Exploring C++

Alice E. Fischer

University of New Haven, AFischer@newhaven.edu

Follow this and additional works at: https://digitalcommons.newhaven.edu/electricalcomputerengineering-books

Part of the Computer Sciences Commons

Publisher Citation

Comments
Copyright ©2009, 2021, 2023 by Alice E. Fischer. All rights reserved. No part of this manuscript may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author. Posted with permission from the author.
Exploring C++

Alice E. Fischer

University of New Haven

January 2009
(Revised 2014, 2022, 2023)

Copyright ©2009, 2021, 2023

by Alice E. Fischer

All rights reserved. No part of this manuscript may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the authors.
## Contents

1 Preamble

1.1 Commandments .......................................................... 1

1.2 Style ........................................................................ 1

1.2.1 Guidelines and Rules for this Course. ......................... 2

1.2.2 Naming .................................................................. 3

1.2.3 Usage .................................................................... 4

1.2.4 Indentation ................................................................ 4

1.2.5 Function Definitions ................................................. 5

1.2.6 Types, type definitions and struct ......................... 6

1.3 Procedures for CSCI 4526 / 6626 Projects ...................... 6

1.3.1 Projects ............................................................ 6

1.3.2 Using the Tools Library ...................................... 7

2 Programming for Reliability .............................................. 9

2.1 Why did C need a ++? ..................................................... 9

2.1.1 Design Goals ....................................................... 9

2.1.2 C++ Supports Modeling .......................................... 9

2.1.3 Modeling ............................................................. 10

2.2 Object-Oriented Principles ............................................ 11

2.3 Differences between C and C++ .................................. 12

2.4 What is OO? ............................................................. 15

2.5 Header Files and Code Files ........................................ 17

2.6 Class Basics ................................................................ 17

2.6.1 Data members ....................................................... 18

2.6.2 Typical Class Methods ......................................... 19

2.7 A First Example: Sorting a list of strings. ....................... 20

2.7.1 Main Program ....................................................... 21

2.7.2 The MemData Class Declaration ............................ 21

2.7.3 The MemData Class Definition ............................... 22

2.8 Review of Topics from Intermediate C/C++ .................... 24

2.8.1 Command-line arguments ..................................... 24

2.8.2 Enum Classes ....................................................... 24

2.8.3 The UML diagram ................................................ 25

2.8.4 Main Program ....................................................... 25

2.8.5 The Stack Class Declaration ................................ 25

2.8.6 The Brackets Class Declaration ............................. 26

2.8.7 The Brackets Class Definition ................................. 26

2.8.8 The Token Class Declaration ................................. 27

2.8.9 The Token Class Definition ................................... 27
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.3</td>
<td>Example Program: Making a Bar Graph</td>
<td>122</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Techniques Illustrated</td>
<td>122</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Specification</td>
<td>122</td>
</tr>
<tr>
<td>9.3.3</td>
<td>The Bargraph Application</td>
<td>123</td>
</tr>
<tr>
<td>10</td>
<td>Array Data Structures</td>
<td>129</td>
</tr>
<tr>
<td>10.1</td>
<td>Allocation and Deallocation of Arrays</td>
<td>129</td>
</tr>
<tr>
<td>10.2</td>
<td>2D Arrays of Chars</td>
<td>132</td>
</tr>
<tr>
<td>10.3</td>
<td>The Flexible Array Data Structure</td>
<td>133</td>
</tr>
<tr>
<td>10.3.1</td>
<td>Implementation in C++</td>
<td>133</td>
</tr>
<tr>
<td>10.3.2</td>
<td>Tracing the operation of the FlexItems</td>
<td>137</td>
</tr>
<tr>
<td>10.3.3</td>
<td>Implementation in C</td>
<td>138</td>
</tr>
<tr>
<td>10.4</td>
<td>The Flex Template</td>
<td>138</td>
</tr>
<tr>
<td>10.4.1</td>
<td>The Flex Template</td>
<td>139</td>
</tr>
<tr>
<td>10.4.2</td>
<td>The Flex Template</td>
<td>139</td>
</tr>
<tr>
<td>10.5</td>
<td>Hashing</td>
<td>140</td>
</tr>
<tr>
<td>10.5.1</td>
<td>The Hash Table Array</td>
<td>141</td>
</tr>
<tr>
<td>10.5.2</td>
<td>Hash Functions</td>
<td>142</td>
</tr>
<tr>
<td>10.6</td>
<td>Example: A Dictionary</td>
<td>142</td>
</tr>
<tr>
<td>10.6.1</td>
<td>The Dictionary Class</td>
<td>143</td>
</tr>
<tr>
<td>10.6.2</td>
<td>A test function</td>
<td>146</td>
</tr>
<tr>
<td>11</td>
<td>Derived Classes</td>
<td>149</td>
</tr>
<tr>
<td>11.1</td>
<td>Derivation Basics</td>
<td>149</td>
</tr>
<tr>
<td>11.2</td>
<td>Code Example: Publications</td>
<td>150</td>
</tr>
<tr>
<td>11.2.1</td>
<td>Allocation, Constructors and Initializer Lists</td>
<td>151</td>
</tr>
<tr>
<td>11.2.2</td>
<td>Visibility and Protection Level</td>
<td>152</td>
</tr>
<tr>
<td>11.2.3</td>
<td>Resolving Ambiguous Names</td>
<td>153</td>
</tr>
<tr>
<td>11.2.4</td>
<td>Inherited Functions</td>
<td>154</td>
</tr>
<tr>
<td>11.3</td>
<td>Class Derivation Demo</td>
<td>154</td>
</tr>
<tr>
<td>11.3.1</td>
<td>Inherited Data Members</td>
<td>155</td>
</tr>
<tr>
<td>11.4</td>
<td>Derivation and Inheritance</td>
<td>156</td>
</tr>
<tr>
<td>11.5</td>
<td>Playing Hangman</td>
<td>157</td>
</tr>
<tr>
<td>11.5.1</td>
<td>Hangman: The Main Program</td>
<td>157</td>
</tr>
<tr>
<td>11.5.2</td>
<td>The Game Class</td>
<td>158</td>
</tr>
<tr>
<td>11.5.3</td>
<td>The Board Class</td>
<td>160</td>
</tr>
<tr>
<td>11.6</td>
<td>The Word Classes</td>
<td>163</td>
</tr>
<tr>
<td>11.6.1</td>
<td>The BaseWord Declaration</td>
<td>164</td>
</tr>
<tr>
<td>11.7</td>
<td>The Derived Word Classes</td>
<td>165</td>
</tr>
<tr>
<td>11.8</td>
<td>UML Diagrams: A View of the Class Relationships</td>
<td>167</td>
</tr>
<tr>
<td>12</td>
<td>Polymorphism and Virtual Functions</td>
<td>169</td>
</tr>
<tr>
<td>12.1</td>
<td>Basic Concepts</td>
<td>169</td>
</tr>
<tr>
<td>12.1.1</td>
<td>Definitions</td>
<td>169</td>
</tr>
<tr>
<td>12.1.2</td>
<td>Virtual functions</td>
<td>170</td>
</tr>
<tr>
<td>12.2</td>
<td>Creating a Polymorphic Container</td>
<td>171</td>
</tr>
<tr>
<td>12.2.1</td>
<td>Container: An Abstract Class</td>
<td>171</td>
</tr>
<tr>
<td>12.2.2</td>
<td>Linear: A Polymorphic Class</td>
<td>172</td>
</tr>
<tr>
<td>12.2.3</td>
<td>Cell: The Helper Class</td>
<td>174</td>
</tr>
<tr>
<td>12.2.4</td>
<td>Exam: The Actual Data Class</td>
<td>175</td>
</tr>
<tr>
<td>12.2.5</td>
<td>Class diagram</td>
<td>176</td>
</tr>
<tr>
<td>12.3</td>
<td>Stack: Fully Specific</td>
<td>176</td>
</tr>
<tr>
<td>12.4</td>
<td>Queue: Fully Specific</td>
<td>178</td>
</tr>
<tr>
<td>12.5</td>
<td>A Main Program and its Output</td>
<td>179</td>
</tr>
</tbody>
</table>
Chapter 1: Preamble

This book is intended for use by students who hope to deepen their understanding of C++ and learn about advanced features. It is also useful for C or Java programmers who want to learn C++ and OO style... fast. It assumes that the reader knows basic programming including types, type-matching rules, control structures, functions, arrays, pointers, and simple data structures. The material should help you develop a deeper understanding of the implementation of C++, of clean program design and of the features that make C++ a powerful and flexible language.

1.1 Commandments

A nationally-known expert has said that C++ is a coding monster that forces us to use a disciplined style in order to tame it. This leads to a fundamental rule for skilled C++ programming:

**Can is not the same as should.**

The style guidelines below are motivated by years of experience writing and debugging C and C++. None of them are arbitrary. I expect you to read them, understand them, and follow them.

**Commandment 1.** Use C++, not C for all work in this course. The biggest differences are in the use of classes to organize and simplify the code, and the way we do I/O.

**Commandment 2.** The use of global variables for any purpose will not be tolerated. If your code contains a global variable declaration, your work will be handed back ungraded. Note, however, that global constants are OK.

**Commandment 3.** Test every line of code you write. Turn in screen shots or capture the output from the terminal window. **It is your job to prove to me that your entire program works.** If I get a program without a test plan and output, I will assume that it does not compile. A program with a partial or inadequate test plan will be assumed to be buggy.

**Commandment 4.** Choose a reasonable style and use it consistently. This covers naming, indentation, comments, layout, and every other aspect of programming.

1.2 Style

If you want to be a professional, learn to make your work look professional.

When you work for a company, you will be expected to follow all the style guidelines and coding rules established at that company. They may or may not be the best guidelines and styles. However, following them is not optional. Note: it does not help to tell the boss that he/she is prescribing obsolete or strange customs. As an example, look at NASA’s guidelines below.

**Example of Local Style Guidelines** Following are NASA’s rules for developing safety-critical code. Many of them are beyond the scope of this course, but they give an example of local style rules.

1. Restrict all code to very simple control flow constructs; do not use goto statements, setjmp or longjmp constructs, or direct or indirect recursion.

2. Give all loops a fixed upper bound.
3. Do not use dynamic memory allocation after initialization.
4. No function should be longer than what can be printed on a single sheet of paper in a standard format with one line per statement and one line per declaration.
5. The code’s assertion density should average to minimally two assertions per function.
6. Declare all data objects at the smallest possible level of scope.
7. Each calling function must check the return value of non-void functions, and each called function must check the validity of all parameters provided by the caller.
8. The use of the preprocessor must be limited to the inclusion of header files and simple macro definitions.
9. Limit pointer use to a single dereference, and do not use function pointers.
10. Compile with all possible warnings active; all warnings should then be addressed before release of the software.

1.2.1 Guidelines and Rules for this Course.

These programming techniques deal with details that are specific to C/C++ programming and are important for success in this course. There is a well-considered reason behind each one, and I expect you to follow them, even if they do not seem “right” to you. In the early weeks of this course, I will help you learn to apply the principles. Your willingness, diligence, and ability to write clean code in my prescribed style will determine my opinion of you and your skills.

1. Use only the standard C++ language and its standard template library. Do not use proprietary extensions and language additions such as clrs() and conio.h. Also, do not use system commands such as pause. Instead, learn how to do these things within the standard language. Your code must compile and run properly on whatever systems the TA and the Professor have.

2. For a compiler, use g++ (on Linux) or clang++ (on the Mac). If you turn in work done in visual studio, it may or may not be accepted, because it must compile and run on my system.

3. Learn how to include and use a local library, how to create, organize, and compile a multi-module program, and how to compile, link, and run a program from a command shell.

4. Wherever possible, use symbolic names in your code. For example, use quoted characters instead of numeric ASCII codes.

5. Whitespace. In general, put spaces around your operators and either before or after each parenthesis. Do not write a whole line of code without spaces! Readability matters!


7. Blank lines. Use blank lines to group your code into “paragraphs” of related statements. Put a blank line after each paragraph.
   • DO NOT put a blank line after every line of code.
   • Do not use more than one blank line.
   • Do not put a blank line before an opening or closing brace.
   • Do not separate the first or the last line of a function from its body.

8. Simplicity is good; complexity is bad. If there are two ways to do something, the more straightforward or simpler way is preferred.

9. Locality is good; permitting one part of the program to depend on a distant part is bad. For example, initialize variables immediately before the loop that uses them, not somewhere else in the program.
10. Avoid writing **useless code**. For example, don’t initialize variables that don’t need initialization. Do not leave blocks of code in your program that you have commented out. Remove them before you send me the code.

11. Avoid writing the **same code** twice. For example, don’t call strlen(s) twice on the same string. Don’t write the same thing in both clauses of an if statement. Don’t write two functions that output the same thing to two different streams. This way, when you correct an error once, it is corrected. You don’t need to ask how many times the same error occurs.

12. **Brevity.** Keep all functions short. At worst, a function should fit on one screen. Ideally, the function should be shorter than a dozen lines. Break up long or complex chunks of logic by defining more functions.

13. Limit classes to 20 or fewer members, including both data and function members.

14. Learn to use `const` and use it widely for variables and parameters.

15. File system **path names** are not portable. If you must write one in your program, put it in a `#define` or a `const` declaration at the top of main so that it can be found easily and changed easily.

16. **Initialize** By the end of the constructor, every class data-members should be initialized and the object should be ready to use. Initialize data members in the class declaration, if possible. Otherwise, initialize them in a constructor-initializer list. If neither is possible, use assignment statements in the body of the constructor.

### 1.2.2 Naming

1. Please **do not use** `i`, `I`, `l`, or `O` as a variable name because `i` and `l` look like `1` and `O` looks like `0`. Use `j`, `k`, `m`, or `n` instead.

2. **Long, jointed names and short, meaningless names** make code equally hard to read. Try for moderate-length names. Use camelCase for compound names, not underscores.

3. To be consistent with Java usage, the name of a class should start with an Upper case letter (`MyClassName`) and use camel-case after that. CamelCase has an upper case letter for the beginning of every word segment, with lower case letters between them. Variable and function names should always start with a lower case letter: `myVar`, `myFunction()`.

4. **Local variables and class members** should have short names because they are always used in a restricted context. Global objects and functions should have longer names because they can be used from distant parts of the program.

5. I should be **able to pronounce** every name in English, and no two names should have the same pronunciation or the same spelling with different cases.

6. Names need to be **different enough** to be easily distinguished. The first part of a name is most important, the last letter next, and the middle is the least important. Any two names in your program should differ at the beginning or the end, not just in the middle.

7. Do not use the same name for two purposes (a class and a variable, for example). Do not use two names that differ only by an ‘s’ on the end of one. Do not use two names that differ only by the case (for example `Object` and `object`).

8. Learn to use the names of the various **zero-constants** appropriately. `nullptr` is a pointer to nowhere, `\0` is the null character, `""` is the null string, `false` is a `bool` value, and `0` and `0.0` are numbers. Use the constant that is correct for the current context.

9. Use names written entirely in UPPER CASE for `#defined constants` but not for other things. This is a long-honored `C` custom.
1.2.3 Usage

1. Use the C++ language as it is intended. Learn to use ++ (instead of +1) and if...break instead of complex loop structures with boolean flags.

2. Please do not misuse the conditional operator to convert a true or false value to a 1 or a 0:
   
   write this  \((a < b)\)
   instead of  \((a < b) \ ? 1 : 0\)

3. Use the \(-\rightarrow\) operator when you are using a pointer to an object or a struct. Example:
   
   write this  \(p->next->data\)
   instead of  \(*((p).next).data\)

   Using * and . for this purpose leads to code that fails the simplicity test: it is hard to write, hard to debug, and hard to read.

4. Use pointers for sequential array access and subscripts for random access.

5. C++ has subscripts...use them. They are easy to write, easy to debug, and easy to read. Use pointer arithmetic rarely, if at all. Example: suppose that \(ar\) is an array of MAX ints and \(p\) is a pointer to an array of ints.
   
   write this  \(ar[n]\) or  \(p[n]\)
   instead of  \(*ar+n\) or  \(*p+n\)

   sometimes you should write this, however  \(\text{int}\ast ar\_end = ar + MAX;\)

   The last line in this example is the ordinary way to set a pointer to the end of the array

1.2.4 Indentation

Randomly indented code is unprofessional and inappropriate for a senior or graduate student.

1. The indentation style preferred by experts is this:

   ```
   while ( k < num\_items) {
     cin >> ageIn;
     if (inRange( ageIn ) {
       age[k] = ageIn;
       k++;
     }
     else {
       cout << "An invalid age was entered, try again.");
     }
   }
   ```

   This style is space-efficient and helps avoid certain kinds of common errors. Note that the lines within the loop are indented a modest amount: more than 1 or 2 spaces and less than 8 spaces. The opening bracket is on the end of the line that starts with the keyword and the closing bracket is directly below the first letter of the keyword. Some IDE’s use this style by default, others can be controlled by setting preferences.

2. Please use the indentation style shown above. However, the other two generally approved indentation styles are shown below.
1.2. STYLE

Brackets aligned on the left
– actually less readable.

Brackets aligned with the indented code:
– rarely used.

while ( k < max )
{
    cin >> ageIn;
    if ( inRange( ageIn )
    {
        age[k] = ageIn;
        k++;
    }
    else
    {
        cout << "Invalid age."
    }
}

3. Never let comments interrupt the flow of the code-indentation.

4. Break up a long line of code into parts; do not let it wrap around from the right side of the page to the left edge of the page on your listings. Confine your lines of code to 80 columns.

5. Put the indentation into the file when you type the code originally. If you indent one line, most text editors will indent the next line similarly.

1.2.5 Function Definitions

1. If a function is simple and fits entirely on one line, write it that way, usually in the class header file. Example:

   bool square_sum( double x, double y ) { return x*x + y*y; }

2. Each function definition in the .cpp file definition should start with a whole-line comment that forms a visual divider. If the function is nontrivial, a comment describing its purpose is often helpful. If there are preconditions or postconditions for the function, state them here.

   // ---------------------------------------------------------------------
   // If needed, put description of function and preconditions here.
   // Document postconditions after the preconditions.
   // Document return values that represent failure or success.
   void print( Stack* St ) // Print contents of stack, formatted.
   {
       char* p = St->s; // Scanner and end pointer for data.
       char* pend = p + St->top;

       printf( "The stack %s contains: -[", St->name );
       for ( ; p < pend; ++p) printf( " %c", *p );
       printf( " ]>");
   }

3. Write the return type on a line by itself immediately below the comment block then write the name of the function at the beginning of the next line, as in the above sample. Why? As we progress through this language, return types become more and more complex. It is greatly helpful to be able to find the function name easily and quickly.

4. In general, try to stick to one return statement per function. The only exception is when a second return statement substantially simplifies the code.

5. In a well-designed program, all function definitions are short. Brief comments at the top of the function are often enough to explain the operation of the code. Be sure that you can see the entire function definition on one screen in your IDE. If the function code has identifiable sections or phases,
there is a good chance that each phase should be broken out into a separate function. When a long function cannot be broken up, each phase should be introduced by a whole-line comment.

1.2.6 Types, type definitions and struct

1. As this course progresses, a clear concept of the type of things is going to become essential. Regardless of examples you see in other contexts, when writing the name of a pointer or reference type, write the asterisk or ampersand as part of the type name, not as part of the object name. Example:

```c
cell* make_cslist( cell& item ); // write it this way
cell *make_cslist( cell &item ); // not this way.
```

2. Use an `enum` declaration to give symbolic names to error codes and other sets of related constants. The name of the constant then becomes adequate documentation for the line that uses it. Suppose you want to define codes for various different kinds of errors. Write it this way in C++:

```c
enum errorType { sizeOK, TooSmall, TooLarge };
```

Declare and initialize an `errorType` variable like this:

```c
errorType ercode = sizeOK;
```

3. When using structured types, use a class declaration. Please DO NOT use the `struct` declaration or the `struct` keyword this term, even when the class has no explicit function members.

1.3 Procedures for CSCI 4526 / 6626 Projects

1.3.1 Projects

All programs for this course must be done as projects.

1. Suppose your course number is CSCI 1234. Then establish a directory named cs1234 on your disk for all the work in this course.

2. Within the cs1234 directory, create a separate subdirectory for each programming assignment. Name your project directories with the assignment number and your last name: P1-Smith, P2-Smith, P3-Smith, etc. store all the files for the project in the relevant directory: source files, input files, output files, and screen shots.

3. Each programming assignment is one phase of the eventual term project. When you finish one phase, make sure the output for that phase is in the directory, and copy all the code files to the directory for the next phase. This is a very conservative way to ensure that you do not lose important files.

4. I do not use an IDE for making projects, although you may if you wish. Whether working on my own code or yours, I do all compilation from the command line.

5. To submit your work, make a temporary subdirectory called “P1-SmithSubmit” Copy the source code, headers, input, and output files, or whatever your instructor requires, into P1-SmithSubmit, zip them, and submit according to the procedures established for your course. DO NOT SEND all the files generated by your compiler. They are useless to the Professor and the TA, and they can make a project too large to go through the UNH email system.

6. Warning: the UNH mail system will reject files that are too long or look like executable files to it.
1.3.2 Using the Tools Library

All programs for this course must use the functions `banner()`, `bye()` from the tools library. Many will use `fatal()`, also, for situations where exceptions are not appropriate.

1. From the Canvas site, please download a small library called “tools”. It has two files: a source code file `tools.cpp` and a header file `tools.hpp`. The ideal place to put your two tools files is at the top level of your cs4526 directory.

2. To use the tools, put `#include "tools.hpp"` in your main program and in any `.hpp` files you write yourself. Do not include “`tools.cpp`” anywhere but be sure that you compile it separately.

3. Various useful functions and macros are included in the tools library; please look at tools.hpp and learn about them. You will need to use `banner()`, `bye()`, and `fatal()`. You may also need `flush()`, `DUMPp`, `DUMPv`, `menu()`, `today()`, `o'clock()`.

4. Look at the first several lines of `tools.hpp`; note that it includes many useful C and C++ library headers. When you include the tools, you do not need to include these standard header files. Please don’t clutter your code with duplicate `#include`s.

5. Personalization. Before you can use the tools, you must put your own name in the `#define NAME` line on line 10 of `tools.hpp`, in place the phrase that appears there. Put your course number (4536 or 6626) on line 11.

6. Start each program with a call on `banner()` (or `fbanner()` for output to a file). This will label the top of your output with your name, and the time of execution. End each program with a call on `bye()`, which prints a “Normal termination” message.

7. If you need to abort execution, call `fatal(format, ...)`. The parameters are the same as for printf. This function formats and prints and error comment, then flushes the stream buffers and aborts execution properly.
Chapter 2: Programming for Reliability

2.1 Why did C need a ++?

For application modeling. Shakespeare once explained it for houses, but it should be true of programs as well:

  When we mean to build, we first survey the plot then draw the model.
  ...Shakespeare, King Henry IV.

2.1.1 Design Goals

C was designed to write Unix. C is sometimes called a “low level” language. It was created by Dennis Ritchie so that he and Kenneth Thompson could write a new operating system that they named Unix. The new language was designed to control the machine hardware (clock, registers, memory, devices) and implement input and output conversions. Thus, it was and still is essential for C to be able to work efficiently and easily at a low level.

Ritchie and Thompson worked with small, slow machines, so they put great emphasis on creating a simple language that could be easily compiled into efficient object code. There is a direct and transparent relationship between C source code and the machine code produced by a C compiler. Today, the language is in widespread use for low-level work but has never been a good vehicle for large-scale application design.

Python was designed to make scripting easy, useful, and accessible. It serves that purpose admirably, at the cost of eliminating efficiency and the possibility to check for code validity prior to runtime. The language is hugely popular, but not suitable for applications in which runtime failure is unacceptable.

2.1.2 C++ Supports Modeling

The designer of C++. Bjarne. Stroustrup wanted to retain the efficiency and transparency of C, and simultaneously improve the ability of the language to model abstractions. The full C language remains as a subset of C++ in the sense that anything that was legal in C is still legal in C++ (although some things have a slightly different meaning). In addition, many things that were considered “errors” in C are now legal and meaningful in C++.

Readability. Basic C++ was no more readable than C, because C was retained as the basic vehicle for coding in C++ and is a proper subset of C++. However, an application program, as a whole, may be much more readable in C++ than in C because of the new support for application modeling. Further, using a disciplined style and following guidelines for clear communication and presentation make a huge difference.

Functions and Methods. In C++, a function name no longer has a single unified definition. Instead, a function is a collection of one or more methods, possibly defined in multiple classes, that operate on different combinations of parameter types. This is a big improvement because the same method name defined in one class can be defined in others. The burden on a programming team to choose non-conflicting names for every function is gone, function names can be shorter, and programs are more readable.

Ideally, all of the methods of a function would have the same (or a very similar) general purpose. For example, a method named print() would be expected to print its argument to a file or display it on the screen, and to do so without modifying the argument.
Flexibility. Work on C++ began in the early days of object-oriented programming. The importance of types and type checking was understood, but this led to a style of programming in which the type of everything had to be known at compile time. (This is sometimes called “early binding”). Languages such as Lisp did not do compile-time type checking, they deferred it until run-time (late binding).

C++ operates on a middle ground. While simple code can still be compiled efficiently, an application that relies on late binding can be fully modeled because types are not isolated from each other. Polymorphic class hierarchies can be defined with a base class and several derived subclasses. A derived class inherits data and function members from its base class. Data members no longer have unique types – they simultaneously have all the types above them in the inheritance hierarchy.

Compile-time and Run-time Dispatching. An OO language checks at compile time that the arguments to each function call have the appropriate base types for at least one of the methods of the function. If it is exactly one, that method is used to compile the call. However, for polymorphic types, more than one method may be appropriate, and a final choice of which function method to compile for the call must be deferred until run-time, when the subtype of the argument object can be checked and matched against the parameter lists of the available function methods.

Thus, an OO language uses late binding for polymorphic types and early binding for simple types. This allows a combination of flexibility (where needed) and efficiency (everywhere else) that is not achievable in a language with only late binding.

Portability. A portable program can be “brought up” on different kinds of computers and produce uniform results across platforms. By definition, if a language is fully portable, it does not exploit the special features of any hardware platform or operating system. It cannot rely on any particular bit-level representation of any object, operation, or device; therefore, it cannot manipulate such things. A compromise between portability and flexibility is important for real systems.

A program in C++ can be very portable if the programmer designs it with portability in mind and follows guidelines about segregating sections of code that are not portable. Skillful use of the preprocessor and conditional compilation can compensate for differences in hardware and in the system environment. However, programs written by naive programmers are usually not portable because C’s most basic type, \texttt{int}, is partially undefined. Programs written for the 4-byte integer model often malfunction when compiled under 2-byte compilers and vice versa. C++ does nothing to improve this situation.

Reusability. Code that is filled with details about a particular application is not very reusable. In C, the \texttt{typedef} and \texttt{#define} commands do provide a little bit of support for creating generic code that can be tailored to a particular situation as a last step before compilation. The C libraries even include two generic functions (\texttt{qsort()} and \texttt{bsearch()}) that can be used to process an array of any base type. C++ provides much broader support for this technique and provides new type definition and type conversion facilities that make generic code easier to write.

Teamwork potential. C++ supports highly modular design and implementation and reusable components. This is ideal for team projects. The most skilled members of the group can design the project and implement any non-routine portions. Lesser-skilled programmers can implement the routine modules using the expert’s classes, classes from the standard template library, and proprietary class libraries. All these people can work simultaneously, guided by defined class interfaces, to produce a complete application.

2.1.3 Modeling.

The problems of modeling are the same in C and C++. In both cases the questions are, what data objects do you need to define, how should each object work, and how do they relate to each other? A good C programmer would put the code for each type of object or activity in a different file and could use type modifiers \texttt{extern} and \texttt{static} to control visibility. A poor C programmer, however, would throw it all into one file, producing an unreadable and incomprehensible mess. Skill and style make a huge difference in C. In contrast, C++ provides classes for modeling objects and several ways to declare or define the relationship of one class to others.
2.2. OBJECT-ORIENTED PRINCIPLES.

What is a model? A model of an object is a list of the relevant facts about that object in some language. A low level model is a description of a particular implementation of the object, that specifies the number of parts in the object, the type of each part, and the position of each part in relation to other parts. C supports only low level models.

C++ also supports high-level or abstract models, which specify the functional properties of an object without specifying a particular representation. This high-level model must be backed up by specific low-level definitions for each abstraction before a program can be translated. However, depending on the translator used and the low-level definitions supplied, the actual number and arrangement of bytes of storage that will be used to represent the object may vary from translator to translator.

A high level model of a process or function specifies the pre- and post-conditions without specifying exactly how to get from one to the other. A low level model of a function is a sequence of program definitions, declarations, and statements that can be performed on objects from specific data types. This sequence must start with objects that meet the specified pre-conditions and end by achieving the post-conditions.

High level process models are not supported by the C language but do form an important element of project documentation. In contrast, C++ provides class hierarchies and virtual functions which allow the programmer to build high-level models of functionality, and later implement them.

2.2 Object-Oriented Principles.

The way a language is used is more important in OO programming than the language being used. C++ was designed to support OO programming; it is a convenient and powerful vehicle for implementing an OO design. However, with somewhat more effort, that same OO design could also be implemented in C. Similarly, a non-OO program can be written in C++.

Principles central to object-oriented programming are encapsulation, grouping related data and methods together, and generic or polymorphic functions.

Classes. The term “object-oriented” has become popular, and “object-oriented” analysis, design, and implementation has been put forward as a solution to several problems that plague the software industry. OO analysis is a set of formal methods for analyzing and structuring an application from the application data’s perspective, as opposed to the traditional functional or procedural point of view. The result of an OO analysis is an OO design. OO programs are built out of a collection of modules, often called classes that contain both function methods and data. Classes define data structures and the operations that can be performed on them. Access to all of the data and some of the method elements should only be through the defined methods of the class.

Some languages (Ruby, Python) say they are OO because they have classes. However these languages support only the surface syntax of classes, not the semantic properties: privacy, consistency and compile-time type checking. The strength of C++, the classes are both clearly defined and stable. They be relied on to implement a stable semantics for every instance of a class. In contrast, in languages like Ruby and Python, a class definition can vary over time.

Encapsulation. The most fundamental OO design principle is that a class should take care of itself. Typically, a class has data members: constants, variables, and internal type declarations. The class also includes constructors (for automatic initialization), destructors (for automatic memory management), and function members that operate on the data members. The OO principle of encapsulation says that only a function inside a class should ever change the value of a data member. This is achieved by declaring member variables to be private. A member function can then freely use any data member, but outside functions cannot.

Functions that are intended for internal use only are also made private. Functions that are part of the function’s published interface are made public. Finally, constant data members are sometimes made public.

In C++, class relationships can be declared, and the simple guidelines for public and private visibility are complicated by the introduction of protected visibility and of class friendship.

---

1 C++ is not the first or only OO language. Earlier OO languages such as SIMULA-67 (from Dahl and Nygaard) and Smalltalk (from Alan Kay) embodied many OO concepts prior to Stroustrup’s C++ definition in 1990.
2 This was how I wrote programs before C++ was invented.
Initialization. One way a class takes care of itself is to define how class objects should be initialized. Initialization is done by constructors, which are like functions except that they have no return type. The name of the constructor is the same as the class name. A constructor is called automatically whenever a class object is declared or dynamically allocated. It uses its parameters to initialize the class’s data members. A class often has multiple constructors, allowing it to be initialized using various combinations of arguments. A constructor might also dynamically allocate and initialize parts of the class object.

Cleanup. Cleanup, in C++, is done by destructors. Each class has exactly one destructor that is called when a class object is explicitly freed or when it goes out of scope at the end of the code block that declares it. The name of the destructor is a tilde (~) followed by the class name. The primary job of a destructor is to free dynamically allocated parts of the class object.

Generic code. A big leap forward in representational power was the introduction of generic code. A generic function is one like “+” whose meaning depends on the type of the operands. Floating-point “+” and integer “+” carry out the same conceptual operation on two different representations of numbers. If we wish to define the same operation (such as “print”) on five data types, C forces us to introduce five different function names. C++ lets us use one name to refer to several methods which, taken together, comprise a function. Sometimes, the same symbol is used for purposes that are only distantly related to the original. For example, the + symbol is often used to concatenate strings together. (The result of "abc" + "xyz" is "abcxyz".)

Generic code enabled the system designers and every programmer to build a very simple and convenient I/O system. Any kind of primitive object can be output using <<. User-defined objects can be output this way by extending the << operator. The generic input operator is >> and can also be extended to new types by a simple definition.

In C++ terminology, a single name is “overloaded” by giving it several meanings. The translator decides which definition to use for each function call based on the types of the arguments in that call. This makes it much easier to build libraries of functions that can be used and reused in a variety of situations.

OO Drawbacks. Unfortunately, a language with OO capabilities is also more complex. The OO extensions in C++ make it considerably more complicated than C. It is a massive and complex language. To become an “expert” requires a much higher level of understanding in C++ than in C, and C is difficult compared to Pascal or FORTRAN. The innate complexity of the language is reflected in its translators; C++ compilers (like Java compilers) are slower and can give very confusing error comments because program semantics can be more complex.

The ease with which one can write legal but meaningless code is a hallmark characteristic of C. The C programmer can write all sorts of senseless but legal things (such as a>b>c). C++ has a better-developed system of types and type checking, which improves the situation considerably. However C++ also provides powerful tools, such as the ability to add new definitions to old operators, that can easily be overused or misused. A good C++ programmer designs and writes code in a strictly disciplined style, following design guidelines that have evolved from experience over the years. Learning these guidelines and how to apply them is more important than learning the syntax of C++, if the goal is to produce high-quality, debugged, programs.

2.3 Differences between C and C++

- Classes. Structures, as in C are still available in C++, but a C++ struct is not the same as a C struct. Instead, it is almost exactly like a C++ class, with privacy options and methods inside the struct. The single difference between a C++ struct and a C++ class is that the default level of protection in a C++ struct is public, whereas the default in a C++ class is private. So which should you use in your programs? Simplify and be explicit. Use class consistently and add the public declaration if the members should be public.

- Function methods. Any function can have more than one definition, as long as the list of parameter types, (the signature) of every definition is different. The individual definitions are called methods of the function. In the common terminology, such a function is called overloaded. I prefer not to use this term so broadly. In this course, I will distinguish among extending, overriding, and overloading a function.
2.3. DIFFERENCES BETWEEN C AND C++

– We extend a function when you define a method for that function on a new class, and the method is semantically parallel to the original definition.

– We override a function in a base class when we redefine it in a derived class. This is frequently done to tailor the action of a function to each derived class.

– We overload a function when we use the same operator or function name to define an action that is not substantially the same as the original action.

In this course, please use these words appropriately.

• I/O. C and C++ I/O are completely different, but a program can use both kinds of I/O statements in almost any mixture. The C and C++ I/O systems both have advantages and disadvantages. For example, simple output is easier in C++ but output in columns is easier in C. Since one purpose of this course is to learn C++, please use only C++ output in the C++ programs you write for this course. Chapter 3 gives a comprehensive list and extensive examples of C++ input and output facilities.

• Templates. All of the common data structures and simple algorithms have been implemented as template classes in the standard library: stacks, queues, sorts, swap, max, min, etc. Using them (instead of writing one’s own code) can save a programmer time and hassle. This code has been tested by many people over decades, and is, therefore, safer than anything an individual can write.

• Using the C++ libraries. To use one of the standard libraries in C, we write an #include statement of the form #include <stdio.h> or #include <math.h>. A statement of the same form can be used in C++. For example, the standard input/output library can be used by writing #include <iostream.h>. However, this form of the include statement is obsolete. It should not be used in new programs and eventually may not work with modern compilers. Instead, you should write two lines that do the same thing:

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

The new kind of include statement still tells the preprocessor to include the headers for the iostream library, but it does not give the precise name of the file that contains those headers. It is left to the compiler to map the abstract name “iostream” onto whatever local file actually contains those headers. The second line brings the function names from the iostream library into your own context. Without this declaration, you would have to write std:: in front of every call of any library function or constant.

• Comments. Comments can begin with // and end with newline. Please use only this kind of comment to write C++ code. Then the old C-style kind of comments can be used to /* comment out */ larger sections of code during debugging.

• Executable declarations. Declarations can be mixed in with the code. Why is this useful?
   – C++ declarations can trigger file processing and dynamic memory allocation and it is VERY helpful during program construction to precede each major declaration by displaying a message that lets the programmer track the progress of the program.
   – When you put a declaration in a loop, the object will be allocated, initialized, and deallocated every time around the loop. This is unnecessary and inefficient for most variables but it can be useful when you need a strictly temporary object to use for input.

However, declarations must not be written inside one case of a switch statement unless you open a new block (using curly braces) surrounding the code for the case.

• A new kind of for loop. Starting with C++11, an additional kind of for loop is available. This loop syntax can be used to iterate through an array or any kind of data structure with multiple elements that is supported by the C++ Template Library (STL). The loop below adds 1 to each component of an array and prints the result. In this code fragment, ar is an array of 10 floats.
for (float f : ar) {
    f += 1.0;
    cout << f;
}

Each time around this loop, the name f gets bound to the next slot of the array. This makes it possible to both read and modify the array element, as in the above example.

- **Type identity.** In C, the type system is based on the way values are *represented*, not on what they *mean*. For example, pointers, integers, truth values, and characters are all represented by bitstrings of various lengths. Because they are represented identically, an int variable can be used where a char value is wanted, and vice versa. Truth values and integers are even more closely associated: there is no distinction at all. Because of this, one of the most powerful tools for ensuring correctness is compromised, and expressions that should cause type errors are accepted. (Example: k < m < n.)

In C++, as in all modern languages, type identity is based on *meaning*, not representation. Thus, truth values and integers form distinct types. To allow backwards compatibility, automatic type coercion rules have been added. However, new programs should be written in ways that do not depend on the old features of C or the presence of automatic type conversions. For example, in C, truth values were written as 0 and 1, In C++, they are false and true.

- **Type bool.** In standard C++, type bool, whose values are named false and true, is not the same type as type int. It is defined as a separate type along with type conversions between bool and int. The conversions will be used automatically by the compiler when a programmer uses type int in a context that calls for type bool. However, good style demands that a C++ programmer use type bool, not int and the constants true (not 1) and false (not 0), for true/false situations.

- **Enumerated types.** Enumerated type declarations were one of the last additions to the C language prior to standardization. In older versions of the language, #define statements were used to introduce symbolic codes, and a program might start with dozens of #define statements. They were tedious to read and write and gave no clue about which symbols might be related to each other. Enumerations were added to the language to provide a concise way to define sets of related symbols. For example, a program might contain error codes and category codes, all used to classify the input. Using #define, there is no good way to distinguish one kind of code from the other. By using two enumerations, you can give a name to each set and easily show which codes belong to it.

In C, enumeration symbols are implemented as integers, not as a distinct, identifiable type. Because of this, the compiler does not generate type errors when they are used interchangeably, and many C programmers make no distinction. In contrast, in C++, an enumeration forms a distinct type that is not the same as type int or any other enumeration. Both traditional enumerations and the newer enum class. are supported. The constant symbols in an enum class are “type safe” and will not be confused with similarly-named constants in other enum classes. Compilers will generate error comments and warnings when they are used inappropriately. Examples are given in the Brackets program in the next chapter.

- **Type conversions.** C provides “type cast” operators that convert a data object from one type to another. Casts are predefined from any numeric type (double, float, int, unsigned, char) to any other numeric type, and from a pointer of any base type to a pointer of any other base type. These casts are used automatically whenever necessary:
  - When an argument type fails to match a parameter type.
  - When the expression in a return statement does not match the declared return type.
  - When the types of the left and right sides of an assignment do not match.
  - When two operands of a binary operator are different numeric types.

These rules are the same in C++, but, in addition, the programmer can define new type casts and conversions for new classes. These operations can be applied explicitly (like a type cast) and will also be used by the compiler, as described above, to coerce types of arguments, return values, assignments, and operands.
2.4. WHAT IS OO?

What is a class? The description of a data structure and its associated behaviors. The behaviors include functionality, restrictions on visibility, and relationships with other classes. There are several kinds of classes:

- A data class models the state and behavior of a real-world object.
- A controller class uses inputs to control a process.
- A viewer class presents the state of a computation in a way that can be understood by the user.
- A business logic class implements the policies of an organization.

A program, or application, is constructed of a main function and many classes that, together, model some real-world process (a game, a calculation, data collection and processing, etc.).
What is an object? An object is an instance of a class, that is a structure that has all the data members and functionality that have been declared by the class definition. In C++ every instance of the same class has the same parts.

In some languages, a distinction is made between objects (instances of classes) and primitive values (integers, characters, etc.). This is not true in C++; a program can use a primitive value in the same contexts in which objects are used.

The behavior of a class is defined by its functions. In C++, a function can have more than one definition; each definition is called a method. In this text, we will use the term function to refer to a collection of methods with the same function name and the word method to refer to an individual method definition.

This chapter explains how state and behavior are represented within classes and introduces the concept of data encapsulation.

What class relationships are supported by C++? A program is built from interacting classes.

• A class A can compose another class, B. This means that an object of type B is one of the parts of an A object.
• A class A can be associated with another object, B. This means that an A is connected by a pointer to a B object.
• A class B can be derived from class A. This means that B has all the parts of A plus more parts of its own. We say that A is the base class and B is a derived class, or subclass that inherits members from A. Typically, A will have more than one subclass.

Instances of class B have two types, A and B, and can be used in any context where either type A or type B is accepted.

• An base class represents the common parts of more than one subclass.
• The base class is polymorphic if it defines at least one virtual function. If all of its functions are non-virtual, it is not polymorphic even if it has two or more derived classes. Polymorphic classes will be explained fully in a future chapter.
• An abstract class is a polymorphic class that characterizes some functionality without implementing it. All implementation is left for the derived classes.
• A class A can give friendship to B. This means that A permits B to manipulate its private parts. Friend classes will be explained fully in a future chapter. At this time, it is enough to know that they are used to build data structures (such as linked lists) that require two or more mutually-dependant class declarations.
• A object of class A can sometimes be cast to an object of class B.

