: second attempt 40
19 Maximum subvector: an efficient solution 41
20 Passing arguments in C++ 45
21 Passing arguments in C++ and Java 46
22 Finding the mean from a file of numbers 48
23 Writing random numbers to a file 49
24 Stream bits and their meanings 50
25 Reading and setting the stream bits 51
26 Output manipulators 52
27 Finding the mean 53
28 Testing end-of-file 54
29 Storing values in a vector 56
30 Complaints from the compiler 58
31 Input for grading program 64
32 Output from grading program 64
33 The main program 65
34 Function `showClass` 65
35 Class `Student` 66
36 Function `process` for class `Student` 67
37 Finding the median 68
38 Comparison functions 69
39 Extractor for class `Student` 70
40 Inserter for class `Student` 71
41 Directives required for the grading program 72
42 `student.h`: header file for class `Student` 76
43 `student.cpp`: implementation file for class `Student`: first part 77
44 `student.cpp`: implementation file for class `Student`: second part 78
45 `median.h`: header file for function `median` 78
46 `median.cpp`: implementation file for function `median` 79
47 Dependencies in the grading program 79
48 `grader.cpp`: implementation file for main program 80
49 Two pointers, one value 84
50 Dangerous tricks 88
51 Subtler dangerous tricks 88
52 Heap allocation and deallocation 90
53 Properties of iterators 91
54 Iterators, Containers, and Algorithms 92
55 Separating passes and failures: first version 94
56 Compiling Figure 55 on page 94 fails 94
57 Separating passes and failures: second version 95
58 Declaration for class `Rational` 100
59 Declaration for class `Account` 101
60 Functions to normalize instances of class `Rational` 104
61 Pointer problems 105
62 Arithmetic assignment operators for class `Rational` 109
63 An ***incorrect*** implementation of `operator=` 110
64 Assignment operator for class `Account` 111

65 Arithmetic operators for class Rational, implemented using the arithmetic assignments of Figure 62 on page 109 . . . 113
66 An exponent function for class Rational . . . 113
67 Comparison operators for class Rational . . . 115
68 Declaration and implementation of class AccountException . . . 120
69 A simple map class . . . 122
70 Test program for class AddressBook (left) and results (right) . . . 122
71 Class AddressBook . . . 123
72 Changing Swap into a template function . . . 132
73 Part of a template class for coordinates . . . 140
74 A more elaborate version of operator+ . . . 142
75 A specialized version of class Coordinate . . . 145
76 Not an "is-a" inheritance hierarchy . . . 150
77 The problem of the flightless penguin . . . 151
78 Recognizing that some birds don't fly . . . 152
79 Constructors and destructors in inheritance hierarchies . . . 153
80 Base and derived objects . . . 154
81 The danger of a non-virtual destructor . . . 157
82 Class Account as a base class . . . 161
83 Accessors for class Account . . . 162
84 Mutators for class Account . . . 162
85 The constructor of a derived class should call the constructor of its base class. 163
86 Class SavingsAccount derived from Account . . . 164
87 Constructors for class SavingsAccount . . . 165
88 Assignment for class SavingsAccount . . . 166
89 Withdrawing from a savings account . . . 166
90 Adding interest . . . 166
91 Inserting a savings account into a stream . . . 167
92 Testing the banker's hierarchy . . . 167
93 Dominance . . . 169
94 A problem with const . . . 172
95 private inheritance . . . 174
96 Pointers for class Vec . . . 178
97 Declaration for class Vec: version 1 . . . 179
98 Constructors and destructor for class Vec . . . 180
99 Assignment operator for class Vec . . . 180
100 Appending data to a Vec . . . 182
101 Implementation of two overloads of create . . . 183
102 Revised definitions of functions that use create . . . 184
103 Declaration for class Vec: version 2 . . . 185
104 An extract from the STL class list . . . 186
105 Pseudocode for binary tree traversal . . . 187
106 Binary tree: class declaration and related functions . . . 189
107 Class ListNode . . . 190
108 Abstract base class Store . . . 190
109 Class Stack . . . 191
110 Class Queue . . . 192
111 Generic traversal using inheritance . . . 193

112 Generic traversal using templates 193
113 Test program and output for generic tree traversal 194
114 Code organization for traversal program 196
115 The Curiously Recurring Template Pattern 196
116 A class that provides `operator>` using `operator<` 197
117 Defining relational operators with templates in the STL 198
118 Base class `Counted` 199
119 Testing the `Counted` hierarchy 200
120 Output from the counting test of Figure 119 on page 200 200
121 Error messages 202
122 The consequence of ambiguous return values 203
123 Catching errors in `y = f(g(h(x)))` 204
124 Rethrowing 208
125 Inheriting exception specifications 209
126 Throwing unexpected exceptions 209
127 Expecting the unexpected 210
128 Constructor for class `Vec` 211
129 Popping a stack: version 1 212
130 Popping a stack: version 2 212
131 Generating a report 214
132 Acquiring a file and a lock 214
133 Buying a car with pointers 215
134 Buying a car with autopointers 216
135 Copying an autopointer 217
136 Fragments of a multithreaded program 225
137 Adding numbers with threads 226
138 Co-operative multithreading with POSIX threads 227
139 A base class for synchronization 228
140 Using RAII to simplify locking 228
141 An account class with internal locking 229
142 The best of both worlds 231
143 A lock with restricted uses 233
144 Hello, C++11 Thread! 234
145 Starting multiple detached threads with parameters 236
146 RAII for joining threads 237
147 Locking a shared data structure through a mutex 238
148 Deferred locking of multiple mutexes to avoid deadlocks 239
149 Interface for a thread-safe queue with condition variables 240
150 Implementation for a thread-safe queue with condition variables 242
151 Is a cache part of an object? 246
152 Futures and Promises 247
153 MapReduce Workflow 250
154 Counting words, MapReduce-style 251
155 Counting words with MapReduce: Map function 252
156 Counting words with MapReduce: Reduce function 252
157 Physical dependency: `c1.h` “reads” `c2.h` 255
158 A class for points: version 1 258
159 A class for points: version 2 259

160 A class for points: version 3 259
161 A class for points: version 4 260
162 A class for points: version 5 261
163 A class for points: version 6 262
164 A class for points: version 7 262
165 A class for points: version 8 262
166 The importance of reducing compile time 263
167 Forward declarations: 1 263
168 Forward declarations: 2 264
169 Forward declarations: 3 264
170 Forward declaration for `ostream`: 1 265
171 Forward declaration for `ostream`: 2 265
172 Inlining 267
173 Test program for class `Account` 268
174 Output from the program of Figure 174 269
175 Header file for original class `Account` 269
176 Implementation for class `Account`: part 1 270
177 Implementation for class `Account`: part 2 271
178 Header file for Pimpl class `Account` 272
179 Implementation file for Pimpl class `Account`: part 1 273
180 Implementation file for Pimpl class `Account`: part 2 274
181 Implementation file for Pimpl class `Account`: part 3 275
182 Output from the Pimpl version of class `Account` 275
183 Header file for autopointer Pimpl class `Account` 276
184 Implementation file for autopointer Pimpl class `Account` 277
185 Declaration for class `Singleton` 284
186 Definitions for class `Singleton` 284
187 A program that tests class `Singleton` 286
188 Declarations for the text class hierarchy 287
189 A class for paragraphs 288
190 Typesetting a `Paragraph` 288
191 A grid of classes and operations 289
192 Choosing operations using `switch-case` 289
193 The base class of the `Visitor` hierarchy 290
194 Adding `accept` to the base class of the `Text` hierarchy 292
195 A class for typesetting derived from `Visitor` 292
196 Implementation of class `setVisitor` 292
197 A class with various states 294
198 State pattern: scanners for white space, numbers, and identifiers 295
199 State pattern: the scanner controller class 296
200 State pattern: using the scanner 297
201 Setting the pointer (Koenig & Moo, page 245) 298
202 Separating interface and implementation (Koenig & Moo, pages 243–244) 299
203 A simple command class 300
204 Using the command class 300
205 `main.cpp` 301
206 Output from the program of Figure 205 on page 301 301
207 `subject.h` 302

208 `subject.cpp` 303
209 `thermometer.h` 303
210 `thermometer.cpp` 304
211 `observer.h` 304
212 `digital.h` 304
213 `digital.cpp` 305
214 `analog.h` 305
215 `analog.cpp` 306
216 Policy classes for "Hello, world" in two languages 310
217 Template "Hello, world" class using a language policy 310
218 Testing the "Hello, world" class with two policies 310
219 Policy classes for dynamic allocation 312
220 Declaration of the standard basic output stream class 314
221 Iterator traits in the STL 314
222 Two implementations of `swap` 316
223 Choosing the version 316
224 Converting binary to decimal at compile-time 318
225 Matrix addition 319
226 Fun with Blitz++ 320
227 Setting tabs in `Code::Blocks` 334
228 Setting tabs in Visual C++ V7 335
229 Using Doxygen to document a function 337
230 Browsing Doxygen documentation 337
231 A dangerous programming style 347
232 A ray tracing program 353

1 Getting Started

1.1 `Hello, world!` 2

1.2 **Compiling C++** 6

1.3 `Hello, world!` – **The Details** 9

1.4 **Namespaces** 11

1.5 **Strings** 12

1.6 **A Pretty Frame** 14

Although this course will cover programming practices in general, the focus will be on C++. Why not Java?

- C++ is more complex than Java. If you can write good programs in C++, you can transfer this skill to Java; the converse is not true.
- From 1995 until around 2010, Java usage increased and C++ usage decreased. Equality was reached some time during 2006, and now Java is used more than C++. Nevertheless, there is still a need for C++ programmers. Moreover, due to the sale of Java to Oracle, programmers increasingly seem to look for alternatives due to the uncertainties surrounding future developement of Java, which lead to renewed interest in C++ adoption since around 2010.
- Many games and most software for the telecommunications industry is written in C++. Companies that use C++ include: Adobe products; Alias/Wavefront products (e.g., Maya); Amazon; Facebook; Google; JPL/NASA; Mozilla (Firefox, Thunderbird etc.), Microsoft, and Siemens. More recently, algorithms for Deep Learning (e.g., Google's TensorFlow) or crypto-currency mining (e.g., Ethereum) have been implemented in C++. For more applications, see http://www.stroustrup.com/applications.html.
- There are many Java jobs and many Java programmers. There are not quite so many C++ jobs but there are very few *good* unemployed C++ programmers. Strong C++ programmers can find interesting and well-paid jobs.

It is a cliché to say that software is becoming ubiquitous. However, it is noteworthy that programs are getting larger:

Entity	Lines of code
Cell phone:	2×10^6
Car:	20×10^6
Telephone exchange:	100×10^6
Civil aircraft:	10^9
Military aircraft:	6×10^9

The crucial problem for all aspects of software development is ***scalability***. Approaches and techniques that do not work for millions of lines of code are not useful. C++ scales well.

1.1 `Hello, world!`

2 C++ is the result of a long history of developing programming languages:[1]

1965: BCPL (Martin Richards)

1968: B (Ken Thompson @ Bell Labs) ($8K \times 18b$ PDP/7)

1969: C (Ken Thompson and Dennis Ritchie) ($64K \times 12b$ PDP/11)

1979: C with classes (Bjarne Stroustrup)

1983: C++ named

1990: templates and exceptions

1992: Microsoft C++ compiler

1993: RTTI, namespaces

1994: ANSI/ISO Draft Standard

1995: Java

1998: **C++98**

2009: C++0x Draft Standard

2011: **C++11** Standard (a.k.a. C11, C1X)

2014: C++14 (minor updates to C++11)

2017(?): C++17 aka C++1z (new features)

3 C++ has strengths:

- Low-level systems programming
- High-level systems programming
- Generic programming
- Embedded code
- High performance programming
- Numeric/scientific computation
- Games programming
- General application programming

and weaknesses:

- Legacy of C
- Insecurities
- Complexity
- No standard GUI library

[1] A numbered box in the outer margin indicates that nearby text is used as a slide during a lecture.

Even the lead designer of C++, Bjarne Stroustrup, does not claim that C++ is the ideal programming language: 4

> *C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg.*

The C Programming Language (Kernighan and Ritchie 1978), the "classic" text for C, appeared in 1978. Its first example program has set the standard for all successors: 5

```
main()
{
    printf("hello, world\n");
}
```

While C++ originated from C, it has evolved considerably and is now a distinct and significantly more complex programming language. A typical C++ version of the "hello, world" program is shown in Figure 1 (Weiss 2004, page 11).

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Hello, world" << endl;
   return 0;
}
```

Figure 1: Hello.1

A modern C++ compiler should compile both programs, because C++ is an extension of C.[2] To be more precise, it was originally the intention that C++ and C should be compatible, but that goal was abandoned when it became clear that C++ would suffer if it was followed rigorously: it is possible to write programs that are legal C but not legal C++. An unfortunate side-effect of this compatibility is that there is now a mixture of C and C++ programming styles, ranging from "classic C" style to "modern C++" style. In this course, we will emphasize the modern style.

The important features of Figure 1 are:

- `#include <iostream>` informs the compiler that the program uses components from the ***stream library*** to perform input and output. (`#include` is similar to Java's `import`.)
- `using namespace std;` a ***using directive*** that allows to use classes from the `std` namespace without a `std::` prefix (similar to the `import` directive in Java).
- `int main()` introduces a function that is usually called the ***main program***. Compilers are quite flexible about the declaration of this function. We will see later, for example, that `main` may take arguments. The initial `int` is important, however: it indicates that

[2] Stroustrup claims that "every example in *The C Programming Language* (2nd Edition) (Kernighan and Ritchie 1978) is also a C++ program." Of course, this does ***not*** imply that every program that a C compiler compiles will also be accepted by a C++ compiler.

the function will return an integer that the operating system may use to choose the next action.

- `cout` is the new version of the old “`printf`” (or `System.out.print` in Java). It is part of the `std` ***namespace*** included above. Because of the `using` directive, we do not have to mention this explicitly with `std::cout`. `cout` (pronounced “see-out”) is the name of the ***standard output stream***; when a program runs, text written to `cout` appears on the console window.
- `<<` is an ***operator***, analogous to + or ×. In this case, its right operand consists of stuff to be written to standard output.
- `endl` is the new version of `\n`: it is short for “end line” and outputs a return character. (We can still use `\n`, however.)
- `return 0` sends zero back to the operating system, indicating that the program terminated successfully.

In general, we also note that:

- Statements in C++ are terminated with a semicolon (“`;`”).
- Blocks of code are delimited by braces (“`{`” and “`}`”).

C has a reputation for being cryptic. The following code for copying a string (Kernighan and
6 Ritchie 1978, page 101) is often quoted:

```
char *s;
char *t;
....
while (*s++ = *t++)
    ;
```

This works for several rather subtle reasons: the post-increment operator `++` has an effect and also returns a value; the assignment operator `=` returns a value; C strings are terminated by the special value ‘`\0`’ (also known as the “null character”); and the null character, interpreted as a boolean, represents `false`. The loop terminates because a C string, by convention, is terminated with a null character (‘`\0`’).

This style of programming was appropriate when C was introduced. Ritchie and Thompson designed C for systems programming on minicomputers of the early 1970s. They did most of their work on a DEC PDP/11 with 64 Kb of RAM and they wanted a small and simple compiler. The `while` statement above can be compiled into efficient code without any further optimization.

By way of contrast, here is the string copy implemented in “C-for-dummies” style. Note the explicit termination test, comparing `s[i-1]` to `'\0'`. The `do-while` loop is used because the terminator must be copied to the output string.

```
int i = 0;
do
{
   s[i] = t[i];
```

```
        i = i + 1;
    } while (s[i-1] != '\0');
```

An experienced C programmer would probably prefer the library call `strcpy(s, t)` to either of these versions.

From a modern perspective, however, the `while` statement is unnecessarily cryptic and potentially ***inefficient*** because it places unnecessary constraints on what the compiler can do. Accurately implementing what the programmer has written requires the use of two registers, separately incremented, and each containing a pointer. At each cycle, the code must test for a null character.

In contrast, a C++ programmer would write something like this: 7

```
    std::string s;
    std::string t;
    ....
    s = t;
```

In this version, strings are instances (objects) of a standard class `string` rather than "pointers to characters". The copying operation is performed by the library function that implements assignment ("`=`"). This function can be written to be optimal for the architecture for which the library is written: for example, it might use "block moves" rather than copying characters one at a time. It might even avoid copying altogether. The code does not depend on any particular representation for strings. For example, it does not require a null terminating character. In general, a C++ programmer will tend to work at a ***higher level of abstraction*** than a traditional C programmer.

Here is another, much more advanced, example of modern C++. Suppose we want to parse strings according to the following grammar: 8

$$\begin{array}{lcl}
\mathit{expr} & = & \mathit{term}\ \text{‘}\texttt{+}\text{’}\ \mathit{expr} \\
 & | & \mathit{term}\ \text{‘}\texttt{-}\text{’}\ \mathit{expr} \\
 & | & \mathit{term} \\
\mathit{term} & = & \mathit{factor}\ \text{‘}\texttt{*}\text{’}\ \mathit{term} \\
 & | & \mathit{factor}\ \text{‘}\texttt{/}\text{’}\ \mathit{term} \\
 & | & \mathit{factor} \\
\mathit{factor} & = & \mathit{integer} \\
 & | & \text{‘}\texttt{(}\text{’}\ \mathit{expr}\ \text{‘}\texttt{)}\text{’}
\end{array}$$

Using the Boost Spirit library (Abrahams and Gurtovy 2005), we can write code corresponding to these productions as shown in Figure 2 on the next page (some additional surrounding code is needed to make everything work). Notice how close the C++ code is to the original grammar. 9

That C++ is compatible with C is both good and bad. It is good because at Bell Labs, where C++ was developed, a new language that could not compile existing code would almost certainly have not been accepted. It is bad because C is an old language with many features that would now be considered undesirable; compatibility meant that C++ had to incorporate most of these undesirable features, even though it does quite a good job of covering them up. A well-trained

```
expr =
      ( term[expr.val = _1] >> ’+’ >> expr[expr.val += _1] )
    | ( term[expr.val = _1] >> ’-’ >> expr[expr.val -= _1] )
    | term[expr.val = _1]
    ;

term =
      ( factor[term.val = _1] >> ’*’ >> term[term.val *= _1] )
    | ( factor[term.val = _1] >> ’/’ >> term[term.val /= _1] )
    | factor[term.val = _1]
    ;

factor =
      integer[factor.val = _1]
    | ( ’(’ >> expr[factor.val = _1] >> ’)’ )
    ;
```

Figure 2: Parsing with Boost/Spirit

and conscientious programmer can write secure and efficient programs in C++; an inexperienced programmer can create a real mess.

Figure 2 illustrates another aspect of modern C++. All competent C programmers are essentially equal: they all have a good understanding of the entire language and its standard libraries. With C++, however, a wide gulf separates programmers who can ***use*** the Spirit parser components — which is straightforward — from the programmers who can ***create*** the Spirit components; these programmers form a small minority of C++ programmers. (Parser generators are constructed by ***template metaprogramming***, which few C++ programmers understand, let alone use.)

1.2 Compiling C++

When choosing a C++ compiler, you have to carefully check with version of the language standard it supports. In particular, for code examples using C++11, you have to make sure that your compiler supports this standard.

1.2.1 The compilation process

Compiling a program consists of a number of steps. Errors may occur at any step, and it is important to distinguish the different kinds of error. The compiler processes the program as a number of ***compilation units***. Each compilation unit corresponds to a source code file together with any other files `#included` with it.

syntax error

1. The compiler parses each compilation unit. This may produce ***syntax errors***.

   If we omit a semicolon in Figure 1 on page 3, writing

   ```
   cout << "Hello, world!" << endl
   ```

the compiler issues a syntax error:[3] 10

```
        f:\Courses\COMP345\src\Hello\hello.cpp(32):
            error C2143: syntax error : missing ';' before '}'
```

The "32" is the number of the line on which the error is detected; in this case, it is the line containing `}`, the line ***following*** the line with the error. However, the error message itself is quite helpful: it actually tells you what is wrong with your program.

2. The compiler checks the semantic correctness of the program. For example, it checks that each function is called with arguments of an appropriate type. A program may have ***semantic errors***.

   semantic error

   If we omit the "`l`" from "`endl`", the compiler issues two semantic errors: 11

   ```
           f:\Courses\COMP345\src\Hello\hello.cpp(31):
               error C2065: 'end' : undeclared identifier
           f:\Courses\COMP345\src\Hello\hello.cpp(31):
               error C2593: 'operator <<' is ambiguous
   ```

   Syntax errors and semantic errors are collectively called ***compile-time errors***.

   compile-time error

3. The compiler generates object code for the compilation unit.

4. When each compilation unit has been controlled, the ***linker*** is invoked. The linker attempts to link all of the object code modules together to form an ***executable*** (or, occasionally, a library). This may produce ***link-time errors***.

   link-time error

   Suppose that we declare and use a function `f` but forget to define it. The compiler expects that `f` will be defined somewhere else and so does not generate a semantic error. We do not get an error until the linker discovers that there is no definition: 12

   ```
   Hello error LNK2019:
       unresolved external symbol "void __cdecl f(int)"
       (?f@@YAXH@Z) referenced in function _main
   ```

   The mysterious string `?f@@YAXH@Z` is the name that the compiler has given to the function `f`. The conversion from `f` to `?f@@YAXH@Z` is called ***name mangling***.

   name mangling

5. When the program has been linked, it can be ***executed*** or, more simply, ***run***. It may run correctly or it may generate ***run-time errors***.

   run-time error

   If you write `cout` twice by mistake, the compiler accepts the program without any fuss:

   ```
           cout << cout << "Hello, world!" << endl;
   ```

   However, when the program runs, it displays something like: 13

   ```
           0045768CHello, world!
   ```

   Output like this can be disconcerting for beginners: the program has displayed the address of the object `cout` in hexadecimal!

[3] The error messages were produced by VC++. Other compilers produce similar, but not identical, diagnostics.

Although you don't really want any errors at all, you should prefer compile-time and link-time errors to run-time errors, if only because your customers will never see them. Fortunately, C++ compilers perform thorough syntactic and semantic checking (much better than C compilers) and will catch many of your errors. 14

An event that occurs during compilation is called static. An event that occurs during execution is called dynamic.

1.2.2 The Preprocessor

#include

C++ inherited from C a particular way of preparing source code for compilation: Before being compiled, source code is transformed by a *preprocessor* that accepts special commands for modifying the source code, like including other code parts, executing macros, or omitting parts of your code (using conditional statements). The `#include` statement is such a directive to the preprocessor. 15

For now, you do not have to worry about the preprocessor, but simply be aware that the code "seen" by the compiler might be rather different from what you see in your editor/IDE. Depending on the development environment used, you can look at the output of the preprocessor (the actual input to the compiler) using a special command, e.g., for GCC you can type

```
g++ -E hello.cpp > hello.out
```

to obtain the results of preprocessing step in the file `hello.out`. For the simple "Hello, world" program in Figure 1 on page 3, you might be surprised to see that the file produced for the compiler contains now almost 30 000 lines of code!

1.2.3 Compiling in practice

16 There are a number of platforms available for C++ development.

gcc The Gnu C Compiler is a command-line compiler, not an IDE (see http://gcc.gnu.org/). It is available for several platforms; for Windows, use the MinGW toolchain (see http://www.mingw.org/). For simple programs with all of the source code in one file, just use "`gcc`" (defaults to C) or "`g++`" (defaults to C++) as a command. For more complex programs, it's best to write a `Makefile` or use an IDE.

Code::Blocks is a free (open-source) cross-platform (Linux, Mac, Windows) IDE for C++ that can be used with various compilers (see http://www.codeblocks.org/). One of the binary download packages for Windows includes MinGW, which is the GNU compiler for Windows.

Code::Blocks is a more friendly and easy-to-use IDE than most of the alternatives listed here. Its main disadvantage is libraries: it comes with the standard C++ libraries but, if you want to use any other libraries, you will have to compile them and install them yourself.

Eclipse CDT The CDT (C/C++ Development Tools) Project (see http://www.eclipse.org/cdt/) provides a fully functional C and C++ Integrated Development Environment (IDE) for the Eclipse platform. This is particularly interesting for those who are already familiar with the Eclipse platform (like Java developers). For Linux users, the *Linux Tools Project* (http://www.eclipse.org/linuxtools/) combines the CDT with a number of other useful tools, like Autotools build integration and the Valgrind memory analysis tool.

NetBeans Like Eclipse, NetBeans (see http://netbeans.org/features/cpp/index.html) also fully supports C++ development. It is available for Windows, Linux, MacOS, and Solaris.

Qt Creator Now maintained by Qt after being de-merged from Digia, it was originally developed by TrollTech, which was later bought by Nokia. It is a full-featured IDE for the cross-platform *Qt* framework for C++ development (see http://qt-project.org).

Visual C++ (VC++) is a popular platform for developing C++ programs. It has some disadvantages:

- Early versions did not support modern C++. In particular, *use VC++ 7.1 or later* for this course; earlier versions cannot handle some of the C++ features that we will be using.
- VC++ tries very hard to wrap your programs in MS junk. It is possible, but difficult, to produce the "vanilla", standard C++ programs that we will use in this course.
- Unlike all the other IDEs mentioned above, it is not free/open source software.

1.3 `Hello, world!` – The Details

Here is the first C++ example program again, this time without the `using` directive: 17

```
#include <iostream>

int main()
{
   std::cout << "Hello, world!" << std::endl;
   return 0;
}
```

"`#include`" is a ***compiler directive***. Strictly speaking, it is not part of the program, but instead is an instruction telling the compiler what to do (see Section 1.2.2 for details). In this case, the directive tells the compiler to include the stream part of the ***standard library***, so that it becomes accessible to the program.

Streams are one of the media that C++ programs use to communicate. The communication is usually external — data is read to or from a device such as a keyboard, screen disk, or network — but may also be internal to the program (that is, to or from memory). A stream is not the same as a file, although a file may be read or written with a stream. streams

An ***output stream*** typically allows the program to send data ***to*** an external device, such as a window. An ***input stream*** typically allows the program to receive data ***from*** an external device, such as a keyboard. The library component `iostream` provides both: "`i`" for input and "`o`" for output.

The program itself consists of a single ***function definition***. The function's name is `main` and its ***return type*** is `int`. The word `int` is a ***keyword*** of C++, meaning that we cannot use it for anything other than its intended application. In fact, `int` is a ***type*** – the type of ***signed integers***. A value i of type `int` is typically represented using 32 bits and satisfies $-2^{31} \le i < 2^{31}$. Many modern C++ systems provide 64-bit `int`s.

The ***body*** of the function consists of a sequence of ***statements*** (two in this example) enclosed in braces (`{ ... }`). The last character of each statement is a semi-colon (`;`).

The return type of the function (`int`) is related to its last statement: `return 0`. The program behaves ***as if*** the function `main` is invoked by the operating system, executes its two statements, and returns the integer 0 as its result. The operating system interprets 0 as "normal termination". We can return other values if we want to tell the operating system that something went wrong.

The only thing left to discuss is the line beginning with `std::cout`. This illustrates an important general principle of C++. The text

```
std::cout << "Hello, world!" << std::endl
```

is an ***expression*** that yields a value. By putting a semicolon at the end of this expression, we discard the value and turn the expression into a ***statement***.

Expressions consist of ***operands*** and ***operators***. For example, $x + 5$ is an expression with two operands (x and 5) and one operator, +. We say that + is a ***binary operator*** because it takes two operands: x is its ***left operand*** and 5 is its ***right operand***. An expression can be ***evaluated*** to yield a ***result***. If x has the value 2, then evaluating $x + 5$ yields 7.

In C++, `<<` is a binary operator, usually called "insert". Its left operand is an output stream; its right operand may be a ***string***, a ***manipulator***, and various other things. Its result is a stream. When an expression contains more than one insertion operator, they are evaluated from left to right. The first step in evaluating

```
std::cout << "Hello, world!" << std::endl
```

is to evaluate

```
std::cout << "Hello, world!"
```

This has the effect of appending the string `"Hello, world!"` to the standard output stream and yields the updated output stream, say `uos`. The next step is to evaluate

```
uos << std::endl
```

which has the effect of appending a new line to the standard output stream (and a bit more, discussed below) and yields the updated output stream. In the program, this expression is followed by a semicolon, which throws away the final value. `cout` is a persistent object, however, and it still exists in its updated state.

There is a small, but significant, difference between the statements

```
        std::cout << "Hello, world!" << std::endl;
```

and

```
        std::cout << "Hello, world!" << "\n";
```

which could also be written as

```
        std::cout << "Hello, world!\n";
```

The difference is that `std::endl` writes the `\n` and also ***flushes the output buffer***. What does this mean?

It would be inefficient for a program to go through all of the operations of transferring data for every character in a stream. To save time, the program stores characters in a temporary area called a ***buffer*** and performs the actual transfers only when the buffer is full. If the program is writing data to a file, this behaviour is undetectable to the user. But, if the program is writing to a window, buffering makes a lot of difference. If we use `\n`, a significant amount of output may be generated by the program before we actually see it displayed. Using `std::endl` ensures that each line of output will be displayed as soon as it has been computed.

Here is yet another way of writing "Hello, world!" followed by end-of-line:

```
        std::cout << "Hello, world!"
                     "\n";
```

When C++ sees a quote (`"`) at the end of a string, followed by white space (blanks, tabs, and line breaks), followed by a quote at the beginning of a string, it erases quotes and the white space and treats the result as a single string. This is useful for writing long strings or strings that run over several lines.

1.4 Namespaces

As we noted above, we write `std::cout` to tell the compiler that the name `cout` comes from the namespace `std`. The name `cout` is a ***simple name*** (or just a ***name***), `std::cout` is a ***qualified name***, and `std::` is a ***qualifier***. "`::`" is called the ***scope operator***.

We can use this convention but, as programs get longer and more complex, the frequent appearance of "`std::`" becomes irritating, as well as being tedious to type. An alternative is to say to the compiler "When I write `cout`, I mean `std::cout`". This is the purpose of the `using` directive, as shown in Figure 3 on the next page. In general, we write one `using` directive at the beginning of the source file for each name that we intend to use. 18

We can go further, saying to the compiler "When I write any name that belongs to `std`, take it from there". The form of the `using` directive that does this is shown in Figure 4. Note that, in both Figure 3 on the following page and Figure 4 on the next page, we use `cout` and `endl` without the qualifier `std::`. 19

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    cout << "Hello, world!" << endl;
}
```

Figure 3: Hello.2

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello, world!" << endl;
}
```

Figure 4: Hello.3

Some people will tell you that the point of namespaces is to make the source of names explicit; they will tell you that version on page 9 is the best way to write programs, Figure 3 is acceptable, and Figure 4 is bad – the kind of code produced by lazy programmers who should be fired. This attitude is ***wrong***. The actual situation is more complex.

Namespaces were introduced into C++ to prevent name clashes in programs that use more than one library. Suppose that you are writing a program that uses a library `Cowboy` and another library `Artist`. Both libraries provide a function called `draw`. When you try to use `draw`, the compiler complains that it cannot tell which library to take it from. To avoid this problem, the libraries wrap their names in namespaces – perhaps `namespace cowboy` and `namespace artist`. The programmer can then write `cowboy::draw` or `artist::draw`, depending on which function is needed.

Most of the time, however, name clashes do ***not*** occur. In particular, it is very unlikely that a library writer would use a well-known name such as `cout`. Consequently, it is quite acceptable to use the form of Figure 4, while being aware that it may one day be necessary to use explicit qualification when a name clash actually occurs.

20

1.5 Strings

Figure 5 on the next page (Koenig and Moo 2000, page 9) shows a program that asks for the
21 user's name and then greets the user. Running it produces a dialogue like this:

```
What is your name? Nebuchadnezzar
Hi, Nebuchadnezzar!
```

Much of this program should already be familiar because it is similar to `Hello, world!`. The new features are the class `string` and the input stream `cin`.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    cout << "What is your name? ";
    string name;
    cin >> name;
    cout << "Hi, " << name << "!" << endl;
    return 0;
}
```

Figure 5: Greeting.1

The directive `#include <string>` informs the compiler that we will be using class `string`. Since the names provided by `string` are included in the namespace `std`, we can write simply "`string`" in the program rather than "`std::string`".

The second statement of the main program,

```
string name;
```

is both a ***declaration*** and a ***definition***. (We will discuss the distinction between declarations and definitions in detail later. For now, note that all definitions are also declarations, but a declaration is not necessarily a definition.) It introduces a new variable, `name`, into the program. The type of `name` is `string`. We could also say that `name` is a new ***object***, and an ***instance*** of class `string`.

The value of a `string` object is a string of characters, such as `"Hello, world!"`. After the declaration, the value of `name` is actually the ***empty string***, written `""`.

There are various ways of putting characters into a string. In this program, the statement

```
cin >> name;
```

reads data from the ***standard input stream***, `cin` (pronounced "see-in"). By default, `cin` gets data from the keyboard. In other words, whatever text you enter in response to the prompt `"What is your name?"` gets stored in the variable `name`.

Actually, this description is not quite accurate: `cin` reads only until it encounters white space (a blank, tab, or ENTER) and then stops. This explains the following dialogue:

```
What is your name? King Kong
Hi, King!
```

After the program has stored a value in `name`, it can use this value, as in the second `cout` statement.

There is a subtlety in Figure 5 on the preceding page that is worth noting. On the basis of the discussion at the end of Section 1.3 on page 9, you would be right to wonder why the statement

```
cout << "What is your name? ";
```

produces any output at all. Why is the data not left in the buffer until `endl` is output in the second `cout` statement? The answer is that the streams `cin` and `cout` are ***linked***. Any use of `cin` causes the buffer for `cout` to be flushed. This ensures that programs that implement a dialog in which the user must respond to displayed text will work as expected.

Memory management is an important aspect of C++ programming. In Figure 5 on the preceding page, memory for the string `name` is allocated on the run-time stack when the definition `string name` is evaluated. The class `string` manages memory when characters are put into the string (e.g., by `cin`) or the string is changed in other ways. At the final closing brace (“}”) of the program, any memory used by `name` is de-allocated.

A definition, of a variable or other named entity, occurs within a ***scope***. The scope consists either of the closest enclosing braces or, if there are no enclosing braces, the entire source file. In the latter case, we say that the definition is “at file scope”. Uses of the name must follow its definition. In other words, we cannot refer to a variable earlier in the program text than its definition: this is called the “definition before use rule”. The name ceases to be accessible at the end of its scope.

The definition of a name must textually precede its use.

In many cases, there are actions connected with the definition and the end of the scope. In Figure 5 on the previous page, which is typical, the object `name` is ***constructed*** at the point of its definition and ***destroyed*** at the end of the program. What actually happens is this: when an object definition is processed, a ***constructor*** for the object is called and, at the end of a scope, the ***destructors*** for all stack-allocated objects are called.

constructor (ctor)
destructor (dtor)

Some C++ programmers (and books) use the abbreviations “ctor” and “dtor” for constructor and destructor, respectively.

1.6 A Pretty Frame

The greeting produced by Figure 5 on the preceding page is dull and boring. The next version, in Figure 6 on the next page (Koenig and Moo 2000, page 12), produces dialogues like this: 22

```
Please enter your first name: Ferdinand

************************
*                      *
* Hi there, Ferdinand! *
*                      *
************************
```

23

The new feature in this program is `const string`. There are four definitions of the form 24

`const string` ⟨*name*⟩ `.... ;`

The keyword `const` tells the compiler that the values of the names are not going to change. We cannot declare `name` as `const` because its initial value (`""`) gets changed when we use `cin` to copy characters into it.

When we introduce a `const` name, we must provide a value for it in the same declaration. The value can be a simple value, as in this example:

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
cout << "Please enter your first name: ";
string name;
cin >> name;

const string greeting = "Hi there, " + name + "!";
const string spaces(greeting.size(), ’ ’);
const string second = "* " + spaces + " *";
const string first(second.size(), ’*’);

cout << endl;
cout << first << endl;
cout << second << endl;
cout << "* " << greeting << " *" << endl;
cout << second << endl;
cout << first << endl;

return 0;
}
```

Figure 6: Greeting.2

```
const string WELCOME = "Welcome to COMP 345!";
```

Figure 6 on the preceding page shows several other ways of providing an initial value using features of class `string`:

- `greeting = "Hi there, " + name + "!"`

  The operator `+` concatenates strings (i.e., joins them together). After `name` has received the value `"Ferdinand"`, the definition gives `greeting` the value `"Hi there, Ferdinand!"`.

  Although `greeting` is `const` and `name` is not `const`, we can use `name` as part of the value of `greeting`. The `const` qualifier says that `greeting` will not be changed later; it does not say that the value is known at compile-time.

- `spaces(greeting.size(), ' ')`

  The expression `greeting.size()` makes use of a ***member function*** of class `string`. Member functions are called with the "dot notation":

  ⟨*object name*⟩ `.` ⟨*function name*⟩ `(` ⟨*parameters*⟩ `)`

  In this case, the function name is `size` and it returns the size of (i.e., the number of characters in) the object `greeting`. Thus `greeting.size()` is a ***number*** (in this case, it is 20).

  Class `string` has a constructor which expects two arguments: a numeric value (type `int`) and a character value (type `char`). By writing `spaces(n, ' ')` we are saying: construct an instance of class `string` with name `spaces` that contains `n` characters where each character is `' '` (that is, a blank).

  Thus the effect of this particular definition is to define a constant `string` object `spaces` with value `"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"`. (␣ denotes a blank.)

- `second = "* " + spaces + " *"`

  The definition of `second` uses the definition of `spaces` and the concatenation operator, `+`.

- `first(second.size(), '*')`

  The definition of `first` is similar to that of `spaces`. It has the same number of characters as `second`, but each character is an asterisk, `'*'`. Note that C++ distinguishes between ***strings***, which have zero or more characters between double quotes (`"...."`) and ***characters*** (instances of type `char`), which have exactly one character between single quotes (`'.'`).

Having defined the strings `greeting`, `spaces`, `second`, and `first`, the program uses them to generate the desired output.

Although this program is trivial, it illustrates two important points about programming in general:

Avoid unnecessary assumptions.

The program is not written for people with five-letter names but for people with names of any (reasonable!) length.

Make the program do the work.

The strings needed for the display are ***computed*** (as much as possible). This makes it easy to generate a display whose size matches the name of the user.

It is not necessary to call cout once for each line of output. The six calls of cout in Figure 6 on page 15 could be replaced by a single call: 25

```
cout <<
   endl <<
   first << endl <<
   second << endl <<
   "* " << greeting << " *" << endl <<
   second << endl <<
   first << endl;
```

References

Abrahams, D. and A. Gurtovy (2005). *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley.

Kernighan, B. W. and D. M. Ritchie (1978). *The C Programming Language*. Prentice-Hall.

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.

2 Testing and Looping

2.1 Conditions 19
2.2 Conditional Expressions 22
2.3 Loops 23
2.4 Example: Computing the Frame 24
2.5 Counting 27
2.6 Loop Design 29
2.7 Assertions 34
2.8 Order Notation 37
2.9 Example: Maximum Subsequence 39

The control flow of a program is determined by tests and loops. In this section, we review the main C++ structures for branching and looping and discuss some aspects of loop design. In particular, we look at how to systematically write ***correct*** and ***efficient*** loops.

Engineering is the application of theory (mathematics, physics, etc.) to practice. Programming has not yet reached the maturity of engineering because we do not yet have adequate theories. We can prove that trivial programs satisfy a specification, but the techniques do not scale well.

Nevertheless, we should do what we can. It is feasible, for example, to use systematic techniques, rather than guesswork, to code loops, and these techniques are introduced in Section 2.3 on page 23.

2.1 Conditions

A ***condition*** is an expression whose value is either ***true*** or ***false***.[4]

C++ has a standard type, `bool`,[5] with exactly two values, `true` and `false`. Type `bool` also provides operators, as shown in Table 1. 27

Table 1: Boolean operators

Operation	Logic Symbol	C++ operator
conjunction	$\wedge$	`&&`
disjunction	$\vee$	`||`
negation	$\neg$	`!`

C++ also has operators that work on all of the individual bits of their operands in parallel, shown in Table 2. Do not confuse these with the boolean operators!

lazy evaluation

The Boolean operators `&&` and `||` are ***lazy***; this means that they do not evaluate their right

[4] This assumes a two-valued logic. There are other logics with more than two values. For example, the value of a formula in a three-valued logic might be ***true***, ***false***, or ***unknown*** (used, for example, in relational database systems).

[5] Named after George Boole, the English mathematician who wrote *The Laws of Thought*, where he describes what we now call *Boolean logic*, which has become a foundation of modern computer science.

Table 2: Bitwise operators

Operation	C++ operator
complement	~
shift left	`<<`
shift right	`>>`
and	`&`
exclusive or	ˆ
inclusive or	|

operands unless they need to. In detail, `p && q` is evaluated like this:

1. Evaluate `p`. If it is `false`, yield `false`.
2. Otherwise, evaluate `q` and yield the result.

Similarly, `p || q` is evaluated like this:

1. Evaluate `p`. If it is `true`, yield `true`.
2. Otherwise, evaluate `q` and yield the result.

Lazy evaluation has both good and bad consequences:

- It is more efficient, because the right operand is not evaluated unnecessarily.
- 28 It enables us to write tests such as

```
if (y != 0 && x / y <= MAX_RATIO)
    ....
```

- In C++, conjunction (and) and disjunction (or) are not commutative! That is, `p && q` is not always equivalent to `q && p`. However, they cannot yield different truth values: at worst, one would succeed and the other would fail.

C++ also provides ***comparison operators***, which compare two operands and yield a boolean value. They are `<` (less than), `<=` (less than or equal to), `==` (equal to), `!=` (not equal to), `>` (greater than), and `>=` (greater than or equal to). Do not confuse `==` with `=` (the assignment operator). These work with operands of most types but may not always make sense.

How to read C++:

`x == y` ≡ "`x` equals `y`"
`a = b` ≡ "`a` gets `b`" or "`a` becomes `b`" or "`a` is assigned `b`"

In addition to the type `bool`, C++ inherits some baggage from C. In C, and therefore in C++:

- 0 is considered `false`
- Any non-zero value is considered `true`

This convention has several consequences:

- A condition such as `counter != 0` , in which `counter` is an integer, has the same truth-value as `counter`. Many C++ programmers therefore use the simpler form, `counter`, in a context where a truth value is expected.

- Expressions that are `true`, when considered as truth values, are not necessarily equal. For example, suppose we want to express the fact that two counters are either both zero or both non-zero. We could express this condition as

  ```
  (counter1 == 0) == (counter2 == 0)
  ```

  or, equivalently, as

  ```
  (counter1 != 0) == (counter2 != 0)
  ```

  Using the abbreviation above, we might be tempted to simplify this to

  ```
  counter1 == counter2
  ```

  But this expression has a different meaning!

> ***Write conditions carefully. Prefer complete expressions to abbreviations.***

The convention also suggests the question: what ***is*** zero in C++? The answer is that there are many things that are considered to be zero — and are therefore also considered to be `false` in a condition:

- Integers of type `short`, `int`, and `long`, with value 0
- Floating point numbers of type `float`, `double`, and `long double`, with value 0.0
- The character `'\0'`
- Any null pointer (we will discuss pointers later)
- The first element of an enumeration
- The `bool` value `false`
- And perhaps others . . . .

The empty string is ***not*** equivalent to `false`:

```
const string emptyString = "";
```

2.2 Conditional Expressions

A ***conditional expression*** is an expression with a boolean value. Conditional expressions are often called ***tests***, for short.

29 Programs typically evaluate conditions using `if` statements, which have one of two forms:

```
if ( ⟨condition⟩ )
   ⟨statement⟩
```

or

```
if ( ⟨condition⟩ )
   ⟨statement⟩
else
   ⟨statement⟩
```

and work as you would expect them to.

The ⟨*statement*⟩s in an `if` statement can be simple or compound. This example illustrates both
30 possibilities:

```
if (angle < PI)
{
   cout << "Still going round ...";
   angle += 0.01;
}
else
{
   angle = 0;
}
```

The braces around the final assignment, `angle = 0`, are not essential. We could alternatively have written

```
....
else
   angle = 0;
```

Whether you include the braces or not is a matter of taste and preference. Including them adds two lines to the code (or one line if you put the left brace on the same line as `else`). But including them makes it easy to add another line later – which is often necessary – and avoids the error of adding a line but forgetting to add the braces.

The ?: Operator. C++ also knows the ternary ***conditional operator*** `?:` (in fact, it is the only operator that requires three operands): ?: operator 31

```
⟨condition⟩ ? ⟨expression1⟩ : ⟨expression2⟩
```

It works like this: if the ⟨*condition*⟩ evaluates to `true`, the result is ⟨*expression1*⟩, otherwise ⟨*expression2*⟩. For example, to assign the bigger of two variables `a` and `b` to a variable `x`, you could write

```
x = a > b ? a : b;
```

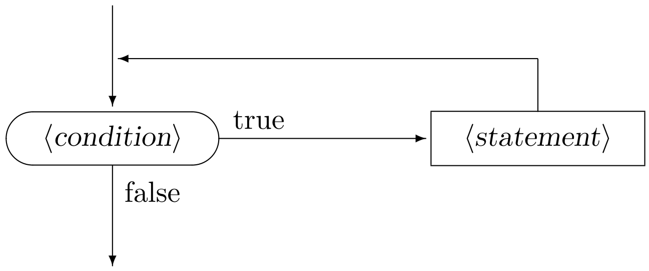
2.3 Loops

C++ provides three looping constructs. In the order in which you should consider using them, they are: `for`; `while`; and `do`/`while`.

The `while` loop has this structure 32

```
while ( ⟨condition⟩ )
    ⟨statement⟩
```

and it works like this:



An important feature of the `while` loop is that the loop body may not be executed at all. This is a very useful feature and, when writing a `while` loop, you should always check that its behaviour when the condition is initially `false` is correct.

A loop of this form 33

```
⟨initialize⟩
while ( ⟨condition⟩ )
{
    ⟨action⟩
    ⟨step⟩
}
```

should usually be written more concisely in this (almost) equivalent form:

```
for ( ⟨initialize⟩ ;  ⟨condition⟩ ; ⟨step⟩ )
{
   ⟨action⟩
}
```

34 For example, instead of writing

```
int i = 0;
while ( i < MAX )
{
   doSomething(i);
   ++i;
}
```

write this:

```
for (int i = 0; i < MAX; ++i)
{
   doSomething(i);
}
```

in C++, prefer '++i' to 'i++'

It is a good idea to get into the habit of using pre-increment (`++i`) rather than post-increment (`i++`). For integers, it doesn't make much difference, but `++` and `--` are overloaded for other types for which the difference is more significant. The reason for preferring pre-increment is that `i++` may force the compiler to generate a temporary variable, whereas `++i` does not.

35 The `do/while` loop is used only when you want to evaluate the condition ***after*** performing the loop body:

```
do
   ⟨statement⟩
while ( ⟨condition⟩ )
```

2.4 Example: Computing the Frame

36 37 The program in Figure 7 on the next page uses `if`, `while`, and `for` statements. Figure 8 on page 26 shows an example of its use. The outer loop, formatting the rows of the display, is a `for` loop, because exactly one row is processed during each iteration. This pattern does not work for the inner loop, because progress is not always one column at a time. The `if` statements make decisions about what text to output, and they all have complex conditions with `&&` and `||` operators.

This program is easier to modify than Figure 6 on page 15: to change the size of the frame, all we have to do is change numbers in the `const` declarations. In fact, just changing the value of `pad` will change the spacing all around the greeting.

Design programs so that a few easily changed parameters change the behaviour of the program in a consistent way.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
   cout << "Please enter your first name: ";
   string name;
   cin >> name;
   const string greeting = "Hello, " + name + "!";
   const int pad = 1;
   const int rows = pad * 2 + 3;
   const string::size_type cols = greeting.size() + pad * 2 + 2;
   cout << endl;
   for (int r = 0; r != rows; ++r)
   {
      string::size_type c = 0;
      while (c != cols)
      {
         if (r == pad + 1 && c == pad + 1)
         {
            // We are positioned for the greeting.
            cout << greeting;
            c += greeting.size();
         }
         else
         {
            if (r == 0 || r == rows - 1 ||
                c == 0 || c == cols - 1)
               // We are on a border.
               cout << "*";
            else
               cout << " ";
         ++c;
         }
      }
      cout << endl;
   }
   return 0;
}
```

Figure 7: Greeting.3

```
Please enter your first name: Wilberforce

***********************
*                     *
* Hello, Wilberforce! *
*                     *
***********************
```

Figure 8: A dialogue with Greeting.3

Source Code Comments. The original program (Koenig and Moo 2000, page 29) contains a
38 number of comments:

```
// say what standard-library names we use
using std::cin;         using std::endl;
....
// ask for the person's name
cout << "Please enter your first name: ";
....
// read the name
string name;
cin >> name;
```

These comments are acceptable, but only because this program appears in an introductory text book. In general, comments like this should not be written unless:

- They provide information that is not obvious from the code
- They make the code more readable without adding noise to it

There is one comment in this program which might serve a purpose:

```
// the number of blanks surrounding the greeting
const int pad = 1;
```

Without the comment, the meaning of `pad` would not be obvious, although it is not hard to guess its meaning by reading the next few lines of code. But any comment that is provided to explain the role of a variable raises an immediate question: could we eliminate the need for the comment by choosing a better name?

In this case, we could replace `pad` by `spaceAroundGreeting`, or some such name. Then the comment would be unnecessary.

Of course, it takes longer to type `spaceAroundGreeting` than `pad`. But the time programmers take to type a name a few times (five times for this program) is negligible compared to the time maintainers take to figure out what they meant.

39 Another problem with Figure 7 on the preceding page is the mysterious numbers 2 and 3:

```
    const int rows = pad * 2 + 3;
    const string::size_type cols = greeting.size() + pad * 2 + 2;
```

Although there is a comment explaining the meaning of pad, there is no comment explaining the formulas pad * 2 + 3 and pad * 2 + 2. Again, we can excuse the authors in this case, because the explanation appears in the book (Koenig and Moo 2000, page 18–22). In production code, however, these numbers should be accompanied by explanatory comments or even defined as constants.

We could write the definitions in a way that shows how the values are obtained. This requires more typing but should not make any difference to the compiled code:

```
    const int rows = 1 + pad + 1 + pad + 1;
    const string::size_type cols = 1 + pad + greeting.size() + pad + 1;
```

string::size_type

Compiler Warnings. Why does the program above use int as the type of rows – which seems quite natural – and the curious expression string::size_type as the type of cols? This type is used because it is the type returned by the function string::size(). It is an integer type, but we don't know which one (probably unsigned long but possibly something else). If we declare "const int cols", the compiler issues a warning: 40

```
        frame.cpp(15) : warning C4267: 'initializing' :
           conversion from 'size_t' to 'const int', possible loss of data
```

We don't want warning messages when we compile, and the best way to get rid of this message is to use the correct type.

> ***If the compiler issues warning messages, revise your code until they disappear.***

It is not always easy to eliminate warnings, but the effort is worthwhile: Even if it doesn't save your time now, it may save a maintainer's time later.

2.5 Counting

In everyday life, if we have N objects and want to identify them by numbers, we assign the numbers $1, 2, 3, \ldots, N$ to them. Another way to look at this is to say that the set of numbers forms a ***closed interval*** which we can write as either $1 \le i \le N$ or $[1, N]$.

Programmers are different. Given N objects, they number them $0, 1, 2, \ldots, N - 1$. This set of numbers forms a ***semi-closed*** (or ***semi-open***) interval which we can write as either $0 \le i < N$ or $[0, N)$. The advantages of semi-closed intervals include:

fencepost error

- They reduce the risk of "off-by-one" or "fencepost" errors.

  To see why off-by-one errors are called "fencepost errors", consider the length of a fence with N posts spaced 10 feet apart. The length of the fence is $10(N - 1)$ feet: see Figure 9. 41

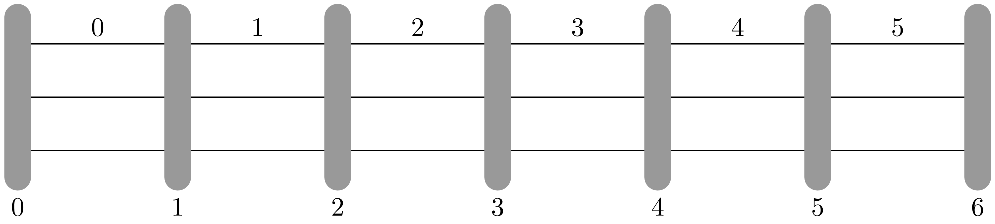


Figure 9: Fencepost numbering

- The closed interval $[M, N]$ has $N - M + 1$ elements; the semi-closed interval $[M, N)$ has $N - M$ elements, which is easier to remember and calculate.
- In particular, the closed interval $[M, N]$ is empty if $M > N$, which many people find counter-intuitive. The semi-closed interval $[M, N)$ is empty if $M = N$, which is easier to remember and check.
- Semi-closed intervals are easier to join together. Compare

  $$[A, B),\ [B, C),\ [C, D), \ldots$$

  to

  $$[A, B - 1],\ [B, C - 1],\ [C, D - 1], \ldots$$

- The index of the first element of an array A in C++ is 0. The address of the I'th element of the array is $\&A + sI$ where $\&A$ is the address of the array and s is the size of one element (we will discuss array addressing in more detail later).

Typical C++ loops do ***not*** have the form

```
for (int i = M; i <= N; ++i) ....
```

in which `M` is often 1. Instead, they have the form

```
for (int i = M; i < N; ++i) ....
```

in which `M` is often zero. Note that, in the first case, `N` is the last element processed but, in the second case, `N` is the first element ***not*** processed. In fact, we will see later that there are good reasons for writing the termination condition as `!=` rather than `<`:

```
for (int i = M; i != N; ++i) ....
```

Start a range with the index of the first item to be processed; end the range with the index of the first item <u>not</u> processed. The first index of a range is often 0.

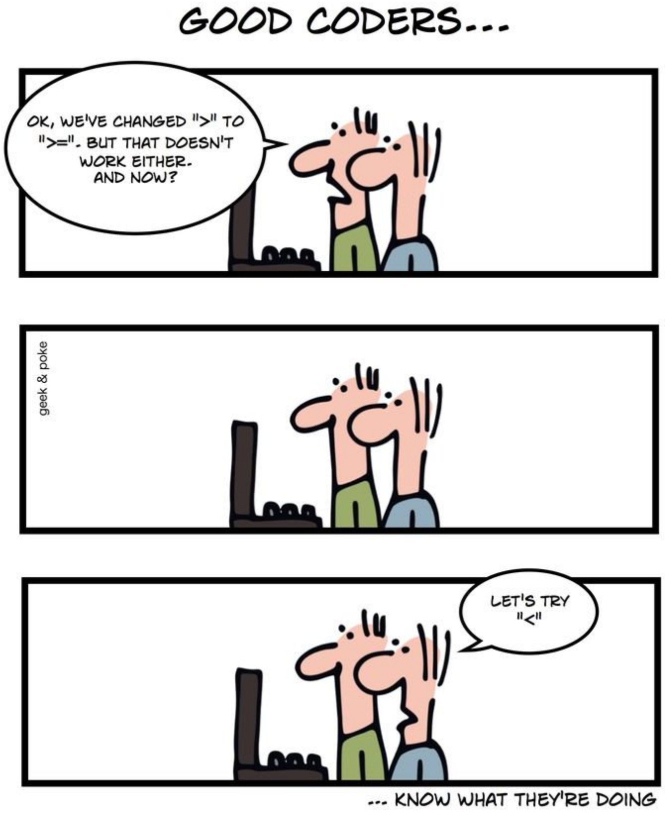

GOOD CODERS...
OK, WE'VE CHANGED ">" TO ">=". BUT THAT DOESN'T WORK EITHER. AND NOW?
geek & poke
LET'S TRY "<"
... KNOW WHAT THEY'RE DOING

Figure 10: There should be a better way...

2.6 Loop Design

Figure 10[6] shows a popular, but unreliable approach to loop design. Let's try to do better.

We discuss the design of loops with the assumption that they are going to be `while` loops. If they later turn out to have the appropriate pattern, we can convert them to `for` loops. With experience, we learn to recognize loops that have the `for`-loop pattern in advance and avoid the conversion step.

We design `while` loops using the following schema (the numbers are for reference, not part of the code): 42

```
1    ⟨initialize⟩
2    //  $I$
3    while (C)
4    {
5        //  $I \wedge C$
6        ⟨body⟩
7        //  $I$
8    }
9    //  $I \wedge \neg C$
```

[6] Copyright Geek&Poke (http://geek-and-poke.com/), licensed under Creative Commons Attribution 3.0 Unported (CC BY 3.0) https://creativecommons.org/licenses/by/3.0/deed.en_US

loop invariant

- The comment on line 2 says that, after initialization, the condition I is true. I is the ***invariant*** of the loop.
- The comment on line 5 says that, at the start of the body of the loop, the loop invariant I and the loop condition C are both true. This provides an ***assumption*** that we can use in coding the loop.
- The comment on line 7 says that, after the body of the loop has been executed, the invariant I is still true (this is what being an ***invariant*** means).
- The comment on line 9 says that, when the loop exits, the invariant I is true but the loop condition C is false.

> ***"Good programmers instinctively know what the invariant should be in a*** `while` ***loop."*** — ***Joel Spolsky,* Joel on Software*, page 164.***

2.6.1 Counting Lines

As a simple example of loop design, we consider the problem of writing `rows` lines of output, as in Figure 7 on page 25. We use a counter `r` to count the number of lines written and, following the convention for initializing counters, we will initialize it to zero. We will make the minor change of writing `ROWS` rather than `rows`, to indicate that `ROWS` is a constant and to improve readability.

Here is a suitable invariant: r *lines have been written*. Note that initializing r to zero makes the invariant true, because we haven't written any lines yet.

43 If we have printed $ROWS$ lines, there is no more to do. Consequently, the condition for the `while` loop is `r != ROWS` and the code begins:

```
1       int r = 0;
2       // r lines have been written
3       while (r != ROWS)
4       {
5           // r lines have been written and r != ROWS
```

The body of the loop must generate one line of output. We don't care (for this exercise) what that output will be, so we will just write a `cout` statement. The body must also count the lines produced. Thus the code continues:

```
6           cout << .... << endl;
6.1         // r+1 lines have been written
6.2         ++r;
7           // r lines have been written
8       }
9       // r lines have been written and not (r != ROWS)
```

Line 6 generates one line of output. This ***invalidates*** the invariant, as the comment on line 6.1 shows. Incrementing `r` makes the invariant valid again. We note that an invariant is not ***always*** true, but is true at certain well-defined points in the program. Also, whenever we perform an action that invalidates the invariant, we must perform another action (`++r` in this case) that makes it valid again.

Line 9 can be simplified as follows: 44

```
   r lines have been written and not (r != ROWS)
⇒  r lines have been written and r == ROWS
⇒  ROWS lines have been written
```

which is exactly what we needed.

Additional points:

- If we had written the `while` condition as `r < ROWS`, this reasoning would lead to a different conclusion:

  ```
     r lines have been written and not (r < ROWS)
  ⇒  r lines have been written and r >= ROWS
  ```

  That is, we could claim only that the code generates ***at least*** `ROWS` lines of output. This is correct, but it is less precise than the original conclusion, which is that the code generates ***exactly*** `ROWS` lines of output. One advantage of `!=` over `<` as a `while` condition is that it gives us a more precise conclusion. (This advantage was first pointed out by (Dijkstra 1976, page 56n). We will discuss other advantages later, in connection with the STL.)[7]

- This is an example of the situation mentioned above: the final code matches the pattern of the `for` statement and we can write the solution with a `for` loop. The invariant still applies: 45

  ```
  for (int r = 0; r != ROWS; ++r)
     // r lines have been written
     cout << .... endl;
  // ROWS lines have been written
  ```

- In addition to the invariant, which expresses something that does not change, we need something in the loop body that ***does*** change. Otherwise, the loop condition would never be satisfied and the loop would never terminate. In this example, the thing that changes is obviously the row counter; in other cases, it might not be so obvious.

- Suppose that we start counting from 1. A plausible invariant is: "`r` is the next line to be written", but this turns out not to be an invariant, because it is not true after we have written the last line. An invariant that works is "`r-1` lines have been written", which yields the following code in Figure 11 on the following page. 46 The code is correct, but it is more complicated and error-prone than the solution that counts from zero. As previously mentioned, the last line implies only that ***at least*** `ROWS` lines have been written rather than ***exactly*** `ROWS` lines have been written.

[7] However, this only applies to integer data types – never use equality/inequality comparisons with floating point numbers!

```
1      int r = 1;
2      // r-1 lines have been written
3      while (r <= ROWS)
4      {
5          // r-1 lines have been written and r <= ROWS
6          cout << .... << endl;
6.1        // r lines have been written
6.2        ++r;
7          // r-1 lines have been written
8      }
9      // r-1 lines have been written and not (r <= ROWS)
```

Figure 11: Starting from 1

Loop Coding Errors: The Zune Bug. Despite the fact (or, perhaps, especially because) loop conditions often seem deceptively simple, they nevertheless are one of the most common causes of bugs in software products. As an example, consider the code in Figure 12 that caused a crash[8] of the Zune media player on 31.12.2008.[9] It is part of a function that computes the current year from the number of days that have passed since a certain starting date (in this case, 01.01.1980). As you can see, the code has to take leap years into account. Why did it crash on 31.12.2008
47 (hint: 2008 was a leap year)?

```
while (days > 365)
{
    if (IsLeapYear(year))
    {
        if (days > 366)
        {
            days -= 366;
            year += 1;
        }
    }
    else
    {
        days -= 365;
        year += 1;
    }
}
```

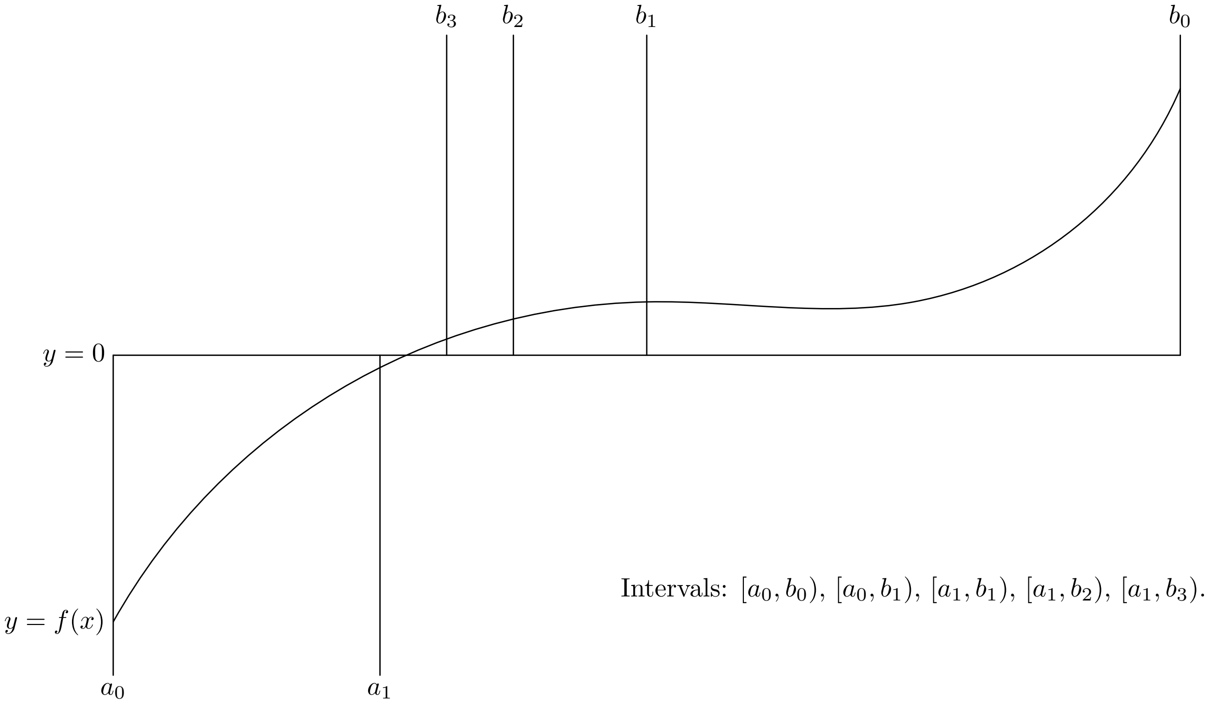
Figure 12: The cause of the Zune crash 31.12.2008

[8] See http://www.engadget.com/2008/12/31/30gb-zunes-mysteriously-begin-to-fail-at-12am-december-31st/

[9] The Zune software is actually written in C#, not C++, but the lesson here is really about loop design.

2.6.2 Finding Roots by Bisection

Suppose that f is a continuous real-valued function, $A < B$, and $f(A) \leq 0$ and $f(B) > 0$. Then a fundamental theorem of real analysis says that there must be a value of x such that $f(x) = 0$ and $A \leq x < B$; that is, a ***root*** (or ***zero***) of f. Using the bisection method, we find a sequence of approximations to x by halving the interval $[A, B)$ yielding smaller intervals $[a, b)$. Figure 13 48
illustrates the process.



Intervals: $[a_0, b_0)$, $[a_0, b_1)$, $[a_1, b_1)$, $[a_1, b_2)$, $[a_1, b_3)$.

Figure 13: Finding zeroes by bisection

Part of the invariant is $f(a) \leq 0 \wedge f(b) > 0$. This ensures that there is a root of f in $[a, b)$. To be complete, we will also require $a < b$. We cannot expect to find the root exactly, so we will stop when the interval is sufficiently small; specifically, when $b - a < \varepsilon$. This suggests that the loop condition should be $\neg(b - a < \varepsilon)$, which is equivalent to $b - a \geq \varepsilon$. Thus we have: 49

```
double a = A;
double b = B;
//  I ≡ f(a) ≤ 0 ∧ f(b) > 0 ∧ a < b
while (b - a >= eps)

   ....

//  I ∧ b − a < ε
```

Note that the final comment, obtained by and'ing the invariant and the negation of the `while` condition, is what we need: there is a root within a small interval.

To complete the loop body, we find the midpoint of the interval $[a, b)$, which is at $m = (b - a)/2$. If $f(m) > 0$, there must be a root between a and m. If $f(m) \leq 0$, there must be a root between m and b. We can write the code below. Note carefully how the `if` statement maintains the invariant. 50

```
double a = A;
double b = B;
//  I ≡ f(a) ≤ 0 ∧ f(b) > 0 ∧ a < b
while (b - a >= eps)
{
   double m = 0.5 * (a + b);
   if (f(m) > 0)
      b = m;
   else
      a = m;
   //  I
}
//  I ∧ b − a < ε
```

51

52

Figure 14 on the next page expands this idea into a complete function and a test program. The
53 `assert` statement is discussed in Section 2.7. When this program is run, it displays:

```
pi = 3.14159
e  = 2.71828
Assertion failed: a < b && f(a) <= 0 && f(b) > 0 &&
                  "bisect: precondition violation",
  file f:\courses\comp345\src\bisect\bisect.cpp, line 13
```

2.7 Assertions

54 The program in Figure 14 on the next page contains an ***assertion***:

```
assert(a < b && f(a) <= 0 && f(b) > 0 &&
      "bisect: precondition violation");
```

As with most programming constructs, there are three things that are useful to know about assertions: syntax *(what do we write?)*; semantics *(what happens?)*; and pragmatics *(when and why do we use assertions?)*.

```
#include <iostream>
#include <cassert>
#include <cmath>

using namespace std;

double bisect(
    double f(double),
    double a,
    double b,
    double eps = 1e-6 )
{
    if (a > b)
        return bisect(f, b, a, eps);

    assert(a < b && f(a) <= 0 && f(b) > 0 &&
          "bisect: precondition violation");

    while (b - a > eps)
    {
        // a < b && f(a) <= 0 && f(b) > 0
        double m = 0.5 * (a + b);
        if (f(m) > 0)
            b = m; // f(b) > 0
        else
            a = m; // f(a) <= 0
    }
    return 0.5 * (a + b);
}

double logm(double x)
{
    return log(x) - 1;
}

int main()
{
    cout << "pi = " << 0.5 * bisect(sin, 4, 8) << endl;
    cout << "e  = " << bisect(logm, 0.5, 3) << endl;
    cout << "e  = " << bisect(logm, 3, 3) << endl;
}
```

Figure 14: Finding zeroes by bisection

Syntax. Any code unit that uses assertions must contain either the (preferred) new-style directive

```
#include <cassert>
```

or the old-style directive

```
#include <assert.h>
```

assert

The assert statement itself has the form

```
assert( ⟨condition⟩ );
```

Semantics.

- When the program executes, the ⟨*condition*⟩ is evaluated.
- If the ⟨*condition*⟩ yields `true`, or anything equivalent to `true`, the `assert` statement has ***no effect***.
- If the ⟨*condition*⟩ yields `false`, or anything equivalent to `false` (i.e., any kind of zero), the program is terminated with an error message.

The precise form of the error message depends on the compiler. In general, it will contain the text of the ⟨*condition*⟩, the name of the file in which the `assert` statement occurs, and the line number of the statement within the file.

Pragmatics. Here is a reliable guide to the use of assertions:

If an assertion fails, there is a logical error in the program.

Think of `assert(`C`)` as saying "I, the programmer, believe that C should always be true at this point in the program". Then the failure of an assertion implies that the programmer's belief was mistaken, which further implies that there was something wrong with the reasoning and therefore something wrong with the program.

55 Assertions are useful for expressing preconditions, postconditions, and invariants.

- A ***precondition*** is a condition that should be true on entry to a function. It imposes an obligation on the caller to ensure that the arguments passed to the function are appropriate. The assertion in Figure 14 on the preceding page is of this form.
- A ***postcondition*** is a condition that should be true when a function returns. It is a promise by the function to the caller that the function has done its job correctly.
- An ***invariant*** is a condition that should be true at certain, well-defined points in the program. For example, a loop invariant.

A useful trick for assertions is to append "`&&` ⟨*string*⟩" to the condition, in which ⟨*string*⟩ describes what has gone wrong. This does not change the value of the condition, because any string is considered to be non-zero and therefore `true`, but it may improve the diagnostic issued by the compiler.

For example, when the assertion 56

```
assert( a < b && f(a) <= 0 && f(b) > 0 &&
        "bisect: precondition violation");
```

in the program of Figure 14 on page 35, the run-time system displays

```
Assertion failed: a < b && f(a) <= 0 && f(b) > 0 &&
                "bisect: precondition violation",
file f:\courses\comp345\src\bisect\bisect.cpp, line 13
```

`#define NDEBUG`

Assertions can be disabled by writing "`#define NDEBUG`" ***before*** "`#include <cassert>`". If the assertions were not failing, the only effect of this will be to save a few nanoseconds of execution time. Since conditions that were evaluated with `NDEBUG` undefined are no longer evaluated with `NDEBUG` defined, it is important that:

Asserted conditions must not have side-effects.

Exceptions provide another way of recovering from errors; we will discuss them later in the course (Section 10.2 on page 204).

2.8 Order Notation

$\mathcal{O}()$ notation

It is useful to have a concise way of describing how long a program or algorithm takes to run. The conventional way of doing this is to use "Big O" notation.

> This section presents a simplified view of Big O notation. The Bachmann-Landau family of notations defines $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$, and $\omega(\cdot)$ as well as $\mathcal{O}(\cdot)$.

The time taken to compute something usually depends on the size of the input. We will use n to stand, in a general way, for this size. For example, n might be the number of characters to be read, or the number of nodes of a graph to be processed. If the time required is ***independent*** of n, we say that the ***time complexity*** is $\mathcal{O}(1)$. If the time required increases linearly with n, we say that the complexity is $\mathcal{O}(n)$.