During this course, all of these relationships will be introduced and used. A diagram that shows all of an application’s classes and class relationships is called a UML class diagram. UML diagrams are an important way to understand the overall structure of a complex application.

What is a Template? A template is a partially-abstract class declaration. By itself, a template is not compilable code. Typically, templates are defined to represent data structures (often called collections) such as stacks and queues. They are abstract because the type of the data to be stored in the collection is not declared as part of the template. Instead, a type-variable name is used in the code.

To use a template, a program instantiates it by specifying the type of data that will be stored in the collection. The compiler then replaces the type-variable name by the instantiating type name to produce concrete, compilable code.

Templates are important at two levels. First, the templates supplied by the C++ standard template library implement most of the data structures and many of the standard algorithms that programmers need. Using STL templates can save any programmer (especially beginners) time and hassle. However, we are not limited to the templates provided by STL. This book covers the techniques needed to make new data structures and new templates.
2.5 Header Files and Code Files

Most C++ classes have both a header .hpp file and an implementation .cpp file. Very simple classes are sometimes defined entirely in the header file.

The purpose of a header file is to provide information about the class interface for inclusion by other modules that use the class. A basic C++ header file has several parts:

- It #includes all headers needed for any declaration or function used by the class.
- It may contain #define and/or typedef statements.
- It may contain other type declarations, especially enumerated types.
- The header file contains the class declaration, which starts with the word class and the name of the class. Following that are declarations for the members of the class, both data declarations and method prototypes, enclosed in braces.
- The actual code for the class methods might or might not be inside the class definition. Short method definitions are there, but longer ones are defined separately, usually in an implementation file (.cpp, but possibly after the end of the class declaration, using the keyword inline.
- The header file must contain the full definition of all inline functions related to the class. Functions that are fully defined inside the class are automatically inline. Those outside the class declaration must start with the keyword inline.
- The header file must not contain data declarations outside the class declaration except those initialized by a constexpr expression.
- Executable code, other than definitions of inline methods, does not belong in a .hpp file.

**Inline and non-inline definitions.** A class method can be fully defined within the class declaration or just prototyped there and defined elsewhere. Although there are no common words for these placement options, I call them **in-class** and **remote**. Being in-class or remote has no connection to privacy: both public and private methods can be defined both ways.

An inline function call is replaced prior to compilation, by the full definition of the function. This always saves execution time. It also saves memory space at run time if the function is very short (one line or two), or if it is called only once.

The definitions of remote class methods, if any, are written in a .cpp file. Each .cpp file will be compiled separately and later linked with other modules and with the library methods to form an executable program. This job is done by the system linker, usually called from your IDE. Non-inline methods that are related to a class, but not part of it, can also be placed in its .cpp file.

2.6 Class Basics

A class is used in C++ in the same way that a struct is used in C to form a compound data type. In C++, both may contain method members as well as data members; the methods are used to manipulate the data members and form the interface between the data structure and the rest of the program. There is only one difference between a struct and a class in C++:

- Members of a struct default to public (unprotected) visibility, while class members default to private (fully protected).
- Privacy allows a class to encapsulate or “hide” members. This ability is the foundation of the power of object-oriented languages.
- In both a struct and a class, the default visibility can be overridden by explicitly declaring public, protected, or private.
The rules for class construction are very flexible: method and data members can be declared with three different protection levels, and the protections can be breeched by using a friend declaration. Wise use of classes makes a project easier to build and easier to maintain. But wise use implies a highly disciplined style and strict adherence to basic design rules.

Public and private parts. The keywords public, protected, and private are used to declare the protection level of the class members. Both data and method members can be declared to have any level of protection. However, data members should almost always be private, and most class methods are public. We refer to the collection of all public members as the class interface. The protected level is only used with polymorphic classes.

```
// Documentation for author, date, nature and purpose of class.
//-------------------------------------------------------------------
#pragma once
#include "tools.hpp"

class Mine {  
    friend class and friend function declarations, if any;
    private:  // -----------------------------------------------------
        Put all data members here, following the format:
        TypeName  variableName = initializer;                // Comment on purpose of member
        Put private function prototypes here and definitions in the .cpp file.
        Or put the entire private inline function definitions here.

    public:  // -----------------------------------------------------
        Mine (param list) : ctor initializer list {                                   // constructor
            initialization actions
        }
        ~Mine() {   ...  }                                       // destructor
        ostream& print ( ostream& s );                // print function, defined in .cpp file
        Put other interface function prototypes and definitions here.
    
    inline ostream& operator<<(ostream& st, Mine& m){ return m.print(st); }
}
```

Figure 2.1: The anatomy of a class declaration.

The Form of a Class Declaration Figure 2.1 illustrates the general form of a header (.hpp) file and the parts of a typical class declaration.

2.6.1 Data members.

A class normally has two or more data members that represent parts of some real-world object. An individual data member is used by writing an object name followed by a dot and the member name. Data members are normally declared to be private members because privacy protects them from being corrupted, accidentally, by other parts of the program.

- Please declare data members at the top of the class. Although they may be declared anywhere within the class, your program is much easier for me to read if you declare data members before declaring the methods that use them.
- Taken together, the data members define the state of the object.
- Read only-access to a private data member can be provided by a “get” method that returns a copy of the member’s value. For example, we can provide read-only access to the data member named name, by writing this public method: const string getName(){ return name; } In this method, the const is needed only when the return type is a pointer or structured object; for simple numbers it can be omitted.
• In managing objects, it is important to maintain a consistent and meaningful state at all times. To achieve this, all forms of assignment to a class object or to any of its parts should be controlled and validated by methods that are class members. As much as possible, we want to avoid letting a function outside the class assign values to individual class members one at a time. For this reason, you must avoid defining “set” methods in your classes.

• Some OO books illustrate a coding style in which each private data member has a corresponding public “set” method to allow any part of the program to assign a new value to the data member at any time. Do not imitate this style. Public “set” methods are rarely needed and should not normally be defined. Instead, the external function should pass a set of related data values into the class and permit member methods to validate them and store them in its own data members if they make sense.

The Form of a Class Implementation Figure 2.2 shows the skeleton of the corresponding .cpp file. It supplies full definitions for member methods that were only prototyped within the class declaration (.hpp). Note that every implementation file starts with an #include for the corresponding header file.

```cpp
#include "tools.hpp"
#endif
```

Figure 2.2: The anatomy of a class declaration.

2.6.2 Typical Class Methods

Almost every class should have at least three public members: a constructor, a destructor, and a print function.

• The name of a constructor is the same as the name of the class. A constructor method initializes a newly created class object. More than one constructor can be defined for a class as long as each has a different signature.

• The name of the destructor is a tilde followed by the name of the class (∼Point). There are no parameters, and a class can have only one destructor. A destructor method is used to free dynamic storage occupied by a class object when that object dies. Any parts of an object that were dynamically allocated must be explicitly freed. If an object is created by a declaration, its destructor is automatically called when the object goes out of scope. If an object is created by new, the destructor must be invoked explicitly by calling delete.

---

3The signature of a method is a list of the types of its parameters.
• A print() method defines the class object’s image, that is, how should look on the screen or on paper. Print functions normally have a stream parameter so that the image can be sent to the screen, a file, etc. Name this function simply “print()”, with no word added to say the type of the thing it is printing, and define a method for it in every class you write. In your print() method, format your class object so that it will be readable and look good in lists of objects.

2.7 A First Example: Sorting a list of strings.

The purpose of this section is to illustrate interactive input and output in the context of a simple one-class program. The code embodies a variety of C++ and OO design, coding, and style principles. Use this example as a general guide for doing your own work. The most important themes are:

Objectives. This is a C++ implementation of a very simple program: input some names, sort them, and display the results. The instructional targets are to:

- The general form of all C++ programs.
- Modular code with a short main function.
- Header files and implementation files.
- Include commands - what, where, and why?
- A class with all the normal parts: private data, constructor, destructor, and public functions.
- Inline and out-of-line functions.
- Debugging printouts.
- Strings and vectors
- Standard algorithms: sort and iterators.
- Inline and out-of-line functions.
- Encapsulation: this class takes care of itself. The functions that belong to the MemList class are the only ones that operate on the class’s data members.

The main program is given first, followed by the header and implementation files for the class. Detailed notes, keyed to the line numbers in the code, help the reader interpret the meaning. A call chart for the program is given in Figure 2.3. In the chart, white boxes surround functions defined in this application, light gray boxes denote functions in the standard libraries and dark gray denotes the tools library.

![Figure 2.3: A call chart for MemSort.](image-url)
2.7. A FIRST EXAMPLE: SORTING A LIST OF STRINGS.

2.7.1 Main Program

```cpp
// // Main program for the Membership List. main.cpp
// Created on May 29, 2023.
// // ---------------------------------------------------------------
#include "tools.hpp"
#include "memlist.hpp"

// // Read a series of names from the keyboard, store, sort, and display them.
int main( void ){
    int nMembers;
    banner();
    MemList theData;
    cout << "Constructed empty MemList" ;
    cout << "\nEnter names of your club members, period to finish. \n" ;
    theData.readData();
    cout <<"\n----------------- Beginning to sort.\n";
    theData.sortData();
    cout <<"\n----------------- Sorted results:\n";
    theData.printData();
    bye();
    // return 0; // Optional on my machine
}
```

- Lines 1–4: Every file in every program should start with a comment identifying the author’s name, the
date last modified, and the name of the file.
- Lines 5–6: Following that are the necessary #include commands. I expect you to include and use functions
from the tools files in every program.
- Lines 12 and 24: The main program starts by calling banner() and ends by calling bye(). Do this!
- Line 14 instantiates the MemList class and creates an object, named “theData”.
- The rest of main() (lines 15..23) is simply an outline of the processing steps alternating with user feedback.
All work is delegated to functions. A call chart (Figure 2.3) shows the overall dynamic structure of the
program and is an important form of documentation. The arrows indicate which functions called which
other function. Frequently, the standard I/O functions are omitted from such charts.

2.7.2 The MemData Class Declaration

- Type declarations, #defines, and prototypes are normally found in header files.
- Line 34 must be in each of your programs. Note the use of quotes, not angle brackets for a local library.
Note also that we include the header file, not the code file, because we are doing a multi-module compile
with a separate link operation.
- Lines 31–33 can be omitted because they include files that are also included by the local tools library,
- These headers are included here only for emphasis.
- Lines 36–47 define the class MemList, which has one data member, a constructor, a destructor, and three
functions.
- A vector is an array of data that can, and does, grow longer when needed. In C++, data members are
declared to be private. The member functions provide access to the data structure and they are the only
way to access it.
• The constructor is inline, fully defined in this header file. There is nothing for it to do, so it is defined with the default constructor, which does nothing.

• Likewise, there is nothing for the destructor to do because dynamic allocation is not used in the class. It is defined to be the default destructor, which does nothing.

• One of the three functions fits entirely on one line so it is defined in the .cpp file. We say it is its “inline”. Because it is so short, defining it this way decreases both run time and memory space used.

• Two of the three functions are longer than 2 lines, so they are defined in the .cpp file and only prototyped here.

2.7.3 The MemData Class Definition

• Line 52: The code file for a module must include its corresponding header file. Normally, it should not #include any other header files.

• The readData() function (lines 57...66) reads whitespace-delimited names from the keyboard and uses them to fill the array with data. We call this from line 17 of main. Look at the output at the end of this section and note that it does not matter whether the user types a space or a newline – they are treated the same way by >>.

• Line 59: Usually, a read-loop is terminated by an if...break (line 61). This is because, whether reading from the keyboard or from a file, a program must read PAST the last real input to discover that there is no more data. DO NOT try reading your files with a while loop; that just complicates the job.

• Line 64: The last action in the loop is to store the new data received into the vector, mems. It will be placed at the end of the vector, after prior data.

• Note that every function begins with a divider line and a brief comment about its purpose. Do this.

• Note that the return type is one one line, the name of the function on the next. This makes it easier to scan down a listing and find the function you are looking for. Do this!

• The printData() function (lines 69 – 73) prints an output header followed by the names in the list, one per line. We call it from line 23 of main.

• Most of the work in this function is done on one line (line 72) by a for-each loop. This kind of loop is very convenient when you want to process everything in a vector.
2.7. A FIRST EXAMPLE: SORTING A LIST OF STRINGS.

A. Fischer  
CSCI 4526 / 6626  
Sun Jun 12 2022 15:21:41

A. Fischer  
CSCI 4526 / 6626  
Sun Jun 12 2022 15:21:41

The output.

--- Constructed empty MemList  
Enter names of your club members, period to finish.  
Andy  
Carla Wendy Bill Fran  
1. Carla  
2. Wendy  
3. Bill  
4. Fran  
.  

--- Beginning to sort.

--- Sorted results:  
5 names were entered.  
Andy  
Bill  
Carla  
Fran  
Wendy

Normal termination.
2.8 Review of Topics from Intermediate C/C++

The example program in this section uses C++ features that should be familiar to students at this level but often are not. All are important in this class. Topics include:

- Using command-line arguments.
- Using enumerated types effectively.
- A UML class diagram.
- Using the tools library.
- A stack implemented by a vector.
- public and private
- Constructors and destructors.
- Include commands - what, where, and why?
- Constructing a multi-module program.

More extensive documentation is given below on the first three topics. Following that is the code. This program takes source code as input and answers whether the various kinds of parens and brackets are properly balanced. A simplified UML class diagram for the program is given in Figure 2.4. The main program is given first, followed by the header and implementation files for the three classes. Detailed notes are given for the program, keyed to the line numbers in the code, help the reader interpret the meaning.

2.8.1 Command-line arguments

In any fully functional C-environment, a program has the choice between accepting control information from the operating system at start-up time and operating with no context. The prototype used for main is different for the two purposes.

- No context: int main( void ).
- Arguments from the OS: int main( int argc, char* argv[] ).
  Where argc is the number of arguments supplied and argv is an array of C-strings that supplies the control information needed.

In these cases, the arguments (type char*) that come in from the OS must be converted to a useable form. Arguments typically have three purposes:

- File names, which can be used directly to open the file.
- Integers that specify a repetition count or the number of inputs to process. These must be converted from ASCII representation to numeric representation using atoi().
- Switches: codes for a random assortment of control information. (Beyond the scope of this course.)

The main program, below, gives a very simple example. Advanced programmers often use the second form, especially during development and testing, because it is the easiest and fastest way to supply a variety of input files to the program.

2.8.2 Enum Classes

Enumerations are used to clarify the intent of a program’s code, and to make it easy to classify complex information such as a person’s age. To use an enum class effectively, one needs these things:

- An enum class declaration: enum class Owner Mine, Yours, Theirs ;
- Variables of the enum type: Owner mary, john;
- A function, probably containing a switch, to translate from an input code or input format to one of the enum constants.
• A way to output the enum constant in English.
• Assignment statements that store enum constants in enum-type variables

The best way to explain all these things is through a meaningful program. In the example below, The Token class defines two enums and two enum variables. The Brackets class uses the enum variables in doTok() to make clear sense out of what is pushed onto the stack and what is popped.

### 2.8.3 The UML diagram

A UML class diagram can be detailed or simple. The top rectangle gives the class name. The second rectangle lists the member variables, and the third one lists the class functions. In a simplified diagram, no details are given in the lower two boxes, so they are often omitted. Black diamonds indicate composition: the composed class dies when its parent dies. The white diamond indicates aggregation: the contents of the stack are added after the stack is created. The asterisk indicates that many tokens are stored in a stack.

![Class Diagram](image)

Figure 2.4: A class diagram for Brackets.

### 2.8.4 Main Program

• Lines 1–4: Document file name, author’s name, date of each revision.
• Line 5: This contains include commands for all the standard header files that are commonly used. It also contains `using namespace std;`. You do not need to include any of these things in your own code. Further, using “std::” in your code is redundant. Please omit it. Finally, please customize line 10 your copy of tools.hpp with your own name.
• Line 9: This announces that we will be using command-line arguments.
• Line 10: Call banner() on the first line of code. Having some output is a big help when your program is malfunctioning.
• Line 13: Responsible programming calls for checking whether you have the number of arguments you expect and need.
• Line 14: This output is for debugging. Tracking your progress is useful when things go wrong.
• Line 16: Use the C-string from the argument array to open the file.
• Line 17: DO NOT try to read from a file until you know it is really open. Several things can go wrong (spelling, permissions, path name).
• Lines 19–20: We create the application controller and call its primary function to read and process the file.
• Line 21: Clean up after yourself. Close your open files.
• Line 22: In this class, you are required to end your code by calling bye().

### 2.8.5 The Stack Class Declaration

• Lines 33–35: We presume that the student is already familiar with the use of vectors. Here, we adapt the vector class by adding definitions for the traditional stack functions.
• Lines 38–39: Default initializations here will trigger default initializations of the vector.
• Line Lines 41–44: The four traditional stack functions are translated to the functions defined by vector.
• Line 44: Vector::empty returns a boolean result. DO NOT USE IF to test things like this. Just use the
bool result provided.
• Line 47: A for-each loop is used to print the contents of the vector. All standard templates support
range-based for loops.
• All of these functions are brief and are defined inline. There is no need for a separate .cpp file.

2.8.6 The Brackets Class Declaration

Brackets is the controller class in this application.

• Lines 29, 60, and 134: All header files start with #pragma once or with the traditional three line include
guard: ifndef CLASSNAME, #define CLASSNAME, and finally #endif.
• Lines 67: All data members should be private.
• Lines 68–69: There are two private functions. Both are called from class functions and should never be
called from outside the class. This is how a class preserves consistency and validity.
• Lines 72–73: It is good style to explicitly declare the constructor and destructor. However, these two lines
could be omitted without changing the meaning of the program.
• Line 74: The actions of this controller class are all called from this one public function. The parameter is
an open input stream, passed by reference.

2.8.7 The Brackets Class Definition

The analyze() function. This function reads the file one character at a time.

• Line 85: Most input loops are best written with an infinite for loop. Using a while loop does not work
right unless you are also using getline().
• Line 86: DO NOT use getline to read a single char. Use the language the way it was intended.
• Lines 87–91: Test for eof after reading input, not before. File handling will be explained in Chapter 3.
• Line 87: If there is good input, doTok() is called to process it.
• Lines 92–96: This is a test for a well-nested set of brackets. If the stack is NOT empty at end of file, there
is an an unmatched open-bracket.

The mismatch() function provides all available information about the error for the user.

The doTok() function. This function creates a token for each bracket and ignores all other characters.

• Line 108: A token is created to store the input. The Token constructor will classify it and assign two
c enum constants to its two data members.
• Line 109: Debugging output.
• Lines 110-128 ignore, push, or pop the token, depending on its classification.
• Lines 117–123: Test for a matching left-right pair of symbols. Two error conditions might be detected:
an mismatched pair, and nothing left on the stack.
2.8.8 The Token Class Declaration

- Lines 137–138: These enum classes are defined outside the token class so that they will be accessible to other parts of the program. They are set in the Token class and used in the Bracket class.
- Lines 137–138: The constant “NONE” is listed in both enums. These two constants look the same, but because these are enum classes, not C-style enums, they are not equal. When used in Brackets::doTok (line 111) and in print(), the class name is used with the enum constant.
- Line 137: This defines concise symbols for the four kinds of brackets we handle. The fifth symbol is for non-brackets.
- Line 138: In an expression, each kind of bracket must have an opening character ( (, [, {, and < ) on the left and a matching closing character ( ), ], }, and > ) on the right.
- Lines 142–144: The relevant style and sense for a bracket character are stored in the token with the char.
- Lines 148–149: This class constructor will be defined in the .cpp file; the destructor is defined here.
- Lines 150–151: Two brief inline functions are defined, a getter and a symbolic name for a simple test. Using symbolic names for class actions creates clearer, more maintainable, code.

2.8.9 The Token Class Definition

The Token constructor. Convert a character into a Token, complete with its classification codes.

- Line 166: Between the ) and the { is a constructor-initializer, also called a ctor, and introduced by a colon. The first ch refers to a class data member, which is being initialized. The second ch, in parentheses, refers to the parameter. These names may either be the same or different.
- Note: Contrary to common belief, there is no need to use “this->” anywhere inside a class function.
- Line 167: Every time a token is constructed, this string of bracket symbols is needed. We make it static so that it will be allocated and initialized at load time and remain available any time it is needed until program termination. Because it is declared inside a function, its scope of visibility is only within the same function.
- Line 168: Type ”string” supports an extensive library of functions; find() is one of the most useful. The string of bracket symbols ( () [] {} <> ) is searched for the current input character. If present, find() returns the subscript of the input in the string. If not, the constant npos is returned. The result is useful for validation and classification.
- Lines 169–172: If the input is not a bracket, we store “NONE” in the two enum data members.
- Lines 173–176: Thus clause handles input chars that ARE bracket symbols. If so, the style and sense of the bracket is stored in the Token.
- Line 174–175: The subscript of the current input in the string is k. If k is an even number (k%2 == 0) then it is an open-bracket. Odd symbols are close-brackets.
- Also, k/2 using integer divide, tells us which kind of bracket: an answer of 0 means parenthesis, an answer of 1 means square bracket, etc. Mapping the matching left and right brackets onto the same code is a big help in determining whether we have a matching pair.

The print function.
The output. This program was tested with four different input files; the output shown here is from the correct file.

Welcome to the bracket checker.
Checking file good1.txt

New token: Left-Paren’(
The stack is: bottom~~ Left-Paren’(~~top

New token: Left-Angle’<
The stack is: bottom~~ Left-Paren’( Left-Angle’<~~top

New token: Right-Angle’>
The stack is: bottom~~ Left-Paren’(~~top

New token: Right-Paren’)
The stack is: bottom~~~~top

New token: Left-Curly’{
The stack is: bottom~~ Left-Curly’{~~top

New token: Right-Curly’}
The stack is: bottom~~~~top

New token: Left-Square’[
The stack is: bottom~~ Left-Square’[~~top

New token: Left-Square’[
The stack is: bottom~~ Left-Square’[ Left-Square’[~~top

New token: Right-Square’]
The stack is: bottom~~ Left-Square’[~~top

New token: Right-Square’]
The stack is: bottom~~~~top
The brackets are properly nested and matched.

Normal termination.
Chapter 3: **C++ I/O**

How to learn C++:

*Try to put into practice what you already know, and in so doing you will in good time discover the hidden things which you now inquire about.*

— Henry Van Dyke, American clergyman, educator, and author.

This chapter assumes basic knowledge of the C++ input and output. It attempts to help the reader develop more advanced I/O understanding and skills. C++ supports a generic I/O facility called “C++ stream I/O”. Input and output conversion are controlled by the declared type of the I/O variables: you write the same thing to output a double or a string as you write for an integer, but the results are different. This makes casual input and output easier. However, controlling field width, justification, and output precision can be a pain in C++. The commands to do these jobs are referred to as “manipulators” in C++ parlance and are defined in the `<iomanip>` library.

You can always use C formatted I/O in C++; you may prefer to do so if you want easy format control. Both systems can be, and often are, used in the same program. However, in this class, please use only C++ I/O.

### 3.1 Streams and Files

A *file* is something that is stored on a memory device. A stream is a data structure within a program that lets us read or write data to a file. The stream data structure contains storage for a block of data and the error flags, cursors, etc. needed to access that data.

Throughout, we will deal exclusively with text files. Please remember that a text file is a sequence of ASCII or Unicode characters—numbers are represented by strings of digits. Basic input for all predefined types is the same and is very easy.

Stream classes are built into C++. They form a class hierarchy whose root is the class `ios`. This class defines flags and functions that are common to all stream classes. To the right of `ios` are the two classes whose names are used most often: `istream` for input and `ostream` for output. The predefined stream `cin` is an `istream`; `cout`, `cerr`, and `clog` are `ostreams`. These are all sequential, character-based streams—data is either read or written in strict sequential order. The `iostreams` are derived from both `istream` and `ostream`; they permit random-access input and output, such as a database would require. Further to the right are the main stream classes for file-based streams and strings that emulate streams.

![Figure 3.1: The stream type hierarchy.](image-url)
Each class in the stream hierarchy is a variety of the class to its left. When you call a function that is defined for a class toward the left side, you can use an argument of any class that is connected to it on its right. For example, if `fin` is an `ifstream` (used for an input file) you can call any function defined for class `istream` or class `ios` with `fin` as an argument.

The functions and constants for all these classes you will need this term are defined in four header files:

- `<iomanip>` for format control.
- `<iostream>` for interactive I/O.
- `<fstream>` for file I/O.
- `<sstream>` for strings that emulate streams, used to parse a line of input after reading it.

The remainder of this chapter will cover many details of the C++ I/O system in approximately the same order as the columns in the diagram.

### 3.1.1 The base class for streams: ios.

**Manipulators.**  
#include `<iomanip>`; Or include tools.hpp. A manipulator in C++ is a function that affects the internal settings of an stream or an output stream. It modifies the stream and returns that same stream, so that it can be used in a chain of input or output operators. Manipulators are designed to be used in with the I/O operators `>>` and `<<` on streams. For example: `cout << boolalpha;` sets a flag in the standard output stream to output true and false values as text instead of as integers.

Many manipulators are supported, others could be defined, if desired:

**Number-base flags:**

- `dec`: Use base-10 number conversion.
- `hex`: Use hexadecimal (base 16) number conversion.
- `oct`: Use octal number conversion

**Input manipulators:**

- `skipws`: Skip leading whitespace when reading text.
- `noskipws`: Do not skip leading whitespace

**Output manipulators**  The last three require inclusion of `<iomanip>`.

- `endl`: Insert newline and flush the buffer.
- `ends`: Insert a null character.
- `flush`: Flush the stream buffer.
- `boolalpha`: Set the stream to output true and false values as text instead of as integers.
- `noboolalpha`: Set the stream to output true and false values as integers 1 and 0.
- `setfill(‘.’)`: Set the fill character to be a period
- `setprecision(n)`: Output numbers with n places of decimal precision
- `setw(n)`: Set the width of the next number to n.

**Floating-point format flags**  These manipulators set the floatfield flag in the stream:

- `fixed`: Use fixed-point number conversion.
- `scientific`: Use scientific notation for number conversion.
Output alignment flags These manipulators set the adjustfield flag in the stream:

- **left**: Left-align the output.
- **right**: Right-align the output.
- **internal**: Adjust width of field by inserting characters at an internal position.

**Manipulators are functions.** The C++ standard defines many useful manipulators but misses some that are useful, such as the two that are shown below. The language standard provides manipulators to change numeric output from the default format (like `%g` in C) to fixed point `%f` or scientific `%e` notation. However, it does not provide a manipulator to change back to the default `%g` format. This function does the job:

The tools library contains a definition of `general` as a manipulator for `ostreams`. This makes it more intuitive for the programmer to format real numbers.

```cpp
// ----------------------------------------------------------------------
ostream& general( ostream& os ){ // Use: cout << fixed << x << general << y;
    os.unsetf( ios::floatfield );
    return os;
}
```

The tools also define `flush` as a manipulator for `istreams`, allowing the programmer to empty the keyboard buffer after an input error and leave the buffer in a predictable state: empty.

```cpp
// ----------------------------------------------------------------------
// Flush cin buffer: cin >> x >> flush >> y; or cin >> flush;
istream&
flush( istream& is ) { return is.seekg( 0, ios::end ); }
```

### 3.1.2 Class `iostream`

Four instances of this class are defined and opened each time a new process is started up, and before control is transferred to the main() function of the new process.

- **cin**: This is the standard input stream. It is connected to the computer’s keyboard.
- **cout**: This is the standard output stream. It is connected to the computer’s video screen.
- **cerr**: This stream is used by the system for run-time errors. It, also, is connected to the computer’s video screen. Thus, a user may see output from cout and cerr mixed together. You may wish to use the cerr stream for debugging output.
- **clog**: The clog stream is used by processes that log transactions. It will not be used in this course.

**Buffered and unbuffered streams.** When using `cout` or an `ofstream`, information goes into the stream buffer when an output statement is executed. It does not go out to the screen or the file until the buffer is flushed. In contrast, something written to `cerr` is not buffered and goes immediately to the screen.

When using `cin`, a line of text stays in the keyboard buffer until the return key is pressed. This allows the user to backspace and correct errors. When return is pressed, the entire line goes from the keyboard buffer to the stream buffer, where it stays until called for by the program. The program can read the entire line or any part of it. The rest will remain in the stream buffer, available for future read operations. In general, newlines in an input file are treated the same as spaces.

**Interactive vs. file-based streams.** The interactive streams (defined in the class `iostream`) are pre-declared and pre-opened for every process. Using a file-based stream requires additional code:

- `#include <fstream>;
- Open the stream to read or write the file.
- Test for successful opening.
- During processing, the program must check for end-of-file in an ifstream.
- Eventually, the file must be closed, either by the program or by the system when the program terminates.
There are several ways to get a file name into a program:

1. Make it a command-line argument, as in prior examples. This is the most powerful and flexible method. Use it when there might be more than one input file in your test suite.

2. Define the filename as a constant above main. This method is OK, but makes it difficult to take different files as inputs, and therefore, difficult to debug a program.

3. Prompt the user to input a file name. A good idea for beginners. Its shortcomings are that testing becomes slower and clumsier, and that interaction of this sort does not work with automated testing.

4. Embed the file name in an open statement somewhere in the code. Do not do this. It makes a program non-portable and difficult to maintain. Keep in mind that a Windows pathname does not work on Linux or on a Mac.

Opening an fstream. There are two ways to open a stream. The preferred way is to use the ifstream or ofstream constructor with a file name parameter. The other way is to declare an instance of a stream, then open it in a separate statement.

```cpp
ifstream fin ( "myin.txt" ); // Declare stream and open for reading.
ofstream fout ( "myout.txt" ); // Declare stream and open for writing.
fin.open( "myfile.in" ); // Alternate way to open a stream.
```

In either case, one must test whether the stream is actually open before using it:

```cpp
if (!fin.isopen()) fatal(...); // Test for unsuccessful open.
```

If you attempt to open a file that does not exist, the operation fails and returns a null pointer. There are several reasons (spelling, pathnames, permissions) why an opening command might fail. A very common beginner’s error is to try to use the file without checking. Forgetting to check for opening failure results in confusing error comments and many bewildered students.

Flushing buffered output streams. The output from a buffered stream stays in the stream buffer until it is flushed, which happens under the following circumstances:

- When the buffer is full.
- When the stream is closed.
- When the program outputs an `endl` on any buffered output stream. `endl` is a manipulator that ends the output line, but its semantics are subtly different from a newline character.
- When the program calls `flush`.

Flushing input streams. The built-in `flush` and `fflush` apply only to output streams. Some students are convinced that that `flush` also works on input stems. It does not, but they are confused because of the stream ties. However, after detecting an error in an interactive input stream, it is usually a good idea to flush away whatever remains in the input stream buffer.

Appending to a file. A C++ output stream can be opened in the declaration. When this is done, the previous contents of that file are discarded. To open a stream in append mode, you must use the explicit `open` function, thus:

```cpp
ofstream fout;
fout.open( "mycollection.out", ios::out | ios::app );
```

The mode `ios::out` is the default for an ofstream and is used if no mode is specified explicitly. Here, we are also specifying append mode, “app”. In this case, one must write the “out” explicitly.
3.2. INPUT STREAMS: ISTREAM AND IFSTREAM.

Reading and writing binary files. The open function can also be used with binary input and output files:

```cpp
ifstream bin;
ofstream bout;
bin.open( "pixels.in", ios::in | ios::binary );
bout.open( "pixels.out", ios::out | ios::binary );
```

Stream ties. The stream cout is tied to cin, that is, whenever the cout buffer is non-empty and input is called for on cin, cout is automatically flushed. This is also true in C: stdout is tied to stdin. Both languages were designed this way so that you could display an input prompt without adding a newline to flush the buffer, and permit the input to be on the same line as the prompt.

Closing an fstream. An fstream may be closed in more than one way:

- Call the close function: `fin.close();`
- The fstream's destructor will be run when the fstream goes out of scope.
- When execution ends normally, the fstream will be closed.

Closing an output stream by any of these means will automatically flush its buffer. If the program ends with a call on `exit()`, however, the buffer might not be flushed.

When should you close streams explicitly?

- Always, if you want to develop good, clean, habits that will never mislead you.
- Always, when you are done using a stream well before the end of execution.
- Always, when you are using Valgrind or a similar tool to help you debug.
- In general, it is considered good practice to close a stream as soon as the program is done with it.

Detecting end-of-file. In both C and C++, the end-of-file flag does not become true until you attempt to read data beyond the end of a file. The last line of data can be read fully and correctly without turning on the end-of-file flag if there is are one or more newline characters on the end of that line. Therefore, an end-of-file test should be made between reading the data and attempting to process it. The cleanest, most reliable, simplest way to do this is by using an if...break statement to leave the input loop. In general, the test should come immediately after the line that reads the input data.

```cpp
if (fin.eof()) break; // Test for end of file.
```

Error detection and recovery. In a production program, error detection and recovery is often essential. However, this is a large and detailed subject that is usually omitted from classroom instruction for beginners and intermediate students. As a programmer's skill advances, however, these things must be mastered. These are the error detection functions:

```cpp
if (fin.good()) ... // Test for no errors after a read operation.
if (fin.fail()) ... // Test for hardware failure OR number conversion error.
if (fin.bad()) ... // Test for hardware error.
fin.clear(); // Clear the stream's error flags to permit processing to continue.
fin.ignore(1); // Remove the offending character from the stream.
```

Error recovery is covered in detail, with a full example, at the end of this chapter.

3.2 Input streams: istream and ifstream.

3.2.1 Simple types.

On simple types like numbers, chars, and pointers, input is done in one of two ways:
• Use the *extraction operator, `>>`. It parses the input stream, extracts characters from it, and does any appropriate number conversion (ASCII to numeric). The result is then stored in a variable.

• You can call `getline()` to read an entire line of input into a char array or a string. If using an array, you must worry about buffer overflow. Then you must parse it yourself, and do the number conversion yourself by calling functions like `atoi` and `strtol`. Input can be done this way, however, DO NOT DO IT. It is not necessary and not helpful to reinvent the wheel. Learn to use the `>>` operator as it was intended!

**How much was read?** After any read operation, the function `gcount()` contains the number of characters actually read and removed from the stream. Saving this information is sometimes useful. The value returned by `gcount()` or `get()` is the number of characters that have been removed from the stream.

**Operator `>>`.** The extraction operator is polymorphic. It is predefined for all the built-in types and can be extended to handle any program-defined class. Basic input is quite simple when using it. It skips leading whitespace characters, reads the digits of a number, and stops at the first whitespace or non-numeric input character. It is not necessary and not helpful to do your own number conversion using `atoi` or `strtol`. Learn to use the stream library as it was intended!

Several operations that use `>>` can be chained (combined in one statement) but those based on `get()` and `getline()` (described below) cannot. A single call on a `get()` or `getline()` function is a complete statement.

**Input errors.** Operator `>>` works properly if the input is correct. However, if a number is expected, and anything other than a number is in the input stream the stream will signal an error. A bullet-proof program tests the stream status to detect such errors using the strategies explained in this chapter. The `clear()` function is used to recover from such errors.

**Single character input.** When reading a single character, a programmer might with to read only visible inputs, or might need to read whatever character is next, even if it is a whitespace character. In C++, two different functions exist for the two different tasks.

• Use `>>` to read the next non-whitespace char. (It skips leading whitespace.)

• Use `get()` to read the next character. This is useful when a program needs to know exactly what characters are in the input stream, and does not want whitespace skipped or modified. Example: a program that compresses a file.

```cpp
char ch;
ch = cin.get();
get (ch);
```

### 3.2.2 String input.

The simplest form for a string input command should NEVER be used: `cin >> w1;`. It skips leading whitespace, reads characters starting with the first non-whitespace character, and stops reading at next whitespace. The input characters are stored in `w1`. However, there is no limit on the length of the string that is read. **DO NOT DO THIS!** If you try to read an indefinite-length input into a finite array, and you do not specify a length limit, the input can overflow the boundaries of the array and overlay nearby variables. A program that makes this error is open to exploitation by hackers.

The simplest method of string input is called “string getline” and is used to input characters into variable of type string. Please use these data declarations:

```cpp
ifstream fileIn( "demo.txt")
string input1;
char arr[80];
```

The string input functions, with examples are:
3.2. **INPUT STREAMS: ISTREAM AND IFSTREAM.**

- **istream& getline (istream& is, string& str);**
  - Read an entire line, up to the newline character, from an input stream into a dynamically allocated string of an appropriate length. Remove the newline character from the stream.
  - getline (cin, str);
  - getline (fileIn, input1);

- **istream& getline (istream& is, string& str, char delim);**
  - Read a line up to the specified delimiting character, from an input stream into a dynamically allocated string of an appropriate length. Remove the delimiter from the stream.
  - getline (cin, str, char delim);
  - getline (fileIn, input1, char ':');

**istream::getline()** Do not confuse the string-getline functions with the older C++ istream::getline() functions that read data into a character array. The istream::getline() functions are only safe when used with a length limit. Prototypes and examples are given below.

- **istream& istream::getline (char* arr, arraysize n );**
  - Read an entire line up to the newline character, from the given input stream into the array named **arr**. The second parameter is the length of **arr**. Remove the newline character from the stream.
  - cin.getline( arr, 80 );
  - fileIn.getline( arr, 80, ';' );

- **istream& istream::getline (char* arr, arraysize n, char delim );**
  - Read a line, up to the specified delimiting character, from the given input stream into the array named **arr**. Remove the delimiter from the stream.
  - getline( in, str );
  - getline(fileIn, input1);

- **istream& istream::get (char* arr, arraysize n );**
  - **istream& istream::get (char* arr, arraysize n, char delim);**
  - Read a line, up to the newline or delimiting character, from the given input stream into the array **arr**. Stop after n-1 chars have been read. **DO NOT** remove the delimiting character from the stream. Therefore, after using **get()** to read part of a line (up to a specified terminating character), you must remove that character from the input stream. The easiest way is to use **ignore(1)**.

**What should you use?** The string-getline() functions are safer and easier to use, and can do almost everything that the istream::getline() functions do. Unless you are doing something unusual, you should use the newer, safer, easier versions. If you truly need the string in an array, not a string variable, extract the contents when the read operation is finished: arr = str.data();

**Whitespace Problems.** When you are using >>, leading whitespace is automatically skipped. However, before reading anything with **get()** or **getline()**, whitespace must be explicitly skipped unless you want the whitespace in the input. Use the **ws** manipulator for this purpose, not **ignore()**. This skips any whitespace that may (or may not) be in the input stream between the end of the previous input operation and the first visible keystroke on the current line. Usually, this is only one space, one tab, or one newline at the end of a prior input line. However, it could be more than one keystroke. By removing the invisible material using **ws** you are also able to remove any other invisible stuff that might be there.
3.3 Output streams: ostream and ofstream

The C++ output stream, `cout`, was carefully defined to be compatible with the C stream, `stdout`; they write to the same output buffer. If you alternate calls on these two streams, the output will appear alternately on your screen. However, unless there is a very good reason, it is better style to stick to one set of output commands in any given program. For us, this means that you should use C++ version consistently and avoid `printf()`.

**Simple types, pointers, and strings.** The table below shows alternative output format specifiers for several types. It uses the stream operator: `<<` and the stream manipulators `hex`, `dec`, and `endl`. Note that the string "\n", the character '\n', and the manipulator `endl` can all be used to end a line of output.

However, for file-streams there is one difference: the manipulator flushes the output stream; the character and string do not.

- For output, the *insertion* operator, `<<` inserts values into an output stream.

<table>
<thead>
<tr>
<th>Type</th>
<th>Function call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric or char:</td>
<td><code>cout &lt;&lt;&quot;Initial:\&quot; &lt;&lt;c1 &lt;&lt;&quot; age:&quot; &lt;&lt;k1 &lt;&lt;\n’;</code></td>
</tr>
<tr>
<td>char[] or char*</td>
<td><code>cout &lt;&lt;word &lt;&lt;&quot;...&quot; &lt;&lt;w2 &lt;&lt;&quot; &quot; &lt;&lt;w3 &lt;&lt;&quot;\n”;</code></td>
</tr>
<tr>
<td>pointer in hexadecimal</td>
<td><code>cout &lt;&lt;px1 &lt;&lt;endl;</code></td>
</tr>
<tr>
<td>integer in hexadecimal</td>
<td><code>cout &lt;&lt;hex &lt;&lt;k1 &lt;&lt;‘ ” &lt;&lt;k2 &lt;&lt;dec &lt;&lt;endl;</code></td>
</tr>
</tbody>
</table>

**Output in hexadecimal notation.** Manipulators are used to change the setting of a C++ output stream. When created, all streams default to output in base 10, but this can be changed by writing the manipulator `hex` or `oct` in the output chain. Once the stream is put into hex or oct mode it stays there until changed back by the `dec` manipulator.

**Field width, fill, and justification.** This table shows how to use the formatting functions for justification, field width, and fill character. Note: you must specify field width in C++ separately for every field. However, the justification setting and the fill character stay set until changed.

<table>
<thead>
<tr>
<th>Style</th>
<th>How To Do It</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 columns, default justification (right).</td>
<td><code>cout &lt;&lt;setw(12) &lt;&lt;k1 &lt;&lt;&quot; &quot; &lt;&lt;k2;</code></td>
</tr>
<tr>
<td>k1 in 12 cols (. fill) then k2 in default width</td>
<td><code>cout &lt;&lt;setw(12) &lt;&lt;setfill(‘.’) &lt;&lt;k1 &lt;&lt;k2 &lt;&lt;endl;</code></td>
</tr>
<tr>
<td>12 columns, left justified, twice.</td>
<td><code>cout &lt;&lt;left &lt;&lt;setw(12) &lt;&lt;k1 &lt;&lt;setw(12) &lt;&lt;k2 &lt;&lt;endl;</code></td>
</tr>
<tr>
<td>12 columns, right justified, - fill.</td>
<td><code>cout &lt;&lt;right &lt;&lt;setw(12) &lt;&lt;setfill(‘-’) &lt;&lt;k1 &lt;&lt;endl;</code></td>
</tr>
</tbody>
</table>

**Floating point style and precision.** This table shows how to control precision and notation, which can be fixed point, exponential, or flexible. All of these settings remain until changed by a subsequent call.