Formally, $\mathcal{O}(\cdot)$ defines a ***set*** of functions: 57

$$g(n) \quad \in \quad \mathcal{O}(f(n))$$

if and only if there are constants A and M such that, for all $N > M$, $g(N) < A \cdot f(N)$.

Big-oh notation does two things: it singles out the dominant term of a complicated expression, and it ignores constant factors. For example,

$$\begin{aligned} n^2 &\in \mathcal{O}(n^2) \\ \text{and} \qquad 1000000n^2 &\in \mathcal{O}(n^2) \end{aligned}$$

because we can chose $A = 1000001$ in the definition above. Also

$$n^3 + 100000n^2 + 100000n + 100000 \quad \in \quad \mathcal{O}(n^3)$$

because, for large enough n, the first term dominates the others. By similar reasoning

$$n + 10^{-10}n^2 \quad \in \quad \mathcal{O}(n^2)$$

even though the second term looks very small.

Informally, we don't say "is a member of $\mathcal{O}(n^2)$" but, less precisely, "the complexity is $\mathcal{O}(n^2)$" (or whatever the complexity actually is). 58

Figure 15 on the next page shows a small part of the ***complexity hierarchy***. Each line defines a set of functions that is a proper subset of the set defined on the next line. So, for example, $\mathcal{O}(n) \subset \mathcal{O}(n \log n)$ (all linear functions are log-linear), and so on.

Very few algorithms are $\mathcal{O}(1)$. We use this set to describe operations that have a constant time bound. For example, "reading a character" is (or at least should be) $\mathcal{O}(1)$. Logarithmic and linear algorithms are good. Log-linear algorithms are acceptable: sorting, for example, is log-linear.[10] Polynomial algorithms, $\mathcal{O}(n^k)$ with $k > 2$, tend to be useful only for small problem sizes.

Exponential and factorial algorithms are useless except for very small problems. A problem whose best known solution has exponential or factorial complexity is called ***intractable***. Such problems must be solved by looking for approximations rather than exact results. One of the best-known intractable problems is TSP: the "travelling salesperson problem". A salesperson must make a certain number of visits and the problem is to find an ordering of the visits that minimizes some quantity, such as cost or distance. The only known way to find an exact solution is to try all possible routes and note the minimal route. If n visits are required, the number of possible paths is $\mathcal{O}(n!)$, making the exact solution infeasible if n is in the hundreds or even thousands.

Note that TSP is typical of problem descriptions. We are not really interested at all in travelling salespersons and, in any case, their actual problems (which might involve 20 visits at most) are easily solved. But TSP represents the generic problem of finding a minimal path in a weighted graph, and many practical problems can be put into this abstract form. One example is: find the quickest path for a drilling machine that has to drill several thousand holes in a printed-circuit board. 59

Figure 16 on the facing page shows the progress that has been made in solving three-dimensional elliptic partial differential equations. Equations of this kind must be solved for VLSI simulation, oil prediction, reactor simulation, airfoil simulation, and other significant problems. The difference between $\mathcal{O}(N^7)$ and $\mathcal{O}(N^3)$ corresponds to a factor of N^4, or a million times for a problem for which $N = 100$.

[10] Provided that you don't use bubblesort.

Function	Condition	Description
$\mathcal{O}(1)$		constant
$\mathcal{O}(\log n)$		logarithmic
$\mathcal{O}(\sqrt{n})$		square-root
$\mathcal{O}(n)$		linear
$\mathcal{O}(n \log n)$		log-linear
$\mathcal{O}(n^2)$		quadratic
$\mathcal{O}(n^3)$		cubic
$\mathcal{O}(n^k)$	$k > 1$	polynomial
$\mathcal{O}(a^n)$	$a > 1$	exponential
$\mathcal{O}(n!)$		factorial

Figure 15: Part of the complexity hierarchy

It is often claimed that hardware has improved more rapidly than software. For some problems, however, the improvements in software have been just as dramatic as those for hardware. Putting the two together, a modern supercomputer can solve differential equations more than a trillion ($10^6 \times 10^6 = 10^{12}$) times faster than was possible in 1945.

Method	Year	Complexity
Gaussian elimination	1945	$\mathcal{O}(N^7)$
SOR iteration (suboptimal parameters)	1954	$\mathcal{O}(N^5)$
SOR iteration (optimal parameters)	1960	$\mathcal{O}(N^4 \log N)$
Cyclic reduction	1970	$\mathcal{O}(N^3 \log N)$
Multigrid	1978	$\mathcal{O}(N^3)$

Figure 16: Solving three-dimensional elliptic partial differential equations (adapted from ***Numerical Methods, Software, and Analysis***, by John Rice (McGraw-Hill, 1983))

2.9 Example: Maximum Subsequence

The following problem arises in pattern recognition: find the maximum contiguous subvector of a one-dimensional vector (see (Bentley 1986, pp. 69–80)). The problem is trivial if all of the values in the vector are positive, because the maximum subvector is just the whole vector. If some of the values are negative, the problem is interesting. For example, given the vector 60

$$31 \quad -41 \quad 59 \quad 26 \quad -53 \quad 58 \quad 97 \quad -93 \quad -23 \quad 84$$

our algorithm should find the subvector

$$59 \quad 26 \quad -53 \quad 58 \quad 97$$

We assume that an empty subvector has sum 0. This implies that the maximum subvector can never be negative because, if we had a negative subvector, we could always replace it with the (larger) adjacent empty subvector.

There is an obvious solution: we can simply sum ***all possible*** subvectors and note which one has the largest sum. We assume that the given vector has N elements. We need three nested loops: one to choose the first element of the subvector, one to choose the last element, and one 61 to sum the elements in between. Figure 17 shows a solution based on this idea. It uses `max`, a standard C++ library function that returns the greater of its two arguments.

```
int maxSoFar = 0;
for (int i = 0; i != N; ++i)
{
    for (int k = i; k != N; ++k)
    {
        int sum = 0;
        for (int j = i; j <= k; ++j)
            sum += v[j];
        maxSoFar = max(maxSoFar, sum);
    }
}
```

Figure 17: Maximum subvector: first attempt

The algorithm of Figure 17 has complexity $\mathcal{O}(N^3)$. It is not efficient and, by inspecting it carefully, we can see how to do better. During each cycle of the outer loop, we can use a single loop to sum all subvectors starting at that point. This gives the second version of the algorithm, 62 with two nested loops and complexity $\mathcal{O}(N^2)$, shown in Figure 18.

```
int maxSoFar = 0;
for (int i = 0; i != N; ++i)
{
    int sum = 0;
    for (int j = i; j != N; ++j)
    {
        sum += v[j];
        maxSoFar = max(maxSoFar, sum);
    }
}
```

Figure 18: Maximum subvector: second attempt

Many, perhaps most, programmers would give up at this point and simply assume that quadratic complexity is the best that can be achieved. But, being more persistent, we will seek a better solution using invariants.

Here is a useful, general technique for solving problems with one-dimensional vectors: process the vector one element at a time, maintaining and updating as much information as is needed to proceed to the next step. Specifically, suppose we are about to process element `i`. What useful information can we obtain?

The first point to notice is that if we have a subvector "ending here", we can update it simply by adding `v[i]` to it. This will give us a new subvector "ending here" that we can keep if it is bigger than anything we have already seen.

The second point to notice is that we can remember the value of the largest subvector seen "so far" (this corresponds to what "we have already seen" in the previous sentence). The invariant that we need is:

`maxEndingHere` $\equiv$ the largest subvector that ends here
`maxSoFar` $\equiv$ the largest subvector we have seen so far

To make the invariant true initially, we set both variables to zero. When we examine `v[i]`, we
note that `maxEndingHere` will not get smaller if `v[i]` > 0. Figure 19 shows the final version of 63
the algorithm. This version requires time proportional to the length of the sequence. This is
much better than the first version, which required time proportional to the ***cube*** of the length
of the sequence.

```
int maxSoFar = 0;
int maxEndingHere = 0;
for (int i = 0; i != N; ++i)
{
    maxEndingHere = max(maxEndingHere + v[i], 0);
    maxSoFar = max(maxSoFar, maxEndingHere);
}
```

Figure 19: Maximum subvector: an efficient solution

References

Bentley, J. (1986). *Programming Pearls.* Addison-Wesley.

Dijkstra, E. W. (1976). *A Discipline of Programming.* Prentice-Hall.

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example.* Addison-Wesley.

3 Batches of Data and the STL

3.1 Lvalues and Rvalues 43
3.2 Functions 44
3.3 Working with files 47
3.4 End-of-file Handling 52
3.5 Storing Data: STL Containers 56
3.6 Summary of the Standard Template Library (STL) 59

For additional information on the topics covered in this lecture you can consult (Weiss 2004, Section 2.3) (parameter passing), (Prata 2012, Chapter 6) (simple file I/O), (Prata 2012, Chapter 17) (details on streams, buffers, and I/O), as well as (Prata 2012, Chapter 16) (STL, `vector` class).

3.1 Lvalues and Rvalues

The names “Lvalue” and “Rvalue” are derived from the assignment statement 65

```
v = e;
```

Although the assignment looks symmetrical, it is not. When it is executed, the right side, `e`, must yield a ***value*** and the left side, `v`, must yield a ***memory address*** in which the value of `e` can be stored.

Anything which can appear on the left of `=` (that is, anything that can represent an address) is called an Lvalue.

Anything which can appear on the right of `=` (that is, anything that can yield a value) is called an Rvalue.

“Lvalue” and “Rvalue” are often written without capitals, as “lvalue” and “rvalue”. We use initial capitals in these notes for clarity and emphasis.

All Lvalues are Rvalues because, having obtained an address, we can find the Rvalue stored at that address. The operation of obtaining an Rvalue from an Lvalue is called ***dereferencing***. But there are Rvalues that are not Lvalues. Lvalues include: simple variables (`x`); array components (`a[i]`); fields of objects (`o.f`); and a few other more exotic things. Rvalues that are not Lvalues include literals (for example, `67`, `"this is a string"`, `true`) and expressions.

dereferencing

This is (roughly) the original definition for Lvalues and Rvalues, but the introduction of `const` complicated things a little, since we can now have `const` variables that have an address, but cannot be used on the left-hand side of an assignment. To be more precise, we have to distinguish between ***non-modifiable Lvalues*** (such as `const` variables) and ***modifiable Lvalues***.

non-modifiable vs. modifiable Lvalues

3.2 Functions

free vs. member functions

In C++, we can define functions that are not associated with any class. These functions are sometimes called "free functions" to distinguish them from ***member functions***, which are associated with a class. (In other object-oriented languages, member functions are often called "methods".)

C++ provides various ways of passing arguments to functions. The same rules apply to both free functions and member functions.

Terminology. In the function definition

```
double sqr(double x) { return x * x; }
```

the list (double x) is a ***parameter list*** and x is a ***parameter***. When we call the function, as in

```
cout << sqr(2.71828) << endl;
```

the expression 2.71828 is an ***argument*** that is ***passed*** to the function.

parameter vs. argument

Some authors say "formal parameter" instead of "parameter" and "actual parameter" instead of "argument". We will use the shorter (and more correct) terms ***parameter*** and ***argument***. ("Parameter" can be roughly translated from Greek as "unknown quantity". When we write sqr above, we don't know the value of x, so it is reasonable to call x a parameter.)

> ***Functions have parameters. When a function is called, arguments are passed to it.***

Passing Arguments. There are a number of ways of passing arguments in C++. Three of them 66 suffice for most applications. The program in Figure 20 on the next page includes examples of 67 these three. When run, it produces the following output:[11]

68

```
Pass by value: 65
Pass by reference: 66
Pass by constant reference: 55
```

Pass by value: the parameter is unqualified. For example, double x. When the function is called, the run-time system makes a copy of the argument and passes the copy to the function. The function can change the value of its parameter but these changes have no effect in the calling environment. This explains the output "65": the function increments its parameter k, but the argument pbv remains unchanged.

```
#include <iostream>

using namespace std;

void passByValue(int k)
{
    ++k;
    cout << k;
}

void passByReference(int & k)
{
    ++k;
    cout << k;
}

void passByConstantReference(const int & k)
{
//  ++k;                               Not allowed!
    cout << k;
}

int main()
{
    cout << "Pass by value: ";
    int pbv = 5;
    passByValue(pbv);
    cout << pbv;

    cout << endl << "Pass by reference: ";
    int pbr = 5;
    passByReference(pbr);
    cout << pbr;
//  passByReference(5);                Not allowed!

    cout << endl << "Pass by constant reference: ";
    int pbcr = 5;
    passByConstantReference(pbcr);
    cout << pbcr << endl;
}
```

Figure 20: Passing arguments in C++

reference type

Pass by reference: the parameter is qualified by `&`, which is read as "reference to". When the function is called, the run-time system passes a ***reference*** to the argument to the function. Pass by reference is usually implemented by passing an address. When the parameter is used in the function body, it is effectively a synonym for the argument. Any change made to the parameter is also made to the argument. This explains the output "`66`": the function increments its parameter `k`, and the argument `pbr` is also incremented.

An argument to be passed by reference must be a Lvalue. The call `passByReference(5)` does not compile, because 5 is not an Lvalue.

`const &`

Pass by constant reference: the parameter is qualified by `const &`, which is read "const reference to". The run-time system passes an address, but the compiler ensures that this address is used only as an Rvalue. Changing the value of the parameter in the function body is not allowed. Note that this will work for both `const` arguments (non-modifiable Lvalues) and non-`const` arguments (modifiable Lvalues) – another good reason for using `const` wherever possible.

69 Figure 21 compares ways of passing arguments in C++ and Java.

Method: pass by . . .	C++	Java
value	`T x`	primitive types (e.g., `int`)
reference	`T & x`	objects (e.g., `Integer`)
`const` reference	`const T & x`	no equivalent

Figure 21: Passing arguments in C++ and Java

70 Use the following rules when deciding how to pass an argument:

1. ***Pass small arguments by value.***

   Addresses are used for passing by reference. An address is 4 bytes or, on modern 64bit machines, 8 bytes. If an argument is smaller, or not much bigger, than an address, it is usually passed by value. In practice, this means that the standard types (`bool`, `char`, `short`, `int`, `float`, `double`, etc.) are usually passed by value, and user-defined types (that is, instances of classes) are usually passed by reference.

2. ***Pass large objects by constant reference.***

   "Constant reference" is usually considered to be the "default" mode for C++; Default is in quotes because constant reference is not the compiler's default: it must be selected explicitly by the user. It should be used for large objects, including instances of user-defined classes, except when there is a good reason not to use it.

[11] These modes are sometimes referred to as "call by value", "call by reference", etc. They mean the same thing but "pass" seems more precise than "call".

3. ***Pass by reference only when caller needs changes.***

   Pass by reference should be used ***only*** when changes made by the function must be passed back to the caller. The usual way of returning results is to use a `return` statement. However, `return` can return only one value and it is sometimes necessary to return more than one value. In these and similar situations, reference parameters may be used to return several items of information.

C++ `const &` vs. Java `final`

Note that C++'s constant reference is significantly different from Java's `final` modifier. In Java, even when you declare a parameter as final:

```
void foo( final Student s )     // Java final is not const!
```

you can still call member functions of `s` within `foo`, such as `s.fail()`, which change the state of `s`.

3.3 Working with files

One of the nice features of C++ streams is that they make most kinds of input and output look the same to the programmer. The program in Figure 22 on the next page asks the user for a file name, reads a list of numbers from the file, and displays the mean. 71 72

Points of note include:

- The directive "`#include <fstream>`" is required for any program that uses input or output files.
- The type `ifstream` is the type of input streams. The variable `fin` is an instance of this type.
- The constructor for `ifstream` needs a file name. Curiously, it cannot accept the file name as a `string`; instead, it requires a C style string, of type `const char*`. Consequently, we have to use the conversion function `c_str` to convert the `string` to a `const char*`. Calling the constructor with a file name has the effect of opening the file.

  stream file names must be C style strings, not `string` objects!

- After the file has been opened, we use `fin` to read from the file in exactly the same way that we use `cin` to read from the keyboard.
- Recall that `fin >> observation` returns the updated input stream as its value. There are two possibilities:

  1. A value has been read and the stream is in a "good" state. In this case, the value of `fin >> observation` will be considered `true`.
  2. A problem was encountered and the stream is in a "bad" state. In this case, the value of `fin >> observation` will be considered `false`.

  In case 2, the most likely "problem" is that we have reached the end of the file.

- When the program has finished reading data from the file, it calls `fin.close()` to close the stream. This step is not strictly necessary, because the destructor, called at the end of the scope, would close the stream if it was still open. Nevertheless, it is good practice to explicitly close a stream that is no longer required by the program.

```
#include <iostream>
#include <fstream>
#include <string>

int main()
{
    cout << "Enter file name: ";
    string fileName;
    cin >> fileName;
    ifstream fin(fileName.c_str());

    int obsCount = 0;
    double sum = 0;
    double observation;
    while (fin >> observation)
    {
        sum += observation;
        ++obsCount;
    }
    fin.close();
    cout <<
        "The mean of " << obsCount <<
        " observations is " << sum / obsCount <<
        endl;
}
```

Figure 22: Finding the mean from a file of numbers

73

- The operations of constructing and opening a file can be separated. Instead of

  ```
  ifstream fin(fileName.c_str());
  ```

  open() a stream

  we could have written:

  ```
  ifstream fin;
  // ....
  fin.open(fileName.c_str());
  ```

- Whenever a program tries to open a file for input, there is a possibility that the file does not exist. As above, we can use the stream object as a boolean to check whether the stream was opened successfully:

  ```
  ifstream fin;
  fin.open(fileName.c_str());
  if (!fin)
  {
      cerr << "Failed to open " << fileName << endl;
      return;
  }
  ```

- Another possibility would be to trigger an assertion failure when a file cannot be opened:

  ```
  ifstream fin;
  fin.open(fileName.c_str());
  assert(fin);
  ```

  However, this would go against the recommendations of Section 2.7 on page 34: There are many reasons why opening a file might fail, and the failure does not imply that there is a ***logical*** fault in the program.

3.3.1 Writing to a file

Writing is very similar to reading. The class for output files is `ofstream`. We use the insert operator `<<` to write data to the file. The methods `open` and `close` work in the same way as they do for input files. Figure 23 shows a very simple program that writes random numbers to an output file. Note that you need to `#include <cstdlib>` in addition to `fstream`. 74

random numbers with `rand()`

```
int main()
{
    ofstream fout("randomnumbers.txt");
    for (int n = 0; n != 50; ++n)
        fout << rand() << '\n';
    fout.close();
    return 0;
}
```

Figure 23: Writing random numbers to a file

3.3.2 Stream States

In general, the states of an input stream are ***good***, ***bad***, and ***end-of-file***. These are not mutually exclusive. If the state is ***good***, it cannot also be either ***bad*** or ***end-of-file***. However, if the state is ***bad***, it may or may not be ***end-of-file***. The input statement `cin >> n`, where `n` is an integer, for example, will put the stream into a ***bad*** state if the next character in the stream is not a digit or "–" (the minus sign). But reading can continue after calling `cin.clear()` to clear the bad state.

This section provides a brief overview of stream states. For details, consult a reference work, such as (Langer and Kreft 2000, pages 31–35) or (Josuttis 2012, Chapter 15.4).

The state of the stream is represented by four bits; Figure 24 on the following page shows their names and meanings. Here are some examples of how the bits can get set: 75

- The program wants to read an integer and the next character in the stream is `'x'`. After the input operation, `failbit` is set and the stream position is unchanged.
- The program wants to read an integer. The only characters remaining in the file are "white space" (blanks, tabs, and newlines). After the input operation, `failbit` and `eofbit` are both set and the stream is positioned at end-of-file.

Name	Meaning
`goodbit`	The stream is in a “good” state – nothing’s wrong
`eofbit`	The stream is positioned at end-of-file – no more data can be read
`failbit`	An operation failed but recovery is possible
`badbit`	The stream has “lost integrity” and cannot be used any more

Figure 24: Stream bits and their meanings

- The program wants to read an integer. The only characters remaining in the file are digits. After the input operation, `eofbit` is set and the stream is positioned at end-of-file.

- The program wants to read data from a disk file but the hardware (or operating system) reports that the disk is unreadable. After the operation, `badbit` is set and the stream cannot be used again. 76

Figure 25 on the next page shows how to test the stream bits. These functions are members of the stream classes. For example, if `fin` is an input file stream, then `fin.bad()` tests its `badbit`. There are several things to note about these functions:

- The first five return `bool` values – that is, `true` or `false`.

- `fail()` returns `true` if `failbit` is set ***or*** if `badbit` is set.

- `operator!()` is called by writing `!` before the stream name, as in

  ```
  if (!fin)
      // failbit is set or badbit is set
  else
      // file is OK
  ```

- `operator void*()` is called by writing the stream name in a context where an expression is expected. For example:

  ```
  if (fin)
      // file is OK
  else
      // failbit is set or badbit is set
  ```

- `operator void*()` is also called when we write, for example

  ```
  if (fin >> data)
  ```

Output streams use the same state bits, but it is not often necessary to use them. An output stream is always positioned at end-of-file, ready for the next write. Output operations fail only when something unusual happens, such as a disk filling up with data.

Function	Value/Effect
`bool good()`	None of the error flags are set
`bool eof()`	`eofbit` is set
`bool fail()`	`failbit` is set or `badbit` is set
`bool bad()`	`badbit` is set
`bool operator!()`	`failbit` is set or `badbit` is set
`operator void*()`	Null pointer if `fail()` and non-null pointer otherwise
`void clear()`	Set `goodbit` and clear the error bits

Figure 25: Reading and setting the stream bits

3.3.3 Summary of file streams

The stream class hierarchy is quite large. For practical purposes, there are four useful kinds of stream: 77

`ifstream`: input file stream

`ofstream`: output file stream

`istringstream`: input string stream

`ostringstream`: output string stream

We will discuss string streams later. For each of these kinds of stream, there is another with "`w`" in front (for example, `wifstream`). These are streams of "wide" (16–bit or Unicode) characters.[12] Unicode

File streams are opened by providing a file or path name. The name can be passed to the constructor or to the `open` method. When a file is no longer needed, the `close` method should be called to close it.

The extract operator `>>` reads data from an input stream. The insert operator `<<` writes data to an output stream. The right operand of an extractor must be an Lvalue. The right operand of an inserter is an Rvalue.

The right operand of an extractor or inserter may also be a ***manipulator***. Manipulators may extract or insert data, but they are usually used to control the state of the stream. We have already seen `endl`, which writes a new line and then flushes the buffer of an output stream. stream manipulators

Although `endl` is defined in `iostream`, most other manipulators are not. They are defined in `iomanip` and so we have to write `#include <iomanip>` in order to use them. Here is a selection of commonly used manipulators for output streams:

`left`: Start left-justifying output data (appropriate for strings) 78

`right`: Start right-justifying output data (appropriate for numbers)

`setprecision(`n`)`: Put n digits after the decimal point for `float` and `double` data

`setw(`n`)`: Write the next field using at least n character positions (only applies to the next field, not subsequent fields) 79

[12] C++11 added two more types for dealing with Unicode, `char16_t` and `char32_t`.

`fixed`: Use fixed-point format for `float` and `double` data (for example, `3.1415926535`)

`scientific`: Use “scientific” format for `float` and `double` data (for example, `1.3e12`)

80 For example, the program shown is Figure 26 displays:

81

```
Alice          41    1169699.780
Boris        6500    3014133.560
Ching        1478    7915503.960
Daoust       4464    1605672.250
```

3.4 End-of-file Handling

82

Figure 27 on the next page shows a program that reads a list of numbers, computes their mean, 83 and displays it. This is what the console window looks like after running this program:

```
#include <iostream>
#include <iomanip>
#include <string>
#include <vector>
#include <cstdlib>

using namespace std;

int main()
{
    vector<string> names;
    names.push_back("Alice");
    names.push_back("Boris");
    names.push_back("Ching");
    names.push_back("Daoust");

    for ( vector<string>::const_iterator it = names.begin();
          it != names.end();
          ++it)
    {
        cout << left << setw(8) << *it;
        cout << right << setw(10) << rand() % 10000;
        cout << fixed << setprecision(3) << setw(15) <<
                rand() * (rand() / 100.0);
        cout << endl;
    }
    return 0;
}
```

Figure 26: Output manipulators

```
Enter observations, terminate with ^D:
2.6 9.32 5.67 2.1 6.78 5.1 ^D
The mean of 6 observations is 5.26167
```

```
int main()
{
    cout << "Enter observations, terminate with ^D:" << endl;
    int obsCount = 0;
    double sum = 0;
    double observation;
    while (cin >> observation)
    {
        sum += observation;
        ++obsCount;
    }
    cout <<
        "The mean of " << obsCount <<
        " observations is " << sum / obsCount <<
        endl;
    return 0;
}
```

Figure 27: Finding the mean

CTRL–D vs. CTRL–Z for end-of-file

The character `^D` (CTRL–D on the keyboard) is used to indicate the end of an input stream reading from the keyboard (i.e., `cin`), for Unix/Linux/MacOS – on Windows, end-of-file is indicated with `^Z`.[13]

The loop in this program, as shown in Figure 22 on page 48, is controlled by

```
while (cin >> observation)
```

The condition `cin >> observation` raises two questions: First, why does it work? Second, why do other, seemingly more natural constructions, ***not*** work?

Here is why it works:

1. The expression `cin >> observation` returns an updated value of `cin`. Note that this is ***not*** a boolean value (`true` or `false`), but rather of type `istream`. Consequently,

   ```
   if (cin >> observation) ....
   ```

   is equivalent to

   ```
   cin >> observation;
   if (cin) ....
   ```

[13] Note that not all C++ implementations handle EOF from keyboard input in a consistent way, so you might get different results when experimenting with this.

2. When `cin` appears in a condition context, the compiler attempts to convert it into something that can be considered boolean. By means of a technical trick, this boolean value is `true` if the stream is in a "good" state and `false` if the stream is in a "bad" state.
3. If the stream is in a good state, then the operation `cin >> observation` must have succeeded and a value for `observation` was read successfully.
4. If the stream is in a bad state, the operation failed and the value of `observation` is undefined.
5. Looking at the way in which `while (cin >> observation)` is used in the program above, we see that everything works out nicely: the program ***either*** successfully reads a value and processes it ***or*** does not manage to read a value and terminates the loop.

The "technical trick" mentioned in step 2, in case you are interested, is as follows. Since the compiler cannot interpret an instance of `istream` (the class of `cin`) as a boolean, it looks for a conversion that would enable it to do so. Class `istream` provides a conversion from `istream` to `void*`, the type of pointers that point to nothing in particular. If the stream is in a good state, the result of this conversion is a non-null pointer (probably, but not necessarily, the address of the stream object), which is considered `true`. If the stream is in a bad state, the conversion yields a null pointer, which is zero, and is therefore considered `false`.

There are several reasons why a stream can get into a bad state. The most likely reason, and the one we expect here, is that the program has reached the end of the stream (indicated, in this example, by the `^D` key). Another reason is that a disk has failed, although obviously this does not apply to `cin`.

```
void v1()
{
   vector<int> v;
   int k;
   while (cin >> k)
   {
      v.push_back(k);
   }
   for (vector<int>::const_iterator it = v.begin(); it != v.end(); ++it)
      cout << *it << ' ';
   cout << endl;
}
```

Figure 28: Testing end-of-file

The second question we have to answer is: why do other approaches not work? Problems arise because the normal input mode skips blanks to find the next datum. We will use the program in Figure 28 to explore the potential problems: This program works correctly whether or not there is white space between the last datum and the end of the file. In this example and subsequent examples, end-of-file is indicated by `^D`. Note the blank after `4` in the second example.

```
1 2 3 4^D
1 2 3 4
```

```
1 2 3 4 ^D
1 2 3 4
```

Class `stream` provides a function `eof` ("end-of-file") that yields `false` except at the end of the stream, where it returns `true`. A function that returns a boolean value is called a ***predicate***; `eof` is therefore a predicate. We can test for end-of-file *before* attempting to read the input: `eof()` 84

```
while (!cin.eof())
{
   cin >> k;
   v.push_back(k);
}
```

This is what happens:

```
1 2 3 4^D
1 2 3 4

1 2 3 4 ^D
1 2 3 4 4
```

If there is no white space after the last datum, this code works correctly. If there is a blank, the following sequence of events occurs:

- The stream reads the last datum, `4`, correctly.
- Since the next character is a blank, `cin.eof()` is `false`.
- The stream reads the blank, finds no integer, and puts the stream into a "bad" state. The value of `k` is not changed.
- The value in `k`, which is still `4`, is stored in the vector *again*.

Another possibility is to test for end-of-file *after* reading: 85

```
while (true)
{
   cin >> k;
   if (cin.eof())
      break;
   v.push_back(k);
}
```

This code gives the following results:

```
1 2 3 4^D
1 2 3

1 2 3 4 ^D
1 2 3 4
```

If there is no blank after `4` then, after this datum has been read, `cin.eof()` is `true`. The loop terminates and `4` is not stored in the vector. When there is a blank, the program works correctly.

3.5 Storing Data: STL Containers

For a few applications, such as computing an average, it is sufficient to read values; we do not have to store them. For most applications, it is useful or necessary to store values as we read them. The program in Figure 29 shows one way of doing this. 86

The new feature in this program is

```
vector<double> observations;
```

STL `vector<>`

This declaration introduces `observations` as an instance of the class `vector<double>`. The ***template class*** `vector` is one of the ***containers*** provided by the Standard Template Library (STL). A template class is ***generic*** – it stands for many possible classes – and must be instantiated by providing a type. In this case, the type is `double`, giving us a `vector` of `double`s. When we use a vector, we must also write

```
#include <vector>
```

```
int main()
{
    cout << "Enter file name: ";
    string fileName;
    cin >> fileName;
    ifstream fin(fileName.c_str());

    vector<double> observations;
    double obs;

    while (fin >> obs)
    {
        observations.push_back(obs);
    }
    fin.close();

    for ( vector<double>::size_type i = 0; i != observations.size(); ++i )
        cout << observations[i] << '\n';
}
```

Figure 29: Storing values in a vector

The relation between a generic class and an instantiated class is analogous to the relation between a function and a function application: 87

	Generic	Application
Function	log	$\log x$
Class	`vector`	`vector<double>`

There are various things we can do with a vector. The operation v.`push_back`(x) inserts the value x at the back end of the vector v. We do not have to specify the initial size of the vector, and we do not have to worry about how much stuff we put into it; storage is allocated automatically as the vector grows to the required size.

v.`push_back`(x)

The argument for `push_back` is passed ***by value***. This means that the vector gets its own copy of the argument, which is usually what we want.

The method `size` returns the number of elements in the vector. As with `string`, the type of the size is not `int` but `vector<double>::size_type`. This is the type that we use for the control variable `i` in the final `for` loop of the program.

A vector can be subscripted, like an array. If v is a vector, $v[i]$ is the i^{th} element. In the program, `observations[i]` gives the $\mathtt{i}^{th}$ element of the vector of observations.

Vector indexes are not checked!

This means that, when we write `observations[20]`, for example, there is no check that this element exists.[14] If it doesn't exist, the result will be garbage. Even worse, if we use this expression on the left of an assignment, we can write into any part of memory!

`<sermon>` This is an astonishing oversight. Here is Tony Hoare (1981):

> *"...we asked our customers whether they wished us to provide an option to switch off these [array index] checks in the interests of efficiency on production runs. Unanimously, they urged us not to – they already know how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson. In any respectable branch of engineering, failure to observe such elementary precautions would have long ago been against the law."*

In this example, the language was Algol 60 and the computer was an Elliott 503. The 503 weighed several tons, had 8K words of memory, and needed 7 μsec to add two numbers.[15]

Another 30 years have passed, and it is more than 40 years since Hoare's customers wanted subscript checking. Today's computers are around 100,000 times faster than the 503, and programmers are still concerned about the time taken to check subscripts. A high proportion of viruses, worms, phishers, and other kinds of malicious software exploits precisely this loophole. `</sermon>`

Fortunately, there are safer ways of accessing the elements of a vector than using subscripting. One way is to use the function `at`. The call `array.at(i)` has the same effect as `array[i]` but checks that the index is within range and throws an exception otherwise.

`array.at(i)` provides index checking

[14] Unlike in Java, which would throw an exception in this case.

[15] "Some Old Computers," http://members.iinet.net.au/~daveb/history.html

Iterators are another alternative to subscripts and, in many cases, a better one. An iterator is an object that keeps track of the objects in a container and provides facilities for accessing them. The type of the iterator that we need is

```
vector<double>::const_iterator
```

Two of its values are `observations.begin()`, which refers to the first element of the vector `observations`, and `observations.end()`, which refers to the first element ***not*** in the vector – that is, one *past* the end. Iterators have increment (`++`) and decrement (`--`) operators. Iterators also have pointer-like behaviour: dereferencing an iterator yields an element of the container.

88 Putting all these things together, we can write the following loop to access the elements in the vector `observations`:

```
for (  vector<double>::const_iterator i = observations.begin();
       i != observations.end();
       ++i )
    cout << *i << '\n';
```

This code has two significant advantages over the original version:

- Using the iterator and, specifically, the function `end`, ensures that we access exactly the elements that are stored.
- This code can be used with other kinds of container.

For example, if we replace each occurrence of `vector` by `list`:

```
list<double> observations;
....
for (  list<double>::const_iterator i = observations.begin();
   i != observations.end();
   ++i )
cout << *i << '\n';
```

the program compiles and runs with exactly the same effect.

89 However, if we replace `!=` in the loop condition by `<`, the compiler complains – see Figure 30. The problem is that list iterators, unlike vector iterators, provide equality (`==` and `!=`) but not ordering (`<`, etc.).

```
error: no match for 'operator<' in
    'i < observations.std::vector<_Tp, _Alloc>::end
    [with _Tp = double, _Alloc = std::allocator<double>]()'
```

Figure 30: Complaints from the compiler

In this case, the compiler diagnostic starts with `no match for 'operator<'` and, since introducing `<` was the only change we made to the program, it is not hard to figure out that it is this operator that caused the problem. Unfortunately, the STL can produce even worse error messages that can be very hard to interpret.[16]

Iterator loops. C++11 introduced a new kind of loop, the range-based `for` loop, which works similar to the *for-each* loop added in Java 5:

```
for ( double i: observations )
    cout << i << '\n';
```

Note that you will need a compiler supporting C++11; in `gcc` it was added in version 4.6. 90

Sorting the vector of observations is very easy. Only one extra line of code is required: 91

```
sort(observations.begin(), observations.end());
```

However, `sort` is not a part of `vector`; it is one of the algorithms provided by the STL. Consequently, we also need the directive

```
#include <algorithm>
```

3.6 Summary of the Standard Template Library (STL)

The STL provides containers, iterators, algorithms, function objects, and adaptors:

Containers are data structures used to store collections of data of a particular type. The operations available for a container, and the efficiency of the operations, depend on the underlying data structure. For example, `vector` provides array-like behaviour: elements can be accessed randomly, but inserting or deleting elements may be expensive. In contrast, a `list` can be accessed sequentially but not randomly, and provides efficient insertion and deletion.

The containers also include `set` for unordered data and `map` for key/value pairs without duplicates. The containers `multiset` and `multimap` are similar but allow duplicates.

Iterators provide access to the elements stored in containers. They are used to ***traverse*** containers (i.e., visit each element in turn) and to specify ***ranges*** (i.e., groups of consecutive elements in a container).

Algorithms provide standard operations on containers. There are algorithms for finding, searching, copying, swapping, sorting, and many other applications.

Function objects are objects that behave as functions. Function objects are needed in the STL because the compiler can sometimes select an appropriate object in a context where it could not select an appropriate function. However, there are also other uses of function objects.

[16] Some programmers write Perl scripts to parse compiler diagnostics and pick out the key parts.

Adaptors allow interface modification and increase the flexibility of the STL. Suppose we want a stack. There are several ways to implement a stack: we could use a vector, or a list, or some other type. The STL might provide a class for each combination (`StackVector`, `StackList`) and perhaps `Stack` as a default.

In fact, the STL separates abstract data types (such as `stack`) and representations (such
92 as `vector` and `list`) and provides adaptors to fit them together. Thus we have:

`stack<T>`: stack of objects of type `T` with default implementation

`stack<T, vector<T> >`: stack of objects of type `T` with vector implementation

`stack<T, list<T> >`: stack of objects of type `T` with list implementation

Assuming that the STL provides M container classes and N algorithms, it is tempting to assume that there are $M \times N$ ways of using it, because we should be able to apply each algorithm to each container. However, this is not in fact how the STL works. Instead:

- A container/algorithm combination works only if the algorithm is appropriate for the data structure used by the container.

STL guarantees performance as complexity class

- If the combination does work, its performance is guaranteed in terms of a complexity class, e.g., $\mathcal{O}(N)$.

We have already seen an example of this in Section 3.5 on page 56. Suppose that `i` and `j` are iterators for a container and that `*i` and `*j` are the corresponding elements. We would expect `==` and `!=` to be defined for the iterators. It is also reasonable to expect `++`, because iterators are supposed to provide a way of stepping through the container. But what does `i < j` mean? Presumably, something like "`*i` appears before `*j` in the container". This is easy to evaluate if the container is a `vector`, because vectors are indexed by integers (or perhaps pointers), which can be compared. But evaluating `i < j` for a linked list is inefficient, because it requires traversing the list. This is why the iterator for a vector provides `<` but the iterator for a list does not.

It is important to check that the algorithm you want to use works with the container that you
93 are using. The penalty for not checking is weird error messages. For example,

```
void main()
{
    std::vector<int> v;
    std::stable_sort(v.begin(), v.end());
}
```

compiles correctly, but

```
void main()
{
    std::list<int> v;
    std::stable_sort(v.begin(), v.end());
}
```

94 produces the message

```
stl_algo.h: In function ‘void __merge_sort_loop<_List_iterator
  <int,int &,int *>, int *, int>(_List_iterator<int,int &,int *>,
  _List_iterator<int,int &,int *>, int *, int)’:
stl_algo.h:1448:   instantiated from ‘__merge_sort_with_buffer
  <_List_iterator<int,int &,int *>, int *, int>(
   _List_iterator<int,int &,int *>, _List_iterator<int,int &,int *>, int *, int *)’
stl_algo.h:1485:   instantiated from ‘__stable_sort_adaptive<
  _List_iterator<int,int &,int *>, int *, int>(_List_iterator
  <int,int &,int *>, _List_iterator<int,int &,int *>, int *, int)’
stl_algo.h:1524:   instantiated from here
stl_algo.h:1377: no match for ‘_List_iterator<int,int &,int *> & -
  _List_iterator<int,int &,int *> &’
```

Why doesn't the STL generate more useful diagnostics? The reason is that it is based on templates. The compiler first expands all template applications and then tries to compile the resulting code. If the code contains errors, the compiler cannot trace back to the origin of those errors, saying perhaps "`list` does not provide `stable_sort`", but can only report problems with the code that it has.

Various objects and values associated with containers have types. These types may depend on the type of the elements for the container. For example, the type of an iterator for `vector<int>` may not be the same as the type of an iterator for `vector<double>`. Consequently, the container classes must provide the types we need. In fact, we have already seen expressions such as `vector<double>::const_iterator`, which is the type of `const` iterators for a vector of `double`s.

Using `typedef`. These type names can get quite long. It is common practice to use `typedef` directives to abbreviate them. A `typedef` has the form 95

`typedef` ⟨*type expression*⟩ ⟨*identifier*⟩

and defines ⟨*identifier*⟩ to be a synonym for ⟨*type expression*⟩. For example, after

```
typedef vector<double>::const_iterator vci;
```

we can write `vci` instead of the longer expression.

References

Josuttis, N. M. (2012). *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley. http://www.cppstdlib.com/.

Langer, A. and K. Kreft (2000). *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley.

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.

4 Application: Grading a class

4.1 Problem Statement 63
4.2 First version 63
4.3 Program Structure 72
4.4 Structuring the grading program 75
4.5 Dependencies 78
4.6 Strings and characters 81

In this lecture, we provide a first introduction to *class design* in C++. The program developed in this chapter is similar, but not identical, to the grading program described in Chapter 4 of (Koenig and Moo 2000).

4.1 Problem Statement

Here is a statement of the problem that the grading program solves: 97

> The program reads a file of marks (e.g., Figure 31 on the next page) and writes a 98
> report of grades (e.g., Figure 32 on the following page). Each line of the marks file 99
> consists of a name, a mark for the midterm, a mark for the final, and marks for assignments. The number of assignments is not fixed; students do as many as they want to. The name is a single name with no embedded blanks.
>
> A line of the output file is similar to a line of the input file, but the first number is the total mark, computed as 30% of the midterm mark plus 60% of the final mark plus the median of the assignments. The output is written twice, once sorted by name, and once sorted by total mark.

4.2 First version 100

Figure 33 on page 65 shows the main program. The program is designed with two objectives in mind: 101

- Use an object to store a student record
- Put as much problem-specific information as possible into the corresponding class

The first paragraph of the program asks the user for a file name and tries to open the file. The second paragraph declares the principal data object of the program, a vector of `Students`. From this paragraph, we can tell that the `Student` class must provide a reading capability (`>>`) and a method `process` to compute the final mark.

It is important that the argument to `push_back` is ***passed by value***. If it was passed by reference, each entry in the vector `classData` would refer to the same local variable, `stud`!

The last part of the program opens an output file and writes the data to it twice, first sorted by name and then sorted by total marks. From this section, we deduce that the `Student` class must provide two sorting functions, ltNames and ltMarks. We also need a free function showClass to write the class list.

102 The function showClass is straightforward; it is shown in Figure 34 on the facing page. The output stream os is ***passed by reference*** because the function will change it when writing. The student data is ***passed by constant reference*** because it will not be changed by the function. It uses an iterator to traverse the vector of marks data. The statement

```
os << *it << endl;
```

requires class Student to provide an ***inserter*** – a function that overloads the operator (<<).

```
Thomas  47 83 9 8 4 7 6 9 8
Georges 36 88 8 6 7 4 9 7 8 7
Tien    49 91 9 6 7 8 5 6 7
Lei     41 82 8 8 8 8
Oanh    45 76 9 9 8 9 9 8 8 9
Lazybones  31 45
Mohamad 39 99 8 6 7 9 5 6 9
Jane    36 64 7 5 8
```

Figure 31: Input for grading program

```
Sorted by name:
Georges    67.5 36 88  8  6  7  4  9  7  8  7
Jane       52.6 36 64  7  5  8
Lazybones  33.2 31 45
Lei        65.4 41 82  8  8  8  8
Mohamad    74.2 39 99  8  6  7  9  5  6  9
Oanh       63.6 45 76  9  9  8  9  9  8  8  9
Thomas     67.2 47 83  9  8  4  7  6  9  8
Tien       71.4 49 91  9  6  7  8  5  6  7

Sorted by marks:
Lazybones  33.2 31 45
Jane       52.6 36 64  7  5  8
Oanh       63.6 45 76  9  9  8  9  9  8  8  9
Lei        65.4 41 82  8  8  8  8
Thomas     67.2 47 83  9  8  4  7  6  9  8
Georges    67.5 36 88  8  6  7  4  9  7  8  7
Tien       71.4 49 91  9  6  7  8  5  6  7
Mohamad    74.2 39 99  8  6  7  9  5  6  9
```

Figure 32: Output from grading program

103 Figure 35 on page 66 shows the declaration for class Student. There are several points to note:

```
int main()
{
    string classFileName;
    cout << "Please enter class file name: ";
    cin >> classFileName;
    ifstream ifs(classFileName.c_str());
    if (!ifs)
    {
        cerr << "Failed to open " << classFileName << endl;
        return 1;
    }

    vector<Student> classData;
    Student stud;
    while (ifs >> stud)
    {
        stud.process();
        classData.push_back(stud);
    }

    ofstream ofs("grades.txt");
    sort(classData.begin(), classData.end(), ltNames);
    ofs << "Sorted by name:\n";
    showClass(ofs, classData);

    sort(classData.begin(), classData.end(), ltMarks);
    ofs << "\nSorted by marks:\n";
    showClass(ofs, classData);
}
```

Figure 33: The main program

```
void showClass(ostream & os, const vector<Student> & classData)
{
    for (  vector<Student>::const_iterator it = classData.begin();
           it != classData.end();
           ++it )
        os << *it << endl;
}
```

Figure 34: Function `showClass`

```
class Student
{
    friend ostream & operator<<(ostream & os, const Student & stud);
    friend istream & operator>>(istream & is, Student & stud);
    friend bool ltNames(const Student & left, const Student & right);
    friend bool ltMarks(const Student & left, const Student & right);
public:
    void process();
private:
    static string::size_type maxNameLen;
    string name;
    int midterm;
    int final;
    vector<int> assignments;
    double total;
};
```

Figure 35: Class `Student`

- In C++, the declaration of a class and the definitions of its functions are separate. The function definitions may be – and often are – in a different file (as described later in this chapter).

friend functions

- A class declaration may introduce functions as `friend`s. Although these functions are not member functions, they have access to the classes' private data.

`public` and `private` visibility modifiers

- The declarations `public` and `private` introduce a ***group*** of declarations with the given accessibility (unlike in `Java`, where each member has its own visibility modifier).

  Some authors put the `private` declarations before the `public` declarations. This seems backwards: the users of a class need to know about only the `public` attributes and these should therefore appear ***first***.[17]

- Functions declared within a class are called ***member functions***. Functions declared outside a class are called ***free functions***.[18] Member functions can be called only with an object, as in `obj.fun()`. Free functions are called without an object, as in `fun()`.

default constructor performs no initializations

- There is no constructor. We rely on the default constructor provided by the compiler; this constructor allocates space for the object but performs no other initialization. When we define a new instance of `Student`, we must ensure that all fields are correctly initialized.

- There is only one public method, `process`, which performs any necessary computation on the data read from the marks file.

- The private data includes the information that is read from the marks file (`name`, `midterm`, `final`, and `assignments`) and computed information, `total`.

[17] It would be even better if the `private` attributes could be hidden from users altogether. Although C++ does not allow that, documentation tools such as *Doxygen* (see Section B.3) can provide the required effect.

[18] Note that `Java` does not have free functions. However, classes such as `Math` provide static functions that are effectively the same as free functions. In `Java`, you write `Math.sqrt(x)`, in C++, you just write `sqrt(x)`.

- For formatting the output, we need to know the length of the longest name. This is an attribute of the class, not the object, and so it is declared as a `static` data member. Memory for static variables is automatically allocated at program start and cleaned up when it terminates.

static data members

- We need methods for input (`>>`) and output (`<<`); these are declared as friends.

- We need comparison functions that will be used for sorting: `ltNames` orders by student's names, and `ltMarks` orders by student's total marks.

  There is an important design choice here. The four `friend` functions cannot be member functions, because of the way they are called. The alternatives are either to provide access functions to private data in the class or to declare these functions as `friends`. Access functions should be avoided if possible, especially functions with write access, as would be required for `>>`. Although `friend` functions should be used only where necessary, they sometimes provide better encapsulation, as in this case.[19]

`friend` functions vs. member access functions

- A class declaration – just like any other declaration – must end with a '`;`'

don't forget the '`;`' at the end of a class declaration!

The next step is to complete the implementation of class `Student` by providing definitions for functions and initial values for static variables. The static data member must be initialized like this:

initializing static data members

```
string::size_type Student::maxNameLen = 0;
```

This is the ***only*** way to initialize a static data member. It has the form of a declaration rather than an assignment (the type is included) and it appears at global scope, that is, outside any function.

The public function `process` has two tasks: it maintains the length of the longest name seen so far and it calculates the total mark. Calculating the total mark requires finding the median of the assignments. The median is meaningless for an empty vector, and the median function requires a non-empty vector as its argument (see Figure 37 on the following page). Thus `process`, shown in Figure 36, calls `median` only if the student has done at least one assignment. 104

```
void Student::process()
{
    if (maxNameLen < name.size())
        maxNameLen = name.size();
    total = 0.3 * midterm + 0.6 * final;
    if (assignments.size() > 0)
        total += median(assignments);
}
```

Figure 36: Function `process` for class `Student`

In general a function should not perform two unrelated tasks, as `process` does. The rationale in this case is that `process` performs ***all*** of the processing that is needed for one student record. There might be more tasks to perform than just these two. An alternative would be to define

[19] We will discuss the undesirability of access functions later in the course, when we address class design issues.

two functions, one to update maxNameLen and the other to calculate total. These two functions would always be called together, so it makes sense to combine them into a single function.

As a general principle, it should always be possible to explain the purpose of a function with a one-line description. If you need three sentences to say what a function does, there's probably something wrong with it. We could describe the purpose of function process as "perform all calculations needed to generate the marks file".

Every function should have a complete, one-line description.

105 The median calculation is performed by the function in Figure 37. The main design issue for this function is how to pass the vector of scores. Since we have to sort the vector in order to find the median, we cannot pass it by constant reference. If we pass it by reference, the caller will get back a sorted vector. Although this does not matter much for this program, a function should not in general change the data it is given unless the caller needs the changed value. Consequently, we choose to pass the vector by value, incurring the cost of copying it.

Finally, note that median has a precondition: it does not accept an empty vector. The only use of median in this program is in the context

```
if (assignments.size() > 0)
    total += median(assignments);
```

It follows that, if the assertion fails, there is a logical error in the program.

Perhaps, after this program has been used for a few years, another programmer decides to extend it. The function median is called from another location, without the check for an empty vector. During testing, the assertion fails, and the programmer immediately sees the problem with the extension.

```
// Requires: scores.size() > 0.
double median(vector<int> scores)
{
    typedef vector<int>::size_type szt;
    szt size = scores.size();
    assert(size > 0);
    sort(scores.begin(), scores.end());
    szt mid = size / 2;
    return size % 2 == 0 ?
        0.5 * (scores[mid - 1] + scores[mid]) :
        scores[mid];
}
```

Figure 37: Finding the median

106

Figure 38 on the next page shows the comparison functions that we need for sorting. The parameter lists of these functions are determined by the requirements of the sort algorithm:

there must be two parameters of the same type, both passed by constant reference. Since we have declared these functions as `friends` of `Student`, they have access to `Student`'s private data members. The type of `name` is `string` and the type of `total` is `double`; both of these types provide the comparison operator `<`.

After sorting, the records will be arranged in increasing order for the keys. Names will be alphabetical: `Anne`, `Bo`, `Colleen`, `Dingbat`, etc. Records sorted by marks will go from lowest mark to highest mark. To reverse this order, putting the students with highest marks at the “top” of the class, all we have to do is change “<” to “>” in `ltMarks`. As a courtesy to readers, it would also be a good idea to change the name to `gtMarks`.

```
bool ltNames(const Student & left, const Student & right)
{
    return left.name < right.name;
}

bool ltMarks(const Student & left, const Student & right)
{
    return left.total < right.total;
}
```

Figure 38: Comparison functions

The compiler has to perform a number of steps to determine that these functions are called by the statements

```
sort(classData.begin(), classData.end(), ltNames);
sort(classData.begin(), classData.end(), ltMarks);
```

The reasoning goes something like this:

1. The type of the argument `classData.begin()` is `vector<Student>::const_iterator`
2. The compiler infers from this that the elements to be sorted are of type `Student`
3. The comparison functions must therefore have parameters of type `const & Student`
4. There are functions `ltNames` and `ltMarks` with parameters of the correct type 107

Figure 39 on the following page shows the extractor (`>>`) for class `Student`. It is a bit tricky, because we rely on the failure management of input streams. The key problem is this: since students complete different numbers of assignments, how do we know when we have read all the assignments? The method we use depends on what follows the last assignment: it is either the name of the next student or the end of the file. If we attempt to read assignments as numbers, either of these will cause reading to fail. Consequently, we can use the following code to read the assignments:

```
while (ifs >> mark)
    stud.assignments.push_back(mark);
```

```
istream & operator>>(istream & ifs, Student & stud)
{
    if (ifs >> stud.name)
    {
        ifs >> stud.midterm >> stud.final;
        int mark;
        stud.assignments.clear();
        while (ifs >> mark)
            stud.assignments.push_back(mark);
        ifs.clear();
    }
    return ifs;
}
```

Figure 39: Extractor for class `Student`

However, we must not leave the stream in a bad state, because this would prevent anything else being read. Therefore, when the loop terminates, we call

```
ifs.clear();
```

to reset the state of the input stream.

We assume that, if a student name can be read successfully, the rest of the record is also readable. If the name is ***not*** read successfully, the function immediately returns the input stream in a bad state, telling the user that we have encountered end of file.

What happens if there is a format error in the input? Some markers, although they are asked to provide integer marks only, include fractions. Suppose that the input file contains this line:

```
Joe    45 76 9 9 8.5 9 9 8 8 9
```

The corresponding output file contains these lines:

```
Joe       63.6 45 76   9   9   8
.5         15.2   9   9   8   8   9
```

We see that Joe has lost all his assignment marks after 8.5 and we have a new student named "`.5`". It is clear that, if this was a production program, we would have to do more input validation.

108 The inserter (`<<`) in Figure 40 on the next page does not have these complications. The main points to note are:

- The manipulators:
  - `left` aligns text to the left
  - `right` (the default) aligns text to the right
  - `fixed` chooses fixed-point (as opposed to scientific) format for floating-point numbers

```
ostream & operator<<(ostream & os, const Student & stud)
{
    os <<
        left << setw(static_cast<streamsize>(Student::maxNameLen)) <<
        stud.name << right <<
        fixed << setprecision(1) << setw(6) << stud.total <<
        setw(3) << stud.midterm <<
        setw(3) << stud.final;
    for (  vector<int>::const_iterator it = stud.assignments.begin();
           it != stud.assignments.end();
           ++it )
       os << setw(3) << *it;
    return os;
}
```

Figure 40: Inserter for class `Student`

  - `setprecision(1)` requests one decimal digit after the decimal point
  - `setw(`n`)` requests a field width of N characters

- We use the longest name to align columns. The type of the variable `Student::maxNameLen` is `string::size_type` but the type expected by `setw` is `std::streamsize`. To avoid warnings from the compiler, we cast the type. Since the cast can be performed at compile time, we use a ***static cast***: static_cast

  ```
  static_cast<streamsize>(Student::maxNameLen)
  ```

- We use an iterator to output the assignment marks.

Extractors and Inserters. All extractors and inserters follow the same pattern: 109

```
istream & operator>>(istream & is, T & val)
{
    // perform read operations for fields of T
    return is;
}

ostream & operator<<(ostream & os, const T & val)
{
    // perform write operations for fields of T
    return os;
}
```

defining operator>> and operator<<

In both cases, the function is passed a reference to a stream and returns a reference to a stream. In fact, of course, both references are to the same stream, but the state of the stream changes during the operation. The second argument for the extractor is passed by reference, because its value will be updated when the stream is read. The second argument for the inserter is passed by constant reference, because the inserter should not change it. When writing inserters and extractors, remember to return the updated stream.

110 Compiling this program requires the inclusions shown in Figure 41. The comments are not necessary, because experienced C++ programmers are familiar with these names.

```
#include <algorithm>  // sort
#include <cassert>    // assertions
#include <fstream>    // input and output file streams
#include <iomanip>    // stream manipulators
#include <iostream>   // input and output streams
#include <string>     // STL string class
#include <vector>     // STL vector class
```

Figure 41: Directives required for the grading program

4.3 Program Structure

translation unit

A C++ program consists of ***header files*** and ***implementation files***. The program is compiled as a collection of translation units. A ***translation unit*** normally consists of a single implementation file that may `#include` several header files. This is known as *separation of interface and implementation* (Weiss 2004, Section 4.11).

Building a program is a process that consists of compiling each translation unit and linking the resulting object files. A ***build*** is the result of building. Some companies have a policy such as “daily build” to ensure that an application under development can always be compiled and passes basic tests.

4.3.1 Header Files

Header files contain declarations but not definitions. This implies that the compiler:

1. does not generate any code while processing a header file
2. may read a header more than once during a build

If a header file that contains a definition such as

```
int k;
```

is read more than once during a build, the linker will complain that `k` is redefined and will not link the program. This is why it is important not to put definitions into header files.

never put definitions into header files!

Include Guards. Although a header file may be read more than once during a build, a header file should be read only once during the compilation of a translation unit. Suppose that translation unit A includes headers for translation units B and C, and these units both include `utilities.h`. To prevent the compiler from reading `utilities.h` twice, we write it in the following way: 111

```
#ifndef UTILITIES_H
#define UTILITIES_H

// Declarations for utilities.h

#endif
```

include guards

This is the standard pattern for ***all*** header files. The directives are called "include guards" (Weiss 2004, Section 4.12). You do not have to use the exact name `UTILITIES_H`, but it is important to choose a name that is unique and has some obvious connection to the name of the header file. For example, (Koenig and Moo 2000) uses `GUARD_utilities_h`.

In most cases, the guards are not logically necessary. Since header files contain only declarations, reading them more than once should not cause errors. Some header files, however, cannot be read twice, and these can cause problems if they don't have guards. A more important reason for using guards is efficiency: header files can be very long, and they may include other header files. Without the guards, the compiler may be forced to read thousands of lines of declarations that it has seen before.

#pragma once

Instead of using include guards in header files, you can write just

```
    #pragma once
```

as the first line of a header file. This directive is recognized by most modern compilers and will sometimes be slightly faster than include guards. It is used in header files generated by VC++.NET. However, it is not completely portable, because compilers implement it in different ways. Include guards have been standard since the early days of C and always work correctly. Therefore, some industry coding standards, like the Google C++ Style Guide (Google 2017), forbid the use of `#pragma once` even for Windows platforms (see Section B.13.2).

Typically, a header file will contain declarations for types, constants, functions, and classes.

A header file should `#include` anything that the compiler needs in order to process it. For example, if a class has a data member of type `string`, its header file must contain

```
#include <string>
```

It is not a good idea to include `using` declarations in header files. A header file may be included in many translation units that may not want namespaces opened for them.

Do not write `using` ***declarations in header files.***

4.3.2 Implementation Files

Implementation files contain definitions for the objects declared in header files. An implementation file is processed only once during a build. An implementation file should `#include` its own header file and header files for anything else that it needs.

As a general rule, the first `#include` directive should name the header file corresponding to the
112 implementation file. For example, `utilities.cpp` would have the following structure:

```
#include "utilities.h"

// #includes for other components of this program

// #includes for library components

// definitions of the utilities
```

Header and implementation files create ***dependencies***, which are discussed in Section 4.5 on page 78 below. A header file may depend on other header files and an implementation file may depend on one or more header files. A file should never depend on an implementation file; in other words, you should never write

```
#include "something.cpp"
```

Do not `#include` ***implementation files.***

When an implementation file `#include`s header files, the compiler obviously reads ***all*** of the files. Amongst other things, it checks that declarations in header files match definitions in implementation files. It is important to realize that the checking is not complete. For example,
113 the header file

```
#ifndef CONFLICT_H
#define CONFLICT_H

double mean(double values[]);

#endif
```

and the implementation file

```
#include "conflict.h"
#include <vector>

double mean(std::vector<double> values)
{
    // ....
}
```

will ***not*** produce any error messages. Since C++ allows functions to be overloaded, it assumes that the two versions of `mean` are different functions and that the vector version will be declared somewhere else. If the program calls either version, the linker will produce an error message.

4.4 Structuring the grading program

We split the grading program of Section 4.2 on page 63 into three translation units: 114

1. class `Student`
2. function `median`
3. the main program

The translation unit for function `median` is rather small, but it demonstrates the idea of splitting of generally useful functions in a larger application. For a small program such as this one, we could have put `Student` and `median` into the same implementation file.

4.4.1 Translation unit `Student` 115

116

Figure 42 on the next page shows the header file for class `Student`, `student.h`. There is an `#include` directive for each library type mentioned in the class declaration. The class declaration is unchanged from the original program.

Although function `showClass` is not a `friend` of class `Student`, it is closely related to the class; consequently, we put its declaration in `student.h`.

Figures 43 and 44 show the implementation file for class `Student`, `student.cpp`. The first `#include` directive includes `student.h`; this ensures that `student.h` does not depend on anything that it does not mention (if it did, the compiler would fail while reading it). Then we include `median.h` for this program, and finally the library types that we need. Since `student.h` includes `iostream`, `string`, and `vector`, we need only include `iomanip` for the output statements.

The implementation file `student.cpp`, shown in Figures 43 and 44 implements the member 117
function of `Student`, `process`, and the `friend` functions. It also initializes the static data 118
member `maxNameLen`.
119

120

4.4.2 Translation unit `median`

The header and implementation files for `median` are both short: see Figure 45 on page 78 and 121
Figure 46 on page 79. In a more typical application, other useful functions might be incorporated 122
into a single translation unit.

```
#ifndef STUDENT_H
#define STUDENT_H

#include <iostream>
#include <string>
#include <vector>

class Student
{
    friend std::ostream & operator<<(std::ostream & os,
        const Student & stud);
    friend std::istream & operator>>(std::istream & is,
        Student & stud);
    friend bool ltNames(const Student & left, const Student & right);
    friend bool ltMarks(const Student & left, const Student & right);
public:
    void process();
private:
    // Data read from file
    std::string name;
    int midterm;
    int final;
    std::vector<int> assignments;

    // Data computed by process
    double total;
    static std::string::size_type maxNameLen;
};

void showClass(std::ostream & os,
    const std::vector<Student> & classData);

bool ltNames(const Student & left, const Student & right);
bool ltMarks(const Student & left, const Student & right);

#endif
```

Figure 42: `student.h`: header file for class `Student`

4.4.3 Translation unit for the main program

123 The last step is to write an implementation file for the main program, `grader.cpp` (see Figure 48
124 on page 80). We do not need a header file (which would be called `grader.h`) because no other
125 translation unit refers to anything in the main program. It is a good idea in general to avoid
dependencies on the main program.

This translation unit includes only the header files for other translation units that it needs –

```
#include "student.h"
#include "median.h"

#include <iomanip>

using namespace std;

string::size_type Student::maxNameLen = 0;

void Student::process()
{
    if (maxNameLen < name.size())
        maxNameLen = name.size();
    total = 0.3 * midterm + 0.6 * final;
    if (assignments.size() > 0)
        total += median(assignments);
}

ostream & operator<<(ostream & os, const Student & stud)
{
    os <<
        left << setw(static_cast<streamsize>(Student::maxNameLen)) <<
        stud.name << right <<
        fixed << setprecision(1) << setw(6) << stud.total <<
        setw(3) << stud.midterm <<
        setw(3) << stud.final;
    for (  vector<int>::const_iterator it = stud.assignments.begin();
           it != stud.assignments.end();
           ++it )
       os << setw(3) << *it;
    return os;
}

istream & operator>>(istream & ifs, Student & stud)
{
    if (ifs >> stud.name)
    {
        ifs >> stud.midterm >> stud.final;
        int mark;
        stud.assignments.clear();
        while (ifs >> mark)
            stud.assignments.push_back(mark);
        ifs.clear();
    }
    return ifs;
}
```

Figure 43: `student.cpp`: implementation file for class `Student`: first part

```
bool ltNames(const Student & left, const Student & right)
{
    return left.name < right.name;
}

bool ltMarks(const Student & left, const Student & right)
{
    return left.total < right.total;
}

void showClass(ostream & os, const vector<Student> & classData)
{
    for (  vector<Student>::const_iterator it = classData.begin();
           it != classData.end();
           ++it )
        os << *it << endl;
}
```

Figure 44: `student.cpp`: implementation file for class `Student`: second part

```
#ifndef MEDIAN_H
#define MEDIAN_H

#include <vector>

double median(std::vector<int> scores);

#endif
```

Figure 45: `median.h`: header file for function `median`

`student.h` in this case – and any library types.

126

4.5 Dependencies

Figure 47 on the next page shows the dependencies between the files of the grading program. Dependencies on libraries are not shown. File X ***depends on*** file Y if the compiler must read Y in order to compile X. In general:

- Implementation files depend on header files
- Header files may depend on other header files
- An implementation file ***never*** depends on another implementation file
- There must be no circular dependencies

low coupling

- Fewer dependencies are better (few dependencies = “low coupling”)

```
#include "median.h"

#include <algorithm>
#include <cassert>

using namespace std;

// Requires: scores.size() > 0.
double median(vector<int> scores)
{
    typedef vector<int>::size_type szt;
    szt size = scores.size();
    assert(size > 0);
    sort(scores.begin(), scores.end());
    szt mid = size / 2;
    return size % 2 == 0 ?
        0.5 * (scores[mid - 1] + scores[mid]) :
        scores[mid];
}
```

Figure 46: `median.cpp`: implementation file for function `median`

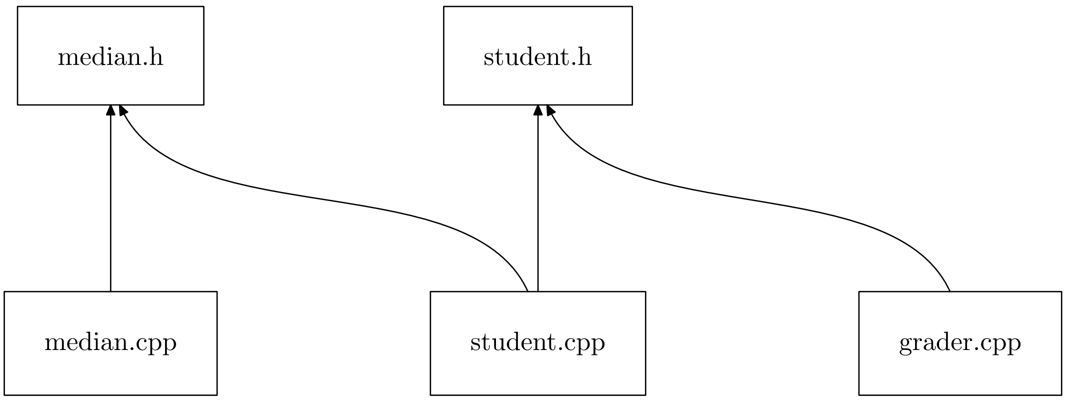


Figure 47: Dependencies in the grading program

dependencies and compilation time

Dependencies have an important effect on compilation time. A file with a high in-degree will trigger extensive recompilation when it is changed.

In Figure 47, a change to either `median.h` or `student.h` will cause two of the three implementation files to be recompiled. If `grader.cpp` depended on `median.h`, changing `median.h` would cause all three implementation files to be recompiled.

In a small program like the grader, the effect of dependencies on compilation is negligible. In large programs, the effect can be significant. Large programs require hours or even days to compile. Some header files are used by hundreds or even thousands of implementation files. A

```
#include "student.h"

#include <algorithm>
#include <fstream>
#include <string>
#include <vector>

using namespace std;

int main()
{
    // Attempt to open file provided by user
    string classFileName;
    cout << "Please enter class file name: ";
    cin >> classFileName;
    ifstream ifs(classFileName.c_str());
    if (!ifs)
    {
        cerr << "Failed to open " << classFileName << endl;
        return 1;
    }

    // Process the file
    vector<Student> classData;
    Student stud;
    while (ifs >> stud)
    {
        stud.process();
        classData.push_back(stud);
    }

    // Print reports
    ofstream ofs("grades.txt");
    sort(classData.begin(), classData.end(), ltNames);
    ofs << "Sorted by name:\n";
    showClass(ofs, classData);

    sort(classData.begin(), classData.end(), ltMarks);
    ofs << "\nSorted by marks:\n";
    showClass(ofs, classData);
}
```

Figure 48: `grader.cpp`: implementation file for main program

change to one of the header files can trigger hours of recompilation time.

An important component of large-scale C++ design is to reduce the dependencies between source files. We will discuss ways to do this as the course progresses.

4.6 Strings and characters

Some more information on working with strings and characters.

Strings. Class `string` provides a large number of functions in addition to those that we have already seen. See http://www.cppreference.com/cppstring/all.html for a complete reference.

The following operators work for strings: 127

```
<    <=    ==    !=    >=    >    +    <<    >>    =    []
```

Operator `+` concatenates strings. There are several overloads, allowing for all combinations of `char`, C strings, and strings. Operator `[]` provides indexing for strings.

Strings function as containers for characters, working in a similar way to the type `vector<char>`. Consequently, iterators, `push_back`, and similar functions work for strings. In particular, `string` provides `insert` to insert characters into a string, `erase` to remove characters from a string, and `replace` to replace a sequence of characters in a string.

There are several functions for finding characters or substrings in strings. These functions usually return a position in the string in the form of an iterator. For example:

```
find
find_first_of
find_first_not_of
find_last_of
find_last_not_of
```

Characters. The library `cctype` provides the same functionality as the C header file `ctype.h`. cctype library
Although many of these functions should now be considered obsolete (e.g., `strcpy` and friends), others are still useful. In particular: 128

`isalpha(c)` returns `true` if `c` is a letter

`isdigit(c)` returns `true` if `c` is a digit

`isspace(c)` returns `true` if `c` is a blank, tab, or linebreak

`tolower(c)` returns the lower case equivalent of an upper case character and leaves other characters unchanged

`toupper(c)` returns the upper case equivalent of an lower case character and leaves other characters unchanged

References

Google (2017). Google C++ Style Guide. http://google.github.io/styleguide/cppguide.html. Last accessed 24.10.2017.

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.

5 Pointers, Iterators, and Memory Management

5.1 Pointers 83
5.2 Dynamic Memory Allocation 87
5.3 Iterators 90

Pointers were used in C as a machine-oriented feature for systems programming. C++ provides pointers because it is compatible with C. But C++ also provides a more abstract concept: the iterator. In this lecture, we discuss pointers and iterators, the relationship between them, and the importance of (manual) memory management in C++ programs.

5.1 Pointers

An important difference between C++ and Java is that, in C++, dynamic memory allocation and deallocation is ***explicit*** (done by code written by programmers) whereas in Java it is ***implicit*** (done by built-in system code). It is not quite true to say "Java does not have pointers", but it is true to say "Java does not allow direct access to pointers". (Weiss 2004, Chapter 3) includes a detailed comparison of the Java vs. C++ types and memory models.

A pointer is a variable that contains the memory address of another variable.

Like all variables in C++, a pointer has a type. The type depends on the object addressed by the pointer. For example, a pointer that points to an `int` has type "pointer to `int`", written `int*`.

For every type T, there is a type $T*$ or pointer to T.

To assign a value to a pointer, we need a way of obtaining addresses. There are two common ways, and we define the unimportant one first. The operator `&`, applied to a variable, yields the address of that variable. The address can be stored as a pointer. The following code defines two pointers, `pk1` and `pk2`, both pointing to the integer `k`, as shown in Figure 49 on the next page. address-of operator & 130 131

```
int k = 42;
int *pk1 = &k;
int *pk2 = &k;
```

If we have a pointer to an object, we can obtain the object itself by applying the ***prefix*** operator `*`. In the example above, `*pk1` is an integer variable (it is actually, of course, the integer `k`). operator *

The declaration `int *pk1` is a kind of pun. We can read it as `(int *)pk1` ("`pk1` has type `int *`") or as `int (*pk1)` ("`*pk1` has type `int`"). The forms with parentheses are illegal, but the C++ compiler is not fussy about spaces: we can write any of 132

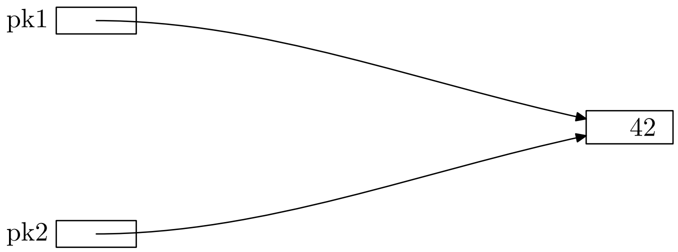


Figure 49: Two pointers, one value

```
int*pk0 = &k;
int *pk1 = &k;
int* pk2 = &k;
int * pk3 = &k;
```

Some programmers prefer the second form and some the third; few use the first or the fourth, although they are legal. There is one syntactic trap that you should be aware of. The statement

```
int* p, q;
```

declares p to be a pointer to int, but q is just a plain int.