<table>
<thead>
<tr>
<th>Style</th>
<th>HowTo Do It</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default notation &amp; precision (6)</td>
<td><code>cout &lt;&lt;y1 &lt;&lt;’’ &lt;&lt;y2 &lt;&lt;endl;</code></td>
</tr>
<tr>
<td>Change to precision=4</td>
<td><code>cout &lt;&lt; setprecision(4) &lt;&lt;y1 &lt;&lt;’’ &lt;&lt;y2 &lt;&lt;endl;</code></td>
</tr>
<tr>
<td>Fixed point, no decimal places</td>
<td><code>cout &lt;&lt;fixed &lt;&lt;setprecision(0) &lt;&lt;y1 &lt;&lt;endl;</code></td>
</tr>
<tr>
<td>Scientific notation, default precision</td>
<td><code>cout &lt;&lt;scientific &lt;&lt;y1 &lt;&lt;endl;</code></td>
</tr>
<tr>
<td>Scientific, 4 significant digits</td>
<td><code>cout &lt;&lt;scientific &lt;&lt; setprecision(4) &lt;&lt;y1 &lt;&lt;endl;</code></td>
</tr>
<tr>
<td>Return to default %g format</td>
<td><code>cout &lt;&lt;general; /* Defined in tools.hpp. */</code></td>
</tr>
</tbody>
</table>
3.4 String streams: sstream.

A `stringstream` (string stream) is a stream whose data is stored as a string in main memory. A string-stream has all the flags, settings, and functions of a normal stream, so when a string-stream is used, all the standard input functions are available.

An `ostringstream` (output string stream) is like a Java StringBuilder. The purpose is to allow a program to build up a line of output, one part at a time, then output it as a single unit. While this is not usually necessary, it is useful in threaded applications to keep the output from each thread coherent and separate from other threads’ output.

An `istringstream` (input string stream) allows a program to read a line of data, then parse it using istream input operations. If an error occurs during the parsing, the program can write the whole line to `cerr` or `clog`, providing excellent information for a human who needs to clean up the damaged data.

3.5 Example 1: Command line, Files, and Error Handling

The focus of this section is basic C++ concepts and tools for input file handling. Topics include using the command line to select an input file, opening a file, end of file detection and input error detection. The code examples do nothing useful; they simply provide examples of using input streams containing text and numbers in a variety of ways. Each example presents a different technique.

3.5.1 Using the command line and string getline().

Command-line arguments allow the programmer to specify runtime parameters, such as a file name, at the time when the command is given to run a program. This is an important professional tool that needs to be understood. It is useful when you do not know the name of the input file at compile time. It is an alternative to querying the user for that name.

There are two ways to use file names as command line arguments:

1. The normal way: run the program from a command shell and type the file names after the name of the executable program.

2. In an integrated development environment (IDE), find a menu item that pops up a window that controls execution and enter the file names in the space provided. The location of these windows varies from one IDE to another, but the principle is always the same.

The following brief program illustrates how to use these command line arguments and the simplest way to read file input (so long as you only input lines of text, not numbers).

In `main()`: Command-line arguments and opening a file.

- Line 5 declares that the program expects to receive arguments from the operating system. If you have never seen command-line arguments, refer to Chapter 22 of *Applied C*, the text used in our C classes, or [www.cprogramming.com/tutorial/c/lesson14.html](http://www.cprogramming.com/tutorial/c/lesson14.html) “Accepting command line arguments in C using argc and argv”.
- Lines 7 and 26: As always, use `banner()` and `bye()` to begin and end each program.
- Line 9 says that the user should write the name of the executable file and one file name on the command line.
- Line 11 picks up the file-name argument and uses it to open an input stream. It is a good idea to open all files at the beginning of the program, especially if a human user will be entering data. There is nothing more frustrating than working for a while and then discovering that the program cannot proceed because of a file-opening error.
- Line 12 tests that the stream was actually opened properly. If not, execution is aborted.
• Line 16–22 are an input loop that exits when the program tries to read past the end of the file.
• Line 17 calls string-getline to read one line of the file.
• Line 21 tests the end-of-file flag. It is turned on when a program tries to read a line that is not there. There IS NO special end-of-file mark, and the flag does NOT go on when the last data is read.
• Line 21 would normally come immediately after the line 17, but this is a stream-demo program so we print out the stream flags before testing them and, potentially, leaving the loop.
• Line 25 closes the file. Closing a stream is considered good style. However, all files will be closed automatically when the program ends.

Reading the stream-status reports.

The file eofDemo.in will be used in this example. It is a normal text file with three normal lines, each ending in a newline character.

The three flags, defined in the ios class, are used to record the status of a stream. Their names are: eofbit, failbit, and badbit. They are turned on during a read operation when some exceptional condition is found. They remain set until explicitly cleared.

• Lines 13 and 14 print headings for the output. After each line in the file is read, the stream flags will be printed out.
• Lines 18–20 print the stream status, flags, and the actual input after executing getline().
• The value returned by rdstate() is a number between 0 and 7, formed by the concatenation of the status bits: fail/eof/bad.
• The fail() function returns a true result (which is printed as 1) if no good data was processed on the read operation. Usually this means that the input could not be converted to a number.
• The eof() function returns true when an end-or-file condition has occurred. There may or may not be good data, also.
• The bad() function returns true if a fatal low-level IO error occurred.
3.6. EXAMPLE 2: EOF AND ERROR HANDLING WITH NUMERIC INPUT.

- There is no “good” bit. The good() function returns true (which is printed as 1) when none of the three exception flags are turned on. As shown in the output, eof() can be true when good data was read, and also when no good data was read. When eof happens and there is no good data, the fail() is true.
- The results after reading the file eofDemo.in are:

```
Stream flags are printed after each read in this order:
state = goodbit : failbit : eofbit : badbit
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
0 = 1:0:0:0: Third line, with a newline on the end.
6 = 0:1:1:0:
```

This means that three lines were read (including the newlines on the end) and there were no read errors of any sort. The 4th read was unsuccessful because the third newline was the last char in the file. So the fail flag and end-of-file flag were turned on.

- In contrast, here are the results after reading the file eofDemo2.in. It is an almost identical file, but ends without a newline:

```
Stream flags are printed after each read in this order:
state = goodbit : failbit : eofbit : badbit
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
2 = 0:0:1:0: Third line, without a newline on the end.
```

We see that two lines were read without errors, but the third line turned on the eofbit because it hit an end of file before completing the read. When using getline(), the read operation is not complete until a newline character is found.

- The moral? It is normal to put a newline on the end of every line. If it is missing, errors can happen.

3.6 Example 2: EOF and error handling with numeric input.

Reading numeric input introduces the possibility of non-fatal errors and the need to know how to recover from them. Such errors occur during numeric input when the type of the variable receiving the data is incompatible with the nature of the input data. For example, if we attempt to read alphabetic characters into a numeric variable.

It is important to check for errors after reading from an input stream. If this is not done, and a read error or format incompatibility occurs, both the input stream and the input variable may be left containing garbage. The program in this section shows how to detect read errors and handle end-of-file and other stream exception conditions. After detecting an input error, the programmer should clear the bad character or characters out of the stream and reset the stream’s error flags.

End-of-file handling interacts with error handling, with the way the line is read (get() or getline() or a numeric read), and with the way the data file ends (with or without a newline character).

The file eofNumeric.in

```
Reading numbers.
0= 1:0:0:0: 1
0= 1:0:0:0: 278
0= 1:0:0:0: 45
4= 0:0:1:0: 4= 0:0:1:0: 4= 0:0:1:0: 0= 1:0:0:0: 8
0= 1:0:0:0: 7
6= 0:1:1:0: ----------------------------
```

```
IOstream flags are printed after each read, in order:
state = good() : failbit : eofbit : badbit : $input$
```
// C++ demo program for error handling while reading numbers.

// A. Fischer, June 2022, revised May 2023  file: IONumeric.cpp
#include "tools.hpp"

void useNums( istream& instr );

int main( int argc, char* argv[] ) {
    banner();
    // Command line must give name of the program followed by a file name.
    if (argc < 2) fatal( "Usage: eofDemo numberfile" );

    ifstream numStream( argv[1] );
    if (! numStream) fatal( string("Cannot open numeric file ") += argv[2] );

    cout << "
IOstream flags are printed after each read, in order:
" <<"state= good(): failbit: eofbit: badbit: $input$: $if good, output$";

    useNums( numStream );  // Tested also with use_numhex();
    bye();
}

void useNums( istream& instr ) {
    cout << "
Reading numbers.
";
    int number;
    for(;;) {
        instr >> number;
        cout << instr.rdstate() <<"= " <<instr.good() <<"->"
        <<"failbit: eofbit: badbit: $input$: $if good, output$";
        if (instr.good()) cout << number << endl;
        else if (instr.eof() ) break;
        else if (instr.fail()) { // Without these three lines
            instr.clear();  // an alphabetic character in the input
            instr.ignore(1);  // stream causes an infinite loop.
        }
        else if (instr.bad()) // Abort after an unrecoverable error.
            fatal( "Low-level error while reading input stream." );
    }
    cout << "-----------------------------------\n";
}

Notes on IONumeric().

- Line 18 calls the function that reads the file.
- As with text files, the eof and bad flags can go on. But with numbers, there is a new possibility: in attempting to read a number, number conversion might fail because of a non-numeric character in the stream. This will cause the fail flag to be turned on.
- The file eonumeric.in (below, on the left) has two fully correct lines followed by one line with non-numeric characters, then another correct line.
- Within the useNums function, Line 27 reads one thing into an int variable. Line 30 prints the input in both base 16 and base 10 if instr.good() is true.
- When >> is used for input and the target variable has a numeric type, the stream is read one character at a time until a non-numeric character is found. Then the prior chars are converted to a binary number and stored in the given variable.
- On the third line of input, the first two characters are read correctly, then reading stops when the ‘a’ comes in.
- The next time around the internal read loop, the system tries again to read the fourth character, ‘a’. A conversion error occurs because ‘a’ is not a base-10 digit, and the fail but is turned on. The eofbit is not turned on, so control passes to line 32, which detects failure caused by the conversion error.
3.7. Example 3: Interacting Classes and Formatted IO

- Lines 31–37 show how to deal with numeric inputs. The stream is not “good” – this could be caused by an eof or an error. Since eof is normal, we must check for that first, and possibly break out of the input loop.
- Lines 33 and 34 recover from this error. First, the stream’s error flags are cleared, putting the stream back into the good state. Then the offending character is read and discarded. Control then goes back to the top of the loop, and we try again to read a number.
- After the first error, we cleared only one keystroke from the stream, leaving the “bc”. So another read error occurs. In fact, we go through the error detection and recovery process three times, once for each non-numeric input character. On the fourth recovery attempt, a number is read, converted, and stored in the variable number, and the program prints it and goes on to normal completion. Compare the input file to the output: the “abc” just “disappeared”, but you can see there were three “failed” read attempts before the final 8 was read successfully.
- **Warning:** because of the complex logic required to detect and recover from format errors, we cannot use the eof test as part of a while loop. The only reasonable way to handle this logic is to use an “infinite” for(;;) loop and write the required tests as if statements in the body of the loop. Note the use of the if...break; please learn to write input loops this way.

3.6.1 Hexadecimal Input and Output.

**Handling hexadecimal data.** The line 45abc8 is a legitimate hexadecimal input, but not a correct base-10 input. Using the same program, we could read the same file correctly if we wrote lines 28–31 to use hex:

```cpp
instr >>hex >>number >>dec;
cout "<<instr.rdstate()" "<<instr.good() ""-""
  "<<instr.fail() ""-"" "<<instr.eof() ""-"" "<<instr.bad()""," 
if (instr.good()) cout "hex "number "dec "number "endl;
```

In this case, all input numbers would be interpreted as hexadecimal and the characters ”abcdef” would be legitimate digits.

The output shown below is given in both bases 16 and 40 to make comparisons easy. The results are:

```
Reading numbers.
0= 1->0:0:0. 1 1
0= 1->0:0:0. 278 632
0= 1->0:0:0. 45abc8 4565960
0= 1->0:0:0. 7 7
6= 0->1:1:0. --------------------------
```

Note: whenever you put an input or output stream into hex mode, it is important to put it back into dec mode before leaving the context.

3.7 Example 3: Interacting Classes and Formatted IO

This example continues to develop basic C++ file-handling tools and syntax. It also begin to build complexity by introducing classes that interact, or collaborate with each other to get the job done. Classes serve a variety of purposes in OO programming, and we try to define our classes so that they serve these purposes well. A class is...

- A type from which objects (instances) can be formed. We say the instances belong to the class.
- A collection of things that belong together; a struct with its associated functions.
- A way to encapsulate behavior: a public interface with a private implementation.
- A way to protect the integrity of data by providing the rest of the world with functions that permit a view of the data, and ways to use the data, but no way to modify it.
- A way to organize and automate allocation, initialization, and deallocation.
- Sometimes, a reusable module.
• A way to break a complex problem down into manageable, semi-independent pieces, each with a defined interface.
• An entity that can collaborate with other classes to perform a task.

Types of Classes  In designing an OO application, we must first identify the classes that we need, based on the problem specification. This is more of an art than a science and experience plays a large role in making good choices. For a beginner, it helps to have some idea of the kind of classes that others have defined in many kinds of OO applications. These include:

• Data classes: These represent physical objects or places: person, airplane, airport, document, floorPlan, receipt.
• Controller classes: these represent and carry out the logic of an application or a major subsystem of an application: game controller, database controller, web controller.
• Container classes: These are used by a controller to manage the objects in a data set.

Application architecture. A UML class diagram is used both to develop and document the architecture of an application.

In a fully-developed class diagram, a class is represented by a rectangle with three (or, optionally four) layers, as shown on the right:

- class name
- data members (type and name)
- function members (prototypes)
- other information related to this class: friends, an extension of <<, etc.

Members are marked with + for public, – for private, or # for protected.

In the initial stages of developing an application, most of this detail has not yet been developed. At first, a class is just a named rectangle with connections to other classes. Then data members are added, then methods, then other details.

In the Inventory application there is a brief main program and three classes: a controller class (inventory), a container class (vector<Part>), and a data class (Part). Techniques used include a stringstream, reading lines of mixed numeric and alphabetic input and formatting the output in a table.

![Diagram](Image)

Figure 3.2: A partially developed class diagram for Inventory.

Key to the UML Symbols. A typical main program for C++: brief and with one purpose.

• Main instantiates Inventory on line 10. This is symbolized the line from main to inventory. The white diamond marks main() as the creator of Inventory.
• Inventory has two private (-) data members. The function members are not listed here because they are not of interest in this context.
• One of the data members is an instance of the class `vector<Part>`. This is symbolized by the line connecting `Inventory` and `vector`. The line begins with a black diamond because the vector is a part of the `Inventory` and will be born and die when the `Inventory` is born and dies.

• The vector stores, but does not create, many instances of class `Part`. The asterisk means “zero to many”. The diamond is white because the vector does not create the Parts, it simply stores them.

**Notes on main()**

• This function is brief (according to style guidelines) and delegates all application-specific actions to the controller class.

• Line 2 includes the header file for the controller class, `Inventory`.

• Line 6 checks for a meaningful command line and gives the user information if it is not present.

• Lines 7 and 8 open and test the data file whose name is on the command line.

• Line 10 instantiates the controller class

• Lines 11 and 12 call on the controller to do the work and display the results.

• Line 13 closes the input file because we are done with it.

```cpp
#include "tools.hpp"
#include "inventory.hpp"

int main(int argc, char* argv[]) {
  banner();
  if (argc != 2) fatal( "Usage: eofDemo filename" );
  ifstream instr( argv[1] ); // argv[0] is the name of the program.
  if (!instr.is_open()) fatal( string("Cannot open text file ") += argv[1] );
  Inventory partList( instr );
  partList.readParts();
  fin.close();
  cout << partList;
  bye();
}
```

### 3.7.1 The `Inventory` class.

A “controller” class is in charge of the logic of the application. It instantiates one or more other classes to organize its data and get its task done. It is the only class that should interface with the main program.

• The tasks of reading and printing the data for a single part are delegated to the `Part` class. This class is concerned with organizing many Parts into an accessible data structure.

• Line 26: The first data member is the address of an input stream, which was opened in `main()` and will be passed to the `Inventory` class when its constructor is called. An open file must be passed by reference.

• Line 27: The second is a data structure to store the validated inputs. We use a `vector<Part>` because we do not know how many parts will be in the inventory.

• Line 29 is the entire definition of the constructor. It does one thing: initialize the `ifstream` declared on line 25 by using a `constructor initializer` or `ctor`. This is the only way to initialize a reference data member.

• Line 30 is an explicit definition of the destructor. The do-nothing default is appropriate because a `Part` does not have any dynamically allocated portions.

• Lines 32 and 33 are prototypes for the functions found in the class-implementation file.

• Line 35 is outside of the class, but belongs with it; it defines a new method for operator `<<`. This permits us to print the inventory using the `<<` operator.

• This is a global function method. It cannot access the private members of `Inventory`, but it can (and does) call the public `Inventory::print()` method.
class Inventory 
private:
  ifstream& fin;
  vector<Part> stock;
public:
  Inventory( ifstream& fileIn ) : fin(fileIn) {} // Inventory()= default;
  void readParts();
  ostream& print( ostream& fout );
};

inline ostream&
operator <<(ostream& out, Inventory& p){ return p.print(out); }

The Inventory implementation. The implementation file for Inventory contains the definitions of the two functions that were declared in the class header. Short functions can be fully defined in the header file. Longer functions should be in the implementation file. The prototypes for ALL functions must be in the header file because these are included by other modules and allow them to be linked together.

- The readParts() function is long because it realizes the application’s data, that is, it uses stream input to bring the data from a file into memory and it stores that data in a structured form. When the checks are made in this order, all good data is captured whether or not the file ends in a newline character.

- Over half of the lines in the read loop are involved with checking for and handling stream errors. The loop ends on line 55 when the end-of-file is encountered.
The read loop.

- Line 48 reads one complete line from the file into a local temporary, buf. Line 49 echo-prints it. A properly formatted data file has the information for one Part on each line.

- Reading a line may set error flags. Echoing the input helps the programmer/user to interpret any error comments that might appear. Lines 49 and 50–59 check for and handle errors. This logic is relatively simple because validating the data will be done in the Part constructor.

- It is possible to have input and an end-of-file. The call on fail() will return true if either the fail flag or the bad flag is set. So lines 51–54 will be executed whether or not an end-of-file happened.

- Lines 51–54 handle input lines that did not cause error flags to be set. The input buffer is sent to the Part constructor for further analysis. Blank lines and malformed lines will be screened out by the processing in the Part class.

- Line 52: If the input is all OK, the newly-constructed part is pushed onto the end of the stock vector.

- Lines 55–59 handle the error flags. This is relatively simple because getline() does not cause conversion errors. Those will be handled in the Part class.

Printing the inventory array. [Lines 65...70]

- A for-each loop is the easiest way to print the contents of any standard template data structure.

- This print function returns an ostream& so that it can be easily used in a definition of a new method for operator << (line 34) which is called from main() on line 12.

- Line 66 uses the operator << method defined in the Part class.

3.7.2 The Part class

```
70 //-------------------------------------------------------------------------------
71 // Header file for hardware store parts. part.hpp
72 // Created by Alice Fischer on Mon June 7, 2023.
73 //-------------------------------------------------------------------------------
74 #include "tools.hpp"
75 //-------------------------------------------------------------------------------
76 class Part {
77 private:
78   string partName = "";
79   int storeCode = 0;
80   int quantity = 0;
81   float price = 0;
82 public:
83   Part( string buf );
84   ~Part() =default;
85   //-------------------------------------------------------------------------------
86   float getPrice(){ return price; }
87   ostream& print( ostream& fout );
88   inline ostream&
89   operator <<(ostream& out, Part& p){ return p.print(out); }
```

Notes on the Part header file (Lines 70...88)

- There are four private data members, all of which are initialized here to default values. In the Part constructor, they will be reset to data from the input buffer. The constructor (line 83 and 94–99) will parse the line in the buffer and initialize the data members.

- Line 84 defines a do-nothing destructor because this class contains no dynamically allocated memory.

- Line 88 defines a new method for the output operator. As always, this is a global method that calls the public print function in the Part class.
Notes on the Part implementation  (Lines 90...109)

• The Part constructor uses a stringstream and its built-in number-conversion functionality. The input was read as a string, which is stored in buf. Parsing and number conversion is a picky and error-prone job, so DO NOT try to do it yourself. Use a stringstream and the built-in tools.

• Line 96 “wraps:” a stream around the existing string, buf. This lets us use the normal input operators to do parsing and conversion.

• The first thing on a line of data is the part name, which must end in a comma. Line 97 reads the chars from buf up to the first comma, and stores the string in the data member named partName. (It also reads and discards the comma.)

• Line 98 reads and stores the three numbers that follow the comma.

• Line 99: When doing a numeric read, it is important to check the stream status. Here, we are reading from a string stream that is a local temporary in this function. It will be deallocated on line 100, so we do not need to clear the erroneous keystroke from the stream or clear the stream flags. Using a stringstream to parse numeric data has real advantages and leads to simpler code. However, we DO need to return an error indicator to the Inventory class, and that is done by using a reference parameter.

The print() function. We print one data member per line here, nicely formatted in a table by controlling the field width of each part of the line. The code is laid out in easy-to-read columns.

• The string member and the store code are left justified, the other parts are right justified. Justification remains set until it is changed.

• The fill character is set to a dot for the part name, space for the other data members.

• The field width is set to the programmer’s best estimate of how much space is needed. Too much space damages readability. Too little space results in truncation of the last part of the part name.

• Integers are easier to print because they have only one part. When printing a float, however, more things need to be specified: the overall format (scientific notation or fixed decimal notation) and the desired precision should be given. With fixed notation, the precision specifies the number of places after the decimal that will be displayed.
3.7.3 Testing the Code

Two input files. In the input file, each data set should start with a part description, terminated by a comma and followed by two integers and a price. A program should not depend on the number of spaces before or after each numeric field. It is trick to read a file correctly if you do not know whether or not it ends in a newline character.

On the left, below is a fully-correct input file used for testing. Please note that it does end in a newline character. On the right is a file that is full of errors. Please note that it does not end in a newline character.

<table>
<thead>
<tr>
<th>The file parts1.txt</th>
<th>The file parts2.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>claw hammer, 3 28 9.99</td>
<td>claw hammer, 3 28 9.99</td>
</tr>
<tr>
<td>claw hammer, 5 3 8.95</td>
<td>claw hammer, 5 3</td>
</tr>
<tr>
<td>long nosed pliers, 5 57 2.38</td>
<td>long nosed pliers, 5 57 2.38</td>
</tr>
<tr>
<td>pliers, 3 31 2.38</td>
<td></td>
</tr>
<tr>
<td>roofing nails-1 lb, 5 15 1.59</td>
<td>roofing nails-1 lb, 5 15 1.59</td>
</tr>
</tbody>
</table>

Output from parts1.txt. Using the given input file, the output looks like this:

Reading inventory from parts input file.
claw hammer, 3 28 9.99
claw hammer, 5 3 8.95
long nosed pliers, 5 57 2.38
pliers, 3 31 2.38
roofing nails-1 lb, 5 15 1.59

| Partname................. Store Quant Price |
|-------------------------|---------------------|
| claw hammer.............. 3 28 9.99        |
| claw hammer.............. 5 3 8.95         |
| long nosed pliers........ 5 57 2.38       |
| pliers................... 3 31 2.38       |
| roofing nails-1 lb....... 5 15 1.59       |

An input file with errors: parts2.txt. This file has four lines with problems:
* The third line is incomplete and produces an error comment.
* The fifth line is blank.
* The sixth line contains only a period, not real data.
* The last line does NOT end in a newline character.

Output from parts2.txt.

Reading inventory from parts input file.
claw hammer, 3 28 9.99
claw hammer, 5 3
   Not a good part: "claw hammer, 5 3 "
   31 2.38
   Not a good part: " 31 2.38 "
   roofing nails-1 lb, 5 15 1.59
      Not a good part: ""
     Not a good part: ".
   long nosed pliers, 5 57 2.38

| Partname................. Store Quant Price |
|-------------------------|---------------------|
| claw hammer.............. 3 28 9.99        |
| roofing nails-1 lb....... 5 15 1.59       |
| long nosed pliers........ 5 57 2.38       |
Files with no data.

The result of running the program on an empty file is not ideal, but it is also not misleading:

Reading inventory from parts input file.

<table>
<thead>
<tr>
<th>PartName</th>
<th>Store Quant</th>
<th>Price</th>
</tr>
</thead>
</table>

Normal termination.
Chapter 4: Functions and Parameter Passing

It is logically impossible to have a language that is simple, powerful, and flexible. C++ is powerful and flexible.
— A. Fischer, reflecting on 25 years of teaching this language.

Design principles. In this chapter, we present some two major topics: function declarations vs. function definitions, inline functions, the various ways that a parameter may be passed in C++, and the resulting difference in the way functions are called. Throughout, please keep in mind these fundamental principles of OO design:

- **Simpler is better**, if it does the job.
- **A class takes care of itself:** It keeps its data private and it’s only public members are the functions that define a safe public interface.
- **What you SHOULD do with classes is a small subset of what you CAN do:** This applies to the many ways that parameters can be passed.

4.1 Operators, Functions, and Methods

Operators are functions. In C and Java, functions and operators are two different kinds of things. In C++, they are much the same. All operators are functions, and can be written in either traditional infix operator notation or traditional function call notation, with a parenthesized argument list. Thus, we can add \( a \) to \( b \) in two ways:

\[
\begin{align*}
    c &= a + b; \\
    c &= \text{operator } + (a, b);
\end{align*}
\]

Functions and methods. In C, a function has a name and exactly one definition. In C++, a function has a name and one or more definitions. Each definition is called a *method* of the function, and a function is a collection of methods. For example, the `close()` function has a method for input streams and a method for output streams. The two methods perform the same function on different types of objects.

Similarly, `operator+` is predefined on types `int` and `double`, and you may add methods to this function for numeric types you create yourself, such as `complex`. Part of the attraction of OO languages is that they support *generic programming*: the same function can be defined for many types, with slightly different meanings, and the appropriate meaning will be used each time the function is called.

At compile time\(^1\), the C++ system must select which method to use to execute each function call. Selection is made by pattern matching: the compiler looks for a method that is appropriate for the type of the arguments in each function call. If exactly one is found, it is used. If zero or more than one exist, it is a compile-time error. This process is called *dispatching* the call.

Global functions. Functions can be defined globally (outside all of the classes) or as class members\(^2\).

The main function is always defined globally. With few exceptions, the only other global functions should be those that interact with predefined C or C++ libraries or with the operating system. Common examples of global functions include those used in combination with the standard sort() function and extensions to the C++ output operator\(^3\). The `operator <<` extensions form a special case. Each new method is directly associated with a class, but cannot be inside the class because standard `operator <<` is a global function. Another case is the `getline()` function for strings; it is a global function because it is a stream function, not a string function.

---

\(^1\)Polymorphic functions are dispatched at run time.
\(^2\)Almost all C++ methods are member functions.
\(^3\)These cases will be explained later.
Non-class functions. Functions can be defined inside or outside classes in C++. For those defined outside any class, the syntax for definitions and calls is the same as in C. Non-class functions in C++ are used primarily to modularize the main program, to extend the input and output operators, and to permit use of certain C library functions.

Member functions. Functions defined or prototyped within a class declaration are called member functions. A member method can freely read from or assign to the private class members, and the member methods are the only functions that should accesses class data members directly. If a member method is intended to be called from the outside world, it should be a public member. If the method’s purpose is an internal task, it should be a private member.

Taken together, the public methods of a class are called the class interface. They define the services provided by the class for a client program and the ways a client can access private members of the class.

Each function method has a short name and a full name. For example, suppose Point is a class, there is a method for plot() in that class. The full name of the method is Point::plot; the short name is just plot. Normally, we use the name of the function (the short name) in function calls.

4.2 Comparisons to C

The implied argument. In C, all arguments are written inside the parentheses. In contrast, in C++, every call to a class function has an “implied argument”, which is the object whose name is written before the function name. The rest of the arguments (if any) are written inside the parentheses that follow the function name, as they would be in C. For example:

```c
if ( instr.eof() ) ... 
if ( inWord.compare( searchKey )) ... 
```

The implied argument is a pointer to the object used to call the function. It is called this.

Number of parameters. Function calls in C++ tend to be shorter and easier to write than in C. In C, functions often have several parameters, and functions with no parameters are rare.

Functions with empty or very short parameter lists are common in C++. First, the implied argument itself is not written as part of the argument list. Also, classes are used to group together related data, and that data is available for use by class functions. We often eliminate parameters by using class data members to store information that is created by one class function and used by others.

Code organization. When writing in a procedural language, many programmers think sequentially: do this first, this second, and so on. Function definitions often reflect this pattern of thought, so in C, functions tend to be organized by the sequence in which things should be done.

In an object-oriented language, the code is not arranged sequentially. Thus, in C++, functions are part of the class of the objects they work on. Control passes constantly into and out of classes; consecutive actions are likely to be in separate functions, in separate classes with a part-whole relationship. One object passes control to the class of one of its components, which does part of the required action and passes on the rest of the responsibility to one of its own components.

The number and length of functions. There are many more functions in a C++ program than in a C program and C++ functions tend to be very short. Each C++ function operates on a class object. The actions it performs directly should all be related to that object and to the purpose of that class within the application. Actions related to class components are delegated (passed on) to the components to execute. Thus, long functions with nested control structures are unusual, while one-line functions and one-line loops are common. Inline functions are used to improve the efficiency of this scheme.

4Within a virtual function, se use the full name. However, this does not arise until derived classes are introduced.
4.3 Declarations, Definitions. and Inline Functions

Calling functions. A member function can be called either with a class object or with a pointer to a class object. Both kinds of calls are common and useful in C++. To call a member function with an object, write the name of the object followed by a dot followed by the function name and an appropriate list of arguments. To call a member function with a pointer to an object, write the name of the pointer followed by an -> followed by the function name and an appropriate list of arguments. Using the Point class again, we could create Points and call the plot() function two ways:

```cpp
Point p1(1, 0); // Allocate and initialize a point on the stack.
p1.plot();
Point* pp = new Point(0, 1); // Dynamically create an initialized point.
pp->plot();
```

In both cases, the declared type of p1 or pp supplies context for the function call, so we know we are calling the plot function in the Point class.

Static member functions. An object or pointer to an object is needed to call an ordinary member function. However, sometimes it is useful to be able to call a class function before the first object of that class is created. This can be done by making the method static. A static method can only use static data, that is, data that was allocated and initialized at program-load time. To call a static member function, we use the class name:

```cpp
static bool validate( xInput, yInput );
bool dataOK = Point::validate( xInput, yInput );
```

A declaration for a method is a prototype. It is written in a header file because this information must be known to (included by) other classes that use the method. A definition of a method supplies the code for the method.

If the declaration and definition are both in the header file, the definition is inline. For short methods, this can improve both space efficiency and run-time efficiency. If the definition is in the implementation file, the method is not inline. This

4.3.1 Inline vs. Out-of-line Function Translation.

C++ permits the programmer to choose whether each function will be compiled out-of-line (the normal way) or inline. An out-of-line function is compiled once per application. No matter how many times the function is called, its code exists only once.

Out-of-line functions. When a call on an out-of-line function is translated, the compiler generates code that enables the linker/loader to connect the function call to the function definition. At run time, when the function call is executed, a stack frame is built, the argument values are copied into it, and control is transferred from the caller to the function code. After the last line of the function, control and a return value are passed back to the caller and the function’s stack frame is deallocated (popped off the system run-time stack). This process is done efficiently but still takes non-zero time, plus memory space for the stack frame, and space for the machine instructions that perform the calling sequence.

For a one-line function, the compiled code is probably shorter than the code required to implement an ordinary function call. Short functions should be inline because it always saves time and can even save space if there are only a few machine instructions in the function’s definition. However, if the function is not short and it is called more than once, inline expansion will save time but it may lead to a longer object file.

Inline function translation. When an inline function is compiled, the arguments are substituted into the function code and the entire code is copied code of the caller. The expanded code replaces the function call. If the same function is called many times, many copies of its code will be made.

Inline expansion always saves execution time because no stack frame is built at run time and no time is spent jumping to and back from the function code. Depending on the length of the function code, it might either save memory space or consume more memory.

---

5 A static data member is called a class variable because there is only one copy of it per class. All instances of the class share that same static variable.
For this reason, both time and space efficiency are saved when 1-line functions are expanded inline. However, if the function is not short and it is called more than once, inline expansion will save time but it will also lead to a longer object file.

In C++ a program often has many 1-line functions that are called frequently, so using inline expansion can significantly improve performance.

Macros. A macro is a short piece of code, often with parameters, that a program \texttt{\#defines}. In C, macros are widely used to achieve inline execution of brief functions. (C did not have inline functions.) Macros still exist in the C++ language but are rarely used because inline functions have cleaner semantics and do the job macros used to do.

Macro expansion is much like inline-function expansion except for these things

- The macro parameters are untyped and no type-checking is done during expansion. In contrast, inline-function parameters are typed and the types are checked by the compiler. The number and types of the arg
- Macros are handled by a preprocessor; macro calls are identified by a preprocessor very early in translation and replaced by the definition to make compilable code. All this is done before parsing and interpretation begins. The expansion process is like the search-and-replace process in a text editor. In contrast, functions (inline or not) are compiled later and go through all the phases of translation.

Usage of inline. It is helpful to be able to see an entire class declaration at once. If a class has some long functions, we put them in a separate file so that the rest of the class will fit on one computer screen. For this reason, any function that is longer than one or two lines is generally given only a prototype within the class declaration and is fully defined in the \texttt{.hpp} file.

The main reason for using inline definitions is that it always increases runtime efficiency. The reasons for not using an inline function are:

- A recursive function cannot be expanded inline.
- Code size: Long functions called from 2 or more places in the program should not be inline.

4.3.2 What is Inline?

The \texttt{inline} keyword. A function defined fully within the class declaration is automatically inline and the keyword \texttt{inline} is not used. In addition, you can declare a function prototype to be \texttt{inline}, and provide the rest of the function definition in the \texttt{.hpp} file after the brace that closes the class definition.

Summary of inline rules and concepts.

- An inline function is expanded like a macro, in place of the function call. No stack frame is built and there is not jump-to-subroutine.
- Methods defined fully within a class (between the class name and the closing curly brace) default to inline. \textbf{Do not use the inline keyword inside a class}
- In order to compile, the full definition of an inline function must be available to the compiler when every call to the function is compiled. That means it \textbf{must} be in the header file.
- The compiler treats “inline” as advice, not as orders; if a particular compiler thinks it is inappropriate in a particular situation, the code will be compile out-of-line anyway.
- Methods for global functions can be inline. To make extensions of operator \texttt{<<} and operator \texttt{>>} inline, put them at the end of the \texttt{.hpp} file. Normally, these functions are implemented by calling the public \texttt{print()} method within the class.
4.4 Parameter Passing

C supports two ways to pass non-array parameters to a function: call-by-value and call-by-pointer. Arrays are passed by reference. C++ supports all three parameter-passing mechanisms for any parameter. The purpose of this section is to demonstrate how the three parameter-passing mechanisms work and to help you understand which to use, when, why, and how.

The three functions given below do not compute anything sensible or useful; their purpose is to illustrate how the three parameter-passing mechanisms are similar and/or different, and how to use each one. Each function computes the average of its two parameters, then prints the parameter values and the average. Finally, both parameters are incremented. The computation works the same way in all versions. However, the increment operator does not work uniformly: sometimes a pointer is incremented, sometimes an integer; sometimes the change is local, sometimes not.

- Call-by-value. The argument values are copied into the parameter storage locations in the function’s stack frame. There is no way for the function to change values in main’s stack frame. All changes in the parameters are strictly local.

- Call by pointer. The argument must be an l-value or a pointer to a variable that is accessible to the caller. These addresses are stored in the parameter variables in the function’s stack frame. They permit the function to read and modify its caller’s memory locations. If the pointer, itself, is incremented, the effect is purely local.

- Call by reference. The argument must be a variable accessible to the caller. The L-value of that variable is stored in the & parameter in the function’s stack frame. This permits the function to read and modify its caller’s memory locations.

Call by R-valued reference is an important but specialized way to pass some parameters. It will be explained in a later chapter.

4.4.1 The Three Odd Functions

```cpp
//-------------------------------------------------------------------
// Three odd average functions. file: odds.hpp
//-------------------------------------------------------------------
#ifndef ODDS
#define ODDS
#include <iostream>
using namespace std;

void odd1( int aa, int bb ); // Call by value.
void odd2( int* aa, int* bb ); // Call by address or pointer.
void odd4( int& aa, int& bb ); // Call by reference
#endif
```

4.4.2 Calling The Odd Functions

This main program calls the three odd functions with arguments selected from the array named `ar`. It prints the whole array before and after each call so that you can see any changes that take place. Each call (and its results) is explained in the paragraphs that follow and stack diagrams are given to help you understand how the system works.

```cpp
// Instructional targets:
// 1. Call by value makes a copy of the argument in parameter variable.
// 2. Call by reference stores the address in the parameter variable. It is denoted by an & after the type of the parameter. The argument looks the same as call-by-value.
// 3. Call-by-pointer has a pointer argument or an ampersand after its type.
```

// The parameter must be a pointer type; the address of the argument is stored in the parameter.
// 4. Each style is preferred or used for some purposes.
// 5. The name of an array is translated as a pointer to its slot 0.
//-----------------------------------------------------------------------
// Calling the odd average functions. file: oddM.cpp
//-----------------------------------------------------------------------
#include "odds.hpp" // Contains the three odd functions.

void print( int* ap, int n ) { // Print the n values in an array.
    for( int k=0; k<n; ++k ) cout << " [" << k << "] = " << ap[k] ;
    cout << endl;
}

int main( void ) {
    int ar[6] = {11, 22, 33, 44, 55, 66};
    int* ip = ar; // Set a pointer to beginning of an array.
    int* iq = &ar[2]; // Set a pointer to 3rd item of array.
    cout << "Initially, ar is: ------------------------------------
";
    print( ar, 6 );
    odd1( ar[0], *iq ); print( ar, 6 ); // See Figure 5.1.
    odd2( ip, iq ); print( ar, 6 ); // See Figure 5.2.
    odd2( &ar[1], &ar[4] ); print( ar, 6 ); // See Figure 5.3.
    odd4( ar[2], *(iq+2) ); print( ar, 6 ); // See Figure 5.4.
}

// In the output below, note what causes an argument to change.
/*
Initially, ar is: -------------------------------------
In odd1, average of 11, 33 = 22
In odd2, average of 11, 33 = 22
In odd2, average of 22, 55 = 38
In odd4, average of 34, 56 = 45
*/

The output.

Initially, ar is: -------------------------------------
In odd1, average of 11, 33 = 22
In odd2, average of 11, 33 = 22
In odd2, average of 22, 55 = 38
In odd4, average of 34, 56 = 45
4.4.3 Parameter Passing Illustrated

In these diagrams, a round-cornered box represents a stack frame or activation record on the system’s run-time stack. A stack frame is the data structure that is built by the run-time system to implement a function call. Within the box, the dot-dashed line divides the local variables (above it) from the parameters (if any, below it). The leftmost column gives the name of the function. The middle two columns supply the type, name, and address of each object allocated for the function. The rightmost column lists the current value, which is a literal number, a pointer (arrow with one stem), or a binding (arrow with a double stem).

**Odd1: Call-by-value.** Call-by-value is like making a Xerox copy: the value of the argument in the call is copied into the parameter variable. This parameter passing method isolates the function from the caller; all references to the parameter are kept local; the function can neither see nor change the caller's variable. If the function increments the parameter variable or assigns a new value to it, the change is purely local and does not affect variables belonging to the caller.

```
void odd1( int aa, int bb ){ // Make copies of the argument values.
  int anser = (aa + bb) / 2; // See diagram 1.
  cout <<"\nIn odd1, average of " <<aa <<", " <<bb <<" = " <<anser <<end1;
  ++aa; ++bb; // Increment the two local integers.
}
```

The call: `odd1( ar[0], *iq );`

![Diagram of Odd1](image)

In this example, `aa` is a call-by-value parameter, so executing `++aa`, changes its value locally but does not affect `main`'s variable `ar[0]`. After the function returns, `ar[0]` still contains 11.

**Odd2: Call-by-value with a pointer parameter.** Even in a call-by-value language like C, an argument that is a pointer can be used with a pointer parameter to give the function access to the underlying variable in the stack frame of the caller. If a parameter is declared with a pointer type, the argument can be either a pointer variable or the address of any variable.

```
void odd2( int* aa, int* bb ){ // Call by address or pointer.
  int anser = (*aa + *bb) / 2; // See diagram 2.
  cout << "\nIn odd2, average of " <<*aa <<", " <<*bb <<" = " <<anser <<end1;
  ++aa; // Increment the local pointer
  ***bb; // increment main's integer indirectly
}
```

The pointer parameter can be used two ways within the function. For example, suppose the parameter was declared to be `int* param`. Then writing `++param` will change the address stored locally in `++param` and make it point to the next address. But if you write `++*param`, the variable that `param` points at in the caller will be incremented.

The call: `odd2( ip, iq );`
2a. Call-by-value with a pointer parameter and a pointer argument. The first call on odd2 is diagrammed in Figure 4.2, above. The arguments are the values of two pointers. When these arguments are stored in the parameter locations, the parameters point at the same locations as the pointers in main(). Writing ++aa increments the local value of the pointer aa, which changes the array slot that the parameter points to (initially it was ar[1], after the increment it is ar[2]). Writing +++bb changes the integer in the caller’s array slot.

2b. Call-by-pointer with a pointer parameter and an & argument.
The call: odd2( &ar[1], &ar[4] );

The second call on odd2 is shown in Figure 4.3. The effect is the same as the prior call; The arguments are the addresses of two slots in caller’s array and are diagrammed as simple arrows. Writing ++aa changes the array slot that the parameter points to (initially it was ar[1], after the increment it is ar[2]). Writing +++bb changes the integer in the caller’s array slot.