Using pointers can have strange results – although they are not all that strange if you think
133 about them carefully. For example, the code

```
int k = 42;
int *pk1 = &k;
int *pk2 = &k;
++(*pk1);
cout << k << ' ' << *pk2 << endl;
```

displays 43 43. Since both pointers point to the same integer, any change to its value will be seen by ***all*** pointers.

5.1.1 Pointer Arithmetic

pointer arithmetic

C++ provides arithmetic operations (+ and −) on pointers. However, the arithmetic is rather
134 special. If p1 and p2 are pointers and i is an integer, then:

++p1	is a pointer
--p1	is a pointer
p1 + i	is a pointer
p1 - i	is a pointer
p1 - p2	is a signed integer (p1 and p2 must have the same type)

These operations would be utterly meaningless if it were not for the fact that C++ uses the size of the object pointed to when doing arithmetic. For example, suppose that an instance of type T occupies 20 bytes and that we write: 135

```
T t;
T *pt = &t;
```

Then pt+1 is an address 20 bytes greater than pt.

This kind of arithmetic is exactly what we need for arrays. In fact, subscript operations in C++ are ***defined*** in terms of pointer arithmetic. The declaration 136

```
double a[6];
```

introduces a as an array of 6 doubles. C++ treats a as a const pointer. We have: arrays and pointers

$$\begin{aligned}
\texttt{a[0]} &\equiv *\mathtt{a} \\
\texttt{a[1]} &\equiv *(\mathtt{a}+1) \\
\texttt{a[2]} &\equiv *(\mathtt{a}+2) \\
&\dots \\
\texttt{a[}i\texttt{]} &\equiv *(\mathtt{a}+i) \qquad \text{for any integer } i
\end{aligned}$$

5.1.2 Pointer types vs. reference types

The use of & as the "address-of" operator is different from its use as the "reference-to" operator, as you have seen it in Chapter 3.2 for pass by reference parameter definitions (int & k). However, the uses are closely related, in the sense that a reference is an address. Compare this code, working with pointers: 137

```
int i;
int *pi = &i;
++*pi;
cout << i << ’ ’ << *pi << endl;
```

with this code, working with references:

```
int i;
int &ri = i;
++ri;
cout << i << ’ ’ << ri << endl;
```

In C programming, there are no reference data types, so pointer types were used to achieve the same effect as pass by reference. C++ introduced reference types for this purpose and you should not use pointers anymore for modifying a function's arguments.

5.1.3 A note on null pointers

Every pointer type has a special value called ***null***. The definition of the null pointer is that it does not point to anything. The representation is the value zero; this works because, on almost all machines, the memory address zero is reserved for the operating system and cannot be used by programs to store data.

In old-style C programming, the null pointer was defined as a constant `NULL`, typically by a
138 preprocessor directive such as

```
#define NULL 0
```

This definition is compatible with the spirit of C's rather loose approach to types, but is not consistent with C++'s safer typing. Improvements such as

```
#define NULL (int) 0
#define NULL (void*) 0
```

do not help, because the first version does not work for pointers and the second version requires an ugly-looking cast whenever we use `NULL` for a pointer type other than `void*`. For example, to obtain the null pointer for integers, we would have to write

```
static_cast<int*>(NULL)
```

In C++, `#define`s are deprecated, and we are supposed to use constant declarations instead. Unfortunately, any reasonable declaration of `NULL` has the same problems as the attempts to `#define NULL`:

```
const int NULL = 0;
const void * NULL = 0;
```

Even if we could find a satisfactory definition for `NULL`, there would still be problems. Suppose
139 that we declare these functions:

```
void f(int n);     // first overload
void f(char *p);   // second overload
```

and call `f(NULL)` thinking that the compiler would say to itself: "The programmer has used `NULL`, which suggests a pointer, and therefore I will pass `static_cast<char*>(NULL)` to the second overload".

However, the compiler does not reason like this. Instead, it says: "`NULL` is 0 and 0 is an `int`, and therefore I will pass 0 to the first overload".

As Meyers (Meyers 1998, Item 25) explains, this is an unusual case because people tend to think that there is an ambiguity but the compiler does not. (Usually, people think their meaning is perfectly obvious and are annoyed when the compiler calls it ambiguous.) Meyers recommends:

Prefer not to overload a function with integer types and pointer types.

The simplest solution for writing null pointers is just:

Forget about* `NULL` *– just use* `0`*.

5.1.4 Null pointers in C++11

C++11 includes a new 'null pointer constant', **`nullptr`**. This makes the confusing double-use of 'O' as both the integer zero and the `NULL` pointer obsolete. The **`nullptr`** is of type **`nullptr_t`** and can be used like this: nullptr 140

```
char *p = nullptr;
```

Use* `nullptr` *instead of* `0` *or* `NULL` *for* C++11 *and above.

5.2 Dynamic Memory Allocation

So far, we did not worry about managing memory for variables: even for dynamically growing data structures like vectors, we relied on automatic memory management (in case of vectores, this is handled by the STL). However, for most C++ programs you will need to take care of memory management yourself, as C++ does not offer built-in ***garbage collection*** like Java. Hence, it is important to understand when and how to manually allocate and de-allocate memory.

5.2.1 Stack Allocation

Most data in C++, and all the data we have seen so far in this course, is allocated on the ***run-time stack***.

The run-time stack, "stack" for short, is initially empty. When execution starts, the runtime parameter (program arguments) are pushed onto the stack. On entry to a function, the local data associated with the function (including its arguments) are pushed onto the stack. When the function returns, the local data is popped off the stack. Thus the stack varies in size as the program runs.

When a function has returned, its local data no longer exists.[20] Normally, this is not a problem, because the function's local variables are no longer accessible. Playing tricks with pointers, however, can cause problems. Consider the code shown in Figure 50 on the following page. 141

This program compiles because there are no syntactic or semantic errors. The final assignment, `p = f()`, makes `p` a pointer to `k`. But `k` no longer exists, having been popped off the stack when `f` returns. Any attempt to use `p` will have unpredictable results.

```
int * f()
{
    int k = 42;
    int *pk = &k;
    return pk;
}

int main()
{
    int *p;
    p = f();
    return 0;
}
```

Figure 50: Dangerous tricks

```
char *read()
{
    char buffer[20];
    cin >> buffer;
    return buffer;
}

int main()
{
    char *pc = read();
    cout << pc << endl;
    return 0;
}
```

Figure 51: Subtler dangerous tricks

142

Figure 51 shows a more subtle version of the same problem. The function `read` has result type `char*` but actually returns an array of characters: this works because the compiler treats these types as the same. Similarly, the main function treats function `read` as having type `char*`.

The reason that this is a bad program is that the array `buffer` is allocated on the stack. It is destroyed when the function returns. The pointer `pc` is undefined and cannot be used safely.

Some compilers, including GCC, actually recognize the mistake and issue a warning message:

```
In function char* read():
warning: address of local variable buffer returned
```

[20] Actually, it's still there, on the stack, but will be overwritten when the next function is called. Consequently, we must ***assume*** that it no longer exists.

This provides another reason for paying attention to warnings from the compiler!

Problems like this tend to occur when we use "old-style" C++ code. No problems arise if we use the STL class `string` instead of an array of characters.

> ***Use `string`, in preference to `char *`, wherever possible.***

5.2.2 Heap Allocation

Stack allocation works because function calls and returns match the "last-in-first-out" (LIFO) discipline of a stack. Sometimes this is not good enough. For example, we might want to allocate data within a function, use that data for a while after the function has returned, and then deallocate the data. We can do this by allocating the data on the ***heap***, an area of memory that does not obey any particular discipline such as LIFO or FIFO. The heap is used like this: 143

```
T *p = new T();
....
delete p;
```

operator `new` for heap allocation

The operator `new` requires a call to a constructor on its right. The value returned by `new` is a pointer to the constructed object. We can use this pointer to perform operations on the object. When we have finished with the object, we apply `delete` to the pointer to destroy the object and deallocate the heap memory it was using.

Suppose class `T` provides a public function `f`. To call `f` using the pointer `p`, we must first use '`*`' to dereference `p` and then use '`.`' to call `f`. Thus we write `(*p).f()`. (The parentheses around `*p` are necessary, because the compiler reads `*p.f()` as `*(p.f())`, which is wrong.) Since this construction occurs often, there is an abbreviation for it:

-> operator

$$\texttt{p->f()} \equiv \texttt{(*p).f()}$$

144

145 Figure 52 on the following page provides a simple example of heap allocation and deallocation. The function `maketest` constructs a new instance of class `Test` on the heap, calls its function `f`, and returns a pointer to it. The function `killTest` deletes the object. The program displays the following output: 146

```
makeTest
Constructor
Function
killTest
Destructor
```

In C++, it is important to always be aware of the memory location where a particular object is stored at run-time (stack or heap) – in Java, this question does not arise, because objects are only created on the heap, never on the stack.

```
class Test
{
public:
    Test() { cout << "Constructor\n"; }
    ~Test() { cout << "Destructor\n"; }
    void f() { cout << "Function\n"; }
};

Test *makeTest()
{
    cout << "makeTest\n";
    Test *pt = new Test();
    pt->f();
    return pt;
}

void killTest(Test *p)
{
    cout << "killTest\n";
    delete p;
}

int main()
{
    Test *p = makeTest();
    killTest(p);
    return 0;
}
```

Figure 52: Heap allocation and deallocation

5.3 Iterators

iterators, containers, and algorithms

Iterators are one of the keys to the flexibility of the STL. We have seen that the STL provides containers and algorithms. Iterators provide the glue that allows us to attach one to the other:

- Each container specifies the iterators that it provides
- Each algorithm specifies the iterators that it needs

For example, `vector<T>` provides the kind of iterators that `sort` requires; it follows that we can sort vectors.

147 A pair of iterators specifies a range of container elements. The range defines a semi-closed interval: the `first` iterator of a range accesses the first element of the range, and the `last` iterator accesses the first element ***not*** in the range. In this typical loop:

```
for (⟨iterator⟩ it = first; it != last; ++it)
    .... *it ....
```

⟨*iterator*⟩ stands for some iterator type, and we see that the iterator must provide the operations:

- `first != last` (and therefore `first == last`): equality comparison
- `++it`: increment, or step to next element
- `*it`: dereference to provide access to the container element

C++ programmers will recognize that all of these operations are provided by pointers. In fact, we can use raw pointers as iterators: pointers as iterators 148

```
const int MAX = 20;
double values[MAX] = .... ;
sort(&values[0], &values[MAX]);
```

5.3.1 Kinds of iterator 149

There are several ways of classifying iterators. Below, we define individual properties; most iterators possess several of these properties. In each of the following cases, we use `it` to stand for an iterator with the given property. Figure 53 summarizes the properties that various kinds 150 of iterators provide.

- All iterators implement the operator `=` (assignment).
- With the exception of the output iterator, all iterators implement the operators `==` and `!=` (comparison).
- An ***input iterator*** can be used to read elements from a container but does not provide write access. That is, `*it` is an Rvalue.
- An ***output iterator*** can be used to update elements in a container but may not provide read access. That is, `*it` is a Lvalue. Note that output iterators ***cannot*** be compared with `==` or `!=`.

Property	Input	Output	Forward	Bidirectional	Random Access	Insert
Assign (`=`)	√	√	√	√	√	√
Compare (`==`, `!=`)	√		√	√	√	√
Read (`*it`)	√		√	√	√	
Write (`*it`)		√	√	√	√	√
Order (`<`, `<=`, `>`, `>=`)					√	
Increment (`++`)			√	√	√	
Decrement (`--`)				√	√	
Arithmetic (±i, `+=`, `-=`, `-`)					√	

Figure 53: Properties of iterators

An iterator may be both an input and an output iterator. That is, it may allow both read and write access to elements of the container. In fact, this is probably the most common kind of iterator.

- A ***forward iterator*** is an input and output iterator that can traverse the container in one direction. A forward iterator must implement `++`.
- A ***bidirectional iterator*** is an input and output iterator that can traverse the container forwards and backwards. A bidirectional iterator must implement `++` and `--`.

  There are no "backwards only" iterators; an iterator is either forward or bidirectional.
- A ***random access iterator*** must allow "jumps" in access as well as traversal. The principal operation that a random access iterator must provide is indexing: `it[n]`. Random access iterators also provide:
  - Addition and subtraction of integers: `it + n` and `it - n`
  - Assignment operators: `it += n` and `it -= n`
  - Subtraction: `it1 - it2` yields a signed integer
  - Comparisons: the operators `<`, `<=`, `>=`, and `>`
- An ***insert iterator*** is an output iterator that puts new values into a container rather than just updating the values that are already there.

All of the STL containers provide bidirectional iterators. It follows that they all provide input, output, and forward iterators. These categories are useful because we can construct special iterators that may not have all of the properties.

The only container classes that provide random access iterators are `deque`, `string`, and `vector`. This is because these containers are required to store elements in consecutive locations, which 151 means that random access is a simple address calculation.[21] Figure 54 shows the iterators associated with a ***selection*** of containers and algorithms. The table is far from complete and, for details, you should consult an STL reference, such as (Josuttis 2012).

Container	Iterator provided	Algorithm	Iterator required
`vector`	random access	`find`	input
`list`	bidirectional	`search`	forward
`slist`	forward	`count`	input
`deque`	random access	`copy`	input/output
`set`	constant bidirectional	`reverse`	bidirectional
`multiset`	bidirectional	`sort`	random access
`map`	bidirectional	`stable_sort`	random access
`multimap`	bidirectional	`binary_search`	forward

Figure 54: Iterators, Containers, and Algorithms

[21] The address of `c[i]` is $\&c + s \times i$, where $\&c$ is the address of the container and s is the size of an element.

We have often mentioned that a range is specified by an iterator accessing the first element of the range and another iterator accessing the element following the last element of the range – which, in most cases, does not even exist. Here are some reasons for this choice:

1. Two equal iterators specify an empty range.
2. We can use `==` and `!=` to test for an empty range and for end of range – we do not need `<` and friends.
3. We have an easy way to indicate "out of range", namely, the `last` iterator of the range.

   A function can return an iterator for a valid element to indicate success or an iterator for an invalid element to indicate failure. This avoids the need for special values, flags, etc.

An iterator with type `iterator` is allowed to change the contents of its associated container. An iterator with type `const_iterator` (called a ***constant iterator***) is ***not*** allowed to change the contents of its associated container. It is best to use constant iterators whenever possible. const_iterator

5.3.2 Using iterators

Suppose that we want to divide the students of the grading program into two groups, according as to whether they passed or failed the course. The first thing we will need is a criterion for deciding whether a student has passed. We add the following member function to class `Student`: 152

```
bool passed() { return total > 50; }
```

Note:

- The definition of this function appears within the class declaration. This is allowed, and a consequence is that the compiler may ***inline*** calls to the function. We will discuss the implications of this later.
- Beginners might write `passed` in this way: 153

```
bool passed()
{
   if (total > 50)
      return true;
   else
      return false;
}
```

  This is ***stupid***! The statements `return true` and `return false` are occasionally required but, in many cases, a predicate should return a boolean expression.

Let us first consider a straightforward approach, in which we create two empty vectors for passed and failed students, and iterate through the class assigning each student to one or the other vector: see Figure 55 on the next page. In this code, remember that `it->passed()` is an abbreviation for `(*it).passed()`. 154

Unfortunately, this code does not compile: see Figure 56 on the following page. The reason is rather subtle: 155

```
vector<Student> passes;
vector<Student> failures;
for (  vector<Student>::const_iterator it = data.begin();
       it != data.end();
       ++it )
    if (it->passed())
        passes.push_back(*it);
    else
        failures.push_back(*it);
```

Figure 55: Separating passes and failures: first version

```
f:\....\grader.cpp(111):
   error C2662: 'Student::passed' :
   cannot convert 'this' pointer from
       'const std::allocator<_Ty>::value_type'
   to  'Student &'
   with
        [
            _Ty=Student
        ]
```

Figure 56: Compiling Figure 55 fails

- By using `const_iterator`, we are asserting that the iterator cannot be used to change the value of a `Student`.
- The loop contains the call `it->passed()`. ***We*** know that this call will not change the value of the `Student` object `*it`, but the ***compiler*** doesn't.
- 156 Consequently, the compiler rejects the program.

There is a "quick and dirty" fix for this error: we could just use an `iterator` instead of a `const_iterator`. This is a ***bad*** solution because it does not address the real cause of the problem. The correct solution is to convince the compiler that `passed` does not change the value of the object it acts on. We can do this by adding `const` to the function declaration:

const member functions

157

```
bool passed() const { return total > 50; }
```

After executing the code in Figure 55, we now have ***three*** vectors and two copies of each student record, one in the original vector and the other in either `passes` or `failures`. It would be more efficient in terms of space to move the failed students into a new vector and remove them from the original vector. Vectors provide the function `erase` to remove elements from a container. The code in Figure 57 on the facing page does this.

The first point to note is that we have to use a `while` loop rather than a `for` loop, because the loop step is not necessarily `++it`. However, for students that pass the course, `++it` is all we have to do.

```
vector<Student> failures;
vector<Student>::iterator it = data.begin();
while (it != data.end())
{
    if (it->passed())
        ++it;
    else
    {
        failures.push_back(*it);
        it = data.erase(it);
    }
}
```

Figure 57: Separating passes and failures: second version

When a student fails, the corresponding record is stored in failures. In order to remove the record from the class data vector, we write

```
it = data.erase(it);
```

The effect of data.erase(it) is to remove the element indicated by it from the vector data. After this has been done, the iterator is ***invalid*** because it accesses an element that no longer exists. The function erase returns an iterator that accesses the next element of the vector.

iterator invalidation

It is important to use the iterator returned by erase and not to assume that it is just ++it. An iterator operation that invalidates an iterator may do other things as well, even including moving the underlying data. When an iterator operation invalidates an iterator, it potentially invalidates ***all*** iterators.

For example, calling erase in Figure 57 invalidates the iterator for the end of the vector, because removing one component forces the remaining components to move. It would therefore be a serious mistake to attempt to "optimize" Figure 57 like this: 158

```
vector<Student> failures;
vector<Student>::iterator it = data.begin();
vector<Student>::iterator last = data.end(); // Save final iterator
while (it != last)
    ....
```

Check whether an operation invalidates iterators before using it.

The solution we have developed works correctly, but is inefficient. The inefficiency is negligible for a class of sixty students but could be a problem if we used the same strategy for very large vectors. To understand the reason for the inefficiency, suppose that an entire class of 50 students fails. The program would execute as follows:

Remove first record, move remaining 49 records
Remove second record, move remaining 48 records
Remove third record, move remaining 47 records
....

The total number of operations is $\underbrace{49 + 48 + 47 + \cdots + 1}_{49 \text{ terms}}$ and is clearly $\mathcal{O}(N^2)$ for N students.

To improve the performance, we must change the data structure. A list can erase in constant time (i.e., $\mathcal{O}(1)$) and can perform the other operations that we require. It is a straightforward exercise to replace each `vector` declaration by a corresponding `list` declaration.

It is obviously important to know which operations invalidate iterators. Fortunately, good STL reference documents usually provide this information. If you are not sure, you can make a good guess by thinking about how the operation must work on a given data structure – but it's much safer to look up the correct answer.

5.3.3 Range Functions

Most containers have ***range functions*** – that is, functions with a pair of parameters representing a range of elements in the container. If a suitable range function exists, it is better to use it than to use a loop.

Suppose that you want to create a vector `v1` consisting of the back half of the vector `v2` (Meyers
159 2001, Item 5). You could do it with a loop:

```
v1.clear();
for (   vector<Widget>::const_iterator ci = v2.begin() + v2.size()/2;
        ci != v2.end();
        ++ci )
   v1.push_back(*ci);
```

but it is quicker to write any one of the following statements:

```
v1.assign(v2.begin() + v2.size()/2, v2.end());

v1.clear();
copy(v2.begin() + v2.size()/2, v2.end(), back_inserter(v1));

v1.insert(v1.end(), v2.begin() + v2.size()/2, v2.end());
```

In this case, `insert` is probably the best choice.

References

Josuttis, N. M. (2012). *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley. http://www.cppstdlib.com/.

Meyers, S. (1998). *Effective C++: 50 Specific Ways to Improve Your Programs and Designs* (2nd ed.). Addison-Wesley.

Meyers, S. (2001). *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.

6 Designing classes

6.1 Constructors 102
6.2 Destructors 107
6.3 Operators 108
6.4 Conversions 117
6.5 Accessors 119
6.6 Mutators 119
6.7 Odds and ends 121
6.8 Summary 126

Class design is a big topic in C++, with many aspects to consider. Since any one class does not illustrate all of the aspects of class design, we use two example classes to illustrate the issues. The first example class is `Rational`, which implements fractions and provides examples of the issues that arise in developing a numerical or algebraic class. The second example class is `Account`, which – with a bit of artificial manipulation – provides examples of memory management and other problems.

161

Rational. Figure 58 on the next page shows the declaration of class `Rational`. Comments 162
that would normally be included in such a declaration have been omitted to save space. The 163
accompanying text provides adequate explanation. Note that this `Rational` class is similar to, 164
but not identical with, the rational example in (Weiss 2004, Section 5.5).

An instance of `Rational` is a rational number, or "fraction", represented by two long integers. The fraction $\frac{n}{d}$ is represented by the pair (n, d). The class provides assignment, copying, arithmetic operations (`+`, `-`, `*`, `/`), and exponentiation (`^`), the associated assignment operators (`+=`, `-=`, `*=`, `/=`), and comparisons (`==`, `!=`, `<`, `>`, `<=`, `>=`) for fractions. It also provides stream operators for reading and writing fractions.

Rational numbers are stored in normalized form: the pair (n, d) is in ***normal form*** if $d > 0$ and $\gcd(n, d) = 1$. Thus we can consider $d > 0 \wedge \gcd(n, d) = 1$ to be a ***class invariant***. For example, the construction `Rational(4,-6)` yields the pair $(-2, 3)$. An attempt to create a fraction with zero denominator raises an exception.

Account. The other example class, `Account`, is rather different. Some of the differences are due to the application and some are due to an intentional complication: `Account` has a data member that is a pointer.

An account is associated with a person who has a name. In class `Account`, the name is represented as a `char*` (pointer to array of characters). This introduces various problems, ***all of which could be avoided*** just by using the standard class `string` instead of `char*`. We use `char*`, however, just to show what the problems are when designing classes with pointer members, and how they can be solved.

Other differences between `Rational` and `Account` concern comparison, accessors, and mutators. 165

166

```
#ifndef RATIONAL_H
#define RATIONAL_H

#include <iostream>

class Rational
{
public:
    Rational(long n = 0, long d = 1);
    const Rational & operator+= (const Rational & right);
    const Rational & operator-= (const Rational & right);
    const Rational & operator*= (const Rational & right);
    const Rational & operator/= (const Rational & right);
    const Rational operator-() const;
    const Rational operatorˆ (int e) const;
    friend const bool operator== (const Rational & left,
                                  const Rational & right);
    friend const bool operator!= (const Rational & left,
                                  const Rational & right);
    friend const bool operator<  (const Rational & left,
                                  const Rational & right);
    friend const bool operator>  (const Rational & left,
                                  const Rational & right);
    friend const bool operator<= (const Rational & left,
                                  const Rational & right);
    friend const bool operator>= (const Rational & left,
                                  const Rational & right);
    friend std::istream & operator>> (std::istream & is,
                                      Rational & r);
    friend std::ostream & operator<< (std::ostream & os,
                                      const Rational & r);
    double toDouble() const;
    enum Exceptions { BAD_INPUT, ZERO_DENOMINATOR };

private:
    void normalize();
    long num;
    long den;
};

const Rational operator+ (const Rational & left, const Rational & right);
const Rational operator- (const Rational & left, const Rational & right);
const Rational operator* (const Rational & left, const Rational & right);
const Rational operator/ (const Rational & left, const Rational & right);

#endif
```

Figure 58: Declaration for class `Rational`

```
#ifndef ACCOUNT_H
#define ACCOUNT_H

#include <string>

class Account
{
public:
    Account();
    Account(char *name, long id, long balance = 0);
    Account(const Account & other);
    ~Account();
    Account & operator=(const Account & other);
    long getBalance() const;
    void deposit(long amount);
    void withdraw(long amount);
    void transfer(Account & other, long amount);
    friend bool operator==(const Account & left,
                           const Account & right);
    friend bool operator!=(const Account & left,
                           const Account & right);
private:
    void storeName(char *s);
    char *name;
    long id;
    long balance;
};

#endif
```

Figure 59: Declaration for class `Account`

Figure 59 shows the declaration of class `Account`. The data members of an account are: the `name` of the owner or client; an `id` number; and the `balance` in the account.

C++ does not have a type (like `Currency`) that is really suitable for financial work.[22] Curiously, the Boost library developers have never accepted a currency class, although examples exist[23] that could be raised to the standard of a Boost class fairly easily.

Accountants do not like `double` because rounding prevents precise balancing of accounts. Integers used to represent cents are quite good, provided that input and output functions make appropriate conversions between dollars and cents. 32-bit integers permit values between $\pm\$21,474,836.47$, which is not enough for a bank president's annual income. 64-bit integers are necessary for realistic financial calculations, as the following table shows (but remember that the concrete maximum values are platform-dependent).

never use a float variable for financial applications!

[22] Surprisingly few languages do have such a type. `Java` provides `BigDecimal`.

[23] See, for example, http://www.colosseumbuilders.com/sourcecode.htm.

Type	Maximum signed value ($)
`int`	327.67
`long`	21,474,836.47
`long long`	92,233,720,368,547,758.07

`long long`

New `long long` type in C++11
The integer type `long long` has been standard in C since 1999, to support 64-bit integers. It has finally been approved for C++11 (together with `unsigned long long`), but some compilers allowed it for a few years now, including GCC, even though it was not officially part of the language. If you are interested in the technical details of adding a type to a standardized language, see http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2005/n1811.pdf.

We avoid the issue of a currency class in our example by using `long`.

There are other ways, too, in which the account class given here is unrealistic and nothing like a class that would be used in a banking application, for example. Nevertheless, the example provides some useful insights into class design.

6.1 Constructors

All classes have constructors. If you do not provide a constructor, the compiler generates a default constructor that allocates memory for the object but doesn't do anything else. Except in rare cases, for very simple classes, it is best to provide one or more constructors. If you provide any constructor at all, default or otherwise, the compiler does ***not*** generate a default constructor.

A ***default constructor*** is a constructor without any parameters. There are a number of situations in which a default constructor is required; for example, an array declaration is allowed only for types that have a default constructor.

Define a default constructor for every class.

calling constructors 167

Warning. There is a small but important inconsistency in C++ notation. Suppose that `C` is a class with two constructors, one of which is a default constructor. Now consider these statements:

```
C c1(45);
C c2();
```

It is natural to read the statements like this:

- `c1` is an instance of `C` constructed from an integer;
- `c2` is an instance of `C` constructed using the default constructor.

Unfortunately, this interpretation is wrong! In fact, c2 has been declared (but not defined) as a function that takes no arguments and returns a C. To invoke the default constructor, you must ***leave out the parentheses***:

```
C c1(45);
C c2;
```

Rational. Class Rational requires only one constructor. It is a general purpose constructor with two long parameters, corresponding to the numerator and denominator of the fraction, stored as num and den, respectively. Both parameters have default values (n=0, d=1), which implies that this constructor is considered to be a default constructor.

The following declarations are equivalent: in each case, a Rational object is constructed with r.num $= 0$ and r.den $= 1$: 168

```
Rational r;
Rational r(0);
Rational r(0, 1);
Rational r = Rational(0);
Rational r = Rational(0, 1);
```

The constructor calls normalize to put the fraction in normal form. Note that the default parameter values are ***not*** repeated in the function definition. 169

```
Rational::Rational(long n, long d)
    : num(n), den(d)
{
    normalize();
}
```

The data members are set using initializers rather than assignments. This is the best way to initialize data members and you should use it whenever possible.

constructor initializers

Use initializers in constructors.

The private member function normalize (see Figure 60 on the next page) is called whenever a new fraction is created. If den $= 0$, it throws the exception ZERO_DENOMINATOR, which is one of the values of the enumeration Exceptions declared in the class. If num $= 0$, it sets den $= 1$, to ensure that the fraction zero has the unique representation $(0, 1)$. If den < 0, it changes the sign of both numerator and denominator. Finally, it divides both by their greatest common divisor to cancel common factors.

170

171

throw

Providing default values for the parameters of the constructor has an important consequence. If N is an expression of type long, the Rational(N) is the rational number $(N, 1)$. The C++ compiler interprets this as permission to silently convert long to Rational whenever such a

default values in constructors and automatic type conversions

```
long gcd(long i, long j)
{
    assert(i > 0 && j > 0 && "Error in gcd arguments");
    while (true)
    {
        long tmp = j % i;
        if (tmp == 0)
            return i;
        j = i;
        i = tmp;
    }
    assert(false && "Logical error in gcd");
}

void Rational::normalize()
{
    if (den == 0)
        throw Rational::ZERO_DENOMINATOR;
    if (num == 0)
    {
        den = 1;
        return;
    }
    if (den < 0)
    {
        num = -num;
        den = -den;
    }
    assert(num != 0 && den > 0 && "Logical error in normalize");
    long g = gcd(abs(num), den);
    num /= g;
    den /= g;
}
```

Figure 60: Functions to normalize instances of class `Rational`

conversion is required. Since other integral types, such as `char` and `int`, may also be implicitly converted to `long`, these types may be implicitly converted to `Rational`.

172 This is convenient, because it allows us to use "mixed-mode" arithmetic. For example:

```
Rational r(1, 2);
r += 2;
cout << r << endl;       // Writes "5/2"
cout << 1/r << endl;     // Writes "2/5"
```

Automatic conversion also leads to less expected results:

```
Rational r;
r = 'a';
cout << r << endl; // Writes "97"
```

Account. The normal constructor for an `Account` is passed the name, identification code, and opening balance for the new account. If the opening balance is omitted, the balance is set to zero.

The name is passed as a `char*`. Simply copying the pointer would be a serious mistake. As one
of the many ways in which things could go wrong, consider Figure 61. Since the bad guys use 173
buffer overflow as the basis of many attacks, it is always a mistake to read characters into a character array. To make it hard for them, we can use a large buffer (which is not a properly secured solution either). The real problem with this function, however, is that the buffer is destroyed when the function returns.

```
Account * createAccount()
{
    char buffer[10000];
    cout << "Please enter your name: ";
    cin >> buffer;
    int id;
    cout << "Please enter your account ID: ";
    cin >> id;
    return new Account(buffer, id);
}
```

Figure 61: Pointer problems

It would be unreasonable to require the user of the `Account` class to take care of this problem. We would have to provide an arcane restriction along the lines of "the name of the client passed to the constructors must have a longer lifetime than the `Account` object". Instead, we write a constructor that makes its copy of the name, allocates memory for it, and stores it. It turns out that storing the name is something that we will have to do several times. Consequently, it makes sense to put it into a function, called `storeName` here. Here is the constructor: 174

```
Account::Account(char *name, long id, long balance)
    : id(id), balance(balance)
{
    storeName(name);
}
```

Space for the name is allocated dynamically (i.e., using `new`, with space taken from the heap) and we must remember to allocate one character position for the terminator, `'\0'`. (We are using the old-style functions `strlen` and `strcpy`. As mentioned above, these problems do not arise with the more modern class `string`.)

```
void Account::storeName(char *s)
```

```
        {
            name = new char[strlen(s) + 1];
            strcpy(name, s);
        }
```

The constructor above is ***not*** a default constructor because the parameters do not have default values. We could provide default values or we could define another constructor for use as the default, like this: 175

```
        Account::Account()
            : id(0), balance(0)
        {
            storeName("");
        }
```

copy constructor

The ***copy constructor*** is an important component of every C++ class. It is used whenever an object has to be copied. Common uses of the copy constructor include:

- Initialized declarations of the form

```
  Account anne = bill;
  Account anne(bill);
```

- Passing an object by value.
- Returning an object by value.

If we do not define a copy constructor, the compiler generates one for us. This ***default copy constructor*** simply copies the values of the data members of the object. The default copy constructor for `Rational` does exactly what we want and there was no need to define our own version. Class `Account` is different.

Class `Account` has the data member `name`, which is a pointer. The default copy constructor just copies the pointer, not the object it points to. This would be a disaster for accounts, because we would end up with more than one pointer pointing to the same name. If one account is deleted, the others would be left with ***dangling pointers***. (Technically, the copy constructor performs a ***shallow copy*** but what we need is a ***deep copy***.)

The copy constructor must therefore behave like the constructor, making a new copy of the name. Fortunately, we have a function `storeName` that does exactly the right thing.

The signature (or prototype) of the copy constructor for a general class `T` is `T::T(const T &)`. For our class, `Account`, the implementation looks like this:

```
        Account::Account(const Account & other)
              : id(other.id), balance(other.balance)
        {
            storeName(other.name);
        }
```

Define a copy constructor for any class that has pointer members.

There is an alternative way of defining a copy constructor that is sometimes useful:

- Declare the copy constructor `T(const T &)` in the `private` part of the class declaration.
- Do ***not*** provide a definition of the copy constructor.

preventing copying an object

The effect is to prevent copying of the object. By declaring the copy constructor, we prevent the compiler from generating it. By making the copy constructor private, we prevent outsiders from calling it. By not defining the copy constructor, we prevent member functions from using it. The result is that any initialized declaration, passing by value, or returning by value, will be flagged as an error by the compiler.

It is quite possible that a real-life banking application might choose to prevent copying of accounts. This is the mechanism that could be used.

Note that preventing copying does not make `Account` a Singleton; we can have as many accounts as we need, but we cannot make copies of them.

6.2 Destructors

A ***destructor*** is a member function that is called when an object is to be deallocated or "destroyed". The compiler provides a default destructor if you don't. The default destructor releases the memory held by the object and does nothing else. There are two circumstances in which you ***must*** define a destructor:

1. The class has members that are pointers.
2. The class will be used as a base class.

We will discuss the second aspect in detail in Lecture 8.

Rational. Class `Rational` does not have any pointer members and is not intended to be used as a base class. Consequently, we do not define a destructor for it.

Account. We must define a destructor for class `Account` because it has a pointer member. Later, we will discuss using `Account` as a base class, which provides another reason for having a destructor.

> ***Define a destructor for any class that has pointer members.***

delete x vs. delete [] x

The destructor must destroy any data that was created dynamically (using `new`) by the constructor. Destruction is performed by `delete`, which has two forms:

- `delete x` for simple objects
- `delete [] x` for arrays

The distinction between the two kinds of destruction is very important, because the compiler may not detect the error if you are wrong. In `Account`, we must use `delete [] name`; if we wrote `delete name` instead, only the first character of the name would be deallocated.

Use* `delete` *for simple variables and* `delete[]` *for arrays.

176 Here is the destructor for class `Account`:

```
Account::~Account()
{
    delete [] name;
}
```

6.3 Operators

C++ allows most operators to be overloaded. This is an extremely useful feature of the language, but it should not be abused. If operators are defined, they should be defined consistently and they should behave in a reasonable way.

Rational. Class `Rational` is an obvious candidate for overloaded operators, because fractions are numbers and users will expect to use arithmetic operations with fractions. To respect C++ conventions, if you provide `+`, you should also provide `+=`, and similarly for the other operators. It turns out to be easier, and more efficient, to define the assignment operators (`+=`, `-=`, `*=`, `/=`) as member functions and then to use them in the definitions of the simple operators (`+`, `-`, `*`, `/`). 177

178 Figure 62 on the next page shows the implementation of the assignment operators. Each one normalizes its result and returns a reference to the new fraction. Returning a reference avoids making an unnecessary copy of the object. This is another convention that you should follow; it allows users to write statements such as

```
p += q *= r;
```

assuming that they can figure out what such expressions mean.

But why did we declare the return type `const` for the operators, as in

```
const Rational & Rational::operator+= (const Rational & right)
const Rational operator+ (const Rational & left, const Rational & right);
```

If the return types weren't `const`, a user of our `Rational` class would be allowed to write code like

```
(r + s) = t;
```

which is clearly nonsense. To prevent mistakes as in `if(r+s = t)` (instead of `==`), we define the return types of all our operators as `const`. As Meyers recommends (Meyers 2005, Item 3):

Use `const` ***whenever possible.***

this

The keyword `this` can be used only in a member function (i.e., method of a class). It is a constant pointer to the current object. To obtain the whole object, we dereference `this` with the `*` operator. The functions in Figure 62 use this trick to return the current object as `*this`.

```
const Rational & Rational::operator+= (const Rational & right)
{
    num = num * right.den + den * right.num;
    den *= right.den;
    normalize();
    return *this;
}

const Rational & Rational::operator-= (const Rational & right)
{
    num = num * right.den - den * right.num;
    den *= right.den;
    normalize();
    return *this;
}

const Rational & Rational::operator*= (const Rational & right)
{
    num *= right.num;
    den *= right.den;
    normalize();
    return *this;
}

const Rational & Rational::operator/= (const Rational & right)
{
    num *= right.den;
    den *= right.num;
    normalize();
    return *this;
}
```

Figure 62: Arithmetic assignment operators for class `Rational`

Account. Since adding and subtracting accounts does not make much sense, we do not provide operators for class `Account`.

6.3.1 Assignment (`operator=`)

By default, the compiler will generate a default assignment operator (`operator=`). The effect of this operator will be to copy all of the fields of the object. If this is what you want, you do not need to define your own version of `operator=`.

Rational. The default assignment operator is just what we need for `Rational` objects: the statement `r = s` will copy the numerator and denominator of `s` to `r`. Consequently, we do not define `operator=`.

Account. Since `Account` has a pointer member, we ***must*** define an assignment operator for it. Not doing so will lead to the same problems as not having a copy constructor: we will end up with many accounts all pointing to the same name. The assignment operator looks rather like the copy constructor, but there are two important differences. Consider the ***incorrect*** version of the assignment operator shown in Figure 63. 179

```
Account & Account::operator=(const Account & other)
{
    storeName(other.name);
    id = other.id;
    balance = other.balance;
    return *this;
}
```

Figure 63: An ***incorrect*** implementation of `operator=`

Note first that the type of the assignment operator is `Account&` and that the function returns `*this`. This is conventional for assignment operators and it allows statements such as

```
a1 = a2 = a3 = a4 = a5;
```

for programmers who want to write such statements.

memory leaks

The definition of `Account::operator=` above has a memory leak. When `storeName` is called, the old name of the account is lost.

There is another problem. Suppose that the programmer writes

```
a = a;
```

self-assignment

and think about the effect in `Account::operator=` as defined above. `storeName` allocates space for the name, and copies the name into. The old name is lost – another memory leak! If we insert the statement `delete [] name` to avoid the memory leak, things get even worse: `storeName` would attempt to copy the deleted name! The solution is to check for ***self-assignment***. These considerations lead to the correct assignment operator shown in Figure 64 on the facing page. 180

You might ask: "What programmer could be so stupid as to write `a = a`?" The answer is that a programmer might not write this assignment as such, but it could easily be generated by template expansion. Also, a programmer might quite reasonably write

```
Account & Account::operator= (const Account & other)
{
    if (this == &other)
        return *this;
    delete [] name;
    storeName(other.name);
    id = other.id;
    balance = other.balance;
    return *this;
}
```

Figure 64: Assignment operator for class `Account`

```
a[i] = a[j];
```

and not feel it necessary to check `i != j`.

Define an assignment operator for any class that has pointer members. The assignment operator must check for self-assignment and return a reference to* `*this`*.

If you want to prevent assignment of instances of a class, declare `operator=` as a `private` member function and do not provide an implementation for it.

6.3.2 Arithmetic

If it makes sense to perform arithmetic operations on instances of the class, we can provide the appropriate operators. These are usually implemented in conjunction with the corresponding assignment operators, described above.

In general, you should implement the arithmetic functions so that they obey standard laws of algebra. For example, `-` is the inverse of `+`, `/` is the inverse of `*`, and so on.

There are occasional exceptions to this rule. For example, `string` uses `+` for concatenation, even though concatenation is neither commutative nor associative, and has no inverse. But people seem to accept `+` as a concatenation operator, so this is perhaps excusable.

Arithmetic operators, such as `operator+`, can be implemented as member functions with one parameter or as free functions with two parameters. The form of declaration for a member function is:

implementing operators: impacts of free vs. member functions

```
Rational operator+(const Rational & rhs);
```

The left operand is the current object (`*this`) and the right operand is the value passed to the parameter `rhs`.

If `operator+` is implemented as a member function, then 181

```
x + y
```

is effectively translated as

```
x.operator+(y)
```

The compiler will use the static type of `x` to choose the appropriate overload of `operator+` and may convert `y` to match the type of `x`. Suppose that we implemented `+` for class `Rational` in this way and that `i` is an `int` and `r` is a `Rational`. Then `r+i` converts `i` to `Rational` and adds the resulting fractions but `i+r` does not compile because the integer version of `operator+` cannot accept a `Rational` right argument. In other words, the advantage of defining comparison operators as free functions is that the order of operands does not matter.

There is another issue to consider when we choose to implement free functions associated with a class: do we provide access functions for data members of the class, or do we declare the free functions as `friends`? The choice depends very much on the particular application. For example, if the class already provides accessor functions for some reason, the free functions can make use of them. If security is important, and data members should not be exposed, then `friend` functions may be a better choice. 182

183 **Rational.** Figure 65 on the next page shows the standard arithmetic operators for class `Rational`, implemented as free functions that use the corresponding assignment operators. Even though class `Rational` does not export the numerator and denominator of a fraction, these (free) functions are not declared as `friends` of the class, because they are implemented using the corresponding assignment member functions (`+=`, `-=`, `*=`, `/=`).

If you give users the arithmetic operators, they will expect unary minus as well. Unary minus is best implemented as a member function with no arguments. It does ***not*** negate the value of the fraction, but instead returns the negated value, while the object remains unchanged. Consequently, we can qualify it with `const`: 184

```
const Rational Rational::operator- () const
{
    return Rational(-num, den);
}
```

C++ programmers expect to use `pow` for exponentiation. But `pow` takes arguments of many types, even for the exponent. For fractions, we take the slightly daring approach of providing `^` as an exponential operator. The exponent must be an integer, but the integer may be positive or negative: if e is negative, then r^e is evaluated as $\left(\frac{1}{r}\right)^{-e}$.

overloading operator^ and precedence

If $r = 0$ and $e = 0$, then $r^e = 1$. However, if $r = 0$ and $e < 0$, then r^e evaluates $\frac{1}{r}$, which throws an exception.

The function uses the identity $x^{2e} = \left(x^2\right)^e$ when e is even to achieve complexity $\mathcal{O}(\log e)$ rather than $\mathcal{O}(e)$. Since exponentiation is not commutative, and the left operand must be a `Rational`, we implement `operator^` as a member function, as shown in Figure 66. 185

There is a minor problem with using `operator^` as an exponent operator. Although we can overload operators in C++, we cannot change their precedence. As it happens, `operator^`, which is normally used as exclusive-or on bit strings, has a ***lower*** precedence than the other arithmetic operators. Its precedence is even lower than that of `operator<<` and `operator>>`. So we had better warn our users to put exponential expressions in parentheses!

```
const Rational operator+ (const Rational & left, const Rational & right)
{
    Rational result = left;
    result += right;
    return result;
}

const Rational operator- (const Rational & left, const Rational & right)
{
    Rational result = left;
    result -= right;
    return result;
}

const Rational operator* (const Rational & left, const Rational & right)
{
    Rational result = left;
    result *= right;
    return result;
}

const Rational operator/ (const Rational & left, const Rational & right)
{
    Rational result = left;
    result /= right;
    return result;
}
```

Figure 65: Arithmetic operators for class `Rational`, implemented using the arithmetic assignments of Figure 62 on page 109

```
const Rational Rational::operator^ (int e) const
{
    if (e < 0)
        return (1/ *this)^(-e);
    else if (e == 0)
        return 1;
    else if (e % 2 == 0)
        return (*this * *this)^(e/2);
    else return *this * (*this^(e - 1));
}
```

Figure 66: An exponent function for class `Rational`

Account. Class `Account` does not provide any operators.

6.3.3 Comparison

There are two important kinds of comparison: ***equality*** and ***ordering***. Identity comparison corresponds to the operators `==` and `!=` and is useful for many kinds of objects. Ordering corresponds to the operator `<` and its friends and is useful only for objects for which some kind of ordering makes sense.

By convention, the ordering operators return a Boolean value `true` or `false`. They can implement only a ***total ordering*** in which, for any objects x and y, one of the following must be true: $x < y$, $x = y$, or $x > y$. They cannot be used to implement a partial ordering, in which two objects may be unrelated.

Rational. Fractions are totally ordered. We can define the ordering by

$$\frac{n_1}{d_1} > \frac{n_2}{d_2} \quad \Longleftrightarrow \quad n_1 \times d_2 > n_2 \times d_1.$$

186 187 Figure 67 on the facing page shows the corresponding functions for class `Rational`. They are implemented as free, `friend` functions.

Account. Considering the comparison operators for class `Account` raises interesting questions about equality.

- Does it make sense to say that two accounts are "equal"?
- If it does make sense, what does it mean to say "account A equals account B"?

extensional vs. intensional equality

Comparing balances probably is not very helpful. Comparing names is unsafe, because two people might have the same name.[24] There are two comparisons that might be reasonable:

- Two accounts are equal if they have the same ID (***extensional equality***)
- Two accounts are equal if they are the same object (***intensional equality*** or ***identity***)

The following comparison functions check for identity (accounts are equal only if they are the same object). Note that this is not really realistic because, in a practical application, account objects would spend most of their lives on disks and would only occasionally be brought into memory. 188

```
bool operator==(const Account & left, const Account & right)
{
    return &left == &right;
}

bool operator!=(const Account & left, const Account & right)
{
    return &left != &right;
}
```

[24] This is, in fact, the cause of many "mistaken identity" problems.

```
const bool operator== (const Rational & left, const Rational & right)
{
    return left.num * right.den == right.num * left.den;
}

const bool operator!= (const Rational & left, const Rational & right)
{
    return !(left == right);
}

const bool operator<  (const Rational & left, const Rational & right)
{
    return left.num * right.den < right.num * left.den;
}

const bool operator>  (const Rational & left, const Rational & right)
{
    return left.num * right.den > right.num * left.den;
}

const bool operator<= (const Rational & left, const Rational & right)
{
    return left < right || left == right;
}

const bool operator>= (const Rational & left, const Rational & right)
{
    return left > right || left == right;
}
```

Figure 67: Comparison operators for class `Rational`

It would be straightforward to compare IDs instead.

Accounts could be ordered by name or by ID; this would allow us to sort a `vector` of accounts, for example. These operators are easy to add if needed.

6.3.4 Input and output

For many classes, it is useful to provide the insertion operator (`<<`), if only for debugging purposes. The extraction operator, `>>`, is less often needed, but can be provided as well. When these operators are provided, they are commonly implemented as `friend`s.

Rational. For fractions, both input and output operators make sense, as we implement them as friends. Both present complications.

For input, the first decision we have to make is: what should the user enter? Let’s say that the
189 user must enter two integers separated by a slash:

```
Enter a rational:  2/3
```

The next decision is: how does the program respond if the user does ***not*** enter the data in the form that we expect? There are many possible solutions, ranging from accepting only input that is exactly correct to parsing what the user enters and figuring out what was meant.

In the following function, we follow a middle way: we read an integer, a character, and another integer, and we throw an exception if the character is not `’/’`. Although rationals can be constructed from integers, the user is required to enter a complete fraction, even if it is `3/1`.

```
istream & operator>> (istream & is, Rational & r)
{
    long m, n;
    char c;
    is >> m >> c >> n;
    if (c != ’/’)
        throw Rational::BAD_INPUT;
    r = Rational(m, n);
    return is;
}
```

Output is usually more straightforward than input, but there is one problem. Suppose that we
190 implement the inserter in the “obvious” way

```
os << num << ’/’ << den;
```

and the user writes

```
cout << setw(12) << r;
```

in which `r` is a `Rational`. Our function will use 12 columns to write the numerator and then will write the slash and the denominator using the default field width of zero – probably not what the user expected!

Of the various ways to correct this error, the simplest is to format the entire fraction, in the obvious way, into a buffer, and then to insert the buffer into the output stream. This ensures that any width modifiers will be applied to the complete fraction, not just to the numerator.

The remaining issue is how to write whole numbers (fractions with denominator 1). To avoid sillinesses like `3/1`, we will not write the slash or the denominator for whole numbers. This reasoning leads to the following function.

```
ostream & operator<< (ostream & os, const Rational & r)
{
    ostringstream buffer;
    buffer << r.num;
    if (r.den != 1)
        buffer << ’/’ << r.den;
    return os << buffer.str();
}
```

string stream

String streams can be used for either output (`ostringstream`) or input (`istringstream`) and require the directive

```
#include <sstream>
```

- An output string stream behaves like any other output stream (e.g., `cout`).
- After writing things to it, you can extract the string of formatted data using the function `str`, which returns a `string`.
- After `str` has been invoked on an `ostringstream`, the stream is ***frozen*** and does not permit further write operations.
- When the string stream is deleted (usually at the end of the current scope), data associated with it is deleted.

An input string stream can be used to parse a string. In the following code, the input string stream `isstr` is initialized with `myString`. The `isstr` is used, like any other input stream (e.g., `cin`), with extract operators. 191

```
string myString = "3 4 5 words";
istringstream isstr(myString);
int a, b, c;
string w;
isstr >> a >> b >> c >> w;
```

Hence, string streams in C++ perform a similar function to the `StringTokenizer` class in Java – see (Weiss 2004, Section 9.7) for a comparison.

6.4 Conversions

Conversions are a delicate issue in C++. Programmers don’t like writing explicit conversions and want the compiler to do the dirty work for them. Too many conversions, however, can be a bad thing.

Rational. It seems reasonable to convert rationals to floating-point numbers. Sometimes, for example, we might want 0.66667 rather than 2/3. C++ provides a powerful way of implementing such conversions, using operator notation. We can define a conversion from Rational to double by adding this member function to class Rational: 192

```
operator double () const
{
    return double(num) / double(den);
}
```

implicit type conversions

This function provides an ***implicit conversion*** from Rational to double: in any context where the compiler expects to find an expression of type double, but in fact finds an expression of type Rational, it will insert a call to this function to perform the conversion.

Similarly, we could provide implicit conversion from Rational to bool, so that we could write

```
if (r) ...
```

as an abbreviation for

```
if (r != 0) ...
```

This conversion would be written

```
operator bool () const
{
    return num != 0;
}
```

ambiguity problems with implicit conversions

Unfortunately, both of these functions are a bit ***too*** effective! If we write 1/r, for example, the compiler complains about ambiguity: Given 1/r, the compiler can either convert 1 to Rational and evaluate the reciprocal of r as a Rational ***or*** it can convert both 1 and r to double and evaluate the reciprocal of r as a double. Since we would like to keep the convenience of being able to write 1/r for the Rational reciprocal of a Rational, we choose ***not*** to provide operator double. A similar argument applies to operator bool.

It is not hard to find a better alternative: we simply provide the same function with a funny name to prevent the compiler from using it implicitly. The following function does what we need and will be invoked only when the user explictly requests it by writing r.toDouble(). 193

```
const double Rational::toDouble() const
{
    return double(num) / double(den);
}
```

6.5 Accessors

An ***accessor*** or ***inspector*** is a member function that returns information about an object without changing the object. It follows immediately from this definition that accessors should always return a non-`void` value and should be declared `const`.

Rational. The only candidates for accessors for class `Rational` are `getNum` and `getDen`, implemented in the obvious way. The decision ***not*** to provide these was that the representation of a rational number is a "secret" and providing these accessors would give away part of the secret.

If the class did provide `getNum` and `getDen`, the `friend` functions would no longer need to be declared as friends, because they could use the accessors instead.

Account. There are several candidates for accessors for class `Account`. We provide just an accessor for the account balance, to demonstrate the general idea. 194

```
long Account::getBalance() const
{
    return balance;
}
```

6.6 Mutators

A ***mutator*** is a member function that changes the state of the object. Usually, the return type of a mutator is `void` (otherwise the mutator would be a function with a side-effect).

Rational. Any function that changes the state of a rational should do so in a way that makes semantic sense. Thus `+=` is acceptable, because it changes the state by adding another rational. Arbitrary mutations of the numerator and denominator do not make semantic sense, and so we do not provide mutators for class `Rational`.

Account. There are several ways in which the state of an account may reasonably be changed. We provide three "obvious" functions: `deposit`, `withdraw`, and `transfer`. Of these, `deposit` is straightforward. 195

```
void Account::deposit(long amount)
{
    balance += amount;
}
```

For withdrawals, we have to decide what to do when the balance would go negative. The solution adopted here is to throw an exception. Since we do not know who will be handling the exception, we must ensure that sufficient information is provided. To achieve this, we declare a special exception class (described below) and store the account ID and a helpful message in the exception object.

```
void Account::withdraw(long amount)
{
    if (amount > balance)
        throw AccountException(id,
              "Withdrawal: amount greater than balance");
    else
        balance -= amount;
}
```

what()

Exceptions. Standard exception classes provide a function `what` that returns a description of the exception. Although we could inherit from one of the standard exception classes, we
196 choose not to do so here, but we provide `what` anyway. Figure 68 shows the declaration and
197 implementation of the class `AccountException`.

The third mutator for class `Account` is `transfer`, which transfers money from one account to another. We allow an account to transfer funds ***to*** another account but not to transfer funds ***from*** another account. This function will throw an exception if the transfer would leave the
198 giving account with a negative balance.

```
void Account::transfer(Account & other, long amount)
{
    withdraw(amount);
    other.deposit(amount);
}
```

```
class AccountException
{
public:
    AccountException(long id, std::string reason);
    std::string what();
private:
    long id;
    std::string reason;
};


AccountException::AccountException(long id, string reason)
    : id(id), reason(reason)
{}

string AccountException::what()
{
    ostringstream ostr;
    ostr << "Account " << id << ".  " << reason << ’.’;
    return ostr.str();
}
```

Figure 68: Declaration and implementation of class `AccountException`

The function `transfer` is implemented using the previously defined functions `withdraw` and `deposit`. It is always a good idea to use existing code when it applies, rather than introducing more code, with possible errors.

In real-life banking, an operation such as `transfer` must be implemented very carefully: either the transaction must succeed completely, or it must fail completely and have no effect. (Database gurus are familiar with this problem of ***transactional integrity***.) Without pretending that `transfer` is really a secure function, we note that there is only one (obvious) way that it can fail – `withdraw` may throw an exception – and that this failure leaves both accounts unchanged.

6.7 Odds and ends

6.7.1 Indexing (`operator[]`)

C++ allows us to overload the "operator" `[]`. The overload has one parameter, which can be of any type, and returns a value, which can be of any type. The syntax for calling the function is `a[i]`, in which `a` is an object and `i` is the argument passed to the function. Typically, we would use `operator[]` for a class that represented an array, vector, or similar kind of object. Figure 69 shows an inefficient and incomplete map class that associates names with telephone numbers, both represented as strings. The store is accessed by calling `operator[]` with a string argument. It makes use of the `pair` template defined in `<utility>`. The following code illustrates the use of this store (note the final statement). 199

```
Store myPhoneBook;
myPhoneBook.insert("Abe", "486-2849");
myPhoneBook.insert("Bo", "982-3847");
cout << myPhoneBook["Abe"];
```

200

In this example, `operator[]` has one parameter. This is the only possibility: you cannot declare `operator[]` with no parameters or with more than one parameter.

6.7.2 Calling (`operator()`)

C++ allows us to overload the "operator" `()`. The overload may have several parameters of any type, and returns a value, which can also be of any type. The syntax is `f(x, y, ...)`, in which `f` is an object and `x, y, ...` are the arguments passed to the function. Typically, we would use `operator()` in a situation where it makes sense to treat an instance as a function. Such objects are also known as ***function objects*** (or functors).

function objects (functors)

201 Figure 70 on the following page shows a program that tests a rather simple class called `AddressBook`. The object `myFriends` is an `AddressBook` into which a few names have been entered. The output statement uses this object as if it was a function, providing two names as arguments. The effect, shown on the right, is to display the names within the given range.

The function that makes an instance of `AddressBook` behave like a function is `operator()` in the 202
declaration shown in Figure 71 on page 123. In this example, the function takes two arguments 203
of type `string` and returns a value of type `AddressBook`. 204

205

206

```
class Store
{
public:
    void insert(string key, string value)
    {
        data.push_back(pair<string, string>(key, value));
    }
    string operator[](string key)
    {
        for (   vector<pair<string, string> >::const_iterator it =
                                               data.begin();
                it != data.end();
                ++it )
            if (it->first == key)
                return it->second;
        return "";
    }
private:
    vector<pair<string, string> > data;
};
```

Figure 69: A simple map class

```
int main()
{
   AddressBook myFriends;
   myFriends.add("Anne");
   myFriends.add("Bill");
   myFriends.add("Chun");
   myFriends.add("Dina");
   myFriends.add("Eddy");
   myFriends.add("Fred");
   myFriends.add("Geof");
   cout << myFriends("Bill", "Fred");
   return 0;
}
```

```
Bill
Chun
Dina
Eddy
```

Figure 70: Test program for class `AddressBook` (left) and results (right)

6.7.3 Explicit constructors

A constructor with a single parameter provides a form of type conversion. For example, suppose
207 class `Widget` has a constructor with a parameter of type `int`:

```
class Widget
{
public:
```

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

class AddressBook
{
public:
   void add(string name)
   {
      names.push_back(name);
   }

   AddressBook operator()(string first, string last)
   {
      AddressBook result;
      bool ins = false;
      for (
         vector<string>::const_iterator it = names.begin();
         it != names.end();
         ++it )
      {
         if (*it == first)
            ins = true;
         if (*it == last)
            ins = false;
         if (ins)
            result.add(*it);
      }
      return result;
   }

   friend ostream & operator<<(ostream & os, const AddressBook & st)
   {
      for (
            vector<string>::const_iterator it = st.names.begin();
            it != st.names.end();
            ++it )
         os << *it << endl;
      return os;
   }

private:
   vector<string> names;
};
```

Figure 71: Class `AddressBook`

```
    Widget(int n);
    ....
```

This constructor gives the compiler permission to convert an integer to a `Widget` whenever it has an opportunity to do so. This behaviour might be appropriate, but it might also be an error.
208 For example, class `string` used to have a constructor with an integer argument[25] so that

```
string s(N);
```

could construct a string with N characters. A possible consequence was that

```
string s = 'a';
```

would construct a string with 97 characters – probably not what the programmer intended.

explicit

This kind of implicit conversion can be prevented by qualifying the constructor with `explicit`.
209 To prevent `Widget`s being constructed from integers, rewrite the example above as

```
class Widget
{
public:
    explicit Widget(int n);
    ....
```

The keyword `explicit` does not prevent the constructor from being used at all, of course. It says that, in order to use this constructor, the call `Widget(N)` must actually appear in the program; the compiler will never call the constructor implicitly.

In C++98, the keyword `explicit` can be used with constructors only; it cannot be used with conversion operators such as `operator double` described in Section 6.4 on page 117.

`explicit` conversion operators in C++11

In C++11, you can now also use `explicit` with a conversion operator, to prevent implicit type conversions as discussed in Section 6.4 on page 117. For example, you can now declare

```
explicit operator double () const
```

and use it by explicitly calling

```
double x = double(r)
```

6.7.4 Friends

A function associated with a class can have three properties (Stroustrup 1997, page 278): 210

1. it can access private members of the class
2. it is in the scope of the class
3. it must be invoked on an object

Property 2 means that the function `f` associated with class `C` must be invoked in one of the following ways:

- `C::f()`
- `c.f()` where `c` is an instance of `C`
- `pc->f()` where `pc` is a pointer to an instance of `C`

Property 3 means that statements in the function can use the `this` pointer to the object for which the function is invoked.

The following table shows which kinds of function have these properties: 211

	Property		
Function kind	1	2	3
`friend`	√		
`static`	√	√	
member	√	√	√

The table shows that the three kinds of function form a hierarchy, with member functions having the most, and friend functions the least, access to the class.

Contrary to popular belief (especially amongst Java programmers), friends do not violate encapsulation in C++. The important point is that friends ***must be declared within the class***. This means that a class has complete control over its friends – enemies cannot use friends to subvert the class' protection mechanisms. Here is Stroustrup's opinion of friends (Stroustrup 1994, page 53): 212

> *A friendship declaration was seen as a mechanism similar to that of one protection domain granting a read-write capability to another. It is an explicit and specific part of a class declaration. Consequently, I have never been able to see the recurring assertions that a* `friend` *declaration "violates encapsulation" as anything but a combination of ignorance and confusion with non-C++ terminology.*

Here are some useful things to know about friends: 213

- A `friend` declaration can be placed anywhere in a class declaration. It is not affected by `public` or `private` attributes.
- A function or a class can be declared as a `friend`. In either case, friends of a class can access private members (data and functions) of the class.

[25] This constructor was eventually removed to avoid confusions like the one described here.

- Two or more classes can declare the same class or function as a friend.

Used correctly, friends actually provide ***better*** protection than other techniques. Here is an example (Stroustrup 1997, page 278): we have classes `Vector` and `Matrix` and we want to define functions that, for example, multiply a matrix by a vector. We could define the multiply function as a free function, external to both classes, but then we would have to expose the representations 214 of both `Vector` and `Matrix` for this function to use. Instead, we use friends:

```
class Vector
{
    friend Vector operator* (const Matrix & m,
                             const Vector & v);
    ....
}

class Matrix
{
    friend Vector operator* (const Matrix & m,
                             const Vector & v);
    ....
}

Vector operator* (const Matrix & m,
                  const Vector & v)
{
    ....
}
```

This provides a neat solution to the problem: the multiply function has access to private data in ***both*** classes but we have not "opened them up" to anyone else.

Another design consideration for choosing between a member function or a friend is conversion, as discussed in Section 6.4: in the call of the free (friend) function `f(x,y)`, the compiler may perform conversions to match the types of both the arguments `x` and `y` to the parameter types. The call to the member function `x.f(y)` can only invoke `X::f` where `x` is the class of `X` (although `y` may still be converted).

6.8 Summary

Here is a summary of functions and operators that might be included in a class declaration. We use `T` to stand for the name of the class being declared.

`T()` ***Default constructor.*** The compiler generates a default constructor, but only if the programmer declares ***no*** constructors. The compiler-generated default constructor calls the default constructors of the instance variables of the class.

The programmer may declare a default constructor. A default constructor does not need any parameters. Any parameters that it does have must have default values.

`T(...)` ***Constructor.*** Programmer may provide as many constructors as needed. The parameters of constructors must differ in number and type in accordance with normal overloading rules.

`explicit T(...)` ***Explicit constructor.*** If a constructor has a single parameter, the compiler is allowed to use it to perform conversions. For example, if class `T` has a constructor `T(int n)`, the following code implicitly invokes this constructor as `T(42)`:

```
T x;
x = 42;
```

Sometimes this behaviour is undesirable. To prevent implicit conversion, qualify the constructor with `explicit`. An `explicit` constructor is called only if its name appears in the source code. If `T(int)` was declared `explicit`, the code above would not compile; it would have to be rewritten as:

```
T x;
x = T(42);
```

We could also write just

```
T x(42);
```

because the compiler considers this to be an explicit call of the constructor.

`T(const T & x)` ***Copy constructor.*** The compiler generates a default copy constructor if the programmer does not declare a copy constructor. The default copy constructor performs a bitwise copy of each instance variable of the object. This is usually appropriate for value fields but incorrect for pointer fields.

The programmer may declare a copy constructor with the signature given above. The definition can perform any actions but should normally construct a semantic copy of the argument.

`~T()` ***Destructor.*** The compiler generates a default destructor if the programmer does not declare a destructor. The default destructor deallocates memory used by the object but does nothing else.

The programmer may declare a destructor with the signature given above. The destructor has no parameters and no return type. It may perform any actions but is normally used to destroy objects created dynamically during the object's lifetime.

`T & operator=(const T & x)` ***Assignment operator.*** The compiler generates a default assignment operator that performs a bitwise copy of all of the fields of the argument.

The programmer may declare an assignment operator with the signature given above. It may perform any actions but is normally used to construct a semantic copy of the argument and return a reference to it.

`T operator`⟨***unop***⟩`()` ***Unary operator.*** This is a unary operator. The implicit argument is `*this` and the function applies the unary operation to it and returns the result. The return type may be a reference (i.e., `T &`).

The unary operators that may be overloaded are:

```
!   &   ~   *   +   -   ++   --
```

For `++` and `--`, the compiler must be able to distinguish between prefix usage (e.g., `++n`) and postfix usage (e.g., `n++`). The appropriate declarations for class `T` are shown below. The parameter `int` enables the compiler to distinguish prefix and postfix; it must not be accessed inside the function definition. Note that the prefix form returns an Lvalue but the postfix form returns a Rvalue.

```
T& operator++();        // Prefix increment operator.
T operator++(int);      // Postfix increment operator.
T& operator--();        // Prefix decrement operator.
T operator--(int);      // Postfix decrement operator.
```

Unary operators in class `T` do not have to return a value of type `T`, although they normally do so.

Unary operators may also be declared as free functions, outside a class, with syntax

`T operator`⟨*unop*⟩`(const T & x)`

but then their connection with class `T` is through the parameter types only.

`T operator`⟨***binop***⟩`(const T & x)` ***Binary operator.*** This is a binary operator declared as a member function.

If `operator+`, for example, is defined as a member function, the expression `a+b`, with `a` an instance of `T`, is treated as `a.operator+(b)`. This means that `a+b` and `b+a` are treated differently, because the right operand may be converted but the left operand cannot be converted.

The binary operators that can be overloaded are:

```
,   !=   %   %=    &     &&   &=   *    *=    +    +=
-   -=   ->   ->*   /    /=   <    <<   <<=   <=
=   ==   >   >>    >>=   ^    ^=   |    |=    ||
```

The types in these examples are all the same (`T`), but this is not necessary. Code such as the following is permissible, although somewhat unusual:

```
class Money
{
   ...
   double operator+(int n);
};
```

This declaration introduces a member function that adds a `Money` value to an `int` and returns a `double` value.

Binary operators can also be overloaded in free functions. In this case, they have two parameters, corresponding to the left and right operands. For example, the following declaration permits multiplying a scalar by a vector:

```
Vector operator*(double scal, const Vector & vec);
```

`operator X ()` ***Conversion.*** In this form, `X` is a type not equal to the class type, `T`. The effect of providing a member function with this signature is to provide an automatic conversion from `T` to `X`. In other words, if the compiler is expecting a value of type `X`, but finds a value of type `T`, it will automatically insert a call to this function. A conversion operator declaration must not have parameters (the operand is the object itself).

The last line in the following example implicitly calls `operator Foo` to perform the conversion from `Bar` to `Foo`:

```
class Foo { ... };

class Bar
{
   ...
   operator Foo() { ... };
   ...
};

Bar b;
Foo f = b;
```

`X & operator[](...)` ***Indexing.*** If a class provides `operator[]`, instances of the class can be "indexed", as in `a[i]`.

The return type is usually not the class type. Typically, the class would be a template class representing a collection of some kind, and `X` would be the template parameter. For example:

```
template<typename Vehicle>
class ParkingLot
{
   ...
   Vehicle operator[](int i) const;
   Vehicle & operator[](int i);
   ...
};

class Car { ... };

Car myVeyron;
ParkingLot<Car> mall;
...
mall[76] = myVeyron;
```

It is usually necessary to provide two versions of `operator[]`, as in the example above. The first version returns an Rvalue that can be used with `const` objects but not on the left of `=`. The second version returns an Lvalue that can be used on the left of `=` but cannot be used with `const` objects.

A declaration of `operator[]` must have exactly one parameter. The parameter can have any type, and any type may be returned.

`X operator()(...)` ***Function.*** If a class provides `operator()`, instances of the class can be used as if they were functions, as in `f(x)`.

A declaration of `operator()` can have any number of parameters, including zero, and may return a value of any type.

Friends. Friends of a class are classes and functions that have access to the private data of the class. Their declarations must appear within the class declaration; their definitions may be elsewhere.

References

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley.

Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley.

Stroustrup, B. (1997). *The C++ Programming Language*. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.

7 Templates and Generic Programming

7.1 Template Functions 131
7.2 Template Classes 138
7.3 Compiling Template Code 142
7.4 Template Return Types 143
7.5 Template Specialization 144
7.6 Template Metaprogramming 146
7.7 Default Arguments 147

Templates enable us to write generic, or parameterized, code. The basic idea is simple but some of the implications (at least for C++) are subtle.

You can find more information on the material in this section in (Weiss 2004, Chapter 7), (Koenig and Moo 2000, Chapter 8), or (Prata 2012, Chapter 14). For a more detailed and complete description of templates, (Vandevoorde, Josuttis, and Gregor 2017) is recommended.

7.1 Template Functions

Consider these three functions:[26] 216

note that you cannot write a swap function like this in Java, since it does not provide call-by-reference

```
void Swap(char & x, char & y)      // Function (1)
{
   char t = x; x = y; y = t;
}

void Swap(int & x, int & y)
{
   int t = x; x = y; y = t;
}

void Swap(double & x, double & y)
{
   double t = x; x = y; y = t;
}
```

These functions perform the same task for different types. Since C++ allows overloading, we could use all three in the same program, but there does appear to be some redundancy.

7.1.1 Type Parameters

We can use templates to avoid source code redundancy. We replace the previous three definitions by a ***template function declaration*** as shown in Figure 72 on the following page. 217

[26] The name Swap, rather than swap, was chosen to avoid confusion with the standard library function.

```
template <typename T>
void Swap(T & x, T & y)
{
    T t = x;
    x = y;
    y = t;
}
```

Figure 72: Changing `Swap` into a template function

218 The new function can be called in the same way as the previous versions. It is not necessary to specify the type of the arguments, because the compiler already has this information. The code

```
char c1 = 'a';
char c2 = 'b';
Swap(c1, c2);
cout << c1 << ' ' << c2 << endl;

int m = 3;
int n = 5;
Swap(m, n);
cout << m << ' ' << n << endl;
```

prints

```
b a
5 3
```

219 To compile the call `Swap(c1, c2)`, the compiler must perform the following steps:

1. Infer the types of the arguments `c1` and `c2` (in this case: `char` and `char`).
2. Look for functions named `Swap`.
3. Find (in this case) a template function `Swap`.
4. Check that the substitution $T =$ `char` matches the call.
5. Generate source code for the function `Swap<char>` (which should be the same as Function (1) above).
6. Compile the generated source code.
7. Generate a call to this function.

The compiler detects errors in the template code, if there are any, at Step 6., when it compiles the code obtained by expanding the template. This means that errors in template code are reported ***only if the template is used***. You can write all kinds of rubbish in a template declaration and the compiler won't care if you never use the template.

The process of deriving an actual function from a template declaration is often called “instantiating” the template. This usage is confusing, because we also talk about objects obtained by instantiating a class. These notes use the expression ***applying a template*** or ***the application of a template***, by analogy with “applying a function”.

Although templates remove redundancy in the source code, they do not affect the object code. If the program calls `Swap` with N different argument types, there will be N versions of `Swap` in the object code.[27]

The following template function `Max` returns the greater of its two arguments.[28] 220

```
template <typename T>
const T Max(const T & x, const T & y)
{
    return x > y ? x : y;
}
```

We would expect this function to work with various types, and indeed it does. Each of the following statements compile and execute correctly:

```
cout << Max(7, 3) << endl;
cout << Max(’a’, ’5’) << endl;
cout << Max(7.1, 3.3) << endl;
```

This statement ***appears*** to execute correctly:

```
cout << Max("COMP", "6441") << endl;
```

But further investigation reveals problems. For instance, executing 221

```
cout << Max("apple", "berry") << endl;
```

displays `berry` but executing 222

```
cout << Max("berry", "apple") << endl;
cout << Max("apple", "berry") << endl;
```

displays `apple` twice. Even worse, the statement 223

```
cout << Max("berry", "cherry") << endl;
```

gives a compiler error:

```
error C2782: ’T Max(const T &,const T &)’ :
   template parameter ’T’ is ambiguous
```

224 To find out what is going wrong, we add a line to Max:

```
template <typename T>
const T Max(const T & x, const T & y)
{
    cout << "Max called with " << typeid(x).name() << endl;
    return x > y ? x : y;
}
```

In order to compile this, we must add the directive

```
#include <typeinfo>
```

225 Executing

```
cout << Max("apple", "berry") << endl;
cout << Max("cherry", "orange") << endl;
```

displays

```
Max called with char const [6]
apple
Max called with char const [7]
orange
```

and reveals the problem: we cannot compare strings of different lengths because they have different types. Also, Max is not comparing the strings; it is comparing the addresses of the strings (i.e., the pointers).

There are no good, general solutions to problems like this. For Max, the best thing to do is to
226 define a non-template version for strings:

```
const string Max(const string & x, const string & y)
{
    return x > y ? x : y;
}
```

When the compiler encounters Max("apple", "berry"), it will consider the string version a better match than the template version.

7.1.2 Missing Functions

Suppose, however, that we define our own class `Widget` as follows: 227

```
class Widget
{
public:
    Widget(int w) : w(w) {}
private:
    int w;
};
```

Note that the private variable `w` is set using a *constructor initializer*, which we discussed in the previous lecture.

Attempting to use `Max` with widgets

```
Max(Widget(1), Widget(2));
```

gives several error messages, including this one:

```
error C2676: binary ’>’ : ’const Widget’ does not define
    this operator or a conversion to a type acceptable
    to the predefined operator
```

It is easy to see what has happened: the compiler has generated the function 228

```
const Widget Max(const Widget & x, const Widget & y)
{
    return x > y ? x : y;
}
```

and has then discovered that `Widget` does not implement `operator>`. The correction is also straightforward: we just have to add the function

```
friend bool operator>(const Widget & left, const Widget & right)
{
    return left.w > right.w;
}
```

to the declaration of class `Widget`.

> ***Ensure that template arguments satisfy the requirements of the corresponding template parameter.***

[27] Note that this is quite different from Java’s *Generics*, where only one object copy of the generic code is used for all type parameters (the generic types are removed at compile-time through a process called *type erasure).*

[28] The name `Max` is used to avoid confusion with the library function `max`.

7.1.3 Conversion Failure

229 The use of templates prevents some of the conversions that we expect. If we declare

```
const double Max(const double & x, const double & y)
{
    return x > y ? x : y;
}
```

then the following calls all compile and execute correctly:

```
Max(1.2, 3.4);
Max(1, 3.4);
Max(1, 3);
```

The first call works because the types match exactly, and the other two calls work because the compiler includes code to convert 1 and 3 from `int` to `double` before the function is called.

With the template version, however, the compiler does not allow the second call. It matches `Max(int,int)` and `Max(double,double)` to the template pattern, but `Max(int,double)` does not match and the compiler will not insert conversions to make it match. 230

231 There are several ways to make the second call work properly:

- We can cast the argument that is causing the problem:

  ```
  Max(static_cast<double>(1), 3.4);
  ```

- We can specify the template argument explicitly:

  ```
  Max<double>(1, 3.4);
  ```

- We can avoid calls of the form `Max(1, 3.4)` with mixed-type arguments.

- We can define a template for mixed types using casting:

  ```
  template <typename T, typename U>
  const T Max(const T & x, const U & y)
  {
      return x > static_cast<T>(y) ? x : static_cast<T>(y);
  }
  ```

This is dangerous in this example, since the result can now be different depending on whether you call `(int, double)` or `(double, int)`!

- We can provide ***both*** a template version and a specialized version of the function. C++ will always pick the function that provides the most closely matched argument types. 232

7.1.4 Non-type Parameters

Template parameters are not restricted to class types. We can also use integral types (`char`, `short`, `int`, `long`, ...) as parameters. This function has an integer template parameter.[29] 233

```
template<int MAX>
int randInt()
{
    return rand() % MAX;
}
```

and is used like this:

```
// Throwing dice
for (int i = 0; i != 20; ++i)
    cout << setw(2) << randInt<6>() + 1;
cout << endl;
```

The argument corresponding to a non-type template parameter can be a constant, but it cannot be a variable: 234

```
const int MAXRAND = 100;
.... randInt<MAXRAND>() ....    // OK

int MAXVAR = 100;
.... randInt<MAXVAR>() ....     // Compiler error
```

Of course, we could have written this function without using templates: 235

```
int randInt(const int MAX)
{
    return rand() % MAX;
}
```

Then we would call it in the usual way: `randInt(6)`.

The difference between the two versions of `randInt` is that the template version substitutes the integer ***at compile time*** whereas the conventional version substitutes the integer ***at run time***. In this case, the difference is slight – the template version probably runs slightly faster than the conventional version but the difference will hardly be noticeable – but may be significant in more realistic situations.

templates: design trade-off between compile-time and run-time overhead

7.2 Template Classes

Classes can be parameterized with templates; the notation is similar to that of function templates.
236 Here is a simple class for 2D coordinates, in which each coordinate consists of two `float`s:

```
class Coordinate
{
public:
    Coordinate(float x, float y) : x(x), y(y) {}
    void move(float dx, float dy);
private:
    float x;
    float y;
};
```

To parameterize this class, we:

1. Write `template<typename T>` in front of it;
2. replace each occurrence of `float` by `T`; and
3. – important! – within the declaration, replace occurrences of the class name C by C`<T>`.

Performing these steps for class `Coordinate` yields the following declaration. Note that, whenever the class name (`Coordinate`) appears ***within*** the declaration, it must have the argument `<T>`:

```
template<typename T>
class Coordinate
{
public:
    Coordinate<T> (T x, T y) : x(x), y(y) {}
    void move(T dx, T dy);
private:
    T x;
    T y;
};
```

Unlike functions, the compiler cannot infer the argument type for classes. Whenever we create an application of a template class, we must provide a suitable argument. The following statements
237 create ***two different classes*** and one instance of each:

```
Coordinate<int> p(1, 2);
Coordinate<float> q(3.4, 5);
```

The integer argument `5` is acceptable for the `Coordinate<float>` constructor because the template application is explicit: the compiler knows that `float` values are expected, and inserts the appropriate conversion.

Here is how `move` is defined for class `Coordinate`:

[29] This is a terrible way to generate random integers! We will consider better ways later.

```
template<typename T>
void Coordinate<T>::move(T dx, T dy)
{
    x += dx;
    y += dy;
}
```

In general, if the definition of a member function for a template class uses the template parameter, we must:

1. Write `template<typename T>` before the function; and
2. write *C*`<T>` wherever the class name *C* is needed.

Like functions, template classes can have non-type template parameters, provided that the parameters have integral types. Instances of the class must be provided with constant arguments of appropriate types.

7.2.1 A Template Class for Coordinates 238

Figure 73 on the next page shows part of a template class for coordinates. It is parameterized by 239
`Type`, the type of a coordinate element, and `Dim`, the dimension of the coordinates. A typical 240
declaration would be 241

```
Coordinate<double, 3> c;
```

If the program uses many coordinates of this type, we would probably define a special type for them, to reduce the amount of writing required and improve the clarity of the program:

```
typedef Coordinate<double, 3> coord;
```

Some points to note about the declaration of `Coordinate`:

- There is a default constructor that sets the elements of the coordinate to zero. The compiler's default constructor would leave the elements uninitialized.
- The function `operator[]` returns a reference to a coordinate element. This function allows a user to get or set any element. This function is called by code like this:

  ```
  x = c[0];
  c[1] = y + 1;
  ++c[2];
  ```

- It is good practice (as we will discuss later) to provide two versions of `operator[]`, one returning an Lvalue (as done here) and the other returning an Rvalue.
- The function `operator[]` performs a range check and aborts the program if the range check fails. It would probably be better to handle a range check error by throwing an exception (which we will also discuss later).

```
template<typename Type, int Dim>
class Coordinate
{
public:
Coordinate<Type, Dim>()
{
    for (int d = 0; d != Dim; ++d)
       c[d] = 0;
}

Type & operator[](int i)
{
    assert(0 <= i && i < Dim);
    return c[i];
}

friend Coordinate<Type, Dim> operator+(
      const Coordinate<Type, Dim> & left,
      const Coordinate<Type, Dim> & right )
{
    Coordinate<Type, Dim> result;
    for (int d = 0; d != Dim; ++d)
        result.c[d] = left.c[d] + right.c[d];
    return result;
}

friend ostream & operator<<(ostream & os,
                            const Coordinate<Type, Dim> & coord)
{
    os << ’(’;
    for (int d = 0; d != Dim; ++d)
    {
        os << coord.c[d];
        if (d < Dim - 1)
            os << ", ";
    }
    return os << ’)’;
}

private:
    Type c[Dim];
};
```

Figure 73: Part of a template class for coordinates

- Several of the functions have `for`-loops for the range `[0,Dim)`. A good compiler might optimize these away for small values of `Dim` (this optimization is a special case of ***loop unrolling***). If we were worried about the overhead of a `for`-loop, we could rewrite the code using `if` or `switch` statements. Figure 74 on the following page shows how this 242 might be done for `operator+`. The code looks long but, when the compiler expands 243 `Coordinate<double,3>`, it will see something like this: 244

  ```
  if (3 == 1)
      ....
  else if (3 == 2)
      ....
  else if (3 == 3)
      ....
  else
  ```

  Any reasonable compiler should be smart enough to compile the code for the "`3 == 3`" case and generate no code for the conditional expressions or the other cases.

  `<aside>` People sometimes wonder why a compiler should bother optimizing code such as

  ```
  if (1 == 0)
      ....
  ```

  on the grounds that no sane programmer would write code like this. Such optimizations are important, and very common, because a lot of code is ***not*** written explicitly by programmers, but is generated by template expansion, code generators, and in other ways. Unless the generator is very smart, generated code may be very stupid. `</aside>`

When we have obtained a type by applying a class template, we can do all of the usual things with it: 245

```
Coordinate<double, 3> c;            // 3D coordinate
Coordinate<double, 3> & rc;         // reference to a 3D coordinate
const Coordinate<double, 3> & rc;   // constant reference
                                    //    to a 3D coordinate
Coordinate<double, 3> * pc;         // pointer to a 3D coordinate
....
```

7.2.2 `class` or `typename`?

Early versions of C++ with templates used "`class`" where we have been using "`typename`". For example: 246

```
template<class T>
class Coordinate { ....
```

This usage suggested that the argument replacing `T` had to be a class and could not be a built-in type, such as `int`. The keyword `typename` suggests that ***any*** type can be used, including the built-in types.

```
friend Coordinate<Type, Dim> operator+(
      const Coordinate<Type, Dim> & left,
      const Coordinate<Type, Dim> & right )
{
    Coordinate<Type, Dim> result;
    if (Dim == 1)
    {
        result.c[0] = left.c[0] + right.c[0];
    }
    else if (Dim == 2)
    {
        result.c[0] = left.c[0] + right.c[0];
        result.c[1] = left.c[1] + right.c[1];
    }
    else if (Dim == 3)
    {
        result.c[0] = left.c[0] + right.c[0];
        result.c[1] = left.c[1] + right.c[1];
        result.c[2] = left.c[2] + right.c[2];
    }
    else
    {
        for (int d = 0; d != Dim; ++d)
            result.c[d] = left.c[d] + right.c[d];
    }
    return result;
}
```

Figure 74: A more elaborate version of `operator+`

Prefer* `typename` *to* `class` *for template parameters.

7.3 Compiling Template Code

We have seen that normal practice in C++ programming is to put declarations into header files
and definitions into implementation files. This does not work for templates. The compiler cannot
247 generate code for a function such as

```
template<typename T>
void Coordinate<T>::move(T dx, T dy)
{
    x += dx;
    y += dy;
}
```

without knowing the argument that replaces `T` and, if this definition is in an implementation file, the compiler cannot access it when it compiles a call such as `v.move(2,3)` (remember that implementation files must never depend on other implementation files, as discussed in Section 4.3 on page 72).

There are several solutions. Different platforms have different policies, but a solution that works on all platforms is to treat any template code, even a function such as `move` above, as a ***declaration***, and put it into a header file.

Put all template code into header files.

7.4 Template Return Types

It does not make sense to use a template type as a return type like this: 248

```
template<typename T>
T f()
{
    return ???
}
```

Even if we could write a sensible expression after `return`, the compiler could not deduce the template argument at the call site: In C++, it is not possible to overload a function based on the return type alone.

In general, if the return type of a function is a template parameter, then the function must have at least one parameter typed with the same template parameter, as in:

```
template<typename T>
T f(const T & param) { .... }
```

But even with a template parameter type, it can be difficult to figure out the actual return type. Consider a template function 249

```
template<typename T>
??? f(T t)
{
    return t.foo();
}
```

Since `foo` can return different types for different parameter types `T`, we cannot provide a fixed return type for this function.

In C++98, there is no solution for this. Since C++11, it is possible to ***automatically deduce*** the type, using the `auto` keyword. We have to make two changes to our template function: First, move the (still unknown) return type ***after*** the input type(s). This is another new C++11 feature, called ***trailing return type***, specified with the `->` operator. Second, let the compiler deduce the type of (here) `f.foo()` through the `decltype` keyword:

`auto`
`->`
`decltype`

```
template<typename T>
auto f(T t) -> decltype(t.foo())
{
    return t.foo();
}
```

It is also possible to declare new variables using `decltype` within the body of a function:

```
decltype(t.foo()) bar;      // make bar the same type as t.foo()
```

Of course, the trailing return syntax, `decltype` and `auto` also work for non-template functions.

7.5 Template Specialization

It is sometimes necessary or desirable, for efficiency or other reasons, to provide a special
implementation for some particular value of the template parameters. Suppose, for example,
that we want to have both a general template class for all kinds of coordinates, as above, but we
also want to give special treatment to three-dimensional coordinates with `double` elements. This
250 is called ***specialization***. Suppose the original class was

```
template<typename T>
class Widget
....
```

The specialized version will begin

```
template<>
class Widget<specType>
....
```

where `specType` is the value of `T` for which we are providing a specialized implementation. In
the rest of the class, we must replace all occurrences of `<T>` with `specType`, including changing
251 `Widget<T>` to `Widget<specType>`.

252 Figure 75 on the facing page shows the result of specializing the generic coordinate class for 3D
253 `double` coordinates. The constructor has been modified with the addition of an output statement
254 to demonstrate that the specialized version is actually used. The definition

```
Coordinate<double,3> c;
```

produces the message

```
3D Coordinate
```

```
template<>
class Coordinate<double,3>
{
public:
    Coordinate<double,3>()
    {
        c[0] = 0; c[1] = 0; c[2] = 0;
        cout << "3D Coordinate" << endl;
    }

    double & operator[](int i)
    {
         assert(0 <= i && i < 3);
         return c[i];
    }

    friend Coordinate<double,3> operator+(
          const Coordinate<double,3> & left,
          const Coordinate<double,3> & right )
    {
        Coordinate<double,3> result;
        result.c[0] = left.c[0] + right.c[0];
        result.c[1] = left.c[1] + right.c[1];
        result.c[2] = left.c[2] + right.c[2];
        return result;
    }

    friend ostream & operator<<(ostream & os,
                                const Coordinate<double,3> & coord)
    {
        return os << '(' <<
            coord.c[0] << ", " <<
            coord.c[1] << ", " <<
            coord.c[2] << ')';
    }

private:
    double c[3];
};
```

Figure 75: A specialized version of class `Coordinate`

Within the declaration of the specialized version, we have unrolled the for-loops and made other small changes. We could also add functions, such as cross product, that are useful for 3D coordinates but not other coordinates.

It is also possible to ***partially specialize*** a template class with two or more template parameters. For example, we could specialize Coordinate to 3D coordinates with any element type. The class declaration would begin 255

```
template<typename Type>
class Coordinate<Type,3>
....
```

Within the class declaration, instances of Dim are replaced by 3, but instances of Type are left unchanged. References to the class all have the form Coordinate<Type,3>.

7.6 Template Metaprogramming

Template specialization is the key to ***template metaprogramming***. The following class declaration is allowed: 256

```
template<int N>
class Fac
{
public:
    static const int val = N * Fac<N-1>::val;
};
```

However, there is a problem: instantiating Fac<4> requires instantiating Fac<3> requires instantiating . . . . We can terminate the recursion by providing a specialized class Fac<0>, as follows:

```
template<>
class Fac<0>
{
public:
    static const int val = 1;
};
```

257 With these declarations, the following code returns 5040:

```
int n = Fac<7>::val;
```

Note that this computation is performed at *compile-time*, not at *run-time*! When the program is executed, the value 5040 has already been computed and is assigned like a constant. Like with other templates, this only works for constants or literals, since the value must be known at compile-time.

A consequence of template metaprogramming is that the compilation can take significantly longer, since now a potentially large amount of computations have to be performed by the compiler (think of the template metaprogram as a program that is running inside the compiler, leaving its results in the object code).

Unfortunately, it is generally not possible to look at the C++ code generated from a template, since this is handled internally in the compiler (rather than by a preprocessor, where you can inspect the results before compilation). However, you *can* look at the produced object code using a debugger or disassembler. On the x86 architecture, the following instructions[30] are generated for a function containing only the call to `int n = Fac<7>::val;`:

```
pushl %ebp
movl %esp, %ebp
subl $4, %esp
movl $5040, -4(%ebp)
leave
ret
```

here you can see the constant `$5040` computed by the template metaprogram. You've seen another example for template metaprogramming in the first lecture (Figure 2), where a metaprogram generated a parser for a given grammar at compile-time.

7.7 Default Arguments

Function declarations may have default arguments: 258

```
void foo(int n = 0) { .... }
```

The same is true of template class declarations. For example, if we changed the declaration of `Coordinate` in Figure 73 on page 140 to

```
template<typename Type = double, int Dim = 3>
class Coordinate
....
```

and defined

```
Coordinate<int> u;
Coordinate<> c;
```

then `u` would be a 3D coordinate of `int`s and `c` would be a 3D coordinate of `double`s.

Note that the definition

```
Coordinate c;
```

is ***not allowed***. The brackets `<>` are required even when we are using the default values of all parameters. (The same is true for functions, of course.)

[30] For details on x86 assembly syntax, you can consult the Wikibook "x86 Assembly": http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax.

References

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Vandevoorde, D., N. M. Josuttis, and D. Gregor (2017). *C++ Templates* (2nd ed.). Addison-Wesley. http://tmplbook.com.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.

8 Inheritance and Polymorphism

8.1 What is inheritance? 149

8.2 Designing a base class 154

8.3 Designing a derived class 161

8.4 Conversion and Casting 169

8.5 Advanced inheritance topics 174

With the knowledge of the previous lectures, we can finally look into working with inheritance and polymorphism in C++. You can find more details on this topic in (Prata 2012, Chapter 13). For a comparison with Java, see (Weiss 2004, Chapter 6). More details on specific aspects can be found in Scott Meyers's book (Meyers 2005, Chapter 6).

We will use (mostly) C++ terminology for inheritance: a ***derived class*** inherits from a ***base class***. The following table shows alternative terminology used in other languages and in general OOP discourse: 260

C++	Alternatives	
Base class	Parent	Super class
Derived class	Child	Subclass

Inheritance may be `public` (the most common case) or `private`, depending on the keyword used before the base class in the derived class declaration: 261

```
class Base { .... };
class D1 : public Base { .... };     // public inheritance
class D2 : private Base { .... };    // private inheritance
```

8.1 What is inheritance?

Perhaps the most confusing aspect of the word "inheritance" is that in C++, it is used with several quite different meanings:

1. Inheritance as an "is-a" relationship. is-a

   Inheritance as an "is-a" relationship models the human urge to form hierarchies of classification ("taxonomies"). Each of the following sets contains its successors: living things, animals, mammals, dogs, terriers. Read backwards, these sets form an is-a hierarchy: a terrier ***is a*** dog, a dog ***is a*** mammal, a mammal ***is an*** animal, etc. taxonomy

   The important feature of a taxonomy is that a smaller set ***inherits*** all of the behaviour of the larger sets that contain it. A dog has ***all*** of the properties of a mammal – otherwise it wouldn't be a mammal. (We discuss apparent exceptions to this rule below.)

   There are two kinds of is-a relationship, depending on what gets inherited. A base class declares various functions, and the derived class(es) may inherit:

a) the declarations of the function only, or

b) the declarations and definitions of the functions.

interface inheritance and implementation inheritance

We refer to the first case as ***interface inheritance*** and the second case as ***implementation inheritance***.[31]

2. Inheritance as "implemented using" (a "has-a" relation). For example, we could implement a `Stack` by inheriting a `vector`, giving a "stack implemented using vector" class. The important point here is that `vector` and `Stack` provide different sets of functions, but the `Stack` functions are easily implemented using the `vector` functions. However, users of `Stack` must not be allowed to use the `vector` functions because that would violate the integrity of the stack.

The implementation mechanisms for these kinds of inheritance in C++ are, roughly:

1. `public` inheritance implements "is-a"
   a) abstract base class with pure `virtual` functions (see Section 8.2.1 on page 154)
   b) base class with `virtual` declarations and default definitions
2. `private` inheritance implements "using" (or "has-a")

abstract base class

The distinction between 1(a) and 1(b) can become blurred. A base class with one or more pure virtual functions is called an ***abstract base class*** (or "ABC") and defines, in effect, an interface: this is 1(a). A base class with implemented virtual functions is a complete, working class that can be specialized by derived classes that inherit some or most of its implementation and modify the rest: this is 1(b). In between, there are base classes with a mixture of pure and impure virtual functions, defining a sort of partially-implemented interface.

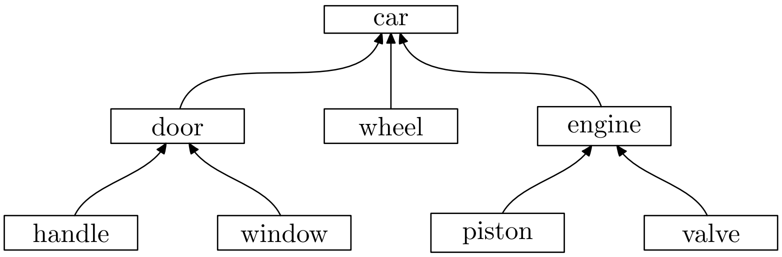


Figure 76: Not an "is-a" inheritance hierarchy

is-a vs. has-a

262

It is very important not to confuse "is-a" with "has-a". In some (older) books, you may see things like Figure 76 described as "inheritance hierarchies". But this is a "has-a" hierarchy: a car has an engine, an engine has pistons, etc. It is absurd to say "a valve is an engine" or "a handle is a door". Hierarchies of this kind are represented by ***layering*** (also called ***composition***), in which an instance of class `Car` contains instances of classes `Door`, `Wheel`, and `Engine`, etc. This confusion stems from the fact that C++ can support both "has-a" hierarchies (with the `private`

[31] These correspond roughly to `implements` and `extends` in Java. Note, however, that the Java keyword `implements` corresponds to ***interface inheritance*** in C++ and the Java keyword `extends` corresponds to ***implementation inheritance*** in C++.

```
class Bird
{
public:
    virtual void fly();
    ....
};

class Penguin : public Bird
{
    ....
};

int main()
{
    Bird *pb = new Penguin;
    pb->fly();                // oops - penguins can’t fly!
}
```

Figure 77: The problem of the flightless penguin

inheritance) and “is-a” hierarchies (through `public` inheritance). However, in modern object-oriented programming, inheritance generally refers to “is-a” hierarchies (also called ***taxonomies***) only; and in this course, we focus almost exclusively on this type of inheritance (we’ll briefly discuss `private` inheritance in Section 8.5.1 on page 174).

Figure 77 shows a popular example used to demonstrate the “difficulties” of inheritance. There 263
are various solutions for this “problem”. One solution is to redefine `Penguin::fly`: 264

```
class Penguin : public Bird
{
    void fly() { throw PENGUINS_CANNOT_FLY_EXCEPTION; }
};
```

This is not a good solution, because it violates the assumption in base class `Bird` that “birds
fly”. Figure 78 on the following page shows a better solution. It is obtained by realizing that the 265
class hierarchy that we have defined is incomplete: there are birds that fly and birds that don’t 266
fly. The revised class hierarchy easily accommodates kiwis, ostriches, turkeys, and cassowaries.

8.1.1 Constructors and destructors

Constructors and destructors work in a special and well-defined way with derived classes.

- When a constructor of a derived class is called, the constructor of the base class is invoked first, then the constructor of the derived class.
- When the destructor of a derived class is called, the destructor for the derived class is invoked first, then the destructor for the base class.

```
class Bird
{
public:
    // no definition of fly()
    ....
};

class FlyingBird : public Bird
{
    virtual void fly();
};

class NonFlyingBird : public Bird
{
public:
    // no definition of fly()
    ....
};

class Penguin : public NonFlyingBird
{
    ....
};
```

Figure 78: Recognizing that some birds don't fly

267 This behaviour is illustrated by the code in Figure 79 on the next page. Executing this program
268 gives the output:

269

```
Construct Parent
Construct Child
Construct GrandChild
Destroy GrandChild
Destroy Child
Destroy Parent
```

8.1.2 Slicing

A derived class inherits data members from its base class and may define its own data members.
Consequently, a derived class instance d is as large or larger than an instance b of the corresponding
270 base class. There are several consequences: see Figure 80 on page 154.

- The assignment b = d is allowed. Since the data in d does not fit into b, it is not copied: we say that d is ***sliced***. Slicing occurs not only during assignment but also when a derived object is passed or returned by value and the destination is a base class object.
- The assignment d = b is not allowed because it would leave the derived data in d undefined.

```
class Parent
{
public:
    Parent() { cout << "Construct Parent" << endl; }
    ~Parent() { cout << "Destroy Parent" << endl; }
};

class Child : public Parent
{
public:
    Child() { cout << "Construct Child" << endl; }
    ~Child() { cout << "Destroy Child" << endl; }
};

class GrandChild : public Child
{
public:
    GrandChild() { cout << "Construct GrandChild" << endl; }
    ~GrandChild() { cout << "Destroy GrandChild" << endl; }
};


int main()
{
    GrandChild g;
}
```

Figure 79: Constructors and destructors in inheritance hierarchies

- It is usually a mistake to use the base class of a hierarchy as a template argument. Consider: 271

  ```
  vector<Base> bases;
  Derived der( .... );
  bases.push_back(der);
  ```

  Since the argument of `push_back` is passed by value, `der` is sliced; data in `Derived` but not in `Base` is lost. To avoid this problem, use pointers, as in `vector<Base*>`.

Slicing does not occur when we address the object through references or pointers. Suppose that `pb` is a pointer to a base class instance and `pd` is a pointer to a derived class instance. Then:

- The assignment `pb2 = pd` is allowed. After the assignment, `pb2` will point to the ***complete object***, containing base and derived data, but only base class functions and data will be accessible, because of the type of `pb2`. Assignments of this kind are called ***upcasts*** because they go “up” the class hierarchy, from derived class to base class. upcasts

- The assignment `pd = pb1` is not allowed. Allowing it would give the program apparent access to functions and data of the derived class, but those fields do not exist. Assignments of this kind are called ***downcasts***, because they go “down” the class hierarchy. downcasts

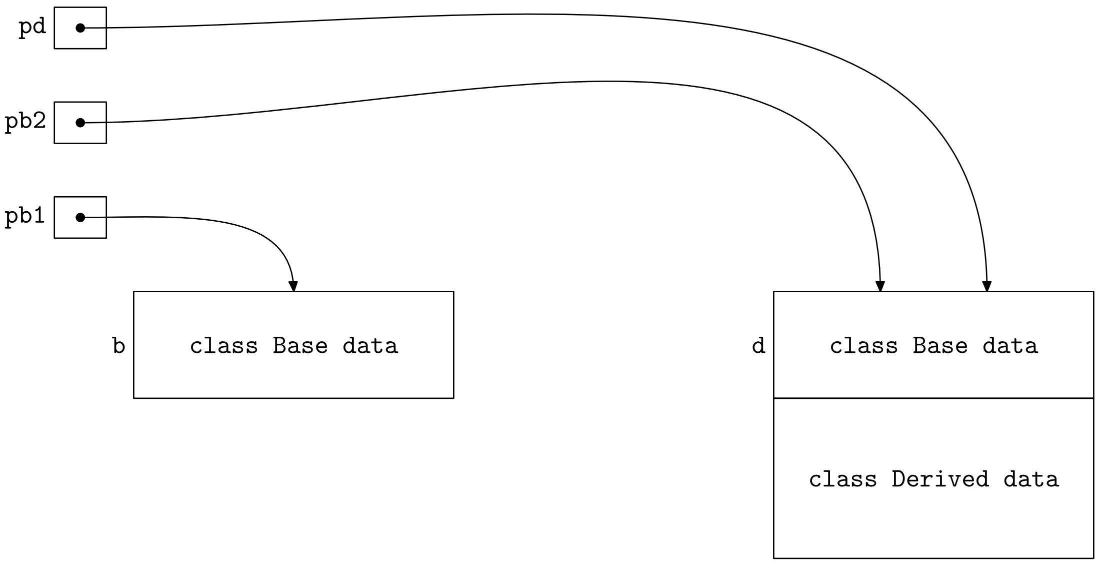


Figure 80: Base and derived objects

- References work in the same way as pointers: whenever we have the address of an object (that is, a reference or a pointer to it), slicing does not occur and dynamic binding works.

For now, the thing to remember is: ***upcast good, downcast bad***. We will discuss these casts in more detail later.

8.2 Designing a base class

Designing a class is difficult because you have to think about all the ways in which the class might be used. Designing a base class is harder still, because you have to think about the people who want to inherit from your class as well as the people who want to use it directly (if it is possible to do so). The differences between a simple class and a potential base class are:

- `virtual` functions
- `protected` attributes

polymorphism

Virtual functions open up the possibility of redefinition in derived classes, leading to polymorphic behaviour. Protected attributes can be accessed by derived classes but not by outsiders. These two features combine to make inheritance useful and manageable.

8.2.1 `virtual` functions

A ***virtual function*** is a function that is defined in a base class and can be redefined in a derived class.[32] Virtual functions are qualified by the keyword `virtual`.

[32] In Java, ***all*** functions are virtual in the C++ sense. Java programmers must remember to write `virtual` when writing C++ programs!

Inheritance polymorphism, which means simply class-dependent behaviour, requires a particular set of circumstances. All of the following conditions must hold for polymorphism to take place: 272

- There must be a base class `B` containing a `virtual` function `f`.
- There must be a class `D`, derived `public` from `B`, that redefines `f`.
- There must be an instance, `d`, of `D`.
- There must be a reference, `&rd`, or a pointer, `*pd`, to the object `d`. These might be defined as:

  ```
  B & rd = d;
  B * pd = &d;
  ```

- The function `f` must be invoked using either a reference or a pointer, as in these statements:

  ```
  rd.f();
  pd->f();
  ```

  Each of these statements will call the redefined function `D::f()`, even though `rd` (`pd`) is a reference (pointer) to the base type. This is ***inheritance polymorphism*** or ***dynamic binding*** (i.e., binding a function name to a function at run-time – as opposed to ***static binding***, where the function to be called is determined at compile-time).

dynamic binding vs. static binding

A virtual function may be ***pure virtual***, meaning that it has no implementation. The body of a pure virtual function is written using the special notation `=0`. A class that has one or more pure virtual functions cannot be instantiated. It may have constructors, but these constructors can be called only by derived classes that provide definitions for the virtual functions.

pure virtual
=0

There are essentially three different kinds of member functions in a base class: normal functions, virtual functions, and pure virtual functions. Base class `Animal` has one example of each kind. 273

```
class Animal
{
public:
    string getName() { .... }
    virtual void move() { /* how to walk */ };
    virtual void communicate() = 0; // no implementation
    ....
};
```

From this declaration, we can infer the following intentions of the designer of the `Animal` class hierarchy. Assume that `pa` is a pointer declared as `Animal*` but actually pointing to an instance of a class derived from `Animal`.

- `getName` is a non-virtual function with an implementation. The idea in this example is that the mechanism for accessing the name of an animal is the same for all derived classes. A non-virtual function should not be redefined in a derived class for reasons explained below (Section 8.2.2 on page 157).

> ***Use a non-virtual function for behaviour that is common to all classes in the hierarchy and should not be overridden in derived classes.***

- Function `move` is virtual and has an implementation. In this example, `Animal::move` defines walking, which is a way of moving shared by many animals, and is a kind of ***default behaviour***. A derived class has the option of either inheriting this method of walking or of redefining `move` in a special way for some other kind of movement. Calling `pa->move()` invokes movement that depends on the particular kind of animal.

Use a virtual function to define default behaviour that may be overridden in derived classes.

- Function `communicate` is a pure virtual function. In this example, the use of a pure virtual function suggests that there is no default behaviour for communication. It corresponds to the fact that animals communicate in many different and unrelated ways (using sound, gesture, smell, touch, etc.). Since class `Animal` has a pure virtual function, it cannot be instantiated: there are no objects of type `Animal`. This corresponds to the real world, where if you see an "animal", it is always a particular kind of animal: aardvark, beaver, cat, dog, elephant, giraffe, hedgehog, etc. Any class for which instances are needed must provide an implementation of `communicate`.

Use a pure virtual function to require behaviour for which there is no reasonable default.

274 There is one function that should ***always*** be virtual in a base class: it is the destructor. If a class is intended as a base, declare:

```
class Base
{
public:
    virtual ~Base() { .... }
    ....
};
```

even if the body of the destructor is empty.

275 276 To see the importance of this, consider the code in Figure 81 on the facing page. Running this program produces the following output:

```
Deleting Base
```

We have constructed an instance of `Derived`, containing data members `x` and `y`, but the run-time system has deleted an instance of `Base`, with data member `x` only. The memory occupied by `y` will almost certainly ***not*** be deallocated – a memory leak!

277 If we change the declaration of class `Base` to

```
class Base
{
public:
    virtual ~Base() { cout << "Deleting Base" << endl; }
    ....
```

```
class Base
{
public:
    ~Base() { cout << "Deleting Base" << endl; }
private:
    int x;
};

class Derived : public Base
{
public:
    ~Derived() { cout << "Deleting Derived" << endl; }
private:
    int y;
};

int main()
{
    Base * pb = new Derived;
    delete pb;
}
```

Figure 81: The danger of a non-virtual destructor

and run the program again, it displays

```
Deleting Derived
Deleting Base
```

By making `Base::~Base` virtual, we have ensured that `delete pb` calls `Derived::~Derived`. This destructor first executes its own code, then invokes the destructor for the base class, and finally deallocates the memory for all of the variables of the derived object.

A base class must have a virtual destructor.

8.2.2 Don't redefine non-virtual functions

C++ allows you to redefine non-virtual functions, as in this example: 278

```
class Animal
{
public:
    Animal() : name("Freddie") {}
    string getName() { return name; }
private:
    string name;
};

class Bison : public Animal
{
public:
    string getName() { return "Grrrrr"; }
};
```

Although it is allowed, redefining non-virtual functions is not a good idea, for several reasons
279 (Meyers 2005, Item 36). Suppose that we define some variables:

```
Bison b;
Animal *pa = &b;
Bison *pb = &b;
```

These definitions give us two pointers, `pa` and `pb`, pointing to the same object, `b`. It is therefore something of a surprise when we execute

```
cout << pa->getName() << endl;
cout << pb->getName() << endl;
```

and obtain

```
Freddie
Grrrrr
```

What has happened here is that, since `Animal::getName` is not `virtual`, the compiler uses the type of the pointer to choose the function. Thus `pa` causes `Animal::getName` to be called and `pb` causes `Bison::getName` to be called.

If `Animal::getName` was a `virtual` function, then the function called would depend on the type of the object, not the pointer. This is more natural behaviour: we expect a bison to answer `"Grrrrr"` whether we refer to it as an animal or as a bison (at least in this example).

More formally, overriding a non-virtual function is a ***violation of the "is-a" relationship***. The purpose of a non-virtual function in a base class is to define behaviour that is invariant over the whole class hierarchy. In the example, to be an `Animal` (that is, to satisfy the predicate "is-a `Animal`"), you must respond to `getName` by returning your name. If you don't do that, you are not an `Animal` and do not belong in the `Animal` hierarchy. There are two possibilities:

- It is essential that bisons have a way of saying `"Grrrrr"`. In this case, the function that does this should not be called `getName`.
- It is correct modelling to say that bisons reply `"Grrrrr"` when asked their name. In this case, `Animal::getName` should be a virtual function.

Never redefine non-virtual functions in derived classes.

8.2.3 Virtual methods in C++11: override and final

280

In C++11, you can use the new `override` keyword to make it explicit that a function is intended to override an *existing* virtual base class version: override

```
void foo() override;
```

If the base class does not have a `virtual` function `foo()`, the compiler will issue an error. This protects you from a number of mistakes, including accidentally hiding a function from a base class (see 'Dominance' in Section 8.3.3).

In C++11, always add* `override` *when you are overriding a* `virtual` *function.

Similar to Java, C++11 now also has a `final` specifier, which indicates that a function cannot be further overridden in derived classes: final

```
void foo() final;
```

Fun fact: `override` and `final` are not ordinary keywords: you can still use variables or functions in your code named `override` or `final` (although that is emphatically not recommended for new code). This works because they are designed as so-called *contextual keywords*, which helps to preserve backwards compatibility with existing code.

C++11 has a number of additional new features related to inheritance that we do not cover in this introduction; see (Prata 2012, Chapter 18) for an overview.

8.2.4 protected attributes

Attributes in the `public` part of a class declaration are accessible to anyone who owns an instance of the object. Attributes in the `private` part of the declaration are accessible only to members and friends of the object.

A class declaration may contain a third part, introduced by the keyword `protected`, for attributes that are accessible to members and friends of classes derived from the class. protected

According to (Stroustrup 1994, page 301), the keyword `protected` was introduced into C++ at the request of Mark Linton, who was writing ***InterViews***, an extensible ***X/Window***[33] toolkit. Five years later, Linton banned the use of `protected` data members in InterViews because they had become "a source of bugs: 'novice users poking where they shouldn't have in ways they ought to have known better than'". Stroustrup goes on to say (Stroustrup 1994, page 302):

[33] See http://en.wikipedia.org/wiki/X_Window_System

In my experience, there have always been alternatives to placing significant amounts of information in a common base class for derived classes to use directly. In fact, one of my concerns about `protected` is exactly that it makes it too easy to use a common base the way one might sloppily have used global data.

Fortunately, you don't have to use protected data in C++; `private` is the default in classes and is usually the better choice. Note that none of these objections are significant for protected member ***functions***. I still consider `protected` a fine way of specifying operations for use in derived classes.

Prefer `private` ***to*** `protected` ***for data members of base classes.***

Use `protected` ***functions to access*** `private` ***base class data.***

8.2.5 Example: A base class for bankers

As an example of base class design, we will take class `Account` from Section 6 (see page 101) and redesign it as a base class. Class `Account` is quite appropriate as a base class, because banks provide many kinds of account, and the various kinds are often represented as a hierarchy of classes in banking software. A number of design decisions are mentioned or implied in the following notes; few of them are cast in stone, and most might be made differently in specific circumstances. 281

282 Figure 82 on the next page shows the new version of class `Account`. The type of `Account::name` has been changed from `char*` to `std::string`, to avoid the memory management problems that we encountered in Section 6 on page 99. Some of the functions, and the class `AccountException`, are not changed, and we do not repeat their descriptions here. The redesign takes into account the following considerations:

- Constructors cannot be virtual, so we leave them unchanged.
- The destructor must be virtual, as explained in Section 8.2.1 on page 154.
- The assignment, `operator=`, cannot be virtual, so we leave it unchanged.
- We provide some accessors (`getName`, `getID`, and `getBalance`) that were not present in the original class but are likely to be useful. They are `public` and non-virtual, since they access private data members and there should never be a need to redefine them. As explained above, these functions express ***invariant properties*** of the account class hierarchy: the meaning of `getBalance` is independent of the type of account we are dealing with. Figure 83 on page 162 shows their definitions. 283
- The definitions of `deposit`, `withdraw`, and `transfer` are not changed. However, it is quite possible that a derived class might need to modify their behaviour; consequently, we declare them to be `virtual`.

```
class Account
{
public:
    Account();
    Account(std::string name, long id, long balance = 0);
    Account(const Account & other);
    virtual ~Account();
    Account & operator=(const Account & other);
    const std::string getName() const;
    long getID() const;
    long getBalance() const;
    virtual void deposit(long amount);
    virtual void withdraw(long amount);
    virtual void transfer(Account & other, long amount);
    friend std::ostream & operator<<(std::ostream & os,
                                     const Account & acc);

protected:
    void setID(long newID);
    void setBalance(long newBalance);

private:
    std::string name;
    long id;
    long balance;
};

bool operator==(const Account & left, const Account & right);
bool operator!=(const Account & left, const Account & right);
```

Figure 82: Class `Account` as a base class

- There are some functions that should be available to derived classes but not to everybody. These functions are declared in the `protected` section of the class declaration. This group consists of the new functions `setID` and `setBalance`, which have the obvious definitions shown in Figure 84 on the next page. 284
- The comparison operators compare accounts by comparing their addresses. Consequently, they do not have to be defined as `friend`s, and their declarations can be moved outside the class declaration.

8.3 Designing a derived class

Designing a derived class is usually easier than designing a base class because there are fewer options. The base class designer has decided what gets inherited and what can be redefined; the derived class designer has to decide:

```
const string Account::getName() const
{
    return name;
}

long Account::getID() const
{
    return id;
}

long Account::getBalance() const
{
    return balance;
}
```

Figure 83: Accessors for class `Account`

```
void Account::setID(long newID)
{
    id = newID;
}

void Account::setBalance(long newBalance)
{
    balance = newBalance;
}
```

Figure 84: Mutators for class `Account`

- which functions to inherit without change
- which functions to redefine
- what new data to introduce
- what functions to introduce to operate on the new data

If there are no new data or functions, there is probably no need for a derived class at all. In a banking application, for example, it would be pointless to introduce new classes for different rates of interest: the interest rate is just a data member of an appropriate base class.

functions that cannot be inherited

Some functions ***cannot be*** inherited and must be defined if the derived class needs them. The functions that are not inherited are:

- constructors
- the assignment operator (`operator=`)
- friends

Derived class constructors can, and usually should, call the base class constructors explicitly, as in this example.[34] Note how the notation for initialization in the constructor of class `Derived` provides a natural way of calling the class `Base` constructor: Figure 85. 285

The ***Liskov Substitution Principle*** provides a helpful guideline for designing derived classes:

> ***Subclasses must be usable through the base class interface without the need for the user to know the difference.***

"Liskov" is Barbara Liskov of MIT, who has made many contributions to object-oriented programming. An application of the principle for class `Account` would be as follows: if we have a pointer `p` to an instance of class `Account` or any of its derived classes, we should be able to call `p->withdraw(500)` without knowing or caring the actual class of the object `*p`.

8.3.1 Example: A derived class for bankers

In this section, we develop a class `SavingsAccount` derived from the base class `Account`.[35] The important features of a savings account are:

- A savings account ***is an*** account. `SavingsAccount` provides all of the functionality of `Account`.
- `SavingsAccount` has a default constructor and a full constructor that allows all data for the account to be initialized.
- `SavingsAccount` provides a copy constructor and an assignment operator.

```
class Base
{
public:
    Base(int n) { .... }
};

class Derived : public Base
{
public:
    Derived(int k, char c) : Base(k), c(c) { .... }
private:
    char c;
};
```

Figure 85: The constructor of a derived class should call the constructor of its base class.

[34] If you do not call a base class constructor explicitly, the compiler will implicitly call the default constructor, as seen in Figure 79 on page 153. Note that this will fail if your class does not have a default constructor.

[35] The savings account is designed to demonstrate features of object-oriented programming. Any resemblance to actual banking practice is entirely coincidental.

- A savings account collects interest. `SavingsAccount` has a data member for the interest rate and a member function for computing interest and adding the interest to the balance.
- Interest is paid only if the balance exceeds a specified minimum. `SavingsAccount` updates the minimum balance when money is withdrawn from the account.
- The insertion `operator<<` shows the interest rate and minimum balance for the `SavingsAccount` in addition to the information shown for an `Account`. 286

Figure 86 shows the class declaration for `SavingsAccount`. The class declaration contains exactly those functions that we wish to define or redefine. Functions that are inherited from the base class are not mentioned in the declaration of the derived class.

The data members of `SavingsAccount` have the following roles:

- `rate` is the interest rate for the account.
- `minimumBalance` is the minimum balance that the client must maintain over each period in order to earn interest.
- `lowestInPeriod` is the actual minimum balance during the period.

Since constructors are ***not*** inherited, we must define constructors for `SavingsAccount`. These constructors can, and should, invoke the base class constructors when necessary.

- 287 The default constructor in Figure 87 on the next page(a) uses the default constructor of class `Account` and sets the new data members to zero.[36]

```
class SavingsAccount : public Account
{
public:
    SavingsAccount();
    SavingsAccount(std::string name, long id, long balance = 0,
        double rate = 0, long minimumBalance = 0);
    SavingsAccount(const SavingsAccount & other);
    SavingsAccount & operator=(const SavingsAccount & other);
    void withdraw(long amount);
    void addInterest();
    friend std::ostream & operator<<(std::ostream & os,
        const SavingsAccount & sacc);
private:
    double rate;
    long minimumBalance;
    long lowestInPeriod;
};
```

Figure 86: Class `SavingsAccount` derived from `Account`

[36] The use of `Account` in the constructor, and `Account::` in other functions, is similar to the use of `super` in `Java`. `Java` can use a single keyword because the parent of a class is unique; C++ allows multiple inheritance and must therefore indicate the base class name explicitly.

As we have seen, it is a good idea to provide a default constructor for any class. It is an even better idea for a class that might be used as a base class because, if the base class does not have a default constructor, then none of the derived classes can have a default constructor.

- The normal constructor in Figure 87(b) allows the caller to specify all of the fields of 288
an account with default values for the `balance`, `rate`, and `minimumBalance`. Like the default constructor, it calls the base class constructor and initializes the new data members separately.
- The copy constructor in Figure 87(c) must call the copy constructor for the base class and 289
then initialize the new data members explicitly. Calling the copy constructor for `Account` with the `SavingsAccount other` is an example of upcasting.

The assignment operator is implemented by using `*this` (implicitly) to call the assignment
operator of the base class, and then assigning the new data members explicitly: see Figure 88 on 290
the following page.

We must update the value of `lowestInPeriod` whenever the balance might get smaller. This
can happen only when money is withdrawn from the account. Figure 89 on the next page shows 291
the new withdrawal function. We override the definition of `Account::withdraw` with a function that updates `lowestInPeriod`.

The new function `addInterest`, which we assume is called periodically, checks that the client has the required minimum balance and, if so, adds interest. Since `balance` is private, `addInterest`

```
SavingsAccount::SavingsAccount()
    : Account(), rate(0), minimumBalance(0), lowestInPeriod(0)
{ }
```

(a) Default Constructor

```
SavingsAccount::SavingsAccount(string name, long id,
      long balance, double rate, long minimumBalance)
    : Account(name, id, balance), rate(rate),
      minimumBalance(minimumBalance), lowestInPeriod(balance)
{ }
```

(b) Normal Constructor

```
SavingsAccount::SavingsAccount(const SavingsAccount & other)
:   Account(other),
    rate(other.rate),
    minimumBalance(other.minimumBalance),
    lowestInPeriod(other.lowestInPeriod)
{ }
```

(c) Copy Constructor

Figure 87: Constructors for class `SavingsAccount`

```
SavingsAccount & SavingsAccount::operator=
                 (const SavingsAccount & other)
{
    Account::operator=(other);
    rate = other.rate;
    minimumBalance = other.minimumBalance;
    lowestInPeriod = other.lowestInPeriod;
    return *this;
}
```

Figure 88: Assignment for class SavingsAccount

```
void SavingsAccount::withdraw(long amount)
{
    Account::withdraw(amount);
    if (lowestInPeriod > getBalance())
        lowestInPeriod = getBalance();
}
```

Figure 89: Withdrawing from a savings account

must call the protected member function Account::setBalance to update the balance. This function also reinitializes lowestInPeriod: see Figure 90. 292

The insertion operator calls Account(sacc) to upcast the SavingsAccount to an Account before inserting it into the stream. It then inserts the new data members into the stream: see Figure 91. 293

If sa1 and sa2 are savings account objects, the expressions sa1 == sa2 and sa1 != sa2 compile and evaluate correctly. This is ***not*** because operator== and operator!= are inherited from Account but because the compiler can convert sa1 and sa2 to Account. 294

Figure 92 on the next page shows a short test program for accounts and savings accounts. When it is run, this program displays: 295

```
void SavingsAccount::addInterest()
{
    if (lowestInPeriod >= minimumBalance)
    {
        long interest = static_cast<long>(rate * getBalance());
        setBalance(getBalance() + interest);
    }
    lowestInPeriod = getBalance();
}
```

Figure 90: Adding interest

```
std::ostream & operator<<(std::ostream & os,
                          const SavingsAccount & sacc)
{
    return os <<
        Account(sacc) <<
        fixed << setprecision(6) << setw(10) << sacc.rate <<
        setw(6) << sacc.minimumBalance;
}
```

Figure 91: Inserting a savings account into a stream

```
Anne Bailey           1234567    1000
Anne Bailey           1234567    1000
Before interest: Charles Daumier    7654321   1500  0.010000  1000
After  interest: Charles Daumier    7654321   1515  0.010000  1000
Before interest: Charles Daumier    7654321    515  0.010000  1000
After  interest: Charles Daumier    7654321    515  0.010000  1000
```

The first two lines show that an account can be constructed and copied. The next two lines show that Charles earns $15 interest because his balance is greater than $1,000. After transferring $1,000 to Anne, his balance drops to $515 and does not earn interest, as shown in the last two lines.

Note that the transfer is performed using a reference to an `Account`, not a `SavingsAccount`. (This is realistic, because the object doing the transferring should not have to know the kind of accounts involved in the transfer.) Since `transfer` is not redefined in `SavingsAccount`, the

```
Account anne("Anne Bailey", 1234567, 1000);
cout << anne << endl;

Account & ra = anne;
cout << ra << endl;

SavingsAccount chas("Charles Daumier",
                  7654321, 1500, 0.01, 1000);
cout << "Before interest: " << chas << endl;
chas.addInterest();
cout << "After  interest: " << chas << endl;

Account *pa = & chas;
pa->transfer(anne, 1000);

cout << "Before interest: " << chas << endl;
chas.addInterest();
cout << "After  interest: " << chas << endl;
```

Figure 92: Testing the banker's hierarchy

function called is `Account::transfer`. When `transfer` called `withdraw`, it uses `this`, which is a pointer to a `SavingsAccount` object. Consequently, `SavingsAccount::withdraw` gets called, and updates the lowest balance.

8.3.2 Additional base class functions

After completing the base class `Account` and the derived class `SavingsAccount`, we notice that there is information that might be helpful to users: does a particular account pay interest? It is easy to add a function to `SavingsAccount`: 296

```
bool paysInterest() { return true; }
```

This is of little use, however, to the owner of a pointer to an `Account` that may or may not be a `SavingsAccount`. If `paysInterest` is to be useful, it must be declared `virtual` in the base class, `Account`:

```
virtual bool paysInterest() { return false; }
```

Thus, by default, an account class does not pay interest. Any class corresponding to an account that does pay interest must redefine `paysInterest` to return `true`.

This seems fairly innocuous: after all, the question 'does it pay interest?' can be asked of any account, and therefore the function `paysInterest` should be defined in the base class. The problem – trivial in this example – is that the concept of 'paying interest' is not otherwise mentioned in class `Account`.

In a hierarchy with many classes, there will be many functions of this kind. The base class will become cluttered with accessors that provide specialized information that is only relevant to particular derived classes. This can become a maintenance problem because, each time one of these functions is added, the base class is changed, requiring recompilation of the entire hierarchy. Unfortunately, there does not seem to be a clean solution to this problem.

8.3.3 Dominance

297 There is a somewhat surprising aspect of overloading a virtual function: A function with the same name as a function in a base class hides it (instead of overriding it), even when they have different signatures. This is called ***dominance***: a derived class dominates all aspects of its base class, including functions.

As a consequence, care must be taken when trying to overload functions from a base class: this is only possible by ***forwarding*** all calls to the corresponding version of the function in the base class. E.g., for the example in Figure 93, the derived class must define a function with the same signature as the base class and forward each function call:

```
void foo(double d) { Base::foo(d); };
```

shadowing

To ensure that you are not accidentally hiding (or *shadowing*) a function from a base class that you intended to override, always use the `override` keyword in new code (C++11 or newer, see Section 8.2.3).

```
class Base
{
public:
   virtual void foo(double);
};

class Derived : public Base
{
public:
   virtual void foo(string);
};

Base *pb = new Derived;
pb->foo("bar");            // calls Derived::foo
pb->foo(3.14);             // error!
pb->Base::foo(3.14);       // ok, calls Base::foo()
```

Figure 93: Dominance

8.4 Conversion and Casting

Casting a value means converting its type. Casting is needed when the type of a value must actually be changed or when the compiler is confused about the type and has to be told what to do.

8.4.1 Implicit Conversion

There are many situations in which the compiler will perform a conversion implicitly. An assignment statement in which the left and right sides have different types may cause an implicit conversion: a "small" value (e.g., `char`) may be implicitly converted to a "large" value (e.g., `int`), or an `int` may be implicitly converted to `double`.

Similar implicit conversions occur when a value of one type is passed as an argument to a function expecting another type.

8.4.2 C-style Casting

The following definition causes a warning message ("possible loss of data"): 298

```
int k = 3.5;
```

If this is really what you want to do, you can eliminate the error message with a C-style cast:

```
int k = (int)3.5;
```

C++ provides an alternative, equivalent form that resembles constructor syntax. It is called a *function-style cast*:

function-style cast

```
int k = int(3.5);
```

8.4.3 Modern C++ Casting

C++ also provides a deliberately ugly syntax for casting. There are four kinds of cast. The first three have the same effect as the old-style casts above; `dynamic_cast` is a new feature that cannot be simulated with the old syntax.

static_cast: Use a `static_cast` when the compiler can check that the conversion is feasible and can generate any required code at compile-time.

299 For example,

```
int i = 5;
int j = 7;
double q1 = i/j;
```

will set `q1` to zero, which may not be the intended result. If you want to obtain 0.714286, use a `static_cast`:

```
double q2 = static_cast<double>(i) / j;
```

For reasons we have discussed previously, the compiler allows conversion from a derived class `D` to a base class `B` (although information may be lost by slicing) but does not allow conversion from a base class `B` to a derived class `D`: 300

```
b = static_cast<B>(d);   // OK
d = static_cast<D>(b);   // compile-time error
```

In other words, casting does not allow you to do anything that you could not do with a simple assignment:

```
b = d;   // OK
d = b;   // compile-time error
```

301 With pointers or references, conversions in both directions are allowed:

```
B *pb = static_cast<B*>(&d);   // OK - upcast
D *pd = static_cast<D*>(&b);   // OK - downcast
```

However, downcasting (converting from base class to derived class) is not recommended: use `dynamic_cast` instead.

dynamic_cast: A dynamic_cast converts only between reference or pointer types. Its main use is for safe downcasting. The run-time system checks that the value to be converted is of the expected type. The target type must be polymorphic.

Suppose that you have a pointer pb of type Base*. You believe that the object pointed to is an instance of a derived class Derived. Then you could use the following code: 302

```
Derived *pd = dynamic_cast<Derived*>(pb);
```

If your belief is correct (*pb is a Derived instance), then pd will be set pointing to it. If you are wrong, pd will be a null pointer.

The target type for a dynamic cast must be polymorphic. For this reason, dynamic casts may not compile even though the corresponding assignment does compile:

```
Base *pb = pd;                       // OK
Base *pb = dynamic_cast<Base*>(pd);  // compile-time error:
                                     //   Base* is not polymorphic
```

References can be cast dynamically:

```
B bb = dynamic_cast<B&>(d);
```

However, if this cast fails (because d is not an instance of a class derived from Base), the run-time system throws the exception std::bad_cast.

const_cast: Use const_cast when you have a const value in a context that requires a non-const value.

you can cast away const!

The program in Figure 94 on the following page does not compile, because the parameter m of function g cannot be converted from const int to int. One way to get around this problem is to change the call to g, in function f, to 303 304

```
g(const_cast<int&>(n));
```

With this change, the program compiles, and running it gives the following output, showing that const_cast really does “cast away const”.

```
g: 6
g: 7
f: 7
```

Next, consider the following test of the functions f and g: 305

```
void g(int & m)
{
   cout << "g: " << m << endl;
   ++m;
   cout << "g: " << m << endl;
}

void f(const int & n)
{
   g(n);
   cout << "f: " << n << endl;
}

int main()
{
   f(6);
}
```

Figure 94: A problem with `const`

```
int main()
{
   int k1 = 6;
   f(k1);
   cout << "main 1: " << k1 << endl << endl;

   const int k2 = 6;
   f(k2);
   cout << "main 2: " << k2 << endl;
   return 0;
}
```

This test gives the following output; note the difference between “`main 1`” and “`main 2`”:

```
g: 6
g: 7
f: 7
main 1: 7

g: 6
g: 7
f: 7
main 2: 6
```

306 You could avoid the `const_cast` by coding `f` like this:

```
void f(const int & n)
```

```
{
   int m = n;
   g(m);
   cout << "f: " << n << endl;
}
```

The output of this program is:

```
g: 6
g: 7
f: 6
main 1: 6

g: 6
g: 7
f: 6
main 1: 6
```

These examples show the dangers of casting away const-ness. In a large application, the programmer writing main (or any client code) might only be able to see the declaration

```
void f(const int & n);
```

and would infer that the argument passed to f cannot change. But, if f uses const_cast, this assumption is invalid (we will later see a more sensible example for using const_cast).

reinterpret_cast: On very rare occasions, you may have to change the interpretation of a bit-string. The following function is intended to create a hash code from an address. It converts the address to an unsigned long and shifts the result right to obtain the high order bits. 307

```
int hash(void *p)
{
   return reinterpret_cast<unsigned long>(p) >> 12;
}
```

However, reinterpret_cast cannot be used for arbitrary type conversions. This attempt to see how a double value is represented by printing it in hexadecimal does not compile. The compiler explains that there is a better way to perform the indicated conversion:

```
cout << hex << reinterpret_cast<unsigned long>(3.14159) << endl;

 casting.cpp(60) : error C2440: 'reinterpret_cast' :
           cannot convert from 'double' to 'unsigned long'
 Conversion is a valid standard conversion, which can be performed
 implicitly or by use of static_cast, C-style cast or function-style cast
```

8.5 Advanced inheritance topics

We covered the concepts that are most important for modern C++ programming: (single) *is-a* inheritance and templates. However, C++ has a number of additional concepts, like multiple inheritance, which are significantly more complex. We only mention some aspects here; should you come across these in a project, it is time to pick up one of the advanced C++ reference books.

8.5.1 Private Inheritance

Inheritance using the `private` keyword (or no keyword at all, as it is the default behaviour)
308 represents a “using” or “is-implemented-by” relationship between classes.

Most importantly, there is ***no*** polymorphism in `private` inheritance. The compiler will never convert an object of a derived class to a base class. Everything inherited from a base class becomes `private` in the derived class.

```
class Image {
public:
    virtual void draw() const;
    ...
};

class Timer {
public:
    explicit Timer(int tickFrequency);
    virtual void onTick() const;
    ...
}

class Image : private Timer {
public:
    virtual void draw() const;
private:
    virtual void onTick() const;
    ...
};

Image * pi = new Image;
pi->draw();                 // ok
pi->onTick()                // error!

Timer pt = new Image;     // error!
```

Figure 95: `private` inheritance

So, when should you use private inheritance? As the "is-implemented-by" relation indicates, it is purely an implementation technique: a class is implemented using an already implemented base class. This is typically not decided during the object-oriented design phase, but rather during implementation.

An example is shown in Figure 95. Suppose you are developing an Image manipulation class and want to add some profiling functionality to better determine the performance of your class. You rummage in your toolchest and come up with a Timer class that does exactly what you need: at a pre-defined interval, it will call the onTick() function. To make it work, you need to overload this function in your Image class. But this is not a *is-a* relationship: an Image is certainly not a Timer, and it should never be possible to use an Image object in place of a Timer object in a program. With private inheritance, you can inherit the onTick function in your Image class and add whatever profiling code you need. A similar example would be the implementation of a stack data structure using an already existing vector class: implementing the stack will be trivial by using the existing vector as a base, but users of stack should never be able to access elements of a stack directly (using []). With private inheritance, the stack can be implemented using the vector class, without its public interface being exposed to the stack users. 309 310

Together with private inheritance, you might also encounter the using declaration for class members, which can be used to "unhide" the public and protected members of the base class to the derived class' users. using

Note that a similar effect to private inheritance can be achieved in most cases with the *Composite* design pattern, which is easier to understand and manage than private inheritance.

For more details on private inheritance, see (Prata 2012, Chapter 14).

Protected Inheritance. Finally, there exists a variant of private inheritance: ***protected inheritance***. With the keyword protected, derived classes gain protected access to members of the base class in an otherwise private inheritance:

```
class Derived :  protected Base
```

As with private inheritance, polymorphism does ***not*** work with protected inheritance.

Templates and Inheritance. Template classes can also make use of inheritance. We will see an example for this later; but we do not cover the design of templates with inheritance in detail in this course.

8.5.2 Multiple Inheritance

C++ allows multiple inheritance, that is, inheriting from more than one base class. Unlike Java, where only multiple *interface* inheritance is possible, C++ allows multiple inheritance of the *implementation*:

```
class Derived :  public Base1, public Base2
```

The consequences are rather complex and we will not discuss them here: If you ever need to develop (or modify) C++ code containing multiple implementation inheritance, you should consult an advanced reference book.

311 312 An issue that becomes important with multiple implementation inheritance is the ambiguity introduced when the same function or member name is inherited from different base classes. In this case, it becomes necessary to explicitly indicate which base class version should be used.

Together with multiple inheritance, you will probably encounter ***virtual inheritance***, specified with the `virtual` keyword. This is important in multiple (implementation) inheritance, indicating that a member is *shareable* across a number of derived classes:

```
   class Derived :  virtual public Base
or class Derived :  public virtual Base
```

The rules for initializing virtual base classes are rather complicated and not as intuitive as for non-virtual bases, and we will not discuss them in this introductory course.

References

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley.

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.

9 Template Programming

9.1 Designing a template class 177
9.2 A Note on Iterators 184
9.3 Templates vs. Inheritance: Tree traversal 187
9.4 The Curiously Recurring Template Pattern 196

9.1 Designing a template class

In this lecture, we discuss the development of a template class with features similar to an STL container. (Koenig and Moo 2000, Chapter 11) provides a very similar presentation. For a comprehensive reference on templates, (Vandevoorde, Josuttis, and Gregor 2017) is recommended.

Our objective is to implement a class `Vec` that behaves in more or less the same way as the STL class `vector`. We will provide memory management, but in a somewhat less sophisticated way than Koenig and Moo's example. We will obtain the class declaration by filling in the blanks of this skeleton: 314

```
template <typename T>
class Vec
{
public:
    // interface
private:
    // hidden data
};
```

The data in the vector will be stored in a dynamic array of the template type, `T`. We need a pointer to the first element of the array and either: (a) the number of elements in the array, or (b) a pointer to one-past-the-last element of the array. Following STL conventions, we will adopt choice (b) and, of course, the user will see our pointers as iterators.

It would be possible to store exactly the number of elements that we need. This can be inefficient, however, if the user inserts elements into the array one at a time. Consequently, we provide a pointer to one-past-the-last element that is actually in use and another pointer to one-past-the-last element that is available for use. The three pointers are 315

1. `data` points to the first element
2. `avail` points to one-past-the-last element in use
3. `limit` points to one-past-the-last element of allocated memory

These three pointers define a ***class invariant*** for class `Vec`:

- `data` points to the first element
- `data` $\le$ `avail` $\le$ `limit`

- The semi-closed interval [`data`, `avail`) contains the allocated data
- 316 The semi-closed interval [`avail`, `limit`) contains the allocated but uninitialized space

Figure 96 shows these pointers for an array in which 8 elements are in use and 13 elements are available altogether.

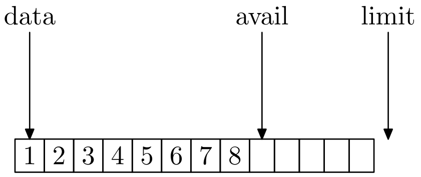


Figure 96: Pointers for class Vec

Following STL conventions, class `Vec` defines several types for its clients. These types hide, for example, the fact that `Vec` uses pointers to implement iterators. (We do not hide this fact because we are ashamed of using pointers, but rather to present a consistent abstraction to our clients.) The types we define are:

- `iterator` for the type of iterators
- `const_iterator` for the type of constant iterators
- `size_type` for the type of the size of a `Vec`
- `value_type` for the type of an element of a `Vec`

For `size_type`, we use `size_t` from namespace `std`.

We will provide three constructors: a default constructor; a constructor that specifies a size (number of elements) and an initial value for each element; and a copy constructor. The default constructor creates an empty vector. We declare the second constructor to be `explicit` to avoid accidental conversion. In the second constructor, the value of the element defaults to `T()`, which implies that the type `T` provided by the user must have a default constructor. Clearly, we will
317 also need a destructor. Figure 97 on the facing page shows the class declaration with the features
318 we have incorporated so far.

319 Figure 98 on page 180 shows implementations for the constructors and destructor. Because this
320 is a template class, these declarations are in the header file `vec.h` after the class declaration. The second constructor is inefficient: for example, if the caller assumes the default value for the second parameter, the constructor `T()` will be called $2n$ times: n times for `new T[n]` and another n times in the `for` statement. (Koenig and Moo 2000) show how to avoid this inefficiency by allocating uninitialized memory, but we will not bother with this improvement.

The copy constructor requires a function that returns the size (number of elements) of the `Vec`
321 object. Since this is also a useful function in general, we make it `public`:

```
size_type size() { return avail - data; }
```

```
template <typename T>
class Vec
{
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef size_t size_type;
    typedef T value_type;

    Vec();
    explicit Vec(size_t n, const T & val = T());
    Vec(const Vec & other);
    ~Vec();

    // rest of interface

private:
    iterator data;
    iterator avail;
    iterator limit;
};
```

Figure 97: Declaration for class Vec: version 1

Since the assignment operator is somewhat similar to the copy constructor, we consider it next. Although they are similar, the difference between assignment and initialization is significant and must not be overlooked. Before assigning to a variable, we must destroy its current value. Also, as we have seen previously, we must provide the correct behaviour for the self-assignment, `x = x`. The implementation of `operator=` shown in Figure 99 on the following page takes care of all this. 322
The constructors and assignment operator introduce duplicated code into the implementation, but we can clean that up later.

9.1.1 Iterators

It is straightforward to define the functions `begin` and `end` that provide clients with iterators pointing to the first and one-past-the-last elements. Two versions are required, one returning an `iterator` and the other returning a `const_iterator`. 323

```
iterator begin() { return data; }
const_iterator begin() const { return data; }

iterator end() { return avail; }
const_iterator end() const { return avail; }
```

Since the operators `!=`, `++`, and `*` are all provided for pointers, these functions provide all that is needed for code such as this loop:

```
template <typename T>
Vec<T>::Vec<T>() : data(0), avail(0), limit(0) {}

template <typename T>
Vec<T>::Vec<T>(size_t n, const T & val)
    : data(new T[n]), avail(data + n), limit(data + n)
{
    for (iterator p = data; p != avail; ++p)
        *p = val;
}

template <typename T>
Vec<T>::Vec<T>(const Vec<T> & other) :
    data(new T[other.size()]),
    avail(data + other.size()),
    limit(avail)
{
    const_iterator q = other.begin();
    for (iterator p = data; p != avail; ++p, ++q)
        *p = *q;
}

template <typename T>
Vec<T>::~Vec<T>()
{
    delete [] data;
}
```

Figure 98: Constructors and destructor for class `Vec`

```
template <typename T>
Vec<T> & Vec<T>::operator=(const Vec<T> & rhs)
{
    if (&rhs != this)
    {
        delete [] data;
        data = new T[rhs.size()];
        avail = data + rhs.size();
        limit = avail;
        const_iterator q = rhs.begin();
        for (iterator p = data; p != avail; ++p, ++q)
            *p = *q;
    }
    return *this;
}
```

Figure 99: Assignment operator for class `Vec`

```
    for (Vec<int>::const_iterator it = v.begin(); it != v.end(); ++it)
        .... *it ....
```

We enable the user to subscript Vecs by implementing operator[]. Two overloads of this operator are required: 324

```
    T & operator[](size_type i) { return data[i]; }
    const T & operator[](size_type i) const { return data[i]; }
```

op[] vs. const op[]

The need for two versions arises as follows. We want a version of operator[] that allows us to use subscripted elements on the left of an assignment, as in a[i] = e. The first version of operator[] does this by returning a reference. But the compiler will not accept this function if it is applied to a const Vec, because the reference makes it possible to change the value of the object. Consequently, we need a second version that returns a const T & and promises not to change the object. Consider the following code:

```
Vec<int> v(5);
v[3] = 5;                    // (1)

const Vec<int> w(v);
n = w[2];                    // (2)
```

The statement labelled (1) fails if the first (non-const) version of operator[] is omitted, because it changes the value of v. The statement labelled (2) fails if the second (const) version of operator[] is omitted, because the compiler needs assurance that the call w[2] does not change the value of w.

Normally, we cannot provide two versions of a function with the same parameter list that differ only in their return type (see Section 7.4 on page 143). In this case, the object (*this) is an implicit first parameter, and the overloads distinguish a const Vec and a non-const Vec.[37]

As with similar STL functions, neither version of operator[] checks to see if the subscript is in range. Such a check could easily be added, perhaps as an assertion.

9.1.2 Expanding Vectors

The next function that we provide is push_back, which appends a new value to the end of the array. There are two cases: if there is space already allocated, we store the new element at the position indicated by avail and then increment avail. In the other case, avail = limit, and we must allocate more space. We will introduce a private function, grow, to find more space. Figure 100 shows the definitions of both functions. 325

326 The algorithm that we use for increasing the size of a Vec has important implications for efficiency. It is usually not possible simply to increase the size of a dynamic array, because the adjacent memory may already be allocated. Consequently, if we have an area of M bytes and we want to

[37] This problem can also be solved using the const_cast seen in Section 8.4: Simply provide the implementation for const, and re-use it by removing the constness using const_cast. This can avoid code duplication when operator[] is more complex. For details, see (Meyers 2005, p.23ff).