Call-by-reference (&). In call-by-reference (Figure 4.4), the parameter value is called a “binding”. It is immutable and is diagrammed below as a shaded double arrow.

```
void odd4( int& aa, int& bb ) // Call by reference
{
    int ansr = (aa + bb) / 2; // See diagram 4.
    cout << "In odd4, average of " <<aa <<", " <<bb <<" = " <<ansr <<endl;
    ++aa; ++bb; // increment two integers in main.
}
```

The call: odd4( ar[2], *(iq+2) );
The value stored locally in the parameter cannot be changed because the parameter name becomes transparent: it is just an alias, or a synonym, for the caller’s variable. The program cannot read or change the
address stored in the parameter. For example, in line 15, using ++ changes the caller's value, not the address stored locally in the parameter.

**Call-by-reference vs. call-by-value.** When call-by-reference is used, the parameter name becomes an alias for the argument variable, and any use of the parameter name in the code is translated as a “fetch-indirect” machine instruction. There are several reasons for using a reference parameter instead of call-by-value, which makes a copy.

- Sometimes you need to modify the caller’s object, which is not possible when call-by-value is used. For example, when passing an open stream as a parameter, the purpose is always to modify that stream (extract input or add output). The reference parameter lets you do this.

- If the argument variable is a large object, copying it takes both time and space. Passing a reference instead saves that space with a minor additional cost for the indirect reference each time the argument is used.

- Call-by-reference should be used when the argument is or might be an object with a dynamically allocated part. This avoids copying the argument and, later when the function returns, a double-delete problem. (This topic will be covered in detail later.)

**Call-by-reference vs. call-by-pointer.** When call by pointer is used, the function can dereference the parameter and assign a new value to the argument variable in caller’s stack frame. Or, the function can change the address stored in the parameter and make it refer to a different variable.

In contrast, when call-by-reference is used, there is no need and no ability to dereference the parameter because the parameter name is an alias for the argument variable. It is not possible to change the address in the local stack frame. Thus, call-by-reference is more restrictive and, therefore, better for any job where both techniques will suffice. Preventing meaningless changes in memory is important for security.

### 4.5 Function Returns

#### 4.5.1 L-values and R-values.

An *L-value* is an address, or location where something can be stored. It can be used to receive a value in an assignment operator; you can apply increment and decrement to it. An *R-value* is a value that can only be used on the right side of an assignment operator; it is the value that will be stored in an L-value. Figure 4.5, below, illustrates that an L-value is a storage location and an R-value is the contents of a storage location.

```plaintext
int k = 3; // L-value 1020, R-value 3
int* ptr = &k; // L-value 10FC, R-value 1020
```

**Conceptually:**

```
ptr  k
10FC 1020
```

**Physically:**

```
1020 3
```

Figure 4.5: L-values and R-values.
CHAPTER 4. FUNCTIONS AND PARAMETER PASSING

It is not that simple, really! Textbooks often say that the L-values go on the left side of an assignment operator and the R-values go on the right. But that is an oversimplification; it depends on context. Complexity arises in three ways:

- A pointer variable can be dereferenced to get a pointer R-value, which can be dereferenced again. If a function returns a pointer-value, the function’s result must be dereferenced before it can be used on the left side of =. In the example below, ret1() and ret2() return pointer R-values that can be dereferenced to get L-values.

- When a value is assigned to one slot of an array, an R-value integer is used to subscript that array. The array name and subscript are on the left side of the assignment, even though the subscript is treated as an R-value.

- Any L-value can be used where an R-value is expected and needed. The compiler automatically fetches the value out of the variable and uses it.

- A non-const reference is an L-value. When used as an R-value (on the right side of an assignment) a reference is treated the same way that a variable name would be treated in that context: the value at that address is fetched from memory.

```c
int j, k=3;
int& ref = &k;
int j = ref + 1;  // j will contain 4.
```

4.5.2 Using L-values.

Zombies. A serious problem will occur if a function returns the address of or a pointer to one of its local variables. At return time, the function’s local variables will be deallocated, so the return-value will refer to an object that no longer exists. The resulting problems may occur immediately or may be delayed, but this is always bad and often hard to track down.

Violating privacy. The following five functions all return a different flavor of int. In the code example, the function ret0() returns an integer R-value, which we sometimes “safe” and conforms to OO principles, since the function is not giving the caller access to anything that is not already accessible to it.

Two of the functions, ret2() and ret4(), return results that provide read-only access to the array: ret2 returns a const int* and ret4 returns a const int&. These are safe, also.

```c
//---------------------------------------------------------------------------------
// Return by value, reference, and pointer, and how they differ.
// A. Fischer Apr 19, 1998 file: returns.cpp
//---------------------------------------------------------------------------------
int ret0( int* ar ) { return ar[0]; }  // Returns an integer R-value.
int* ret1( int* ar ) { return &ar[1]; } // A pointer R-value.
const int* ret2( int* ar ) { return &ar[2]; } // A read-only pointer R-value.
int& ret3( int* ar ) { return ar[3]; }  // An L-value (reference).
const int& ret4( int* ar ) { return ar[4]; } // A read-only L-value reference.
```

However, the two other functions, ret1() and ret3(), return int* and int&. These functions can break encapsulation because they give the caller a way to change a private variable belonging a different class. Normally, functions should not be permitted to do this. For this reason, returning non-const addresses and pointers is avoided whenever possible in C++.

For example, suppose the caller is a function in class A, the the function called is in class B. Then if the function in class B returned a non-const reference to a private member, class A could change the value of a private class member to anything! Normally, functions i should not be permitted to change the value of private members other classes. All changes to class members should be made by functions in the same class that can ensure that the overall state of the class object is always valid and consistent.

\[\text{There are a few exceptions, for example, when the subscript function is extended for new kinds of arrays, it may return a non-const reference because that is the entire purpose of subscript.}\]
4.5. FUNCTION RETURNS

4.5.3 Using Return Values in Assignment Statements

The following program calls each of the \texttt{ret()} functions and uses its return value to select one integer from array named \texttt{ar}. This integer is saved in a variable and later printed. Four of the functions return the location of one slot in the \texttt{ar} array. Two of these return values are non-const and we use them to modify the array.

```cpp
// Instructional targets:
// 1. The meaning of a const return value.
// 2. A function can and often does return a storage location.
// 3. The difference in syntax between a pointer and a reference.
// 4. By using const, it is possible to protect the caller's variable
// 5. A pointer that points at an array can be subscripted.
//-----------------------------------------------------------------------
// Calling functions with the five kinds of function-return values.
// A. Fischer Apr 19, 1998 file: retM.cpp
//-----------------------------------------------------------------------
#include <iostream>
using namespace std;
#include "returns.hpp"

#define DUMPv(k) "\n" "#k " @ " <<&k <<" value = " <<k
#define DUMPp(p) "\n" "#p " @ " <<&p <<" value = " <<p " " #p " --> " << dec << *p
//-----------------------------------------------------------------------
void print5( int* ap ) { // Print the five values in an array.
  for( int k=0; k<5; ++k ) cout << "[" << k << "] = " << ap[k] ;
}

int main( void ) {
  int ar[5] = {11, 22, 33, 44, 55};
  int h, j, k, m, n;
  int *p = ar;

  cout << "Initially, variables are: ------------------------------\n";
  print5( ar );
  cout <<DUMPp(p) <<endl;

  h = ret0( ar ); // Answer should be 11
  j = *ret1( ar ); // Answer should be 22
  k = *ret2( ar ); // Answer should be 33
  m = ret3( ar ); // Answer should be 44
  n = ret4( ar ); // Answer should be 55
  cout <<DUMPp(p) <<DUMPv(h) <<DUMPv(j) <<DUMPv(k) <<DUMPv(m) <<DUMPv(n) <<endl1;
  p = ret1( ar ); // Answer should be a pointer to ar[1].
  *ret1( ar ) = 17; // Storing through the pointer.
  //ret2( ar ) = 17; // Illegal: ret2 returns a const int*.
  ret3( ar ) = -2; // Assigning to the reference.
  //ret4( ar ) = -2; // Illegal: ret4 returns a const int&.

  cout << "\nAfter return from functions variables are: -----------\n";
  print5( ar );
  cout <<DUMPp(p) <<endl;
}
```

The Dump macros. A macro lets us define a shorthand notation for any string of characters. We use macros when the same thing must be written several times, and that thing is long and error-prone to write. In this program, we want to be able to print out information that will be useful for understanding the program. We define one macro, \texttt{DUMPv} that will print the address and contents of a variable, and a second, \texttt{DUMPp}, that prints the address of a pointer variable, the address stored in it, and the value of its referent.

Remember that macro parameters are not typed, and macro calls are not type checked. Thus, the macro defined here can be used to dump any object for which \texttt{<<} is defined.
The output is:

Initially, variables are: ------------------------------

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>44</td>
</tr>
<tr>
<td>4</td>
<td>55</td>
</tr>
</tbody>
</table>

p @ 0xbffffd40 value = 0xbffffd18 p --> 11
h @ 0xbffffd2c value = 11
j @ 0xbffffd30 value = 22
k @ 0xbffffd34 value = 33
m @ 0xbffffd38 value = 44
n @ 0xbffffd3c value = 55

After return from functions variables are: -------------

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>-2</td>
</tr>
<tr>
<td>4</td>
<td>55</td>
</tr>
</tbody>
</table>

p @ 0xbffffd40 value = 0xbffffd1c p --> 17

The main function. We define an array, initialize it to easily-recognized values, and print it before and after calling the ret functions. By comparing the lines at the beginning and end of the output, you can verify that lines 45 and 47 changed the contents of the array. Calls on the DUMP macros in lines 35, 42, and 50 allow you to trace the effects of each function call.

Return by value. This is the commonest and safest kind of function return; the returned value is an R-value, (an integer, not an address. Among the functions above, ret0() returns an integer by value. The sample program calls ret0() (line 37) and assigns the result to an integer variable.

Return by pointer. The return value of ret1() is a non-const pointer R-value (the address of ar[1]) that can be stored in a pointer variable or dereferenced to get an L-value. It is called three times:

- On line 38, the result is explicitly dereferenced to get an integer L-value, then implicitly dereferenced again to get an integer R-value, which is stored in the variable j.
  
  j = *ret1( ar ); // Copy the referent of the return value.

- On line 44, the result is implicitly dereferenced to get a pointer R-value, which is stored in the pointer variable p.
  
  p = ret1( ar );

- On line 45, the result is on the left side of an assignment. It is explicitly dereferenced to get an integer L-value, which is used to store an integer constant.
  
  *ret1( ar ) = 17; // Store through pointer into referent.

Return by const pointer. The return value is an address that provides read-only access to the underlying variable. The function result cannot be incremented and it cannot be used on the left side of an assignment to change the underlying variable. In the sample program, ret2() is called once (line 39). Its result is dereferenced to get an integer L-value, which is automatically dereferenced again because it is on the right side of an assignment statement. The result is an int R-value, which is stored in k:

  k = *ret2( ar ); // Copy referent of return value into k.

Return by reference. A reference can be used in the same ways that a variable name is used. The function ret3() returns an integer reference: an integer L-value that can be used (without *) on either side of an assignment operator. ret3() is called twice and used both as an R-value (line 40) and as an L-value (line 46):

  m = ret3( ar ); // Coerce function result to R-value and store.
  ret3( ar ) = -2; // Store -2 in address returned by function.

Return by const reference. The function ret4() returns a const int& (a reference to a const int). This result can be only be used for read-access, because the underlying variable is a constant. ret4() is called once (line 41) and used with implicit dereference on the right side of an assignment statement:

  n = ret4( ar ); // Copy referent of return value into n.
Illegal calls. The following lines won’t compile. For both, my compiler gives the error comment assignment of read-only location, because the result of the function is a const L-value.

    *ret2( ar ) = -5;    // return by const *
    ret4( ar ) = 25;     // return by const &

4.6 Code Example: Are the Bracket Nested Properly?

In this section, we present a program to analyze a text file and determine whether its bracketing symbols are correctly nested, matched, and balanced. The demo provides an example of how a C++ program should be organized.

- The main function is in its own separate file.
- Brackets is a controller class: it implements the logic of the application.
- Stack is a container class used as a tool by the controller class to store Tokens while it checks the nesting.
- Token is a data class, used to store one opening bracket and its properties.

Anatomy of the application. The Brackets class contains the code that analyzes an input text to find mismatched brackets. Good design practices call for the separation of functionality, and for each class to do one distinct part of the whole. Here, we use three user-defined modules in addition to main().

1. The command line arguments are sent by the OS to main(), when the program is started up. Handling them is (and should be) completely separated from the work of analyzing a text. Main instantiates and transfers control to Brackets, the application controller.
2. In turn, Brackets is completely separated from the Stack class. Brackets contains all the logic needed to match pairs of open- and close-brackets. Stack is a container class that is used as a tool by Brackets, for the purpose of matching the pairs.
3. Brackets and Stack are both completely separated from the Token class. Token is a data class that describes the kinds of brackets we are looking for. It defines the form of a Token and how it can be printed.
4. Each class gives you a constructor function, where all the initializations can be written, and a destructor function for writing calls on delete (not needed here). This makes it easy to initialize and, later, free your storage, and you are unlikely to forget those tasks.

4.6.1 The main function, from file main.cpp

Notes on the main program:

- Every program must have a function named main. Unlike Java, main() is not inside a class. Like C, the proper prototype of main is one of these two lines. (Copy exactly, please. Do not use anything else.)

      int main( void );
      int main( int argc, char* argv[] );

- Separation of functionality is one of the basic design goals in OO programming. The tasks of main: deal with the command-line arguments then initialize and start up the application, are separated from the actions of the application itself. The task of this application is to read a file and analyze the brackets within it.
- We include the header file for the tools library. The functions banner(), fatal(), and bye() are defined in tools.cpp. Note that some of these functions use C-style formats, and that the output produced that way mixes freely with C++
- We call banner and print a greeting comment (lines 10–11). This is the minimum that a program should do for its human user.
• Line 13 tests for a legal number of command-line arguments. It provides user feedback and aborts execution if the number is not correct. Note the form of the usage error comment and imitate it when you are using command-line arguments.

• We use the command-line argument to name an input file, open it, and check for success of the opening process (lines 16–17).

• The header file for the brackets module (the application controller) is included on line 6. On line 19, we instantiate a Brackets object, and on line 20, we call its primary function with the open input stream as an argument.

• The brackets-object, b, is instantiated using new. This is how it would be done in Java, but in C++, the `new` is totally unnecessary and is truly bad style. Moreover, its presence in main(), which is not inside a class, means that the programmer has an obligation to free it before the end of the function. This is done on line 21. line 24. Please compare this to the non-dynamic version of the program available on Canvas.

• The last two lines (22 and 23) close the input file and print a termination message. Neither is strictly necessary. Both are good practice.

```cpp
#include "tools.hpp"
#include "brackets.hpp"

int main(int argc, char* argv[]){
    banner();
    cout <<"Welcome to the bracket checker." <<endl;
    if (argc!=2) fatal(string("usage: ") +argv[0] + " file.
";
    cout <<"Checking file " <<argv[1] <<endl;
    ifstream in( argv[1] );
    if (! in.good()) fatal( string("Can't open file ") +argv[1] + " for reading";
    if (! in.good()) fatal( string("Can't open file ") +argv[1] + " for reading";
    Brackets b; // Declare and initialize controller class.
    b.analyze( in ); // Execute the primary application function.
    in.close();
    bye();
}
```

### 4.6.2 The Brackets class.

```cpp
#include "tools.hpp"
#include "token.hpp"
#include "stack.hpp"

class Brackets {
    private:
        Stack stk;
        void doTok( char ch);
        void mismatch( const char* msg, Token tok );
    public:
        Brackets() =default;
        Brackets() =default;
        void analyze(istream& in); // Check nesting, matching in file.
};
```
Notes on the Brackets header file.

- The first two lines of each header file and the last line (29, 30, and 48), are called include guards. They ensure that no header file gets included twice in any compilation step. The symbol in upper case must be a unique symbol, within the application. It is customary to use the name of the class and underscore ‘H’ for this purpose.
- Next come the #include commands (lines 32–34) for classes and libraries that will be needed by functions defined in this class. Put them here, not in the .cpp file. The file tools.hpp includes all the necessary standard header files.
- The data declarations (lines 38-39) are private so that other parts of the program cannot accidentally modify their values.
- stk is initialized to a dynamically allocated empty stack. Dynamic allocation is completely unnecessary here, and is done simply to illustrate how to write a destructor.
- The other data member, toptok, is uninitialized; it will be used later as a way for the analyze() function to communicate with the mismatch() function.
- This class has two private functions, doTok() and mismatch() (Lines 40 and 41). They are private because neither should ever be called from anywhere except the analyze() function in this class. They were broken out into separate functions to shorten and simplify the logic of analyze().
- A constructor (line 44) normally initializes the object’s data members. However, one was initialized in the declaration and the other does not need initialization. There is nothing left to do, so the constructor is defined as =default.
- A destructor (line 45) is responsible for freeing all dynamic memory allocated by the constructor and/or by other class functions. This one calls delete to deallocate the stack that was allocated on line 38.
- This constructor and destructor are both short, so they are defined inline. The analyze() function is long and contains control structures, so it is just declared here (lines 46) and defined in the .cpp file.

Notes on Brackets.cpp. This is the heart of the application.

Layout and style

- The .cpp file for a class should #include the corresponding header file and no other files.
- Please note the line of dashes before each function. This is a huge visual aide. Do it. It saves time during debugging and helps the reader.
- The return types of functions in this class are written, with the class name, on the line above the function name. As the term goes on, return types and class names will get more and more complex. Writing the function name at the left margin on a second line improves program readability.
- The definitions of analyze and mismatch belong to the Brackets class, but they are not inside the class (between the word class and the closing curly brace). Unlike Java, a C++ compiler does not look at your file names, and has no way to know that these functions belong to your Brackets class. Therefore, you must explicitly supply the full name of each function (i.e. Brackets::analyze) in the .cpp file.

The analyze function handles the input and error checking.

- The argument to analyze (lines 46 and 55) is a reference to an open istream. Streams are always passed by reference in C++.
- This function can be called using cin as an argument, or using an open ifstream because both are varieties of istream.
- An infinite for loop (lines 57–61) is used to read and process the input file one character at a time. The loop exit is on line 60 because it is not valid to test for end of file until after the input statement, so you normally do not start an input loop by writing: while (!in.eof()) ...
• We use the >> function (line 58) to read the input. This is a generic function: the type of its argument determines what will be read. In this case, ch is a char, so the next non-whitespace character in the file will be read and stored in ch. We use >> instead of get() when we want to skip leading whitespace.

• All of this application’s work is done in doTok(), which is called for each char that is successfully read.

• This application quits on the first error. Lines 63–67 look at all the evidence and explain what the error is, then call mismatch() (lines 71–77) to print the error comment and terminate (line 74) by calling fatal().

• Note that the * on line 74 is necessary because the stack was allocated by new.

```cpp
// ===============================================================
// Bracket-matching example of stack usage File: brackets.cpp
// ===============================================================
#include "brackets.hpp"

void Brackets::
analyze( istream& in) {
  char ch; // For the input.
  for (;;) { // Read and process the file.
    in >> ch; // Does not skip leading whitespace.
    if (!in.fail()) doTok(ch); // Process input, if any.
    else break;
  }
  // Control comes here on EOF or stream failure.
  if (in.bad()) fatal("Low-level failure in stream\n");
  if (stk.empty()) {
    cout << "The brackets are properly nested and matched.\n"
  } else cout << "\nPremature eof. " <<stk <<"\n";
}

void Brackets::
mismatch( const char* msg, Token tok ) {
  cout <<"\nMismatch in file: " <<msg <<stk; // print stk content
  fatal("\n\nQuitting on first mismatch\n");
}

void Brackets::
doTok(char ch) {
  Token curtok = Token( ch );
  cout <<"\nNew token: " <<curtok; //debugging
  switch (curtok.getSense()) {
    case TokenSense::NONE: break; // Not a bracket character.
    case TokenSense::LEFT:
      stk.push(curtok);
      break;
    case TokenSense::RIGHT:
      if (stk.empty()){
        mismatch("Too many right brackets", curtok );
      }
      if (! curtok.sameAs(stk.top())){
        mismatch("Closing bracket has wrong style", curtok );
      }
      else stk.pop();
      break;
  }
  cout <<" " << stk <<endl; // debugging
}
```
The heart of the heart: doTok(). Stores the input characters in Tokens and manages them according to their properties.

- Line 81 declares a local temporary variable named curtok. This calls the Token constructor with the input character, ch, to initialize the Token and classify its sense (left or right) and style (paren, angle, brace, square).
- We create Token objects so that we can store the input along with its two classifications. The task of figuring out the proper classification is delegated to the Token class because the Token class is the expert on everything having to do with Tokens.
- The Token object is created inside doTok() as a local variable. Every time we go around this loop, a new Token is created, initialized, used, and possibly copied into the stack. The local temporary is deallocated automatically when control reaches the end of the function on line 102. This is efficient, convenient and provides maximal locality.
- When we get to line 83, the token has been classified and we can test whether it is of interest (some sort of bracket) or not (most characters). If it is of no interest, we leave the loop, delete the dynamic memory, and return from the function.
- Within a class, the this-> prefix is generally not useful. So there is no need to write the longer version: this->stk-> on lines 64, 74, 87, etc.

The switch. For a bracket-matching application, there are three possible kinds of input characters: an open-bracket, a close-bracket and a non-bracket.

- Non-brackets are of no interest to this program so they are skipped (line 84).
- An open-bracket must be saved to compare, later, to the next close-bracket. A stack is used here, as in many other applications, to check for proper nesting/matching of symbols. Line 87 pushes the new Token onto the stack named stk declared on line 38. This transfers custody of the token to the stack.
- A close-bracket must be compared to the Token currently at the top of the stack. (lines 91 - 98). This is complex because we must test for two error conditions:
  - An empty stack means that there are more close-brackets than open-brackets.
  - If something is on the stack, it might not match the style of the new Token.
  - If the styles do match all is well. We pop the open-bracket off the stack and discard both it and the matching close-token. Then we return to the analyze() function.
- If an error happened, the stack is not printed and the mismatch() function terminates the program.

4.6.3 The Token Class

Notes on token.hpp
Token is a data class. Its role is to organize all aspects of the data and functions relating to tokens.

- The class begins with the usual parts: include guards (lines 107-108) and an include command (line 110). It ends with a print() function and, outside the class, a definition for the << operator.
- Lines 112–113 declare two enumerations: BrackStyle and TokenSense that are used to describe essential properties of a token. An enum class is a modernized version of the old C enum type and is used in almost the same way. However, in C++, an enum is not the same type as an int.
- The private data members are limited to the essential parts: a character and the two classifications of that character, as defined by the enumerations.
- The two classifiers are computed once and used multiple times. This is a better design plan than recomputing a classifier each time it is needed. A major principle for improving security, maintainability, and stability is: “Never compute something twice and rely on the two answers to be the same.”
• There is one private function, used to simplify and clarify the code of the constructor. It is private because no other class should ever call it.

• There are two methods for the constructor. Lines 123/141–153 construct and initialize a Token, given an input character. Line 124 constructs a Token with the default initialization. This is useful to store Tokens that are popped off the top of the stack.

• Line 126 declares a function to look at the private parts of two Tokens and say whether they match. This is part of the Token class because Token is the expert on all things relating to Tokens.

• One public accessor function is defined, `getSense()`, (line 127). It is needed by Brackets to decide whether to push the Token onto the stack, or pop one off the stack.

    ```cpp
    enum class BrackStyle { SQR, PAREN, CURLY, ANGLE, NONE };
    enum class TokenSense { LEFT, RIGHT, NONE };
    class Token {
    private:
        char ch;
        BrackStyle style;
        TokenSense sense;
        void classify( char ch );
    public:
        Token( char ch );
        ~Token() = default;
        bool sameAs( Token tk ){ return (style == tk.style); }
        TokenSense getSense() { return sense; }
        ostream& print( ostream& out);
    };  
    inline ostream& operator<<( ostream& out, Token t ) {
        return t.print( out );
    }
    ```

Notes on `token.cpp`.

• Within a class, a member name without `this->` means the same thing as that name with `this->` unless the member name is the same as a parameter name. In that case, the name without `this->` refers to the function’s parameter, and with `this->`, it refers to the class member.

• The constructor uses `this->` (line 142) to initialize the data member named `ch` because the function’s parameter has the same name as a data member. Using a member name for a parameter is preferred by some programmers over inventing a new name for the parameter. Others dislike it because it makes the references ambiguous and requires the use of `this->`.

• The print function (lines 156–172) uses two switches to print appropriate strings for the enum constants. Following that, the Token’s character is printed.

• Line 143 lists the chars that we are designating as “brackets”. In the bracket string, even numbered subscripts are all open-brackets; the close-brackets are all stored in the odd-numbered subscript following the matching open-bracket.

• Line 144: `find()` is a standard function in the C++ string library. It returns the subscript of the character in the brackets string (if found) or `npos` (otherwise).

• Line 144 does two things for us:
4.6. CODE EXAMPLE: ARE THE BRACKET NESTED PROPERLY?

- A return value of `npos` (line 145) tells us that the input character is not a bracket. In this case, we initialize the Token's two classifiers to `NONE`, and the Token will be ignored. (lines 146–147)
- If the input character is one of the brackets in the string, Line 144 tells us its position in the string. This, in turn, allows us to classify the bracket. (lines 150–151)

- There are twice as many bracket characters as constants in the enum declaration, and they are in the same order. So dividing \( k \), the position of a characters, by 2 (line 150) gives the position of the symbol in the enumeration. Integer divide truncates the remainder if \( k \) is odd. values, resulting in the same style-classifier for the open-symbol and close-symbol in each pair. The cast converts the integer to an enum constant.
- On line 151, we say that if \( k\%2 \) is an even number, the token is an open-bracket, otherwise it is a close bracket.

```cpp
int k = brackets.find( ch );
if (k==brackets.npos) { // If not found in brackets string.
    style = BrackStyle::NONE;
    sense = TokenSense::NONE;
}
else {
    style = (BrackStyle)(k/2); // int divide & truncate.
    sense = (k%2 == 0) ? TokenSense::LEFT : TokenSense::RIGHT ;
}
```

4.6.4 The Stack Class

A stack is a necessary tool for this application. There is a standard stack template that could be used, but that class does not define a print function, and there is no nice way to add that feature except by making a derived class. However, for understanding what is happening, and for debugging, it is important to print the stack as it changes.

This class is an adapter for the standard vector template that defines a print function and the old-time names for the stack operations. It also blocks access to unwanted vector functionality. Vectors manage their

---

`Deferred for later because we are not yet ready for using derivation.`
own storage safely and grow longer, as necessary, to contain all that the programmer puts into them. Both of
these properties are very helpful for programmers.

- There is only one data member: a vector. Here we want an empty Stack. Thus, the constructor is defined
equal to default.

- The destructor of Stack has the responsibility of deallocating all the dynamically allocated objects attached
to pointers in the stack’s vector. The pointers themselves will be deallocated by the Vector destructor.

- The traditional function names `push`, `pop`, `top`, and `empty` are defined by delegating the task to functions
in the vector class. Three of those have longer, less obvious names.

- The print function (lines 194–198) produces a readable printout of the stack that was very important
during debugging.

- Traditionally, `print` would be defined in a .cpp file because it is 5 lines long and because it contains a
loop. This class defines it as inline. A modern compiler will probably look at the way the function is used,
and how many times it is called, and actually compile it as a normal, non-inline function.

```cpp
167 //-------------------------------------------------------------------------------
168 // Declaration of a Stack container class to store Tokens.
170 //-------------------------------------------------------------------------------
171 #pragma once
172 #include "token.hpp"
173 #include "tools.hpp"
174
175 class Stack {
176 private:
177 vector<Token> v; // To store opening brackets.
178
179 public: //---------------------------------------------------------
180 Stack() =default;
181 ~Stack() =default;
182 void push( Token tk ){ v.push_back( tk );}
183 void pop() { v.pop_back(); }
184 Token top(){ return v.back(); } // Don't use 'if' here!
185 bool empty(){ return v.empty(); } // Don't use 'if' here!
186 ostream& print( ostream& out ){
187     out << "The stack is: bottom~~ ";
188     for (Token tk : v) out << tk << ' ';
189     return out << " ~~top";
190 }
191 inline ostream& operator << (ostream& out, Stack& s) {
192     return s.print(out); // Connect Stack::print() to the << operator.
193 }
194
195
196 inline ostream& operator << (ostream& out, Stack& s) {
197     return s.print(out); // Connect Stack::print() to the << operator.
198 }
```

4.6.5 The Input and Output

This shows the results of trying to run the program without an appropriate input file:

<table>
<thead>
<tr>
<th>Contents of input file:</th>
<th>Output produced:</th>
</tr>
</thead>
<tbody>
<tr>
<td>No file name supplied.</td>
<td>Welcome to the bracket checker.</td>
</tr>
<tr>
<td></td>
<td>usage: brk filename</td>
</tr>
<tr>
<td></td>
<td>Press '.' and 'Enter' to continue</td>
</tr>
<tr>
<td>Incorrect file name</td>
<td>Welcome to the bracket checker!</td>
</tr>
<tr>
<td></td>
<td>Checking file 'bad4.txt'</td>
</tr>
<tr>
<td></td>
<td>Can't open file 'bad4.txt' for reading</td>
</tr>
</tbody>
</table>

Following is a list of the contents of four input files (left column) and the outputs that are produced from them. Banners, byes, and debugging code have been deleted from the output. Each set of output on the right was produced by the Brackets program after processing the input on the left. This program ends when the first
4.6. CODE EXAMPLE: ARE THE BRACKET NESTED PROPERLY?

bracketing error is discovered.

Making a test plan. A test plan is a list of tests that you propose to perform when your program is ready. These tests are drawn from the specifications for a program and may be augmented later by more tests drawn from the nature of the implementation.

Each test should say what is being tested and why, followed by the actual output that you expect and require. The set of tests should include at least one correct example and also exercise every possible error condition. Often, the tests are embedded in a testing function that can easily be run and rerun many times and used again after each program modification.

<table>
<thead>
<tr>
<th>Contents of input file:</th>
<th>Output produced:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Well-matched and balanced.</td>
<td>Welcome to the bracket checker!</td>
</tr>
</tbody>
</table>
| (<>){[]}]                                  | Checking file 'ok.txt'
|                                            | The stack is: bottom ... top                                                   |
|                                            | The brackets are properly nested and matched.                                   |
| (<>){ }[ ]                                | Welcome to the bracket checker!                                                  |
| Missing close bracket.                      | Checking file 'bad1.txt'
|                                            | New token: Left-Square['
|                                            | Mismatch in file: Too many left brackets                                        |
|                                            | The stack is: bottom Left-Square['
|                                            | ...top                                                                          |
|                                            | Quitting on first mismatch                                                      |
| Missing open bracket.                      | Welcome to the bracket checker!                                                  |
| (<<<>}){}                                  | Checking file 'bad2.txt'
|                                            | New token: Right-Square']
|                                            | Mismatch in file: Too many right brackets                                        |
|                                            | The stack is: bottom...top                                                       |
|                                            | Quitting on first mismatch                                                      |
| Longer file w. non-brackets.               | Welcome to the bracket checker!                                                  |
| (<< This is some text                      | Checking file 'bad3.txt'
| >>)                                         | New token: Close Paren : )                                                      |
| Some more text <<                           | Mismatch in file: Too many right brackets                                        |
| >>)                                         | The stack is: bottom ... top                                                     |
| {}                                           | Quitting on first mismatch                                                      |
Chapter 5: Dynamic Memory Management

Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information on it.

This principle also applies to data: you can have a variable that stores your data, or you can have a pointer to it that tells you where to find it.

5.1 Dynamic Memory Management

In Java, dynamic allocation is needed for arrays and for the data objects stored in the standard collections. In comparison, C++ minimizes the use of dynamic allocation and, by doing so, allows program code to run more efficiently.

This section presents an overview of the basic principles involved with dynamic memory management in C++. They are straightforward and not difficult to understand and apply.

5.1.1 Creation and Deletion.

When you declare a variable in a block, or as part of an object, or as a parameter, it is automatically created and initialized when control enters that block of code. When control leaves the block, it is deallocated by running the class destructor. Almost all objects in C++ programs are allocated, initialized, and deallocated this way. They are called auto variables because this is all automatic.

A few objects are static. They are created at load time and persist until program termination, when they are automatically deallocated. This includes variables declared outside of all classes and class members that are explicitly declared static.

Finally, a few objects are created dynamically by calling new or new[]. These are not freed automatically. Some part of the program must explicitly free them by using delete or delete[].

Using Dynamic Memory. Dynamic memory is used for three essential purposes:

1. To build a container for an indefinite amount of data. Example: a string, a vector, or a linked list.
2. To build some data structures, particularly trees and graphs.
3. To enable late binding, that is, defer until run time the choice of which variant of a polymorphic type to use for a particular application.

General rule: Do not use dynamic allocation unless you need it. Do not use it for ordinary variables and arrays.

5.1.2 Dynamic Allocation Incurs Overhead.

The C++ run-time system must know how many bytes are in each dynamically allocated storage block in order to free that memory when the block is deleted. For this reason, the compiler calculates the size of every type and stores it as part of the type, for later use by new and delete. After a new object is allocated, its size is stored as part of the allocation block, before the beginning of the part used by the new object.

There is no rule in the language standard that the system must allocate a particular amount of overhead space, or where it must be. However, every compiler must do something of this sort. Here we illustrate an old and common strategy, which was used for both C and C++ for many years. Extra space was allocated at the beginning of every dynamically allocated area and initialized to the actual number of bytes in the allocation block, including the overhead. In the Figures, this length field is shown against a pale gray background. Note
that this field precedes the user’s storage area, and the address returned by malloc() was the address of the beginning of the user’s area.

Example: Suppose we are implementing a graphics program that uses Points. One component might be a Point object, pt.

\[ \text{Point* pt = new Point(0.0, 0.0); // The origin of our graph.} \]

Figure 5.1 shows how the allocation area is configured on many systems: a single instance of class Point is dynamically allocated and initialized by the Point constructor. The resulting pointer is stored in pt. In addition to space for 81 short ints, the length field includes 4 bytes for itself.

\[ \text{Figure 5.1: Allocation overhead for an array.} \]

**Freeing the dynamic memory.** When the newed object is no longer useful, it should be freed by applying delete to the variable that points to the dynamic object:

\[ \text{delete pt;} \]

**Allocating a dynamic array.** When an array is allocated dynamically, two pieces of information are stored in the new object: (1) the total allocation length and (2) the number of slots in the array. Example: Suppose we want to dynamically allocate an array named rect of 4 Points. We would declare the array thus:

\[ \text{Point* rect = new Point[4];} \]

Now, we have defined a point to be two doubles, so a Point occupies 16 bytes of memory. Then to dynamically allocate an array of 4 Points, we need \( 4 \times 16 \) bytes for the data and 8 bytes for the overhead = 72 bytes. The dynamic object looks like this:

\[ \text{Figure 5.2: Allocation overhead.} \]

**Freeing a dynamic array.** Deallocating a dynamic array is done in two steps:

1. First the destructor of the objects stored in the array is run N times to delete the N items in the array.
2. Then the dynamic array is deallocated.

The correct code in this case is:

\[ \text{delete[] rect;} \]

### 5.2 Inventory with Pointers

#### 5.2.1 UML Class Diagram Basics

Several relationships may exist between two classes: composition, aggregation, association, derivation, and friendship. These are explained in the paragraphs below and illustrated by class relationship diagrams. The four relationships at the top of this list provide the most protection for members of class B. As we move down the list, protection decreases, that is, class A functions get more privileges with respect to the private parts of B. On a separate dimension, a class can be instantiated from a class template. The template, itself, is not a class.
5.2. INVENTORY WITH POINTERS

5.2.2 Composition.
Class A composes class B if a B object is part of every A object, from the time the A object is created. When A dies, B dies, and B can be be part of exactly one A object. Class A could compose any definite number of B objects; the number is indicated at the B end of the A-B link. The black diamond on the A end of the link indicates composition.

In this diagram, class A has one member named m of class B, which is constructed when an A object is constructed. When the A object is deallocated, it will cause the deallocation of the corresponding instance of B. A can call the public functions of B to operate on m.

5.2.3 Aggregation.
Class A aggregates class B if a B object is part of every A object, but the lifetime of the B object does not match the lifetime of the A object. The B object may be allocated later (or even earlier) and attached to A with a pointer. Unique pointers are appropriate for this relationship. A could aggregate multiple B objects and one B object might be aggregated by more than one A object.

Some experts believe that aggregation does not exist as a separate, definable, class relationship. In pointer-based languages such as Java\(^1\), that is probably true.

In the diagram, the white diamond on the A end of the A-B link indicates aggregation. Class A has one member named mp of type B*. A can call the public functions of B by using mp->. A can call the public functions of B by using mp->.

When an instance of A is deallocated, B will be deallocated if a unique pointer was used, or if A’s destructor explicitly deletes it. Otherwise, B becomes a memory leak.

5.2.4 Association.
Class A is associated with class B if a member of A points at an instance of class B (or a set of instances), but both kinds of objects have an independent existence. Neither one is “part of” the other. The B object is generally not created by A’s constructor; it is created independently and attached to A.

Simple Association. In a simple association, class A is associated with a definite number of B objects, often one. This is usually implemented in A by one pointer member that may be null or may be connected to an object or array of objects of type B. A can use the public functions of B by using mp->.

One-many Association. Class A is associated with a set of zero or more objects of class B. The relationship can be implemented by A pointing to an array of B’s; in this case, A can use the public functions of B by using subscript and mp->.

Notes on the UML for Inventory version 2\(L\). Chapter 3 gave a UML diagram in which

- Black diamonds: Main instantiates Inventory, and Inventory instantiates a vector as one of its data members. This is class composition because the lifetimes of main(), Inventory, and the vector are all the same.
- Line with *: This is not composition because the vector is created before any parts are stored in it. It is one-many association: one vector will store many Part*s. The Part*s are created by calling new and must later be explicitly deleted.

\(^1\)Java does not support C-style pointers. However, every object in Java is implemented through dynamic allocation and attached to a variable through a pointer.
5.2.5 The main program

Chapter 3 presented and explained an Inventory application. Nothing in the program required pointers. Here we add a small bit of functionality: the revised program prints a list of screwdrivers at the end, and their average cost. We use this example to show how to use pointer variables, new, and delete.

Notes on the main program.

- This is a copy of the main function in Chapter 2, Inventory, with one line added.
- Line 19 prints out the average price of all the screwdrivers in the chain of stores.

```cpp
int main( int argc, char* argv[] ) {
  // Demo program for pointers new and delete.
  // Michael and Alice Fischer, June 8, 2023.
  // Instructional targets:
  // 1. Pointer usage and syntax
  // 2. new and delete.
  #include "tools.hpp"
  #include "inventory.hpp"
  #include "part.hpp"

  // Header file for hardware store parts. Inventory.hpp
  #pragma once

  class Inventory {
    private:
      ifstream& fin;
      vector<Part*> stock;
    public:
      Inventory( ifstream& fileIn ) : fin(fileIn) {}
      ~Inventory() { for(Part* p : stock) delete p; }
      void readParts();
      void average();
    }; inline ostream&
    operator <<(ostream& out, Inventory& p){ return p.print(out); }
```

5.2.6 The Inventory class version 2

```cpp
#include "tools.hpp"
#include "part.hpp"

class Inventory {
  private:
    ifstream& fin;
    vector<Part*> stock;
  public:
    Inventory( ifstream& fileIn ) : fin(fileIn) {}
    ~Inventory() { for(Part* p : stock) delete p; }
    void readParts();
    void average();
  }; inline ostream&
  operator <<(ostream& out, Inventory& p){ return p.print(out); }
```
5.2. INVENTORY WITH POINTERS

Notes on the Inventory header file.

- Three lines have been changed since Chapter 3;
- Line 34: We declare that the vector will store type Part* instead of type Part.
- Line 37: Because we use dynamic memory in this class, we need a real destructor. Line 19 goes through the vector of Part*s and deletes each one.
- Line 41: This is the prototype for the function that calculates and prints the new information.