```
template<typename T>
void Vec<T>::push_back(const T & val)
{
    if (avail == limit)
        grow();
    *avail = val;
    ++avail;
}

template<typename T>
void Vec<T>::grow()
{
    size_type oldSize = avail - data;
    size_type newSpace = (oldSize == 0) ? 1 : 2 * (limit - data);
    iterator newData = new T[newSpace];
    iterator p = newData;
    for (const_iterator q = data; q != avail; ++q, ++p)
        *p = *q;
    delete [] data;
    data = newData;
    avail = data + oldSize;
    limit = data + newSpace;
}
```

Figure 100: Appending data to a `Vec`

use N bytes, where $M < N$, we must: (1) allocate a new area of N bytes; (2) copy M bytes from the old area to the new area; and (3) delete the old area. This operation requires time $\mathcal{O}(M)$.

If we add one element, or in fact any ***constant*** number of elements each time a `Vec` grows, the performance of `push_back` will be quadratic.[38] For example, if we add one element at each call of `grow`, we will copy 1, then 2, then 3 elements. To get 4 elements into the `Vec`, we will have to copy $1 + 2 + 3 = 6$ elements. To get N elements into the `Vec`, we will have to copy $\frac{1}{2}N(N-1)$ elements.

If, instead, we ***multiply*** the size of the `Vec` by a constant factor, the time spent copying falls to $\mathcal{O}(N \log N)$, which is much better. The implementation of `grow` in Figure 100 uses this technique with a constant factor of 2.

If we measure the time taken to expand an array one element at a time by calls to `push_back`, we will notice that most calls are fast ($\mathcal{O}(1)$) but a few calls (when reallocation is done) are slow ($\mathcal{O}(N)$). We account for this by defining the ***amortized time complexity*** for `push_back`, which is the time for n calls divided by n, as $n \to \infty$. The amortized time complexity for our version of `push_back` is $\mathcal{O}(\log N)$.

[38] This explains the catastrophic performance of Java programs that misuse the `+` operator for strings.

9.1.3 Refactoring

We can improve the clarity of the implementation of Vec by doing some simple refactoring. ***Refactoring*** means rearranging code to improve its maintainability or performance without changing its functionality (we will discuss refactoring in more detail in Section 12.7 on page 276). In this case, we introduce two overloaded versions of a private function called create to manage the creation of new Vecs. We can use this function to simplify the code for the constructors and the assignment operator. The two versions of create are declared in the private part of the class declaration: 327

```
void create(size_type n = 0, const T & val = T());
void create(const_iterator begin, const_iterator end);
```

The first version has two parameters, for the size of the array and the initial values, respectively. Both parameters have default values, so that calling create() yields an empty Vec. The second version also has two parameters that must be iterators defining a range of some other compatible Vec.

The function create is responsible for allocating memory and for initializing the pointers data,
avail, and limit. Figure 101 shows the implementations of both versions. Figure 102 on the 328
following page shows the revised definitions of functions that use create. Finally, Figure 103 on 329
page 185 shows the declaration of class Vec with all the changes that we have discussed. 330

```
template<typename T>                                        331
void Vec<T>::create(size_type n = 0, const T & val = T())   332
{
    data = new T[n];
    avail = data + n;
    limit = avail;
    for (iterator p = data; p != avail; ++p)
        *p = val;
}

template<typename T>
void Vec<T>::create(const_iterator begin, const_iterator end)
{
    size_type size = end - begin;
    data = new T[size];
    avail = data + size;
    limit = avail;
    for (iterator p = data; p != avail; ++p, ++begin)
        *p = *begin;
}
```

Figure 101: Implementation of two overloads of create

```
template <typename T>
Vec<T>::Vec<T>()
{
    create();
}

template <typename T>
Vec<T>::Vec<T>(size_t n, const T & val)
{
    create(n, val);
}

template <typename T>
Vec<T>::Vec<T>(const Vec<T> & other)
{
    create(other.begin(), other.end());
}

template <typename T>
Vec<T> & Vec<T>::operator=(const Vec<T> & rhs)
{
    if (&rhs != this)
    {
        delete [] data;
        create(rhs.begin(), rhs.end());
    }
    return *this;
}
```

Figure 102: Revised definitions of functions that use `create`

9.2 A Note on Iterators

It might seem from the example of class `Vec` that an iterator is just a pointer and that we can use `++`, `==`, and so on, just because these functions are defined for pointers.

In fact, this is not always the case. The STL class `list`, for example, stores data in a linked list
of nodes. An iterator value identifies a node. Incrementing the iterator means moving it to the
333 next node. As Figure 104 on page 186 shows, achieving this requires defining all of the iterator
334 functions, including `*` and `->`.

335 In particular, note that the prefix operators `--it` and `++it` are more efficient than the postfix operators `it--` and `it++`, because the postfix operators use the copy constructor to create the result.

```
template <typename T>
class Vec
{
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef size_t size_type;
    typedef T value_type;
    Vec();
    explicit Vec(size_t n, const T & val = T());
    Vec(const Vec & other);
    ~Vec();

    Vec & operator=(const Vec & rhs);

    T & operator[](size_type i) { return data[i]; }
    const T & operator[](size_type i) const { return data[i]; }

    size_type size() const { return avail - data; }

    iterator begin() { return data; }
    const_iterator begin() const { return data; }

    iterator end() { return avail; }
    const_iterator end() const { return avail; }

    void push_back(const T & val);

private:
    void create(size_type n = 0, const T & val = T());
    void create(const_iterator begin, const_iterator end);
    void grow();
    iterator data;
    iterator avail;
    iterator limit;
};
```

Figure 103: Declaration for class `Vec`: version 2

```
const_reference operator*() const
    {    // return designated value
    return (_Myval(_Ptr));
    }

_Ctptr operator->() const
    {    // return pointer to class object
    return (&**this);
    }

const_iterator& operator++()
    {    // preincrement
    _Ptr = _Nextnode(_Ptr);
    return (*this);
    }

const_iterator operator++(int)
    {    // postincrement
    const_iterator _Tmp = *this;
    ++*this;
    return (_Tmp);
    }

const_iterator& operator--()
    {    // predecrement
    _Ptr = _Prevnode(_Ptr);
    return (*this);
    }

const_iterator operator--(int)
    {    // postdecrement
    const_iterator _Tmp = *this;
    --*this;
    return (_Tmp);
    }

bool operator==(const const_iterator& _Right) const
    {    // test for iterator equality
    return (_Ptr == _Right._Ptr);
    }

bool operator!=(const const_iterator& _Right) const
    {    // test for iterator inequality
    return (!(*this == _Right));
    }
```

Figure 104: An extract from the STL class list

9.3 Templates vs. Inheritance: Tree traversal

In this section, we discuss an example built on the following observation: we can define a general traversal routine for a tree using a generic store, and then specialize the traversal by providing different kinds of store.

For simplicity, we will use binary trees, although the technique can be extended to general trees
and graphs. Figure 105 shows pseudocode for the general traversal. In this pseudocode, we 336
assume that we are given the `root` of a binary tree and an empty `store`. The `store` provides
operations `put` and `get`, and a test `empty`. The order of the traversal is determined by the
behaviour of the store. If the store is a stack with LIFO behaviour, we obtain depth-first search;
if the store is a queue with FIFO behaviour, we obtain breadth-first search. If we have some
knowledge about the nodes, we can obtain smarter traversals by using, for example, a priority
queue as the store.

```
store.put(root);
while (!store.empty())
{
    node = store.get();
    node.visit();
    if (node.right != 0)
        store.put(node.right);
    if (node.left != 0)
        store.put(node.left);
}
```

Figure 105: Pseudocode for binary tree traversal

The precise requirements of the store are as follows:

- `put` must add a new element to the store.
- `empty` must return `true` iff the store is empty and `false` otherwise. `empty` must return `false` if there is any element that has been `put` into the store but has not been retrieved by `get`.
- `get` must return an element from the store unless the store is empty, in which case its behaviour is undefined.

The goal, then, is to implement a traversal function that is passed a store as argument and implements the traversal without knowing precisely how the store behaves. There are two obvious ways in which we might implement a generic traversal function: using inheritance and using templates. Before we show how to do this, we will look at the various features the program needs. All of the classes are intended to have just enough complexity to support this particular application. To avoid confusion, they use simple, pointer-based techniques rather than STL features.

First, we introduce the binary trees that are to be traversed: see Figure 106 on page 189. There 337
is no class for complete trees, just a class for nodes of trees. Each node has an integer key and 338
pointers to its left and right subtrees. Either or both of these pointers, of course, may be null. 339

There are two functions associated for binary trees. One of them overloads `operator<<` and is used as a check to see what is in the tree. The other, `makeTree`, constructs a tree containing all of the keys in a given semi-closed range.

340 For both kinds of store (stack and queue), we will use a single-linked list to store items, as shown in Figure 107 on page 190. The main difference is that a stack accesses only one end of the list (the "top" of the stack) and the queue accesses both ends (the "front" and "back" of the queue). We have assumed that list items are pointers to tree nodes but, of course, we could generalize this by making `ListNode` a template class.

Stores are implemented with an abstract base class with pure virtual functions for the required operations: see Figure 108 on page 190.

341 342 Figure 109 on page 191 shows the declaration of class `Stack`. To save space, the member functions are defined in the body of the class declaration. (This is allowed, and has the side-effect of permitting the compiler to inline them, as we will discuss later.)

Pushing an element onto the stack (`put()`) is easy: we just have to create a new list node and update the pointer to the head of the list, which represents the top of the stack.

Retrieving an element from the stack is slightly harder and care is necessary. We have to get the pointer from the first list node, delete the list node, and update the `top` pointer. This requires creating temporaries, `result` and `tmp`, to store the node to be returned and the new top pointer while we delete the old one. The assertion will detect an attempt to pop an element from an empty stack, although this should never happen: in the traversal pseudocode of Figure 105 on the preceding page, we call `get` only after checking that the store is not empty.

343 344 Figure 110 on page 192 shows the declaration of class `Queue`. The code is somewhat more complicated than the stack code because we have to maintain two pointers, to the front and back of the list.

Points to note about class `Queue`:

- The list pointers run from the front of the queue to the back of the queue.
- New items are inserted at the back of the list. The `back` pointer is updated to point to the new item.
- Items are removed from the front of the list. The `front` pointer is updated to point to the next item in the list.
- If the list is empty, `front` = `back` = 0. When an item is inserted into an empty queue, both `front` and `back` pointers are set pointing to it.
- If removing an item makes the `front` pointer null, then the `back` pointer must be set to null as well, and the queue is then empty.
- The test `empty` examines the `back` pointer, although it could in fact examine either pointer.
- An undocumented class invariant: ***either the front and back pointers are both null, or both are non-null.***

345 At this point, we have all of the code required for the demonstration. Figure 111 on page 193 shows a generic traversal function that is passed a pointer to the abstract base class `Store`. This
346 function can be invoked either by calling

```
// A class for binary trees with integer nodes.
class TreeNode
{
public:
    TreeNode(int key, TreeNode *left, TreeNode *right);
    ~TreeNode();
    int getKey() { return key; }
    TreeNode *getLeft() { return left; }
    TreeNode *getRight() { return right; }
private:
    int key;
    TreeNode *left;
    TreeNode *right;
};

TreeNode::TreeNode(int key, TreeNode *left, TreeNode *right)
    : key(key), left(left), right(right)
{}

TreeNode::~TreeNode()
{
    delete left;
    delete right;
}

// Display a tree using inorder traversal.
ostream & operator<<(ostream & os, TreeNode *ptn)
{
    if (ptn != 0)
    {
       if (ptn->getLeft() != 0)
           os << ptn->getLeft();
       os << ptn->getKey() << ’ ’;
       if (ptn->getRight() != 0)
           os << ptn->getRight();
    }
    return os;
}

// Construct a tree with keys in [first,last).
TreeNode * makeTree(int first, int last)
{
    if (first == last)
        return 0;
    else
    {
        int mid = (first + last) / 2;
        return new TreeNode(mid, makeTree(first, mid), makeTree(mid + 1, last));
    }
}
```

Figure 106: Binary tree: class declaration and related functions

```
class ListNode
{
public:
    ListNode(TreeNode *ptn, ListNode *next) : ptn(ptn), next(next) {}
    TreeNode *ptn;
    ListNode *next;
};
```

Figure 107: Class `ListNode`

```
class Store
{
public:
    virtual void put(TreeNode *ptn) = 0;
    virtual TreeNode *get() = 0;
    virtual bool empty() = 0;
};
```

Figure 108: Abstract base class `Store`

```
traverseUsingInheritance(tree, new Stack);
```

or by calling

```
traverseUsingInheritance(tree, new Queue);
```

The decision as to which kind of store to use is made at ***run-time***. Each call to `put`, `get`, or `empty` is dynamically bound to the corresponding function in either `Stack` or `Queue`. The overhead of dynamic binding is small but could be significant if these operations are performed
347 very often.

Figure 112 on page 193 shows a traversal function that obtains its genericity with a template. The template parameter `StoreType` must be replaced by a class that provides the operations `put`, `get`, and `empty`. It must also provide a default constructor that creates an empty store. This constructor is invoked by the statement

```
StoreType *pst = new StoreType;
```

The inheritance version does not need this statement because it is passed an empty store. Except for this statement, the bodies of the two functions are identical.

348 The template function is invoked either by calling

```
traverseUsingTemplates<Stack>(tree);
```

or by calling

```
class Stack : public Store
{
public:
    Stack() : top(0) {}

    void put(TreeNode *ptn)
    {
        top = new ListNode(ptn, top);
    }

    TreeNode *get()
    {
        assert(top);
        TreeNode *result = top->ptn;
        ListNode *tmp = top->next;
        delete top;
        top = tmp;
        return result;
    }

    bool empty()
    {
        return top == 0;
    }

private:
    ListNode *top;
};
```

Figure 109: Class `Stack`

```
traverseUsingTemplates<Queue>(tree);
```

Note that we must provide the template argument `<Stack>` or `<Queue>` explicitly in calls to `traverseUsingTemplates`, because the compiler cannot infer which kind of store we want from the function argument `tree`. The call 349

```
traverseUsingTemplates(tree);
```

produces the error message (quite intuitively, for a change):

```
error C2783: 'void traverseUsingTemplates(TreeNode *)' :
   could not deduce template argument for 'StoreType'
```

350

351 Figure 113 on page 194 shows a program that tests the generic traversal, using both the inheritance and the template versions, and the output from this program.

Comparison of the solutions:

```
class Queue : public Store
{
public:
    Queue() : front(0), back(0)
    {}

    void put(TreeNode *ptn)
    {
        ListNode *tmp = new ListNode(ptn, 0);
        if (back == 0)
            front = tmp;
        else
            back->next = tmp;
        back = tmp;
    }

    TreeNode *get()
    {
        assert(front);
        TreeNode *result = front->ptn;
        ListNode *tmp = front->next;
        delete front;
        front = tmp;
        if (front == 0)
            back = 0;
        return result;
    }

    bool empty()
    {
        return back == 0;
    }

private:
    ListNode *front;
    ListNode *back;
};
```

Figure 110: Class `Queue`

```
void traverseUsingInheritance(TreeNode *root, Store *pst)
{
    pst->put(root);
    while (!pst->empty())
    {
        TreeNode *ptn = pst->get();
        assert(ptn);
        cout << ptn->getKey() << ' ';
        if (ptn->getRight() != 0)
            pst->put(ptn->getRight());
        if (ptn->getLeft() != 0)
            pst->put(ptn->getLeft());
    }
}
```

Figure 111: Generic traversal using inheritance

```
template<typename StoreType>
void traverseUsingTemplates(TreeNode *root)
{
    StoreType *pst = new StoreType;
    pst->put(root);
    while (!pst->empty())
    {
        TreeNode *ptn = pst->get();
        assert(ptn);
        cout << ptn->getKey() << ' ';
        if (ptn->getRight() != 0)
            pst->put(ptn->getRight());
        if (ptn->getLeft() != 0)
            pst->put(ptn->getLeft());
    }
}
```

Figure 112: Generic traversal using templates

- For inheritance to work, the classes `Stack` and `Queue` must be derived from the same base class. The traversal function has a parameter of type pointer-to-base-class.

  The template solution does not require any relationship between classes `Stack` and `Queue`. The only requirement is that each class has a default constructor that creates an empty store and implements the required member functions.

- The solutions are a trade-off between compile-time overhead and run-time overhead. Dynamic binding has a small performance penalty because virtual functions are called indirectly (through a pointer) rather than directly.

  Templates have no run-time overhead but it does slow down compilation, sometimes

```
int main()
{
    TreeNode *tree = makeTree(0, 10);
    cout << "Initial tree (in order): " << tree << endl;

    cout << endl << "Traverse using inheritance with Stack: ";
    traverseUsingInheritance(tree, new Stack);
    cout << endl << "Traverse using inheritance with Queue: ";
    traverseUsingInheritance(tree, new Queue);
    cout << endl;

    cout << endl << "Traverse using templates with Stack: ";
    traverseUsingTemplates<Stack>(tree);
    cout << endl << "Traverse using templates with Queue: ";
    traverseUsingTemplates<Queue>(tree);
    cout << endl;

    delete tree;
    return 0;
}
```

Output:

```
Initial tree (in order): 0 1 2 3 4 5 6 7 8 9

Traverse using inheritance with Stack: 5 2 1 0 4 3 8 7 6 9
Traverse using inheritance with Queue: 5 8 2 9 7 4 1 6 3 0

Traverse using templates with Stack: 5 2 1 0 4 3 8 7 6 9
Traverse using templates with Queue: 5 8 2 9 7 4 1 6 3 0
```

Figure 113: Test program and output for generic tree traversal

significantly. A small amount of time is spent expanding templates. Much more time is wasted reading additional header files, because the bodies of template functions must be put in header files rather than implementation files.

> *"This is a real problem in practice because it considerably increases the time needed by the compiler to compile significant programs.."* (Vandevoorde, Josuttis, and Gregor 2017)

- As a general rule, early binding provides efficiency and late binding provides flexibility. With both versions, we can use both types of store, as the test program generates. However, 352 we may have to make a ***dynamic*** choice of store, as shown in the following code:

```
Store *ps;
if ( ⟨condition⟩ )
```

```
        ps = new Stack;
    else
        ps = new Queue;
    traverseUsingInheritance(tree, ps);
```

or perhaps like this:

```
    traverseUsingInheritance(tree, ⟨condition⟩ ? new Stack : new Queue);
```

If this kind of code cannot be avoided (i.e., the value of ⟨*condition*⟩ must be evaluated at run-time), there is no reasonable alternative to inheritance. The alternative 353

```
    if ( ⟨condition⟩ )
        traverseUsingTemplates<Stack>(tree);
    else
        traverseUsingTemplates<Queue>(tree);
```

is not particularly attractive. It forces the compiler to generate two expansions of the traversal function, which will occupy lots of memory, and the gain in performance will probably be insignificant.

- In this example, the base class `Store` contains no code. In a larger, more practical, application, code duplicated in derived classes could be moved into the base class.

  Common code cannot be as easily shared in the template version. However, it is possible to use the template approach with a class hierarchy and thereby to obtain the benefits of code sharing. The difference, as before, is just that the inheritance version chooses the derived class at run-time but the template version makes the choice at compile-time.

- In terms of maintenance, there is not a lot to choose between the two solutions. An advantage of the inheritance version is that a new derived class can be added to the hierarchy and linked in to the program without recompiling the traversal function. This might be useful in a large-scale project.

9.3.1 Organizing the code

354

Figure 114 on the next page shows the organization of the various components of the traversal program, not including the `main` function. The header and implementation files for `TreeNode` are conventional. The header file `store.h` contains the forward declaration 355 356

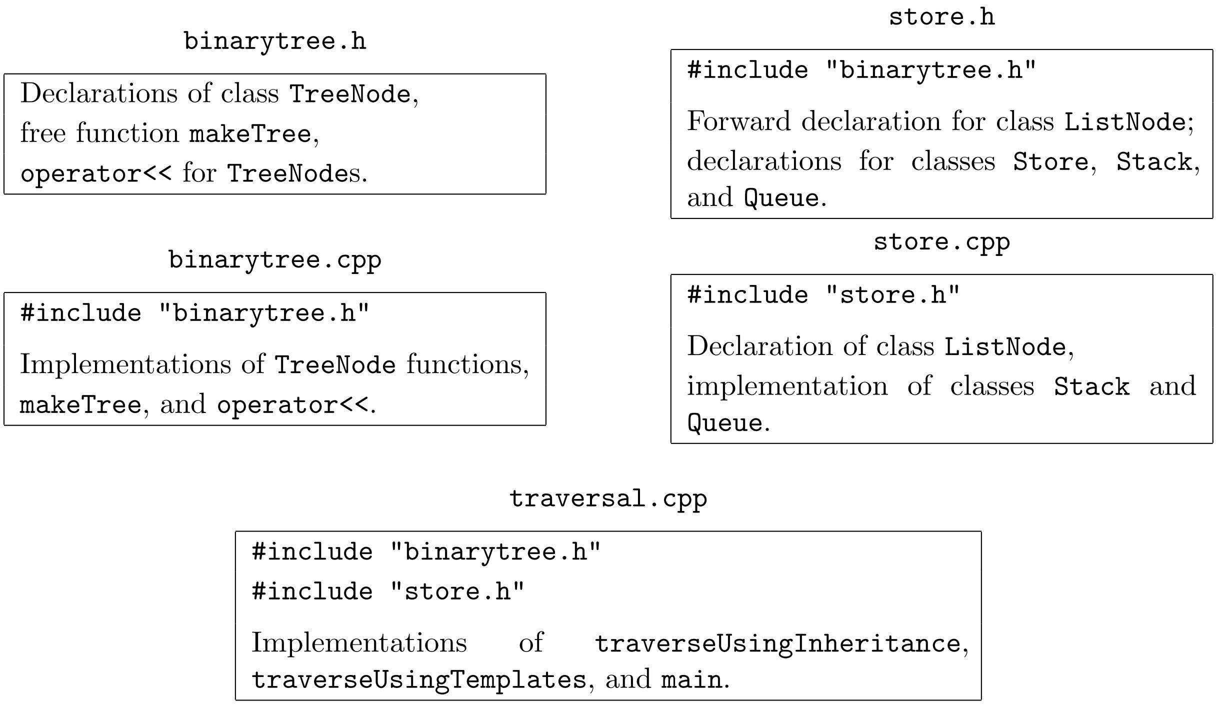
forward declaration

```
class ListNode;
```

It does not need to contain the complete declaration of class `ListNode` because all of the references to `ListNode` are pointers (or references). Provided that the compiler is informed that `ListNode` is a class, it does not need to know how big an instance is (we will discuss forward declarations in more detail later).

The implementation file `store.cpp` contains the full declaration of `ListNode`. Since it is in an implementation file, this declaration is inaccessible to other parts of the program. That its members are all public does not matter, because they are only available to the class `Stack` and `Queue`.

There is no implementation for class `Store` because it is an abstract class that contains only pure virtual functions.



binarytree.h

Declarations of class `TreeNode`, free function `makeTree`, `operator<<` for `TreeNode`s.

binarytree.cpp

`#include "binarytree.h"`

Implementations of `TreeNode` functions, `makeTree`, and `operator<<`.

store.h

`#include "binarytree.h"`

Forward declaration for class `ListNode`; declarations for classes `Store`, `Stack`, and `Queue`.

store.cpp

`#include "store.h"`

Declaration of class `ListNode`, implementation of classes `Stack` and `Queue`.

traversal.cpp

`#include "binarytree.h"`
`#include "store.h"`

Implementations of `traverseUsingInheritance`, `traverseUsingTemplates`, and `main`.

Figure 114: Code organization for traversal program

9.4 The Curiously Recurring Template Pattern

357 The Curiously Recurring Template Pattern (CRTP) was named by James “Cope” Coplien (Coplien 1995). Figure 115 shows the basic idea. At first sight, it is rather mysterious. An example may clarify things.

```
class X : public Base<X>
{
    ....
};
```

Figure 115: The Curiously Recurring Template Pattern

Consider this problem: we would like to provide a way of extending any class that provides `operator<` so that it also provides `operator>`.

358 We might try to apply inheritance in the usual way. We define a base class:

```
class Ordered
{
   virtual bool operator<(const Ordered & other) = 0;
   bool operator>(const Ordered & other)
   {
      return other < *this;
   }
}
```

Then we can inherit this base class to give the desired functionality:

```
class Widget : public Ordered
{
    bool operator<(const Widget & other) { .... }
    ....
};
```

Unfortunately, this doesn't work. The redefinition of `operator<` in class `Widget` is incorrect because the parameter type is different. Moreover, `Ordered::operator>` provides an instance of `Ordered` for comparison, but we want to compare another `Widget`.

```
template <class T>
class Ordered
{
public:
    bool operator>(const T & rhs) const
    {
        const T & self = static_cast<const T &>(*this);
        return rhs < self;
    }
};
```

Figure 116: A class that provides `operator>` using `operator<`

CRTP comes to the rescue! We define the base class `Ordered` as a template class, as shown
in Figure 116. We have to cast from `Ordered` to `T`, but we will see that this doesn't matter in 359
practice. The revised version of class `Widget` inherits from a version of class `Ordered` that is
customized for `Widget`s, using the "recursive" template argument:

```
class Widget : public Ordered<Widget>
{
    bool operator<(const Widget & other) { .... }
    ....
};
```

After templates have been expanded, we have a class that looks something like this (or rather,
would look like this if we could see it): 360

```
class OrderedWidget
{
public:
    bool operator>(const Widget & rhs) const
    {
        const Widget & self = static_cast<const Widget &>(*this);
        return rhs < self;
    }
};
```

We can now see that the cast is perfectly harmless, because it is just casting a `const Widget &` to itself!

We could have solved this problem more easily with templates, as is done in the STL namespace `rel_ops`: see Figure 117. But, as another example of CRTP, consider the problem of keeping track of the number of instances of a class. This is easily done by providing a `static` counter in the class. But suppose we want to encapsulate this behaviour, providing a base class whose children can all keep track of their instances.

```
namespace std
{
  namespace rel_ops
  {
    template <class _Tp>
      inline bool
      operator!=(const _Tp& __x, const _Tp& __y)
      { return !(__x == __y); }

    template <class _Tp>
      inline bool
      operator>(const _Tp& __x, const _Tp& __y)
      { return __y < __x; }

    template <class _Tp>
      inline bool
      operator<=(const _Tp& __x, const _Tp& __y)
      { return !(__y < __x); }

    template <class _Tp>
      inline bool
      operator>=(const _Tp& __x, const _Tp& __y)
      { return !(__x < __y); }

  } // namespace rel_ops
} // namespace std
```

Figure 117: Defining relational operators with templates in the STL

The obvious way of doing this doesn't work. If we put a `static` counter in the base class, we will count *all* instances of child classes. We need a way of providing a separate counter for *each* child class. Once again, CRTP shows the way. Figure 118 shows the base class. 361

```
template<class T>
class Counted
{
public:
   Counted() { ++counter; }
   ~Counted() { --counter; }
   static long num() { return counter; }
private:
   static long counter;
};
```

Figure 118: Base class `Counted`

To make class `T` a "counted" class, it must inherit from `Counted<T>`. Whenever we create such a class, we must also provide an initialized definition for its counter:

```
class Widget : public Counted<Widget> { .... };
long Counted<Widget>::counter = 0;
```

We can create as many counted classes as we need:

```
class Goblet : public Counted<Goblet> { .... };
long Counted<Goblet>::counter = 0;
```

362

Figure 119 on the following page shows a program that tests these ideas. The output shown in Figure 120 on the next page demonstrates that the counts are maintained correctly as objects are created and destroyed. 363

Note that the destructor is ***not*** virtual. We don't have to make it virtual, since CRTP does not use polymorphism, even though it makes use of inheritance. This is a concrete example of a programming technique known as ***Mixin***. In general, this technique allows you to add features to a class without inheriting from a base class. The class that provides the new (typically a single) feature, like counting in the example above, is the Mixin.[39] The CRTP idiom is also sometimes referred to as ***F-bound polymorphism*** or ***Upside-Down Inheritance***. For more on CRTP, see (Vandevoorde, Josuttis, and Gregor 2017, Chapter 21.2). Mixin

References

Coplien, J. O. (1995). Curiously recurring template patterns. *C++ Report 7*(2), 24–27.

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example.* Addison-Wesley.

[39] Java 8 added default method implementations in interfaces, which can be seen as a way of providing Mixins.

```
void report(string title)
{
   cout << title << endl <<
      "Widgets: " << Widget::num() << endl <<
      "Goblets: " << Goblet::num() << endl << endl;
}

int main()
{
   Widget w1;
   Goblet g1;
   Goblet g2;
   {
      Widget w2;
      Widget w3;
      Goblet g3;
      report("Inner:");
   }
   report("Outer:");
   return 0;
}
```

Figure 119: Testing the `Counted` hierarchy

```
Inner:
Widgets: 3
Goblets: 3

Outer:
Widgets: 1
Goblets: 2
```

Figure 120: Output from the counting test of Figure 119

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley.

Vandevoorde, D., N. M. Josuttis, and D. Gregor (2017). *C++ Templates* (2nd ed.). Addison-Wesley. http://tmplbook.com.

10 When things go wrong

10.1 Simple Mechanisms 202

10.2 Exceptions 204

10.3 Exception-safe Coding 211

10.4 Resource Acquisition Is Initialization (RAII) 213

10.5 Coding Policies 218

10.6 Static assertions 220

10.7 Summary 222

In any non-trivial program, there will come a time when things go wrong. There are several ways of responding: 365

1. terminate the program
2. do nothing
3. report an error
4. set an error status flag
5. return a special value
6. trigger an assertion failure
7. throw an exception

In this chapter, we describe each of these mechanisms in more detail, and give criteria for choosing the most appropriate one to use in a given situation.

We begin by characterizing the kind of problems we are considering.

- If the problem is caused by a ***logical inconsistency in the program***, the best way to deal with it is to use assertions. It is not appropriate to use `if` statements, error messages, and the like to deal with simple incorrectness.

  > If the program needs the value of $\sqrt{X}$, and the laws of physics ensure that $X \geq 0$, the program should not have to check X before taking the square root. If X is negative, the program probably has a logical error.

- If the problem is something that can ***reasonably be expected to happen***, the best solution is normal code. For example, an `if` statement to detect the condition followed by an appropriate response.

  > Users cannot be relied upon to enter valid data. Any system component that is reading data from the keyboard should validate the input and complain immediately if anything is wrong. This does not require fancy coding techniques.

Problems that do not fit into the above categories are probably ***serious***, ***rare***, and ***unavoidable***. The following subsections discuss suitable responses to problems of this kind.

10.1 Simple Mechanisms

10.1.1 Error messages

cerr

The easiest response to a problem is an error message, sent to `cout`, or (better) `cerr`, or, in a windowing environment, a dialog or message box.

The streams `cout` and `cerr` behave in essentially the same way. Both send messages to a stream that C++ calls ***standard output***. The difference is that `cout` is buffered and `cerr` is not. When the program crashes, it is more likely to have displayed `cerr` output than `cout` output.

If this method works for your application, use it. However, it is inappropriate for many 366 applications:

- The software may be ***embedded*** in a car, pacemaker, or electric toaster. There is nowhere to send messages to and perhaps no one to respond to them anyway.
- The software is part of a system that must do its job as well as possible without giving up or complaining. For example, an internet router or an airplane fly-by-wire system.
- The user is not someone who can handle the problem. For example, a web browser should not display error messages – although, of course, they sometimes do – because there isn't much typical users can do (see Figure 121[40]).

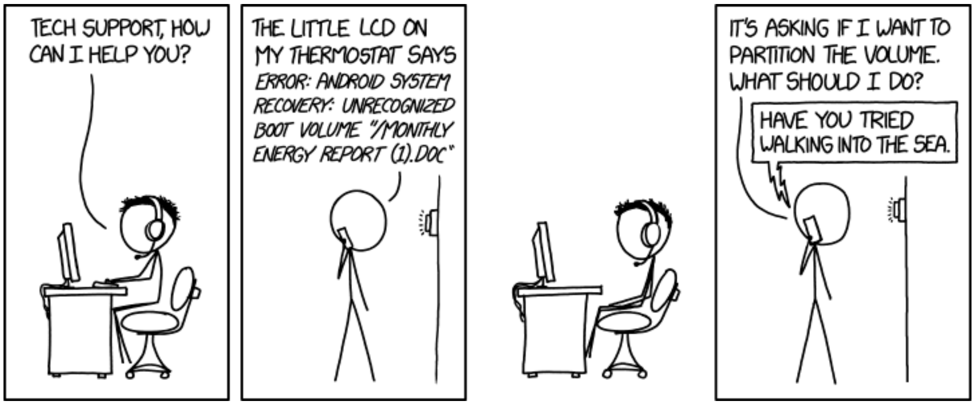

TECH SUPPORT, HOW CAN I HELP YOU?
THE LITTLE LCD ON MY THERMOSTAT SAYS ERROR: ANDROID SYSTEM RECOVERY: UNRECOGNIZED BOOT VOLUME "/MONTHLY ENERGY REPORT (1).DOC"
IT'S ASKING IF I WANT TO PARTITION THE VOLUME. WHAT SHOULD I DO?
HAVE YOU TRIED WALKING INTO THE SEA.

Figure 121: Error messages

10.1.2 Error codes

An effective system used by many software packages, such as UNIX and OpenGL, is to set an error flag and continue in the best feasible way.

- A UNIX library function may set the global variable `errno`. Most library functions also do something else to indicate failure, such as returning `-1` instead of a `char`.
- error handling in OpenGL
  Many OpenGL functions perform according to the following specification: either the

[40] Copyright Randall Munroe, http://xkcd.com/1912/, licensed under a Creative Commons Attribution-NonCommercial 2.5 License

function behaves as expected, or an error occurred and ***there is no other effect***. (The user calls `glGetError` to find out whether an error occurred and what went wrong.)

This system requires some discipline from the programmers using it. They should look at the error status from time to time to make sure that nothing has gone wrong. A common strategy is to check the error status just after doing something that is likely to fail, and otherwise ignore it.

Error codes are ideal for an application in which the consequences of failures are usually not serious, such as a graphics library. They are not secure enough for safety-critical applications.

10.1.3 Special return values

A function can return a funny value to indicate that it has not completed its task successfully. We will call these values ***special return values***. The classic example of this behaviour is the C function `getchar`: you would expect its type to be `char` or even `unsigned char`, but in fact it returns an `int` (thus causing much grief to C beginners). It does this so that it can report end-of-file by returning `-1`.

Special return values are easy to code and require no special language features. They have strong advocates (see http://joelonsoftware.com/items/2003/10/13.html). They also have disadvantages: 367

- There may be no suitable value to return. This is not a problem for `exp` and `log`, whose values are always positive, but how does `tan` report an error (its range is all of the reals from $-\infty$ to ∞)?

- In some cases, it is not feasible to pack all of the information about errors and returned data into a single value. Then we have to add reference parameters to the function to enable it to return all of the information.

- Using the return value for multiple purposes leads to awkward code. In particular, conditions with side-effects are often needed, as in the C idiom of Figure 122.

- Code can become verbose. Instead of writing `y = f(g(h(x)))`, we have to write something like the code in Figure 123 on the next page. 368

- Programmers have a tendency not to check for special return values. This problem is notorious in Unix. 369

```
int ch;
while ( (ch = getchar()) >= 0 )
{
    // process characters
}
// end of file
```

Figure 122: The consequence of ambiguous return values

```
double t1 = h(x);
if (t1 == h_error_value)
    // handle h_error
else
{
    double t2 = g(t1);
    if (t2 == g_error_value)
        // handle g_error
    else
    {
        double y = f(t2);
        if (y == f_error_value)
            // handle f_error
        else
        {
            // carry on
        }
    }
}
```

Figure 123: Catching errors in `y = f(g(h(x)))`

- The caller may not know how to handle the problem. There will be multiple levels of functions, all specially coded to return funny results until someone can deal with the situation appropriately.

10.2 Exceptions

Exceptions provide a form of communication between distant parts of a program. The problem
that exceptions solve is: something has gone wrong but, at this level or in this context, there is
370 no way to fix it. Examples:

- In the middle of complex fluid-flow calculations, the control system software of a nuclear reactor detects that a pressure sensor is sending improbable data.
- A library function that has a simple, natural interface (e.g., `Matrix::invert`) but may nevertheless fail for some inputs (e.g., a singular matrix).

The main advantage of exceptions is that they can transfer control over many levels without
371 cluttering the code. Unfortunately, this is also their main disadvantage. The sequence

```
allocate_my_memory();
delete_my_memory();
```

looks perfectly fine until you realize that `allocate_my_memory` might throw an exception.

In more detail, the advantages of exceptions include: 372

- The thrower does not have to know where the catcher is.
- The catcher does not have to know where the thrower is.
- A catcher can be selective, by only catching exceptions of particular types.
- If no exceptions are thrown, the overhead of the exception handling system is essentially zero.
- The `try` and `throw` statements make it obvious in the code that a problem is being detected and handled.

But exceptions also have some disadvantages: 373

- A programmer who calls a function that might throw an exception, either directly or indirectly (i.e., at some deeply nested place) cannot rely on that function to return. (This is sometimes called "the invisible `goto` problem".)
- Exception handling in a complex system can quickly become unmanageable.
- Unhandled exceptions cause the program to terminate unexpectedly.
- Exceptions violate the "one flow in, one flow out" rule.
- It may be difficult for maintenance programmers to match corresponding `try` and `throw` statements. (The first problem is "who threw this?" and the second problem is "where is it going to end up?".)

The cure for the disadvantages of exceptions is ***disciplined use***: 374

- Use exceptions only when there is no alternative that is better.
- Document exceptions wherever they might affect behaviour in significant ways.
- Learn how to write "exception-safe code".
- For a large project:
  - plan an exception strategy during the initial design phase
  - define and use conventions for throwing and handling exceptions
  - make use of ***exception domains***, which are regions of the program from which no exception can escape

An application should execute correctly in all normal situations with the exception handlers removed.

Exception handling systems are required for large programs in which components are developed independently. If a component cannot perform the task assigned to it, it throws an exception.

When no meaningful action can be performed locally, throw an exception.

10.2.1 Exception handling syntax

375 Exception handling has two parts:

throw

- The statement

  ```
  throw ⟨expression⟩
  ```

  raises, or throws, an exception. The exception is the object obtained by evaluating ⟨*expression*⟩. It can be anything that could be passed to a C++ function – a simple value (`char`, `string`, `int`, etc.), an object, or a reference or pointer to any of these things.

try...catch

- The statement

  ```
  try
  {
      ⟨try-sequence⟩
  }
  catch ( ⟨exc-type⟩ ⟨exc-name⟩ )
  {
      ⟨catch-sequence⟩
  }
  ```

  handles exceptions thrown in ⟨*try-sequence*⟩. More precisely:

  > If a statement in ⟨*try-sequence*⟩ throws an exception `e` whose class is ⟨*exc-type*⟩ or a class derived from ⟨*exc-type*⟩, then the statements ⟨*catch-sequence*⟩ are executed with ⟨*exc-name*⟩ bound to `e`.

10.2.2 Things to know about exceptions

There are several things worth knowing about `try/catch` statements:

- The `throw` statement may not be visible in ⟨*try-sequence*⟩: it may be (and often is) nested inside one or more levels of function call. 376

- The pair ⟨*exc-type*⟩ ⟨*exc-name*⟩ is analogous to the formal parameter of a function. It is as if the `catch` clause is "called" with the exception object as an argument. The semantics of catching an exception with a `catch` clause are exactly the same as receiving an argument with a function.

- Although we mentioned the "class" of the exception object, basic objects such as `int`s and `char`s can be thrown as exceptions.

- 377 If an instance of a derived class is thrown, a handler for a base class will catch it.

  For example, if there is a class hierarchy

  ```
  class Animal { ... };
     class Carnivore : public Animal { ... };
        class Dog : public Carnivore { ... };
  ```

  then the exception in the following sequence is caught:

```
try
{
   throw Dog(...);
}
catch (Animal beast)
{
   ...
}
```

- There may be more than one `catch` clause. The run-time systems matches the exception against each `catch` clause in turn, executing the first one that fits. 378

  Using the same hierarchy as above and a `try` statement

  ```
  try { monster() }
  catch (Dog d)  { .... }
  catch (Carnivore c) { .... }
  catch (Animal a) { .... }
  ```

  then if `monster()` throws a `Dog`, a `Carnivore`, or an `Animal`, the appropriate handler will catch it. But writing

  ```
  try { monster() }
  catch (Animal a) { .... }
  catch (Dog d)  { .... }
  catch (Carnivore c) { .... }
  ```

  is pointless because `Dog`s and `Carnivore`s will never be caught.

- The classes of multiple `catch` clauses do not have to be related by inheritance. 379

- The clause `catch(...)` (that is, you actually write three dots between parentheses) catches all exceptions. If you use this in a multiple `catch` sequence, it should be the last `catch` clause. `catch(...)`

- If there is no `catch` clause that matches the exception, the exception is propagated to the next enclosing `catch` clause.

- In particular, a C++ program behaves ***as if*** it is inside a `try` block: 380

  ```
  try
  {
      main();
  }
  catch (...)
  {
      fail();
  }
  ```

  Thus unhandled exceptions terminate the program, usually with some more or less helpful error message.

- You can throw exceptions by value, or you can create an exception object dynamically (using `new`) and throw a pointer to it. If a pointer is thrown, the handler must delete the exception. 381

```
try
{
   throw X(...);
}
catch (X exc)
{
   ...
}
```

```
try
{
   throw new X(...);
}
catch (X *pexc)
{
   ...
   delete pexc;
}
```

- If a `catch`-block discovers that it cannot handle the exception, it can execute `throw` to pass the same exception to the next level. Note that it is not necessary to mention the exception object again: see Figure 124. 382

- 383 A function can declare the exceptions it throws, like this:

```
void f1() throw();            // doesn't throw anything
void f2() throw(T1);          // may throw a T1
void f3() throw(T2, T3);      // may throw a T2 or a T3
void f4();                    // may throw anything
```

  If a function declaration specifies the exceptions that it throws, the corresponding definition of the function must specify the same exceptions.

- When a virtual function is redefined in a derived class, its throw declarations (if any) must be ***more restricted*** than the base class function. In Figure 125 on the next page, the declarations of `f` and `h` in class `Derived` are legal, because the exceptions that can be thrown by the derived functions are a subset of the base class exceptions. The declaration of `g` in class `Derived` is not allowed. 384

- It is possible to throw an exception instead of executing a `return` statement. If you are searching a tree, for example, the eventual `return` must back out through all the recursive instantiations of the function. Knowing this, a smart (?) programmer might write `throw` rather than `return`. This is not usually a good idea: 385

```
try { .... }
catch (Exception e)
{
    if (understand(e))
        // process e
    else
        throw;
}
```

Figure 124: Rethrowing

```
class Base
{
public:
   virtual void f();
   virtual void g() throw(X);
   virtual void h() throw(X, Y);
};

class Derived : public Base
{
   void f() throw(X);
   void g() throw(X, Y); // Illegal!
   void h() throw(X);
};
```

Figure 125: Inheriting exception specifications

- – The caller must write `throw/catch` instead of a simple call.
- – Since the exception handler must unwind the stack (see Section 10.2.4 on the following page below), the time saved is likely to be small.
- – A mechanism designed for one purpose is being used for another. That C++ programmers love to do this does not make it good software engineering practice.

10.2.3 You didn't expect that

When you declare the exceptions a function can throw, this is ***not*** checked at compile-time: the code in Figure 126 will compile without errors. What happens in this case? When encountering an unexpected exception at run-time, the program will invoke the pre-defined function `unexpected()`.[41] 386

```
void paintball throw(green, blue)
{
   ...
   throw red;
}
```

Figure 126: Throwing unexpected exceptions

unexpected()

By default, `unexpected()` will simply call `terminate()`, which will end your program by calling `abort()`. But you can re-define `unexpected` by defining your own function, e.g., `Ununexpected()`, and then setting this function in your code as the one to call through `set_unexpected()` – see Figure 127 on the following page. 387

[41] No, I'm not making this up. Sorry, it's too late to quit this course!

```
#include <exception>
using namespace std;

void Ununexpected()
{
    cerr << "Can't fool me!";
    throw;
}

main ()
{
    set_unexpected( Ununexpected );
    ...
}
```

Figure 127: Expecting the unexpected

In this code, `throw` will cause the original exception to be rethrown; but you can also throw a different exception, e.g., `std::bad_exception`.

It's perhaps not a surprise that C++11 deprecates exception specifications, so when writing new code, you should not bother with them. However, you might encounter them in existing code, so
388 always be prepared for the unexpected!

Exception Specifications in C++11

Exception specifications didn't work out as, well, expected (one area where Java learned from language design mistakes). Since C++11 they are now deprecated, which means they should not be used for new code.
However, C++11 adds one new feature, the `noexcept` keyword, which indicates that a function is not allowed to throw an exception:

```
int foo() noexcept;
```

There is also a `noexcept` operator that you can use to check if a function is not throwing exceptions, like in `noexcept(foo())`, returning a `bool`.

10.2.4 How exceptions work

It is useful to have an approximate understanding of exceptions. They work roughly as follows:

exception frame

- When the run-time system reaches `try`, it pushes an ***exception frame*** onto the stack.[42] This frame contains descriptors for each `catch` clause.
- If the `try`-block executes normally, the exception frame is popped from the stack.

[42] An implementation may use a separate stack for exceptions. This stack will contain pointers to the main stack.

- If code in the `try`-block throws an exception, there will generally be a number of stack frames on top of the exception frame. The run-time system moves down the stack, popping these frames and performing any clean-up actions required (e.g., calling destructors), until it reaches the exception frame. This is called ***unwinding the stack***. stack unwinding

- The run-time system inspects the `catch` descriptors until it finds a match. It then passes the exception to the `catch` block and gives control to the `catch` block. If no match is found, the run-time system will continue to unwind the stack as before until it finds the next exception frame.

- When the `catch` block terminates, it passes control to the statement following the entire `try-catch` sequence.

10.3 Exception-safe Coding

Exception-safe coding is an important topic for C++ programmers, but a rather large one for this course. Herb Sutter (Sutter 2005) devotes 44 pages to it. The basic rules are: 389

- Write each function to be ***exception safe***, meaning that it works properly when the functions it calls raise exceptions. exception safe

- Write each function to be ***exception neutral***, meaning that any exceptions that the function cannot safely handle are propagated to the caller. exception neutral

As an example, the constructor of the class `Vec`, discussed in Section 9.1 on page 177, is exception-safe: see Figure 128. The only action that may cause an exception to be thrown is `new`, because the default allocator throws an exception when there is no memory available.[43] The constructor does not handle this exception but, by default, passes it on to the caller. Therefore the constructor is ***exception neutral***. 390

The remaining danger is that the constructor does not work properly when the exception is raised. For example, it might happen that the pointers are set even though no data has been allocated. This doesn't matter in practice, because ***a constructor that raises an exception is assumed to have failed completely***. There is no object, and so values of the attributes of the object are irrelevant.

So far so good. Now consider a case where there ***is*** a problem. The code in Figure 129 is 391

```
template <typename T>
Vec<T>::Vec<T>(size_t n, const T & val)
    : data(new T[n]), avail(data + n), limit(data + n)
{
    for (iterator p = data; p != avail; ++p)
        *p = val;
}
```

Figure 128: Constructor for class `Vec`

[43] Some older implementations still return a `null` pointer in this case, which was the default behaviour before exceptions were added to C++.

```
template<class T>
T Stack<T>::pop()
{
    if (vused_ == 0)
    {
        throw "Popping empty stack";
    }
    else
    {
        T result = v_[used_ - 1];
        --vused_;
        return result;
    }
}
```

Figure 129: Popping a stack: version 1

```
template<class T>
void Stack<T>::pop(T & result)
{
    if (vused_ == 0)
    {
        throw "Popping empty stack";
    }
    else
    {
        result = v_[used_ - 1];
        --vused_;
    }
}
```

Figure 130: Popping a stack: version 2

supposed to implement "popping" a stack (Sutter 2005, page 34). The subtle problem with this
code is that `return result` requires a copy operation that may fail and throw an exception. If
it does, the state of the stack has been changed (`--vused_`) but the popped value has been lost.
392 The code in Figure 130 avoids this problem.

The STL is exception-safe. In order to avoid problems like these, it provides ***two*** functions for
393 popping a stack:

`void stack::pop()` removes an element from the stack

`const T & stack::top() const` returns the top element of the stack

Another rule that is followed by and assumed by the STL is: ***destructors do not throw exceptions***.

10.4 Resource Acquisition Is Initialization (RAII)

Generalizing the ideas of the previous section leads to an important principle of C++ coding: ***Resource Acquisition Is Initialization*** – RAII (one of the more bizarre abbreviations, even by C++ standards).

The basic idea is this: Since we need to ensure proper management of resources (allocating/deallocating memory, opening/closing files, etc.), we use *objects* to manage these resources for us. In doing so, we can rely on the fact that constructors, destructors, and exceptions are carefully designed to work together.[44]

A properly-written constructor either ***succeeds*** and creates a complete object or it ***fails*** and leaves no wreckage in its wake. However control leaves a block (by "falling through", executing `return`, or by raising an exception), all objects constructed within that block will be destroyed. 394

Figure 131 on the following page provides a simple example. The function does not close the output file because there is nowhere to put the statement `ofs.close()` in such a way that it is guaranteed to work correctly. (Note, for example, that the exception might be raised because the `ofstream` constructor failed and the report could not be written for this reason.)

However, ***it doesn't matter*** because the run-time system will ensure that `ofs` is destroyed (and therefore closed) whether the exception is raised or not. 395

Figure 132 on the next page is a bit more complicated: it demonstrates the use of RAII in a constructor. The constructor must never return having acquired the `File` but not the `Lock`. The problem is that the constructors for `File` and `Lock` may fail, leaving no object constructed and throwing an exception. The implementation ensures that this code is safe. For example, if the `File` constructor succeeds but the `Lock` constructor fails, the `File` will be destroyed before `Process` terminates.

10.4.1 Autopointers

Consider the following code: 396

```
void f()
{
    Gizmo *pg(new Gizmo);
    .... // do some stuff
    delete pg;
}
```

There are two potential problems with the function `f`:

1. If we forget to include the `delete` statement, the program has a memory leak.

2. It would actually be pretty stupid to forget to write `delete`, but if "`do some stuff`" throws an exception, the program has a memory leak anyway.

```
void report()
{
   ofstream ofs("report.txt");
   try
   {
      // generate report
   }
   catch (string reason)
   {
      cerr << "Report generation failed because " << reason << ".\n";
   }
}
```

Figure 131: Generating a report

```
class Process
{
   public:
      Process(string a, string b) : fs(a), lk(b) {}
   private:
      File fs;
      Lock lk;
};
```

Figure 132: Acquiring a file and a lock

auto_ptr

The standard template class `auto_ptr` uses RAII to avoid problems of this kind. Previously, we have seen how `new` allocates and `delete` deallocates memory, and that these operations must be carefully paired. Autopointers simplify programming by hiding the memory management required for pointers. 397

Figure 133 on the facing page illustrates an attempt to buy a car with pointers. It is obviously wrong, because the `new`s do not match the `delete`s. Even if the caller remembers to `delete` the pointer returned by the function, one of the objects allocated within the function will not be `delete`d, and there will be a memory leak. 398

Figure 134 on page 216 shows the same function implemented with autopointers. It works with autopointers for the following reasons:

1. After an autopointer assignment `p = q`, `p` points to the object and `q` is (effectively) null. In general, autopointers do not allow a situation in which two pointers point to the same object.

   After the `if` statement in the function `buyCar`, `myCar` will be a non-null pointer and one of `bug` and `yug` will be null.

2. When the autopointer copy constructor is used, the same rule applies.

[44] The result is to achieve something like `finally` in Java.

```
Car *buyCar(long balance)
{
   Car *bug(new Bugatti("Veyron"));
   Car *yug(new Yugo("Cabrio"));
   Car *myCar;
   if (balance > 1250000)
      myCar = bug;
   else
      myCar = yug;
   try
   {
      myCar->drive();
   }
   catch (string crash)
   {
      cout << crash << endl;
      myCar = 0;
   }
   return myCar;
}
```

Figure 133: Buying a car with pointers

After `buyCar` returns, `myCar` is null, and the pointer to the purchased is owned by the caller.

3. The destructor of an autopointer deletes a non-null pointer and does nothing for a null pointer.

   When `buyCar` returns, there is one non-null pointer, pointing to the car that was not sold; that object is deleted.

   The caller receives a pointer to a car and, when that pointer goes out scope, the car is destroyed.[45]

An autopointer behaves in a way that is very similar to a pointer: we can use `*` or `->` to dereference it, the increment (`++`) and decrement (`--`) operators work, and so on. The difference is that the pointer target is automatically deleted at the end of the scope or when the stack is unwound during exception handling.

Autopointers have some special properties that make them different from ordinary pointers. These properties are necessary to avoid memory problems when autopointers are copied or passed around. The basic idea is that an autopointer has an ***owner*** who is responsible for deleting the object pointed to. In the following explanations, we assume that `pt` has type `auto_ptr<T>`.

- `*pt` is the object pointed to by the autopointer. 399
- `pt->m` provides access to members of the class of `*pt`, as usual.

[45] And who would want to destroy a Veyron?

```
#include <memory>

auto_ptr<Car> buyCar()
{
   auto_ptr<Car> bug(new Bugatti("Veyron"));
   auto_ptr<Car> yug(new Yugo("Cabrio"));
   auto_ptr<Car> myCar;
   if (balance > 12500000)
      myCar = bug;
   else
      myCar = yug;
   try
   {
      myCar->drive();
   }
   catch (string crash)
   {
      cout << crash << endl;
      myCar.reset(0);
   }
   return myCar;
}
```

Figure 134: Buying a car with autopointers

- `pt.get()` returns the private variable `pt.myptr` which is an ordinary pointer of type `T*` used by the object `pt` to accomplish its magic.

  Note: The '.' indicates that we are calling a method of class `auto_ptr`, not class `T*`.

- `pt.release()` also returns `myptr` but then sets its value to null. The caller has ***taken ownership*** of the autopointer and is now responsible for releasing it.

400
- `pt.reset(newptr)` deletes `myptr` and replaces it by the new pointer `newptr`.

- `pt.reset()` deletes `myptr` and sets it to 0.

- The assignment operator transfers ownership from one autopointer to another. Specifically, the assignment `pt1 = pt2` has an effect equivalent to

  ```
  delete pt1;
  pt1 = pt2.release();
  ```

The ownership property of autopointers has some interesting consequences. When the program
401 in Figure 135 on the facing page is executed, it displays:

402

403

```
Woof!
Walking around the park ...
Aaargh!
Now we're back at home
```

```
#include <memory>
#include <iostream>

using namespace std;

class Dog
{
public:
    Dog() { cout << "Woof!" << endl; }
    ˜Dog() { cout << "Aaargh!" << endl; }
};

typedef auto_ptr<Dog> PD;

void walk(PD pd)
{
    cout << "Walking around the park ..." << endl;
}

void main()
{
    PD pd(new Dog);
    walk(pd);
    cout << "Now we’re back at home" << endl;
}
```

Figure 135: Copying an autopointer

Note the order of events carefully. The autopointer `pd` is set pointing to a newly constructed `Dog`. The autopointer is then passed by value to the function `walk`, which thereby assumes ownership of the `Dog`. The `Dog` walks around the park until the end of the scope. When `walk` returns, `pd` no longer points to the `Dog`, which expired at the end of its walk.

Warning! The ownership behaviour of autopointers is incompatible with the expectations of STL containers. Declarations such as this should never be used: 404

```
vector<auto_ptr<double> > vec;
```

For example, a sorting algorithm may select one element of a vector as a pivot and make a local copy of it. The pivot is then used to partition the array. If the vector contains autopointers, copying an element will invalidate the element! Consequently, sorting will delete the pivot and the sorted array will contain a null pointer.

Don’t store autopointers in STL containers.

More information on autopointers can be found in (Weiss 2004, Chapter 8). (Koenig and Moo 2000, Chapter 14) discusses the implementation of a generic handle class that behaves similar (but is not identical) to autopointers.

10.4.2 Autopointers in C++11

Realizing that a single "smart" pointer type is not suitable for all circumstances, C++11 deprecates `auto_ptr`, replacing it with a number of new smart pointers with different semantics: `unique_ptr`, `shared_ptr`, and `weak_ptr`. 405

`shared_ptr` has a shared ownership semantics: Instead of transferring ownership on assignment, like `auto_ptr` does, it increments a *reference count*. Calling a destructor decrements this count, until it reaches zero, at which point the referenced object is destructed. The `shared_ptr` works correctly with STL containers.

`unique_ptr` is the replacement for `auto_ptr`. It works in almost the same way, but has a few more restrictions. In particular, you can only assign one `unique_ptr` to another if the rhs of the assignment is a temporary variable: 406

```
unique_ptr<string> up1(new string("Hello"));
unique_ptr<string> up2(new string("World"));
up1 = up2; // not allowed!
up1 = move(up2);
```

std::move

Here, `move` is `std::move`, which allows to move ownership from one resource to another. It is another new C++11 feature: for more details, look up ***move constructors*** and ***xvalues*** in a reference book.

`weak_ptr` is a shared, non-owning reference to an object and used in conjunction with `shared_ptr`.

You can find more information in (Prata 2012, Chapter 16). Scott Meyers discusses the new smart pointers in detail (Meyers 2014).

10.5 Coding Policies

407 Suppose that the function

```
      Matrix rotation(const Vector & u, const Vector & v);
```

returns a `Matrix` corresponding to a rotation that takes vector `u` to vector `v`. In other words, the function returns a matrix M such that $M\mathbf{u} = \mathbf{v}$.

The matrix computed by this function is correct only if `u` and `v` are unit vectors: $|\mathbf{u}| = |\mathbf{v}| = 1$. The problem is what to do if `u` or `v` is not a unit vector.

The following sections suggest some answers.

10.5.1 Cowboy coding

We could just do nothing and hope that the vectors we get are normalized. (Or say that callers who do not provide unit vectors deserve what they get.)

However, this might not be an acceptable alternative because the consequences could be serious. (Perhaps the rotation will be used to rotate an aircraft into the correct orientation for landing.)

10.5.2 Pessimistic coding

Assume the worst: normalize `u` and `v` before using them. This is quite an expensive (i.e., time consuming) operation and is a waste of time if callers provide unit vectors anyway.

More importantly, there is not usually an obvious way to correct invalid parameters, so this cannot be considered as a general solution. 408

10.5.3 Defensive coding

Inspect the vectors and take some action if they are not unit vectors: 409

```
Matrix rotation(const Vector & u, const Vector & v)
{
    if (fabs(u.length() - 1) > TOL || fabs(v.length() - 1) > TOL)
        // error handling
    else
        // compute rotation matrix
}
```

This approach is called ***don't trust the caller*** or, more politely, ***defensive coding***. We will discuss possible ways of handling errors later.

10.5.4 Contractual coding

Specify that the vectors passed to the function must be unit vectors and that the behaviour of the function is undefined if they are not. The usual way to do this is to write an assertion or comment called a ***precondition*** in the code: 410

```
/**
 * Construct the matrix that rotates one vector to another.
 * \param u is a vector representing an initial orientation.
 * \param v is a vector representing the final orientation.
 * The matrix, applied to \a u, will yield \a v.
 * \pre The vectors \a u and \a v must be unit vectors.
 */
Matrix rotation(const Vector & u, const Vector & v);
```

The precondition passes the responsibility to the caller. Effectively, the comments define a ***contract*** between the caller and the function: if you give me unit vectors, I will give you a rotation matrix; if you give me anything else, I guarantee nothing. This form of coding is called ***design by contract*** or DBC; it was pioneered by Bertrand Meyer (Meyer 1997) but has been recommended by other people as well (see, for example, (Hunt and Thomas 2000, pages 109–119)). The code for the function does not check the length of the vectors u and v except, perhaps, during development.

Most languages, including C++, do not provide direct support for DBC. However, we can use assertions (Section 2.7 on page 34) as an implementation. Using the matrix rotation example:

```
/** \pre The vectors \a u and \a v must be unit vectors. */
Matrix rotation(const Vector & u, const Vector & v)
{
    assert(fabs(u.length() - 1 <= TOL);
    assert(fabs(v.length() - 1 <= TOL);
    ...
}
```

There are advocates and opposers of both DBC and defensive coding. There is a general trend away from defensive coding (the "traditional" approach) and towards DBC. For example, the home page for the Java Modeling Language[46] says that JML is a "design by contract" language
411 for Java. Here are some arguments for and against each approach.

Efficiency: DBC has no run-time overhead. Defensive coding executes tests which almost always fail (e.g., most vectors passed to `rotation` are in fact unit vectors).

Safety: DBC relies on programmers to respect contracts. Defensive coding ensures that malingerers will be caught. (However, if DBC is implemented with assertions, contract violations will be caught whenever assertions are enabled.)

Clarity: DBC requires maintainers to read comments. With defensive coding, the checks are in the code.

Completeness: There are situations that DBC cannot handle, such as the validity of user input. In these situations, we can – and must – code defensively.

DBC and defensive programming are not mutually exclusive: you can use both. The best policy is to decide which fits the needs of the particular application better and then use it.

10.6 Static assertions

The `assert` macro is useful, but it doesn't do anything until run-time. It is sometimes useful to check a condition at compile-time, causing a compile-time error if the condition is false. This
static assertions is known as ***static assertion*** checking. It is even more useful if the compiler's error message
412 describes the problem. For example, the following program does not compile:

[46] http://www.cs.iastate.edu/~leavens/JML/

```
#define StaticAssert ....

int main()
{
    StaticAssert(sizeof(char) < sizeof(int),
        char_is_smaller_than_int);
    StaticAssert(sizeof(int) >= 8,
        int_has_at_least_eight_bytes);
}
```

The second argument of `StaticAssert` must be a valid C++ variable name but the variable does not have to be defined. The error messages from the compiler are: 413

```
assert.cpp(11): error C2466:
   cannot allocate an array of constant size 0
assert.cpp(11): error C2087:
   'ERROR_int_has_at_least_eight_bytes' : missing subscript
```

The second message tells the programmer that the assumption “`int`s have at least eight bytes” is incorrect.

The LOKI library[47] defines a `StaticAssert` in various ways for different compilers to ensure that the error message is useful. A simple way to achieve the effect can be seen here: 414

```
template<int> struct StaticAssertion;
template<> struct StaticAssertion<true> { };

#define STATIC_ASSERT(expression, message) \
    { StaticAssertion<((expression) != 0)> ASSERTION_##message; }

int main() {
  // static assertion using above macro
  STATIC_ASSERT( sizeof(int) >= 4,
                 size_of_int_less_than_4_bytes);

  // static assertion in C++11
  static_assert( sizeof(int) >= 4,
                 "Size of int is less than 4 bytes.");
  return 0;
}
```

The BOOST library also offers a static assertion check.

[47] LOKI, http://loki-lib.sourceforge.net/

`static_assert` in C++11

Since static assertions have become widely used, C++11 includes static assertion checking built-in via a `static_assert` command that you can use like this:

```
static_assert(sizeof(int) >= 4, "Size of int is less than 4 bytes.");
```

Static assertions are particularly useful for template programming, in order to check the computations performed at compile-time.

10.7 Summary

- Decide on a ***failure management policy*** for your application and follow it consistently.
- Use simple solutions for simple problems. Error codes might be good enough for a simple system.
- Distinguish between events that are ***possible*** (although they might be unwanted, unlikely, etc.) and events that are ***impossible*** (if the program is correct).

  Possible events should be handled with messages, codes, or exceptions. Impossible events should be "handled" with assertions or DBC.

References

Hunt, A. and D. Thomas (2000). *The Pragmatic Programmer*. Addison-Wesley.

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.

Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.). Professional Technical Reference. Prentice Hall.

Meyers, S. (2014). *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* (1st ed.). O'Reilly.

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Sutter, H. (2005). *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions*. C++ In-Depth Series. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.

11 Parallel and Distributed Programming

11.1 Foundations 223
11.2 The POSIX Thread Library 225
11.3 Multithreading in C++11 232
11.4 Distributed Computing 247

Before C++11, there was no concept of parallelism or multithreading within the language or the STL. Hence, programmers had to rely on additional (platform-specific) libraries for parallel programming. We look at one of the most commonly used libraries, POSIX threads, in Section 11.2 on page 225. C++11 finally introduced multithreading into the language standard. We cover some core features in Section 11.3 on page 232. However, within this course we can only provide a brief introduction to multithreaded programming; for further details, such as the design of thread-safe data structures and interfaces, as well as implementation and debugging details, you have to consult a more comprehensive reference, such as (Williams 2012).

Multithreading is not the only way to write parallel programs. Especially in web-scale data processing, software is now commonly run on clusters of hundreds or even thousand of computers. We look at one popular way for developing programs that can make use of such clusters, called MapReduce, in Section 11.4 on page 247.

A Warning. All code shown in this lecture is highly dependent on the platform, compiler, and libraries used. Proceed with caution.

11.1 Foundations

The first computers performed only one task at a time. After a few years of development, however, people became concerned about the waste involved in keeping a very expensive machine idle, even if the idle periods were only the milliseconds required to read a punched card or paper tape. 416

multitasking

Concurrency, called *multitasking*, was first provided as a feature of operating systems, enabling the processor to switch tasks when a task became "blocked", for example, while performing input or output. The name *process* was introduced to describe a task that might or might not be running at a given time.

semaphores

monitors

The theory of *cooperating sequential processes* was worked out by Edsger Dijkstra, Tony Hoare, and Per Brinch Hansen. Dijkstra introduced *semaphores* as a low-level synchronization primitive, Hoare and Brinch Hansen added *monitors* for higher-level control of synchronization.

Since that time, operating systems have provided safe multitasking, usually based on some version of the basic operations worked out by Dijkstra, Hoare, and Brinch Hansen. Unfortunately, the same cannot be said for languages. The early work showed that safe concurrency could be enforced only by the compiler; it is not possible to provide safe functions in a library.

Before going further, we need some definitions. *Sequential code* executes in a fixed order determined by control structures. For example, s_1; s_2; … s_n means "execute s_1, then execute process

s_2, and so on, until s_n". A *process* is the execution of sequential code with interruptions. For example, "execute s_1, then go off and do something else for a while, execute s_2, and so on". A process uses the processor's *registers*, has its own *run-time stack*, and uses the memory-protection facilities of the processor to prevent code from accessing memory outside the area allocated to the process.

thread

A *thread* is like a process, but does not use the processor's memory-protection features. Typically, threads run within a process, and the process is memory-protected. Since changing memory bounds is typically an expensive operation, and control can be transferred between threads without changing bounds, threads are considered *light-weight* and processes are considered *heavy-weight*.

deadlock

A multi-threaded program may *deadlock*, meaning that each thread or process is waiting for something to happen. Here is a simple deadlock scenario: 417

1. Threads T_1 and T_2 both require resources R_1 and R_2.
2. T_1 acquires R_1.
3. T_2 acquires R_2.
4. T_1 requests R_2, fails, and blocks.
5. T_2 requests R_1, fails, and blocks.

starvation

A thread, or group of threads, may monopolize the processor, leading to *starvation* of other threads.

livelock

A group of threads may get into a cyclic set of states in which the threads continue to run but make no progress: this is called *livelock*. A real-life analogy is the situation in which you walk towards someone and attempt to avoid collision by moving left … but the other person moves to its right … so you move right … 418

distributed computing

Multi-threaded programming is notoriously difficult to get right. Hence, in many application scenarios, a different form of parallelism is used: *distributed computing*. In this paradigm, multiple processes work on a task in parallel, but do not use shared memory for communication. Instead, a form of *message passing* is used to initiate tasks and share results. This form of parallelism is extensively used in large-scale data processing, in particular for Web data, which can easily reach several (hundreds of) terabytes. Examples for distributed systems are telecommunication systems, (wireless) sensor networks, and cluster algorithms. Nowadays, application examples for large-scale data analysis are commonly grouped under the "Big Data" keyword.

#BigData

11.1.1 Thread-Safe Programming

Above, we have referred to "safe" concurrency. What does this mean? What does the (incomplete) program in Figure 136 on the next page print assuming that both threads run to completion? 419

race conditions

If we run this program many times, we will find that it usually prints `"1111"` but that it sometimes replaces one or two `1`s by `0`s: e.g., `"1110"` or `"1011"` but never `"0100"` or `"0000"`. The program has what is known as a *race condition*, and its results depend on the exact way in which the operations of the two threads are interleaved.

```
          struct { char c1; char c2; char c3; char c4 } s;
          s.c1 = s.c2 = s.c3 = s.c4 = 0;

      // thread 1                                   // thread 2
      ++s.c1;                                       ++s.c3;
      ++s.c2;                                       ++s.c4;

          cout << s.c1 << s.c2 << s.c3 << s.c4 << endl;
```

Figure 136: Fragments of a multithreaded program

shared variables

critical region

The pioneers recognized that race conditions can occur only when there are variables that can be accessed by two or more threads concurrently. Let's call these *shared variables*. When a thread is manipulating a shared variable, it must be protected from interruption by another thread. A chunk of code that manipulates a shared variable is called a *critical region*. Synchronization primitives ensure that a thread can execute its critical regions without interruption.

pre-emptive vs. co-operative multithreading

A concurrent system is *pre-emptive* if a process can be interrupted at any time; it is *non-pre-emptive* (or *co-operative*) if processes choose when they can be interrupted (usually by periodically calling a system function with a name like `synchronize` or `coordinate`). Programming non-pre-emptive systems is easier, but programmers must obey the discipline of giving up control at frequent intervals; failure to do so locks up the system (e.g., older versions of Windows). Programming pre-emptive systems is more difficult, because any process can be interrupted at any time, but has the advantage that no process can monopolize the system.

11.2 The POSIX Thread Library

Pthreads

Threads are not mentioned in either the C standard nor the C++98 standard. However, there have always been compiler-specific thread libraries. One of the best known is the cross-platform ***POSIX threads*** standard, often abbreviated to ***Pthreads***. The POSIX standard[48] used to be in the public domain but has now been taken over by IEEE, which means that you have to pay for the standard. POSIX threads library components are usually declared in `threads.h`, which is available on many UNIX/LINUX systems, as well as Mac OS X. On Windows, you can use *Open Source POSIX Threads for Win32*.[49] Figure 137 on the following page shows a simple 420
threaded program. Note the declaration of thread objects of type `pthread_t` and the use of 421
`pthread_create` to create the actual thread. The call to `pthread-exit` is needed to prevent the main program from terminating before the threads have finished their work.

The function `counter` defines the effect of the thread. It must have one parameter of type `void*` and must return a result of type `void*`. In the example, this parameter is used only to identify the thread; in a more practical example, it might be used to make threads perform distinct actions.

When this program is run, it prints the following (line breaks have been omitted to save space): 422

[48] POSIX is an acronym for "Portable Operating System Interface", see https://en.wikipedia.org/wiki/POSIX

[49] Pthreads Win32, http://sourceware.org/pthreads-win32/

```
void *counter(void *p)
{
    cout << "Starting thread " << *((int*)p) << endl;
    for (int i = 0; i < 10; ++i)
    {
        cout << i;
    }
    return 0;
}

void main()
{
    cout << "Creating threads ..." << endl;
    static int n1 = 1;
    pthread_t t1;
    pthread_create(&t1, 0, counter, &n1);
    static int n2 = 0;
    pthread_t t2;
    pthread_create(&t2, 0, counter, &n2);
    cout << "Waiting for threads to complete." << endl;
    pthread_exit(0);
}
```

Figure 137: Adding numbers with threads

```
Creating threads ...
Waiting for threads to complete.
Starting thread 1
0 1 2 3 4 5 6 7 8 9
Starting thread 0
0 1 2 3 4 5 6 7 8 9
```

From this output, we infer:

- The main program runs to `pthread_exit` before the threads are executed.
- Once started, a thread does not give up control until it has finished its task.

In a practical application, we would probably want threads to relinquish control from time to time so that other threads could do something. Adding one line to the function `counter` (the call to `sched_yield`) is all that we need: see Figure 138 on the next page. 423

With this change, the program prints the following output, showing that control alternates between the threads: 424

```
void *counter(void *p)
{
    cout << "Starting thread " << *((int*)p) << endl;
    for (int i = 0; i < 10; ++i)
    {
        cout << i;
        sched_yield();
    }
    return 0;
}
```

Figure 138: Co-operative multithreading with POSIX threads

```
Creating threads ...
Starting thread 1
0
Waiting for threads to complete.
1
Starting thread 0
0 2 1 3 2 4 3 5 4 6 5 7 6 8 7 9 8 9
```

A thread library typically provides *mutex*[50] objects for synchronizing threads. If threads are pre-emptive, mutex operations cannot be coded in a high-level language because they depend on special, non-interruptible instructions provided by all modern processors. A typical instruction for mutual exclusion is test-and-set, which checks, in a single atomic (non-interruptible) operation, the value in a memory location, sets it to a new value, and returns the old value.

mutex

test-and-set

POSIX threads are usually implemented without pre-emption. Consequently, the library functions can be written entirely in C, without the need for special machine instructions. The following table shows the mutex types and operations provided by POSIX threads. 425

Identifier	Description
`pthread_mutex_t`	Type of mutex object
`pthread_mutex_init(pthread_mutex*, ...)`	Initialize a mutex object
`pthread_mutex_destroy(pthread_mutex*)`	Destroy a mutex object
`pthread_mutex_lock(pthread_mutex*)`	Lock using a mutex
`pthread_mutex_unlock(pthread_mutex*)`	Unlock using a mutex

11.2.1 Encapsulating Threads

In these section, we will show how to use the low-level, C-style POSIX operations in a C++ environment, using ideas from the Appendix of Alexandrescu's book *Modern C++ Design* (Alexandrescu 2001) (and also http://www.informit.com/articles/article.asp?p=25298&rl=1

[50] Mutex is short for "mutual exclusion".

29/11/05). The first step is to encapsulate mutual exclusion in a base class, as shown in
426 Figure 139. The second step is to use RAII to ensure that locking is secure, as shown in
427 Figure 140.

```
class Synchronized
{
public:
    Synchronized()
    {
        pthread_mutex_init(&mtx, 0);
    }

    void acquireMutex()
    {
        pthread_mutex_lock (&mtx);
    }

    void releaseMutex()
    {
        pthread_mutex_unlock (&mtx);
    }

private:
    pthread_mutex_t mtx;
};
```

Figure 139: A base class for synchronization

```
template<typename T>
class Lock
{
public:
    Lock(T & client) : client(client)
    {
        client.acquireMutex();
    }

    ~Lock()
    {
        client.releaseMutex();
    }

private:
    T client;
};
```

Figure 140: Using RAII to simplify locking

A class T that requires locking inherits from class Synchronized and uses class Lock<T> to obtain
and release locks. Each critical operation constructs a local instance of Lock<T> to ensure that
critical sections are run without interruption. At the end of the critical section, RAII ensures
that the lock is automatically released. Figure 141 shows a simple account class implemented in 428
this way. 429

```
class Account1 : public Synchronized
{
public:
    Account1() : balance(0) {}

    void deposit(int amount)
    {
        Lock<Account1> guard(*this);
        balance += amount;
    }

    void withdraw(int amount)
    {
        Lock<Account1> guard(*this);
        balance -= amount;
    }

    friend ostream & operator<<(ostream & os, const Account1 & acc)
    {
        return os << acc.balance;
    }

private:
    int balance;
};
```

Figure 141: An account class with internal locking

Class Account1 uses *internal locking*. There are two other possibilities, and all three are defined
below: 430

Internal Locking: Each class ensures that concurrent calls cannot corrupt the class. The usual way to do this is to place a lock on any access to the class's data, as in class Account.

External Locking: Classes guarantee to synchronize access to static or global data but do not protect their instance variables. Clients must lock all invocations of their member functions.

Caller-ensured Locking: The class has a Mutex object (like class Synchronized) but does not
manipulate it. Instead, clients are expected to do the locking: 431

```
Account1 acc;
....
Lock<Account1> guard(acc);
acc.deposit(50000);
```

Unfortunately, all three methods have disadvantages. Suppose we use internal locking for an ATM which charges a fee of $5 for each withdrawal. Suppose further that the design requires that there must be no transaction between withdrawing the amount and charging the fee. The following code does not satisfy this requirement because it might be interrupted between the two withdrawals:

```
void ATMwithdraw(Account1 acc, int amount)
{
    acc.withdraw(amount);
    acc.withdraw(5);
}
```

The obvious correction is to apply a lock that holds over both calls:

```
void ATMwithdraw(Account1 acc, int amount)
{
    Lock<Account1> guard(acc);
    acc.withdraw(amount);
    acc.withdraw(5);
}
```

This will work if the thread library supports *recursive locking* – a single thread can lock/unlock the same object more than once.[51] Nevertheless, it is inefficient because two levels of locking are unnecessary. In a large application, a single object might be locked many times before actually being used. Another possibility is that the thread library does not support recursive locking; in this case, the code above will deadlock when it attempts to lock the account twice.

Caller-ensured locking avoids this particular problem, but still requires discipline from clients. Here is the design problem: the class `Account` consists of a finite amount of code and is therefore *bounded*; the rest of the application uses `Account` and may be of any size – it is *unbounded*. With caller-ensured and external locking, we have to examine unbounded code to be sure that synchronization is correct. We have lost encapsulation. 432

433 Figure 142 on the facing page shows how we can combine internal and external locking. Each function in the class comes in two flavours. The first version has a `Lock` parameter and is intended to be used with external locking: the client acquires a lock and passes it by reference to the function. The second version has no `Lock` parameter and implements internal locking by acquiring the lock itself. For a simple transaction, we rely on the internal lock: 434

```
Account2 acc;
acc.deposit(500);
```

[51] Although POSIX threads support recursive locking, some people, including POSIX implementors, argue against its use.

```
class Account2 : public Synchronized
{
public:
    Account2() : balance(0) {}

    void deposit(int amount, Lock<Account2>&)
    {
        balance += amount;
    }

    void deposit(int amount)
    {
        Lock<Account2> guard(*this);
        balance += amount;
    }

    void withdraw(int amount, Lock<Account2>&)
    {
        balance -= amount;
    }

    void withdraw(int amount)
    {
        Lock<Account2> guard(*this);
        balance -= amount;
    }

    friend ostream & operator<<(ostream & os, const Account2 & acc)
    {
        return os << acc.balance;
    }

private:
    int balance;
};
```

Figure 142: The best of both worlds

For a more complex transaction, we use external locking:

```
void ATMwithdraw(Account2 acc, int amount)
{
    Lock<Account2> guard(acc);
    acc.withdraw(amount, guard);
    acc.withdraw(5, guard);
}
```

11.2.2 A Better Lock

435 436 Alexandrescu also shows how to make better locks by exploiting C++ features. Figure 143 shows his class Lock. The new kind of Lock must be a stack-allocated variable and must be associated with a particular object. Locks cannot be passed by value to functions or returned from functions. It is possible to pass Locks by reference because no copy is made. Disabling the address operator makes it hard (but not impossible) to create aliases of a Lock. The result is that if you own a Lock<T> then you have in fact locked a T object and that object will eventually be unlocked.

11.3 Multithreading in C++11

C++11 added multithreading as a major new feature to the language. Rather than relying on compiler- and platform-specific libraries, it is now possible to write portable code involving multithreading. Within this course, we can only cover some parts of the new C++11 standard. The reference book (Stroustrup 2013, Chapter 41) covers all new language features, but for a more programming-oriented discussion, the book (Williams 2012) is recommended. An overview is also available in (Josuttis 2012, Chapter 18).

11.3.1 Hello, C++11 Thread!

437 Figure 144 on page 234 shows a minimal example for a multithreaded program using the C++11 standard thread library. Important new features are:

- We import the standard thread library header, with `#include <thread>`
- Our “hello” code has been moved to a function. This is necessary, because each thread is started with a function. For single-threaded programs, this has always been `main()`, but for any new threads we create, we have to provide the starting function explicitly.
- In the main program, we create a new `std::thread t` and tell it to start the function `hello()`. At this point in the code, there are now two threads running: one for the `main()` program, one executing `hello()`.
- Finally, we synchronize our two threads, by telling the main program thread to wait for thread `t`, using `t.join()`. Without this statement, the main program could finish execution (thus ending the program), before thread `t` had a chance to write any output.

Running this program requires (i) a compiler with support for C++11 threads and (ii) the availability of a multi-threading library for your specific platform. GNU gcc started adding multithreading support in version 4.3,[52] but you might have to explicitly enable C++11 features with the option `-std=c++11` in case your compiler defaults to C++98. You probably also have to tell it which threading library to use on your platform; on Linux, this is usually pthreads. Thus, to compile the program you would run:

```
$ g++ -std=c++11 hellothread.cpp -l pthread
```

If your compiler coughs up something like `error: thread is not a member of std`, then your implementation does not support C++11 threads.

[52] For details on supported C++ language versions in gcc, see https://gcc.gnu.org/projects/cxx-status.html.

```
template <class T>
class Lock
{
public:
    Lock(T & client) : client(client)
    {
        client.acquireMutex();
    }

    ~Lock()
    {
        client.releaseMutex();
    }

private:

    // The user of the lock
    T client;

    // no default constructor
    Lock();

    // no copy constructor
    Lock(const Lock &);

    // no assignment
    Lock & operator=(const Lock &);

    // no heap allocation of individual objects
    void *operator new(std::size_t);

    // no heap allocation of arrays
    void *operator new[](std::size_t);

    // no destruction of heap objects
    void operator delete(void *);

    // no destruction of heap arrays
    void operator delete[](void *);

    // no address taking
    Lock * operator&();
};
```

Figure 143: A lock with restricted uses

```
#include <iostream>
#include <thread>

void hello()
{
  std::cout << "Hello, thread!"  << std::endl;
}

int main()
{
    std::thread t(hello);
    t.join();
}
```

Figure 144: Hello, C++11 Thread!

11.3.2 Working with threads

Let's look at how to work with threads in more detail. Most operations are managed through an object of the `std::thread` class. In the example in Figure 144 above, we've already seen how to
438 start a new thread, by defining a `std::thread` object `t` and passing it a function to execute:

```
std::thread t(hello);
```

You might be tempted to write

```
std::thread t(hello());
```

since `hello()` is a function, but the compiler will read this completely differently: this declares (but does not define) a new function `t` that takes a single parameter (of type 'pointer to function') and returns a `std::thread` object.

Of course, instead of passing a function, you can also pass a function object. In this case, the object will be copied to the thread's local stack space, so you must ensure the copy constructor of your class behaves correctly.

Joining and detaching threads. You now have a new thread object (in this example `t`) that is associated with a running thread. It is created on the stack, and at one point will be deleted – either when the variable goes out of scope or an exception is thrown and the runtime system starts unwinding the stack. In either case, the destructor of the thread object `t` will be called. It is important that you instruct your thread what should happen in this case: otherwise your program will be terminated! You can either wait for the thread to finish using `join()` or ***detach*** the thread and let it run independently of your thread object `t`.

`join()` In cases where you need the results of a thread, you would typically use `join()`, since you cannot do anything useful until it completed its job. Once you `join`ed the thread, all memory associated with the thread will be cleaned up and the thread object will no longer be associated with a running thread. Consequently, you cannot call `join()` on the same thread more than once, so you might want to check your thread object to see if it holds a currently `joinable` thread: joinable() 439

```
    if (t.joinable())
    {
        t.join();
    }
```

The other option is to let your new thread run in the background, completely independent of your other threads. This is achieved by calling `detach()` on your thread object. Once detached, detach() the thread is managed independently by the runtime environment and you can no longer access it through your thread object. The conditions for detaching a thread are the same as for joining a thread, so you can check whether a thread can be detached with `joinable()`. 440

```
    t.detach();
    assert (!t.joinable());
```

Detached threads are useful when you have a specific task for each thread, such as opening multiple files in a GUI, where each file is managed by its own window, running in its own thread. You can also use them for background tasks that perform some periodic maintenance, like cleaning up unused files (in UNIX/LINUX, such background processes are known as ***daemon*** daemon programs).

To initiate background processes, you probably want to be able to pass arguments to the function executed by the thread. The thread library provides a simple way of doing this, as shown in Figure 145 on the following page. This program allows you to specify a sleep time in seconds 441 and then creates a new thread that will wait the specified amount of seconds before waking up, 442 issuing an alarm, and exit (of course, you could have a thread that does something more useful than sleeping). As you can see in the example, arguments are passed to a thread by providing them as additional arguments to the thread constructor (you can pass more than one argument this way). Note that these arguments are always copied; if you want to pass reference parameters you have to explicitly specify this in the thread constructor with `std::ref(arg)`. std::ref

Threads and Exceptions. Once you detached a thread, you no longer have to worry about the thread object that was used to create the thread. But in case of `join`, you have to consider all possible ways the object `t` might get deleted. In case of an exception, the run-time system will start unwinding the stack, thus calling the destructor of `t`, which in turn will call `terminate()` and abort your program. You could try to avoid this by placing additional `t.join()` operations in each `catch` statement. But as shown before, a much better way is to use RAII – see Figure 146. 443 Here, threads are encapsulated in a `thread_join` object. The destructor takes care of joining the 444 thread, so it will work correctly even in case of an exception. The code also shows another C++11 feature: The `=delete` notation for a member function tells the compiler to not create the default =delete version, so it will not be possible to copy or assign a `thread_join` object. This replaces the C++98 idiom of declaring a member function, like the assignment operator, as private, without implementing it (and there's also a corresponding `=default` to make it explicit that you do want =default the default version).

thread destructors can terminate your program!

```
#include <thread>
#include <chrono>
#include <iostream>

using namespace std;

void set_alarm(int secs)
{
  cout << "\n    Setting alarm for " << secs << " seconds." << endl;
  this_thread::sleep_for(chrono::seconds(secs));
  cout << "    " << secs << " seconds have passed. Time to wake up!" << endl;
}

int main()
{
  int s;

  cout << "Welcome to multi-alarm.\n";
  do {
    cout << "How many seconds do you want to sleep? 0 to exit: ";
    if (cin >> s && s > 0)
    {
      thread t (set_alarm, s);
      t.detach();
    }
  } while (s);
}
```

Figure 145: Starting multiple detached threads with parameters

11.3.3 Sharing Data between Threads

std::mutex

As discussed above, mutexes can be used to protect data shared between threads from becoming inconsistent due to concurrent writes (if all threads do is read from a common data structure, there is no problem). The workflow is to first lock the mutex associated with the data structure, then do the changes, then release the lock. All other threads trying to lock the same mutex will be blocked until the first thread releases the lock. Figure 147 on page 238 shows a basic example. 445

lock_guard

Note that we do not unlock the mutex: the lock_guard will automatically unlock the mutex in its destructor, so it will be released at the end of the function scope or if an exception is thrown (another example of RAII at work).

In this example, we used global variables for the data structure and its mutex, but in a real implementation, you would encapsulate both in a class that makes the connection between the mutex and its data structure explicit.

preventing deadlocks

Once you start locking, you have to think about possible deadlocks in your system. Deadlocks

```
#include <iostream>
#include <thread>

using namespace std;

class thread_join
{
public:
    explicit thread_join(void func())
    {
        cout << "starting thread...";
        t = thread(func);
    }

    ˜thread_join()
    {
        cout << "joining thread.";
        t.join();
    }
    thread_join(thread_join const &)=delete;
    thread_join& operator=(thread_join const &)=delete;

private:
    thread t;
};

void hello()
{
    cout << "Hello, thread! ";
}

void multi_hello() {
    thread_join tj1(hello);
    thread_join tj2(hello);
    throw true;
}

int main()
{
    try
    {
        multi_hello();
    }
    catch (...)
    {
        cout << "exceptional code!" << endl;
    }
}
```

Figure 146: RAII for joining threads

```
#include <mutex>

list<Widget> widget_list;
mutex widget_mutex;

void add_widget(Widget w)
{
    lock_guard<mutex> guard(widget_mutex);
    widget_list.push_back(w);
}
```

Figure 147: Locking a shared data structure through a mutex

can happen when a thread needs to lock more than one mutex to do its work and the same resources are needed by a second thread, but locked in a different order: both threads will now wait until the other resources become available, which will never happen. Deadlock avoidance is a complex topic, but one strategy that can be employed is to always lock multiple resources *in the same order*. This can be implemented in multiple ways; one option is to use `defer_lock`, as shown in Figure 148 on the facing page. 446

447 defer_lock

The program attempts to swap data from two members of `Gizmo`, so it will need to lock both objects to be able to perform the swap. The `defer_lock` option leaves the mutex unlocked when the lock object is constructed. Only when we pass the `unique_lock` objects to `lock`, the runtime system will attempt to acquire *all* locks. This operation is atomic: if it is not possible to obtain all locks, `lock` will fail with an exception and also unlock all partial locks it might have acquired.

11.3.4 Communication between Threads

So far, we have not discussed how two threads can work collaboratively on a problem. We can wait for a thread to finish using `join()`, but then this thread must end execution. What if one thread continuously creates new resources (e.g., objects) that are then consumed by a second thread?

avoid spinning (busy waiting)

We can use a shared data structure, protected with a mutex, to exchange data between threads. For example, one thread could add new elements to a queue, and a second thread takes them out for processing. But how does the second thread know there is a new element available in the queue? It could continuously check the queue for any (new) elements, but that is very wasteful, since it consumes resources – this is called ***spinning*** (or *busy waiting*) and is a well-known anti-pattern in programming. We could also let the second thread `sleep_for` a period of time, before checking for new input: this is better, but has the risk of waking up too often (and wasting resources again) or oversleeping (perhaps now the first thread will create new data faster than we consume it, due to taking a nap). A much better solution is to use the standard library facilities for triggering events between threads.

condition_variable

One implementation option is to use ***condition variables***. You can define a variable with

448

```
std::condition_variable my_condition;
```

```
#include <iostream>
#include <utility>
#include <mutex>

using namespace std;

class Gizmo
{

public:
    Gizmo() = default;
    Gizmo(int i):content(i){}

private:
    int content;
    mutable mutex m;

friend void locked_swap(Gizmo &, Gizmo &);
friend ostream & operator<<( ostream &, const Gizmo & );
};

ostream & operator<<( ostream & os, const Gizmo & g)
{
    os << g.content;
    return os;
}

void locked_swap(Gizmo & l, Gizmo & r)
{
  if(&l==&r)
    return;
  unique_lock<mutex> lock_a(l.m, defer_lock);
  unique_lock<mutex> lock_b(r.m, defer_lock);
  lock(lock_a, lock_b);
  swap(l.content, r.content);
}

int main()
{
    Gizmo g1(1);
    Gizmo g2(2);
    cout << "Gizmo_1 = " << g1 << "; Gizmo_2 = " << g2 << endl;
    locked_swap(g1, g2);
    cout << "Gizmo_1 = " << g1 << "; Gizmo_2 = " << g2 << endl;
}
```

Figure 148: Deferred locking of multiple mutexes to avoid deadlocks

notify_one

(note that you will need to #include <condition_variable>) and then call a notification function on this condition object with

```
my_condition.notify_one();
```

wait()

The consuming thread that wants to be notified can wait() on this condition object by supplying a function (e.g., is the queue not empty?), like here:

```
my_condition.wait(data_lock, check_function);
```

where check_function would be implemented as

```
bool check_function() { return !data.empty(); }
```

Using C++11's support for unnamed functions, we can actually write this definition directly into the wait() call, using a so-called ***lambda***:

```
my_condition.wait(data_lock, []{return !data.empty();})
```

(we will discuss lambdas in more detail in the last lecture, see Section 14.4.1 on page 321).

```
#include <queue>

template<typename T>
class threadsafe_queue
{
public:
    threadsafe_queue();
    threadsafe_queue(const threadsafe_queue &);
    threadsafe_queue& operator=(const threadsafe_queue &) = delete;

    void push(T new_value);
    bool try_pop(T& value);
    void wait_and_pop(T& value);
    bool empty() const;

private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
};
```

Figure 149: Interface for a thread-safe queue with condition variables

A thread-safe queue. The threadsafe_queue interface shown in Figure 149 on the preceding page, adapted from (Williams 2012), is an example for a thread-safe queue that provides synchronization between threads through a condition variable. The interface is based on the STL queue, but combines the two functions front() and pop() into a single function try_pop() that returns the front element and removes it from the queue at the same time if one is available (returning false otherwise). In the STL, they are split for the same reason as discussed in Section 10.3 on page 211 for the STL stack: to provide exception safety. But this interface is not thread-safe: two threads could check if the queue has at least one element and then pop it – but both would be able to get the same element and process it, leading to an inconsistent system state: 449 450

```
    Thread 1                        Thread 2

if( !q.empty() )
{                               if( !q.empty() )
    v = q.front();              {
                                    v = q.front();
    q.pop();
    process(v);                     q.pop();
                                    process(v);
}                               }
```

Note that mutexes do not help us here: this problem occurs even when all queue functions are protected by a mutex. The general issue is the design of a ***thread-safe interface***. There are various solutions to this problem; one option is to combine both front() and pop() into a single call. To also make it exception-safe, we pass the argument to pop by reference. To provide for synchronization, we have two versions of pop: The first, try_pop(), always returns immediately but will not succeed in returning an element if the queue is empty. The second, wait_and_pop(), will block in case of an empty queue until another thread added a new element.

designing thread-safe interfaces

The class in Figure 150 on the next page shows a complete implementation for the thread-safe 451
queue. Note that each operation is protected by a mutex. The assignment operator is not 452
implemented to make the example simpler (we do provide a copy constructor, so a thread-safe
queue can still be copied). The call to wait makes use of the lambda syntax mentioned earlier. 453
Also note that we make the mutex mutable to allow locking a const queue.

We can now run two threads, one producing and one consuming data, which synchronize using this queue: 454

```
threadsafe_queue<Widget> widget_queue;

void data_preparation_thread()
{
    while(more_data_to_prepare())
    {
        Widget const w=prepare_widget();
        widget_queue.push(data);
    }
```

```
#include <mutex>
#include <condition_variable>
#include <queue>

template<typename T>
class threadsafe_queue
{
public:
    threadsafe_queue() {}

    threadsafe_queue(threadsafe_queue const& other)
    {
        std::lock_guard<std::mutex> lk(other.mut);
        data_queue=other.data_queue;
    }
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(new_value);
        data_cond.notify_one();
    }
    void wait_and_pop(T& value)
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this]{return !data_queue.empty();});
        value=data_queue.front();
        data_queue.pop();
    }
    bool try_pop(T& value)
    {
        std::lock_guard<std::mutex> lk(mut);
        if(data_queue.empty())
            return false;
        value=data_queue.front();
        data_queue.pop();
        return true;
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lk(mut);
        return data_queue.empty();
    }

private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
};
```

Figure 150: Implementation for a thread-safe queue with condition variables

```
}

void data_processing_thread()
{
    while(true)
    {
        Widget w;
        widget_queue.wait_and_pop(w);
        process(w);
    }
}
```

11.3.5 The Keyword volatile

In multi-threaded code, one thread may wait for an event that is caused by another thread. In this example, an instance of `Gadget` has a function `wait` that checks the status of its variable `sleeping` once every second. Eventually, another thread calls `wakeup`, changes the value of `sleeping`, and terminates the `while` loop. 455

```
class Gadget
{
public:
    void wait()
    {
        while (sleeping)
        {
            pause(1000); // pauses for 1000 milliseconds
        }
    }

    void wakeup()
    {
        sleeping = false;
    }

private:
    bool sleeping;
};
```

Unfortunately, this class has a serious problem: The compiler may notice that `pause`, being a system or library function, cannot possibly change the value of the instance variable `sleeping`. There is an obvious optimization: copy the value of `sleeping` into a processor register and inspect the register each time round the loop, thereby avoiding an expensive memory access. But, of course, the register never changes, the `wakeup` call goes by unnoticed, and the thread is stuck in the `wait` loop.

Compilers always try to use registers as much as possible; these optimizations have a significant effect on performance. Preventing optimizations of this kind globally would not be a good idea.

Both C and C++ provide an alternative in the form of the modifier `volatile`. Changing the 456 declaration of `sleeping` to

```
private:
    volatile bool sleeping;
```

informs the compiler that `sleeping` may change its value at any time, and that this value must therefore be refreshed from memory whenever needed, not saved in a register.

The original use of `volatile` was for peripheral devices that used "direct memory access". A keystroke, for example, would change the value stored at a particular location (and probably also raise an interrupt). The program would bind a variable to that location (C and C++ have a mechanism for that, too) and use the variable to read the keyboard. It was necessary to declare the variable to be `volatile` for the same reasons as above – optimization would be a disaster. Nowadays, `volatile` is still used, but usually to make code thread-safe.

11.3.6 Constant and Mutable

457 The following program compiles successfully and prints "1":

458

```
class Counter
{
public:
   Counter() : timesUsed(0) {}

   void add()
   {
      ++timesUsed;
   }

   void show() const
   {
      cout << timesUsed << endl;
   }

private:
   int timesUsed;
};

int main()
{
   Counter c;
   c.add();
   c.show();
   return 0;
}
```

If we change the declaration of `c` to 459

```
   const Counter c;
```

the compiler complains because `Counter::add` is not declared `const` and therefore cannot be used with a `const` object.

If we attempt to correct this by changing the declaration of `Counter::add` to

```
   void add() const
```

the compiler complains that the value of a `const` object is being changed. This is because all data members of a `const` object are considered `const`. (The problem here, of course, is that we are changing `timesUsed`.)

Suppose that `timesUsed` is not logically part of the object. It might, for example, just be a counter that we are using to see how often a function is called. The keyword `mutable`, used to qualify a variable, informs the compiler that the variable should not be considered as part of the object. "Mutable" can be interpreted as "can never be `const`". The following version of the
program compiles and runs correctly. 460

```
class Counter
{
public:
   ....
private:
   mutable int timesUsed;
};

int main()
   ....
```

Here is another application of `mutable`. We have a class `Widget` with a member function that
returns a string representation of the object. The object's value changes only occasionally, and
it is rather expensive to compute the string representation. Consequently, we would like to
cache the stringified version and do the conversion only when necessary. The class would look
something like `Widget` in Figure 151. The problem with this class is that `getRep` *ought* to be 461
declared `const` because, logically, it does not change the state of the object. But it cannot be 462
declared `const`, because it changes the value of `cache` and `staleCache`, which are not *logically*
part of the object.

The solution, of course, is the qualifier `mutable`: 463

```
class Widget
{
public:
   string getRep() const
   ....
```

```
class Widget
{
public:
   string getRep()
   {
      if (staleCache)
      {
         cache = ....;
         staleCache = false;
      }
      return cache;
   }

   void change()
   {
      ....
      staleCache = true;
   }

private:
   bool staleCache;
   string cache;
};
```

Figure 151: Is a cache part of an object?

```
private:
   mutable bool staleCache;
   mutable string cache;
};
```

11.3.7 Promising the Future

So far, we have been working on a rather low abstraction level, manually creating threads and manipulating them. A higher-level abstraction is the concept of a ***Future***, which encapsulates a unit of (asynchronous) work. The counterpart of a future is a ***Promise***, which a function can provide as a result.[53] Working with futures and promises is especially useful for developing code that can dynamically scale on multi-core systems. It is also an important foundation for modern systems based on ***Reactive Programming***.[54]

multi-core programming

Here is one way to make use of futures and promises: Rather than waiting for the result of a function call to return (and blocking a program in the meanwhile), we let it run asynchronously,

[53] Futures and Promises in C++ roughly correspond to `Java` 8's `CompletableFuture` and `CompletionStage`.

[54] See the ***Reactive Manifesto*** for more information on reactive programming: https://www.reactivemanifesto.org/

in a different thread, until we need the result. Let's say a function `deep_thought` needs some time to calculate an `int` value. Rather than calling it directly with

```
int result = deep_thought();
```

we can process it asynchronously through a future: 464

```
future<int> result = async(deep_thought);
do_other_stuff();
cout << "The answer is: " << result.get() << endl;
```

Here, `result.get()` blocks if the result is not available yet. In this example, we created the `future` object directly. A more general approach is to pass the result through a promise object, as shown in Figure 152. 465

11.4 Distributed Computing

So far, we were concerned with single-computer concurrency, where multiple processors with multiple cores execute the threads of a program. But what if a single computer is not fast enough to solve a problem? A typical example is web-scale data processing, where you have large amount of data to analyze ("Big Data"). "Big Data"

```
#include <future>
#include <chrono>
#include <iostream>

using namespace std;

void deep_calc(int i, promise<int> * promiseResult)
{
    // calculation takes 7.5 million years
    this_thread::sleep_for(chrono::seconds(1));
    promiseResult->set_value(i+1);
}

int main()
{
  promise<int> promiseObj;
  future<int> futureObj = promiseObj.get_future();
  thread t(deep_calc, 41, &promiseObj);
  cout << "I can do other stuff now..." << endl;
  cout << "The answer is: " << futureObj.get() << endl;
  t.join();
}
```

Figure 152: Futures and Promises

cluster computing

One solution is to use *clusters*, collections of individual machines running the same computations on parts of the data. It then becomes desirable to automatically distribute a large job to a large number of machines (e.g., tens of thousands of PCs), without having to make changes to the code doing the processing. A solution that has been widely adopted by industry and science is *MapReduce*. As we will see below, this paradigm can also be applied for parallel programming on multi-core systems.

11.4.1 MapReduce

MapReduce[55] is a parallel programming approach introduced by Google engineers (Dean and Ghemawat 2004) for easing the development and deployment of large-scale data processing applications. It is the framework currently used by Google for many of its applications performing massively parallel processing of hundreds of terabytes of data in its distributed data centers. The MapReduce framework has meanwhile also been adopted by companies such as Yahoo!, Facebook, or LinkedIn and proved to be a useful (albeit not actually new) paradigm for cloud or grid computing. A well-known commercial cloud computing service offering MapReduce is Amazon's *Elastic MapReduce*[56] running on top of its *Elastic Compute Cloud (EC2)* service.

Together with cloud computing, MapReduce can save significant costs and time compared with buying traditional server resources when they are only needed for specific, large-scale jobs ("burstiness"). For example, the *New York Times* used MapReduce running on 100 Amazon EC2 instances and a MapReduce application (implemented with Hadoop) to process 4TB of raw image TIFF data (stored in Amazon's Simple Storage Service, S3) into 11 million finished PDFs in the space of 24 hours at a computation cost of about $240 (not including bandwidth).[57]

11.4.2 MapReduce Concept

The basic idea of MapReduce stems from *functional programming*.[58] A particular feature of functional programming is that functions have no side-effects, hence they are trivially parellelizable. This is part of the reason for the currently growing interest in functional and object-functional programming languages (like Scala), as moderns PCs are increasing the number of cores, but not clock speed – hence requiring better support for parallel programming to make use of these multi-core systems.[59]

MapReduce takes one idea of functional programming and brings it to object-oriented languages (originally only C++, but now many languages offer MapReduce implementations or bindings). To compute a certain result on a large set of data, first split it out into n partitions (e.g., 4 for a quad-core system, or 1000 for a large cluster of PCs). Then, execute the same function on each of the partitions. This is expressed by a `map` function that takes the input job in form of a `list` and computes an intermediate result for each list element (which can be, e.g., a single

[55] MapReduce in Wikipedia, http://en.wikipedia.org/wiki/MapReduce

[56] Elastic MapReduce, http://aws.amazon.com/elasticmapreduce/

[57] New York Times Blog, http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/

[58] See http://en.wikipedia.org/wiki/Functional_programming for an introduction if you are not familiar with the functional programming paradigm.

[59] See Herb Sutter's article, *"The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software"*, Dr. Dobb's Journal Vol. 30, No. 3 (March 2005), http://www.gotw.ca/publications/concurrency-ddj.htm.

document, a single sensor reading or database record). Then, in a second step, the intermediate results have to be combined to the end result. This is the reduce step: 466

```
map (in_key, in_value) -> (out_key, intermediate_value) list

reduce (out_key, intermediate_value list) -> out_value list
```

In more detail, these functions work in the following way. First, a list of input data sources has to be built. For example, a list of documents, database rows, or sensor readings. Then, instead of developing a single function that computes the required result from the input (e.g., a word count over all input documents), we need to split the computations into the map and reduce functions.

Map. The map function takes as input a (in_key, in_value)-pair and computes a corresponding (out_key, out_value)-pair. For example, if we want to count words across a large number of documents, we can map the word-count function onto each individual document. The (intermediate) result is then a word count for each individual document (independent from the other documents).

Reduce. When the map phase is over, the reduce phase starts to combine the individual results into the final end result. In the word count example, we now know how often a word appears in each document, so we still have to add up the results for each word across all documents to obtain the final result. This is done by the reduce function. To distribute the reduce job onto multiple machines, we need to aggregate the results by key (in practice, the key values are often stored in a hash table and then distributed onto parallel reduce jobs modulo the number of available nodes).

Figure 153[60] shows the parallel workflow in a MapReduce application. Note that the reduce phase cannot start until the map phase has been completed. 467

11.4.3 MapReduce Implementations

The original (and current) implementation by Google is developed in C++. Not surprisingly, this implementation is not available outside Google, but many other implementations in various languages have been developed as well, including Java (Hadoop), Ruby (Skynet), Haskell (Holumbus), and Erlang (Disco). Many of them are free/open source.

For writing mappers and reducers in C++, two free options are *QtConcurrent* (used in this lecture) and the *Pipes* interface for Hadoop.

[60] Image Copyright 2007 University of Washington, Distributed under Creative Commons Attribution 2.5 License, Google lectures on MapReduce, http://code.google.com/edu/submissions/mapreduce-minilecture/listing.html

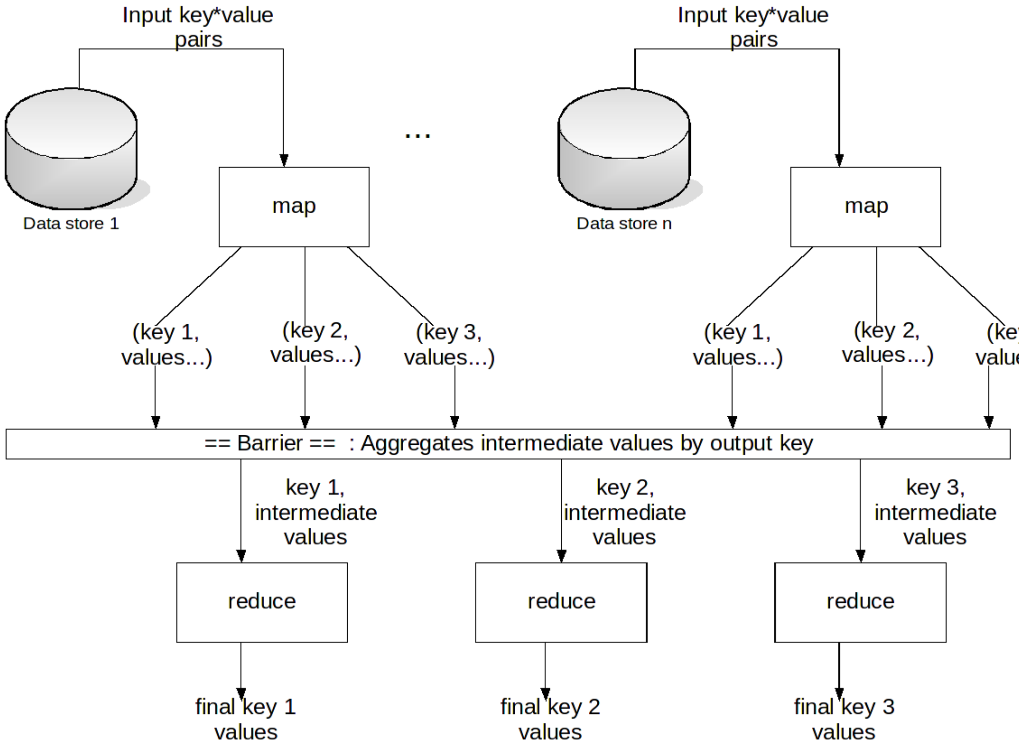


Figure 153: MapReduce Workflow

Hadoop. Apache Hadoop[61] is an open source, Java-based framework for cluster computing using the MapReduce concept. It includes a distributed file system (HDFS) and provides language bindings for many programming languages, including *Pipes* for C++.[62] With these bindings, it is also possible to write mappers and reducers in different languages – e.g., a `map` function in C++ and a `reduce` function in Python.

One of the main code contributors to Hadoop is Yahoo!, who also runs one of the largest Hadoop clusters.[63] Hadoop can be run on a single PC, but is primarily targeting deployment on large clusters (e.g., it is rack-aware) and features support for cloud computing (Cloudstore, Amazon S3).

QtConcurrent. *The Qt Company* maintains the open-source, cross-platform (Linux, Mac, Windows, as well as some mobile platforms) C++ *Qt* template library.[64] Part of this library is the parallel namespace *QtConcurrent*,[65] that comes with a MapReduce implementation. Unlike Hadoop, this implementation is targeting single PCs and optimized for multi-core parallelism. Since it is significantly easier to setup this library than a Hadoop cluster, we use QtConcurrent's

[61] Hadoop, http://hadoop.apache.org/

[62] See https://wiki.apache.org/hadoop/C++WordCount for a Hadoop version of the "WordCount" program.

[63] Yahoo! repeatably won the yearly sorting contests using its Hadoop cluster, e.g., in April 2009 the minute sort by sorting 500 GB in 59 seconds (on 1400 nodes) and the 100 terabyte sort in 173 minutes (on 3400 nodes).

[64] Originally developed by the Norwegian company Trolltech, which was later acquired by Nokia where it became part of QtLabs, which was then sold to Digia around the time Nokia was bought by Microsoft and finally spun off into The Qt Company in 2014. For more on Qt, see https://en.wikipedia.org/wiki/Qt_(software).

[65] QtConcurrent, http://doc.qt.io/qt-5/qtconcurrent-index.html

MapReduce implementation in the following examples. However, note that the programming methodology is the same as for a large cluster.

11.4.4 Hello, MapReduce!

The “Hello, world” for MapReduce is a parallel word count example. Input is a set of documents. Expected output is a count for each word appearing in the documents.

To implement this with MapReduce, we have to:

1. build the list of input documents to count
2. write a mapper function (here called `countWords`)
3. write a reducer function (here called `reduce`)

The main data structure passed between the `map` and `reduce` phases is a `Map` (using Qt’s cross-platform implementation, `QMap`, rather than an STL `map`):

```
typedef QMap<QString, int> WordCount;
```

Then we can call the `mappedReduced` function that takes as input the list to be processed, the map, as well as the reduce function and calls them in the appropriate order on the input documents (Figure 154). 468

```
int main() {
    QStringList files = findFiles("../qt/src/corelib",
         QStringList() << "*.cpp" << "*.h");
    WordCount total = mappedReduced(files, countWords, reduce);
}
```

Figure 154: Counting words, MapReduce-style

The map function `countWords` (Figure 155 on the following page) simply has to count the words in each input list element, i.e., each document.[66] 469

Finally, the `reduce` function (Figure 156 on the next page) has to add up the individual word count results (per-document) for all input documents. 470

The `wordcount` example program shipped with QtConcurrent additionally demonstrates the speedup achieved through MapReduce by comparing the runtime with a single-threaded version: 471

```
finding files...
1115 files
warmup
warmup done
single thread 1881
MapReduce 482
MapReduce speedup x 3.90249
```

[66] The `foreach` statement is a macro supplied by the Qt library, see http://doc.qt.io/qt-4.8/containers.html.

```
WordCount countWords(const QString &file) {
    QFile f(file);
    f.open(QIODevice::ReadOnly);
    QTextStream textStream(&f);
    WordCount wordCount;

    while (textStream.atEnd() == false)
        foreach (QString word, textStream.readLine().split(" "))
            wordCount[word] += 1;

    return wordCount;
}
```

Figure 155: Counting words with MapReduce: Map function

```
void reduce(WordCount &result, const WordCount &w) {
    QMapIterator<QString, int> i(w);
    while (i.hasNext()) {
        i.next();
        result[i.key()] += i.value();
    }
}
```

Figure 156: Counting words with MapReduce: Reduce function

This program was executed on a quad-core processor. However, it is important to realize that the MapReduce framework will automatically scale to a larger number of cores, without requiring any changes to the code: Unlike classical multi-threading, where the number of threads has to be specified by the programmer using features of the multithreading library, here the MapReduce framework takes care of starting and stopping threads. Using a large cluster (and a suitable MapReduce implementation like Hadoop), you can test your program on, e.g., 4 nodes and then deploy the same code on 10,000 nodes.

References

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Generic Patterns Applied*. Addison-Wesley.

Dean, J. and S. Ghemawat (2004, December). MapReduce: Simplified Data Processing on Large Clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, San Francisco, CA, USA. http://labs.google.com/papers/mapreduce.html.

Josuttis, N. M. (2012). *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley. http://www.cppstdlib.com/.

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.

Williams, A. (2012). *C++ Concurrency in Action*. Manning. http://www.manning.com/williams/.

12 System Design

12.1 Logical and physical design 254

12.2 Linkage 255

12.3 Namespaces 256

12.4 Cohesion 256

12.5 Coupling 258

12.6 The Pimpl Idiom 267

12.7 Refactoring 276

12.8 Miscellaneous Techniques 279

For large systems, design must take into account not only the performance of the finished product, but also its development and maintenance. Two anecdotes illustrate the problems of large-scale development.

1. Lakos worked at Mentor Graphics, a company that developed one of the first general-purpose graphics libraries for UNIX systems. Programmers were in the habit of `#include`ing all of the headers that they thought their component might need. The result was that compiling the library on a network of workstations required ***more than a week***.

   The source code was carefully reviewed and unnecessary `#include`s were removed. Compile time was significantly reduced (Lakos 1996).

2. A major software product was completed, the source code and executables were written to CDs, and the product was ready to be shipped to the customer. A couple of days later, a programmer made a small change to a member function, something like this: 473

   ```
   class Widget
   {
   public:
      double get()
      {
          ++accessCount;            // this line added
          return size;
      }
   private:
      static double size;
      unsigned long accessCount;
   };
   ```

   After this change, the program crashed during most runs.

   The problem was solved after several days of debugging. The following code was the culprit: 474

   ```
   Widget *pw;
   // .... several pages of code
   s = pw->get();
   ```

12.1 Logical and physical design

logical design

Object-oriented ***logical design*** concerns the organization of classes from the point of view of functionality, inheritance, layering, and so on. It is discussed in many books on C++ programming and software engineering, often with the use of UML diagrams, patterns, and other aids.

physical design

Physical design concerns the distribution of the various classes and other entities into files, directories, and libraries. Physical design is important for large projects but is less often discussed in books. Unfortunately, the only reference that explicitly covers physical design issues in industry-sized C++ projects is (Lakos 1996). "Unfortunate" because this book is now 20 years old and most of the technical aspects it describes have become outdated.

Logical and physical design are closely related. If the logical design is bad, it will not usually be possible to obtain a good physical design. But a good logical design can be spoilt by a poor physical design.

Logically, a system consists of classes (including enumerations) and free functions, with classes providing the main structure. 475

Physically, a system consists of ***components***. There are various ways of defining components but, fortunately, there is one precise definition that is suitable for systems written mostly or entirely in C++:

> ***A component consists of an implementation file (`.cpp`) and one or more header (`.h`) files.***

The header and implementation files may both use `#include` to read other header files; and parts of these may be skipped by conditional preprocessor directives.

compilation units

translation units

The compiler turns each component into an object file (`.obj`) or sometimes a library file (`.lib`/`.a`/`.o`/`.so` or (in Windows) `.dll`). The linker takes all the object and library files and creates an `executable`. Components are also called ***compilation units*** (see Section 1.2.1 on page 6) and ***translation units***.

Since C++ does not allow a class declaration to be split over several files, the definition of component implies that ***every class belongs to exactly one component***. However, the definition does allow a component to contain more than one class and, in fact, it is often useful to define components with several classes:

- Several closely-related classes may be declared in a single header file.
- A class, or classes, needed by a single component can be declared in the implementation file of that component.

In general, the ***logical structure*** of a design is determined by its classes; the ***physical structure*** of a design is determined by its components.

There are various kinds of logical association between two classes: a class can contain an instance of another class or a pointer to such an instance; inherit another class; have a value of another class passed to it by value, reference, or as a pointer; and so on. Physical associations are, in general, simpler: one component either uses another component or it doesn't.

A ***client*** of component C is another component of the system that uses C. With a few exceptions (see Section 12.5.2 on page 261), clients can be recognized because they `#include` header files: 476
component `C1` is a client of `C2` if either `C1.h` or `C1.cpp` needs to read `C2.h`. Note that:

- "Reading" `C2.h` is not the same thing as `#include`ing `C2.h` because a client might `#include` `X.h` which `#include`s `C2.h`. In Figure 157, compiling `C1.cpp` requires reading `C2.h`.

- The client relationship is transitive: if `C1` is a client of `C2`, and `C2` is a client of `C3`, then `C1` is a client of `C3`.

```
#ifndef C1_H                        #ifndef X_H
#define C1_H                        #define X_H

#include "X.h"                      #include "C2.h"
....                                ....
#endif                              #endif
```

Figure 157: Physical dependency: `c1.h` "reads" `c2.h`

12.2 Linkage

We discussed linkage issues previously (see Section 4.3 on page 72). To get good physical structure, we want to minimize dependencies between components. Doing this requires knowing how dependencies arise.

Declarations in a header or implementation file have no effect outside the file (except, of course, that declarations in a header file are read in the corresponding implementation file). 477

Definitions in a header or implementation file cause information to be written to the object file, and therefore create dependencies. As we have seen, definitions should be avoided altogether in header files.

General rules:

- Put only constant, enumeration, and class declarations into header files. 478

- Put all data and function definitions into implementation files.

- Put data inside classes wherever possible.

- If you have to put data at file scope, declare it as `static`.

- Declarations ***and*** definitions for template classes must be put into header files, so that the compiler can instantiate templates.

The compiler considers inlined functions to be declarations, not definitions, even if they have bodies. It is therefore acceptable to put, for example, 479

```
inline double sqr(double x) { return x * x; }
```

in a header file. However, if you remove the `inline` qualifier, the program may not compile! This is because the header file may be read more than once, putting two copies of the function `sqr` into the program, which causes problems for the linker (we discuss inlining in more detail in Section 12.5.5 on page 266).

12.3 Namespaces

namespace

Namespaces in C++ are used similar to packages in Java. They are helpful in structuring large-scale projects and keeping them manageable. To declare your own namespace `foospace`, you'd write:

480

```
namespace foospace
{
   class foo
   {
        ....
   };
   ....
}
```

Inside the namespace declarations, you put your own classes, functions, etc. For example, the class `foo` defined inside the namespace above is now referenced as `foospace::foo`.

nested namespaces

The `using` directive for your own namespaces works just as for the standard namespace we've seen so far. You can also have nested namespaces, like in `foospace::barspace`. A class `bar` declared in the inner namespace would then be referenced as `foospace::barspace::bar`.

default namespace and `::` prefix

All classes defined outside a namespace declaration go into the ***default namespace*** and can be referenced with a `::` prefix. For example, if you have a class `baz` in the default namespace, you can reference it with `::baz`. This might be necessary in case there is also a `baz` in another namespace you are `using` in your code.

anonymous namespace

Finally, you can have an unnamed, ***anonymous namespace***. Everything inside the anonymous namespace becomes invisible outside the translation unit. Thus, in term of linking, they are a (better) alternative to `static` variables at global scope (we will discuss global variables in more detail below). Note that this is the only way to control visibility: there is no C++ equivalent to package visibility in Java. For more on namespaces, see (Weiss 2004, Chapter 4.15), (Dewhurst 2005, Item 23), (Prata 2012, pp.482–497), and (Stroustrup 2013, Chapter 14).

12.4 Cohesion

Every book on Software Engineering includes the slogan ***low coupling, high cohesion***. This mantra is useful only if we know:

- what is coupling?
- what is cohesion?
- how do we reduce coupling?

- how do we increase cohesion?

Coupling is probably the more important of these two aspects of a system, and we discuss it in Section 12.5 on the next page.

Cohesion is the less important partner of the "low coupling, high cohesion" slogan. Roughly speaking, a component is ***cohesive*** if it is independent of other components (as far as possible) and performs a single, well-defined role, or perhaps a small number of closely related tasks. A component is not cohesive if its role is hard to define or if it performs a variety of loosely related tasks. Cohesion

One way to define cohesion is by saying that the capabilities of a class should be necessary and sufficient (like a condition in logic). The capabilities of a component are ***necessary*** if the system cannot manage without it. The capabilities of a class are ***sufficient*** if they jointly perform all the services that the client requires. 481

Necessary	=	every capability provided is in fact required by the system
	=	***no unused capabilities***
Sufficient	=	every capability required by the system is in fact provided
	=	***no missing capabilities***

The "necessary and sufficient" criterion is not the whole story. For example, we could write the entire system as one gigantic class that provided the required functionality and had no superfluous functions; clearly, this class would be both necessary and sufficient, but it would also be unmanageable and probably useless.

Sufficiency is a precise concept, but there is some leeway. For example, if it turns out that the sequence 482

```
p->f();
p->g();
```

occurs often in system code – perhaps, in fact, ***every*** invocation of `f` is followed by a call to `g` – then it probably makes sense to add a function `h` that combines `f` and `g` to the class. The old class was technically sufficient, but the new function improves clarity, simplifies maintenance, and may even make the program run a little faster.

Clearly there is no point in taking the time to code and test functions that will never be used. Nevertheless, there are programmers who like to spend large amounts of time writing code that "might be useful one day" rather than focusing on writing or improving code that was actually needed yesterday.

Consequently, cohesion also implies a division of the system into components of manageable size. A good guideline is that it should be possible to describe the role of a component with a single sentence. If you ask what a component does and the answer is either a mumble or a 10 minute peroration, there is probably something wrong with the design of the component.

Cohesion is related to coupling in the following way: a component that tries to do too much (that is, plays several roles) is likely to have many clients (perhaps one or more for each role). It may also need to be a client of several other components in order to perform its roles. Consequently, its coupling is likely to be high. On the other hand, a cohesive component is likely to have few clients and few dependencies, and therefore lower coupling.

There are always exceptions.

- A highly-cohesive component might provide an essential service to many parts of a system; in this case it would contribute to heavy coupling.
- Low cohesion in the design can sometimes be corrected by careful physical design. For example, a group of related classes could be put into a single component, exposing only some class interfaces to the rest of the system.

12.5 Coupling

Coupling is any form of dependency between components.

483

Coupling is not an all-or-nothing phenomenon: there are ***degrees*** of coupling. At one end of the scale, if a component has no coupling with the rest of the system, it cannot be doing anything useful and should be thrown away. It follows that ***some*** coupling is essential and therefore that the issue is ***reducing*** coupling, not eliminating it.

The other end of the scale is very high coupling: every component of the system is strongly coupled to every other component. Such a system will be very hard to maintain, because a change to one component often requires changes to other components. In the following, we look at different techniques for reducing coupling.

12.5.1 Encapsulation

The first step towards low coupling is ***encapsulation***, which means hiding the implementation
484 details of an object and exposing only a well-defined public interface. Figure 158 is a first (rather
feeble) attempt at defining a class for points with two coordinates.

```
class Point1
{
public:
    double x;
    double y;
};
```

Figure 158: A class for points: version 1

Class `Point1` provides very poor encapsulation: anyone can get and set its coordinates, and it
has no control over its state at all. Obviously, we should make the coordinate data `private`, but
485 then we would have to provide some access functions, as shown in Figure 159 on the next page.

Access functions that provide no checking, like these, are not much better than `public` data.
But access functions do at least provide the ***possibility*** of controlling state and maintaining
486 class invariants. For example, `setX` might be redefined as

```
class Point2
{
public:
    double getX() { return x; }
    double getY() { return y; }
    void setX(double nx) { x = nx; }
    void setY(double ny) { y = ny; }
private:
    double x;
    double y;
};
```

Figure 159: A class for points: version 2

```
void Point2::setX(double nx)
{
    if (nx < X_MIN) nx = X_MIN;
    if (nx > X_MAX) nx = X_MAX;
    x = nx;
}
```

487 Access functions provide a way of hiding the representation of an object. For example, Figure 160 shows how we could define points using complex numbers without affecting users in any way (except, perhaps, efficiency).

```
class Point3
{
public:
    double getX() { return z.re; }
    double getY() { return z.im; }
    void setX(double nx) { z.re = nx; }
    void setY(double ny) { z.im = ny; }
private:
    complex<double> z;
};
```

Figure 160: A class for points: version 3

Even with access functions, however, `Point` hardly qualifies as an object. Why do users want points? What are the operations that points should provide? A class for points should look more like Figure 161 on the following page, in which function declarations appear in the class declaration, but function definitions are in a separate implementation file. 488

`Point4` has less coupling than `Point3` in another respect. If any of the inline functions of `Point3` are changed, its clients will have to be recompiled. Since the functions of `Point4` are defined in `Point4.cpp` rather than `Point4.h`, they can be changed without affecting clients (although the system will have to be re-linked, of course).

```
class Point4
{
public:
    void draw();
    void move(double dx, double dy);
    ....
private:
    // Hidden representation
};
```

Figure 161: A class for points: version 4

A client of one of the `Point`n classes will be coupled to the `Point`n component. The degree of coupling depends on the `Point` class: it is highest for `Point1` (the user has full access to the coordinates) and lowest for `Point4` (the user can manipulate points only through functions such as `move` and `draw`).

From the point of view of the owner of the `Point` class, lower coupling means greater freedom. If the owner of `Point1` makes almost any change at all, all clients will be affected. The owner of `Point4`, on the other hand, can change the representation of a point, or the definitions of the member functions, without clients needing to know, provided only that the class continues to meet its specification. This is one advantage of low coupling:

Low coupling makes maintenance easier.

A complicated object is entitled to have a few access functions but, in general, a class should ***keep a secret***.[67] If your class seems to need a lot of `get` and `set` functions, you should seriously consider redesigning it.

Here are formal definitions (Lakos 1996, pages 105 and 138):

1. The ***logical interface*** of a component is the part that is programmatically accessible or detectable by a client. 489
2. An implementation entity (type, variable, function, etc.) that is not accessible or detectable programmatically through the logical interface of a component is ***encapsulated*** by that component. encapsulation

In set theoretic notation, for any component C

$$I_C \cup E_C = U \qquad \text{and} \qquad I_C \cap E_C = \emptyset$$

where I_C is the set of implementation entities in the logical interface of C, E_C is the set of entities encapsulated by C, and U is the "universe" of all entities in C.

[67] The useful metaphor "keeping a secret" was introduced by David Parnas in a very influential paper (Parnas 1978).

12.5.2 Hidden coupling

C++ provides various ways in which coupling between components can be hidden: that is, there
is a dependency between two components even though neither `#include`s the other's header file.
For example, the following components `A` and `B` are coupled: 490

```
// file A.cpp                      // file B.cpp

int numWidgets;                    extern int numWidgets;

....                               ....
```

Using the definitions of the previous section, we note that `numWidgets` is in the logical interface
of component `A`, because it can be accessed by component `B` by using `extern`. extern

The object file obtained when `A.cpp` is compiled will allocate memory for `numWidgets`. The object file obtained when `B.cpp` is compiled will not allocate memory for `numWidgets` but will expect the linker to provide an address for `numWidgets`. If the definition in `A.cpp` is changed or removed, both components will compile, but the system won't link.

Don't use global variables.

Don't use `extern` ***declarations.***

12.5.3 Compilation dependencies

Coupling can affect the time needed to rebuild a system. Build times are significant for large
projects, and it is useful to know how to keep them low. If we define class `Point5` as in Figure 162, 491
`sizeof(Point5)` gives 16 bytes, showing that the compiler allocates two 8-byte `double` values to
store a `Point5`. If we decide to add a new data member, `d`, to store the distance of the point from
the origin, we obtain `Point6`, shown in Figure 163 on the next page. Then `sizeof(Point6)` gives 492
24 bytes, showing that the compiler allocates three 8-byte `double` values to store a `Point6`.

```
class Point5
{
private:
    double x;
    double y;
};
```

Figure 162: A class for points: version 5

At this point, we might decide that our class needs a function, as shown in Figure 164. Adding a 493
function has ***no effect*** on the size of the class: `sizeof(Point7)` is 24 bytes, just like `Point6`.

But if we make the function `virtual`, as in Figure 165 on the following page, then `sizeof(Point8)` 494

```
class Point6
{
private:
    double x;
    double y;
    double d;
};
```

Figure 163: A class for points: version 6

```
class Point7
{
public:
    void move(double dx, double dy) { x += dx; y += dy; }
private:
    double x;
    double y;
    double d;
};
```

Figure 164: A class for points: version 7

gives 32 bytes, because a virtual function requires a ***virtual function table*** or ***vtable*** for short.[68]

```
class Point8
{
public:
    virtual void move(double dx, double dy) { x += dx; y += dy; }
private:
    double x;
    double y;
    double d;
};
```

Figure 165: A class for points: version 8

In summary, adding or removing data members changes the size of the objects. Adding a virtual function also changes the size. (Adding more virtual functions after the first makes no difference, because there is only one vtable and it contains the addresses of all virtual functions.)

The changes we have been discussing should not affect users of the class. Private data members cannot be accessed anyway, and whether a function is virtual affects only derived classes. Nevertheless, if we change the size of a `point`, ***every component that includes*** `point.h`

[68] The pointer to the vtable needs 4 bytes on a 32-bit Intel architecture. The additional 4 bytes may be added for alignment purposes.

will be recompiled during the next build! Since building a large project can take hours or even days, changing a class declaration, even the `private` part, is something best avoided (see Figure 166[69]).

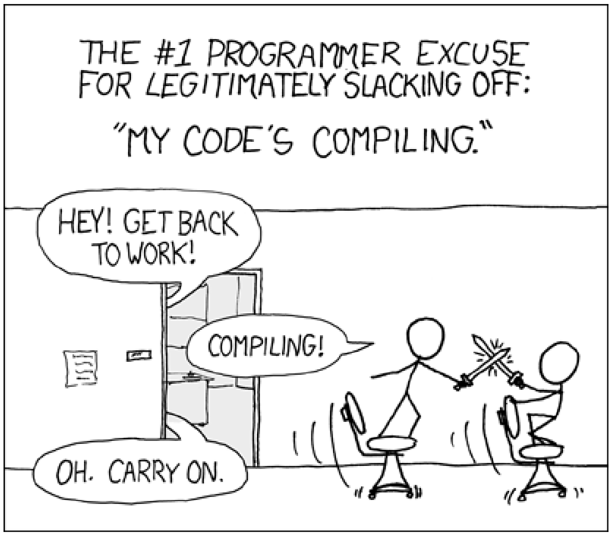


Figure 166: The importance of reducing compile time

To see why this happens, consider the first program in Figure 167. To allocate stack space for `p`, the compiler must be able to compute `sizeof(Point)`. To compute this, it must see the declaration of class `Point`. Finally, to see the declaration, it must read `"point.h"`. If `"point.h"` changes, the program must be recompiled. 495

```
#include "point.h"

int main()
{
    Point p;
    ....
}
```

Figure 167: Forward declarations: 1

The program in Figure 168 on the following page is slightly different. The compiler does ***not*** need to know `sizeof(Point)` to compile this program, because all pointers and references have the same size (the size of an address: usually 4/8 bytes on a 32bit/64bit system). Consequently, reading the declaration of class `Point` is a waste of time. However, the compiler ***does*** need to know that `Point` is a class and, for this purpose, the forward declaration of Figure 169 is sufficient. 496

forward declarations

497

Don't include a header file when a forward declaration is all you need.

[69] Copyright Randall Munroe, http://xkcd.com/303/, licensed under a Creative Commons Attribution-NonCommercial 2.5 License

```
#include "point.h"

int main()
{
    Point* pp;
    ....
    void f(Point & p);
    ....
}
```

Figure 168: Forward declarations: 2

```
class Point; // forward declaration

int main()
{
    Point* pp;
    ....
    void f(Point & p);
    ....
}
```

Figure 169: Forward declarations: 3

498 In early versions of the standard libraries, names such as `ostream` referred to actual classes. Consequently, the code shown in Figure 170 on the next page worked well in a header file. Times have changed, and `ostream` is now defined by `typedef`. However, the standard library defines a header file that contains forward declarations for all stream classes: see Figure 171 on the facing page.

499 Suppose that points have colours. It is tempting to put the enumeration declaration outside the class declaration:

```
enum Colour { RED, GREEN, BLUE };

class Point
{
    ....
```

This is convenient, because we can use the identifiers `RED`, `GREEN`, and `BLUE` wherever we like. But this convenience is also a serious drawback: the global namespace is ***polluted*** by the presence of these new names. It is much better to put the enumeration ***inside*** the class declaration, like this:

```
class Point
{
public:
    enum Colour { RED, GREEN, BLUE };
    ....
```

```
class ostream;
....
friend ostream & operator<<(ostream & os, const Widget & w);
```

Figure 170: Forward declaration for `ostream`: 1

```
#include <iosfwd>
....
friend ostream & operator<<(ostream & os, const Widget & w);
```

Figure 171: Forward declaration for `ostream`: 2

We now have to write `Point::RED` instead of `RED`, but there is no longer any danger of the colour names clashing with colour names introduced by someone else.

In situations where class declarations alone are inadequate, for example in the development of a library, we can use namespaces instead (which are discussed in Section 12.3 on page 256).

12.5.4 Cyclic Dependencies

The worst kind of dependencies are ***cyclic dependencies***. When a system has cyclic dependencies, "nothing works until everything works". Some cyclic dependencies are unavoidable, but many can be eliminated by careful design.

Suppose we have classes `Foo` and `Bar` that are similar enough to be compared. We might write 500

```
class Foo
{
public:
    operator==(const Bar & b);
    ....
};

class Bar
{
public:
    operator==(const Foo & f);
    ....
};
```

This creates a cyclic dependency between `Foo` and `Bar`. The dependency can be eliminated by using `friend` functions and forward declarations:

In file `foo.h`: 501

```
class Bar;

class Foo
{
    friend operator==(const Foo & f, const Bar & b);
    friend operator==(const Bar & b, const Foo & f);
    ....
};
```

502 In file `bar.h`:

```
class Foo;

class Bar
{
    friend operator==(const Foo & f, const Bar & b);
    friend operator==(const Bar & b, const Foo & f);
    ....
};
```

503 In an implementation file (`foo.cpp`, or `bar.cpp`, or somewhere else):

```
operator==(const Foo & f, const Bar & b) { .... }
operator==(const Bar & b, const Foo & f) { .... }
```

12.5.5 Inlining

Normally, a function is called by evaluating its arguments, placing them in registers or on the stack, storing a return link in the stack, and passing control to the function's code. When the function returns, it uses the return link to transfer control back to the call site. There is clearly a fair amount of overhead, especially if the function is doing something trivial, such as returning the value of a data member of a class.

If the compiler has access to the definition of a function, it can compile the body of the function directly, without the call and return. This is called ***inlining*** the function.

An inline function will usually be faster than a called function, although the difference will be significant only for small functions. (For large functions, especially if they have loops or recursion, the time taken by calling and returning will be negligible compared to the time taken to execute the function.)

Heavy use of inlined functions increases the size of the code, because the code of the function is compiled many times rather than once only. Thus inlining is an example of a ***time/space tradeoff***.

Inlining is relevant to this section because ***inlining affects coupling***. The compiler can inline a function only if its definition is visible. Within an implementation file, a definition of the form

```
inline void f() { .... }
```

permits the compiler to inline `f`, but only in that particular implementation file. To inline a function throughout the system, the `inline` qualifier, and the body of the function, must appear in a header file. As we have seen, this implies that any change to the function body will force recompilation of all clients.

```
#ifndef WIDGET_H
#define WIDGET_H

class Widget
{
public:
    int f1() { return counter; }
    int f2();
private:
    int counter;
};

inline double f3()
{
    ....
}

char f4();
```

Figure 172: Inlining

Whenever a function definition appears inside a class declaration, the compiler is allowed to compile it inline. In Figure 172, the compiler may inline `f1` and `f3` and it cannot inline `f2` and `f4`. 504

The qualifier `inline` is a ***compiler hint***, not a directive. The compiler is not obliged to inline functions that are defined inside the class declaration or declared `inline`, and it may inline functions of its own accord.

Do not place too much dependence on inlining: the rewards are not great and there may be drawbacks. Herb Sutter (Sutter 2002, pages 83–86) argues that good compilers know when and when not to inline and programmers should leave the choice to them. Scott Meyers (Meyers 2005, Item 30) discusses inlining in more detail.

12.6 The Pimpl Idiom

A complicated class might have quite a large `private` part. Every time the owner of the class changes the class declaration, every client gets recompiled. A "Pimpl" is a simple way of avoiding this dependence: it is a mnemonic for ***pointer to implementation***.

We will use a slightly modified version of the class `Account` from Section 6 on page 99 to illustrate the Pimpl idiom. We will develop three versions of this class, requiring each one of them to execute the test program shown in Figure 173 on the following page, giving the results 505

506

```
#include "account.h"

Account cbv(Account val)
{
   val.withdraw(2000);
   cout << val << endl;
   return val;
}

void cbr(Account & val)
{
   val.withdraw(2000);
   cout << val << endl;
}

int main()
{
    Account acc("Fred", 10000);
    cout << acc << endl;
    acc.deposit(2000);
    cout << acc << endl;
    acc.withdraw(3000);
    cout << acc << endl;
    cout << "Assign\n";
    Account cop = acc;
    cout << "CBV\n";
    cbv(acc);
    cout << "CBR\n";
    cbr(cop);
    return 0;
}
```

Figure 173: Test program for class `Account`

of Figure 174.

507 The ID numbers in the output show that passing by value and returning by value cause new copies of the account to be created, but passing by reference does not cause any creation.

508 509 Figure 175 on the next page shows the header file of the original class `Account`. There are a few small differences between this version and the earlier version:

- We use a `std::string` to represent the name of the account (like in the version used in the inheritance lecture).

- Unique ID numbers are generated automatically using a `static` counter.

510 511 512 513 Figures 176 and 177 show the implementation of class `Account`. Note that each constructor

```
Fred        1  100.00
Fred        1  120.00
Fred        1   90.00
Assign
CBV
Fred        3   70.00
CBR
Fred        2   70.00
```

Figure 174: Output from the program of Figure 174

```
class Account
{
public:
   Account();
   Account(std::string name, long balance = 0);
   Account(const Account & other);
   Account & operator=(const Account & other);
   const std::string getName() const;
   void deposit(long amount);
   void withdraw(long amount);
   void transfer(Account & other, long amount);
   friend std::ostream & operator<<(std::ostream & os, const Account & acc);

private:
   static long idGen;
   long id;
   std::string name;
   long balance;
};

bool operator==(const Account & left, const Account & right);
bool operator!=(const Account & left, const Account & right);
```

Figure 175: Header file for original class `Account`

increments the account ID. Assignment also increments the ID and the insert `operator<<` displays this value.

The definitions for the comparison operators, `operator==` and `operator!=`, will not be shown again because they do not change.

12.6.1 Introducing Pimpl

Converting this class to a Pimpl class requires changing the head and implementation files. Figure 178 on page 272 shows the new header file. Two changes are required: 514

```
long Account::idGen = 0;

Account::Account()
      : id(++idGen), name(""), balance(0) {}

Account::Account(string name, long balance)
      : id(++idGen), name(name), balance(balance) {}

Account::Account(const Account & other)
      : id(++idGen), name(other.name), balance(other.balance) {}

Account & Account::operator= (const Account & other)
{
   if (this == &other)
      return *this;
   id = ++idGen;
   name = other.name;
   balance = other.balance;
   return *this;
}

const string Account::getName() const
{
   return name;
}

void Account::deposit(long amount)
{
   balance += amount;
}

void Account::withdraw(long amount)
{
   if (amount > balance)
      cerr << "Withdrawal greater than balance.\n";
   else
      balance -= amount;
}

void Account::transfer(Account & other, long amount)
{
   withdraw(amount);
   other.deposit(amount);
}
```

Figure 176: Implementation for class `Account`: part 1

```
ostream & operator<<(ostream & os, const Account & acc)
{
   return os <<
          left << setw(8) << acc.name << right <<
          setw(4) << acc.id <<
          fixed << setprecision(2) << setw(8) << acc.balance / 100.0;
}

bool operator==(const Account & left, const Account & right)
{
   return &left == &right;
}

bool operator!=(const Account & left, const Account & right)
{
   return !(left == right);
}
```

Figure 177: Implementation for class `Account`: part 2

- The implementation class `AccImpl` is declared before class `Account`.
- The private data of class `Account` is replaced by a pointer to an instance of `AccImpl`.

It would be nice to keep the existence of class `AccImpl` private, but the compiler gives errors if the declaration is moved to the `private` section of `Account`.

Figures 179, 180, and 181 show the new implementation file for `Account`.It has two parts: the 515
declaration for class `AccImpl`, and the new function definitions for class `Account`. 516

Class `AccImpl` is very similar to the original class `Account`. It has the same private data and the 517
same functions. We have added messages in the constructors and destructor to show that these 518
functions are called in the correct sequence. In the hope of getting back some of the efficiency
that we have lost by Pimpl'ing, all of the functions are inlined. 519

Insertion (`operator<<`) expects a pointer to an instance of `AccImpl`.

The next step is to define new versions of the functions of `Account`, as shown in Figure 181 on
page 275. The constructors create a new instance of `AccImpl` and initialize it appropriately. The 520
copy constructor is inefficient: it creates a default `AccImpl` and then copies information from the 521
other `Account` object into it. The destructor is required because the implementation object is
created dynamically and must be destroyed. 522

Assignment (`operator=`) is implemented in the same way as the copy constructor. 523

The remaining functions simply forward the original request to the implementation: function `f()` is implemented as `pimpl->f()`.

Insertion (`operator<<`) passes the Pimpl pointer to the version of `operator<<` defined for `AccImpl`.

Figure 182 on page 275 shows the output generated by the test program with the Pimpl'ed class. 524

```
class AccImpl;

class Account
{
public:
   Account();
   Account(std::string name, long balance = 0);
   ~Account();
   Account(const Account & other);
   Account & operator=(const Account & other);
   const std::string getName() const;
   void deposit(long amount);
   void withdraw(long amount);
   void transfer(Account & other, long amount);
   friend std::ostream & operator<<(std::ostream & os, const Account & acc);

private:
   AccImpl *pimpl;
};
```

Figure 178: Header file for Pimpl class `Account`

The additional lines show that the `AccImpl` objects are being created and destroyed correctly.

12.6.2 Combining Pimpl and autopointers

As a variation on the Pimpl idiom, we can use an autopointer instead of a normal pointer.
525 Figure 183 on page 276 shows the revised header for class `Account`.

The implementation of `AccImpl` does not change, so we do not show it. Nor do we show the
unchanged functions of class `Account`. Figure 184 on page 277 shows the functions that are
526 changed. The constructors and `operator=` create instances of `AccImpl` and set autopointers
527 to them. The destructor is retained, but only because it displays a message: it is no longer
necessary.

As expected, the output generated when the autopointer version is executed is the same as the output generated by the pointer version (Figure 182 on page 275).

The Pimpl idiom trades performance for compilation speed. The program executes slightly more slowly, because functions are called indirectly, but compiles faster.

The improvement in compilation time can be dramatic. Herb Sutter (Sutter 2000, page 110) says:

> *"I have worked on projects in which converting just a few widely-used classes to use Pimples has halved the system's build time."*

The ***Bridge design pattern*** (Section 13.5 on page 298) is an extension of the Pimpl idea. For more details on Pimpl, in particular using C++11 features, see (Meyers 2014, Item 22).