The Inventory implementation.

```cpp
45 // Implementation file for hardware store parts. Inventory.cpp
46 //
47 // #include "inventory.hpp"
48 //
49 void Inventory::
50 readParts() {
51   string buf;       // Holds one line of the file.
52   Part* up;        // Pointer to the current Part.
53   cerr << "Reading inventory from parts input file.\n";
54   for(;;) {
55     getline(fin, buf);
56     //cout << buf << endl; // Echo-print every line.
57     if (fin.fail()) cerr <<" Not a good part: " <<buf <<endl;
58     else {
59       up = new Part(buf);
60       if (up->getPrice() == 0) {
61         cerr <<" Bad data for price: " <<buf <<endl;
62         delete up;
63       }
64       else if (! up->filter("screwdriver")) {
65         cerr <<" Not of interest: " <<buf <<endl;
66         delete up;
67       }
68       else stock.push_back(up);
69     }
70   }
71   if (fin.eof()) return;
72 }
73 if (fin.bad()){  // Low-level read error
    cerr <<"Low-level read error\n.";
    exit(1);
74 }
75 
76 void Inventory::
77 average(){
78   float total = 0.0;
79   for (Part* p: stock) total += p->getPrice();
80   cout <<"Average price of screwdrivers: 
" << total / stock.size() << endl;
81 }
82 
83 ostream& Inventory::
84 print( ostream& fout ){
85   cout <<"Partname......... Store Quant Price\n";
86   for (Part* p: stock) fout <<p <<endl;
87   return fout;
88 }```

Notes on the Inventory implementation.

- There are two changes and many additions here.
- Lines 53 and 91 use the type Part* instead of Part.
- Lines 58...71 make more tests, in a more complex pattern, because they are concerned about two issues, not one: whether the Part was constructed properly and whether it describes a screwdriver.
- Lines 63 and 67 must delete any parts that will not be put into the vector.
- The function on lines 79–85 is entirely new. It goes through the screwdrivers in the vector and computes, then prints, their average price.

Notes on the Part class. This version of Inventory uses the Part class from Chapter 3, unchanged. Therefore, Part is not listed again.

5.3 Smart Pointers = Fewer Problems

Pointers are difficult. Pointers are as old as C. They remained unchanged for the first 30 years of the life of C++. From the beginning, the C-style pointers confused and bedeviled programmers – so much so that they were hidden in Java and Python as part of making those languages accessible to beginners. Why were pointers included in the C language?

- Pointers introduced indirect use of variables.
- They made dynamic memory allocation possible and ushered in growing arrays and C++-style strings.
- Explicit pointers enable efficient run-time implementations and efficient storage management.

What aspects of pointers cause difficulty? These important capabilities required adding a layer of complexity, to both syntax and semantics, to support pointers.

- Programs with pointers are harder to write and harder to debug. They require more expertise.
- Indirect addressing is more complex than direct addressing.
- The syntax for declarations is easily confused with the syntax for use.
- New technical words were introduced, for example, “referent”.
- The question of custody arose – programmers became responsible for freeing memory explicitly when a pointer’s referent is no longer useful. Worse, when a pointer to an object is put into or taken out of a data structure, custody changes, affecting the question of which class has the responsibility to delete the objects.
- Dynamic memory allocation requires later explicit deallocation.

Serious pointer problems. An unskilled programmer, trying to use pointers, often makes errors that cause a program to crash or produce bad output. Some of these are:

- A memory leak – failure to delete a dynamic object prior to end of the program.
- Illegal deletion: attempting to delete a pointer that does not store the beginning of a dynamic memory block
- Double deletion: an attempt to delete a memory that was previously deleted.
- Illegal reference:
  - Using a dangling pointer – one whose referent has gone out of scope and no longer exists.
  - Using an uninitialized pointer.
  - Using a pointer after deleting its object.
5.3. SMART POINTERS = FEWER PROBLEMS

5.3.1 Memory Management

Proper memory management is important. If memory is not deallocated when no longer needed, it just sits there, occupying memory until the program is terminated. When memory is allocated in a loop, but not deallocated, the memory “footprint” of the application grows and grows and the application becomes a memory hog, ultimately causing thrashing, system slowdown, and an inability for any running application to make progress.

How C++ Automatic Memory Management Works. Java and Python both rely on garbage collectors to free blocks of dynamic memory that are no longer in use. These languages use dynamic memory allocation for most or all data, and garbage collection is a very expensive way to free up memory space.

In contrast C has always expected programmers to manage memory themselves. This was hard to teach and very error-prone among half-trained programmers. C++ decided to solve both problems. It supplies a set of tools that, together, manage memory efficiently:

- Every class has a destructor. The default destructor is enough for most classes, but those with dynamic extensions require more than that.
- Smart pointers can be used to wrap a C-pointer in an object, and free that object when it is no longer useful.
- C++ supplies an efficient automatic memory management system that is triggered at the end of every block. It frees all the objects and sub-objects in that block by using the destructors defined in the appropriate classes.

Cascading destructors. When an object (any object) goes out-of-scope, its destructor will automatically be run. If there smaller objects are composed inside the first one, their destructors will also be run, and so on. The destructors cascade, freeing everything within their reach. This is illustrated by the BarGraph example program in Chapter 9.

The problem is C-style pointers. The destructor cascade ignores them and the programmer must explicitly delete them. If your class new’s an object, its destructor should delete that object. However, by using smart pointers to wrap pointers, all this can be avoided, and the objects in the custody of the smart pointers are freed as part of the cascade.

Standard container classes. The template library defines a variety of data structures as templates. Every one of them totally manages its own memory. For example, a programmer who is using a `vector< typeX >` knows that the vector implementation uses dynamic allocation. However, because that allocation is deleted by the vector destructor, the application programmer is never aware of when either allocation or deallocation happens. In this case, objects of `typeX` will be stored in the vector. When the vector goes out-of-scope, the class X destructor will be run on each X-object stored there, then the space for the vector will be freed.

But C-pointers do not have destructors, so if you put them in a vector, their referents are not freed and become memory leaks. This is the reason for unique pointers: They ARE objects, and their destructors will be run by the vector destructor. Then the referents of the unique pointers will be freed by the unique pointer destructor. In this way, freeing dynamic memory becomes automatic.

Debugging. Some of the most troublesome program bugs are caused by mishandling storage allocation, or failure to understand what happens, when, and why. During the long process of mastering C++, it is a good idea to track the allocation and deallocation actions:

- By printing comments in constructors and destructors.
- By printing an object (or part of it) when it is created.
- By using dump functions or macros as in Chapter 4.
## 5.4 Unique pointers

To address some of these problems, a new kind of pointers, called *unique pointers* was introduced into C++. The type of a unique pointer is `unique_ptr`. It wraps an old-style pointer in an object with a constructor and destructor and takes custody of the pointer's referent. This solves several problems:

- Deletion is automatic: when the unique pointer goes out of scope, the referent is also deleted.
- There is no need to explicitly delete these wrapped-up blocks of dynamic memory.
- There is no need to track custody of the dynamic memory area.
- It is less confusing.

### Details of unique pointers

A `unique_ptr` supports a limited pointer functionality:

- It provides access to its referent through operators `*` and `->`.
- It supports operator `[[]` for array objects.
- It does not support pointer arithmetic, for safety reasons.
- It automatically deletes the object it manages (using the appropriate deleter) when it is destroyed, or when its value is changed by an assignment operation.

### Syntax and usage for unique pointers

This example relies on having `using namespace std` in your program.

```cpp
#include "tools.hpp"
using unipt = unique_ptr<int>;

struct duo { int a, b; duo(int a2, int b2): a(a2), b(b2){} }

int main( void ) {
    cout << "Testing unique pointers.\n";
    unipt up1(new int); // referent is not initialized.
    unipt up2(new int(100));
    *up1 = *up2 * 2; // Testing dereference and times.
    cout << "up1: " << *up1 << " up2: " << *up2 << '\n';
    // up1 = up2; // Illegal
    unique_ptr<int[]> upar (new int[5]); // referent is an int array
    for (int k=0; k<5; ++k) upar[k] = 2*k;
    cout << "upar[2]: " << upar[2] << '\n';
    cout << "upar[4]: " << upar[4] << '\n';
    unique_ptr<duo> upmp (new duo(3, 9)); // referent is a duo struct.
    cout << "upmp->a: " << upmp->a << '\n';
    cout << "upmp->b: " << upmp->b << '\n';
    return 0;
}
```

### Notes on the unique pointers

- Line 4. To use smart pointers, you must include `<memory>`; it is included by tools.hpp.
- Line 5. This defines a short name for the long, awkward built-in type name.
- Lines 12-13 declare two unique pointers that wrap `int` pointers; `up2` is initialized, `up1` is not.

---

2This section presents only an entry-level set of properties and functionalities. Much more information can be found at cplusplus.com/reference/memory.
5.4. UNIQUE POINTERS

- Line 14 fetches the value of up2’s referent, multiplies it by 2, and stores the result in the referent of up1. The referent scan be used the same ways as the referents of simple pointers.
- Line 15 prints out the results of this section.
- Line 16 is an attempt to do an illegal operation: copy one unique pointer into another. Copying unique pointers is not defined.
- Lines 18–21 show how to create and use a unique pointer for a dynamic array,
- Line 18 creates both the dynamic array and the unique pointer to it.
- Line 19 initializes the array. Lines 20 and 21 output two of its five values.
- Line 7 defines a structure type. Note that this type includes a public constructor.
- Line 23 instantiates the struct, initializes it, and creates a unique pointer to it.
- Lines 24 and 25 show how to access the parts of that struct.

The output.

```
Testing unique pointers.
up1: 200
up2: 100
upar[2]: 4
upar[4]: 8
upmp->a: 3
upmp->b: 9
```

5.4.1 Inventory version 3, with Unique Pointers

```
Figure 5.4: Part*s are now wrapped in upt objects.
```

Notes on inventory.hpp, version 3. The changes in this version are all due to using unique pointers instead of C-pointers for the dynamic objects.

- Line 8 in the header file defines a short and convenient synonym for the typename `unique_ptr<Part>`
- Line 13 declares that the vector will be used to store upt objects.
- Line 16 declares a do-nothing destructor. Unique pointers are self-managing, so there is no need for the delete-loop.

Notes on inventory.cpp, version 3.

- Line 39 creates a new Part and wraps it in a upt object.
- Line 40 and 42 use the upt object the same way an ordinary pointer would be used.
- Line 44 moves the valid upt object into the vector. A unique pointer cannot be copied (or pushed) into a vector; however it can be moved. This issue will be explained in Chapter 6.
• Parts that are not valid do not need to be deleted explicitly because they are wrapped up in unique pointers, which take care of freeing the dynamic memory. The small extra trouble to use unique pointers allows us to forget about memory leaks and deletion in further processing.
• Line 57 is a loop that processes all the Parts in the vector. The upt parameter must be passed by reference because it cannot be copied, and therefore, call-by-value cannot be used.

5.4.2 Shared pointers

Shared pointers are like traditional pointers in some ways, but the shared_ptr also implements a reference counter that keeps track of how many pointers all point to the same object. An object of a shared_ptr type takes ownership of the pointer’s referent and shares that ownership. The group of owners of a pointer/referent become responsible for its deletion when the last one of them releases that ownership. A shared_ptr releases ownership of the pointer/referent it co-owns:

• Whenever the pointer, itself, is destroyed, as when it goes out of scope.
• As soon as the pointer’s value changes either by a call to shared_ptr::reset or by an assignment operation.

Once all shared_ptr objects that share ownership over a pointer have released this ownership, the managed object is deleted automatically. (delete or delete[] is called.)

Shared pointers address and solve these problems:

• Using a pointer after deleting its object.
• Dangling pointers – a pointer whose referent has gone out of scope and no longer exists.

Details. Use * and -> to access the referent. For safety reasons, shared pointers do not support pointer arithmetic. Warning: Sharing only happens by copying a shared_ptr value. Do not create two shared_ptr that have the same referent. If two shared_ptr objects are constructed from the same pointer, they will both own the pointer/referent without sharing it. This causes a dangling pointer when one of them releases it (deleting its managed object) and leaves the other pointing to an invalid location.

Empty vs. null shared pointers. A newly created and uninitialized shared_ptr does not own any pointer and is called “empty”. A shared_ptr that owns a pointer with no referent is called a null shared_ptr and must not be dereferenced. Notice that an empty shared_ptr is not necessarily null, and a null shared_ptr is not necessarily empty.

A related class, weak_ptr, is able to share pointers with shared pointers without owning them. Weak pointers are beyond the scope of this book.

5.4.3 Example for shared pointers.

This example creates several pointers that share a referent, then destroys them one at a time, printing the reference count after each operation.

Notes on the code for shared pointers

• Line 11: P1 is an empty pointer. It does not point at anything and cannot be used.
• Lines 12–14 P2 points to something, but it is a null pointer. Its use count is 0
• Lines 15–16: P3 points at a dynamically allocated int, so its use count is 1. The int is initialized to 22. Note that parentheses denote initialization, square brackets denote an array.
• Lines 17–18: P4 is a copy of P3. It points at the same object as p3. The use-count is now 2.
• Lines 19–20: P4 points at the same object as p3 and the use-count is now 3.
• Line 21 changes the value of the int, p3’s referent. This changes the value of *p3, *p4, and *p5.
• Lines 22–23: P6 is another copy of p3. The use-count goes up and the int’s value is p3’s current value.
• Lines 24–25: P4 is reset, that is, set back to the empty state. The use count for p3 decreases.
• Lines 26–27: Creating p7 shows that p3 is still alive, healthy, and shareable.
• Line 28: P6 is reset. The use count for p3 decreases.
• Line 30: P5 is changed by assignment, so the use count for p3 decreases.
• Line 31 reset p7. Note that this decreases p3’s use-count.
• Line 32: P3 is reset, returning its use-count to 0, like p1.

The output:
Testing shared pointers.
 count contents
 p1: 0
 p2: 0
 p3: 1 22
 p4: 2 22
 p5: 3 22
 changed referent of P3 to 15.
 p6: 4 15
 reset p4, now p3= 3 15
 p7: 4 15
 reset p6, now p3= 3
 assigned to p5, now p3= 2
 reset p7, now p3= 1
 reset p3, now p3= 0

5.5 Shared Pointers for Memory Management

The C++ manual memory management system relies on destructors and a firm grasp on custody. Both kinds of smart pointers wrap a pointer in an object and enable the memory management system to directly handle freeing all allocated objects, including a pointer’s referent, when the pointer goes out of scope.

Complex data structures, including graphs and blockchains, use many pointers to create objects with multiple access paths. This is where shared pointers become very useful. It is difficult to find any simple example of any situation in which a shared pointer is useful. However, the following small example illustrates the nice features of shared pointers. This program is shown in three versions.

Mixed version 1 is very simple code, but irresponsible. It results in two memory leaks because instantiates the Mixed class twice. The Mixed constructor calls new, and the Mixed destructor does not delete the dynamic memory blocks.

Version 1 Output.

Initial values A, B: Joe 12 Mary 10
After copying B = A: Joe 12 Joe 12
After setting B.name[0] = 'T': Toe 12 Toe 12
Normal termination

Mixed version 2 tries to be responsible, but it crashes. Here, Mixed has a proper destructor. However, when A was copied into B, B’s dynamic block (containing “Mary”) was never deleted. Then, after “normal termination”, both A and B go out of scope, and the destructor is called twice. The first time it is fine, but the second time causes a double-deletion error. (The main() function is the same as in version 1.)
Version 2 Output.
  Initial values A, B: Joe 12 Mary 10
  After copying B = A: Joe 12 Joe 12
  After setting B.name[0] = 'T': Toe 12 Toe 12
  Normal termination

Version 3 uses shared pointers and performs flawlessly.

Version 3 Output.
  Initial values A, B: Joe 12 Mary 10
  After copying B = A: Joe 12 Joe 12
  After setting B.name[0] = 'T': Toe 12 Toe 12
  Normal termination

• Line 2 defines a short name for the long, awkward type name “—shared_ptr<string>”
• Line 6 declares “name” to be a shared pointer type.
• Line 10 removes the “delete” from the destructor because it is not needed.
• There are no memory leaks, and very little added complexity.
6.1 Storage Class

- **Auto** objects are created by variable and parameter declarations. An object is allocated in the stack frame for the current block when a declaration is executed and deallocated when control leaves that block.

- **Static** objects are created by variable declarations with the “static” qualifier. These are allocated and initialized at load time and exist until the program terminates. They are only visible within the function block or class that declares them. The word “static” comes from “stay”. Static variables stay allocated and stay in the same memory address, even while control leaves and reenters their defining block many times. Note: Static objects might or might not be constants.

- **Dynamic** objects are created by explicit calls on `new`. They continue to exist until they are deleted or until the program ends. All access is provided through a pointer that is returned by new and may be stored anywhere. Eventually, this pointer is used to delete the dynamic object. Any time a dynamic object is created, the programmer must track which class currently has custody over it.

It is common for a single object to have members of two or three storage classes. In Figure 6.1, a declaration for the Family class was instantiated to create an object that is partly auto (the core portion) and partly dynamic (the extensions). The diagram to the right of the declaration shows the object that would be created by the declaration `Family A("Joe", 3)`. The core portion is allocated on the stack; the extensions are allocated in dynamic memory.

The destructor only deallocates one of the two extensions because the dad’s name is inside a string object that has custody over it and will delete it.

```
class Family {
    string dadName;
    int* kidAges = nullptr;
public:
    Family( char* nm, int num  ) : dadName(nm) {
        kidAges = new int[num];
    }
    ~Family() { delete[] kidAges; }
}
```

It is common for a single object to have members of two or three storage classes. In Figure 6.1, a declaration for the Family class was instantiated to create an object that is partly auto (the core portion) and partly dynamic (the extensions). The diagram to the right of the declaration shows the object that would be created by the declaration `Family A("Joe", 3)`. The core portion is allocated on the stack; the extensions are allocated in dynamic memory.

The destructor only deallocates one of the two extensions because the dad’s name is inside a string object that has custody over it and will delete it.

```
class Family {
    string dadName;
    int* kidAges = nullptr;
public:
    Family( char* nm, int num  ) : dadName(nm) {
        kidAges = new int[num];
    }
    ~Family() { delete[] kidAges; }
}
```

6.1.2 Copying vs. Moving

To understand the difference between copying (a default copy or a full copy) and moving an object, one must first understand how the parts of an object can be allocated in three different memory segments, and how those parts interact:

- The core portion includes most of the members of a class. Typically, these are allocated on the run-time stack but if the entire object is allocated dynamically, they may also be in dynamic memory.

- Static data members are used infrequently – most objects do not have them. If there are static class members, they are shared by all instances of a class. To enable this static objects are allocated in a separate static segment of the process. These addresses of static parts are fixed; they do not change during execution. The fixed address of each static part is built into the code at compile time.
• The dynamic extensions are attached to the core object by pointers; they are located in the dynamic heap.

Here is a very simple string-like class, FlexChars, for which the storage grows longer, as needed, to contain strings of any length. Only the class declaration constructor, and destructor are shown.

```cpp
class FlexChars {
private: // ----------------------------------------------------------
    int max; // Current allocation size.
    int n = 0; // Number of array slots that contain data now.
    char* data; // Pointer to dynamic array of char.
    void grow(); // Double the allocation length when called.

public: // ------------------------------------------------------------
    FlexChars( int ss=START ) : max(ss), data(new char[ss]) {}
    ~FlexChars() { delete[] data; }
    ... several public function prototypes have been omitted here...
};
```

The diagram below shows how the parts of a FlexChars class look immediately after the constructor is run. All parts have meaningful and consistent values.

With this diagram, we can discuss the three forms of copying.

**Deep copy.** It means that the core part and every dynamic extension is fully duplicated. This can be programmed using a copy constructor, but it is rarely appropriate. If the original object is large, making a complete copy of all parts wastes a lot of space and time.

This can be programmed using a copy constructor.

**Shallow copy.** This is the default implementation of both assignment and call-by-value. The core portion of the object is copied into the new location, but nothing is done with the extension. Static members are never copied. The diagram shows the results of executing `Str2 = Str1;`

**Moving an Extension.** When an object is moved from one location to another, the core portions are copied but the extensions are removed from the original and installed in the new copy.

This can be programmed using a move constructor or move assignment.
6.2. FIVE SPECIAL FUNCTIONS

6.1.3 Move solves the Copying problems.

Shallow copy is used to implement call-by-value parameter passing. When an object \( A \), is shallow-copied into another object variable \( B \), the two objects share memory extensions. A variety of nasty interactions can happen:

- If, before the copy, a member of \( B \) is a pointer to an extension, there will be a memory leak because that pointer will be overwritten by the corresponding member of \( A \).
- Changes to the data in the extensions of either object affect both.
- Deleting an extension of either object will damage the other. Ultimately, when destructors are run on the two objects, the first will delete properly, the second will cause a double-deletion error.

Shallow copy followed by double delete. The designers of C decided that shallow copy was best in most situations because it is efficient and is normally what is wanted. So call-by-value and assignment were originally defined to use shallow copy.

When C++ was developed, that decision was carried into the new language and copy constructors were introduced to take care of the cases in which deep copy was needed. However, C++ has classes and destructors. Memory management is handled by automatically running destructors, which are responsible for deleting dynamic extensions. Programmers found out rapidly that handling dynamic extensions safely was tricky.

A class has exactly one destructor and there is no choice about whether it will or will not be run when an object goes out of scope. When objects with extensions are passed as arguments to a function, the parameters will go out of scope when the function returns. This results in deleting the shared extension, which will eventually cause program malfunction.

C++ 11 introduced shared pointers, move semantics, r-valued references and a new notation \( X&& \) to address this problem. The goal was to make it efficient to store any type of objects, including pointers, in a vector, while avoiding memory leaks and double deletion. The practical result is that these goals were achieved, but it now takes more knowledge to be a skilled programmer.

6.2 Five Special Functions

By default or by declaration, every class has a single destructor that is automatically applied when an object “dies” by going out of scope or being deleted. If a program does not explicitly define a destructor, a do-nothing destructor will be automatically supplied.

In addition, a class, \( X \), has these five related operations, called special functions. The first three are provided automatically, if needed:

- A null default constructor: \( X() \). If no constructor is defined, a default do-nothing constructor is provided. If a do-nothing constructor AND another constructor are both needed, the default destructor must be explicitly defined (It is not provided in that case).
- A copy constructor: \( X(const X&) \), used to initialize a new object from an older one of the same type. By default, it is a shallow copy, although that can be redefined. The contents of the source are unchanged. The copy constructor is used for call-by-value parameter passing.
- Copy assignment: \( X& \ operator=(const X&) \): Shallow-copy one object into another of the same type. The contents of the source are unchanged. This is used for ordinary assignment operations.
- A move constructor: \( X(X&&) \), used to initialize a new object from an older one of the same type. By default, it is a shallow copy, although that can be redefined. It nulls-out any dynamic extensions of the first object.
- Move assignment: \( X& \ operator=(X&&) \). Moves one object into another of the same type. It nulls-out any dynamic extensions of the first object.

You can redefine all of the special functions, but only a few combinations make sense. Here is a summary of the conditions under which the five special functions are automatically defined:
<table>
<thead>
<tr>
<th>MemberFunction</th>
<th>Implicitly defined if...</th>
<th>Default actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>default constructor</td>
<td>if no other constructor</td>
<td>does nothing</td>
</tr>
<tr>
<td>destructor</td>
<td>if no destructor</td>
<td>does nothing</td>
</tr>
<tr>
<td>copy constructor</td>
<td>if no move constructor and no move assignment</td>
<td>copies all members</td>
</tr>
<tr>
<td>copy assignment</td>
<td>if no move constructor and no move assignment</td>
<td>copies all members</td>
</tr>
<tr>
<td>move constructor</td>
<td>if no destructor, no copy constructor and no copy nor move assignment</td>
<td>moves all members</td>
</tr>
<tr>
<td>move assignment</td>
<td>if no destructor, no copy constructor and no copy nor move assignment</td>
<td>moves all members</td>
</tr>
</tbody>
</table>

When is using the defaults inappropriate? For each class with dynamic extensions, the programmer is supposed to write a destructor to free the extension. Failure to do so leads to “memory leaks”, areas of storage that cannot be freed and reused until the program ends. Once this is done, a guideline called “the rule of three/five” comes into play: If you define a destructor, or a copy constructor (or move constructor) or a copy assignment operator (or move assignment operator) you probably should define all of them.

Why? If you write a destructor, its purpose is normally to free memory or some other resource, in which case the implicitly defined copy constructor and move constructor will most likely not do what you want. The copy constructor might need to make a deep copy and, if not, a move constructor should be available to transfer parts of the original object to the new one.

For this reason, when move constructors were added in C++11, they made the rules so that no move constructor will be generated if you explicitly define a destructor (or any of the other three mentioned above). Ideally, the same rules should also apply to the copy constructor.

The keywords delete and default can be used to define methods.

class X { // These definitions disallow copying.
   X& operator=(const X&) = delete;
   X(const X&) = delete;
};

class Y { // These define the default copy behavior.
   Y& operator=(const X&) = default;
   Y(const X&) = default;
};

6.2.1 Example: Applying the Rule of Five

This is a more complete version of the FlexChars class seen earlier.

Notes on the special function code.

- Line 17 declares and defines a constructor with one parameter. Because that parameter has a default value, this declaration also functions as a null (no-parameters) constructor. All the initialization work is done in ctors.
- Line 18 is a copy constructor, used during call-by-value. We declare that there IS NO copy constructor for this class because copying it creates a shallow-copy problem.
- Line 19 is a move constructor. It declares that a FlexChars CAN be used safely for call-by-value. Again, all the work is done in ctors.
- Line 20 is a definition of ordinary copy-assignment. It declares that a FlexChars CANNOT be assigned to another FlexChars.
- Line 21 is a definition of move-assignment. It defines how to move a FlexChars object safely to another FlexChars object.
- Line 27 is the destructor. It frees the dynamic portion of a FlexChars, as appropriate.
Moving is faster than copying for many types because it does not construct copies of dynamic extensions. This is very important when you design a template because the template parameter could be any primitive or class type.

```cpp
1 // Class declaration for a flexible array of base type char.
2 // A. Fischer, February 7, 2003 file: flexChars.hpp
3 #ifndef FLEXC
4 #define FLEXC
5 #include "tools.hpp"
6 #define START 4 // Default length for initial array.
7
class FlexChars {
private: // -----------------------------------------------
  int max; // Current allocation size.
  int n = 0; // Number of array slots that contain data now.
  char* data; // Pointer to dynamic array of char.
  void grow(); // Double the allocation length.

class FlexChars { // ------------------------------------------
public: // -------------------------------------------
  FlexChars( int ss=START ): max(ss), data(new char[START]) {} 
  FlexChars(const FlexChars& a ) = delete;
  FlexChars(const FlexChars&& a): max(a.max), n(a.n), data(move(a.data)){} 
  FlexChars& operator= (const FlexChars& a) = delete;
  FlexChars& operator= (FlexChars&& a) {
    max = a.max;
    n = a.n;
    data = move(a.data);
    return *this;
  }
  ~FlexChars() { delete[] data; }
  void inputLine( istream& ins );
  int put( char data ); // Store data and return the subscript.
  char& operator[] ( int k );
  int length() { return n; } // Provide read-only access.
  const char* message() { return data; } // Read-only access to data.
  ostream& print(ostream& out) { // Print the filled part, ignore rest.
    data[n] = '\0';
    return out <<data;
  }
  inline ostream& operator<<(ostream& out, FlexChars& F) { 
    return F.print( out );
  }
} #endif
```

**RValue References** The old C supports syntax for two “flavors” of each type. For example, given type `int`, there were variables and pointers: `int` and `int*`. A variable has contents (its RValue) and an address (its LValue). Using `&` before a variable name would produce its LValue, and using `*` after a pointer name would produce the LValue of the variable it points to.

In C, when you write a variable name in an expression, the compiler deduces which one (LValue or RValue) the context calls for and use it.

```
int j, k = 10;
j = k;   // Store RValue of k at LValue of j.
int* pk = k; // Error, type mismatch.
*pk = k+j;  // Store sum of RValues of k and j at LValue of pk
int* pk = &k;  // Store LValue of k at LValue of pk.
```
From its beginning, C++ supported reference types as a way you could declare an identifier, using the \& notation. This gave us int, int*, and int&. Then C++ 11 introduced a fourth syntax, called an RValue reference to support move semantics. Now we have four variants of each type. For integers: int, int*, int&, and int&&. The purpose of this fourth syntax is to provide a clear and simple way to define functions that move things rather than copy them. Using an RValue reference is a signal that you want move semantics. When you assign an object B to another object A, your program moves it if the type of B is RValue reference, and copies it if it is an ordinary reference or a simple variable.

Suppose X is any type, and the function f() returns and object of type X. Then we can write:

X a;
X& r1 = a; // Store the address of a in r1.
X&& rr1 = f(); // Make rr1 an alias for the value of f.
X&& rr2 = a; // Error: a is an L-value, not an RValue.
X&& rr3 = move(r1); // move() returns an RValue reference to its argument.

6.3 Pointers to Functions

Function Pointers in C. In C, function pointers are used to pass functions as parameters to other functions. They are used to tailor a generic function to work on any kind of data, or with any comparison function. The function-pointer supplies a type-specific function for a generic operation. Examples:

- The sort() functions in C and C++ accept a comparison function as a parameter. It must perform an appropriate comparison for the type of the data being sorted.
- On a spreadsheet, when you sort, you can sort in ascending or descending order.
- Graffle, my diagramming app, provides a large menu of functions. So does Excel.
- Some applications provide a menu of functions that a user can call.

Syntax in C Here, we declare a pointer, fp, that could refer to any function with two double parameters and a double result:

double (*fp)( double, double );

In the above declaration, the first pair of parens and the * say that we are declaring a function pointer type. The second pair of parens enclose the parameter list. As an alternative, we could declare a function pointer type using typedef, and use the new type to declare the function pointer:

typedef double (*dub_dub)( double, double );
dub_dub fp;

In the above declaration, the first pair of parens and the * say that we are declaring a function pointer type. The second pair of parens enclose the parameter list, and the word “double” after the keyword “typedef” is the type returned by this kind of function.

Using a function pointer in C We declare pointers fp and gp that can point (respectively) at functions with two or one double parameters and a double result. Note that when you assign to a function pointer, you must not put ( ) after the name of the function. Using ( ) calls a function, and we want to set a pointer to it, not call it.

dub_dub powp = pow;
double (*expp)( double ) = exp;

When you use a function pointer to call a function, you must use the ( ). Both function pointers are initialized in the declarations, above, to functions from <math.h>. Now we use the function pointers to call the functions and print the answers.

printf( "3^5 = %g\n", powp( 3.0, 5 ));
printf( "e^5 = %g\n", expp( 5 ));
Function Pointer Syntax and Example. Everything said, above, about function pointers in C is available and works the same way in C++ with static functions. It does not work with non-static class functions. A non-static class function cannot be passed as a parameter this way because of its implied argument.

Notes on function pointers.

- Line 5 defines a function-pointer type compatible with any function that has two integer parameters and returns an integer result.
- Lines 8, 9 define function-style names for the integer versions of two built-in operators.
- Lines 12–15 define a function with two int parameters and one parameter that is a pointer to a function.
- In main, lines 22 and 23 define two function pointers and initialize them to point at the functions on lines 8 and 9.
- Lines 26 calls the add function indirectly, through the pointer. Line 27 calls subtract.
- Lines 26 and 27 use the preferred syntax for such calls.
- Line 28 uses the alternate syntax for calling a function through a pointer.
- Lines 31 and 33 call the function that has a functional parameter.

This demo program shows how to declare, define, and use function pointers. The output is here, the program follows.

```
fa( 5,3 )= 8
fs( 5,3 )= 2
(*fs)( 5,3 )= 2

op(7, 4, add)= 11
op(20, m, fa)= 9
```

6.4 Functors

Functors A functor class is any class that includes a definition of operator(), the function call operator. A functor is an instance of such a class. The instance is an object that acts like a function. Essentially, instantiating a functor wraps a function inside an object that can be passed as a parameter like any other object.

- The functor class provides a constructor that should initialize any private data members.
- That class must also provide a public definition of operator(), the function call operator.
- The data members of the class act like static local variables for the function.
- Such a definition works like any other operator extension.
- If you declare an object of a functor type and use the name with function-call parens, the definition of operator() will be executed.

Functor Syntax This creates a functor class with no purpose other than wrapping a function in an object.

```cpp
class myFunctorClass {
private:
    int secret;
public:
    myFunctorClass (int x) : secret( x ) {}
    int operator()(int y) { return (secret+y) * (secret+y); }}
```
Using a Functor  The following brief main program instantiates the functor class twice and uses the functors. It does not do anything useful. Notes follow the code.

```cpp
int main() {
    myFunctorClass mystery(5); // "mystery" is a functor.
    cout << "mystery( 6 ) = " << mystery(6) << endl;
    cout << "mystery( 10 ) = " << mystery(10) << endl;
    myFunctorClass mystery2(-2); // "mystery2" is a functor.
    cout << "mystery2( 6 ) = " << mystery2(6) << endl;
    cout << "mystery2( 10 ) = " << mystery2(10) << endl;
    return 0;
}
```

- Lines 2 and 6 instantiate the functor class and, in the process, initialize the encapsulated variable.
- Lines 3 and 4 call the first functor twice. Each call requires an integer argument because the operator definition has one integer parameter.
- Lines 7 and 8 call the second functor twice.
- As demonstrated, both the “secret” value and the parameter in the call affect the outcome.

MyFunctorClass is not a minimal functor class because it encapsulates a member variable, `secret`. A functor class has a default constructor and destructor, the only necessary member is a definition for `operator()`.

However, a functor class can contain other functions as well as those that are required. For example a normal data class, such as `Book` might contain a comparison function for sorting Books by author.
Chapter 7: Important C++ Features

A number of C++ features are introduced in this chapter: default parameters, constructor initializer lists, const class members, static data, constexpr, operator extensions, dynamic memory management, and functors. Skilled programming requires mastery of all these tools. Many have been used in earlier chapters, without a thorough explanation.

7.1 Default Parameters

Purpose and Rules. Default parameters are never necessary but they are a great convenience. They allow us to write a single function definition that defines two or more function variations with different numbers of parameters. A function with one default parameter is really two functions that share a name. (This is sometimes called overloading a name, but please do not use that term in this class.) A function with several default parameters is a shorthand for several function definitions. The basic requirements are:

- A default value for a parameter is given in the class declaration, not the function definition. It is written (like an initializer) after the parameter name in the function prototype.
  
  The default values may then be either given or omitted in the function definitions that are written in the .cpp file. However, if the function definition does give a default parameter value, it must match the one given in the class declaration.
- Some or all of the parameters of any function may have default values.
- All parameters without default values must come before parameters with default values.
- When a function is called, the programmer may omit writing arguments if the default values are acceptable.
- Documentation should always explain clearly which parameters have default values and what the values are.
- Please avoid giving meaningless default values. Write relevant defaults or none at all.

An example. The most familiar function defined with optional arguments is istream::get(). The prototype of istream::get() looks like this:

```cpp
istream& get (char* s, streamsize n, char delim='\n');
```

It can be called to read a string with either three or two arguments\[1\]. The calls below both read characters into an array named input. Both stop after MAX-1 characters have been read. The first call will also stop if a comma is read; the second will stop when newline, the default delimiter, is read.

```cpp
cin.get( input, MAX, ',' );
cin.get( input, MAX );
```

An example with multiple default parameters. Suppose you have created a class called Box. A box has 3 dimensions, so the Box constructor could be declared like this:

```cpp
Box( int ln, int wd, int ht );
```

This constructor allows a programmer to specify boxes of varied shapes and sizes, but if most of the boxes are one size, this is a lot of tedious typing. Suppose, instead, the programmer defines the constructor with defaults that describe the most common shape of box:

```cpp
Box( int ln=9, int wd=4.5, int ht=2 );
```

\[1\] It can also be called with only one argument, but that is very bad practice.
This single definition is equivalent to a family of four function methods:

```
Box( int ln, int wd, int ht );
Box( int ln, int wd );
Box( int ln );
Box( );
```

It allows us to omit the final zero, one, two, or three arguments when calling the constructor:

This is a very nice C++ feature that saves work for the programmer. In Java, there would need to be four different definitions for this constructor.

**Usage.** Optional parameters save time and writing whenever a function must sometimes handle an unusual case. The parameter connected to the unusual situation is coded last and the corresponding argument is omitted from most calls. For example, suppose you normally want a print function to write output to `cout`, but occasionally you want to write to a file. You can accomplish this with an optional parameter.

### 7.2 Constructor Initializers

In C and C++, initialization follows rules that are more permissive than the rules for assignment.

- Values assigned in the class declaration follow the rules for initialization. If a meaningful initial value can be assigned at compile time, it should be done in the class declaration.

- Values assigned in initializer lists (described below) follow the rules for initialization. If a meaningful value can be specified when a function is called, it should be done in the constructor’s initializer list. Also, some actions must be done this way:
  - Initializing a constant.
  - Initializing a reference variable.
  - Calling the constructor of an aggregated class.
  - In a derived class, calling the constructor of the base class.

- Anything written in the body of a constructor function follows the rules for assignment. This can often be avoided by using one of the first two methods.

**Initializer lists.** The term “constructor initializer” is usually shortened to “ctor”. Ctors exist to provide a way to initialize an object during construction, rather than by using assignment later, in the body of the constructor function. The initial value for any non-static data member may be supplied this way and ctors are often used if the initial value is a parameter to the constructor.

To illustrate ctor syntax, imagine that you want to create a handful of identical dice. Various games use different numbers of dice, and the client might want 1, 2, 3, or 4 of them. In addition, a single die might have 4 sides or 6, 8, or 12. Finally, assume that the client can specify both of these properties at run time The class declaration might look like this:

```
class Dice {
  int n;  // Number of dice
  int s;  // Number of sides on each die.
}
```

Here is the constructor using assignment:

```
Dice( int n, int sides ) {
  this->n = n;  // this-> is needed when member name is same as parameter name.
  s = sides;
}
```

Here is the constructor using constructor initializers: easier and maybe clearer. Note that the `this->` is not needed. The member name comes first, followed by the parameter name in parentheses.

```
Dice( int n, int sides ) : n(n), s(sides) {}  
```
Details of the syntax. The ctors are written before the beginning of the body of a constructor, between the closing parenthesis and the opening curly brace. A colon precedes the list of ctors and they are separated by commas. Each ctor is used to initialize one data member of the class. If there is more than one ctor, they should be listed in the same order as the members are listed in the class declaration.

How you write a ctor depends on the type of the member it will initialize.

- If a class member is a non-class type, the name of the member is written followed by its initial value in parentheses. The initial value follows the same syntactic rules as an initializer in a declaration.
- If a data member is a class type, you must supply arguments for the constructor of the component class. To do so, write the member name followed by parentheses that enclose the parameters for its constructor.
- To initialize the base portion of an object in a derived class, the name of the base class is written followed by the arguments for its constructor, in parentheses.

7.3 Constant is not like Static

Const and static are not the same thing, and they are not even similar.

- A const qualifier means that the contents of the variable cannot change.
- A static qualifier means that the storage address cannot change.

It is quite possible for a variable to be both static and const. That means that the variable is always in the same storage location and always has the same value.

7.3.1 The Const Qualifier

Purpose. The const qualifier has multiple purposes:

- Clarity. A good programmer knows whether he is willing to allow a variable to change, or not. A const declaration communicates this intention to the compiler and to other humans.
- Data protection. If an object is declared const, it cannot be changed after creation and initialization is finished. This prevents coding accidents and makes code easier to debug and maintain.
- Documentation. The proper use of const can be a powerful documentation technique. When you use code from a library or from another developer, you often see only the prototypes of the functions in that class, and not the implementation. However, it is important to know that const parameters will not allow the function to change the argument variables named in the call.
- Security. The principle of “least privilege” is important for enterprise security. This carries over into programming: use of const enforces a policy of least privilege.

For these reasons, a professional uses const broadly and consistently. Programs that do not use const are less than professional.

Syntax for const. A const declaration can be used in a variety of contexts. Here are several:

- Within the body of a function, const can be applied to any local variable. This is often done when the variable depends on one of the parameters, but does not change within the function.
- Sometimes global symbols are used to communicate among all classes in an application. In this case, the const must be used, since using global variables is a terrible idea.
- A const data member of a class must be initialized by a ctor and is constant after that. It might be either private or public.
- The return value from a function: The const can be written before the function’s return type. This is used only when the value returned by the function is a pointer (*) or a reference (＆). There are several ways that the const qualifier. ＆ and * can be arranged. Each one means a different thing.
- `const int& f( int x );`
  The return value from f() is a reference to a constant integer. It gives read-access but not write-access to that integer.

- `const int*g( int x );`
  The return value of g() points at a constant integer, possibly part of an array. It can be stored in a non-constant pointer variable. You could increment that pointer to point at the next element in your array of constant ints, but you could not change any of them.

- `int* const h( int x );`
  The return value of h() is a constant pointer to a non-constant int. The int can be changed, but the pointer cannot. You can use the pointer to increment the underlying variable, but you could not what the pointer points at.

- `const int* const p( int x );`
  The return value of p() points at a constant integer. It can be used for read-access to that integer. Neither the pointer nor the underlying integer can be changed.

- **In a function’s parameter list.**
  - The const can be written before the type of some or all of the parameters in the parameter list. This is used only when the parameter is a pointer or a reference. It means that the function cannot modify the argument variable corresponding to the const parameter.
    
    ```
    void List::search( const Item& s );
    void List::search( const string name );
    ```
    Here, we declare that the Search function will not modify the Item (in the first prototype) or the string (in the second prototype) that is its argument. In this case, the const plays an important role in isolating the caller from the actions of the function. (Note: Class objects are usually passed by reference to avoid triggering the class destructor when the function exits.

- The const can be written between the end of the parameter list and the semicolon that ends the prototype or the left-brace that opens the function body. In this position, const means that the function does not modify the implied parameter.
  
  ```
  void MyClass::print() const;
  int MyClass::getSize () const { return num; ;}
  ```
  In this location, the const is used as a form of documentation. It declares that the Print and Space functions use the data members of a MyClass object, but do not modify them. This can be obvious if you look your own function. However, you need this documentation when using library functions.

**Guidance on usage.** Here are some details with a few rules and guidelines:

- A beginner will write a program without `const`. More advanced programmers learn how to work with it.
- Use const to explicitly state your intentions: if your intention is that the variable will not change, then you should define it as a const.
- The most common use of const is to restrict the usage of reference and pointer parameters and return values, so that they give read-only access to data owned by another program unit.
- Another common use is at the beginning of a function, to declare local variables whose value depends on one of the parameters, but whose value will be constant throughout the execution of one call on that function.

Finally, be aware that compilers vary with respect to const. It is the job of the compiler (not the programmer) to make sure that the const rules are never broken. Writing the code that enforces the const declarations is difficult. Some compilers enforce the constant rules very strictly; others do not, so your program might compile without errors on a sloppy compiler but fail on a compiler that checks the const rules carefully.
7.3.2 Constexpr

The constexpr specifier declares that it is possible to evaluate the value of the operator, function or variable at compile time. Such variables and functions can then be used where only compile-time constant expressions are allowed. The full initialization expression, including all implicit conversions, constructor calls, etc, must be a constant expression.

The most common use is to initialize an array member in a class declaration. In this case, using a constexpr is the easy and efficient alternative to writing an initialization loop in the class constructor.

7.4 Static is not like Constant

Static data. Within a class, a static data member is allocated and initialized once, when the program is loaded, and shared by all instances of the class. It appears to be in the class, even though it is allocated in a separate part of memory. Why is this useful?

The purpose of a static class variable is simple: it is used to communicate among or help define all class instances. A static data member is a shared variable that is logically part of all instances of the class (and all instances of all derived classes).

For example, consider a dynamically allocated matrix whose size is not known until run time. It is important to create all matrix rows the same length, but that length cannot be set using \texttt{#define}. A solution is to declare the array length as a private static class member. Then it can be set at run time, but it is visible only within the class. It can be used to construct all class instances, but only occupies space once, not space in every instance.

Sometimes a set of constants is associated with a class, for example, a Month class might have three associated constant arrays, an array of month abbreviations, one of full names, and the number of days in each month. Since these are constant, it is undesirable to keep a copy inside every Month object, and undesirable to keep creating and initializing local arrays. The ideal solution is to make a set of shared (static) arrays within the class.

Applications.

- One can use a static member of a class to count the total number of class instances that have been created. This could be used to produce sequential serial numbers.
- Suppose a programmer does not have access to Valgrind\footnote{Valgrind is a Linux-based tool for memory debugging, memory leak detection.} (or a similar tool). By incrementing a static counter in each constructor and decrementing it in the destructor, he can verify that there are no memory leaks.

Initializing static data. One cannot initialize a static data member in a constructor; it must be done prior to calling the constructor. The best way to do this is in the class declaration, where the static member is declared. For example, suppose a static variable is needed in the Widget class to track the next available serial number. The class declaration might start like this:

```cpp
class Widget {
    private:
        static int sequence = 0;
        int thisPart;
    ...
    public:
        Widget () { thisPart = ++sequence; }
    ...
}
```

Initializing a static array. Traditionally, the initializer for a static array could not be written inside the class declaration, but needed to be put in the corresponding .cpp file. Now we have in-class initializers and constexpr and can do this job is a simple and graceful way.

The initializer for a static array has a slightly more complex syntax. An array initializer is an expression that must be evaluated by the compiler in order to make the results part of the code. The keyword constexpr tells the compiler to evaluate and compile the initializer expression. Example:

### 7.4.1 Static Functions

**Purpose** Functions can be static also, but this is infrequently needed.

- A static function belongs to the class, not the instances of a class. The purpose of a static class function is to allow a client to use the expertise of a class before the first object of that class exists. This is often useful for giving instructions and for validation of the data necessary to create an object.

- A static function can be used as a functional parameter to template-functions such as `sort` and `search`.

All functions defined outside of a class are global static functions.

**Syntax.** To declare a static function inside a class, write `static` before the return type. To call it, use the name of its class and `::`. For example, suppose a class named `Graph` has two static functions, `instructions()` and `bool valid( int n )`. Calls on these static functions might look like this:

    Graph::instructions();
    if (Graph::valid( 15 )) ...

Since a static class function must be usable before the first class object is created, the code within the function cannot use any of the data members of that class. It can use parameters and global constants, however.

### 7.5 C++ Has Four Kinds of Casts

#### 7.5.1 Static Casts

A static cast is an ordinary type conversion. It converts a value of one type to a value with approximately the same meaning in another type. The conversion can be a lengthening (short to long), a shortening (int to char), or a change in representation (float to int). The C and C++ languages support the built-in type conversions shown in Figure 7.1 These are called “static” casts because the compiler finds out that they are needed at compile time and generates unconditional conversion code at that time.

![Figure 7.1: Built-in type conversions in C and C++.

Explicit casts. A static cast can be called explicitly using ordinary C syntax. In addition, C++ has two new ways to call a cast:

    int k, m, *ip;
    float f, *fp;
    f = (float)k; // traditional C syntax.
    f = float(k); // function-call syntax.
    f = static_cast<float>(k); // explicit C++ syntax.

Coercion. Coercion, or automatic type conversion, happens when a function call is encountered, and the type of an argument does not match the declared type of its parameter. In this case, the argument will be converted to the parameter type, if the compiler can find a method for doing so (either built-in or program-defined). Coercion is also used to make operands match the type-requirements of operators. In C, coercion is limited to primitive types. However, in C++, it can also apply to a class type, sayCls:
C++ HAS FOUR KINDS OF CASTS

- If an object of type T is used where a Cls object is needed, and the class Cls contains a constructor with one parameter of type T, the constructor will be used to coerce the T value to a value of type Cls.
- If an object of type Cls is used where a T object is needed, and the class Cls contains a cast operator whose result is type T, the cast operator will be used to coerce the Cls object so that the context makes sense.

7.5.2 Reinterpret Casts

A reinterpret cast is a type trick performed with pointers. It relabels the pointer’s base type without changing any bits of either the pointer or its referent. This is like putting lamb’s clothing on a wolf. Using a reinterpret cast, a program can access a value of one type using a pointer of a different type, without compiler warnings and without changing. (See Figure 7.2) This lets us perform nonsensical operations such as adding incompatible values. For example, a reinterpret cast can let us relabel the integer 987654321 as a float, then add 1.0 to it to produce garbage:

```
987654321
```

The first line of output involves a static cast from int to float. The answer is fine; after adding 654321 + 1.0, we get 65432. However line 17 is the result of a reinterpret cast. It shows an answer that is garbage. That happens because the large int number becomes a very small float value when we just reinterpret the bits as a float. The integer value and the floating point 1.0 were added to each other without any type conversion. The primary applications for reinterpret casts are hash functions and input conversion functions like strtod() and strtol().

**Alternative syntax.** Ordinary C syntax or C++ syntax with angle brackets can be used to invoke a reinterpret cast:

```
fpu = (float*)p_int // ordinary C/C++ syntax.
fpu = reinterpret_cast<float*>(ip); // explicit C++ syntax.
```

How does the compiler know which kind of cast to use? If the argument to the cast is a pointer, it uses a reinterpret cast. If it is NOT a pointer, it uses a static cast, if one exists.
7.5.3 Const Casts

A const cast provides a way to remove the const property from a pointer variable just long enough to change the value of its referent. This lets us use a constructor (or any class function) instead of a ctor to initialize a const class member.

```cpp
#include <iostream>

using namespace std;

int main( void ) {
  int w = 99;
  const int* cip = &w;
  cout << " &w is " << &w << " referent of cip is " << cip << endl;
  cout << " w= " << w << " *cip= " << *cip << endl;
  * const_cast<int*>(cip) = 33;
  // *cip = 33;
  cout << " w= " << w << " *cip= " << *cip << endl;
}

/* Output: -------------------------------------------
 &w is 0x7ff7b358350c referent of cip is 0x7ff7b358350c
 w= 99 *cip= 99
 w= 33 *cip= 33

If I remove the comment marks on line 11, the compiler says:
constCast.cpp:11:7: error: read-only variable is not assignable
*/
```

When we write `*cip = 33;` without a const cast, we get a const violation error:

```
const.cpp: In function ‘int main ()’:
const.cpp:145: assignment of read-only location
```

With a const cast, we are permitted to change the location and we get output:

```
&w is 0xbffff714 referent of cip is 0xbffff714
w= 99 *cip= 99
w= 33 *cip= 33
```

7.5.4 Dynamic Casts

Dynamic casts are used with polymorphic classes, and can cast either pointers or references. We have not yet covered derivation, polymorphism, and type hierarchies, so details of the dynamic casts cannot be appreciated yet. We will return to this topic in a later chapter.

7.6 Operator Extensions

**Purpose.** The built-in arithmetic operators in C and C++ are generic; they are defined for all built-in numeric types. One of the purposes of an object-oriented language is to enable the programmer to define new classes and use them in the same ways that the built-in classes are used. Combining these two facts, we see a need to add new methods to existing functions.

Traditionally, the word “overload” has been used to describe these added methods. However, the term is overly-general and leads to confusion. There are actually three very different kinds of method definitions.

1. **Overload.** A new method overloads an operator by giving it a new meaning that is only distantly related to its original meaning. For example, the `+` operator is often overloaded for strings, to mean string concatenation. The `<<` operator for output is an overload of its original meaning of shifting bits.

2. **Extend** An operator extension implements a method for a new type that preserves the intention and character of the original operator. For example, any method named `print()` should print the contents of an object without changing it, and any extension of `+` should do an operation that a math teacher would call “addition”.

3. **Override.** A third kind of operator definition can only happen in a derived class. A method operator *overrides* an inherited method if it has the same name as the inherited implementation. For example, the print method defined in a derived class sometimes overrides the method defined in its base class. This happens when the base-class method was called using a pointer to a derived-class object. When derivation is used to create a class hierarchy, functions defined in the parent class are often redefined in the derived class to achieve results that are specific to the derived class.

These distinctions do matter when trying to give advice. For example: Overloads should be used rarely because they can be very confusing, but extensions can be used freely because they avoid, not add, confusion. Overrides are common with polymorphic types.

### 7.6.1 Syntax

A general rule for operator extensions is that the number of operands and precedence must match the built-in operator. Default arguments are illegal in these definitions.

C/C++ has both binary and unary operators. When a unary operator (such as increment or type cast) is extended, the definition is always given as part of the relevant class. When a binary operator is extended, the definition is normally given as part of the class of the left-side operand. If that is not possible, it is given globally (outside all classes).

Below, we consider the varied contexts in which an operator definition might make sense and give one or more examples of each.

**Global Binary Operator Extensions**

These are appropriate for extensions of an operator to a non-class type, or for a pre-defined class that cannot be modified. The syntax for this kind of extension prototype is:

\[
\text{return-type operator op-symbol (left-op-type, right-op-type );}
\]

The example programs in this text all use extensions of the `<<` operator. These are defined globally because we cannot add them to the predefined ostream class. The right operand (second parameter) belongs to the new class in each case. These definitions are commonly inline.

\[
\text{inline ostream& operator << ( ostream& out, Item& T)\{ <code> \}}
\]

\[
\text{inline ostream& operator << ( ostream& out, Stack& S)\{ <code> \}}
\]

The preferred way to write this kind of extension is as an *inline* function just below the bottom of the class declaration. If the function is too long for inline expansion, then the prototype should be given at the bottom of the .hpp file for the class, and the function definition should be in the main module, after the `#include` command for the class header and before the beginning of main().

**Binary Operator Extensions in a Class.**

Binary functions within a class are defined using this prototype syntax:

\[
\text{return-type operator op-symbol ( right-op-type );}
\]

**Extending arithmetic and comparison.** Suppose we defined a new numeric class such as `Complex`. It would be appropriate to extend all the basic arithmetic and comparison operators for this class, so that complex numbers could be used, like reals, in formulas. Assume that the class `Complex` has two data members, `rp` (the real part) and `ip` (the imaginary part). Several definitions are required to define all of the relevant operators; here are two samples:

\[
\text{Complex operator + ( Complex plex )\{ // Add two complex numbers.}
\]

\[
\text{\hspace{1cm}\hspace{1cm}return Complex( rp + plex.rp, ip + plex.ip );}
\]

\[
\text{\}}
\]

\[
\text{bool operator == ( Complex plex )\{ // Compare two complex numbers.}
\]

\[
\text{\hspace{1cm}\hspace{1cm}return rp == plex.rp && ip == plex.ip;}
\]

\[
\text{\}}
\]
In such definitions, the implied argument is the left operand and the explicit argument is the right operand. Any reference to the left operand is made simply by using the part name. References to the right operand are made using the parameter name and the part name.

**Adding to a collection: an overload.** An might be += operator in a container class. The Stack::push() function in the Trains example from Chapter 4 could be replaced by the following extension. (Only the function name would change, not the function’s code.)

```cpp
void Stack::operator += ( char c ); // Push an Item onto the Stack.
```

The Stack::push function is called from three loops in the main program. The first loop could be replaced by this intuitively appealing code:

```cpp
do { cin >> car;
    east += car;
} while (car != QUIT); // Caboose has been entered.
```

Is this kind of overload a good idea? Many people would think that using push is clearer. For them the overload is a poor idea.

**Assignment: an override.** Sometimes the default definition of assignment is changed. For example, here is a definition of = for Stacks that moves the data from one stack to another, rather than doing a shallow copy. This definition illustrates the use of the keyword this to call the += function (defined above) to push a character onto the implied Stack parameter:

```cpp
void operator= ( Stack& z ) { // Pop one stack, push result onto other.
    char c;
    while (!z.empty()) {
        c = z.pop();
        *this += c;
    }
}
```

Is it a good idea to modify assignment? This particular override decreases efficiency without adding clarity. However, sometimes it could be useful to redefine assignment.

**Extending subscript.** The remaining important use of operator extensions is to define subscript for data structures that function as arrays but have a more complex representation. A good example is vector or any similar resizeable array data structure. A flexible array class needs an extension of the subscript operator so that the client program can refer to the object as if it were a normal array. The subscript operator is a little different in three ways.

- It must return a reference to the selected array slot so that the client program can either read or write the selected element, giving us the first definition below.
- However, sometimes that reference should be read-only, giving us the second definition below. A class would define one or the other, never both.
- The syntax for calling subscript is a little different because the left operand is enclosed between the square brackets, not written after them.
- An added advantage of extending subscript is that a bounds check can be incorporated into the definition, as illustrated below.

Here is the part of the class declaration and the extension of the subscript operator that could be part of a FlexString class (a resizeable array of characters).

```cpp
class FlexString {
private:
    int Max;       // Current allocation size.
    int N;         // Number of data chars stored in array.
    char* Buf;     // Pointer to dynamic array of char.
```
7.6. OPERATOR EXTENSIONS

public:
    char& operator[]( int k ); // Access to read and write.
    const char& operator[]( int k ); // Read-only access.
    ...
};

// ----------------------------------- access the kth char in the string.
char& FlexString::operator[]( int k ) {
    if (k >= N) fatal(" FlexString bounds error.");
    return Buf[k];
}

7.6.2 Unary operators.

Extending type cast. The syntax for extending the type cast operator is:

    operator target-type() { return member-name; }

An extension of the type cast operator is used to convert a value from a class type to another type. This is also called “member conversion”, because it converts a class member to some other type. It is potentially useful in linked-list classes such as Cell (below) for converting from a Cell to a pointer type such as Item*, by extracting the pointer from the Cell.

    class Cell {
        private:
            Item* Data;
            Cell* Next;
        public:
            operator Item() { return *Data; } // Type cast from Cell to Item
            ...
    };

Type coercion. Extensions of a type cast operator can be used explicitly, but they are also used by the system to coerce operands and parameters. For example, in a program that contained the above extension of type cast, if you used a Cell object in a context in which a Cell* was needed, the compiler would use the Next field of the object to carry out the operation. Beware: Can is not the same as should. To someone who is unfamiliar with what is going on, this behavior is hopelessly confusing.

Increment and decrement: extensions or overloads. The Prefix forms of increment and decrement are normal unary operators; The postfix forms are different from everything else in the language. The prototypes for these operators have the form:

    Preincrement:    return-type operator ++();
    Postincrement:  return-type operator ++(int)();

The “int” after the ++ in the postfix form serves only one purpose, and that is to distinguish the prefix and postfix forms. The “(int)” looks like a parameter list, but the parameter does not need a name because it is not used; it is just a dummy. Definitions of these two operators inside the complex class might look like this:

    Complex operator ++ (){ rp++; return *this; }
    Complex operator ++ (int){ Complex temp = *this; ++rp; return temp; }

Another use of an increment extension is to make the syntax for a linked list imitate the syntax for an array, so that the exact nature of a class implementation can be fully hidden. This technique is used extensively in implementations of container classes in the Standard Template Library, and can be applied in any list class that has a “current element” member. Here is an appropriate extension for a List class that has members named Head, Curr and Prior:

    Cell* operator ++ (){ Curr = Curr->Next; return Curr; }
    Cell* operator ++ (int){ Cell* tmp = Curr; Curr = Curr->Next; return tmp; }
Given this definition and an appropriate extension for <= in the Item class, we could write a List::Search function as follows:

```cpp
bool List::Search( Item T ){  // Search for T in list with ascending order.
    for (Curr= Head; Curr!=nullptr; )
        if (T <= Curr->Data) break;
    Prior = Curr++;
}
return T == Curr->Data  // true for exact match, false for >.
```

### 7.6.3 Static Operators inside a Class

When working with some standard packages, such as std::sort() in the template algorithms package, we sometimes need to pass an operator or function as a parameter to another function. In such cases, the operator must have the exact prototype declared for the standard algorithm.

For example, std::sort() has two methods. Both sort an array of arbitrary type, BT and both require parameters that supply the beginning and end positions of the array to be sorted. The two-parameter sort method uses the operator<, which must be defined for class BT, using the normal syntax for defining a binary operator inside a class.

The three-parameter method for std::sort() requires a non-class global functional parameter which must have exactly the prototype. The function must compare the two BT objects and return a true or false result:

```cpp
bool funcname ( const BT& a, const BT& b );
```

Clearly, since the comparison operator must use the private parts of the BT object, it should be inside the class. However, a normal function defined inside the class has an implied argument and must be called using a class object. For std::sort(), this does not work.

The conflict can be addressed in various ways; the right way is to use a function (future chapter). The least-ugly other solution is to define a static class function with two parameters that matches the required prototype. Because the function is static, it lacks an implied parameter, and thus, satisfies the needs of std::sort().

Example: In this class, we define a function that will be used to sort an array of Freq objects in descending order, according to the pct member.

```cpp
class Freq {
private:
    char letter;
    double pct;
public:
    Freq( char c=' ', double d=0.0 ) { letter = c; pct = d; }
    ~Freq(){};
    void print (ostream& out) { out <<letter <<setw( 7 ) <<pct; }
    bool operator< (const Freq fr) const { return pct < fr.pct; }  // asdsceding order
    static bool descend( const Freq& f1, const Freq& f2 ){ return f1.pct > f2.pct; }
};
```

To call this function or pass it as a parameter to another function, use its full name, Freq::descend:

```cpp
Freq array[100];
int n;  // The number of data items actually stored in the array.
bool result = Freq::descend( array[1], array[2] );
std::sort( array, array+n, Freq::descend );
```

### 7.7 Complex Arithmetic: an Example of Operator Extensions

This section presents examples of several operator definitions. The Compl class implements basic arithmetic on complex numbers. This is totally unnecessary in C++ because a complex type and its operations are built into the language. However, it is a simple and instructive example of the syntax and usage of operator extensions.
The three code files are a declaration for the class Compl (Compl.hpp), the definitions of three of the class functions (Compl.cpp) and a main function that exercises the operations defined by the class. Note that, in C++, assignment operators such as = and += must return the address to which a value was assigned, not the value itself.

Notes on the main program. The purpose of this example program is to show how operators can be extended. Six operators are defined in the Compl class and called from main.

- operator -, subtract the real parts and the imaginary parts
- operator +, add both the real parts and the imaginary parts
- operator +=, add two complex numbers and store in left-hand-side
- operator *, \((a+bi)(c+di) = (ac - bd) + (ad + bc)i\)
- operator [], for subscript 0 return real part, for 1 return imaginary part

Assignment is also called, but is predefined for every type.

The main program.

```cpp
#include "tools.hpp"
#include "Compl.hpp"

int main(){
    banner();
    Compl cx1(1,2); // 1+2i
    Compl cx2(3,5); // 3+5i
    Compl cx3; // 0+0i
    cout << "cx1 is " <<cx1 <<endl;
    cout << "cx2 is " <<cx2 <<endl;
    cout << "cx3 is zero (default) " <<cx3 <<endl;
    cx3 = Compl(-1, 5);
    cout << "cx3 is now " <<cx3 <<endl;
    cx3 = cx1*cx3; // Should be -11+3i
    cout << "after cx1 = cx1*cx3, cx3 is " <<cx3 <<endl;
    cx1 = cx2;
    cout << "after cx1 = cx2, cx1 is " <<cx1 <<endl;
    cx1 = cx2-cx1;
    cout << "cx2-cx1 is " <<(cx2-cx1) <<endl;
    cx1 += cx3;
    cout << "after cx1+=cx3, cx1 is " <<cx1 <<endl;
    cout << "cx1[0]:real part is " <<cx1[0] <<endl;
    cout << "cx1[1]:imaginary part is " <<cx1[1] <<endl;
    bye();
}
```

Declaration of Compl.

```cpp
class Compl{
private:
    double real = 0;
    double imag = 0;
```
public:
Compl(float rl, float im) : real(rl), imag(im) {}
Compl()=default;
"Compl()=default;
ostream& print(ostream& out){return out <<"("<<real<<"+"<<imag<<"i")");}
// Compl operator =(Compl cx){ real=cx.real; imag=cx.imag; return *this; }
Compl operator +(Compl cx){ return Compl(real+cx.real, imag+cx.imag);}
Compl operator -(Compl cx){ return Compl(real-cx.real, imag-cx.imag);}
Compl operator +=(Compl cx);
Compl operator *(Compl cx);
double operator[](int which); // Select one part or the other.
inline ostream& operator <<(ostream& out,Compl cx){ return cx.print(out); }

Notes on the Compl Declaration.
• The print function displays complex numbers in the conventional manner, e.g. 3 + 1.5i.
• The operator = definition is commented out because it is unnecessary. The default definition copies the
  real part to real part and the imaginary part to imaginary part, as this does.
• On line 14, the assignment operator returns *this; this is a pointer to the implied argument, which
  becomes the left-hand operand. So *this IS the implied argument, and is returned as required.
• Lines 47 and 48 give inline definitions of + and −.
• The last three methodss are too long and complex for one line, so they are defined in Compl.cpp.

Definition of three Compl functions.
//-------------------------------------------------------------------------
// Operator definition practice  A. Fischer, June 2023 Compl.cpp
#include "Compl.hpp"
Compl
Compl::operator+=(Compl cx){
  real = real+cx.real;
  imag = imag+cx.imag;
  return *this;
}
//-------------------------------------------------------------------------
Compl
Compl::operator*(Compl cx){
  return Compl((real*cx.real - imag*cx.imag),(real*cx.imag + imag*cx.real));
}
//-------------------------------------------------------------------------
double
Compl::operator[](int which){
  if(which == 0) return real;
  else if((which == 1) return imag;
  else fatal("input out of range");
  return 0;
}

• These three functions are defined here because they are too long to fit on one line. Writing them this way
  makes the math clearer.
• Operator += adds the right-hand side to the complex number on the left. The computation must be done
  part by part. Then, just like operator =, it must return the location in which the result was stored. So
  line 62 returns *this.
• Lines 65–68 implement the ordinary mathematical definition of multiply for complex numbers. Please
  note the asymmetry between references the this and to cx, the argument.
• Lines 70–76 implement subscript with a bounds-check that traps illegal subscripts. Using subscript in this
  way is unusual, here, but a good example.
The output. The main program, on the next page, calls every class function. Its output is:

cx1 is (1+2i)
cx2 is (3+5i)
cx3 is zero (default) (0+0i)
cx3 is now (-1+5i)
after cx3 = cx1*cx3, cx3 is (-11+3i)
after cx1 = cx2, cx1 is (3+5i)
cx2-cx1 is (3+5i)
after cx1+=cx3, cx1 is (-11+3i)
cx1[0]:real part is -11

cx1[1]:imaginary part is 3

Normal termination.

7.8 Exceptions

From the board game MONOPOLY, the rule to follow when your man lands on the “illegal” square:

Go to jail. Go directly to jail, do not pass GO and do not collect $200.

Function calls, loops, conditionals, switches, and breaks permit the programmer to control and direct the sequence of evaluation of a program and to modify the sequential default order of execution. These statements permit controlled, local perturbations in the order of execution. Almost all situations that arise in programming can be handled well using some combination of them.

The goto statement is also supported in C, but its use is discouraged because it makes uncontrolled, non-local changes in the execution sequence. Use of a goto is even worse in C++. If it is used to jump around the normal block entry or exit code, the normal construction, initialization, and deletion of objects can be short-circuited, potentially leaving the memory in an inconsistent state. Therefore, this statement should simply not be used.

But there are situations in which the ordinary structured control statements do not work well. These involve unusual situations, often caused by hardware or input errors, which prevent the program from making further fruitful progress. For these purposes, exceptions were introduced. In C++, a failure during execution of a constructor is one situation that calls for use of exceptions.

In the old days, these situations would be handled by a variety of strategies:

1. Do nothing. Don’t check for errors; hope they don’t happen. Let the program crash or produce garbage answers if an error happens. This is irresponsible and appropriate only for beginners.

2. The program could identify the error and call an error function. However, this just defers the problem instead of solving it. The error function still must do something about the error.

3. Identify the error, print an error comment, and call exit(). (This is equivalent to using the fatal() function in the tools library.) However, aborting execution is not permissible in many real-life situations. For example, aborting execution of a program that handles customer accounts could leave those accounts in an inconsistent or incorrect state.

4. One could use assert() to check for errors. This is similar to option (3) but worse because it gives no opportunity to print out information about the error or its cause.

5. The function being executed could return with an error code. The function that called it would need to check for that error code and return to its caller with an error code, and so on, until control returned to the level at which the error could be handled.

This method is as old as C. It usually works but distorts the logic of the program and clutters it with a large amount of code that is irrelevant 99.9% of the time. Using this technique discourages use of functions and modular code.
6. A long-distance goto could be used to return to the top level. Using this strategy, any information about the error would have to be stored in global variables before executing the goto. This is like programming in BASIC. Use of this control pattern destroys the modularity that C/C++ programs can otherwise achieve. It is not recommended.

7.8.1 What Exceptions Can Do

An exception is an unusual error situation for which the ordinary structured control statements do not work well and aborting execution is undesirable. An exception system provides an additional control option that is designed for dealing with such emergencies.

In C++, a program activates the exception system by surrounding a block of code in a try block. At its end are a series of catch blocks, each one defining how a particular kind of exceptions try block. At its end are a series of catch blocks, each one defining how a particular kind of exceptions should be handled. Exceptions can, but should not, be used for non-error conditions such as end-of-file. They should be used anywhere it is important to avoid aborting the program.

An exception system lets control go . . .
- From the level at which an error is discovered, often deep within the code, at a stage when several functions have been called and have not yet returned . . .
- Carrying as much information as necessary about the error . . .
- To the level at which the error can be handled, often in the application controller or the main function. Handling an exception means displaying user information, cleaning up the contents of major data structures, cleaning garbage out of the input file, and re-initializing everything to prepare for returning to execution.
- Sometimes it is possible for control to return to normal execution.

Exception declarations. Unlike Java, C++ does not expect or encourage the programmer to declare the list of exceptions that a function might throw. Such specifications are tolerated, to a limited extent, but are deprecated by the current standard because they just cause problems.

Exception objects. An exception handler and the class of exceptions it can handle are defined at a high level of a program – often in the application controller or in main. When an exception condition is identified at a deeper level, an exception object is created and “thrown” upward. This causes immediate exit from the function that discovered the error, and the function that called it... etc. The stack frames for all those functions are deleted and the destructors are run for all objects in them. This continues until some function, higher in the chain-of-command, catches the exception. The catching function then “has the ball” and can either catch the exception and handle it or rethrow it. Failing to catch an exception that is coming up the stack (by ignoring it) causes an immediate return of any function to its caller. Thus, control passes backward along the chain of function calls until some function in the chain-of-command “catches” the exception, or until the main function is terminated. All stack frames in this chain are deleted and the relevant destructors are run.

Exceptions can come in many types and each type of exception must have a matching handler. An exception object may have as many fields as necessary to contain all the important facts about where and why the exception was thrown. This object must outlive the function that created it (exceptions are created in dynamic storage, not on the stack) and it must carry an identifying type-tag, so it can be matched to the appropriate exception catcher and exception handler.

What next? After catching an exception, there are several options for handling it:

1. The catcher can abort the process or return to its caller with an error code, as in C. However, in this case, fatal() (not throw) should probably have been called by the originating function.
2. The catcher can clean up the data and return with some appropriate value.
3. The catcher can clean up the data and call its containing function recursively.
4. The catcher can clean up the data and continue from the line that follows the exception handler (not the line that follows the function call that caused the exception).

5. The catcher can comment, get more information from the operator, and carry on in one of the preceding ways.

6. The catcher can fix or add some data fields and rethrow the same exception or some other exception to a higher authority.

Throwing an exception can cause memory management problems when an object with dynamic extensions is half-constructed.

### 7.8.2 Built-in Exceptions

Simple objects like integers and strings can be thrown and caught. In addition, C++ has about a dozen built-in exceptions in the standard library. The base class for all of them is `<exception>`. The `exception` class also has several types and utilities to assist handling exceptions. The names of the most useful predefined exceptions follow.

**bad_alloc.**
One of these, `bad_alloc` changes the way we write programs. In C, it was necessary to check the result of every call on `malloc()`, to find out whether the system was able to fulfill the request. In C++, such a check is not necessary or helpful. If there is an allocation failure, the run-time system will throw a `bad_alloc` exception and control will not return to the failed call. If your program does not have a handler for such exceptions, and one occurs, the program will be terminated.

**bad_cast.**
The `bad_cast` exception is used when a program executes an invalid downward dynamic cast, and `bad_exception` is thrown when a function violates its own exception specification.

**runtime_error.**
This group of exceptions (`range_error`, `overflow_error`, `underflow_error`, and `system_error`) are all beyond the programmer’s control and are used for mistakes in library functions or the run-time system.

**logic_error.**
Another group of exceptions are defined by the standard but, so far as I know, not used by the system. They seem to be intended for use by any program when an avoidable error is discovered. These include `domain_error`, `invalid_argument`, `length_error`, and `out_of_range`.

**Summary**
Every exception you throw, or need to catch, requires a handler of some sort. The above exceptions can be caught and handled individually or generically. For example, you might catch `bad_alloc` explicitly, but you might catch all the runtime errors generically, by catching `runtime_error`. After the catch, they would all be processed uniformly.

In the hands of an expert, exception handlers can greatly simplify error handling in a large application. However, they should not be used if normal control statements can do the task because exception are difficult for a compiler to handle and force the compiler to interact with the host system in awkward ways.

### 7.8.3 Example: Defining and Using an Exception Class

Exceptions are intended for global, not local, use. Their proper use is to enable control and information to pass across classes and through a chain of function calls. They are very useful, when needed, but they are not often needed. One major principle applies to the use of exceptions: they are not a substitute for proper use of `else` or `while`. If a problem can be fully handled in the function that identifies it, DO NOT use an exception. The thrower is not usually the catcher it. The function that throws an exception normally throws it to some other function that is higher up in the chain of control.
An exception type is defined like any other class. It can have public, protected, and private parts, but usually the data parts are all public, since they exist to be read, or used, by an arbitrary catcher. The exception class definition can be (and often is) global or it can be contained within the definition of another class.

A common source of errors is interactive user input, and a common input pattern is for a controller to read one data set, then send it to a validation function. The input loop is embedded in a try block that ends with a set of catch clauses that handle possible errors. This pattern is illustrated by the example below.

The data class declaration.

```cpp
//=============================================================
// A playing card class and related exception classes.
// Alice E. Fischer, July 2023 cards.hpp
//=============================================================
#pragma once
#include "tools.hpp"
#include "bad.hpp"
enum class SuitType { Spades, Hearts, Diamonds, Clubs, Bad };
//============================================================
// The main data class; it represents one playing card.
class Card {
  int   spot;
  SuitType suit;
  void translate(char inspot, char insuit);
  static const string spotlabels[16];
  static const string suitlabels[5];
  public:
    Card () =default;
    Card (istream& sin) {
      char inspot, insuit;
      sin >> inspot >> insuit;
      if (!sin.good()) fatal( "Low level read error
      translate(inspot, insuit);
      }
    ostream& print(ostream&);
    static void instructions( int n );
};
```

- Lines 12–13: A playing card has two properties: a spot value and a suit. These will be entered from the keyboard as chars and translated to an internal format for easy processing.
- Line 14 translates the two inputs. The spot value is translated to an integer between 1 and 13 with 0 used to represent an invalid input. The suit value is translated to an enum constant to represent one of the four suits or an invalid input. This function is private because only the Card class should be doing this translation step.
- Lines 15–16 will be used to print thigh-quality output for the user. Users do not want to read encoded things – they want to read English. These arrays are private because they are used only by print functions in this class.
- Lines 15–16 are static because the string arrays are not small objects, but are needed with every Card object. Making them private static allows them to be available ONLY to the Card class, and does so without taking up large hunks of memory in each Card instance.
- Line 18: We need a default constructor because the main function creates an array of Cards.
- Lines 19–24: The Card constructor reads and parses the data for one card. Then it calls the translate() function to validate the data and convert it to the internal representation.
- Line 22 calls fatal to end the dysfunctional program. It could throw a string exception, but why bother? That simply defers the issue of inability to move forward. It solves nothing.
- Line 26 is a static function that is called at program startup to provide instructions for the user.
The data class implementation.

```cpp
// Functions and constants for the Card class.
// Alice E. Fischer, July 2023 cards.cpp
#include "cards.hpp"

const string Card::suitlabels[5] = {"spades", "hearts", "diamonds", "clubs", "bad"};

void Card::instructions(int n) {
    cout << "Please enter " << n << " cards."
    << "Spot codes are 2...9, T, J, Q, K, A \n"
    << "Suit codes are S H D C \n";
}

void Card::translate(char inspot, char insuit) { // might throw (BadCard)
    if (inspot >= '2' && inspot <= '9') spot = inspot - '0';
    else switch( toupper(inspot) ){
    case 'T': spot = 10; break;
    case 'J': spot = 11; break;
    case 'Q': spot = 12; break;
    case 'K': spot = 13; break;
    case 'A': spot = 1; break;
    default : spot = 0;
    }
    switch( toupper(insuit) ){
    case 'S': suit = SuitType::Spades; break;
    case 'H': suit = SuitType::Hearts; break;
    case 'D': suit = SuitType::Diamonds; break;
    case 'C': suit = SuitType::Clubs; break;
    default : suit = SuitType::Bad;
    }
    if (spot==0 || suit==SuitType::Bad) throw BadCard(inspot, insuit);
}

ostream& Card::print(ostream& sout) {
    return sout << spotlabels[spot] << of " << suitlabels[(int)suit] << endl;
}
```

- Lines 33-34 are static initializers for the two static arrays of strings defined in the Card class. The print-strings for suits are parallel to the symbols in the enum.
- Lines 37–41 define the static function declared on line 26. The purpose for a static function is to be IN the class it relates to, yet available to use BEFORE the class is instantiated.
- Lines 45–52 translate the single-character input for the spot value to the integer representation. It is worth the small amount of extra effort to provide a convenient interface for users. Single-character inputs are the easiest for the user to enter correctly.
- Lines 54—60 translate the single-character inputs for the suit value to the enum constants. Enum constants are often easier to work with than strings, and they make programs easier to read and comprehend. Again, they are worth the bother. The “SuitType:: is needed because the enum declaration is inside the Card class. If it were a simple enum, not an enum class, and it were outside the class declaration, the SuitType:: would not be needed. Simple enums are marginally easier but are no longer considered good style.
- Line 61 uses the throw statement to construct an exception object and send it up the stack. Throwing an exception causes control to pass backwards through the chain of function calls to the nearest previous catch clause that handles that particular type of exception. The destructors will be run for all objects in the stack frames between the throw and the catch.
• Because this handler is virtual, control will pass immediately to line 38 if a BadSuit was actually thrown, or to line 49 if BadSpot was thrown. This is how we get three different actions from one catch clause.

• Line 66: A Card is printed in English, using the int encoding of the spot value and the enum encoding of the suit. The enum must be explicitly cast to type int to use it as a subscript.

The exception class.

```cpp
68 //===============================================================================================
69 // Exception Class for Playing Card Errors. file: bad.hpp
70 // Exception demonstration program: July 3, 2023
71 // **************************************************************************
72 #pragma once
73 #include "tools.hpp"
74 // **************************************************************************
75 class BadCard {
76 public:
77    char spot;
78    char suit;
79    //------------------------------------------------------
80    BadCard (char n, char s) : spot(n), suit(s) {}
81    "BadCard() =default;
82    void print(){
83        cerr <<" Spot value or suit or both are wrong\n"
84        <<" Legal spot values are 2..9, T, J, Q, K, A\n"
85        <<" Legal suits are H D C S\n";
86        cerr <<" You entered "<<spot <<" of " <<suit
87        <<". Please reenter. \n";
88     }
89  }
```

Notes on the exception class. This is a simple class that is used when erroneous values are read from the keyboard while constructing a playing card. Each card is input as two chars, representing the face value and the suit of the card. When an error happens, we need to let the user know what he typed and what he should have typed.

• Lines 77-78: The original input chars are stored in the exception for later use by the exception handler.

• Line 80: The constructor takes the input chars and stores them in the exception.

• Lines 82–88: print() provides good-quality feedback for the user and prompts for reentry of the intended data. Feedback that distinguished between spot errors and suit errors would be better.

Try blocks and catchers. Code that may generate exceptions (at any nesting level) and wishes to catch those exceptions must be enclosed in a try block (lines 105.–111). The exception handlers are written in catch blocks that immediately follow the try block (lines 113..117).

• The fields inside an exception object may be used according to the normal rules of public and/or private access. The data can be public because there is no need to protect it.

• If the handler does not need to access the information in the exception, the parameter name may be omitted.

• The order in which the handlers are written is important; the general case must come after all related specific cases.

• An exception handler whose parameter is . . . will catch all exceptions. This is primarily useful in programs that must not crash.
The main program.

```cpp
#include "tools.hpp"
#include "cards.hpp"
#include "bad.hpp"
#define NCARDS 3

int main( void ) {
  int k;
  Card hand[NCARDS];
  Card::instructions( NCARDS );
  //-------------------------- Main loop that reads N cards.
  for (k=0; k<NCARDS; ){ // NO INCREMENT here; later if card is good.
    try {
      cout << "Enter card # " <<k+1 <<" (spot-code suit-code): ";
      hand[k] = Card(cin); //---------------------- Input one card.
      cout << " Card successfully entered into hand: ";
      hand[k].print(cout);
      ++k;
    }
  //----------------- Check for the three application-specific exceptions.
    catch(BadCard& bs){ bs.print(); } // Exception handler.
  //------------------- Now check for general exceptions thrown by system.
    catch (...) { //--------------- Catch everything else.
      fatal( "\nLast-ditch effort to catch exceptions.\n" );
    }
  // Control comes here after the try/catch is finished. -----------------
}
```

Notes on the main program. Almost the entire program is a try block and 2 catchers.

In this example, line 150 calls the Card constructor, which can throw exceptions from the Bad class and Bad’s derived classes. When that happens, it is undesirable to execute lines 151–154, so they are also in the try block. The handler for the Bad exceptions is on line 157. This line will catch all three kinds of Bad exceptions and process them by calling the virtual print function in the Bad class.

- Lines 104–119 are an input loop. Lines 120–121 process the data.
- Lines 105–117, the entire body of the loop, are a try block and 2 catchers.
- Understanding line 107 is critically important. This line calls the Card constructor to read data for one card from the keyboard. Control goes from the constructor to one of two places:
  - It returns to line 107 if there were no exceptions (that is, no input errors or validation errors).
  - OR it goes to line 113 because an exception has been thrown.
- Lines 109–110 are executed only if the input was correct and a good Card was stored in the hand. Note that the loop variable is incremented on line 110, instead of line 104. This is a key to getting correct results when re-entering execution after an exception is caught.
- If a BadCard exception was thrown, it is caught on line 113 and control goes first to the exception class to handle it, then to line 118.
- If any other kind of exception was thrown, it is trapped on line 115, where `fatal()` is called to terminate cleanly.
- Control never reaches line 120 until NCARDS good Cards have been read and entered into the hand.
Output The output that follows shows two sample runs of this program with different exception handlers active. The first output is from the program as shown:

```
Please enter 3 cards.
Spot codes are 2..9, T, J, Q, K, A
Suit codes are S H D C

Enter card # 1 (spot-code suit-code): kh
    Card successfully entered into hand: King of hearts

Enter card # 2 (spot-code suit-code): JD
    Card successfully entered into hand: Jack of diamonds

Enter card # 3 (spot-code suit-code): af
    Spot value or suit or both are wrong
    Legal spot values are 2..9, T, J, Q, K, A
    Legal suits are H D C S
    You entered a of f. Please reenter.