```
#include "account.h"

class AccImpl
{
public:
   AccImpl() : name(""), id(++idGen), balance(0)
   {
      cerr << "create " << id << endl;
   }

   AccImpl(std::string name, long balance = 0)
         : name(name), id(++idGen), balance(balance)
   {
      cerr << "create " << id << endl;
   }

   ~AccImpl()
   {
      cerr << "delete " << id << endl;
   }

   AccImpl & operator=(const AccImpl & other)
   {
      name = other.name;
      balance = other.balance;
   }

   const std::string getName() const
   {
      return name;
   }

   void deposit(long amount)
   {
      balance += amount;
   }

   void withdraw(long amount)
   {
      if (amount > balance)
         cerr << "Withdrawal amount greater than balance.\n";
      else
         balance -= amount;
   }

   void transfer(AccImpl & other, long amount)
   {
      withdraw(amount);
      other.deposit(amount);
   }
```

Figure 179: Implementation file for Pimpl class `Account`: part 1

```
   friend ostream & operator<<(ostream & os, AccImpl *p)
   {
      return os <<
             left << setw(8) << p->name << right <<
             setw(4) << p->id <<
             fixed << setprecision(2) << setw(8) << p->balance / 100.0;
   }

private:
   static long idGen;
   std::string name;
   long id;
   long balance;
};

long AccImpl::idGen = 0;

Account::Account()
{
   pimpl = new AccImpl();
}

Account::Account(std::string name, long balance)
{
   pimpl = new AccImpl(name, balance);
}

Account::Account(const Account & other)
{
   pimpl = new AccImpl();
   *pimpl = *(other.pimpl);
}

Account::~Account()
{
   delete pimpl;
}

Account & Account::operator=(const Account & other)
{
   if( this == &other )
       return *this;
   delete pimpl;
   pimpl = new AccImpl();
   *pimpl = *(other.pimpl);
   return *this;
}
```

Figure 180: Implementation file for Pimpl class `Account`: part 2

```
const std::string Account::getName() const
{
   return pimpl->getName();
}

void Account::deposit(long amount)
{
   pimpl->deposit(amount);
}

void Account::withdraw(long amount)
{
   pimpl->withdraw(amount);
}

void Account::transfer(Account & other, long amount)
{
   pimpl->transfer(other.pimpl, amount);
}

ostream & operator<<(ostream & os, const Account & acc)
{
   return os << acc.pimpl;
}
```

Figure 181: Implementation file for Pimpl class `Account`: part 3

```
create 1
Fred       1  100.00
Fred       1  120.00
Fred       1   90.00
Assign
create 2
CBV
create 3
Fred       3   70.00
create 4
delete 4
delete 3
CBR
Fred       2   70.00
delete 2
delete 1
```

Figure 182: Output from the Pimpl version of class `Account`

```
#include <memory>

class AccImpl;

class Account
{
public:
   Account();
   Account(std::string name, long balance = 0);
   Account(const Account & other);
   ~Account();
   Account & operator=(const Account & other);
   const std::string getName() const;
   void deposit(long amount);
   void withdraw(long amount);
   void transfer(Account & other, long amount);
   friend std::ostream & operator<<(std::ostream & os, const Account & acc);

private:
   std::auto_ptr<AccImpl> pimpl;
};
```

Figure 183: Header file for autopointer Pimpl class `Account`

12.7 Refactoring

When we change a system to improve its maintainability, performance, or other characteristics ***without changing its functionality***, we are ***refactoring*** it. A common reason for refactoring is to increase cohesion and decrease coupling. Refactoring is a large topic and entire books have been written about it – for example, (Fowler, Beck, Brant, Opdyke, and Roberts 1999; Kerievsky 2004).[70]

> *"... merciless refactoring of existing code ... greatly decreases the sort of chaos I've seen in "clean and simple" code. It's almost like an emergent property, and I can't quite explain what's going on. The code becomes more fluid. The chunks are smaller so they have less trouble moving them to where they ought to be...*
>
> *Like design patterns, refactoring codifies wisdom. This wisdom is about what good code looks like. I've encountered most of the refactorings in my professional life, but that was over the course of many years. I envy programmers starting out today with this sort of wisdom at their fingertips. Established programmers like me didn't even realize this was something that needed codifying."* (Eric Hodges[71])

Here are brief sketches of some simple refactorings.

[70] See also the Refactoring Home Page, http://www.refactoring.com.

[71] See http://ootips.org/refactoring.html

```
Account::Account()
{
   pimpl = auto_ptr<AccImpl>(new AccImpl());
}

Account::Account(std::string name, long balance)
{
   pimpl = auto_ptr<AccImpl>(new AccImpl(name, balance));
}

Account::Account(const Account & other)
{
   pimpl = auto_ptr<AccImpl>(new AccImpl());
   *pimpl = *(other.pimpl);
}

Account::~Account()
{
   cerr << "delete " << id << endl;
}

Account & Account::operator=(const Account & other)
{
   if( this != &other ) {
       pimpl = auto_ptr<AccImpl>(new AccImpl());
       *pimpl = *(other.pimpl);
   }
   return *this;
}
```

Figure 184: Implementation file for autopointer Pimpl class `Account`

12.7.1 Simplify Calling Patterns

Browsing through source code (the first essential step for ***any*** refactoring), you notice these two lines: 528

```
double xFactor = minPlay(x, y);
target.setParams(xFactor, 997.3);
```

Looking further, you notice that these lines occur frequently in the code, with slight variations in parameters.

You could refactor the code by ***wrapping*** this pair into a single function:

```
void setp(Object target, double x, double y, double scale)
{
   double xFactor = minPlay(x, y);
   target.setParams(xFactor, scale);
}
```

It might be even better to add a new member function to class `Object` and then replace each pair of lines by

```
target.revisedSetParams(x, y, scale);
```

12.7.2 Introduce Design Patterns

Knowing design patterns helps refactoring. Browsing source code, you notice a group of three classes, `A`, `B`, and `C` that are tightly coupled (e.g., each class contains pointers to the others and uses their methods). Uses of these classes elsewhere in the code is complex and confusing: 529

```
pa->f();      // May change B and C
pb->g();      // May change A
....
```

Façade

Using the ***Façade*** pattern (Gamma, Helm, Johnson, and Vlissides 1995), you could create a fourth class, `D`, that provides an interface to the three classes `A`, `B`, and `C`. Class `D` might look something like this:

```
class D
{
public:
   D(A* pa, B* pb, C* pc) : pa(pa), pb(pb), pc(pc) : {}
   void f() { pa->f(); }
   void g() { pb->g(); }
   void h() { pa->f(); pb->g(); }
   ....
private:
   A* pa;
   B* pb;
   C* pc;
};
```

This class doesn't look very useful and even seems rather inefficient, since all it is doing is forwarding calls. However, it simplifies code ***elsewhere in the application***, where only the class `D` is visible and calling patterns are simplified. Sometimes, it is possible to put code in `D` to ensure that the other classes are used appropriately.

Design patterns are discussed in more detail later (Lecture 13).

12.7.3 Encapsulate

Early in development, a data entity seemed so simple that the programmer elected to use a `struct`: 530

```
struct Point { double x, double y, double z; };
```

As the software grows, many calculations with points are added. Redundancy appears: for example, two programmers independently discover that they need to compute the distance between two points and add equivalent, but slightly different functions.

The obvious refactoring is to create a class `Point` and to put as many point-related functions into it.

Experienced programmers avoid this problem by not using `struct` in the first place: unless the data is ***really*** simple and is not associated with ***any*** calculations, a `class` is better than a `struct`.

12.7.4 Move Common Code to the Root

Class hierarchies tend to start simple and grow complex. Reviewing a mature class hierarchy often shows that common problems have been solved independently in different derived classes. Find commonalities and move them up the tree, to the root if possible.

12.7.5 Global Names

Global names should be avoided wherever possible, but any complex application is likely to contain a few of them. Global names declared in odd places and imported indiscriminately will cause confusion. Create a class called `GlobalNames` or something similar, and put all global objects into it. Any component that uses globals will then clearly announce that it does so by starting with

```
#include "GlobalNames.h"
```

In modern programming, global names are typically handled through ***dependency injection***.

12.8 Miscellaneous Techniques

There are several techniques for developing well-structured systems or improving systems with structural weaknesses.

12.8.1 CRC Cards

CRC cards were introduced by Kent Beck and Ward Cunningham at OOPSLA (Beck and Cunningham 1989) for teaching programmers the object-oriented paradigm. A CRC card is an index card that is used to represent the responsibilities of classes and the interaction between the classes. The cards are created through scenarios, based on the system requirements, that model the behavior of the system. The name CRC stands for ***Class***, ***Responsibilities***, and ***Collaborators***.[72]

Although CRC cards are an old methodology and have largely been replaced by heavy-weight methods such as UML, the underlying ideas are still useful in the early stages of system design. The focus on responsibility and collaboration is appropriate and helps to improve the coupling/cohesion ratio.

12.8.2 DRY

DRY stands for ***don't repeat yourself***. As systems grow, it is easy for them to accumulate multiple implementations of a single function. These functions might vary in small details, such as the number and order of parameters, but they do essentially the same thing. Weed out such repetition by converting all the variations into a single function whenever you can. As (Hunt and Thomas 2000, page 27) say:[73]

> ***Every piece of knowledge must have a unique, unambiguous, authoritative representation within a system.***

12.8.3 YAGNI/KISS

YAGNI stands for ***you aren't going to need it*** and is a slogan associated with agile methods. It is a variant of KISS (Keep It Simple, Stupid). Some programmers are tempted to provide all kinds of features that might come in useful one day. When writing a class, for instance, they will include a lot of methods but they don't actually need right now for the application, but look nice. This is usually a waste of time: there is a good chance that the functions will not in fact be needed and, if they are, it does not take long to write them.

It is tempting to write code that is not necessary right now but (you think) might be needed later. Giving in to this temptation has some disadvantages:[74]

- The time spent is taken from necessary functionality.
- The new features must be debugged, documented, and supported.
- Any new feature imposes constraints on what can be done in the future, so an unnecessary feature now may prevent implementing a necessary feature later.

[72] See http://ootips.org/crc-cards.html

[73] I changed their word "single" to "unique" for greater emphasis.

[74] See http://en.wikipedia.org/wiki/You_ain't_gonna_need_it

- Until the feature is actually needed it is not possible to define what it should do, or to test it. This often results in such features not working right even if they eventually are needed.
- It leads to code bloat; the software becomes larger and more complicated while providing no more functionality.
- Unless there are specifications and some kind of revision control, the feature will never be known to programmers who could make use of it.
- Adding the new feature will inevitably suggest other new features. The result is a snowball effect which can consume unlimited time and resources for no benefit.

References

Beck, K. and W. Cunningham (1989, October). A laboratory for teaching object-oriented thinking. In *Object-Oriented Programming: Systems, Languages and Applications*, pp. 1–6.

Dewhurst, S. C. (2005). *C++ Common Knowledge: Essential Intermediate Programming*. Addison-Wesley.

Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison-Wesley.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Hunt, A. and D. Thomas (2000). *The Pragmatic Programmer*. Addison-Wesley.

Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley.

Lakos, J. (1996). *Large-Scale C++ Software Design*. Professional Computing Series. Addison-Wesley.

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley.

Meyers, S. (2014). *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* (1st ed.). O'Reilly.

Parnas, D. L. (1978). Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd International Conference on Software engineering*, Piscataway, NJ, USA, pp. 264–277. IEEE Press. Reprinted as (Parnas 1979).

Parnas, D. L. (1979, March). Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 128–138.

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.

Sutter, H. (2000). *Exceptional C++: 47 Engineering puzzles, Programming Problems, and Solutions*. C++ In-Depth Series. Addison-Wesley.

Sutter, H. (2002). *More Exceptional C++: 40 New Engineering puzzles, Programming Problems, and Solutions*. C++ In-Depth Series. Addison-Wesley.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.

13 Using Design Patterns

13.1 Singleton 283
13.2 Composite 285
13.3 Visitor 289
13.4 State 293
13.5 Bridge 298
13.6 Command 300
13.7 Observer 300
13.8 Pattern Languages 307

Design patterns encode the knowledge of experienced designers. Patterns are design components that can be mapped into code for a particular application. A discussion of patterns in general is beyond the scope of this course; in this chapter, we look at some of the simpler patterns and show how they can be mapped into C++ code.

The "bible" of design patterns is the book *Design Patterns* (Gamma, Helm, Johnson, and Vlissides 1995). The patterns discussed in this chapter are all taken from this book. You should familiarize yourself with at least the GoF[75] patterns, so that you can recognize them when working on an existing code base or talking with experienced designers. Gang of Four

The examples in this chapter may seem to give unnecessarily complicated solutions to trivial problems. They may give you the feeling that patterns are a waste of time. However, patterns are not intended to solve trivial problems. Patterns are useful when similar, but more complex, problems arise in large-scale programming.

You've already seen one well-known design pattern, the ***Iterator***. In fact, by now you probably use it intuitively, without even thinking about it as a complex pattern. Using an iterator offers a number of advantages over traversing a data structure 'manually': With an iterator, you don't have to know the internal representation of the data structure to traverse it (and this representation can change without affecting the traversing clients), you can easily traverse the same structure in parallel (using different iterators), and you can use the same interface (that of the iterator) to traverse a multitude of very different data structures (e.g., list vs. tree vs. set). Another GoF pattern is the ***Façade*** we discussed in the previous lecture (see Section 12.7.2 on page 278). Iterator Façade

13.1 Singleton

Problem: Ensure that a class has only one instance at all times and provide a global point of access to it.

Solution: The problem has two parts: the first part is to ensure that only one instance of a class can exist; the second is to provide users with access to the unique instance. 532

[75] The four authors are often referred to as "The Gang of Four" or "GoF". Their book is sometimes called "The GoF Book".

```
class Singleton
{
public:
    static Singleton & instance();
    void fun();
private:
    int uid;
    static Singleton * pInstance;
    Singleton();
    Singleton(const Singleton &);
    Singleton & operator=(const Singleton &);
    ~Singleton();
};
```

Figure 185: Declaration for class `Singleton`

```
int globalUID = 77;

// Inaccessible pointer to the unique instance.
Singleton * Singleton::pInstance = 0;

// Private constructor.
Singleton::Singleton()
{
    uid = globalUID++;
}

// Public member function allows user to create the unique instance.
Singleton & Singleton::instance()
{
    if (!pInstance)
        pInstance = new Singleton;
    return *pInstance;
}

// Simple test function.
void Singleton::fun()
{
    cout << "I am Singleton " << uid << endl;
}
```

Figure 186: Definitions for class `Singleton`

Figure 185 on the facing page shows a declaration for class `Singleton` based on Alexandrescu's example (Alexandrescu 2001, pages 129–133). Here are some points of interest in this declaration:

- The function `fun` and the data member `uid` are included just to illustrate that the singleton can do something besides merely exist.
- The only way that a user can access the singleton is through the static reference function `instance`.
- Access to the unique instance of the singleton is through the `private`, `static` pointer, `pInstance`.
- The constructor is `private`. The copy constructor and assignment operator are `private` and unimplemented. Thus the singleton cannot be constructed (except for the first time), passed by value, assigned, or otherwise copied.
- The destructor is also `private` and unimplemented. This ensures that the singleton cannot be accidentally deleted. (This could be considered as a memory leak, but that is harmless because there is only one instance anyway.) 533

Figure 186 on the preceding page shows the implementation of class `Singleton`. The constructor is conventional; the body given here merely demonstrates that it works. The function `instance` constructs the singleton when it is called for the first time, and subsequently just returns a reference to it. A user must access the singleton object as `Singleton::instance()`. Any attempt to copy it or pass it by value causes a compile-time error. For the particular singleton defined here, all the user can do is apply the function `fun` to the singleton. Obviously, other functions would be added in a practical application. This is in fact a rather simple version of the Singleton pattern and it is not robust enough for all applications. For details about its improvement, see (Alexandrescu 2001, pages 133–156).

Figure 187 on the following page demonstrates the singleton in action. Running it will result 534
in: 535

```
I am Singleton 77
I am Singleton 77
I am Singleton 77
Inside f: I am Singleton 77
```

The first three lines of output show that there is only one instance; if more instances had been created, `globalUID` would have been incremented. The fourth line shows that the singleton has been passed as a reference. 536

13.2 Composite

Problem: Suppose we have a hierarchy of classes. We may also have collections of instances of these classes. A collection might be homogeneous (members all of the same class) or heterogeneous (members of different classes). The problem is to make these collections behave in the same way as elements of the class hierarchy. The collections are called *composite classes* and the pattern that solves the problem is called Composite.

```
void f(Singleton & s)
{
   cout << "Inside f: ";
   s.fun();
}

int main()
{
   Singleton::instance().fun();
   Singleton::instance().fun();
   Singleton::instance().fun();

   // A singleton can be passed by reference.
   f(Singleton::instance());

   // The following lines are all illegal.
   // Singleton s1;
   // Singleton s2 = Singleton::instance();

   return 0;
}
```

Figure 187: A program that tests class `Singleton`

Solution: To implement the Composite pattern, all we have to do is incorporate the composite classes into the class hierarchy. Since putting them into the hierarchy forces them to implement the base class interface, the effect will be to make their behaviour similar to that of other classes in the hierarchy.

The following example is a highly simplified typesetting application. The abstract base class
`Text` has two pure virtual functions: `set`, which "typesets" text to an output stream provided as
a parameter and `size`, which returns the size of a text. Initially, there are three derived classes,
537 `Blank`, `Character`, and `Word`, as shown in Figure 188 on the next page.

538 Suppose that we want to typeset paragraphs. Paragraphs are related to the classes we have in
539 that they should be able to implement the method `set`, but they are also different in that a
paragraph has many words or characters and is typically typeset between margins. Nevertheless,
540 we can add a class `Paragraph` to the hierarchy, as shown in Figure 189 on page 288.

Class `Paragraph` has a parameter in the constructor giving the width in which to set the paragraph. (A more realistic example would also allow this value to be set after construction.) We provide a function `addElem` that allows us to build paragraphs out of text elements (words, characters, and whatever else might be added later). These text elements are stored in a STL `vector`. The important points of this pattern are that class `Paragraph` inherits from class `Text` and implements `set` and `size`, making it a part of the hierarchy, while at the same time aggregating elements of its own superclass, `Text`.

541 Figure 190 on page 288 shows the implementation of the `Paragraph` function `set`. It uses a `stringstream` to store the text elements that make up a line. When there is not enough space

```
class Text
{
public:
   virtual void set(ostream & os) const = 0;
   virtual size_t size() const = 0;
};

class Blank : public Text
{
public:
   Blank() {}
   void set(ostream & os) const { os << ’ ’; }
   size_t size() const { return 1; }
};

class Character : public Text
{
public:
   Character(char c) : c(c) {}
   void set(std::ostream & os) const { os << c; }
   char getChar() const { return c; }
   size_t size() const { return 1; }
private:
   char c;
};

class Word : public Text
{
public:
   Word(const std::string & w) : w(w) {}
   void set(std::ostream & os) const { os << w; }
   string getWord() const { return w; }
   size_t size() const { return w.size(); }
private:
   string w;
};
```

Figure 188: Declarations for the text class hierarchy

on the line for the next text element (and the blank that precedes it), the string stream buffer is filled to `width` with blanks and written to the output stream.

Note that with this Composite pattern, we can now add a new text element to the hierarchy (e.g., an in-line graphic type) and the `set` function of a `Paragraph` will still work, since each new type must still implement `Text`’s functions `set` and `size`. Note that composite structures can be nested: A paragraph can have another paragraph as an element.

```
class Paragraph : public Text
{
public:
   Paragraph(int width) : width(width) {}
   void set(std::ostream & os) const;
   void addElem(Text* e) { elems.push_back(e); }
   size_t size() const {
      ostringstream oss;
      this->set(oss);
      return oss.str().size();
   }
private:
   vector<Text*> elems;
   size_t width;
};
```

Figure 189: A class for paragraphs

```
void Paragraph::set(ostream & os) const
{
   Blank b;
   ostringstream oss;
   for (vector<Text*>::const_iterator it = elems.begin();
                                      it != elems.end(); ++it)
   {
      if (size_t(oss.tellp()) + 1 + (*it)->size() > width)
      {
         while (size_t(oss.tellp()) < width)
            b.set(oss);
         os << oss.str() << endl;
         oss.str("");
         oss.clear();
      }
      if (size_t(oss.tellp()) > 0)
         b.set(oss);
      (*it)->set(oss);
   }
   os << oss.str() << endl;
}
```

Figure 190: Typesetting a `Paragraph`

13.3 Visitor 542

Problem: Figure 191 shows a grid of classes and operations that extend the `Text` hierarchy of the previous section.

	`Blank`	`Char`	`Word`	`Paragraph`	`Chapter`
`set`	⋆	⋆	⋆	⋆	⋆
`cap`	⋆	⋆	⋆	⋆	⋆
`print`	⋆	⋆	⋆	⋆	⋆
`count`	⋆	⋆	⋆	⋆	⋆

Figure 191: A grid of classes and operations

Note that a column corresponds to a class (that implements each operation) and a row corresponds to an operation (that must be implemented in each class). Each ⋆ indicates an action that must be implemented. Here are two ways of organizing the code that implements the operations:

1. In pre-object-oriented days, each operation was implemented as a single function with a
   big `switch` statement: see Figure 192. This made it easy to add a function, because all of 543
   the code would be in one place, but hard to add a class, because this would involve adding
   a clause to many different functions.

2. In an object-oriented language, we put the classes into a hierarchy; the root class of the hierarchy defines default or null versions of each function; and each class implements the functions in its own way. This design has two consequences:

   a) The code becomes fragmented: a function implements one operation in one class.

```
void set(....)
{
   switch(kind)
   {
   case BLANK:
      ....
      break;
   case CHAR:
      ....
      break;
   case WORD:
      ....
      break;
   ....
   }
}
```

Figure 192: Choosing operations using `switch-case`

b) It is easy to add a class, with an implementation for each operation, but hard to add an operation, because each class has to be modified.

c) If the objects are stored in a data structure, it is likely that each function will be responsible for traversing its part of the data structure.

For example, each time we add an operation to a class hierarchy such as `Text`, we have to add one function to each class in the hierarchy. Furthermore, every function we add to class `Paragraph` will iterate over the words in the paragraph.

The problem is to combine these two modes: we would like to retain the fragmented code, which is easier for maintenance, but we would like to organize it by operation rather than by class. We would also like to keep the organization of the data structure out of the individual functions.

Solution: The idea of the Visitor pattern is to define a "visitor" class that knows how to process each kind of object in the hierarchy. The visitor class is an abstract base class that cannot actually do anything useful, but it has derived classes for performing specific operations.

The following steps are required to support visitors in the `Text` hierarchy.

544 1. Create a base class `Visitor`: see Figure 193. This class has a virtual "visiting" function corresponding to each derived class in the `Text` hierarchy. Each visiting function is passed a pointer to an object of the corresponding type in the `Text` hierarchy.

2. Add functions to "accept" visitors in the `Text` hierarchy, starting with a pure virtual
545 function in the base class, as shown in Figure 194 on page 292. Most of these functions are
546 fairly trivial. For example:

```
void Blank::accept(Visitor & vis) { vis.visitBlank(this); }
void Character::accept(Visitor & vis) { vis.visitCharacter(this); }
```

For composite classes, function `accept` passes the visitor to each component individually:

```
void Paragraph::accept(Visitor & vis)
{
    for (  vector<Text*>::iterator it = elems.begin();
           it != elems.end(); ++it)
       (*it)->accept(vis);
}
```

```
class Visitor
{
public:
   virtual void visitBlank(Blank*) = 0;
   virtual void visitCharacter(Character*) = 0;
   virtual void visitWord(Word*) = 0;
   virtual void visitParagraph(Paragraph*) = 0;
};
```

Figure 193: The base class of the `Visitor` hierarchy

This completes the framework for visiting.

3. The next step is to construct an actual visitor. We will reimplement the typesetting function, set, as a visitor. For this, we need a class setVisitor derived from Visitor: see Figure 195 on the following page. 547

4. Finally, we implement the member functions of setVisitor, as in Figure 196 on the next page. It is not necessary to provide a body for setVisitor::visitParagraph because it is never called: when a Paragraph object accepts a visitor, it simply sends the visitor to each of its elements (see Paragraph::accept above). However, we do have to provide a trivial implementation in order to make Paragraph non-abstract. (Alternatively, we could have defined a trivial default function in the base class.) 548

5. To typeset a paragraph p, we call just: 549

```
p.accept(setVisitor());
```

To demonstrate the flexibility of the Visitor pattern, we can define another visitor that sets the same text in capital letters. The new class is called capVisitor: 550

```
class capVisitor : public Visitor
{
public:
   void visitBlank(Blank*);
   void visitCharacter(Character * pc);
   void visitWord(Word*);
   void visitParagraph(Paragraph*);
};
```

Its member functions are the same as those of setVisitor except for visitCharacter and visitWord: 551

```
void capVisitor::visitCharacter(Character * pc)
{
   char ch = pc->getChar();
   cout << toupper(ch);
}

void capVisitor::visitWord(Word * pw)
{
   string word = pw->getWord();
   for (string::iterator it = word.begin(); it != word.end(); ++it)
      *it = toupper(*it);
   cout << ' ' << word;
}
```

To invoke capVisitor, we simply construct an instance:

```
p.accept(capVisitor());
```

```
class Text
{
public:
   virtual void accept(Visitor & v) = 0;
   ....
};
```

Figure 194: Adding `accept` to the base class of the `Text` hierarchy

```
class setVisitor : public Visitor
{
public:
    void visitBlank(Blank*);
    void visitCharacter(Character * pc);
    void visitWord(Word*);
    void visitParagraph(Paragraph*);
};
```

Figure 195: A class for typesetting derived from `Visitor`

```
void setVisitor::visitBlank(Blank*)
{
    cout << ' ';
}

void setVisitor::visitCharacter(Character * pc)
{
    cout << pc->getChar();
}

void setVisitor::visitWord(Word * pw)
{
    cout << ' ' << pw->getWord();
}

void setVisitor::visitParagraph(Paragraph * pp)
{
   cout << "I should never be called";
}
```

Figure 196: Implementation of class `setVisitor`

For a single task (typesetting), setting up the Visitor classes seems rather elaborate. Suppose, however, that there were many operations to be performed on the `Text` hierarchy. Once we have defined the `accept` functions, we do not need to make any further changes to that hierarchy. Instead, for each new operation that we need, we derive a class from `Visitor` and define its "visit" functions for each kind of `Text`.

The visitor pattern has some disadvantages:

- If we use a separate function for each operation (like `set` in the original example), we can pass information around simply by adding parameters. Since the operations are independent, each can have its own set of parameters.

  The operations in the Visitor version, however, must all use the protocol imposed by `accept`. This means that they all get exactly the same parameters (none, in our example).

- The structure of composite objects is hard-wired into their `accept` functions. For example, `Paragraph::accept` iterates through its vector of `Text` elements. This makes it difficult to modify the traversal in any way.

  In the Composite example above, `Paragraph::set` inserted line breaks whenever the length of a line would otherwise have exceeded `width`. It is more difficult to provide this behaviour with the Visitor pattern, because `Paragraph::accept` does a simple traversal over the elements and does not provide for any additional actions. Nor is it possible to pass an argument to `Paragraph::accept` giving the position on the current line.

The Visitor pattern implements a form of ***double dispatch***. In C++, like in most object-oriented languages, the function to call depends on only two things: the *receiving object* and the *method* in that object. With double dispatch, the function to execute also depends on the *caller* (or sender) of said message.

double dispatch

13.4 State

Problem: An object has various states and behaves in different ways depending on its state.
We could write a `switch` statement to capture this property, as shown in Figure 197 on the 552
following page. This solution is unsatisfactory because any changes – modification of a state's 553
behaviour, adding or removing states, etc. – requires changes to this statement.

Solution: The State pattern is often implemented by inheritance and dynamic binding. The base class, which may be abstract, defines the action corresponding to a default state or no action at all. Each derived class defines the action for a particular state. The following example demonstrates the State pattern with a scanner whose state determines the text that it recognizes.

The abstract base class `Scanner` shown in Figure 198 on page 295 specifies that any scanner 554
object must accept a character pointer and return a `string`. Note that the pointer is passed by 555
reference, because a scanner will advance the pointer over the buffer and must return its new
value to the caller. The result of scanning is the string representing the token scanned. For 556
example, a number scanner might return the string `"123.456"`. Figure 198 on page 295 also
shows derived classes that scan white space, numbers, and identifiers.

The scanners may be selected in various ways. Figure 199 on page 296 shows a simple managing 557

558

559

class. An instance of ScanManager has a pointer to each kind of scanner; its constructor creates the corresponding objects dynamically and its destructor deletes them. The function ScanManager::scan is given a pointer to a character array and it uses the character to choose a
560 scanner object. Figure 200 on page 297 shows a simple example of the scanner in use, resulting in:

```
Scan succeeded: <123> <456> <Pirate456> <789> <anotherID>
```

Remarks:

- The nice thing about the State pattern is that it localizes the behaviour corresponding to a given state. There are many simple classes rather than a possibly huge switch statement or equivalent. It is easy to modify a state without looking at, or even knowing about, other states. Adding a state is also easy and does not require interfering with existing code.

- In this example, there is a single controlling object, the ScanManager, that handles all the state changes. This makes the State pattern look a bit pointless. However, there are other ways of changing state:

  – Each derived class might be responsible for choosing its successor state. A drawback of this approach is that each derived class has to know about one or more other derived classes, which goes against the usual practice of keeping derived classes independent of each other.

```
class StateChanger
{
public:
   void act()
   {
      switch (state)
      {
      case RUNNING:
         // ....
         break;
      case STUCK:
         // ....
         break;
      case BROKEN:
         // ....
         break;
      }
   }
private:
   enum { RUNNING, STUCK, BROKEN } state;
};
```

Figure 197: A class with various states

```
class Scanner
{
public:
   virtual string scan(char * & p) = 0;
};

class ScanBlanks : public Scanner
{
public:
   string scan(char * & p)
   {
      while (isspace(*p))
         p++;
      return string();
   }
};

class ScanNumber : public Scanner
{
public:
   string scan(char * & p)
   {
      string num;
      while (isdigit(*p))
         num += *p++;
      if (*p == '.')
      {
         num += *p++;
         while (isdigit(*p))
            num += *p++;
      }
      return num;
   }
};

class ScanIdentifier : public Scanner
{
public:
   string scan(char * & p)
   {
      string id(1, *p++);
      while (isalpha(*p) || isdigit(*p))
         id += *p++;
      return id;
   }
};
```

Figure 198: State pattern: scanners for white space, numbers, and identifiers

```
class ScanManager
{
public:
   ScanManager();
   ~ScanManager();
   string scan(char *buffer);
private:
   Scanner *ps;
   Scanner *psBlanks;
   Scanner *psNumber;
   Scanner *psIdentifier;
};

ScanManager::ScanManager() :
   ps(0),
   psBlanks(new ScanBlanks()),
   psNumber(new ScanNumber()),
   psIdentifier(new ScanIdentifier())
{ }

ScanManager::~ScanManager()
{
   delete psBlanks;
   delete psNumber;
   delete psIdentifier;
}

string ScanManager::scan(char *buffer)
{
   string result;
   char *p = buffer;
   while (*p != '\0')
   {
      if (isspace(*p))
         ps = psBlanks;
      else if (isdigit(*p))
         ps = psNumber;
      else if (isalpha(*p))
         ps = psIdentifier;
      else
         throw "illegal character.";
      string token = ps->scan(p);
      if (token.length() > 0)
         result += "<" + token + "> ";
   }
   return result;
}
```

Figure 199: State pattern: the scanner controller class

```
int main()
{
   ScanManager sm;
   char *test = "123 456 Pirate456\t789 anotherID ";
   try
   {
      cout << "Scan succeeded: " << sm.scan(test).c_str() << endl;
   }
   catch (const char *error)
   {
      cerr << "Scan failed: " << error << endl;
   }
   return 0;
}
```

Figure 200: State pattern: using the scanner

  - The state could be chosen externally. In the example above, `ScanManager` could provide another member function that was told what kind of token to expect and could set the state accordingly.

- It would be nice to put some useful data into the base class. In C++, we cannot do this efficiently, because the scanning is performed by separate objects that cannot access data in the base object. There are several workarounds for this problem:
  - We could construct state objects dynamically when needed, pass them the information they need, and delete them at the next state transition. This is clearly less efficient than the solution above, but the inefficiency might be acceptable if state transitions were infrequent.
  - We could provide additional functions for updating the `Scanner` objects before executing them. But it is not obvious when to call such functions.
  - Some object oriented languages (e.g., Self) provide *dynamic inheritance*, which allows an object to move around the class hierarchy, adopting the behaviour of the class it belongs to at any given time.

13.5 Bridge

Problem: Tight coupling between an interface and an implementation make it difficult to change either part without affecting the other.

Handle/Body

Solution: The Bridge pattern separates an abstraction from its implementation so that the two can vary independently. It is also known as the ***Handle/Body*** pattern. A Bridge is a kind of generalized Pimpl (see Section 12.6 on page 267).

There is an example of the Bridge pattern (called Handle/Body) in Koenig and Moo (Koenig
and Moo 2000). In Section 13.4 (pages 243–245), there is a declaration of class Student_info in
which the only data member is Core *cp. The pointer cp points an instance of either the base
561 class Core or the derived class Grad: see Figure 202 on the facing page. An instance of class
562 Student_info is constructed by reading data from a file. The class of *cp is determined by a
563 character in the file: see Figure 201.

```
istream& Student_info::read(istream& is)
{
    delete cp;          // delete previous object, if any

    char ch;
    is >> ch;           // get record type

    if (ch == ’U’) {
        cp = new Core(is);
    } else {
        cp = new Grad(is);
    }

    return is;
}
```

Figure 201: Setting the pointer (Koenig & Moo, page 245)

```
#ifndef GUARD_Student_info_h
#define GUARD_Student_info_h

#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

#include "Core.h"

class Student_info {
public:
    // constructors and copy control
    Student_info(): cp(0) { }
    Student_info(std::istream& is): cp(0) { read(is); }
    Student_info(const Student_info&);
    Student_info& operator=(const Student_info&);
    ~Student_info() { delete cp; }

    // operations
    std::istream& read(std::istream&);

    std::string name() const {
        if (cp) return cp->name();
        else throw std::runtime_error("uninitialized Student");
    }
    double grade() const {
        if (cp) return cp->grade();
        else throw std::runtime_error("uninitialized Student");
    }

    static bool compare(const Student_info& s1,
                        const Student_info& s2) {
        return s1.name() < s2.name();
    }

private:
    Core* cp;
};

#endif
```

Figure 202: Separating interface and implementation (Koenig & Moo, pages 243–244)

13.6 Command

The Command pattern *"encapsulates a request as an object, thereby enabling you to parameterize clients with different requests, queue or log requests, and support undoable operations"* (Gamma, Helm, Johnson, and Vlissides 1995, page 233). At the appropriate time, a request is activated in some way. The Gang of Four use a function called `Execute` to activate a request, but `operator()` provides a slightly neater solution.

564 The `Command` shown in Figure 203 class stores requests to print messages. The message to be
printed is passed as an argument to the constructor. To activate a command `c`, the user writes
`c()`. For example, commands can be stored in a vector and then activated sequentially, as shown
565 in Figure 204.

```
class Command
{
public:
    Command(string message) : message(message) {}
    void operator()() { cout << message << endl; }
private:
    string message;
};
```

Figure 203: A simple command class

```
vector<Command> commands;
commands.push_back(Command("First message"));
commands.push_back(Command("Second message"));
// ....
// a long time later:
for (   vector<Command>::iterator it = commands.begin();
        it != commands.end();
        ++it )
    (*it)();
```

Figure 204: Using the command class

Remember that you can declare `operator()` with any number of parameters (see Section 6.7.2 on page 121).

13.7 Observer

Model/View

The Observer pattern has a *Subject*, which is an object that changes its state, and one or more *Observers* that respond to changes in the Subject's state. Other names for Subject/Observer are Publish/Subscribe, and Model/View (in which case there may be a Controller as well).

566 In Figure 205 on the next page, the Subject is a thermometer `th` that registers a temperature. The Observers are a `DigitalReader` `dr` and an `AnalogReader` `ar`, which display the temperature

in different ways. Each Observer has a pointer to its Subject, set when the Observer is created. The Subject must also be told about the Observers: this is done by the call `th.add()`.

When the system has been set up, the temperature changes are recorded by calling `th.setTemp()`.
The thermometer notifies each of the observers of a change in temperature, and they display the
new temperature. If a reader is removed from the thermometer, it stops displaying. Figure 206 567
shows the output generated by the program of Figure 205.

```
#include "thermometer.h"
#include "digital.h"
#include "analog.h"

using namespace std;

int main()
{
   Thermometer th;
   DigitalReader dr(&th);
   AnalogReader ar(&th);
   th.add(&dr);
   th.add(&ar);
   th.setTemp(5);
   th.setTemp(10);
   th.remove(&dr);
   th.setTemp(15);
   return 0;
}
```

Figure 205: `main.cpp`

```
New temperature: 5
Digital: 5
Analog:  -----

New temperature: 10
Digital: 10
Analog:  ----------

New temperature: 15
Analog:  ---------------
```

Figure 206: Output from the program of Figure 205

Rather than working with thermometers and readers directly, we create a framework for the
Observer pattern with abstract base classes. Figure 207 on the following page shows the base 568
class for a Subject. It has functions to add and remove Observers, and a function that enables
the Subject to notify its Observers about updates. The Observers are stored in a list. 569

```
#include <list>

class Observer;

class Subject
{
public:
   virtual ~Subject() {}
   void add(Observer *obs);
   void remove(Observer *obs);
   void notify();
private:
   std::list<Observer*> observers;
};
```

Figure 207: `subject.h`

Figure 208 on the next page shows the implementation of the base class. Functions `add()` and
`remove()` use library functions to insert and remove entries from the list of Observers, and
`notify()` sends a message to each Observer in the list. Note that the Subject passes a pointer
570 to itself when notifying an Observer.

Figure 209 on the facing page shows the class `Thermometer`, which is derived from `Subject`. This class has functions related to the specific kind of Subject: in this case, functions that get and set the temperature. `setTemp()` is called by some external agency to indicate that the temperature has changed. `getTemp()` is called by the Observers to obtain the current temperature. (The example is over-simplified: a real Subject would probably do more than just provide get and set capabilities.)

571 The implementation of `Thermometer`, shown in Figure 210 on page 304, is straightforward. The
572 main point to note is that `setTemp()` calls `notify()`, which is inherited from the base class.

Figure 211 on page 304 shows the abstract base class for an Observer. The only required method is `notify()`, which must be implemented by derived classes. Since no code is needed for the base class, there is no implementation file for `Observer`.

573 In this example, we provide two kinds of Observer. Figure 212 on page 304 shows the header file
574 for class `DigitalReader` and Figure 213 on page 305 shows the corresponding implementation
file. The main point to note is that `notify()` checks that the Subject passed to it is the Subject
to which the `DigitalReader` owns a pointer.

It might seem simpler for `notify()` to use the pointer passed to it, by calling `changed->getTemp()`. However, this would not work because `changed` has type `Subject*`, and the base class `Subject` does not have a function `getTemp()`.

Is the test '`changed == th`' really necessary? To answer this, we have to consider circumstances under which it might be false. There might be a complex network of Subjects and Observers, and Observers might be added indiscriminately to Subjects. In such a context, it makes sense for the Observer to check that the notifier is indeed one that it is interested in. The point being illustrated here is that an Observer may choose to ignore notifications that are of no interest to

```
#include "subject.h"
#include "observer.h"

using namespace std;

void Subject::add(Observer *obs)
{
   observers.push_back(obs);
}

void Subject::remove(Observer *obs)
{
   observers.remove(obs);
}

void Subject::notify()
{
   for (  list<Observer*>::iterator it = observers.begin();
          it != observers.end();
          ++it)
   {
      (*it)->notify(this);
   }
}
```

Figure 208: `subject.cpp`

```
#include "subject.h"

class Thermometer : public Subject
{
public:
   Thermometer() : temp(0) {}
   int getTemp();
   void setTemp(int newTemp);
private:
   int temp;
};
```

Figure 209: `thermometer.h`

```
#include "thermometer.h"

#include <iostream>
using namespace std;

int Thermometer::getTemp()
{
   return temp;
}

void Thermometer::setTemp(int newTemp)
{
   temp = newTemp;
   cout << "\nNew temperature: " << newTemp << endl;
   notify();
}
```

Figure 210: `thermometer.cpp`

```
class Subject;

class Observer
{
public:
   virtual ~Observer() {}
   virtual void notify(Subject *changed) = 0;
};
```

Figure 211: `observer.h`

it. In a more realistic example, the condition ‘`changed == th`’ might be replaced by another condition to test the relevance of the update.

```
#include "observer.h"
#include "thermometer.h"

class DigitalReader : public Observer
{
public:
   DigitalReader(Thermometer *th);
   void notify(Subject *changed);
private:
   Thermometer *th;
};
```

Figure 212: `digital.h`

```
#include "digital.h"

#include <iostream>
using namespace std;

DigitalReader::DigitalReader(Thermometer *th) : th(th)
{}

void DigitalReader::notify(Subject *changed)
{
   if (changed == th)
      cout << "Digital: " << th->getTemp() << endl;
}
```

Figure 213: `digital.cpp`

The second kind of Observer is an `AnalogReader`, shown as a header file (Figure 214) and an 575
implementation file (Figure 215 on the following page). To make things more interesting, class 576
`AnalogReader` is derived from class `AnalogDisplay`. This provides an example of ***multiple implementation inheritance***, in which a derived class has two base classes (see Section 8.5.2 on page 175).

Apart from the multiple inheritance, `AnalogReader` is similar to `DigitalReader`. Its `notify()` function checks that it is interested in the Observer, and then calls function `draw()` from `AnalogDisplay` to display the thermometer reading.

```
#include "observer.h"
#include "thermometer.h"

class AnalogDisplay
{
public:
   void draw(int val);
};

class AnalogReader : public AnalogDisplay, public Observer
{
public:
   AnalogReader(Thermometer *th);
   void notify(Subject *changed);
private:
   Thermometer *th;
};
```

Figure 214: `analog.h`

```
#include "analog.h"

#include <iostream>
using namespace std;

void AnalogDisplay::draw(int val)
{
   cout << "Analog:  ";
   for (int i = 0; i < val; ++i)
      cout << ’-’;
   cout << endl;
}

AnalogReader::AnalogReader(Thermometer *th) : th(th) {}

void AnalogReader::notify(Subject *changed)
{
   if (changed == th)
      draw(th->getTemp());
}
```

Figure 215: analog.cpp

13.8 Pattern Languages

Despite its age, the original book *Design Patterns* (Gamma, Helm, Johnson, and Vlissides 1995) is still the reference to read. The code examples in this book are mostly in C++ (with some in Smalltalk) – this is probably the main reason this book is usually not given to beginner Java programmers. The other reason is that, at the time it was written, patterns were a topic for advanced, professional software developers, not learners of object-oriented programming. In style and addressed audience, the Java based book (Freeman and Freeman 2004) is probably the polar opposite.

Alexander's pattern language

But the original idea of patterns and pattern languages comes from building architecture. Christopher Alexander, together with colleagues, examined re-occurring problems in design and construction and how they could be solved in a generic fashion. Their solution was the development of a ***pattern language***, where (Alexander, Ishikawa, and Silverstein 1977):

> *"Each pattern describes a problem that occurs over and over again in our environment, and then describes the core solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice."*

The book is very accessible to non-architects, which is perhaps one of the reasons it inspired design patterns and pattern languages in software engineering.

References

Alexander, C., S. Ishikawa, and M. Silverstein (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Generic Patterns Applied*. Addison-Wesley.

Freeman, E. and E. Freeman (2004). *Head First Design Patterns*. O'Reilly.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.

14 But wait, there's more!

14.1 Policies 309
14.2 Traits 313
14.3 Metaprogramming 317
14.4 Functional programming 320
14.5 Envoi 323

In this final lecture, we briefly look at some more advanced programming paradigms used in C++: policy-based design, traits, template metaprogramming, and functional programming.

14.1 Policies

Policies, used in *policy-based design* (also known as *policy-based programming*), are a programming technique that is closely connected with C++ template programming (although in principle it can be applied to other languages as well). It was made popular by Andrei Alexandrescu, who described it in his book on *Modern C++ Design* (Alexandrescu 2001). Here's a basic motivation for the kind of problem you can solve with policies: You are developing a template class that provides, among other things, heap memory management. Now you need to delete an object of type T. But which delete function should you use, the one for arrays or the one for objects? Since you are writing a template class, you don't know if it will be applied to an array type (e.g., char *) or a non-array type (e.g., string), so either delete t or delete[] t will be wrong in the other case. The solution is to provide both versions of delete, and select between them at compile-time using a *policy class*. This can be implemented by combining templates with inheritance or by adding the policy class as another template argument – i.e., a template parameter that is itself a template:[76] 578

policy class

```
template <typename T, template <typename> class PolicyClass>
class FooHeap
{
   ...
   PolicyClass<T> t;
   ...
};
```

14.1.1 Hello, Policy!

As a minimal example of a policy class, consider a "Hello World" program that allows you to configure its output language via a policy. First, we define two policy classes, one for an English message, one for a German one (see Figure 216 on the following page). 579

Based on these policy classes, we can now define a HelloWorld template class that can be configured via a language policy class (see Figure 217 on the next page). Note that we are using private inheritance to access the policy class' operations from the HelloWorld class. 580

[76] Template parameters that are template names are called *template template parameters* (Dewhurst 2005, Item 55)

```
class LanguagePolicyEnglish
{
protected:
    string message() const
    {
        return "Hello, world!";
    }
};

class LanguagePolicyGerman
{
protected:
    string message() const
    {
        return "Hallo, Welt!";
    }
};
```

Figure 216: Policy classes for “Hello, world” in two languages

```
template <typename LanguagePolicy = LanguagePolicyEnglish>
class HelloWorld : private LanguagePolicy
{

public:
    void operator()() const
    {
      cout << LanguagePolicy::message() << endl;
    }
};
```

Figure 217: Template “Hello, world” class using a language policy

```
int main()
{
    HelloWorld<> hello_world;
    hello_world();        // prints "Hello, world!"

    HelloWorld<LanguagePolicyGerman> hello_world2;
    hello_world2();       // prints "Hallo, Welt!"
}
```

Figure 218: Testing the “Hello, world” class with two policies

We can now create objects from this template class, which will print the output message depending on the chosen language policy (with English as default), as shown in Figure 218 on the facing page. 581

14.1.2 Policies in LOKI

LOKI

More practical examples can be found in the LOKI library, which was built by Alexandrescu and others (Alexandrescu 2001). LOKI shows how features of C++ can be combined in unexpected ways to give useful results. The complete library implements policy classes, static assertions, type lists, small object allocators, generalized functions, singletons, smart pointers, factories, and a "minimalist" multithreading library. Here, we briefly look at how policy classes are used in LOKI.

It is easy to design simple classes for particular applications; it is much harder to design classes that are useful for many applications. Consider an autopointer class (cf. Section 10.4.1 on page 213). Should it assume a single thread or multiple threads? How should it manage ownership – reference counting, linking, or some other way? Should we define a whole collection of classes: 582

```
BasicSmartPointer
SingleThreadedSmartPointer
MultiThreadedSmartPointer
RefCountedSmartPointer
MultiThreadedRefCountedSmartPointer
```

and so on? Alexandrescu argues that this problem cannot be solved properly either by inheritance (even with multiple inheritance) or with templates. However, we can get the effects we need by using both templates *and* inheritance.

The first step is to define the policy classes. These are classes that implement a particular operation in various ways. Figure 219 on the next page shows three possible ways of allocating a 583 new object: the first uses `new`, as usual; the second uses `malloc` and the "placement" version of `new` 584 (which does not allocate memory itself); the third returns a clone of a prototype (Alexandrescu 2001, page 8). Each class implements a single `static` function, `create`. 585

placement `new`

Prototype

Here, *Prototype* refers to the design pattern (Gamma, Helm, Johnson, and Vlissides 1995), where you avoid creating new objects from scratch by copying (cloning) existing objects. Note that the copy constructor does not help you in case of polymorphic classes when you don't know the dynamic type of the object you want to copy (the copy constructor is always chosen at compile-time, based on the object's static type). Since C++ does not offer a `clone` function like Java, this is often implemented using the *virtual constructor* idiom[a] combined with covariant return types (Dewhurst 2005, Item 29).

[a] See https://en.wikibooks.org/wiki/More_C++_Idioms/Virtual_Constructor

Classes that perform dynamic allocation can then be defined as template classes that inherit a particular policy: 586

```
template <typename T>
class NewCreator
{
public:
    static T* create()
    {
        return new T;
    }
};

template <typename T>
class MallocCreator
{
public:
    static T* create()
    {
        void *buf = std::malloc(sizeof(T));
        if (!buf)
            return 0;
        return new(buf) T;
    }
};

template <typename T>
class PrototypeCreator
{
public:
    PrototypeCreator(T *proto = 0) : proto(proto) {}
    static T* create()
    {
        return proto ? proto->clone() : 0;
    }
private:
    T* proto;
};
```

Figure 219: Policy classes for dynamic allocation

```
template <typename CreatePolicy>
class GenericWidgetManager : public CreatePolicy
{
    ....
};
```

It is often convenient to hide a choice of policy in a typedef:

```
typedef GenericWidgetManager<MallocCreator<Widget> >  WidgetManager;
```

The team leader then tells the grunt programmers to use instances of WidgetManager and to call create when they want to construct a new Widget.

14.1.3 Policies in the STL

The STL uses policy classes for memory allocation in exactly this way. For example, the class vector is actually defined as: 587

```
template <typename T, typename Allocator = allocator<T> >
class vector
{
    ....
};
```

Usually, we write declarations of the form vector<double>, which use the default allocator, so what you are getting is actually a vector<double, std::allocator<double> >. If we want to manage memory in a special way, however, we can define our own allocator class. This class must provide a set of features so that it can be used correctly by the class vector. These features are described in various places, e.g., (Musser, Derge, and Saini 2001, Chapter 24) or (Josuttis 2012, Chapter 19).

To learn more about policies, read (Dewhurst 2005, Item 56) for an introduction and (Alexandrescu 2001) for more details.

14.2 Traits

Traits provide compile-time information about types and are used in template classes. For example, you could use traits to determine whether a type is a simple type, a pointer type, or a reference type. As a more interesting example, you might want to know the most efficient way of passing an argument to a function. With a specific type, such as char or EnormousObject, this is easy. But with a template type T, it is not so obvious. A type traits class can convert T into a type ParameterType which gives the best way of passing Ts to functions.

14.2.1 Traits in the STL

Traits are used in various places in the STL, like in the stream library. Figure 220 on the following page shows the declaration of the standard library class basic_ostream. 588

The template argument corresponding to charT determines the type of character. The default is char and the most common alternative is wchar_t, which gives Unicode.

The template argument corresponding to traits defines properties of the character type. These properties allow the functions of basic_ostream to be written so as not to depend on the particular character type. The class char_traits<char> contains, *inter alia*:

- typedefs for various types associated with char
- functions to assign, compare, move, and copy chars and char arrays

```
template <typename charT, typename traits = char_traits<charT> >
class basic_ostream : virtual public basic_ios<charT, traits>
{
    ....
};

typedef basic_ostream<char> ostream;
```

Figure 220: Declaration of the standard basic output stream class

- a function to test for end of file

iterator traits Another example are ***iterator traits***. Back in Section 5.3.1 on page 91, we have seen that there are different categories of iterators (forward, bidirectional, random access, etc.) that offer different capabilities (increment, compare, order, arithmetic, etc.). When you create an iterator for a concrete container type, you know what iterator type you will get. But what if you want to write a template class and need to know the capabilities of an iterator passed as a template argument? You can get all the required information for an iterator type T from the iterator traits shown in Figure 221. 589

```
namespace std {
    template <typename T>
    struct iterator_traits {
        typedef typename T::iterator_category iterator_category;
        typedef typename T::value_type        value_type;
        typedef typename T::difference_type   difference_type;
        typedef typename T::pointer           pointer;
        typedef typename T::reference         reference;
    };
}
```

Figure 221: Iterator traits in the STL

590 Trait tags are by convention defined as structs:

```
namespace std {
    struct output_iterator_tag {
    };
    struct input_iterator_tag {
    };
    struct forward_iterator_tag
     : public input_iterator_tag {
    };
    ....
}
```

So you can now write template code that queries the capabilities of an iterator type at compile-time with

```
    typename std::iterator_traits<T>::iterator_category()
```

and then select appropriate operators, based on the iterator category. For example, if you need to advance an iterator `n` steps, you can use arithmetic `i+n` with a random access iterator, but step one element at a time for a forward iterator. This is again achieved by overloading the function modifying the iterator. For example, if you want to write a `my_advance` template function, you pass the iterator trait as parameter: 591

```
template <typename T>
inline void my_advance(T beg, T end)
{
    my_advance(beg, end, std::iterator_traits<T>::iterator_category());
}
```

and then provide two overloads as template specializations for `my_advance`, so that the correct version will be chosen at compile-time: 592

```
template <typename T>
void my_advance(T beg, T end, std::bidirectional_iterator_tag)
{ .... }

template <typename T>
void my_advance(T beg, T end, std::random_access_iterator_tag)
{ .... }
```

For more on iterator traits, see (Meyers 2005, Item 47) and (Josuttis 2012, Chapter 9.5).

14.2.2 Traits in Boost

The Boost library `TypeTraits` provides ten basic tests for type properties: 593

```
is_array<T>       is_class<T>       is_enum<T>       is_float<T>
is_function<T>    is_integral<T>    is_pointer<T>    is_reference<T>
is_union<T>       is_void<T>
```

Boost also provides comparisons between types (`is_same<T1,T2>`, `is_base<B,D>`, etc.) and type transformations (`remove_const<T>`, `add_pointer<T>`, etc.).

Boost

Boost (http://www.boost.org/) is a set of libraries intended to help advanced C++ programmers. Some of the libraries are candidates for inclusion in the STL. Aspects of the Boost libraries are described in (Abrahams and Gurtovy 2005). Boost includes traits, but most of its libraries are based on template metaprogramming, described in the next section.

Here is a simplified application of these metafunctions. There is a fast way and a slow way to swap two iterators. If the iterators are the same type and they are both reference types, we can use the standard template function `swap`. In all other cases, we have to explicitly code the swap to provide for conversions. The problem is to make this decision at compile-time.

```
template<>
struct iter_swap_impl<true>
{
    template <typename Iterator1, typename Iterator2>
    static void doSwap(Iterator1 i1, Iterator2 i2)
    {
        std::swap(i1, i2);
    }
};

template<>
struct iter_swap_impl<false>
{
    template <typename Iterator1, typename Iterator2>
    static void doSwap(Iterator1 i1, Iterator2 i2)
    {
        typename iterator_traits<Iterator1>::value_type tmp = *i1;
        *i1 = *i2;
        *i2 = tmp;
    }
};
```

Figure 222: Two implementations of `swap`

The solution requires two versions of the swapping code with a `bool` template parameter, shown
594 in Figure 222, and a way of providing a template argument that is either `true` or `false` depending
595 on the types of the iterators, shown in Figure 223.

596

```
typedef iterator_traits<Iterator1> traits1;
typedef typename traits1::value_type v1;
typedef typename traits1::reference r1;

typedef iterator_traits<Iterator2> traits2;
typedef typename traits2::value_type v2;
typedef typename traits2::reference r2;

const bool use_swap =
    boost::is_same<v1, v2>::value &&
    boost::is_reference<r1>::value &&
    boost::is_reference<r2>::value;

template <bool use_swap>
struct iter_swap_impl;
```

Figure 223: Choosing the version

Traits have been adopted by other programming languages as well, e.g., in PHP. A more detailed introduction to traits can be found in (Meyers 2005, Item 47) and (Dewhurst 2005, Item 54). In C++11, it is possible to replace some traits with the new type functions `auto` and `decltype` to simplify code (Stroustrup 2013, Chapter 28.2.4).

14.3 Metaprogramming

C++ supports the *metaprogramming* paradigm, but this happened more or less by accident, not because it was planned. Metaprogramming generally refers to code that can create or manipulate other *code* rather than mere *data* (or even change its own code). Template metaprogramming (also see Section 7.6 on page 146) appeared in the C++ world in 1989. Towards the end of a workshop on generic programming, Kristof Czarnecki and Ullrich Eisenecker showed C++ source code for a LISP interpreter. LISP interpreters are commonplace but this one had an interesting feature: it was executed *by the compiler*. For example, if you gave it (the equivalent of) a LISP expression with the value 42, it would generate the C++ program

```
cout << 42 << endl;
```

Although the interpreter was initially seen as a mere curiosity, interest developed rapidly, and *template metaprogramming* (TM) is now viewed as an important tool in the advanced programmer's kit.

As a simple example of TM, the following program prints 42: 597

```
int main()
{
    cout << binary<101010>::value << endl;
    return 0;
}
```

This magic is accomplished by the template class shown in Figure 224 on the next page. The first point to note is that 101010 is actually a *decimal* number that contains only the digits 0 and 1. Given a decimal number `N`, the last digit is `N % 10` and the preceding digits are `N / 10`. To obtain the binary value, however, we must multiply the k'th digit by 2^k. 598

The template class `Binary` extracts the last digit of its template argument `N` and uses recursion to get the other digits. Eventually, it will instantiate `Binary<0>`, and we provide a specialization of the template class that assigns 0 to `Binary::value` in this case.

14.3.1 MPL

We can use templates to check dimensions at compile time. The type `Quantity<L,M,T>`, in which the values of `L`, `M`, and `T` are small integers, is a `double` with the indicated dimensions. For example, the type of length is `Quantity<1,0,0>`, momentum is `Quantity<1,1,-1>`, force is `Quantity<1,1,-2>`, and so on. When quantities are multiplied, their dimensions are added: 599

```
template<unsigned long N>
class Binary
{
public:
    static unsigned const value =
        Binary<N/10>::value * 2 + N % 10;
};

template<>
class Binary<0>
{
public:
    static unsigned const value = 0;
};
```

Figure 224: Converting binary to decimal at compile-time

```
template<int L1, int M1, int T1, int L2, int M2, int T2>
Quantity<L1+L2,M1+M2,T1+T2>
operator*(const Quantity<L1,M1,T1> & x, const Quantity<L2,M2,T2> & y)
{
    return Quantity<L1+L2,M1+M2,T1+T2>(x.value() * y.value());
}
```

The Boost library MPL provides a similar service, but it does so in a more elaborate way. The definition of `operator*` in MPL is:

```
template<typename T, typename D1, typename D2>
quantity<T, typename mpl::transform<D1,D2,plus_f>::type>
operator*(quantity<T,D1> x, quantity<T,D2> y)
{
    return
        quantity<T, typename mpl::transform<D1,D2,plus_f>::type>
        (x.value() * y.value());
}
```

The differences are:

- The MPL represents dimensions using type sequences, which appear above as `D1` and `D2`. Thus `D1` might correspond to `<1,0,0>`, which is a mass.

- The MPL provides template metafunctions that take dimension types as arguments and return new dimension types. Thus `transform<D1,D2,plus_f>::type` is the type obtained by adding the components of `D1` and `D2`, and is therefore appropriate for multiplication.

14.3.2 Blitz++

Blitz++ (Veldhuizen 1995) was the first C++ library to make heavy use of template metaprogramming and it had a considerable effect on the C++ community. Blitz++ provides many things, but its single most important contribution is efficient array manipulation using *expression templates*. We will briefly describe the key problem that Blitz++ solves.

expression templates

600 Consider the assignment

```
m = a + b + c + d;
```

601 in which each variable is a matrix. The obvious implementation of `operator+` for matrices is likely to be something like the function in Figure 225 and the assignment above will invoke it three times.

```
Matrix operator+(const & Matrix a, const & Matrix b)
{
    typedef std::size_t ST;
    ST const sza = a.size;
    ST const szb = b.size;
    assert(sza == szb);
    Matrix result;
    for (ST i = 0; i != sza; ++i)
        for (ST j = 0; j != sza; ++i)
            result[i][j] = a[i][j] + b[i][j];
    return result;
}
```

Figure 225: Matrix addition

602 What we really want to happen is something like this:

```
for (ST i = 0; i != sza; ++i)
    for (ST j = 0; j != sza; ++i)
        m[i][j] = a[i][j] + b[i][j] + c[i][j] + d[i][j];
 return m;
```

603 This is in fact pretty much what Blitz++ would generate from the original assignment. It uses template capabilities to parse an expression and generate code that rebuilds the expression.

Figure 226 on the next page shows some simple applications of Blitz++.

- The first three declarations introduce `i`, `j`, and `k` as index variables for the first, second, and third dimensions of arrays.
- The `Range` declaration specifies that indexes should be 1,2,3,4 (instead of the usual 0,1,2,3).
- Matrix `A` is a Hilbert matrix defined by

$$A_{ij} = \frac{1}{i+j-1}$$

- `B` is a "reverse diagonal" matrix:

$$B = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

- `C` is a $4 \times 4 \times 4$ array and the assignment computes

$$C_{ijk} = A_{ik} \times B_{kj}$$

- Finally, matrix `D` is the matrix product of `A` and `B`:

$$D_{ij} = \sum_k (A_{ik} \times B_{kj})$$

```
int main()
{
    firstIndex i;
    secondIndex j;
    thirdIndex k;

    Range r(1,4);

    Array<float,2> A(r,r) =  1.0 / (i+j-1);
    Array<float,2> B(r,r) = (i == (5-j));
    Array<float,3> C(r,r,r) = A(i,k) * B(k,j);
    Array<float,2> D(r,r) = sum(A(i,k) * B(k,j), k);
}
```

Figure 226: Fun with Blitz++

A brief introduction to template metaprogramming can be found in (Meyers 2005, Item 48). For more details, see (Stroustrup 2013, Chapter 28) and (Abrahams and Gurtovy 2005). The general template reference book (Vandevoorde, Josuttis, and Gregor 2017) also has a chapter on metaprograms.

14.4 Functional programming

Lisp

closure

Functional programming is a major programming paradigm, and also one of the earliest, predating object-oriented programming languages by many years: Lisp, one of the oldest programming languages in active use, dates back to 1958. Other well-known functional programming languages you might have heard of are *Scheme, Haskell, Erlang,* or *Clojure.* An introduction to functional programming is well beyond the scope of this course. However, we've actually covered some concepts that come from functional programming: MapReduce (Section 11.4 on page 247) takes one idea from functional languages, the `map` function, to parallelize data processing. The Command design pattern (Section 13.6 on page 300) emulates the functional concept of a *closure* using object-oriented techniques. Many of these functional programming ideas allow to express more complex concepts in fewer lines of code than traditional procedural or object-oriented programming, which explains the ongoing trend of adding functional constructs to existing (e.g., C++11 or Java 8) or new object-oriented languages (e.g., Scala, Python, or Ruby). One such addition to C++11 are *Lambda functions*. lambda

14.4.1 Lambdas

As a motivation, let's assume you have an address book application with a list of objects, representing persons with their names, phone numbers, email addresses, etc. Now you want to implement some `search` functions to select a subset of all entries. Your users might want to search by first name, last name, or city, so you could implement corresponding `searchFirst`, `searchLast` and `searchCity` functions. But then you might get additional feature requests: 604 Could we search with partial names? Ignoring case? Using regular expressions? Search by phone area code? By date of entry/last update? Adding new functions (or parameters to existing ones) can quickly grow out of hand, and it will still not be possible to do arbitrary data analysis (e.g., select all objects where the last letter of the first name is the first letter of the last name, where the city name has 5 characters, . . .). A much better solution is to have a single `search` function that takes a *function* as a parameter, and then applies this function to decide whether to match an element from the address book. We can then cover all the cases above by passing different match functions to the `search` function. In C++, passing functions is possible using ***function pointers*** (we've seen an example for this in the `bisect` function in Figure 14 on page 35) or ***function objects***, also called *functors* (we used a functor in the `AddressBook` example in Figure 70 on page 122). But this needs a lot of code writing: a complete function declaration and definition in the first case, and a complete class with overloaded `operator()` in the second. Lambdas, as added in C++11, allow you to create *unnamed functions* "on-the-fly" in your code, like this: [] ()

605

```
int main()
{
    [] () { cout << "Hello, Lambda!"; };
}
```

This lambda function can then be passed like a function pointer or functor. You can even assign the lambda to a variable, using C++11's new `auto` keyword for automatic type deduction: auto

606

```
auto mylambda = [&count](int x){count += (x % 2 == 0);}
```

So, using lambdas, you can now implement a generic `search` function, by making use of the new `std::function` type for the argument list: std::function

```
#include <functional>
...
AddressBook search( std::function<bool (const entry&)> lambda)
```

For a more complete example, have a look at http://www.cprogramming.com/c++11/c++11-lambda-closures.html.

14.4.2 Closures

The lambda examples above included the brackets `[]` before the function arguments, which do not exist in ordinary, named functions. This is the so-called ***closure*** of the lambda, a list of variables that are made available to the lambda function from the calling scope.

As an example, let's write a lambda that prints a string coming from a variable `msg` in the calling
607 context `n` times (where n is a parameter for the lambda function):

```
string msg = "Hello, closure!\n";
auto lambda = [msg] (int n) { for( int i=0; i != n; ++i ) cout << msg; };
lambda(2);
```

This will print out `"Hello, closure!"` 2 times. Note that redefining `msg` will not result in a different message, since the value of `msg` is captured at the time of the *definition* of the lambda function:

```
....
msg = "Goodby, closure!\n";
lambda(1);   // still prints "Hello, closure!"
```

In this case, the variable `msg` is captured by value, but this is not the only option with closures.
608 You can use:

closure definitions

- `[]` No variables from the calling scope are captured.
- `[=]` All local variables from the calling scope are captured by value.
- `[&]` All local variables from the calling scope are captured by reference.
- `[`*`args`*`]` a list of variables (without type specifiers) are captured by value.
- `[=, `*`args`*`]` *or* `[&, `*`args`*`]` all local variables from the calling scope are captured by value (`=`)/reference (`&`), except for the ones named by *`args`*, which can have a different access method specified.

Command pattern and lambdas

Note that lambdas with closures now enable you to implement the *Command* design pattern (see Section 13.6 on page 300) in a much more direct way, since you can now create lambda objects that encapsulate a function ***and*** data from the calling context. You can store these objects in a data structure, and execute them at a later time, in a completely different part of your code, where they will "remember" the original context at the time of their creation. This ability to treat functions as objects is really the important and new functional programming feature introduced by lambdas.

functions as objects

`<soapbox>` If you are serious about programming, you should learn functional programming during your career – even if you do not keep using a functional language, it will provide you with new insights on how code and data can inter-operate, and this will also improve your general programming skills. Plus, with more object-oriented languages gaining object-functional extensions it will help you to understand their origin and proper application. `</soapbox>`

For further recommendations on learning functional programming, see Appendix A on page 327.

14.5 Envoi

It does seem, sometimes, as if the mentality of the best C++ programmers is very close to the mentality of the best hackers: when you want to do something, choose the construct that looks least appropriate and bend it into the required shape. Consider manipulators. Consider what happens when we write: 610

```
cout << "The answer is " << setw(8) << answer << '.' << endl;
```

The call `setw(8)` returns an instance of the class `ManipulatorForInts`: 611

```
ManipulatorForInts setw(int n)
{
    return ManipulatorForInts(ios::width, n);
}
```

Here is the class `ManipulatorForInts`: 612

```
class ManipulatorForInts
{
    friend ostream & operator<<(ostream & os,
                 const ManipulatorForInts & m);
public:
    ManipulatorForInts (int (ios::*)(int), int);
private:
    int (ios::*memberfunc)(int);
    int value;
}
```

The friend function `operator<<` applies the member function of `ManipulatorForInts` to the value supplied by the user: 613

```
ostream & operator<<(ostream & os,
               const ManipulatorForInts & m)
{
    (os.*memberfunc)(m.value);
    return os;
}
```

The sequence of events when the statement

```
cout << "The answer is " << setw(8) << answer << '.' << endl;
```

executes is as follows:

1. The function `setw` is called.
2. `setw` calls the constructor of `ManipulatorForInts` with the stream function `ios::width` and the integer `8` as arguments. It returns this object (call it `mfi`).

3. `operator<<` is called with arguments `cout` and `mfi` as arguments. It evaluates

   ```
   (os.*memberfunc)(m.value);
   ```

   in which `os.*memberfunc` is `ios::width` and `m.value` is 8, setting the output width of `cout` to 8

Naturally, the actual implementation is much more complicated than this because it is defined in the template class `basic_ostream`. It seems like a lot of effort just to set the width of an output stream.

On the positive side, C++ is still a language that is widely used, and will continue to be used, for important applications. While it is no longer the mainstream object-oriented language used by default in a project, it remains important for many areas, including embedded systems programming, game programming, and high-performance computing. Examples for significant programs that have been implemented in C++ in recent years include Google's *TensorFlow* machine learning library and many crypto-currency mining programs.

614 **Opinions.** Not everyone has warm, fuzzy feelings about C++:

615 Alistair Cockburn: *I consider C++ the most significant technical hazard to the survival of your project and do so without apologies.*

616 Robert Firth: *C++ has its place in the history of programming languages — just as Caligula has his place in the history of the Roman Empire.*

617 Eric Lee Green: *C++ is an atrocity, the bletcherous scab of the computing world, responsible for more buffer overflows, more security breaches, more blue screens of death, more mysterious failures than any other computer language in the history of the planet Earth.*

618 Alan Kay: *I made up the term 'object-oriented', and I can tell you I didn't have C++ in mind.*

619 David Keppel: *It has been discovered that C++ provides a remarkable facility for concealing the trivial details of a program – such as where its bugs are.*

620 Donald Knuth: *Whenever the C++ language designers had two competing ideas as to how they should solve some problem, they said, "OK, we'll do them both". So the language is too baroque for my taste.*

621 Bertrand Meyer: *There are only two things wrong with C++: The initial concept and the implementation.*

622 Thant Tessman: *Being really good at C++ is like being really good at using rocks to sharpen sticks.*

But let's give the last word to the creator of C++.

623 Bjarne Stroustrup: *There are only two kinds of programming languages: those people always bitch about and those nobody uses.*

References

Abrahams, D. and A. Gurtovy (2005). *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley.

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Generic Patterns Applied*. Addison-Wesley.

Dewhurst, S. C. (2005). *C++ Common Knowledge: Essential Intermediate Programming*. Addison-Wesley.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Josuttis, N. M. (2012). *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley. http://www.cppstdlib.com/.

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley.

Musser, D. R., G. J. Derge, and A. Saini (2001). *STL Tutorial and Reference Guide*. Professional Computing Series. Addison-Wesley.

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.

Vandevoorde, D., N. M. Josuttis, and D. Gregor (2017). *C++ Templates* (2nd ed.). Addison-Wesley. http://tmplbook.com.

Veldhuizen, T. (1995, May). Using C++ template metaprograms. *C++ Report 7*(4), 36–43.

A Hic Sunt Dracones

You start your first industry job, and someone tasks you with fixing that bug in a 20-year-running million-lines-of-code safety-and-security-critical system. Hello, real world!

When you find that you understood all the material covered in this course, but existing code (and explanations by your co-workers) still have you baffled, you will need to teach yourself more intermediate and advanced concepts. Here are some recommendations for "essential" readings:

C++ Reference: The authoritative reference (apart from the official standard) is Stroustrup's book *The C++ Programming Language*, now in its fourth edition (Stroustrup 2013). However, this is really a reference book: It is not intended as a "beginner-friendly" introduction, but a book written for software professionals. For a well-rounded guide for self-study, with C++11 coverage, get a book like Prata's *C++ Primer Plus* (Prata 2012).

Intermediate C++ Programming: We covered some "intermediate-level" concepts and design aspects in this course; but for more details, start with Meyers' *Effective C++* (Meyers 2005), *Effective Modern C++* (Meyers 2014), and Dewhurst's *C++ Common Knowledge* (Dewhurst 2005). You should read through these before attempting to move on to advanced C++ techniques.

Template Programming: For a thorough discussion of template programming, get the book *C++ Templates* (Vandevoorde, Josuttis, and Gregor 2017). This book also covers template metaprogramming in one chapter. Alexandrescu's book (Alexandrescu 2001) also covers advanced template programming, in particular template metaprogramming and policy-based design.

Libraries: While you might get by with online references, a good book on the standard libraries including the STL, like Josuttis' *The C++ Standard Library* (Josuttis 2012), is an essential item for your bookshelf. Additional recommended reading is Meyers' *Effective STL* (Meyers 2001). For more material on the stream libraries than you thought possible, see *Standard C++ IOStreams and Locales* (Langer and Kreft 2000).

Object-Oriented Design: The original GoF book on *Design Patterns* (Gamma, Helm, Johnson, and Vlissides 1995) is a must-read, even (especially?) if you studied one of the "Javafied" for-dummies versions before. (Alexandrescu 2001) shows clever implementations for some of these patterns (Command, Singleton, Proxy, Factory Method, Abstract Factory, and Visitor).

Coding: For books on coding in general, see (Bentley 1986; Bentley 1988; Raymond 2004).

Concurrency: C++11 multithreading is well covered by Williams in *C++ Concurrency in Action* (Williams 2012). Josuttis also provides a brief overview in (Josuttis 2012, Chapter 18).

Security: To learn more about security, read (Seacord 2013). Additional useful references are (Graff and van Wyk 2003; Hoglund and McGraw 2004).

Functional Programming: If you are serious about a career in programming, you must learn functional programming, too. With concepts dating back to the 1950's (when LISP arrived), you might wonder what the fuss is all about. For starters, read Paul Graham's (of Y Combinator fame) speech *Beating the Averages* at http://paulgraham.com/avg.html.

Mainstream languages are still catching up with functional features that Lisp had since the 1970's, most recently in C++11 or Java 8. However, just looking at these individual functional features will not give you a good understanding of functional programming. My recommendation is an "old fashioned" approach: Install a *Scheme*[77] interpreter and work through the book *The Little Schemer* (Friedman and Felleisen 1995).[78] Then, continue with the "Wizard Book", SICP (Abelson, Sussman, and Sussman 1984), an undergraduate textbook at MIT. For moving on to Lisp, get hyped by watching the music video and web comic introducing the book *Land of Lisp* (Barski 2010).

References

Abelson, H., J. Sussman, and J. Sussman (1984). *Structure and Interpretation of Computer Programs*. MIT Press. https://mitpress.mit.edu/sicp/.

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Generic Patterns Applied*. Addison-Wesley.

Barski, C. (2010). *Land of Lisp: Learn to Program in Lisp, One Game at a Time!* No Starch Press. http://landoflisp.com/.

Bentley, J. (1986). *Programming Pearls*. Addison-Wesley.

Bentley, J. (1988). *More Programming Pearls*. Addison-Wesley.

Dewhurst, S. C. (2005). *C++ Common Knowledge: Essential Intermediate Programming*. Addison-Wesley.

Friedman, D. P. and M. Felleisen (1995). *The Little Schemer* (4th ed.). MIT Press. https://mitpress.mit.edu/books/little-schemer.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Graff, M. G. and K. R. van Wyk (2003). *Secure Coding: Principles and Practices*. O'Reilly.

Hoglund, G. and G. McGraw (2004). *Exploiting Software: How to Break Code*. Addison-Wesley.

Josuttis, N. M. (2012). *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley. http://www.cppstdlib.com/.

Langer, A. and K. Kreft (2000). *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley.

Meyers, S. (2001). *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley.

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley.

Meyers, S. (2014). *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* (1st ed.). O'Reilly.

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Raymond, E. S. (2004). *The Art of UNIX Programming*. Addison-Wesley. http://www.catb.org/esr/writings/taoup/.

Seacord, R. C. (2013). *Secure Coding in C and C++* (2nd ed.). Addison-Wesley.

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.

Vandevoorde, D., N. M. Josuttis, and D. Gregor (2017). *C++ Templates* (2nd ed.). Addison-Wesley. http://tmplbook.com.

Williams, A. (2012). *C++ Concurrency in Action*. Manning. http://www.manning.com/williams/.

[77] A Lisp dialect, see https://en.wikipedia.org/wiki/Scheme_%28programming_language%29

[78] Note that this is not a textbook; it is written in the now almost-forgotten "conversational style". As one reviewer on Amazon puts it, *"a reminder of older, cleverer times"*.

B Coding Standards

B.1 Rules . . . 330
B.2 Layout . . . 331
B.3 Comments . . . 335
B.4 Names . . . 338
B.5 Constants and Literals . . . 341
B.6 const . . . 341
B.7 Variables . . . 344
B.8 Expressions . . . 344
B.9 Exceptions . . . 344
B.10 Functions . . . 345
B.11 Classes . . . 347
B.12 Libraries . . . 349
B.13 Files . . . 350
B.14 Development . . . 351
B.15 Style . . . 352

A ***Coding Standard*** is a set of rules that describe how programs should be coded. The rules say nothing about what the program does but, instead, they specify the way in which the code should be written. How it should be formatted, how names should be chosen, and so on.

Coding standards can affect you in several ways: 625

- If you are responsible for all of the code of a project, you do not need a coding standard. However, it is a good idea to design a coding standard for your own use and to apply it consistently. The code will look better and will be easier to enhance when you rediscover it a few years from now.
- If you are working with a small team, agreeing to a common coding standard and following it will make it easier for team members to understand each others' code and to change roles.
- If you are working for a company, the company may already have a coding standard that you will be required to follow.
- If you are working for a company, and the company does not have a coding standard, you should consider trying to introduce one.
- Open Source projects often define a coding standard. If there is a coding standard, any code that you contribute you follow the standard. If there is no standard, code that you contribute should use the same conventions as existing code, insofar as these conventions can be derived by reading the code.

Coding standards must be strict enough to be useful and relaxed enough that programmers actually use them. It is unreasonable to expect all of the programmers in a large project to follow a strict set of rules exactly. It is more realistic to provide guidelines that have enough freedom to satisfy creative egos.

There are books that define coding standards. For a detailed discussion, see (Sutter and Alexandrescu 2005). For less money, you can get (Misfeldt, Bumgardner, and Gray 2004), but it's not as good. There are also websites with coding standards: the LLVM project has a coding standard[79] and Todd Hoof has written a long (and sometimes controversial) document.[80] Some of the ideas expressed here are taken from these websites. For an example of a coding standard as used in industry, have a look at the Google C++ Style Guide (Google 2017).

The rest of this chapter defines a Coding Standard based loosely on industry practice. It is not definitive: there are a few clear "don't"s, but in most areas, it suggests reasonable alternatives.

626 **Advantages of a Coding Standard:** A Standard is good because programmers:

1. can look at any project code and understand it quickly
2. can get up to speed quickly
3. do not have to develop their own personal style
4. tend to make fewer mistakes in consistent environments
5. have a common enemy—the author of the Standard

627 **Disadvantages of a Coding Standard:** A Standard is stupid because:

1. it was written by someone who is clueless about C++
2. this isn't the way we do things here
3. it enforces too much structure
4. it restricts our creativity

628 Standards are unnecessary because good programmers:

5. are always consistent
6. ignore standards anyway
7. know how to write code without standards
8. will now refuse to import code because it doesn't meet the standard

B.1 Rules

629 Items in this section are *absolute requirements*.

[79] See http://llvm.org/docs/CodingStandards.html

[80] See http://www.possibility.com/Cpp/CppCodingStandard.html

- *Choose a layout convention and use it consistently within a file of source code.*

  Layout conventions are illustrated in Section B.2 below. It doesn't matter which one you choose; the important thing is to use it consistently.

  If layout conventions have already been established for the project that you are working on, follow them. Otherwise, it is a good idea to use the same convention for every source file that you create; it is *essential* that you stick to one convention within a file.

- *Choose a naming convention and use it consistently within a file of source code.*

  As with layout, it is desirable to use a consistent naming convention for every file you create and essential to stick to a convention within a file.

- *Never use two leading underscores for a name.*

  Names of this form are generated by the system. E.g., `__cdecl`.

- *Require clean compiles.*

  Don't ignore compiler warnings. Don't, except in rare and special circumstances, use a `pragma` or other dirty trick to suppress a compiler warning.

  The compiler, which probably has a better understanding of your program than you do, has indicated in a warning message that something *might* be wrong. It is just possible, but very unlikely, that there is nothing wrong and the compiler is confused. In any case, you should inspect the code, make sure that you understand the problem, and fix it. Your goal should always be a *clean compile* – that is, your entire system should compile without *any* warning messages.

- *Don't use macros.*

  Macros are a leftover (hangover?) from C. For most situations in which you might consider using a macro, C++ has a better solution (`inline`, `typedef`, etc.). Macros provide a small amount of convenience but are a significant source of horrible bugs.

B.2 Layout

This section and subsequent sections outline coding conventions that help you to write code that is easy to understand for both you and other programmers. Remember that maintainers will spend more time reading your code than you will spend writing it; try to make their lives easier.

Layout is the organization of white space in your program. It is pointless to argue about which layout convention is "best" but it is important to organize code in a way that maximizes clarity and minimizes potential confusion. Even experienced programmers can look at 630

```
for (int i = 0; i < MAX; ++i)
    a[i] = 0;
    b[i] = 0;
```

for a minute or two without realizing that the effect of this code on array `b` is indeterminate.

B.2.1 Line Length

Source code lines should not extend past column 80. (Some say 78, others say 90, but most people agree that lines should not be excessively long.)

Long lines break when programs are printed, making code much harder to read. Many programs are too long to be printed,[81] but short lines are still important, because they allow the source to be viewed in a window of reasonable width.

B.2.2 Brace Wars

Here are some of the more commonly used conventions for indenting compound statements and other structures: use one for writing and be prepared to read the others.

Opening brace at end of line.