Enter card # 3 (spot-code suit-code): ah
    Card successfully entered into hand: Ace of hearts

Hand is complete:
King of hearts
Jack of diamonds
Ace of hearts
```
Chapter 9: Class Relationships

9.1 The Roles of a Class

Three more basic OO design principles:

- A class is the expert on its own members.
- A class takes care of itself.
- Delegate to simplify!

9.2 Class Relationships

In Chapter 5, the three basic class relationships were discussed. Here, we introduce the concepts, terminology, and UML for the three more complex relationships.

9.2.1 Derivation.

Class B can be derived from class A, which means that a B object is one possible variety of type A. Using derivation, a single object may have more than one type at the same time. Every instance of B or C is also an instance of A; it has all the properties of A, and inherits all of A’s functions. However, it can only see the public and protected members of A. The derived classes will extend either the set of A’s properties, or its functions, or both, and B and C will extend A differently.

In the diagram, the triangle with its point toward A indicates that classes B and C are derived from class A. We will study derivation after the midterm.

9.2.2 Friendship.

Class B can give friendship to A. This creates a tightly-coupled pair of classes in which the functions of A can use the private parts of B objects. This relationship is used primarily for linked data structures, where A is a container for items of type B and implements the interface for both classes. All members of class B should be private so that outside classes must use the interface provided by A to access them. (An alternative is to declare the class B inside class A, with all members of B being public.)

In the diagram, we show a dashed “rubber band” around the two class boxes, indicating that each one relies on the other.

9.2.3 Template Instantiation.

Suppose T is a template; it is not a class, but a pattern from which a class can be constructed. It is diagrammed like a class, with the addition of a dashed rectangle in the upper-right corner that specifies the name(s) given to the template parameter(s).

To produce a class from a template, one must instantiate the template by supplying actual types or integer values for the parameters. During an early phase of translation, the compiler searches the template code...
CHAPTER 9. CLASS RELATIONSHIPS

for the parameter names, and replaces them by the template parameter(s). This process is called binding, and the result is compilable code for a class, which is then compiled.

9.2.4 Friend Example: a complex data structure and its UML.

Suppose you wish to implement a family datebook:

- The Datebook will have an array of linked lists, one for each family member.
- Each List points at a Node.
- Each Node contains one appointment and a pointer to the next Node.
- One appointment can be shared by two or more lists.

The resulting data structure is illustrated in Figure 9.2. In this structure, the Datebook composes four Lists, each list has a one-many association with a set of Nodes, and each Node aggregates one Appointment. Finally, friendship is used to couple the List and Node classes because the List class will provide the only public interface for the Node class. The UML diagram in Figure 9.3 shows these relationships.

9.2.5 Elementary Design Principles

Several design issues arise frequently. We list some here, with guidelines for addressing them and the reasons behind the guidelines.

Privacy. Data members should be private. Public accessing functions should be defined only when absolutely necessary. [Why] This minimizes the possibility of getting inconsistent data in an object and minimizes the ways in which one class can depend on the representation of another. In the Bargraph program, later in this chapter, all data members of all classes are private.

Expert. Each class is its own expert and knows best how to validate an instance, what should be done with its members and how to do it. All functions that use data members should be class functions. If a class seems to need access to a member of another class in order to carry out an operation, it should delegate the operation to the class that is the expert on that data member.
9.2. CLASS RELATIONSHIPS

There is one exception to this rule: To implement a linked structure, such as a List in the previous example, an auxiliary class is needed (Node). These two classes form a tightly coupled pair with an interface class (List) and a helper class (Node). A “friend class” declaration is used to create this relationship. All access to Nodes is through functions of the List class.

In the Bargraph program (later in this chapter) the two members of class Item are private. All access to Items is through the constructor, destructor, and print functions that are within the class. However, in the tightly coupled pair of classes, Row and Cell, Row provides the interface for Cell, so Cell gives friendship to Row, allowing Row to set, change, and print the members of Cell.

Creation. The class that composes or aggregates an object should create it. [Why] This minimizes the ways in which one class depends on another (coupling).

In the Bargraph program at the end of this section, the Row class allocates space for Cells because the Row class contains the list of cells. It allocates the Items also, because Cell is a subsidiary class for which Row supplies the interface.

Deletion. The class that aggregates an object should generally delete it. [Why] To minimize confusion, and because nothing else is usually possible.

The array that was allocated by the Stack constructor and aggregated by the Stack class is freed by the Stack destructor. Also, in the push() function, each time the stack “grows”, the old storage array is freed as soon as the data is copied into the new allocation area. In the Bargraph program, allocation happens at two levels. The Graph constructor allocates and aggregates 11 Rows, and the Graph destructor deletes them. The function Row::Insert allocates an Item and a Cell, which are then associated with the Row. At termination, the Row destructor deletes all the Items and all the Cells associated with it.

Consistency. Only class functions should modify class data members. There should be no “set” functions that allow outsiders to freely change the values of individual data members. [Why] To ensure that the object always remains in a consistent state.

Consider a class such as Row, that implements a linked list. All actions that modify the list structure should be done by Row class functions. The only class that should even know about its structure is Row, itself. Under no circumstances should the Row class provide a function that lets another class change the pointer in a Cell. If another class is allowed to change list pointers, the list may be damaged beyond recovery.

Delegation. If a class A contains or aggregates a data member that belongs to class B, actions involving the parts of B should be delegated to functions in class B. [Why] To minimize dependency between classes.

In the Bargraph program, each public class has its own print function. The Graph class delegates to the Row class the job of printing each Bar. That class, in turn, lets the Item class print the data fields of each Item.

Don’t talk to strangers. Avoid calling functions indirectly through a chain of references or pointers. Call functions only using your own class members or parameters and let class of the member or parameter perform the task by delegating in whatever way makes local sense. [Why] To avoid and minimize the number of indirect effects that happen when a class definition changes. Basically, an action taken by class A should involve only the classes that are connected to A in the UML.

In the Bargraph program, the Graph class aggregates the Row class, the Row class associates with the Cell class and aggregates the Item class. So the Graph class is free to call Row functions, but it should not be calling Cell functions or Item functions. To do a Cell or Item operation, Graph should delegate the task by calling a Row function (and Row should provide functions for all such legitimate purposes.)

Responsibility. Every class should “take care of” itself, validate its own data, and handle its own emergencies. [Why] To minimize dependencies between classes. Friend classes are the only exception.

In a Stack program, the Stack can check whether it is full and if it is, it allocate more storage. There is no reason that a client program should ever ask about or be aware of a “full” condition.
9.3 Example Program: Making a Bar Graph

9.3.1 Techniques Illustrated.

The data structure used is a linked list, implemented by a friendly pair of classes:

- Class Graph composes 11 Rows, created one at a time.
- Class Row is associated with potentially many Cells.
- A mutually-dependent pair of classes: Row and Cell, are implemented by Cell giving friendship to Row.
- Each Cell Composes one Item.
- Cell is a class with a private constructor, its destructor is also private in the version with C-pointers.

This program was written and debugged three times:

- First with C-style pointers.
- Then with unique pointers.
- Finally, with shared pointers.

The output is much the same in all three cases. The code in the two versions with smart pointers is 98% the same. Of the three implementations, the one with unique pointers seems to be the best one to emulate for a linked list. It is shown in its entirety here. All three are given in zip files on the Canvas course site.

Techniques of note include a static class variable, with static initialization, a static function. Printouts in the destructors to document the process of deallocation, and two ways of freeing dynamically allocated items from a linked list.

9.3.2 Specification

Scope: Make a bar graph showing the distribution of student exam scores.

Input: A file, with one data set per line. Each data set consists of three initials and one exam score, in that order. (You may assume that all lines have valid data.) A sample input file is shown below, with its corresponding output.

Requirements: Implement a linked list, using standard language elements. Use it to create and initialize an array of eleven rows, where each row is initially empty but will eventually contain a linked list of scores in the specified range. Then read the file and process it one line at a time. For each line of data, dynamically allocate storage for the data and insert it into the appropriate row of scores.

Do this efficiently. When adding an item to a row, compute the subscript of the correct row. Do not identify it using a switch or a sequential search. Also, do not traverse that row’s list of entries to find the end of the list; insert the new data at the head of the list. This is appropriate because the items in a list may be in any order.

Formulas: Scores 0...9 should be attached to the list in array slot 0; scores 10..19 should go on the list in slot 1, etc. Scores below 0 or above 99 should go on the last list. The order of the items in each row does not matter.

Output: A bar graph with 11 horizontal bars, as shown below.
9.3. EXAMPLE PROGRAM: MAKING A BAR GRAPH

Input:

AWF 00
MJF 98
FDR 75
RBW 69
GBS 92
PLK 37
ABA 56
PDB 71
JBK -1
GLD 89
PRD 68
HST 79
ABC 82
AEF 89
ALA 105

Output:

Put input files in same directory as the executable code.
Name of data file: bar.txt
File is open and ready to read.

10..19: AWF 0
10..19: MJF 98
10..19: FDR 75
10..19: RBW 69
10..19: GBS 92
10..19: PLK 37
10..19: ABA 56
10..19: PRD 68
10..19: HST 79
10..19: ABC 82
10..19: AEF 89
10..19: ALA 105

The detailed UML class diagram for BarGraph.

9.3.3 The Bargraph Application.

Notes on the Graph header:

- **Graph** has one data member, a stack-based array of 11 Bars. This relationship is composition. There are no pointers here, so no destructor is needed.
- The destructor has no need to delete anything, but is defined to print a trace of its actions. This is an effective device for both learning and debugging.
- Line 22: A public static function can be run early in execution time to tell the user what to do.
- The **insert()** function is private because no outside class should ever insert a Cell into a linked list. Here, it called from the constructor to build the data structure.
class Graph {
private:
Row bar[BARS]; // Each list is one bar of the graph. (Composition)
void insert(char* name, int score);

public:
Graph (istream &infile);
"Graph(){ cout <<"\n "Graph"; } 
ostream& print (ostream &out); // Parameter is an open input stream.
// Static functions are called without a class instance
static void instructions() {
    cout <<"Put input files in same directory as the executable code.\n";
}
};
inline ostream& operator<<(ostream &out, Graph &G){ return G.print(out); }

#include "graph.hpp"
 //--------------------------------------- Use an input file to build a graph.
Graph::Graph(istream &infile) {
    char initials[4]; // used locally for input.
    int score; // used locally for input.
    for (;;) {
        infile >> ws; // Skip leading whitespace before get.
        infile.get(initials,4,','); // Store three initials and null char.
        if (infile.eof()) break;
        infile >> score; // No need to skip ws before using >>.
        if (infile.fail()) {
            infile.clear();
            cleanline(infile);
        } else insert (initials, score); // can be infinite loop without the if.
    }
}

// --------------------------------------------- Insert a node into a Row.
void
Graph::insert(char *initials, int score) { // Function is private within class.
    int index; // Calculate insertion index for score
    if (score >= 0 && score < 100) // If score is between 0-99, it
        index = (score/(BARS-1)); // belongs in one of first BARS-1 rows.
    else
        index = BARS-1; // Errors are displayed on last row
    bar[index].insert(initials, score); // delegation
}

// ------------------------------------------- Print the entire bar graph.
ostream&
Graph::print(ostream &out) {
    out << "\n";
    for (int k=0; k<BARS; ++k) out << bar[k] <<"\n"; // Delegate.
    return out;
}

Notes on the Graph implementation:
• The constructor realizes the data structure, that is, it reads the input file and uses it to build the data structure.
9.3. Example Program: Making a Bar Graph

- The constructor's input loop follows the normal pattern. Omitting line 42 would throw the process into an infinite loop if the read operation failed. The 1-line remedy will avoid that outcome but fails to give reasonable information to the user. Better error handling is important in real applications.
- The insert function delegates the insertion operation to one of the 11 bars, selected according to the score value.

Notes on the Row and Cell classes:

This file declares a tightly bound pair of classes that, together, define a linked list. The major class is Row, the helper class is Cell. "Tightly bound" means that neither is functional without the other.

```cpp
// Class for a linked-list row and its cells
// A. Fischer, October 1, 2000 file: row.hpp
// Modified M. & A. Fischer, 2009, 2022, 2023
//========================================================================
#pragma once
#include "tools.hpp"
#include "item.hpp"

class Cell;
using upt = unique_ptr<Cell>; // define a short name for the type

class Row { // Interface class for one bar of the bargraph.
friend class Row;
private:
    Item info;
    upt next;
    Cell(char* name, int sc, upt& next)
        : info(name, sc), next(move(next)) {}
public:
    ~Cell(){ cout << " ~Cell "; }
};

class Row { // Interface class for one bar of the bargraph.
private:
    static int rowNum;
    char label[10] = " 0.. 9: "; // Row header label
    upt head; // Pointer to row of cells, empty.
public:
    Row ();
    ~Row () { cout <<"\n "Row "; }
    void insert ( char* name, int score ); // delegation
    ostream& print ( ostream& os );
};

inline ostream& operator << (ostream& out, Row& T){return T.print(out); }
```

- In this pair, Row provides the interface for both classes. It is supported by a helper class, Cell. This is a 1-many association: zero or more Cells are needed to store the data for one Row.
- Lines 78 and 79 declare a short name for unique_ptr<Cell>. Concise names greatly improve readability. Line 78 is required because line 79 uses the class name Cell, which is not yet declared.
- Line 84 gives the Row class full access to all members of Cell. This allows Row to manage both classes.
- Line 86 says that Cell composes one data item. This item will be freed when the Cell is freed.
- Lines 87 and 100 declare that the head of the list and the pointers that form the list are all wrapped in unique pointers. These connect the dynamic memory areas to the destructor cascade.
• Line 88: The Cell constructor, as usual, is implemented using ctors. The Cell destructor is used, again, to print a trace comment.
• Line 89 uses move assignment to store a pointer in the new Cell. The parameter, next, is a unique pointer; copy assignment has been disabled for it. This instruction moves the dynamic extension from the caller that created it (Graph::Graph()) into the Cell, and transfers custody to the Cell.
• Lines 90–91: The destructor is public because the destructor of the unique_ptr<Cell> class needs to use it. It must, therefore, be public.
• Cell is a private class except for the destructor. The friend declaration on line 81 gives Row read and write privileges within the Cell class. No other class can access Cells. This is one of the two ways to define a dependent pair of classes. The other way is to define the helper class inside the major class and make all of its members public. Both ways have advantages, but this way is clearer and simpler for non-template classes.
• Line 99: The Row class composes a label for the row and one unique pointer to the first cell on the list. It is the only way to access the cells.
• Line 103: The Row destructor does not need to free memory because the unique pointers take custody of every dynamic object. We could define it =default. However, here we use it to print one piece of a trace of the destructor cascade.

Implementation of class Row.

```cpp
//========================================================================
// Implementation of class Row.
// A. Fischer, April 22, 2000 file: row.cpp
// Modified M. & A. Fischer, September 17, 2009, July 2022
//========================================================================
#include "row.hpp"
int Row::rowNum = 0; // static initializer must be in a .cpp file.
   //------------------------------------------------------------------------
// Row number is used to construct a label for the row
Row::Row() {
   if (rowNum == 10) strcpy( label, "Errors: ");
   else label[0] = label[4] = '0' + rowNum; // example: label="70..79"
   ++rowNum; // increment shared static variable
}
// Create and insert Cell into linked list at head
void Row::insert( char* name, int score ){
   head = upt(new Cell( name, score, head )); // put new cell at head of list
}
// Design pattern: creator. Item is created by Cell constructor.
void Row::print( ostream& os ){
   for (Cell* cur=head.get(); cur!=nullptr; cur=cur->next.get())
      os << cur->info;
   return os;
   // Design decision: print Cell data directly; no delegation of print
   //------------------------------------------------------------------------
   ostream& Row::print( ostream& os ){
   os << label;
   for (Cell* cur=head.get(); cur!=nullptr; cur=cur->next.get())
   os << cur->info;
   return os;
```
9.3. EXAMPLE PROGRAM: MAKING A BAR GRAPH

- The \texttt{insert()} function allocates dynamic memory and attaches it at the head of the linked list.
- \texttt{print()} walks the list and prints the context of each cell. An alternative design would be to let the \texttt{Cell} class implement a print function and delegate to it. This implementation is simpler but is only usable in tightly-coupled classes.
- There are two public functions, \texttt{insert} and \texttt{print}. \texttt{Insert} allocates a new dynamic cell and inserts it in the easiest place: at the head of the list. This can be done because the list is unsorted.

The \texttt{Item} class, declaration and implementation:

```cpp
//===========================================================================
// Item: A student's initials and one exam score.
// A. Fischer, October 1, 2000 file: item.hpp
// Modified M. & A. Fischer, 2009, 2022, 2023
//===========================================================================
#pragma once
#include "tools.hpp"

class Item { // One name-score pair
private: // Variable names are private
    char initials [4]; // Array of char for student name
    int score; // Integer to hold score
public:
    Item (char* inits, int sc) : score(sc){ strcpy(initials, inits); }
    "Item() { cerr " "Item " "<<initials " "score " " ; }
    ostream& print( ostream& os ){return os <<initials " "score " " ; }
};
inline ostream& operator<< (ostream& out, Item& x){ return x.print( out ); }
```

Notes on the \texttt{Item} class:
- This is a data class. Row is container class, and Graph is a controller.
- Many data classes are brief. All are simple. As is true here, there is often no need for a separate implementation file – everything is in the header. Nothing here is unusual.

Notes on the main program:

```cpp
//==========================================================================
// Bargraph of Exam Scores: An introduction to class interactions.
// A. Fischer, September 30, 2000 file: graphM.cpp
// Modified M. & A. Fischer, 2009, 2022, 2023
//==========================================================================
#include "tools.hpp"
#include "graph.hpp"

int main ( void ){
    banner();
    Graph::instructions(); // Call of static class method
    string fname;
    cout "Name of data file: ";
    cin >> fname;
    ifstream infile (fname); // Declare and open input stream
    if (!infile.is_open()) fatal( "Can't open file for input: " + fname);
    cout "File is open and ready to read.\n"
    { // Declare and run graph in a block.
        Graph curve( infile ); // Declare and construct a Graph object.
        // Realize data structure from a file
        cout "Print the graph."
    } // Block exit; graph goes out of scope and is deleted.
    bye();
}
• The task of a main program is to provide a clear interface between the user, the operating system, and the application.

• Line 174 calls the static function in the Graph class to give user instructions. Then main() opens and verifies an input file. Streams are opened and verified in main, because that is part of the interface between the program and the operating system.

• By line 181, all interface actions are done. Then this main() does something unusual: it creates a block enclosing lines 182–186 where the major class, Graph, is instantiated and used. The purpose of this block is to illustrate that deallocation happens when an object (curve) goes out of scope.

• That happens on line 187. Then the output from the trace comments in all the destructors is displayed.

• If the open and close braces were omitted, the trace output would follow the “Normal termination” message.

Notes on the deletion process: Normally, you see nothing when an object is deleted. In this program, however, all the destructors have trace comments so that you can see what happens, and in what order the destructors are run. Here is the trace output:

```
~Graph
~Row ~Cell ~Cell ~Item JBK ~Item ALA
~Row ~Cell ~Cell ~Item MJF ~Item GBS
~Row ~Cell ~Cell ~Cell ~Item GLD ~Item ABC ~Item AEF
~Row ~Cell ~Cell ~Cell ~Item FDR ~Item PDB ~Item HST
~Row ~Cell ~Cell ~Item RBW ~Item PRD
~Row ~Cell ~Item ABA
~Row
~Row ~Cell ~Item PLK
~Row
~Row
~Row ~Cell ~Item AWF
Normal termination.
```

• The Graph goes out of scope on line 187; you see its trace comment first.

• When the graph is deleted, all of its parts must be deleted. Graph composes an array of Rows, which are deleted in the opposite of their creation order, by running the Row destructor on each one, starting with the last row.

• The Row class deleted head, the unique_ptr<Cell> inside it.

• When a Cell is deleted, its dynamic Item and next, a unique_ptr<Cell> are deleted. This deleting the dynamic object stored in each cell and passes the action down the line to the next cell.

• The destructor cascade ends when deletion of row 0 is finished because Graph and main() have no more local data objects declared inside them.
Chapter 10: Array Data Structures

A fundamental principle of good construction and good programming:

Function dictates form. First, get the right blueprints.

This chapter covers some major array-based data structures that have been or will be used in the sample programs and/or assignments in this course. These are:

1. Arrays of objects and arrays of pointers.
2. Ragged arrays of characters.
3. The resizeable array.
4. The hashtable.

At the end of the chapter, these data structures are combined in a hashing program that uses an array of flexarrays of strings.

Basic facts. Common to all data structures in C++ are some fundamental rules and guidelines about allocation, deallocation, and smart pointers. These guidelines are derived from more fundamental rules about how C++ memory management works:

- The method of deallocation should imitate the method of creation.
- If you allocate memory with a declaration, it is allocated on the run-time stack and is freed automatically when control leaves the block that contains the declaration.
- Dynamic memory is allocated using `new` in a memory segment called “the heap”. It can then be wrapped in a smart pointer to handle deallocation. If that is not done, it must be freed using `delete`, otherwise it continues to exist in the heap until program termination.
- Use `delete[]` to free dynamic arrays or wrap the array in a smart pointer. Both automatically call the content-type destructor for each element of the array if the base type is a class type. (For non-class types, there are no destructors to run.)

10.1 Allocation and Deallocation of Arrays.

Which should I use? A fundamental data modeling decision is whether to create the program’s objects using declarations (stack allocation) or using pointers and dynamic (heap) allocation. Even in a simple data structure, such as an array of structures, four basic combinations of dynamic and stack allocation are possible, and all are commonly used.

Four array data structures are shown here, all of which implement some form of two-dimensional structure. However, they have different storage requirements and dynamic possibilities. It is important for you to understand how they differ theoretically and practically, how to create and use each one, and when each one might be preferred. In all cases, the dynamic object can be wrapped in a smart pointer, making deallocation a non-problem.

In this section, we will be using arrays of a representative class type named Item, which has three data members, as shown below.

```cpp
class Item {
    int quant;
    float cost;
    string part_name;
public:
    Item();
    ~Item() =default;
};
```
A diagram is given of the storage allocated for each data structure. Core portions of these objects are
allocated on the run-time stack and are colored gray. Extensions are allocated in the heap and are colored
white. To the right of each diagram is a code fragment that shows how to use new and delete to create and
deallocate the structure.

The first data structure given is a simple array of Items whose size is fixed at compile time. Each succeeding
example adds one level of dynamic allocation to the data structure, until the last is fully dynamic. For the
sake of comparison, these four data structures are implemented as uniformly as possible. All are five slots long
and have offboard end pointers to assist in sequential array processing. (Note: inventory+5 is the same as
&inventory[5].)

1. A declared array. This data structure is used if the length of the array is known at the time control
exters the block containing the declarations. All storage is allocated by the array declaration, so no “new”
is necessary. For an array of class objects, a default constructor must be provided and will be used to
initialize the array.

No “delete” is necessary and using “delete” or “delete[]” is a fatal error because this storage will be
automatically deallocated at the end of the function that contains the declarations and storage must not be
deallocated twice.

```
Item inventory[5];  // Allocate.
Item* end = &inventory[5];  // End pointer.
Item* scan;
```

2. A single dynamic array of class objects. This data structure is used if the length of the array is not
known at compile time but is known at run time, before any data is stored in the array.

The core portion of this object is created by declaring two Item pointers. The extension is created by a
call on “new”, often in the constructor for some class that contains the array. A default Item constructor
must be present and will be used to initialize the array. Dynamic arrays of non-class base types cannot
be initialized.

Since “new” was used with [] to create this array, “delete[]” must be used to deallocate it.

```
Item* inventory;  // Head pointer (core).
Item* end;  // Sentinel pointer (core).
Item* scan;  // Processing pointer (core).

inventory = new Item[5];  // Allocate array.
end = &inventory[5];  // Set end pointer.

delete[] inventory;  // Delete the array.
```

In contrast, we could use a unique_pointer for the dynamic allocation. It creates exactly the same data
structure. (The unique_pointers do not take any added storage or run time.)

```
using upt = unique_pointer<Item>;

upt inventory = upt(new Item[5]);
Item* head = inventory.get();  // Get the pointer.
Item* end = &inventory[5];  // Off-board sentinel ptr.
Item* scan;  // Processing pointer.
```

Explicitly deleting the array is not necessary because a unique_pointer is an object and it has custody of
the dynamic array. The destructor cascade will process the upt and delete the array.
3. **A declared array of pointers to individual dynamic objects.** This data structure can be used if the Item constructor needs parameters or if the maximum amount of storage needed is predictable, but you wish to avoid allocating the storage until you know it is needed. An array of Item pointers is pre-allocated and initialized to `nullptr`. A new Item is allocated and attached to the data structure each time new storage is needed. Any constructor (not just a default constructor) can be used to initialize these items because they are individually created. Since “new” was used without `[]` to create the Items, “delete” must be used without `[]` to deallocate them. Since a loop was used for allocation, a loop must be used for deallocation.

```cpp
Item* inventory[5]; // Create core.
Item** end = &inventory[5]; // Tail sentinel.
Item** scan; // Scanner.
for (p=inventory; p<end; ++p) { // Allocation.
    *p = new Item(field_1...field_n); // Install data.
}
for (p=inventory; p<end; ++p) { // Deletion loop.
    delete *p;
}
```

In this example, if `unique_ptr` pointers are used instead of old-style C pointers, the deletion loop is unnecessary. The default destructor will do the whole job.

```cpp
uptr* inventory[5]; // Create core.
uptr* end = &inventory[5]; // Tail sentinel.
uptr* scan; // Scanner.
for (p=inventory; p<end; ++p) { // Allocation.
    *p = uptr(new Item(field_1...field_n)); // Install.
}
```

4. **A dynamic array of pointers to individual dynamic objects.** This data structure is used when the amount of storage needed is unpredictable. An `Item**` is declared and initialized at run time to a dynamic array of Item pointers, each of which should be initialized to `nullptr`. By using a flex-array data structure, the array of pointers can be easily lengthened, if needed. The cost of the reallocation and copy operations is minimal because only the pointers (not the Items) need to be copied.

Another Item is allocated and attached to the data structure each time new storage is needed. Any constructor (not just a default constructor) can be used to initialize these items because they are individually created. Since “new” was used without `[]` to create the Items, “delete” must be used without `[]` to deallocate them. Since a loop was used for allocation, either unique pointers must be used or a loop must be written for deallocation. After the deallocation loop, the main array (the backbone) must also be freed.

```cpp
Item** inventory = new Item*[5]; // Allocate *s.
Item** p;
for (p=inventory; p<end; ++p) { // Scanner.
    // Read and validate data.
    *p = new Item(field_1...field_n); // Install data.
}
for (p=inventory; p<end; ++p) delete *p;
delete[] inventory;
```

This set of declarations works well, but nobody would implement the data structure this way any more. The only reason for using dynamic allocation for the backbone is to accommodate processes that cannot predict, at the beginning of the program, how long the array might need to be. In that situation nowadays, programmers use vectors. With unique_pointers used to implement the ribs, we get this data structure:
vector<item> inventory;
Item** end = &inventory[5]; // Tail pointer.
upt* scan; // Scanner.
for (;;) {
    // Read and validate data here.
    // Break out of loop if end of data has been reached.
    inventory.push_back( upt( new Item(field_1...field_n))); // Install data.
}

That is all. Vector is self-managing, and the unique_pointers will take care of deleting the Items. To scan the array, you would use a for loop with an int index variable, and iterate up to inventory.size().

10.2 2D Arrays of Chars

The ways of building a 2D array of chars are analogs to the ways of building an array of structures. The four techniques shown above apply to both.

A ragged array is an array of strings, each of which might be a different length (like rags). It is a convenient data structure for representing any list of phrases such as a menu, labels for a graph, or error comments. In C programs, there were two implementations for ragged arrays. In C++, a third implementation has been introduced:

1. A ragged array of quoted literal strings. This is most easily constructed by a declaration with a list of literals as the initializer:
2. In C, an array of dynamically allocated c-strings, each the right size for its contents.
3. In C++, an array (or vector) of C++ strings.

Array of Literals  This is easily built by a programmer to contain all the current menu items, error messages, or other fixed strings. Here is the code in C:

```c
char* vocabulary[] = { "Some", "day", "my", "prince", "will", "come"};
const char* nWords = sizeof(vocabulary)/sizeof(char*);
```

Dynamic Ragged Arrays  A dynamic ragged array is straightforward to build. Each “rag” is constructed separately and attached to a slot in a “backbone” array, which could be either declared or allocated dynamically. Writing the menu items in a separate file is better than building the menu into a program because it allows the menu to be changed easily without recompiling the code.

Creating the rags.  In C, several steps are required to construct a rag. Happily, in C++ we can use a vector for the backbone (instead of an array), and C++ strings (instead of C-strings). Both vectors and strings will “grow”, as needed, to contain data of any length. This frees of from worrying about about buffer length. and data structure length.

Here is a C++ implementation of a Menu program that uses a file. The input loop creates the rags and attaches them to the backbone. Later, they can be accessed using subscripts.
10.3. THE FLEXIBLE ARRAY DATA STRUCTURE

A flexible array is a container class, that is, a class whose purpose is to contain a set of objects of some other type. The C++ standard template library (STL) contains a template class, vector, from which a flexible array of any type may be created. Java supports a type named ArrayList, that is basically similar. These classes are based on a dynamically allocated array that is automatically reallocated, when necessary, to contain more data. Class functions are used to store data in the container and retrieve it.

A flexible array is the modern alternative to a linked list for applications in which an array would be appropriate and desirable but the amount of storage needed is unpredictable. It works especially well when sequential (not random) access is used to add data to the array, as in the simple program below.

In this section, we present a simple version named FlexArray to create a growing array of Items. This data structure was in use to contain unpredictable amounts of data, before ArrayLists and vectors were invented,. There is still a good reason for teaching it: the “magic” that makes the storage grow is very simple and should be understood by programmers.

10.3.1 Implementation in C++

The flexible array data structure consists of a pointer to a long dynamically-allocated array for data, an integer (the current allocation length), and the subscript of the first unused array slot. When there is data to be stored in the container, but it is full, the flexible array automatically doubles in length. The cost is a modest amount of time spent reallocating and copying storage. This can easily be less than the time required to manipulate links and traverse lists. Wasted space is minimal for a flexible array, often less than for a linked structure, but the major advantage of the flexible array is that a programmer can use array-based algorithms (quicksort) and random access to process the data in the array.
The class FlexItems is an implementation of a flexible array with base type char. It is a good example of how a class can (and should) take care of itself. FlexItems has one private function, `grow()` (lines 14 and 68–76), which is called whenever there is a object to store in the FlexItems array, but the array storage space is full. The existence and operation of this function is completely hidden from the client program; the data structure simply “grows” whenever necessary.

The FlexItems class declaration.

```cpp
1 // Class declaration for a flexible array of base type char.
2 // A. Fischer, 2003, 2023 file: flexItems.hpp
3 #pragma once
4 #include "tools.hpp"
5 #include "item.hpp"
6 #define START 4 // Default length >0 for initial array.
7
8 //--------------------------------------------------------------------------------
9 class FlexItems {
10 private:
11    int max = START; // Current allocation size.
12    int n = 0; // Number of array slots that contain data.
13    Item* data; // Pointer to dynamic array of Items.
14    void grow(); // Double the allocation length.
15
16 public: //--------------------------------------------------------------------------------
17    FlexItems(): data(new Item[max]) {}  // Constructor allocates an initial quota of space to store up to MAX items.
18    ~FlexItems() { delete[] data; }    // Destructor uses delete[] because the corresponding constructor used new with [].
19    Item* operator[]( int k ); // Access kth Item in array.
20    int put( Item data ); // Store data, return subscript.
21    Item& operator()( int k ); // Read-only access to array length.
22    int length() { return n; }    // Return reference to array slot.
23    const Item* stock(){ return data; } // Read-only access to data.
24    ostream& print(ostream& out); // Print filled part, ignore rest.
25    inline ostream& operator<<(ostream& out, FlexItems& F){ return F.print( out ); }
26
27 // Implementation of the FlexItems class.
29 #include "flexItems.hpp"
30 // -----------------------------------------------------------------------------
31 Item* FlexItems :: // Access kth Item in array.
32 operator[]( int k ) {
33    if ( k >= n ) fatal("FlexItems bounds error.");
34    return data[k]; // Return reference to array slot.
35 }
36
37 int FlexItems :: // Copy an Item into the array. -----------
38    put( Item it ) {
39        if (n == max) grow(); // Create more space if necessary.
40        data[n] = it;
41        return n++; // Return subscript where item was stored.
42    }
43
44    ostream& FlexItems :: // Print filled part, ignore rest. -------
45    print(ostream& out) { //
46        ...
47        return out;
48    }
49```
for(int k=0; k<n; ++k) cout << data[k] << endl;
return out;
}

void FlexItems :: // Double the allocation length. ------------
grow() {
    cout << "FlexItems is about to grow.\n";
    Item* temp = data; // hang onto old data array.
    max*=2;
    data = new Item[max]; // allocate a bigger one.
    for(int k=0; k<n; ++k) data[k] = move( temp[k] );
    delete[] temp;
}

• Two “get” functions, length() and stock() (lines 22 and 23) provide read-only access to the number of Items stored in the flexible array and to the Item array itself.

• The extension of the subscript operator (lines 21 and 36–39) is not used in this application but will be used later in the chapter. Standard classes such as vector, that contain arrays of data, define subscript so that the “flex” part of the data structure seems transparent to the programmer.

• The put function (lines 20 and 45–50) allows us to store information in the next free slot of the FlexItems. It checks for available space, calls grow() if necessary, stores the data, and increments N, the index of the first available slot. This function provides the only way to store new data in the FlexItems object.

• A print function is (line 485) allows the FlexItems client program to print the contents of the array without knowing how it is implemented.

Notes on the grow() function. The grow function (lines 14 and 52–60) is the heart of this class. It is called from put when the array is full and more input must be stored. The diagrams on the next page illustrate its operation, step-by-step.

• Line 58: Before enlarging the array, we double the value of max. The array is then reallocated at the new, longer, length. Doubling (not lengthening by a fixed amount) is important because it enables the data structure to operate in amortized\[1\] linear time. The lifetime total number of Items moved to a new array is never more than the length of the current array.

• Line 60: The data is moved (not copied) from the old array into the new one, and the old array is freed. Copying would work properly with the kind of items in the application. However, the code will be turned into a template later in this chapter, and templates should be written to work with ANY base type, including those that cannot be safely copied.

The Item class.

```cpp
class Item {
private: // ---------------------------------------------
    int quant;   // Number of Item in stock.
    float cost;  // Cost per item
    string name; // Item's name -- unlimited length.
public: // ---------------------------------------------
    Item() = default;    // Needed to make an empty array of Items.
    Item(int q, float c, string nm) : quant(q),cost(c),name(nm) {}
    ~Item() = default;
    ostream& print(ostream& out) {
```

\[1\]“Amortized” means “Averaged over the life of the data structure”.

```cpp
```
• This is a data class with three members. There are no pointers. It does not need a destructor other than
the default. So an instance of this class is inherently movable.

• No move constructor or move assignment is defined explicitly. Because this is a very simple class with a
default constructor, these methods are also supplied by default.

• Line 80 is a default constructor, needed to create an array of Items.

• Line 81 is the real constructor that populates the object with meaningful data.

• Line 82 is the destructor. Defining it explicitly as “delete[]” makes it clear that the programmer did not
just “forget” the destructor. This default definition does not prevent generating a default move constructor
or assignment.

The main program. The main program for this example shows a simple application for a flexible array: it
reads an indefinite number of lines of input whose length is not predictable. The main program starts with
sic creates a FlexItems named fit (line 87) and uses it to read one line of input. At end of file, Lines 99 and 100
print debugging output and line 101 prints all Items.
The output:

Demo program for FlexItems
Please enter name of inventory file: parts.txt
FlexItems is about to grow.
FlexItems is about to grow.
Debug: 10 Should be 10
Debug: 2.38 57 long nosed pliers Should be long nosed pliers

9.99 28 claw hammer
8.95 3 claw hammer
9.95 2 ball peen hammer
5.95 3 tack hammer
2.38 57 long nosed pliers
2.38 1 pliers, 3
1.59 10 roofing nails-1 lb
6.99 15 roofing nails-5 lb
3.99 12 screwdriver
3.99 18 phillips screwdriver

10.3.2 Tracing the operation of the FlexItems.

The FlexItems data structure created by main() is illustrated in four stages of growth during the input operation. The object named fit is shown in Figure 10.1, just after the FlexItems constructor is done initializing it. The initial length of the array is START, or 4 items.

The object named fit is shown in Figure 10.1, just after the FlexItems constructor is done initializing it. The initial length of the array is START, or 4 items.

Inside the input loop (Lines 91–98) lines are read from the input file one at a time and put them into the flexarray. (An eof condition ends the loop.) The right side of Figure 10.2, shows fit again after four items have been put into it, making it full.

When the command to put a fifth character into fit is given, the FlexItems container must grow. (Figure 10.4, top). This is done in the grow() function. In the C++ implementation, the current maximum length is doubled and a new data array of the new, larger length is allocated. Then the data is moved from the old array to the new and the old array is deleted. Finally, the new (fifth) item is stored in the new, longer array, following the first four items.

Input then continues normally, filling the rest of the slots (Figure 10.3, bottom). When the command is given to store the ninth character, the FlexItems must grow again. The doubling process is repeated, resulting in an array of 16 characters. The ninth character is then stored in the new space and input continues. Figure 8.3 shows the array after the tenth character is input, with ten slots in use and space for six more letters.

Due to space limitations, the name of each item is represented by “xx”
CHAPTER 10. ARRAY DATA STRUCTURES

**Costs.** This is an acceptably efficient storage scheme. After the first doubling, the number of array slots allocated is always less than twice the number in use, and the total number of objects that have been moved, including all move operations, is slightly less than the current length of the array.

**Caution.** Because the entire array is moved during the growth process, any pointers that point into the array must be re-attached after the reallocation. Because of this, a flexible array is not an appropriate data structure when many pointers point into an array. Pointers pointing out of the reallocated array are not a problem.

### 10.3.3 Implementation in C

In C, the doubling is accomplished using the `realloc()` function:

```c
data = (char*)realloc( data, new_size );
```

This function takes any memory area that had been created by `malloc`, `calloc` or `realloc` and changes its size to the new size requested. If the new size is shorter, the excess memory is returned to the system for future re-use. If the new size is longer and the adjacent memory area is free, the existing area is simply lengthened. Otherwise, a new, larger area is allocated somewhere else in memory and the contents of the array are copied into the new area. This is easier and more efficient than the C++ implementation, in which the reallocation and copying must be done step-by-step.

### 10.4 The Flex Template

In section 10.3, the FlexItems class was presented and explained. It is a much-simplified version of a `vector<Item>`, specialized for growing when the input data exceeds initial expectations. If we needed a different flavor of Flex class, we would need to be rewrite and re-debug the code. There’s got to be a better way to accomplish this goal, and there is.

In fact, there are several C++ tools that, singly or together, provide better ways.

1. If we want only one version of a data structure, we can define the class with an abstract base type name, like T, and use a `typedef` or `using` declaration at the beginning of the program file to map T onto a real type. This is illustrated by the Hash Table example later in this chapter.

2. The C++ standard library supports several basic data structures in the form of templates with type parameters. The templates are abstract code written using techniques very much like those used in the generic insertion sort. These templates are often referred to as the “Standard Template Library”, or STL. To use a template from the library, you must supply a real type in the declaration that creates an object of the template class; technically, we say that you `instantiate` the template. The compiler will combine your type with the template’s abstract code at compilation time, to produce normal compilable code.

Templates are a powerful tool, introduced to C++ in 1998, to provide efficient implementation of the major data structures. All are well documented at cplusplus.com. Templates make programming construction and debugging easier. In the design of these templates, extreme attention has been paid to:
(a) Type-safety.
(b) Efficiency.
(c) Freedom from bugs.

STL is an amazing piece of work. It covers all of the most important data structures, and brings them into a unified interface, so that a program written using one data structure could be changed to use another simply by changing the line that instantiates the template. The implementations it provides are efficient and correct, and their performance characteristics are thoroughly documented.

3. You could write your own template type, as we do in this section.

4. You could start with a standard template type or with your own template, and use derivation to add more functions to the class. Such functions might be needed for some base types but not for others. The string type we use constantly was derived from the STL template type vector with many added functions.

10.4.1 The Flex Template

Class templates are not classes; they are patterns from which classes may be generated. They must be instantiated with a specific base type before they can be compiled. During instantiation, every occurrence of the template parameter is replaced by the type given in the angle brackets. The resulting code is then compiled.

The template parameter defines the base type of the data structure. It may be any type that is already defined in a program, a primitive (like double) or a class type or a struct or another instantiated template type. Instantiation is done exactly once, during compilation, for each distinct base type, and the resulting code is compiled. Thus, if you declare three instances of vector<string>, instantiation will happen just once. But if you use a vector<Tree> and a vector<int>, the template for vector will be instantiated and compiled twice.

10.4.2 The Flex Template

Template basics.

- All the code for the template must be in the header file (.hpp) because all the code must be instantiated uniformly before any of it can be compiled.
- The beginning of the class declaration must be preceded by “template <class T>”, where T is an arbitrary identifier. All of the functions could be inline or some could be defined after the end of the class.
- Any code that is not inside the class declaration, for example, the operator extension, must also be prefaced by “template <class T>”.
- Comparing this template version to the FlexItems code, there are three kinds of changes in the code:
  - Every use of Item was replaced by T.
  - The function definitions from the .cpp file are now at the bottom of the .hpp file for the template.
  - The header template <class T> was added before the class name and before every function definition outside the class.
- Now that this is a template, it is important that Lines 33 and 44 use move assignment instead of copy assignment. This template will work for any class without extensions, and for classes WITH extensions that define move assignment.
- Restriction: Because the Flex code delegates the print() function to its base class, that class must support a compatible print function. Similarly, the base type must have an operator<< extension because of Line 40.
// Class declaration for a flexible array of base type T.
// A. Fischer, A. Fischer, July, 2022

#pragma once
#include "tools.hpp"
define move std::move
#define START 4 // Default length for initial array.

template <class T>
class Flex{
private:
    int max = START; // Current allocation size.
    int n = 0; // Number of array slots that contain data.
    T* data; // Pointer to dynamic array.
    void grow();
public:
    Flex() : data(new T[max]) {}
    ~Flex() { delete[] data; }
    int push( T dt ); // Store data, return subscript.
    int length() { return n; } // Read-only access to array length.
    T& operator[]( int k );
    ostream& print(ostream& out){
        for( int k=0; k<n; ++k) out << data[k] << " ";
        return out;
    }
};

template <class T> void Flex<T>:: // -----------------------
grow(){
    T* temp = data;  // hang onto old data array.
    max *= 2;  // Double the allocation length.
    data = new T[max];  // allocate a bigger one.
    for(int k=0; k<n; ++k) data[k] = move(temp[k]);
    delete[] temp;
};

template <class T> T& Flex<T>:: // ------------------------
operator[]( int k ){
    if ( k>=n ) fatal("Flexarray bounds error.");
    return data[k]; // Return & of desired array slot.
}

template <class T> int Flex<T>:: // -----------------------
push( T dt ) {
    if ( n==max ) grow(); // Make more space if needed.
    data[n] = move(dt); // Store current data.
    return n++; // Return index of stored item.
}

operator<<(ostream& out, Flex<T>& F){ return F.print(out); }

Templates will be dealt with more extensively in a later chapter. For now, this is enough.

10.5 Hashing

A hashtable is a container class that can store and retrieve data items quickly and efficiently but does not keep them in a predictable or sorted order. The position of a data item in the hash table is derived from a key data field, or a key constructed from parts of several data fields. When implementing a hash table, three issues must be decided:

1. The backbone of a hash table is an array. How many slots will it have?
2. Each slot of the hash array is called a bucket. What data structure will you use to implement a bucket?
   The number of buckets is a time-space compromise. More buckets lead to a larger memory footprint and faster performance. Fewer buckets mean that the list of entries in each is longer and more time will be spent searching those lists.
3. **Hashing** is the act of applying the algorithm to a data item’s key field to get a bucket number. What algorithm should be used for your hash function? To assign an item to a bucket, a table subscript is calculated from the key field. The algorithm that does the calculation is called the *hash function*. It must be chosen to work well the expected data set. It should use the whole key, not just a part of the key.

In most hashing applications, an Item is a structure with several fields and the key field is just one of many. In such applications, distinct Items may have the same key. In this discussion, we imagine that we are hashing words for a dictionary. Since the Item will be a simple null-terminated string, the entire item will be the key field.

### 10.5.1 The Hash Table Array

**Table structure.** There are two basic ways to implement a hash table:

- A flat table. Each slot in the main hash array can store exactly one Item. Using this plan, the array must be longer than the total number of Items to be stored in it, and will not function well unless it is nearly twice that long. When a key hashes to a bucket that is already occupied, it is called a *collision* and some secondary strategy must be used to find an available bucket. Often, a second (and possibly a third) function is used; this is called *rehashing*.

- A 2-dimensional table: Each bucket in the main hash array can store an indefinite number of Items. Rehashing is not necessary; dynamic allocation takes its place. Ideally, the list of items in a bucket should be short so that it can be searched quickly. This fact is used in calculating NBUK: if the goal is an average of $B$ items per bucket, and we expect to have $N$ items total, then NBUK should be approximately $N/B$.

This kind of hash table is much easier to implement and will be used here. The bucket can be implemented by any dynamic data structure; but in this discussion we use a flexarray, as in Figure 10.5.

Our chosen hash table implementation is an array of NBUK buckets where each bucket is a flex-array of strings, as shown in Figure 10.6. The buckets will be initialized in the Bucket constructor to be empty flex-arrays with 4 slots of type $T*$ where $T$ flexarray as they are read in.

---

Because of space limitations, C++ strings will be represented in the diagrams by just one internal part of a string – the pointer to a dynamic array of chars.
CHAPTER 10. ARRAY DATA STRUCTURES

Table length. For reasons related to number theory, and beyond the scope of this discussion, hashing works best when the number of the buckets is not a multiple of 2. Normally, the programmer would choose a large prime number for the table length. Here, we choose a small one so that the diagrams will fit on a page. When you implement your own hash table, start with a small number of buckets and do the initial debugging with a modest number of data items. When the program is ready for a full-scale trial, change the NBUK to a large prime.

Inserting a new Item. Suppose that \( D \) is a data item and that the program wants to insert it into a hash table. Before insertion, the program should search the table to determine whether it is duplicate data. If not, it is inserted. The key field is hashed to find the correct bucket number, and the FlexArray::put() function is used to insert the resulting pointer at the end of the appropriate list. Because a flex-array is being used, each list will automatically extend itself if it becomes overfull.

Retrieving an Item. To find an item in the table, we must hash its key field and calculate the correct bucket # and search it for the desired key. Since every hash bucket list should be fairly short, a simple sequential search is appropriate. If more than one item may have the same key field, more comparisons are needed to confirm whether or not the desired item (or another with the same key) has been located.

10.5.2 Hash Functions.

When choosing a hash function, several things are important.

1. It should use all of the information in the hash key, not just the beginning or end.

2. Its result must be a subscript between 0 and NBUK-1. This is easily accomplished by using the key to generate a larger number and using the mod operator (\( \% \)) to reduce it to the desired range.

3. It should be efficient. Hash functions that make a lot of unnecessary calculations should be avoided. Bit operations (bitwise exor and shifts) are fast; addition and subtraction are slower, multiplication is slow, and division and mod are very slow. The last operation in the hash function must be mod, but, with that exception, the slower operators should be avoided. Remember—the object is to calculate a meaningless random number based on the key.

4. It should spread the data as evenly as possible throughout the table, that is, it should choose each bucket number equally often, or as near to that as possible.

Thus, the nature of the hash function depends on the data type of the key and the number of bytes in the key. A sample hash function is given below for strings that gives one idea about what a hash might do. There is no limit on the number of possibilities.

Simple hash algorithm. This algorithm works for strings of up to 32 characters.

1. Initialize hash, an unsigned long, to the first char in the string.
2. hash = hash xor first character.
3. Shift hash 1 bit to the left.
4. Repeat shift and xor for each additional character.
5. Return hash \( \% \) table length.

10.6 Example: A Dictionary

This example combines several of the C++ features discussed earlier and adds new feature.

1. Two different array data structures are used together to build a hash table: an array, and a flexarray.
2. A hash function is computed using bit-operations.
3. Extensions of the subscript operator are defined for the hash table and the flexarray.
4. Flexarray is a template class.
5. Move semantics and move assignment are used to put things into the flexarray.
6. A typename is defined by a `using` statement (instead of the older `typedef`).

The application that inspired the Dictionary class is a word game. It requires support for these actions:

1. Enter a word into the dictionary. The words in the application’s vocabulary are stored in a text file. The number of words in a dictionary for a given application can vary greatly, but is normally in the thousands. A re-sizeable data structure is required. A vector or flexarray is easiest to use.
2. Answer quickly whether or not a given string is a word in the dictionary. The fastest data structures for this purpose are the hash table and the binary search tree.
3. For debugging, print the words in the dictionary.

A hash table is very fast for purpose 2, and good for purposes 1 and 3. Following is the implementation for a Dictionary built from a hash table.

### 10.6.1 The Dictionary Class