```
void fun(int x, int y)
{
    while (x > y) {
       if (raining) {
           takeUmbrella();
       }
    }
}
```

This convention is called “K&R” (because it was used in *The C Programming Language* by Kernighan and Ritchie) or “kernel” (because it is used in the Unix kernel). 631

The opening brace for a function is directly under the return type of the function. This convention is left over from C. A closing brace is in the same column as the first character of the keyword controlling the statement. To save space,

```
}
else {
```

632 can be written

```
} else {
```

Braces aligned.

```
void fun(int x, int y)
{
    while (x > y)
    {
       if (raining)
       {
           takeUmbrella();
       }
    }
}
```

[81] A printout of Linux on standard letter-sized paper would be a stack of paper about 125 feet high.

This style is called “Allman” (for Eric Allman, who wrote `sendmail` and made other contributions to BSD Unix) or “BSD” or “ANSI”. It requires one more line for each brace-pair than K&R but many people prefer it because it is easier to match braces. Since most modern editors match braces by colour-coding, the advantage is rather small nowadays. 633

Braces aligned and indented.

```
void fun(int x, int y)
    {
    while (x > y)
        {
        if (raining)
            {
            takeUmbrella();
            }
        }
    allDone();
    }
```

This style is called “Whitesmiths” because it was used in ***Whitesmith C***, the first commercial (i.e., outside Bell Labs) C compiler. 634

A control statement, such as `if` or `while` in which the body has only one line should be treated as a special case.

- C++ does not require braces around a block consisting of a single statement. This form is correct: 635

  ```
  if (overflowing)
      mopUp();
  ```

  The problem with this form is that it is easy to add a statement thinking that the indentation determines the scope:

  ```
  if (overflowing)
      mopUp();
      cout << "Dry floor";
  ```

- It is therefore safer to use braces even when there is only one line of code:

  ```
  if (overflowing)
  {
      mopUp();
  }
  ```

- However, the braces create a high proportion of white space. This suggests the following convention: ***if a controlled block has only a single statement, put it on the same line as the control structure***. For our example, this gives:

  ```
  if (overflowing) mopUp();
  ```

  With this convention, it is unlikely that anyone will make the mistake of adding another statement and forgetting the braces. However, it doesn't work if the condition or the statement are long.

For other styles, see Wikipedia.[82]

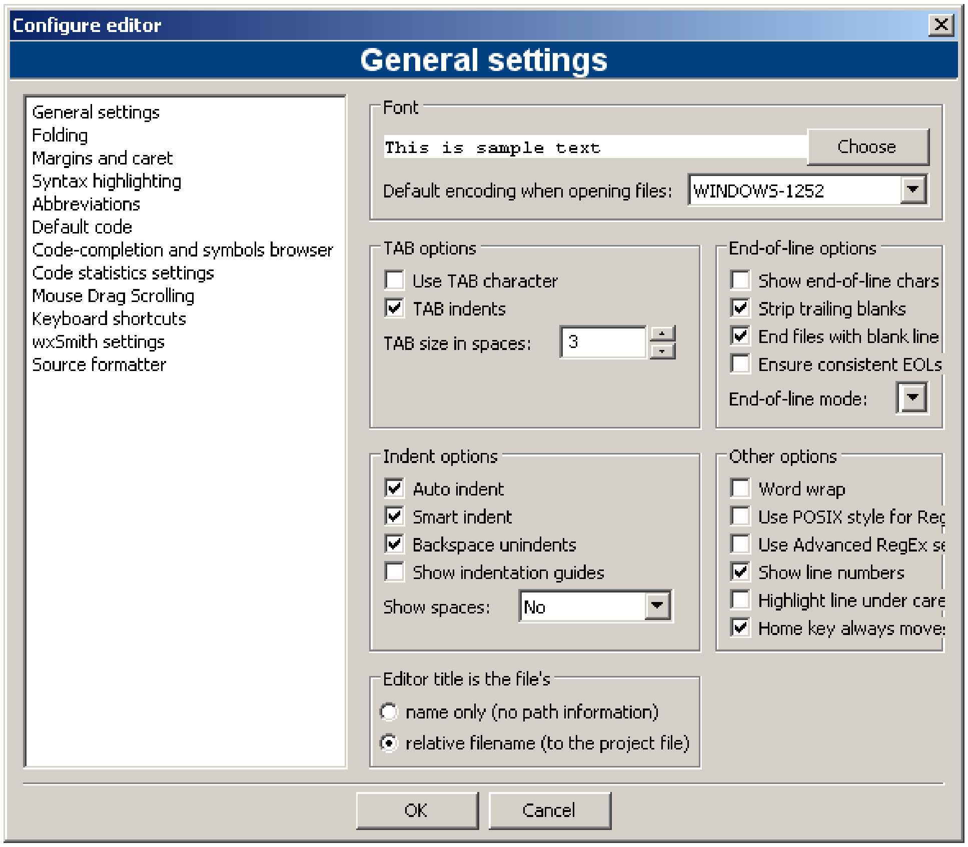
B.2.3 Tabs and Blanks

If there is one thing that can be guaranteed to mess up layout, it is the tab character. It is important to distinguish the tab *key* and the tab *character*. The key is very useful because it allows you to align code consistently with very little effort. The character is a disaster because users interpret it in different ways.

To put it simply: if X writes code with a tab separation of 5 spaces, and Y revises the code with a tab separation of 3 spaces, the result is a mess.

The solution is simple: use the tab key when entering text but never put tab characters in a file.
636 Most IDEs allow you to do this: see Figures 227 and 228.

637 ***Enter tabs, store blanks.***



Font

This is sample text

Default encoding when opening files: WINDOWS-1252

TAB options

- ☐ Use TAB character
- ☑ TAB indents
- TAB size in spaces: 3

End-of-line options

- ☐ Show end-of-line chars
- ☑ Strip trailing blanks
- ☑ End files with blank line
- ☐ Ensure consistent EOLs
- End-of-line mode:

Indent options

- ☑ Auto indent
- ☑ Smart indent
- ☑ Backspace unindents
- ☐ Show indentation guides
- Show spaces: No

Other options

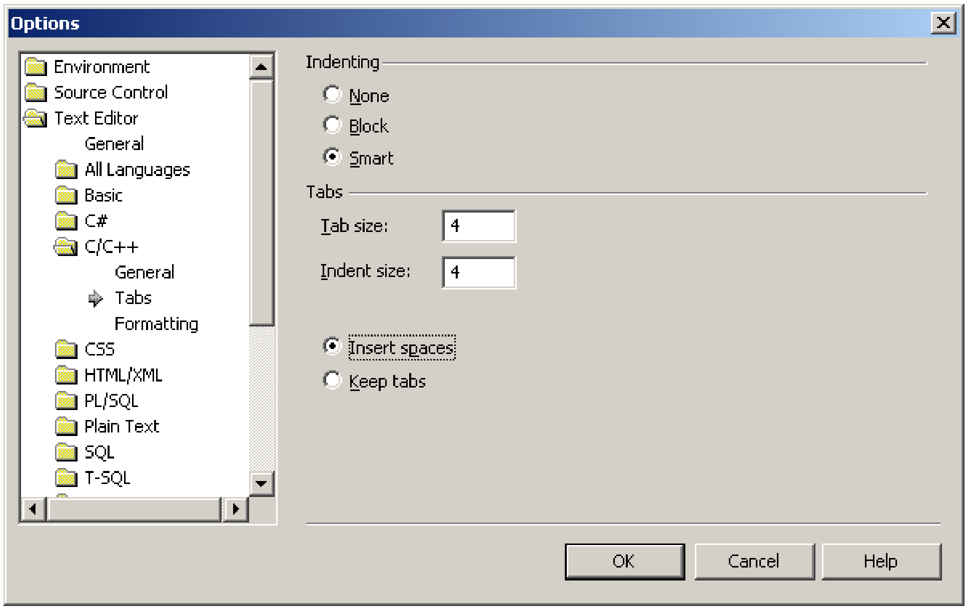
- ☐ Word wrap
- ☐ Use POSIX style for Reç
- ☐ Use Advanced RegEx se
- ☑ Show line numbers
- ☐ Highlight line under care
- ☑ Home key always move:

Editor title is the file's

- ○ name only (no path information)
- ● relative filename (to the project file)

Figure 227: Setting tabs in `Code::Blocks`

[82] `en.wikipedia.org/wiki/Indent_style`



Indenting

○ None

○ Block

● Smart

Tabs

Tab size: 4

Indent size: 4

● Insert spaces

○ Keep tabs

Figure 228: Setting tabs in Visual C++ V7

B.2.4 Automatic Formatting

There are many programs that format source code automatically. Typically, they will provide options so that you can choose one of the indentation styles above and perhaps perform other kinds of customization.

For example, Artistic Style[83] is a customizable formatting program that formats source code in C++ and other languages. It is provided as a plugin for `Code::Blocks`.

Formatting programs provide a quick and easy way of keeping ***your own code*** consistently formatted. However, you should ***never*** run a formatter on code that you do not own (e.g., don't autoformat open source code).

The reason for this restriction is that people often use `diff` to see how a file has been changed. Alternatively, they may rely on their version-control system to view changes. These programs are often sensitive to blanks: that is, they will consider a line to have been changed even if only blanks have been inserted or deleted. Autoformatting a file will generate a large number of spurious change reports that may conceal the real changes.

Don't format code that you don't own.

B.3 Comments

There are two reasons to write a comment:

1. to improve the readability or maintainability of the code; or

[83] `astyle.sourceforge.net`

2. to provide text for a documentation tool.

If you can't do either of these, don't write a comment.

Experienced programmers prefer to read and understand code without the aid of comments. They know, from bitter experience, that the code is a more reliable guide to what the program actually does.

However, you will sometimes come up with tricky code after thinking deeply about a problem and rejecting alternative solutions. In this case, it may help the maintenance programmer (who might be you, six months later) by explaining why you write the code in a particular way.

The only realistic way of keeping documentation consistent with code is to write the documentation as part of the code and then extract it using a tool such as Doxygen.[84] Figure 229 on the next page shows a function commented for Doxygen and Figure 230 on the facing page shows the result as seen by a browser. Note the use of Doxygen markup such as `\b`, `\param`, and `\return`. If you use a tool such as Doxygen, use it consistently: provide documentation for all visible entities in a way that will help others to use the library, or whatever it is, effectively. 638

639

A comment should tell a story.

B.3.1 Disabling blocks of code

Sometimes, you want to remove a block of code from the program temporarily. There are several ways of doing this: 640

- 641 Put "`//`" at the front of each line.

  This method is tedious and error-prone. Some editors make it easier by providing comments to "comment" or "un-comment" a selected block.

- 642 Put "`/*`" before the unwanted code and "`*/`" after it.

  This method requires less writing but it has other problems. If the unwanted block contains "`/* ... */`" style comments, it may not work. (Some compilers accept nested comments, but many do not.)

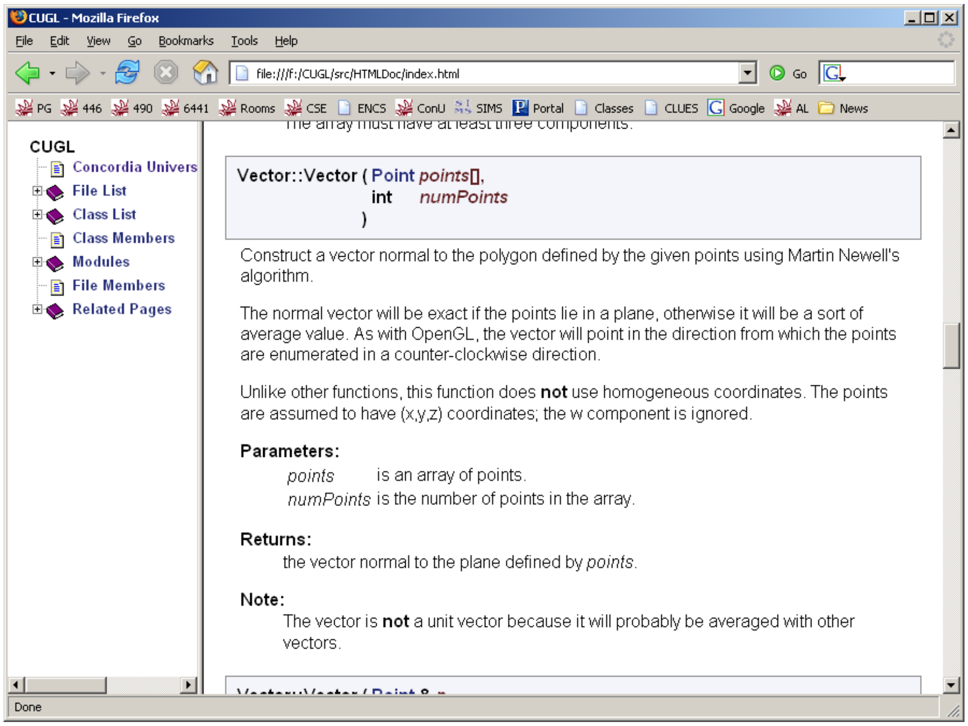
- 643 Put "`#if 0`" before the unwanted code and "`#endif`" after it.

- Put "`#if TRACING`" before the unwanted code and "`#endif`" after it. Use the identifier (`TRACING`) to control inclusion. 644

  This method is safe and effective. You can generalize it by using an identifier rather than 0.

```
/**
 * Construct a vector normal to the polygon defined
 * by the given points using Martin Newell's algorithm.
 * The normal vector will be exact if the points lie in a plane,
 * otherwise it will be a sort of average value.  As with OpenGL,
 * the vector will point in the direction from which the points
 * are enumerated in a counter-clockwise direction.
 *
 * Unlike other functions, this function does \b not use
 * homogeneous coordinates.  The points are assumed to have
 * (x,y,z) coordinates; the w component is ignored.
 * \param points is an array of points.
 * \param numPoints is the number of points in the array.
 * \return the vector normal to the plane defined by \a points.
 * \note The vector is \b not a unit vector because it will probably
 * be averaged with other vectors.
 */
Vector(Point points[], int numPoints);
```

Figure 229: Using Doxygen to document a function


CUGL - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
file:///f:/CUGL/src/HTMLDoc/index.html
Go
PG 446 490 6441 Rooms CSE ENCS ConU SIMS Portal Classes CLUES Google AL News
CUGL
Concordia Univers
File List
Class List
Class Members
Modules
File Members
Related Pages
Done

The array must have at least three components.

Vector::Vector (Point *points*[],
int *numPoints*
)

Construct a vector normal to the polygon defined by the given points using Martin Newell's algorithm.

The normal vector will be exact if the points lie in a plane, otherwise it will be a sort of average value. As with OpenGL, the vector will point in the direction from which the points are enumerated in a counter-clockwise direction.

Unlike other functions, this function does **not** use homogeneous coordinates. The points are assumed to have (x,y,z) coordinates; the w component is ignored.

Parameters:
points is an array of points.
numPoints is the number of points in the array.

Returns:
the vector normal to the plane defined by *points*.

Note:
The vector is **not** a unit vector because it will probably be averaged with other vectors.

Figure 230: Browsing Doxygen documentation

B.3.2 Warnings

Provide clear warnings when the code has some special feature that a maintainer should be aware of. Doxygen has special commands such as `DEPRECATED`, `\TODO`, and `WARNING`. Here are some warnings that you might consider: 645

TODO: reminder that the code is incomplete and must be finished at some time in the future

DEPRECATED: used to indicate that practice has changed but that this code still follows the old practice

WARNING: informs programmers that there is something fishy about this code: it has hidden coupling or dependencies, or doesn't behave in an expected way

BUG or FIXME: indicates that the code is known to be wrong but hasn't been fixed yet

KLUDGE: admits that the code uses dirty tricks that should eventually be replaced by clean and efficient algorithms

COMPILER: warns about compiler-dependencies

B.4 Names

Many different kinds of names appear in C++ programs. Naming conventions make programs easier to read and help to avoid trivial errors. The following conventions are widely used and recommended. 646

- Names of ***constants*** are usually upper case, or start with an upper case letter. Words within a name are separated by underscores.

```
PI   HALF_PI   MAX_NAME_LENGTH   BUFFER_SIZE
```

- Variables and functions usually start with a lower case letter. Words within a name may be distinguished either with upper case letters or underscores.

```
x   isWeird   get_balance   too_many_underscores_may_be_hard_to_read
```

- Names of classes and types (other than the built-in types) usually start with an upper case letter.

```
Widget   ChequingAccount   Name_With_Underscores
```

- Try to use nouns or noun phrases for variables . . . .

```
goal   smallestSoFar   widgetCount
```

- verbs or verb phrases for functions . . . .

```
detoxify   findBlanks   processMarks
```

[84] Doxygen, see http://www.doxygen.org

- and query phrases for predicates.

```
isReady  hasAttributes  goneFishing
```

- Long names are not always better – in fact, a high proportion of long names increase the number of lines of code and make the program harder to read. Shorter names require abbreviations: the important thing is to ensure that the abbreviations are unambiguous in the context.
- Some programmers like to distinguish data members from other variables. E.g., `pos` is a simple variable and `pos_` is a data member.
- Avoid names with a leading underscore. E.g., `_x`.

  Some people (e.g., Herb Sutter) advocate *trailing* underscores (as in `x_`) for some purposes – in particular, data members. I prefer not to use underscore at all, because it looks horrible on some printers, but internal and trailing underscores are definitely less objectionable than leading underscores.

B.4.1 Hungarian Notation

Hungarian Notation (HN) was introduced by Charles Simonyi[85] of Microsoft. People have understood HN in two different ways, leading to great confusion.

In HN, each variable has a short prefix that indicates its type. The confusion has arisen because of the use of the word "type", which refers not to the C++ type of the variable but rather to the function or role of the variable. The notation is called "Hungarian" partly because Simonyi is Hungarian and partly because the notation makes variable names unpronounceable, like Hungarian.[86]

The *wrong* way to use HN is illustrated by Microsoft's own coding conventions in which, for example, the prefix `lpsz` means "long pointer to s with zero terminator", as in `lpszWindowTitle`. This is a *bad* convention because the information provided by `lpsz` is: 647

- redundant. The variable declaration tells you the type of the variable.
- probably wrong. A maintenance programmer who changed the type of the variable would almost certainly not change the name because of the risk of missing some occurrences.
- not helpful. A maintenance programmer probably knows that the variable is a string but needs to know *why* it is a string.

Most of the people who claim not to like HN have misunderstood it and are using it the wrong way.

The *right* way to use HN is to invent prefixes that provide useful information: the prefix indicates the "intent" rather than the "type". In particular, variables with the same C++ type but with different roles often have different prefixes.

[85] Chief designer of some program called "Microsoft Word".

[86] If you don't speak Hungarian, that is.

The “right” way is what Simonyi originally proposed: it was his unfortunate use of the word “type” that led to the confusion. Here is what he says (my italics):[87]

> As suggested above, the concept of “type” in this context is determined by the set of operations that can be applied to a quantity. The test for type equivalence is simple: could the same set of operations be meaningfully applied to the quantities in questions? If so, the types are thought to be the same. If there are operations that apply to a quantity in exclusion of others, the type of the quantity is different.
>
> The concept of “operation” is considered quite generally here; “being the subscript of array `A`” or “being the second parameter of procedure `Position`” are operations on quantity `x` (and A or `Position` as well). *The point is that “integers” `x` and `y` are not of the same type if `Position(x,y)` is legal but `Position(y,x)` is nonsensical.* Here we can also sense how the concepts of type and name merge: `x` is so named because it is an X-coordinate, and it seems that its type is also an X-coordinate. Most programmers probably would have named such a quantity `x`. In this instance, the conventions merely codify and clarify what has been widespread programming practice.

Here are some applications of HN:

- In a graphics application, variables representing coordinates have prefixes `x`, `y`, and, perhaps, `z` (cf. Simonyi’s example above). The position of a car might be `(xCar, yCar, zCar)`. An expression such as `xCar + xBus` probably makes sense but an expression such as `xCar + yCar` probably doesn’t. The call `move(xP,yP)` is more likely to be correct than `move(yP,zP)`. All of these variables have the same type (e.g., `double`); HN is being used to indicate a *direction*.
- Positions in a table are represented by `int`s, but have prefixes `r` for row and `c` for column.
- In a physics application, variables could be prefixed by `l`, `m`, `t`, `v`, etc. This does not enable the compiler to perform checks but it does help programmers to detect obvious mistakes such as adding quantities with different dimensions.

Some people use “Systems Hungarian” to refer to the “wrong” way to use HN and “Apps Hungarian” to refer to the “right” way.

sigil

A ***sigil*** is a “funny character in front of a variable name”.[88] Sigils are used in languages such as Basic (`$` prefixes strings, etc.) and Perl (`$` prefixes scalars, `@` prefixes arrays, `%` prefixes hashes, etc.).

B.4.2 Enumerations

In an enumeration, include the values required and then add one to indicate “none of the above” as an error value. If the error entry is the first, its value will be zero, and it can be checked
648 efficiently. For example:

```
enum TrafficLights { BROKEN, RED, YELLOW, GREEN };
```

[87] See http://msdn.microsoft.com/en-us/library/aa260976%28v=vs.60%29.aspx

[88] See http://en.wikipedia.org/wiki/Sigil_%28computer_programming%29

B.5 Constants and Literals

A *constant* is a fixed value with a name. A *literal* is an unnamed constant. In

```
const double PI = 3.1415926535;
const string TITLE = "Glossary";
```

PI and TITLE are constants, and 3.1415926535 and "Glossary" are literals. 649

Use constants rather than literals whenever possible. Even when a string literal is self-describing, it is often better to give it a name in a constant declaration.

Numeric literals that are not part of a constant definition are called “magic numbers”. This code: 650

```
for (int line = 0; line < 66; ++line)
{
   ...
   if (pos > 66)
   {
      ...
```

contains two magic numbers that happen to have the same value. We cannot tell what the numbers, nor even whether they mean the same thing. Things become clearer if the magic numbers are replaced by constants: 651

```
for (int line = 0; line < LINES_ON_PAGE; ++line)
{
   ...
   if (pos > MAX_BEERS)
   {
      ...
```

Magic numbers make a program not only hard to understand but also hard to change. Imagine the effect of globally changing “66” to “68” in the first extract above!

Don’t use magic numbers.

B.6 `const`

Ensure that all your programs are `const`*-correct.*

This means, roughly, use `const` wherever possible: 652

- If you declare `T x` and `x` is not going to be changed, write `const T x` instead.
- If a function does not change the value of its argument, declare the parameter as `const T & x`.

- If a member function does not change any data members of its object, or call a non-`const` function, declare the function as `const`.
- If you are traversing a container without changing its contents, use a `const_iterator`.

Sometimes, inserting `const` will cause a compiler error, typically of the form "cannot convert from `T` to `const T`." One way to remove the error is to take out a `const`, but this is the *wrong way*. The correct way to remove the error is usually to *put in* a `const` somewhere. For example,
653 suppose that the compiler produces a `const` error for the following code:

```
int totalSize = 0;
for (     vector<int>::const_iterator it = store.begin();
          it != store.end();
          ++it)
    totalSize += it->getSize();
```

The correction is *not* to change `const_iterator` to `iterator` but to add `const` to the declaration of `getSize`. (If `getSize` changes the state of the object, then you're really in trouble!)

B.6.1 The keyword `mutable`

654 Consider this code:

```
class Point
{
public:
    double dist()
    {
        ++uses;
        return(sqrt(sqr(x) + sqr(y)));
    }
    ....
private:
    double x;
    double y;
    unsigned long uses;
};
```

The idea is that `uses` counts the number of times various functions of a particular object[89] are called, but is not otherwise relevant to the state of the object. In this situation, we would like to declare `dist` as `const`, but the statement `++uses` prevents us from doing so.

The solution is to declare `uses` to be `mutable`. Making a variable `mutable` means "the value of this variable can be changed without changing the logical `const`-ness of the object". The new
655 class declaration looks like this:

656

[89] If we wanted to count how many times the function was used by all instances, we would declare `uses` to be `static`.

```
class Point
{
public:
    double dist() const
    {
        ++uses;
        return(sqrt(sqr(x) + sqr(y)));
    }
    ....
private:
    double x;
    double y;
    mutable unsigned long uses;
};
```

B.6.2 const with pointers

There are four ways of declaring a pointer.

1. Both the pointer and the object pointed to may change: 657

   ```
   char * p = &myChar;

   p = &anotherChar;          // OK
   *p = 'x';                  // OK
   ```

2. The pointer can be changed but the object pointed to cannot be changed. The way to remember this is to note that `p` is a pointer to a `const char`.

   ```
   const char * p = &myChar;

   p = &anotherChar;          // OK
   *p = 'x';                  // Illegal!
   ```

3. The pointer cannot be changed but the object pointed to can be. The way to remember this is to note that it says “`const p`”.

   ```
   char * const p = &myChar;

   *p = 'x';                   // OK
   p = &anotherChar;           // Illegal
   ```

4. Neither the pointer nor the object pointed to can be changed:

   ```
   const char * const p = &myChar;

   *p = 'x';                   // Illegal
   p = &anotherChar;           // Illegal
   ```

B.7 Variables

Declare variables as locally as possible. The scope of a variable should not be longer than necessary.

Always initialize variables at the point of declaration or as soon as possible thereafter.

B.8 Expressions

Do not write expressions that have more than one dot or more than one arrow. These are both bad: 658

```
university.faculty.department.prof
city->restaurant->menu->fries
```

Consider the first example. A programmer who wishes to determine the value of this expression must:

- find the type of `university`
- find the type of `faculty` in the declaration of the class of `university`
- find the type of `department` in the declaration of the class of `faculty`
- find the type of `prof` in the declaration of the class of `department`

Maintaining code like this is even worse than reading it. Each expression creates a dependency across four classes!

Law of Demeter

The *Law of Demeter*[90] is a generalization of this principle.

B.9 Exceptions

Experience suggests that complex exception handling schemes, using inheritance hierarchies of exception classes, etc., do not work well in practice. Keep exception classes to a minimum – often one class is enough for a medium-sized application – and do not rely on inheritance relationships between exception classes.

[90] See http://en.wikipedia.org/wiki/Law_of_Demeter

B.10 Functions

Design functions that do *one* thing *well.* People who call your function should no exactly what to expect and the function should fulfill their expectations. A good rule of thumb is: you should be able to explain what a function does in one simple sentence.

In general, a function should *either* return a value *or* have an effect. (Member functions that do this are called *accessors* and *mutators*, respectively.)

A function that has an effect *and* returns a value is difficult to use well. The value must be used in some way, as the right side of an assignment, a term in an expression, or a condition in an `if` or `while` statement. Since the function also has an effect, we immediately have expressions and conditions with side-effects. Understanding and maintaining code is harder in the presence of side-effects. Worse, a maintainer may not even realize that there are side-effects and introduce errors.

B.10.1 Parameters

Choose parameter passing modes that are semantically correct and reasonably efficient. The most common modes are: value, reference, and constant reference.

B.10.2 Operators

Increment/decrement operators. Use the prefix versions (`++i` and `--i`) rather than the postfix versions (`i++` and `i--`): they are never slower and may be faster.

The only reason to use the postfix versions is when you need the old value. But even this usage is deprecated nowadays.

Overloading operators. Operator overloading can be used very effectively, but it can also be misused. The operators themselves are small symbols and they do not stand out in code. Operators should always be defined so that their usage is "natural": a programmer reading "`x + y`" can safely assume that `x` and `y` are being added—in some sense.

- *If you overload a familiar operator, programmers will expect it to work in a familiar way.*

  For example, suppose you provide `operator+` for class `T` so that programmers can write: 659

  ```
  T a = .... ;
  T b = .... ;
  T c = a + b;
  ```

  Programmers will then expect (≡ stands for "has the same value or effect as"):

  $$\texttt{a + b} \equiv \texttt{b + a}$$

  $$\texttt{a += b} \equiv \texttt{a = a + b}$$

If `T(0)` is allowed, they might also expect:

$$\begin{aligned} \texttt{a + 0} &\equiv \texttt{a} \\ \texttt{0 + a} &\equiv \texttt{a} \end{aligned}$$

If `T` is a mathematical sort of object, they might even expect `operator-` in both unary and binary forms:

$$\begin{aligned} \texttt{a - b} &\equiv \texttt{a + (-b)} \\ \texttt{-(-a)} &\equiv \texttt{a} \\ \texttt{a - a} &\equiv \texttt{0} \end{aligned}$$

There are a small number of exceptions to this guideline. For example, it is a common convention to use `+` to concatenate strings, even though concatenation is neither commutative nor associative. In general, however, if you provide one operator, you should provide the other operators that might reasonably be expected to go with it.

- *Use free functions, rather than member functions, for overloaded binary operators.*

660 The operator declared as

```
class T
{
public:
    T operator+(const T & other);
    ....
};
```

is *asymmetric*: `a + b` is equivalent to `a.operator+(b)` and is not necessarily equivalent to `b + a`. To avoid confusing your users, declare operators like this

```
T operator+(const T & lhs, const T & rhs);
```

and make it a `friend` of class `T` if you have to.

- *Don't overload* `&&`*,* `||`*, and* `,` *(comma).*

It is impossible to obtain the semantics that programmers expect for these operators (e.g., lazy evaluation of the conditional operators, evaluation order for comma).

- *Don't depend on the order of evaluation of arguments.*

Although this is a useful guideline to follow, you will not often have problems with simple functions. But you do have to be careful with input and output. The program shown in
661 Figure 231 on the next page prints

```
23    23    23
```

```
#include <iostream>

using namespace std;

static string buffer;

const char * convert(int k)
{
   buffer = "   ";
   do
   {
      buffer.insert(0, 1, k % 10 + '0');
      k /= 10;
   } while (k > 0);
   return buffer.c_str();
}

int main()
{
    cout << convert(23) << convert(46) << convert(79) << endl;
    return 0;
}
```

Figure 231: A dangerous programming style

B.11 Classes

A programmer who is using a class should need to see only the declaration of the class. Thus you should pay particular attention to documenting class declarations. Explain what an instance of the class is for and what it does. Provide an adequate explanation of the effect of each member function of the class. If there are constraints on the sequence in which functions must be called, mention them.

B.11.1 Organization

Use a consistent order for methods and data. For example: 662

public: 663

- `friend` classes
- Enumerations
- Lifecycle: constructors and destructors
- Operators
- Operations: the "business logic"

- Inquiries: `const` functions that give information about the object
- Accessors: `const` functions that return attributes of the object (if really necessary)
- Mutators: functions that change attributes of the object (if ***really*** necessary)
- `friend` functions

protected:

- Accessors: `const` functions that return private attributes needed by derived classes
- Mutators: functions that change private data for derived classes

private:

- Enumerations
- Instance data

B.11.2 Required Methods

The compiler will provide default versions of: a default constructor, a copy constructor, the destructor, and an assignment operator. if the class contains no pointer members, it is quite possible that all of these defaults will do the right thing.

If you rely on the compiler-defined defaults, it is good practice to write a comment saying so. Otherwise, maintainers will assume that you forgot them.

664 If any of these methods has to be specially coded for some reason, ***it is likely that special versions of the others will be needed as well***. Sometimes this rule is stated as:

If you need any one of: destructor; copy constructor; or assignment operator, you probably need them all.

In particular:

- Every class must have a default constructor, either provided by the compiler or written by the owner of the class.
- Every class that might be used as a base class must have a `virtual` destructor.

B.11.3 Initialization

For simple classes with a few constructors, use initializers: 665

```
class Point
{
public:
   Point(double x, double y, double z) : x_(x), y_(y), z_(z) {}
   ...
```

Larger, more complex, classes may have a number of constructors. To ensure that all objects are initialized consistently, provide an initialization method and call it from each constructor.

B.11.4 Composition and Inheritance

Use composition rather than inheritance whenever possible.

B.11.5 Abstract Base Classes

Introduce abstract base classes (a.k.a. "ABCs") to reduce coupling.[91]

ABCs can be very simple. For example, a graphics application might have an ABC 666

```
class Drawable
{
public:
    void draw() = 0;
};
```

Any programmer working on a class that has some screen representation inherits `Drawable` and implements `draw()`. Pointers to instances of these classes would be stored, perhaps, in a `vector<Drawable*>`. All the `display()` function has to do is walk this vector calling `draw()` for each object.

B.12 Libraries

Compile system components as libraries, especially if they are widely used.

When code is compiled normally, most linkers include every function definition, whether the function is used or not. But when the linker scans a library, it includes only those functions that are actually called. (The rule is applied transitively, so that if `A` is needed, and `A` calls `B`, then both `A` and `B` are included in the executable.) Consequently, good use of libraries can reduce the size of the application. Libraries also relieve the programmer of the worry "Does anyone actually need this?".

[91] Java programmers: think "interfaces".

B.13 Files

In C++ programming, a header file usually contains a class declaration. The name of the file should normally be the same as the name of the class.

For each class, there is a corresponding implementation file. This file, too, should have the same name as the class.

For example, the declaration of class `Foo` would normally be in `foo.h` and the implementation would be in `foo.cpp`.

These rules may be broken in particular cases. For example, if class `Bar` is used ***only*** in the implementation of `Foo`, then the declaration and the implementation of class `Bar` might be included in `foo.cpp`, where they are hidden from the rest of the program.

B.13.1 Documentation

Each source file should start with some kind of header. The header should contain some or all of
667 the following items:

- The first line of the header should contain the file name. This makes it easy to locate files when looking through printed code or edit windows.

  (Tabbed editors usually display the file name in the tab, making this rule less important.)

- The name of the owner(s) of the code.
- Software licencing information.
- A revision history.
- Remarks warning maintainers about issues specific to this code.
- A summary of the contents of the file.

B.13.2 Header Files

Always include "guards" to ensure that the information in a header file is read once only.
668 Traditional guards have the following form. Don't forget the **`#endif`**!

```
#ifndef FOO_H
#define FOO_H
...
#endif
```

Most modern compilers recognize the directive

```
#pragma once
```

which requires less writing and may be more efficient. However, unlike the include guards, it is compiler-specific and therefore discouraged in several style guides. For example, the Google C++ style guide (Google 2017) says *"Do not use #pragma once; instead use the standard Google include guards."*

In general,

`#include` *as little as possible.*

B.13.3 Implementation Files

The first **`#include`** directive in an implementation file should read the corresponding header file.

B.13.4 Directory Structure

Organize files related to the project into a logical structure. Use conventional names when it is feasible to do so. For example: **`bin`**, **`doc`**, **`lib`**, **`pkg`**, **`src`**, **`install`**, etc.

B.14 Development

B.14.1 Portability

Write code to be portable whenever possible. 669

Code is ***portable*** if it can be compiled and executed on different platforms with few changes or, ideally, no changes. The fewer changes that are required, the more portable the code.

You can write portable code if you know the C++ Standard and you know about any specific extensions that your compiler provides (and that you don't use).

Unfortunately, not many programmers have committed the Standard (over 1000 pages) to memory and most programmers are somewhat vague about the capabilities of their compiler(s). Consequently, a more practical rule for writing code is to use simple, well-understood features as much as possible and to check carefully when using features you are not sure about.

B.14.2 Change management

Use a source code management system and make sure that all programmers on the project know how to use it properly – and do so.

B.14.3 Automation

Automate whenever possible. Ideally, one keystroke should be sufficient to build the application, run a set of standard tests, update documentation and web pages, and record the build in a logfile. It is a good idea to start writing scripts, using `ant` or some other tool, at an early stage of development. Tools like *Jenkins*[92] allow you to easily setup an automated build server for ***continuous integration***.[93]

B.15 Style

Figure 232 on the facing page shows a ray-tracing program written in C by Anders Gavare.[94] The author has carefully avoided using any keywords and has taken a number of other steps to make this program difficult to read. As a result, it was one of the winners of the 2004 Obfuscated C programming contest.

Gavare's coding style is impressive in its own way, but is not a style to be imitated ouside an obfuscated coding contest. Instead, you should try to develop a clear, simple coding style. 670

- *Prefer standard idioms to clever tricks.*

  Experienced programmers recognize idioms. In fact, idioms, like patterns, are one of the ways in which programmers communicate. A programmer who sees

  ```
  for (int i = 0; i != MAX; ++i)
      ....
  ```

  knows immediately what is happening. The same programmer looking at

  ```
  for (double d = MAX; d > MIN; d /= 1.1)
      ....
  ```

  has to think twice or more.

- *Prefer clarity to efficiency.*

  Efficiency will save you a few machine cycles – not a big deal with the cheap and fast hardware of today and the even cheaper and faster hardware of tomorrow. Clarity will save you hours of maintenance by the expensive programmers of today and the even more expensive programmers of tomorrow.

- *Use `assert` liberally.*

- *Get it right before making it fast.*

  If performance really is important, and it really does require obscure code, then document what you are doing and the reasons for doing it.

- *Design simple components connected by clean interfaces.*

  Two or three simple components that talk to each other will be easier to maintain and more likely to be reused than one big mess.

[92] Jenkins, http://jenkins-ci.org/

[93] See https://en.wikipedia.org/wiki/Continuous_integration

[94] See http://www1.us.ioccc.org/years.html#2004.

- *Design for the future . . . . but not too far ahead.*

  When you write a program, it is wise to think of possible enhancements and to code in a way that will make them simple to implement. However, it is also possible to take this idea to extremes, and spend so much time thinking ahead that you never get the job done.

- *Read good books.*

  You can learn a lot about programming just by writing programs. You can learn even more by consulting the experts – see Appendix A on page 327 for some recommendations. 671

```
X=1024; Y=768; A=3;

J=0;K=-10;L=-7;M=1296;N=36;O=255;P=9;_=1<<15;E;S;C;D;F(b){E="1""111886:6:??AAF"
"FHHMMOO55557799@@>>>BBBGGIIKK"[b]-64;C="C@=::C@@==@=:C@=:C@=:C5""31/513/5131/"
"31/531/53"[b ]-64;S=b<22?9:0;D=2;}I(x,Y,X){Y?(X^=Y,X*X>x?(X^=Y):0,  I (x,Y/2,X
)):(E=X);     }H(x){I(x,    _,0);}p;q(        c,x,y,z,k,l,m,a,          b){F(c
);x-=E*M     ;y-=S*M           ;z-=C*M         ;b=x*       x/M+         y*y/M+z
*z/M-D*D    *M;a=-x              *k/M     -y*l/M-z        *m/M;    p=((b=a*a/M-
b)>=0?(I    (b*M,_      ,0),b    =E,      a+(a>b      ?-b:b)):     -1.0);}Z;W;o
(c,x,y,     z,k,l,    m,a){Z=!    c?      -1:Z;c     <44?(q(c,x         ,y,z,k,
l,m,0,0     ),(p>      0&&c!=     a&&        (p<W         ||Z<0)          )?(W=
p,Z=c):     0,o(c+         1,    x,y,z,        k,l,          m,a)):0     ;}Q;T;
U;u;v;w    ;n(e,f,g,            h,i,j,d,a,    b,V){o(0      ,e,f,g,h,i,j,a);d>0
&&Z>=0? (e+=h*W/M,f+=i*W/M,g+=j*W/M,F(Z),u=e-E*M,v=f-S*M,w=g-C*M,b=(-2*u-2*v+w)
/3,H(u*u+v*v+w*w),b/=D,b*=b,b*=200,b/=(M*M),V=Z,E!=0?(u=-u*M/E,v=-v*M/E,w=-w*M/
E):0,E=(h*u+i*v+j*w)/M,h-=u*E/(M/2),i-=v*E/(M/2),j-=w*E/(M/2),n(e,f,g,h,i,j,d-1
,Z,0,0),Q/=2,T/=2,       U/=2,V=V<22?7:  (V<30?1:(V<38?2:(V<44?4:(V==44?6:3))))
,Q+=V&1?b:0,T                +=V&2?b        :0,U+=V    &4?b:0)     :(d==P?(g+=2
,j=g>0?g/8:g/     20):0,j    >0?(U=     j    *j/M,Q      =255-    250*U/M,T=255
-150*U/M,U=255    -100    *U/M):(U    =j*j     /M,U<M           /5?(Q=255-210*U
/M,T=255-435*U           /M,U=255    -720*      U/M):(U       -=M/5,Q=213-110*U
/M,T=168-113*U    /       M,U=111               -85*U/M)      ),d!=P?(Q/=2,T/=2
,U/=2):0);Q=Q<    0?0:      Q>O?     O:          Q;T=T<0?    0:T>O?O:T;U=U<0?0:
U>O?O:U;}R;G;B    ;t(x,y     ,a,    b){n(M*J+M    *40*(A*x   +a)/X/A-M*20,M*K,M
*L-M*30*(A*y+b)/Y/A+M*15,0,M,0,P,  -1,0,0);R+=Q    ;G+=T;B   +=U;++a<A?t(x,y,a,
b):(++b<A?t(x,y,0,b):0);}r(x,y){R=G=B=0;t(x,y,0,0);x<X?(printf("%c%c%c",R/A/A,G
/A/A,B/A/A),r(x+1,y)):0;}s(y){r(0,--y?s(y),y:y);}main(){printf("P6\n%i %i\n255"
"\n",X,Y);s(Y);}
```

Figure 232: A ray tracing program

References

Google (2017). Google C++ Style Guide. http://google.github.io/styleguide/cppguide.html. Last accessed 24.10.2017.

Misfeldt, T., G. Bumgardner, and A. Gray (2004). *The Elements of C++ Style*. Cambridge University Press.

Sutter, H. and A. Alexandrescu (2005). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. C++ In-Depth Series. Addison-Wesley.

C Glossary

`<<` The *insert* operator (or *inserter* for short), used to "insert" (i.e., write) objects to a stream. With integer arguments, this operator performs a left shift.

`>>` The *extract* operator (or *extractor* for short), used to "extract" (i.e., read) objects from a stream. With integer arguments, this operator performs a right shift.

`?:` The ternary ***conditional operator*** that can be used instead of `if`...`else` statements.

Abstract base class A class in which at least one function is ***pure virtual*** – that is, has a declaration of the form

`virtual` ⟨*type*⟩ ⟨*name*⟩ `(` ⟨*parameters*⟩ `) = 0;`

Accessor A member function that returns information about an object but does not change the object's state. Also called ***inspector***.

Argument A value passed to a ***function*** when the function is called. Also called "actual ***parameter***".

Base class A ***class*** that is intended to be used as a *base* for other classes to ***inherit*** from. Also called "superclass" or "parent class".

Binary operator An ***operator*** with two parameters. Most binary operators are ***infix operators***.

Binding Associating a name and an attribute. The attribute may be a value (e.g., the name of a constant is *bound* to the value of the constant), a type (e.g., a declaration *binds* a type to the name of a variable), or some other property.

Bridge A design pattern that enables the separation of an interface and its implementation. See ***Handle/Body class***.

Buffer A temporary space used to store data, often a ***string***, before it is sent somewhere else, e.g., written to a ***file***.

Build The result of compiling a complete system, with all translation units recompiled.

Building The process of compiling a ***build***.

Casting An explicit conversion from one type to another. In C, `T(e)` is a cast that converts the expression `e` to the type `T`. In C++, the conversion is performed by `static_cast<T>(e)` if the compiler can check the validity of the conversion or `dynamic_cast<T>(e)` if a run-time check is required. There is also `reinterpret_cast<T>(e)` which simply treats the bits of `e` as if they represented a `T`, without any sanity checking.

Class A named collection of functions, data, and possibly other attributes. ***Object***s are ***instance***s of classes.

Command A design pattern that allows a command, action, or sequence of commands or actions, to be stored for now and applied at a later time.

Compilation Unit The source code that is processed by one execution of the compiler. A typical compilation unit consists of a single implementation file; the compiler will of course read `#include`d files as well. A complete program usually consists of several compilation units.

Compile-time The time at which the source text is processed by the compiler. E.g., a "compile-time error" is an error detected during compilation. See ***static***.

Composite A design pattern that allows composite objects (i.e., objects with sub-components) to participate in a hierarchy.

Composition The process of building a subsystem in which classes contain instances of other classes as members. Also called ***layering***.

Constant A value that does not change. After the declaration "`const int MAX = 10;`", the value of `MAX` cannot change. The "`10`" in this declaration is also "constant" but is technically a ***literal***.

Constructor A special function that first allocates memory and then initializes an object. A constructor for class `C` is called `C`, but constructors may be overloaded.

Container An object that is used to store a collection of objects and provide various ways to access them.

Conversion The transformation of a value of one type into a value of another type. Some conversions (e.g., `char` to `int`) are performed implicitly by the compiler; others must be explicitly requested by the programmer as ***casts***.

Copy constructor The constructor that is used to make a copy of an object for passing or returning by value. The copy constructor for class `C` has prototype `C::C(const C & c)`.

ctor Common abbreviation for ***constructor***.

Curiously Recurring Template Pattern (CRTP) A pattern that enables a base class to provide consistent behaviour to a variety of derived classes.

Declaration A declaration introduces a new name into a program. E.g., "`int n;`" is a declaration. Many declarations (including this example) are also ***definition***s. A declaration such as "`void f()`" is not a definition because the function `f` is defined somewhere else. A name can be declared many times in different parts of a program.

Deep copy A copy of a pointer structure in which nodes are copied recursively. See also ***shallow copy***.

Default constructor A constructor that requires no arguments. There are three possible kinds of default constructor for a class `C`:

- A constructor with no parameters: `C::C() { .... }`.
- A constructor in which all parameters have default values: e.g.,

  ```
  C::C(int min = 0, double max = 0) { .... }
  ```

- If the class does not declare any constructors, the compiler will generate a default constructor that allocates space for the object and does nothing else.

Definition A definition associates stored data with a name. Only one definition is allowed for each name.

Dependency A relationship between two program components such that one requires (or *depends on*) the other in order to function or compile.

Dereference Convert a "reference" to the corresponding value. Confusingly, dereferencing in C++ applies to pointers, not references. If `p` points to an object, `*p` is the object pointed to. `(*p).m` is usually abbreviated to `p->m`.

Dereferencing Operator The dereferencing operator is `*`, and it is used to ***dereference*** a pointer. `->` can also be considered as a dereferencing operator.

Derived Class A class that ***inherit***s from a ***base class***. Derived classes are sometimes called "subclasses" or "child" classes.

Destructor A special function that performs any necessary cleaning up before deallocating the memory for an object. A class `C` has at most one destructor that is named ∼`C` and has no parameters.

Directive An instruction to the compiler that does not directly affect the semantics of the program. E.g., `#include <...>`.

Downcast A ***cast*** from a base class type to a derived class type. Downcasting is potentially unsafe, because the result may be a pointer or reference to an object that does not provide the functionality expected by the compiler.

DRY stands for "don't repeat yourself". Every entity in a system should have a single, unambiguous point of definition.

dtor Common abbreviation for ***destructor***.

Dynamic A way of saying that an event occurs during execution of the program, that is, at ***run-time***. E.g., "dynamic binding".

Dynamic binding Choosing which function to call at ***run-time***. The candidate functions all have the same name but are in different ***derived classes***. C++ uses dynamic binding only for ***virtual*** functions.

Expression A combination of ***function***s, ***operator***s, and ***operand***s that can be evaluated to yield a value. E.g., `2+2`.

Extractor The operator `>>` used to ***extract*** objects from a ***stream***.

File A collection of data stored outside the computer, e.g., on a disk, CD, memory-stick, etc.

File scope The ***scope*** consisting of an entire source file. A name that is declared outside any braces (`{...}`) is "declared at (or with) file scope".

Forward declaration An incomplete declaration of a name that provides enough information to use the name. The complete declaration must appear elsewhere in the program.

For example, after the forward declaration

```
class Widget;
```

it is legal to declare references and pointers to variables of type `Widget`. It is not legal to declare actual instances of `Widget` because this would require knowing the size of a `Widget`, which the forward declaration does not provide.

Friend A function or a class that is not a class member but nevertheless has access to private members of a class.

Function A code module that can be invoked by name. Functions may accept ***argument***s and may return a result. E.g., `sqrt` is a function and `sqrt(2)` is a function invocation that should yield $\sqrt{2}$.

Handle/Body Class A class that contains a "handle" (or pointer) to an implementation, separating the interface and the implementation. See ***Bridge***.

Header file A file containing declarations only that is intended to be `#include`d in one or more ***implementation files***.

Hiding A function may be declared in a derived class in such a way as to prevent access to a function in the base class, in which case it is said to *hide* the base class function. This happens, for example, if the base class function is non-virtual and the functions have different parameter lists.

Identity comparison Two objects that are "identically equal" are in fact the same object. The actual comparison is usually performed between pointers or references so that, for example, `p1 == p2` is true if the pointers `p1` and `p2` point to the same object, in which case the objects `*p1` and `*p2` are identical. Compare ***value comparison***.

Implementation file A file containing definitions of variables and functions.

Implementation inheritance A derived class that calls functions defined in its base class is said to "inherit implementation". Compare ***interface inheritance***.

Infix operator An ***operator*** that is written between its arguments. For example, `+`, as used in `a+b`.

Inherit When a ***derived class*** declares its ***base class*** (or classes), it is given access to the ***public*** and ***protected*** features of that class; we say that it *inherits* the features.

Inserter The operator `<<` used to ***insert*** objects into a ***stream***.

Inspector A member function that returns information about an object but does not change the object's state. Also called ***accessor***.

Instance An object that belongs to a class.

Instantiate Create a particular entity from a general pattern. E.g., we instantiate a ***class*** to obtain an ***object***.

Interface inheritance A derived class that inherits function declarations but *not* their implementations is said to "inherit and interface".

Iterator An object that is intended to process the elements of a ***container*** in a particular order.

Layering The process of building a subsystem in which classes contain instances of other classes as members. Also called ***composition***.

Library A collection of ***function***s that form a coherent group. E.g., a linear algebra library.

Link-time The time at which source text is processed by the linker. The distinction between ***compile-time*** and link-time is important mainly for classifying errors: errors within a ***translation unit*** are detected at compile-time but inconsistencies between translation units may not be detected until link-time.

Liskov Substitution Principle "Subclasses must be usable through the base class interface without the need for the user to know the difference."

Literal A ***constant*** that has an immediate value (i.e., is not a name). E.g., `42`, `3.14159`, `'x'`, `"C++"`.

Lvalue An expression that may be used on the left side of an assignment operator. Technically, an Lvalue is a place where something can be stored, i.e., an address. E.g., `x`, `a[i]`. See ***Rvalue***.

Main program The unique ***function*** called `main` that must appear in any C++ program that can run by itself (as opposed to a ***library***).

Mangling When a function name is ***overload***ed, there will be several functions with the same name. C++ generates unique names for the ***linker***, a process informally referred to as *name mangling*.

Member A data item or a function that is introduced within a class declaration. In a C++ context, we usually refer to "data members" and "member functions". In a general object-oriented programming context, data members are called "instance variables" or "attributes", and member functions are called "methods". The term "feature" is sometimes used to mean data or function.

Mutator A member function that changes the state of the object but (usually) does not return a value.

Namespace A named collection of names.

Object An area of memory that stores data in a format defined by the ***class*** of which this object is an ***instance***.

Operand A component of an ***expression*** that has a value. In the expression `(m+7)/2`, `m`, `7`, `(m+7)`, and `2` are all operands.

Operator A ***function*** that has special syntax. In C++, `k+2` is simply an abbreviation for `operator+(k,2)`, showing that `+` is actually a function whose full name is `operator+`.

Overload A single function name may be used to represent more than one actual function, provided that the functions have different parameter types. Such a name is *overloaded*. Informally, people refer to an "overloaded function" but, strictly speaking, it is the name that is overloaded, not the function.

Override If a base class provides a virtual function `f` and a derived class provides another definition of the same function with the same parameters, the second definition is said to *override* or ***redefine*** the first definition.

Parameter A formal name in a function ***declaration*** or ***definition***. E.g., in `double sqrt(double x)`, "`double x`" is a parameter. Parameters are sometimes called "formal parameters" or even "formal arguments".

Pimpl Stands for "pointer to implementation". An implementation idiom where classes are referenced by pointers to reduce compile time.

Polymorphism A name that may refer to different objects (literally, "have many shapes"). C++ has three main kinds of polymorphism: ***overload***ing, ***dynamic binding***, and ***template***s.

Private A ***member*** of a ***class*** that can be accessed only by instances of the class.

Protected A ***member*** of a ***class*** that can be accessed only by instances of the class and its ***derived*** classes.

Public A ***member*** of a ***class*** that can be accessed by anyone via an instance of the class.

Pure virtual A virtual function without an implementation is called a *pure virtual function*. The definition of a pure virtual function has the special form

```
virtual ⟨type⟩ ⟨name⟩ ( ⟨parameters⟩ ) = 0;
```

Qualifier A class, type, or ***namespace*** name used to indicate the scope that the compiler should use to find a name. In `std::endl`, `std` is a qualifier indicating that the compiler should look for `endl` in `namespace std`.

Qualified name A name with one or more ***qualifier***s. E.g., `MathLib::Trig::cos`.

RAII stands for *Resource Acquisition Is Initialization*. If an object is constructed within a scope, its destructor will be called when control leaves the scope. Consequently, the constructor (and destructor) can be used to acquire (and release) resources safely, even in the presence of exceptions.

Redefine If a base class provides a virtual function `f` and a derived class provides another definition of the same function with the same parameters, the second definition is said to *redefine* or ***override*** the first definition.

Return type The ***type*** of value returned by a ***function***. It is usually the first component of a ***function definition*** or ***declaration***. E.g., the return type of `double sqrt(double x)` is `double`.

Run-time The time at which the program is executed. E.g., memory management errors are usually not detected until run-time.

Rvalue An expression that can be used only on the right of an assignment operator. In practice, this means expressions that cannot represent a storage location. E.g., `99`, `delta + 0.001`. See ***Lvalue***.

Scope A textual region of a program. An entire source file forms a scope called ***file scope***. Other scopes are enclosed in braces: `{ .... }`.

Scoping operator This is the technical name of the symbol "`::`" used between a ***qualifier*** and a name.

Shallow copy A copy of a pointer structure that copies the root node only. Pointers in the copied root node point to the subtrees of the root of the original tree.

Singleton A design pattern that ensures uniqueness of an object.

Slicing Suppose that `d` is the name of an instance of a derived class `D` and `b` is a variable declared as an instance of the base class, `B`, of `D`. After the assignment `b = d`, `b` has `B` data only, its `D` parts have been *sliced* off.

Standard library The library that contains all of the commonly used functions of C++; or everything in the ***namespace*** `std`.

Standard template library The library that contains a collection of useful data structures and algorithms. Usually referred to as "STL".

Standard stream A stream that is defined in the standard library. The most-used standard streams are:

- `cout` — "see-out", an output stream usually sent to the console window
- `cerr` — "see-error", an output stream usually sent to the console window
- `cin` — "see-in", an stream usually read from the keyboard

State A design pattern that allows an object to appear to have multiple states and to execute state transitions.

Statement A unit of code that performs an action. A simple statement ends with a semicolon ("`;`"). A compound statement is delimited by braces (`{ .... }`).

Static A way of saying that an event occurs at ***compile-time*** or ***link-time***, i.e., before the program is executed. E.g., ***type checking*** in C++ is static.

Stream An object used to transfer data from one place to another. The word "stream" suggests data "flowing" from a source to a sink.

Stream library The library that contains all of the stream classes and other information that are commonly used by C++ programs. The library is actually split into several parts: `iostream`, `iomanip`, etc.

String A sequence of characters. A string ***literal*** is written using quotes. E.g., `"Hello, world!"`.

String stream A stream that reads from or writes to a memory buffer. An input string streams (`istringstream`) reads from a buffer and an output string stream (`ostringstream`) writes to a buffer.

Template A unit of code that is parameterized. The parameter is replaced by an argument at compile-time, yielding code that can be compiled.

Template metaprogramming A way of performing compile-time computations using the ***template*** system.

Translation unit A collection of files seen by the compiler as a single unit. A translation unit typically consists of an ***implementation file*** and several ***header file***s.

Type A category that assigns an interpretation to a bunch of bits. E.g., we can understand the meaning of a 32-bit word if we know that it is an `int` or a `char[4]`.

Type checking The process of ensuring that the type declarations in the program are consistent and cannot cause meaningless computation. Type checking is performed mostly by the compiler but very occasionally by the run-time system.

Unary operator An ***operator*** with one parameter.

Upcast A cast from a derived class type "up" the class hierarchy to a base class type. Upcasting is always safe, in the sense that nothing is left undefined, but it usually results in a loss of information.

Value comparison A comparison in which the values of objects, rather than their names or addresses, are compared. Value comparison is weaker than ***identity comparison*** because two objects may be value-equal (technically: extensionally equal) but be distinct objects.

Virtual function A function declared in a base class that can be redefined in derived classes to give run-time polymorphic behaviour (***dynamic binding***).

Visitor A design pattern that encapsulates the traversal of a data structure.

YAGNI stands for "you aren't going to need it". It is an exhortation: write only the code that you actually need now, not the code that you might need tomorrow, next week, or . . . . never.

References

Abelson, H., J. Sussman, and J. Sussman (1984). *Structure and Interpretation of Computer Programs*. MIT Press. https://mitpress.mit.edu/sicp/.

Abrahams, D. and A. Gurtovy (2005). *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley.

Alexander, C., S. Ishikawa, and M. Silverstein (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Generic Patterns Applied*. Addison-Wesley.

Barski, C. (2010). *Land of Lisp: Learn to Program in Lisp, One Game at a Time!* No Starch Press. http://landoflisp.com/.

Beck, K. and W. Cunningham (1989, October). A laboratory for teaching object-oriented thinking. In *Object-Oriented Programming: Systems, Languages and Applications*, pp. 1–6.

Bentley, J. (1986). *Programming Pearls*. Addison-Wesley.

Bentley, J. (1988). *More Programming Pearls*. Addison-Wesley.

Coplien, J. O. (1995). Curiously recurring template patterns. *C++ Report 7*(2), 24–27.

Dean, J. and S. Ghemawat (2004, December). MapReduce: Simplified Data Processing on Large Clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, San Francisco, CA, USA. http://labs.google.com/papers/mapreduce.html.

Dewhurst, S. C. (2005). *C++ Common Knowledge: Essential Intermediate Programming*. Addison-Wesley.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison-Wesley.

Freeman, E. and E. Freeman (2004). *Head First Design Patterns*. O'Reilly.

Friedman, D. P. and M. Felleisen (1995). *The Little Schemer* (4th ed.). MIT Press. https://mitpress.mit.edu/books/little-schemer.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Google (2017). Google C++ Style Guide. http://google.github.io/styleguide/cppguide.html. Last accessed 24.10.2017.

Graff, M. G. and K. R. van Wyk (2003). *Secure Coding: Principles and Practices*. O'Reilly.

Hoare, C. (1981, February). The emperor's old clothes. *Communications of the ACM 24*(2), 75–83. Reprinted in (Hoare and Jones 1989).

Hoare, C. and C. Jones (1989). *Essays in Computing Science*. Prentice-Hall.

Hoglund, G. and G. McGraw (2004). *Exploiting Software: How to Break Code*. Addison-Wesley.

Hunt, A. and D. Thomas (2000). *The Pragmatic Programmer*. Addison-Wesley.

Josuttis, N. M. (2012). *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley. http://www.cppstdlib.com/.

Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley.

Kernighan, B. W. and D. M. Ritchie (1978). *The C Programming Language*. Prentice-Hall.

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example*. Addison-Wesley.

Lakos, J. (1996). *Large-Scale C++ Software Design*. Professional Computing Series. Addison-Wesley.

Langer, A. and K. Kreft (2000). *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley.

Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.). Professional Technical Reference. Prentice Hall.

Meyers, S. (1998). *Effective C++: 50 Specific Ways to Improve Your Programs and Designs* (2nd ed.). Addison-Wesley.

Meyers, S. (2001). *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley.

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley.

Meyers, S. (2014). *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14* (1st ed.). O'Reilly.

Misfeldt, T., G. Bumgardner, and A. Gray (2004). *The Elements of C++ Style*. Cambridge University Press.

Musser, D. R., G. J. Derge, and A. Saini (2001). *STL Tutorial and Reference Guide*. Professional Computing Series. Addison-Wesley.

Parnas, D. L. (1978). Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd International Conference on Software engineering*, Piscataway, NJ, USA, pp. 264–277. IEEE Press. Reprinted as (Parnas 1979).

Parnas, D. L. (1979, March). Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 128–138.

Prata, S. (2012). *C++ Primer Plus* (6th ed.). Addison-Wesley.

Raymond, E. S. (2004). *The Art of UNIX Programming*. Addison-Wesley. http://www.catb.org/esr/writings/taoup/.

Seacord, R. C. (2013). *Secure Coding in C and C++* (2nd ed.). Addison-Wesley.

Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley.

Stroustrup, B. (1997). *The C++ Programming Language*. Addison-Wesley.

Stroustrup, B. (2013). *The C++ Programming Language* (4th ed.). Addison-Wesley.

Sutter, H. (2000). *Exceptional C++: 47 Engineering puzzles, Programming Problems, and Solutions*. C++ In-Depth Series. Addison-Wesley.

Sutter, H. (2002). *More Exceptional C++: 40 New Engineering puzzles, Programming Problems, and Solutions*. C++ In-Depth Series. Addison-Wesley.

Sutter, H. (2005). *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions*. C++ In-Depth Series. Addison-Wesley.

Sutter, H. and A. Alexandrescu (2005). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. C++ In-Depth Series. Addison-Wesley.

Vandevoorde, D., N. M. Josuttis, and D. Gregor (2017). *C++ Templates* (2nd ed.). Addison-Wesley. http://tmplbook.com.

Veldhuizen, T. (1995, May). Using C++ template metaprograms. *C++ Report 7*(4), 36–43.

Weiss, M. A. (2004). *C++ for Java Programmers*. Pearson Prentice Hall.

Williams, A. (2012). *C++ Concurrency in Action*. Manning. http://www.manning.com/williams/.