```cpp
//------------------------------------------------------------------
// Class declaration for a dictionary
// A. Fischer, M. Fischer, July, 2022
file: dict.hpp
#pragma once
#include "tools.hpp"
#include "hash.hpp"

// ---------------------------------------------------------------
class Dictionary{
private: //-----------------------------------------------------
HashTable hasher;
public: //------------------------------------------------------
Dictionary() =default;
bool find(const string& key) { return hasher.find(key); }
bool insert(const string& s);
ostream& print(ostream& out);
};
```

**Notes on the Dictionary Declaration.**

- Line 11: The hash table, `hasher` will be used to rapidly store and later locate a particular word in the Dictionary.
- Line 14: The default Dictionary constructor will delegate most of construction to the default hash table constructor, which will delegate construction to the Bucket constructor. The entire initial dictionary is built out of default parts.
- Line 15: While playing this game, we care only whether a word is or is not in the Dictionary. Therefore, both `Dictionary.find()` and `HashTable.find()` return a `bool` result. (We do not need to know where the word is, or anything else about it.)
- Lines 16 and 24–29: The task of `insert()` is to insert a new word into both the vocabulary and the hash table, if it is not already there. Line 117 checks for that. Then line 27 delegates insertion to the hash table.
- Lines 17 and 31–37: are the `print()` function that will be used during debugging.
The Dictionary implementation.

19 // -----------------------------------------------
20 // Class implementation for a Dictionary
22 #include "dict.hpp"
23 // -----------------------------------------------
24 bool
25 Dictionary::insert( const string& st ){
26 if (find(st)) return false; // Already there
27 hasher.insert(st); // Put string into hash table
28 return true;
29 }
30 // -----------------------------------------------
31 ostream&
32 Dictionary::print( ostream& out ){
33 out << "Dictionary::print() called" << endl;
34 for (int k=0; k<hasher.length(); ++k)
35 out << k << " ". " << hasher[k] << endl;
36 return out;
37 }

The HashTable declaration.

38 // -----------------------------------------------
39 // Class declaration for a hash table of strings
40 // A. Fischer, M. Fischer, July, 2022 file: hash.hpp
41 #pragma once
42 #include "tools.hpp"
43 #include "flexT.hpp"
44 #define NBUCK 89
45 using Bucket = Flex<string>;
46 // -----------------------------------------------
47 class HashTable {
48 private:
49 Bucket bucks[NBUCK];
50 long unsigned hash(const string&); const;
51 Bucket& getBuck(const string& key) { return bucks[hash(key)];}
52 public:
53 HashTable() =default;
54 int length() { return NBUCK; }
55 Bucket& operator[](int k){ return bucks[k];}
56 void insert(const string& s) {
57 Bucket& bk = getBuck(s);
58 bk.push(s);
59 }
60 bool find(const string& query);
61 }

Notes on the HashTable. The code follows these notes.

- Line 44: The number of buckets in the hash table is defined at the top to make it easy to find and change. 89 is a very small number of buckets for a hash table. It is appropriate for at most a few hundred entries.
- Line 45 is the new way to define a synonym for a type name. It works better than the old typedef when combined with templates. This line lets us write “Bucket” instead of “Flex<string>”. The new name is shorter, easier to type, and more meaningful in the context of a hash table.
- The decision to use a flexarray, rather than a vector, to implement each bucket was somewhat arbitrary. However, flexarrays are optimized for insertion, and do not supply functions that could affect the data in the flexarray. They do provide all the functionality we need here.
10.6. EXAMPLE: A DICTIONARY

- Line 49: The HashTable is an ordinary fixed-length array of Buckets. It is stack-allocated so there is no need to write a destructor.
- Lines 50 and 68–73: hash() is the hash function we are using. It is defined as a separate function to keep the hashing logic separate from the hash table logic. It is a private function because it should not be used outside this class.
- Lines 68–73: To hash a string, line 70 initializes the answer to 0 bits. Line 71 shifts the answer leftward 1 bit and exors it with the next letter, repeatedly, until all the letters have been included.
- Line 51: getBuck() takes a string as its parameter, hashes it, and returns the Bucket in which that string should be stored. It is the only function that calls the hash function. It is a private function because it should not be used outside this class.
- Line 54: The default constructor for the hash table is appropriate. It will delegate to the default constructor for the Buckets.
- Lines 77–80, the insert() function. Inserting a string into the hash table requires two steps: first identify the correct bucket number by calling getBuck(), second, simply push the item onto the end of the bucket’s flexarray.
- Lines 57-60: This function implements the heart of a hash table: to insert a word, we hash it to find the correct bucket. Then we push it onto the end of the selected bucket.
- Line 61 and 75–83: To find a word that was previously inserted.
- Lines 61 and 75–83, the find() function: To insert an item in a bucket, first we must find the correct bucket number by calling getBuck(), then search the items in the bucket to determine whether one of the words in the bucket matches the search key. This search is fast because the average number of items in a bucket is small. (If it is not small, define NBUCKS to give a larger number of buckets.)

The HashTable Implementation.

```cpp
#include "hash.hpp"

HashTable::HashTable() : hasher(*this), bucks(NBUCK) {
    for (int i = 0; i < NBUCK; ++i) {
        bucks[i] = new Bucket;
    }
}

HashTable::HashTable(const string& key) {
    Buckets[]"'
```
10.6.2 A test function.

We need a test function because each part of a large application should be debugged separately. The goal of a test function is to test every public functionality of a class – not to do anything that makes sense for the larger application. Here, we use `main()` for that purpose. After passing this test script, the class will be ready to integrate into a game.

```cpp
#include "tools.hpp"
#include "dict.hpp"

int main ( int argc, char* argv[]) {
    if(argc != 2) fatal("Usage: "+ string(argv[0]) + " file");
    string filename = argv[1];
    ifstream istr(filename);
    if (!istr.is_open()) fatal("Can't open file "+ filename + ":");
    cout << "Reading words from file " << filename << endl;

    Dictionary dy; // Create and initialize the Dictionary.
    for (;;) {
        string s;
        getline(istr, s);
        if (istr.fail()) break;
        dy.insert(s);
    }
    if (!istr.eof()) fatal("Error reading file "+ filename + ":");
    cout << "File read completed -- printing dictionary" << endl;
    dy.print(cout);

    Dictionary dy; // Create and initialize the Dictionary.
    for (;;) {
        string s;
        getline(istr, s);
        if (istr.fail()) break;
        dy.insert(s);
    }
    if (!istr.eof()) fatal("Error reading file "+ filename + ":");
    cout << "Now some queries... (type . to quit)" << endl;
    for (;;) {
        string key;
        cin >> key;
        if (key == ".") break;
        cout << "Looking up key: " << key << " " << endl;
        if (dy.find(key)) cout << "Found" << endl;
        else cout << "Not found" << endl;
    }
}
```

- This main function starts by doing a job that is often or usually done in `main`: get the name of the input file from the command line (lines 148–149), open the file and check for proper opening (lines 93-96).
- Line 99 instantiates a Dictionary which, in turn, instantiates a hash table.
- Line 100: The test code begins here. We read the file one line at a time, checking for read errors. Line 104 inserts the new word in the dictionary.
- Line 120: The vocabulary is printed out. Comparing it to the input file assures us that all words were properly entered into the dictionary.
- Lines 113–119 read the words again and search for them in the vocabulary. All words in the file were successfully found.
• Lines 122–130 ask the user to enter a series of words that should and should not be in the dictionary. During the test, they were all correctly found or not found.

Figure 10.8: Fully developed UML for the hash table.
Chapter 11: Derived Classes

A consequence of inheritance:

The sins of the father are to be laid upon the children.
...Euripides, Exodus, and Shakespeare.
...And so are the strengths and skills of the father, in C++.

11.1 Derivation Basics

Purposes of derivation. A class may be derived from another class for several possible reasons:

1. To add functions to those defined by an existing class.
2. To extend and specialize the actions of a function defined in an existing class.
3. To mask a function in an existing class and prevent further access to it.
4. To create a different interface for an existing class.
5. To add data members to those included by an existing class.
6. To further restrict the protection level of data members that belong to an existing class.

Declaration syntax. The first line of the class declaration declares the derivation relationship. The following two lines are taken from the demo program below.

```cpp
class Printed : private Pubs { ... };
class Book : public Printed { ... };
```

The base class must be defined first; in this case, it is Pubs. The second class (Printed) was derived from Pubs using private derivation. The third class, Book, was derived from Printed by public derivation.

Rules and usage patterns. The style of derivation (public, protected, or private) determines the protection level of the members in the derived class as well as accessing rules for second-level derived classes. Details will be clarified later. For now, it is enough to know that public derivation is commonly used and private derivation rarely used.

The derived class inherits all parts of its base class. If the derived class has two data members and the base class has two data members, then an object of the derived class will have four members and all the functions defined in both classes. However, some of these members may not be accessible in the derived object— that depends entirely on the privacy properties of the base class and the type of derivation.

1. Often, more than one class is derived from a base class, providing multiple variations on the basic class.
2. Inheritance chains such as `Pubs --> Printed --> Book` do occur in real programs but are not the most important use of derivation.
3. Two or more derived classes at the same level (brothers) are used to implement polymorphic types and/or multiple interfaces for the same type.
4. It is also possible in C++ to derive one class from two or more prior classes. When that is done, an object of each base class is a part of the derived class.
11.2 Code Example: Publications

The three classes and main program that comprise this example will be presented without notes. The next subsection discusses the important features and semantics of the code. One class in the diagram, Periodical, is not included because it adds little to the discussion.

The Pubs class. Pubs is the base class for deriving kinds of publications.

```cpp
class Pubs {
private: static int pubCount; // Number of Pubs that exist.
protected: const int serial; // Serial number of this instance.
public:
    const string name; // Name given in the declaration.

    Pubs( string g ): serial(++pubCount), name(g) {
        cerr <<"Creating " "name " endl;
    }
    ~Pubs(){
        cerr <<"deleting " "name " leaving " "pubCount " Pubs\n";
    }
    ostream& print( ostream& out) const { // Name, serial number, count.
        out <<name " # " "out of " "pubCount " \n';
        return out;
    }
};
inline ostream& operator<<(ostream& out, const Pubs& x){ return x.print(out); }
```

The Printed class. Printed is derived from Pubs and is the base class for Book.

```cpp
class Printed : Pubs {
private: static int prinCount; // Number of Printed objects that exist.
protected: const int serial; // Serial number of this instance.
public:
    const string name; // Name given in the declaration.

    Printed( string np, string pub ): Pubs(pub), serial(++prinCount), name(np) {
        cerr <<"Creating " "name " based on " "name " endl;
    }
    ~Printed(){
        cerr <<"deleting " "name " leaving " "prinCount " Printed\n";
    }
    ostream& print( ostream& out) const { // Name, serial number, count.
        out <<name " #: " " # " "out of " "pubCount " \n';
        return Pubs::print( out);
    }
};
inline ostream& operator<<(ostream& out, const Printed& x){ return x.print(out); }
```
The Book class. Book is a second-level derived class, with parent Published and grandparent Pubs.

```cpp
/* ---------------------------------------------------------------
 * Public Derivation and static const class member. file: Book.h
 * Created by Alice Fischer on 3/2/09.
 */
#pragma once
#include "Printed.h"
//----------------------------------------------------------------------------
class Book : public Printed {
private:
static int bookCount; // Number of Book objects that exist.
protected:
const int serial; // Serial number of this instance.
public:
const string name; // Name given in the declaration.

Book(string b, string np, string pub):
  Printed(np,pub), serial(++bookCount), name(b){
  cerr <<"Creating " <<name
  <<" based on #" <<serial <<" Printed named " <<Printed::name
  <<endl;
}
~Book(){cerr <<"deleting " <<name <<", leaving " <<--bookCount <<" Book\n";}

ostream& print( ostream& out) const {
  out <<name <<" #" <<serial
  <<"\n\tis " <<Printed::name <<endl
  <<"my Pubs' name is hidden from me." <<endl;
  // out << Pubs::name; // error: Pubs::name is inaccessible.
  // Pubs pp = (Pubs)(*this); // error: Pubs is an inaccessible base of Book.
  // out << p.Pubs::name; // error: Pubs is an inaccessible base of Printed.
  return out;
}
};
inline ostream& operator<< (ostream& out, const Book& x){ return x.print(out); }
```

UML for derived classes. To diagram a derived class, we use a line and a big triangle with its point toward the base class and its flat side toward the derived class(es). The sign written on the lines (+, #, or −) denotes public, protected, or private derivation, respectively. Figure 9.1 gives a UML diagram for the Publications demo program, below.

```
Publication
    +
    |
Online          Printed
    |
    +
Periodical     Book
```

11.2.1 Allocation, Constructors and Initializer Lists

The difference between initialization and assignment is important in both C and C++ because the rules for initialization are more liberal than the rules for assignment. For example, you can and must initialize a const variable but you cannot assign a new value to it.

Allocation, construction and initialization happen in this order:
• Space on the run-time stack is allocated for the core portion of the new object. This allocation includes space for the core portion of all inherited and native data members of the class. Base class members come first in this object, followed by members declared in the derived class, in the order they were listed in the class declaration.

Once the core allocation exists, the extensions must be created and both core portions and extensions must be initialized. These tasks are always done in the same order, starting at the base class. Within a class, initialization starts at the first data member and progresses downward through the declarations of class members. Each additional member given a higher address in memory.

• If the object is of a derived class, the members of the base class must be initialized before the constructor for the derived-class portion is run. A call on a base-class constructor must be the first thing in the list of constructor initializers.

• If the constructor for the base class requires parameters, they must be supplied by a constructor-initializer of this form:

  BaseClassName( argument-list )

If the base class has more than one constructor, the compiler will select the one whose parameter list matches the given list of arguments. If no ctor is given for the base class, the compiler will use the default constructor, if it exists.

• If an object (like a Book object) has a base class (such as Printed) that is also a derived class, the constructor for the middle-level class must pass on parameters to the constructor of the original base class.

• Next, the remaining constructor initializers are used and finally, the code portion of the object constructor is run. This code can do anything. It is usually used to allocate any extension portions of the object, and connect pointers into a legal and meaningful data structure.

11.2.2 Visibility and Protection Level

Protected members. Previously, we have used just two protection levels: private and public. A third level, protected, is intermediate between these two. A public member can be read or written by any part of the program in which its name is known. A private member can be used only by the functions in the same class. A protected class member can also be used by functions of any derived class. In “family” terminology, a Printed keeps his bedroom private, shares protected resources (the home) with his derived class and his second-level derived class, but not with strangers. The base class may provide some resources for public use.

<table>
<thead>
<tr>
<th>Original protection level in Pubs</th>
<th>Style of 1st Derivation</th>
<th>After 1st derivation</th>
<th>After 2nd public derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Accessible in Printed</td>
<td>Protection in Printed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Accessible in Book</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Protection in Book</td>
</tr>
<tr>
<td>pubCount(private)</td>
<td>public</td>
<td>No</td>
<td>private</td>
</tr>
<tr>
<td>serial (protected)</td>
<td></td>
<td>Yes</td>
<td>protected</td>
</tr>
<tr>
<td>name (public)</td>
<td></td>
<td>Yes</td>
<td>public</td>
</tr>
<tr>
<td></td>
<td>protected</td>
<td>No</td>
<td>private</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yes</td>
<td>protected</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yes</td>
<td>protected</td>
</tr>
<tr>
<td>pubCount(private)</td>
<td>private</td>
<td>No</td>
<td>private</td>
</tr>
<tr>
<td>serial (protected)</td>
<td></td>
<td>Yes</td>
<td>private</td>
</tr>
<tr>
<td>name (public)</td>
<td></td>
<td>Yes</td>
<td>private</td>
</tr>
</tbody>
</table>

Figure 12.3: How derivation affects protection level.

\[^1\] Dynamic extensions and static members are allocated in two separate memory segments.
Derivation style. Members of a base class can be declared as private, protected, or public. During a derivation step, the protection level can be kept the same (by using public derivation) or tightened up (by using protected or private derivation). If public derivation is used, inherited members have the same protection in the derived class as in the base class. With protected derivation, public members become protected in the derived class. With private derivation, all inherited members become private. The chart above summarizes the effects of each of the possible protection combinations.

In the chart, the first column lists members of the Pubs class with protection levels, the second lists the three ways the Printed class can be derived from the Pubs class. In all cases, the Book class is derived publicly from the Printed. The third and fifth columns show which members of the Pubs class are visible in the derived class after the first and second derivation steps. The fourth and sixth columns show the protection level of members of the derived class after the first and second derivation steps. From the chart you can see:

- The **protection level** of a member in a derived class (Printed or Book) is the maximum of the protection level in the base class (Pubs) and the styles of all following derivation steps.

- **Accessibility** in the second class (Printed) is determined by the **protection level** of a member in the first class (Pubs). The functions of the derived Printed class can access the public and protected members that have been inherited, but not the private members. The inherited private members are there and take up space in a Printed object, but they are “invisible” to the functions of the Printed class. To use such members, a Printed class function would call a public or protected Pubs function.

- **Accessibility** in the third class, (Book), is determined by the **style** of the first derivation (Printed : Pubs). For example, suppose we tried to write the following code in Book::print():

  ```cpp
  cout <<"\n\tMy Pubs is " << Pubs::name<<'\n';
  ```

  If Printed is derived privately from Pubs, this code generates a compile-time protection error: “member ‘Pubs::name’ is private in this context”. However, the same code in Printed::print() is legal because in the context of the Printed class, name is public.

  In earlier versions of standard C++, the Pubs data could be accessed by using a relabeling cast to convert this to a Pubs* and use the result to access the name:

  ```cpp
  out <<" my Pubs is " <<((Pubs*)this)->name <<endl;
  ```

  In the current ISO standard C++, this cast is prohibited, and there seems to be no way that a function in the Book class can access data in the Pubs class, even when that data is public.

  In contrast, if protected derivation is used to derive Printed from Pubs, the same code is legal in the Book class. It is OK because Pubs::name is protected in the Printed class and, therefore, visible within Book.

### 11.2.3 Resolving Ambiguous Names

It is normal for a base class and a derived class to have methods with the same name. In this case, we say that the method in the derived class **overrides** the base method. This relationship is the basis of polymorphism (covered in a later chapter). Most of the time, it causes no confusion to have two methods (or data members) in the same class hierarchy that share the same name. If a function method in Book refers to a class member named **serial**, the instance of **serial** in class Book will be used. If Book did not have such a member, the compiler would look for an inherited instance of **serial**. (It will look in Printed, then continue searching up the hierarchy until a definition of **serial** is found.)

Sometimes, however, a class may contain a definition for **serial** but one of its methods must refer to a different instance of **serial** defined in the base class. For example, to print all the Book data, we need to print the inherited members from Printed and Pubs. To do this, we must be able to call the print method in the base classes. That is the purpose of the **scope-resolution operator**, which is written with two colons (::).

In the demo program, all three classes have members named **serial, name**, and **print()**, so these names are ambiguous. On line 69 of Book::print() we write <<**name** to print Book::name. On the next line, we want to print the name from the base class, so we write <<**Printed::name**.
out <<name <<" #" <<serial
<<"\n\tis " <<Printed::name <<endl
<<"\tmy Pubs' name is hidden from me." <<endl;

On line 78, we want to print Pubs::name, but this member is not visible to the Book functions because of the
private derivation step even though this member is public for the world at large (a strange combination). Lines
800 through 82 show a variety of other things that do not compile.

In earlier versions of standard C++, the Pubs data could be accessed by using a relabeling cast to convert
this to a Pubs* and use the result to access the name. In the current ISO standard C++, even this cast is
prohibited, and there seems to be no way that a function in the Book class can access data in the Pubs class,
even when that data is public.

11.2.4 Inherited Functions

Functions (as well as data members) are inherited, and the protection rules for inherited functions are the same
as for inherited data members. Inherited functions play an important role in maintaining privacy: they enable
an object to use its inherited private parts that otherwise would be "invisible" to objects in the derived class.
The derived class can use the inherited function without modification or redefine it. A redefinition can take the
form of an extension or an override.

Function redefinition. The functions of a class are closely tied to the representation of the class. Also,
certain functions should be defined for every class (print, operator<<) using the same name and with the
same general meaning (output the values of all data members in an appropriate format). Taking these two facts
together, we see that function-naming conflicts will almost always occur between a derived class and its base
class.

An extension is a redefinition of the function that calls the inherited version and also does additional work.
Almost every class needs to have a redefinition of the print() function. Normally, this redefinition will call the
inherited function using the scope-resolution operator, like this: return Pubs::print( out );

An override is a redefinition that fundamentally changes the meaning of the inherited function or blocks
access to it completely, preventing any further derived classes from using it. In each of the two derived classes
above, a new method for print() is defined that overrides the inherited method. As is typical, the function in
the derived class prints some data itself, then calls the inherited function to print the rest of the data.

Cancellation by redefinition. In a derivation hierarchy, a middle-level class sometimes overrides an
inherited function by replacing its original actions by nothing. The effect is to remove the function from the
set of functions available to classes further down the inheritance hierarchy. This is a powerful tool, but rarely
used.

11.3 Class Derivation Demo

Below is a main program that instantiates some Pubs, Printeds and Books and uses them to illustrate the
semantics of derivation and inheritance.

The main program.

```cpp
88 // -----------------------------------------------------------
89 // Class Derivation and Static Class Members  file: main.cpp
90 // A. Fischer  March 2, 2009
91 // -----------------------------------------------------------
92 #include "Pubs.h"
93 #include "Printed.h"
94 #include "Book.h"
95
96 int Pubs::pubCount = 0;  // Number of Pubs objects that now exist.
97 int Printed::prinCount = 0;  // Number of Printed objects that now exist.
98 int Book::bookCount = 0;  // Number of Book objects that now exist.
```
11.3. CLASS DERIVATION DEMO

```c
//----------------------------------------------------------------------
int main( void ) {
Pubs A("A-Wesley");
Printed B("Trade Books", "Pearson");
cout <<endl;
Book D( "Anatomy", "Out-of-print", "P-Hall" );
cout <<endl;
Book G( "Applied C", "Textbook", "McGraw-Hill" );
cout <<"\nThe population is:\n"<< A << B << D << G;
bye();
return 0;
}
```

The creation of one Book. The last object created by main() is a textbook called “Applied C”. Here is the trace of what happens on line 117 when G is declared.

Creating McGraw-Hill
Creating Textbook based on #4 Pubs named McGraw-Hill
Creating Applied C based on #3 Printed named Textbook

You can see that the base class is constructed first, then Printed, then Book.

Creating McGraw-Hill
Creating Textbook based on #4 Pubs named McGraw-Hill
Creating Applied C based on #3 Printed named Textbook

You can see that the base class is constructed first, then Printed, then Book. When creation of four publishers, three classifications of books, and two books, the entire lot is printed (line 118).

The population is:
A-Wesley #1 out of 4
Trade Books: #1
whose base object is Pearson #2 out of 4
Anatomy #1
is Out-of-print
my Pubs’ name is hidden from me.
Applied C #2
is Textbook
my Pubs’ name is hidden from me.

After program termination, the trace comments from the destructors appear, in the opposite order of their creation:

Normal termination.
deleting Applied C, leaving 1 Book
deleting Textbook, leaving 2 Printeds
deleting McGraw-Hill, leaving 3 Pubs
deleting Anatomy, leaving 0 Book
deleting Out-of-print, leaving 1 Printeds
deleting P-Hall, leaving 2 Pubs
deleting Trade Books, leaving 0 Printeds
deleting Pearson, leaving 1 Pubs
deleting A-Wesley, leaving 0 Pubs

11.3.1 Inherited Data Members

Each derivation step adds members to the ones present in the base class because the base-class members are inherited by the derived class. An object of the derived class starts with the data members of the base class, followed by the new members declared within the derived class. You could say that the derived object is an extension of the base object. (To do the same derivation in Java, you would write class Printed extends
Some classes have static data members that are allocated in a non-contiguous part of memory. These members are also inherited.

For example, the first part of a Printed object is a Pubs object. In addition, a Printed has three more data members (name, serial, and prinCount) making a total of six data members in a printed object. Similarly, as shown in the diagram below, the first part of a Book object is a Printed object, to which Book adds another static member (bookCount) and two more ordinary members (name, serial), for a total of nine data members. A Book has all these members even though some of them are allocated remotely and others are private and cannot be accessed from methods in the Book class.

The data diagram above shows the last two objects allocated by main(): two Books named D and G. Each has nine data members: three shared members in the static storage area and six instance members. Medium gray denotes the members that were inherited from the Pubs class; those that came from the Printed class are light gray. Null characters are denoted by a backslash. This diagram is complicated by the fact that all three classes have static members that are stored in another area of storage. The three static members are shared by all Book objects in the program.

Access to inherited members. Various methods in the Printed and Book classes access inherited members.

- The Printed constructor prints trace comments that include two data members from the base class: Pubs::serial and Pubs::name. The scope-resolution operator must be used to print them because the Printed class has members with the same names.
- The inherited member is visible in the Printed class because Pubs::serial is protected (not private) and Pubs::name is public. The derivation method was private derivation, but that does not restrict visibility at this level.
- The Printed::print() method uses the scope-resolution operator to call the inherited print() method. Without the Pubs::, this would be a recursive call (a bug).
- The constructor and print() method in Book do similar things using Printed::.
- Since private derivation was used to create Printed, all the members of the Pubs class become private in Printed. Therefore, Book cannot see any of them, even those that were originally public. The compiler will not compile anything in the Book class that tries to breach the privacy of the Pubs class.

11.4 Derivation and Inheritance

In life and in understanding a complex program: A picture is worth a thousand words.

This chapter consists of one interactive game program that uses vectors, derivation, some C coding “tricks” that are worth knowing. The game output is presented first to familiarize you with how the game works. Following that are the makefile, main program, and pairs of .hpp and .cpp files, with notes on each. These are documented by several kinds of “pictures”, each of which illustrates a different aspect of the application: a module dependency chart, a data structure diagram, UML class diagrams, and a function call chart.
11.5 Playing Hangman

This program plays an interactive word-guessing game called hangman. In this game, the leader (the computer) selects a secret word and displays a line of dashes with one dash for each letter. The player must guess letters, one at a time, and try to figure out what the hidden word is. A sample game is included here, for those who are unfamiliar with it. Suppose the computer chose “hippopotamus” as the secret word. The player would see:

---------Constructing Hangman ---------
Please enter name of vocabulary file (or ENTER to quit): vocab2.in

--------- Welcome to Hangman ---------
You win if you can guess the hidden word.
You lose if you guess 7 wrong letters.

Puzzle is: <[ _ _ _ _ _ _ _ _ _ _ _ _ ]>

       Letters left--> a b c d e f g h i j k l m n o p q r s t u v w x y z
     Bad guesses---> _ _ _ _ _ _ _

Guess a letter:

After one wrong guess (e) and one correct guess (a), the board would look like this:

       You guessed 'a'. You scored!

Puzzle is: <[ _ _ _ _ _ _ _ _ a _ _ _ ]>

       Letters left--> b c d f g h i j k l m n o p q r s t u v w x y z
     Bad guesses---> e _ _ _ _ _ _

After several more guesses (i,o,y,u,t,p,m, and finally s), the game is won and you see:

       Puzzle is: <[ h i p p o p o t a m u s ]>

       Letters left--> b c d f g j k l n q r v w x z
     Bad guesses---> e y _ _ _ _ _

Congratulations -- you win!

You won 1 time out of 1 try.

Type p to play another round, q to quit: q

----------- Have a good day! -----------

11.5.1 Hangman: The Main Program

Using the ? : operator. A conditional operator is used appropriately on line 10 to download the optional file name from the command line. It tests whether command line arguments exist, and if so, returns the c-string supplied for arg[1]. (Remember, arg[0] is always the name of the command being executed. a C/C++ syntax requires that the same type of object must be returned by both clauses of a conditional operator; nullptr is a pointer, as is the pointer to the cstring that might be returned. If no name was typed on the command line, the program uses a default file, vocab.in. The overall code is simplified considerably by using the conditional operator instead of an if-else statement.

Using const. Throughout the application, const is used wherever it makes sense. Your production code should follow this pattern.

The vocabulary file. This program is designed to construct a hangman game from a file of puzzle words, then play one or more rounds of it. Line 12 opens the file, Lines 13–14 verify that it is properly open. Line 16 was useful during program construction to track progress.
11.5.2 The Game Class

The main program creates a Game on line 17. Game g is in charge of one game session. A new gameboard is created by Game::play() (lines 38, 75) for each round. Since the array members of Board are built around the hidden word, each new word requires a new construction job. We could have done the same thing by declaring a Board* as a member of Game, but there would be no advantage in doing it that way because there is no need to use a board after returning from Game::play().

Notes on the Game constructor. The Game constructor’s first task is to read the vocabulary file; an open ifstream is the parameter. The first line of the file must contain the alphabet that is used in the rest of the file. That input is used (Line 52) to initialize the class member alphabet. Then the file is read in (Lines 53–57), stored in strings, and pushed into v, the vocabulary vector. Last, the constructor initializes the random number generator to enable random selection.

Game class declaration

    class Game {
    private:
        string alphabet; // Normally A..Z (English)
        vector<string> v; // vocabulary

    public:
        Game( ifstream& wordfile );
        Game(); =default;
        void play(); // Play hangman.
        const string randword();
        void print( ostream& outs ) const; // for debuggong

    };

Notes on the Game::play() function. This is the main logic of the program. There are three phases:

- Display the instructions. (Lines 69–71)
- Play one or more rounds (Lines 72–88), Keep an overall score; end when the user chooses to quit, line 86.
- Print a polite goodbye message. (Line 87)
Game class implementation

```cpp
#include "game.hpp"

Game::Game(ifstream& infile) {
    string temp;
    getline( infile, alphabet );
    for (;;) {
        // Create vocabulary database from file.
        infile >> temp;
        if( !infile.good() ) break;
        v.push_back(temp);
    }
    infile.close();
    srand( (unsigned int) time(nullptr));
}

void Game::play() {
    int rounds = 0; // Number of wows the player has tried to guess.
    int wins = 0; // Number of times the player has won.
    char response; // For query, "Play again?"
    
    cout << "---------- Welcome to Hangman ----------";
    cout << "You win if you can guess the hidden word."
    cout << "You lose if you guess " << HANG_MAX << " wrong letters."
    do {
        if (v.size() < 1) fatal("No more words in vocabulary!");
        string puz( randword() );
        Board b(alphabet, puz);
        rounds++;
        wins += b.play(); // Play one round of game.
        string timeword ((wins == 1) ? "time" : "times");
        string tryword ((rounds == 1) ? "try" : "tries");
        cout << ";You won " << wins << " " << timeword
        << " out of " << rounds << " " << tryword
        << ";You lose if you guess " << HANG_MAX << " wrong letters."
        if (v.size() < 1) fatal("No more words in vocabulary!");
        string puz( randword() );
        Board b(alphabet, puz);
        rounds++;
        wins += b.play(); // Play one round of game.
        string timeword ((wins == 1) ? "time" : "times");
        string tryword ((rounds == 1) ? "try" : "tries");
        cout << "You won " << wins << " " << timeword
        << " out of " << rounds << " " << tryword
        << ";You lose if you guess " << HANG_MAX << " wrong letters."
        if (v.size() < 1) fatal("No more words in vocabulary!");
        string puz( randword() );
        Board b(alphabet, puz);
        rounds++;
        wins += b.play(); // Play one round of game.
        string timeword ((wins == 1) ? "time" : "times");
        string tryword ((rounds == 1) ? "try" : "tries");
        cout << "You won " << wins << " " << timeword
        << " out of " << rounds << " " << tryword
        << ";You lose if you guess " << HANG_MAX << " wrong letters."
        if (v.size() < 1) fatal("No more words in vocabulary!");
        string puz( randword() );
        Board b(alphabet, puz);
        rounds++;
        wins += b.play(); // Play one round of game.
    } while (tolower(response) != 'q');
    cout << "----------- Have a good day! -----------";
}

const string Game::randword() {
    int r = rand() % v.size();
    string ret = v.data()[r]; // Grab word that was selected.
    v.data()[r] = v.data()[v.size()-1]; // Replace by last word in array.
    v.pop_back();
    // cout << ret << endl; // for debugging
    return ret; // Return chosen word.
}
```

Playing one round.

- Line 72: The game logic is inside the do loop. Lines 73–85 play one round and display the results.
  - Line 73 ends the game if all words in the vocabulary have been used. In real use, this will not happen.
Line 74 selects a random word from the vocabulary. (The selection code is on lines 92–99.) The mystery word is sent to the Board constructor where it is used to construct a playing board for the current round.

Lines 76–77 keep score for the entire Game session, not just the one word. Line 77 calls Board::play and adds the return value (1 for a win, 0 for a loss) to the score. Details of how to play the board and how to calculate the score are delegated to the Board class.

Lines 79–82 report the results of the current round to the user. Lines 79 and 80 are a small programming trick to achieve grammatical output.

The randword function. Game::play() calls randword() on line 74 to select a random string from the vocabulary. We are using an algorithm called “random selection without replacement”. The function has two tasks:

- Select a word. Easy. Call rand to get a random integer and use % to scale it to the current length of the vocabulary.
- Remove the chosen word from the vocabulary so that it will not be selected again. First, make a copy of the selected word in a temporary. Then, move the last word in the vocabulary into the newly vacated array slot. Finally, delete the last word by calling pop_back().
- Return the selected word.

11.5.3 The Board Class

To play a round of hangman, we need a hidden word and an list of unused letters. To make a good interface, we also need a list of wrong guesses. These three arrays of characters (puzzle, alpha, and errors) all have corresponding bit masks that define which letters should be displayed, in each category, after each move. All three are updated and after every guess to help the player make skillful guesses. The Alphabet and HangWord classes define variations on a string with a parallel mask array. Details of how the masks work are left to descriptions of the Baseword, Hangword, and Alphabet classes.
Notes on the Board declaration.

- It is much easier to believe in the correctness of a function when its return values have meaningful names than when integer codes are used. For this reason, we define a private enumerated type to describe the possible outcomes of a guess. The type Status is an enum class, a recent addition to C++. In most ways, an enum class is like an enum, and it is used like an enum type. However, the translator lumps together all the symbols defined in all the old-style enums, and they are all global. An enum class is a scoped, segregated version of the old enum, and gives the programmer a place to put relevant input and output functions. It is considered better practice to use enum classes.

- The enum class defined on line 109-110 lists symbols that are used in categorizing a player’s guesses. The categories are analyzed in the guess() function and used in the move function.

- The errors is used for keeping score: if it reaches 7, the player loses the round.

- found is used to count the number of times the player’s guess-letter is found when searching the puzzle string for matches.

- After each guess, the Status is reported. Then all three masked arrays change and the display changes.

- alpha is the alphabet. Each time a letter is guessed, a bit is turned off in the alphabet’s mask, and that letter will disappear from the display.

- Simultaneously, a bit in the mask for errors is turned on, and the bad guess will then appear in the list of errors.

- If the guess is correct, the letter will appear in the appropriate spots in the puzzle word.

```
126  // ==============================================================
127  // board.cpp: Implementation for hangman board
128  // A. Fischer, August, 2022
129  #include "board.hpp"
130  // ==============================================================
131  Board::Board( const string a, const string puz ):
132      alpha(a, true), errors(a, false), puzzle(puz) {
133      //cerr << "\nConstructing Board. ";
134  }
135  // ==============================================================
136  Board::play() {
137      print(cout);
138      while (errcnt < HANG_MAX && found < puzzle.len() ) move();
139      if (found == puzzle.len()) {
140          cout << "Congratulations -- you win!" << endl;
141          return 1;
142      }
143      cout << "Sorry, you lose!" " \nThe answer is: " << puzzle << endl;
144      return 0;
145  }
146  // ==============================================================
147  Board::move() {
148      char ch;
149      cout << "Guess a letter: ";
150      cin >> ch;
151      cout << "You guessed " << ch << "\n";
152      switch (guess( ch )) {
153        case Status::NOT_IN_ALPHA:
154            cout << " -- but it’s not in the alphabet." << endl; break;
155        case Status::USED_ALREADY:
156            cout << " -- but you guessed it once before." << endl; break;
157        case Status::BAD_GUESS:
158            cout << " -- too bad." << endl; break;
159        case Status::GOOD_GUESS:
160            cout << ". You scored!" << endl; break;
161      }
162      print(cout);
163  }
```
// ----------------------------------------------------------- process a guess
Board::Status
Board::guess(char c) {
    int where = alpha.find(c);
    if (where == -1) return Status::NOT_IN_ALPHA;
    if ( !alpha.showSlot(where) ) return Status::USED_ALREADY;
    alpha.showSlot(where) = false;
    int matches = puzzle.tryLetter(c);
    if (matches <= 0) {
        errors.showSlot(where) = true;
        errcnt++;
        return Status::BAD_GUESS;
    }
    found += matches;
    return Status::GOOD_GUESS;
}

// --------------------------------------------------------- display the board
ostream&
Board::print( ostream& out ) {
    out << "Puzzle is: " <<puzzle << "\n\n";
    out << " Letters left-->" <<alpha <<"\n";
    out << " Bad guesses-->" <<errors;
    for (int k = errcnt; k < HANG_MAX; k++) out << " _";
    return out << endl;
}

Notes on the Board implementation
The Board class plays one round of hangman, with the help of classes Alphabet and HangWord, both derived from BaseWord. During the round, the Board object must keep score and end the round when the number of correct guesses (round) equals the length of the puzzle word, or when the number of bad guesses (errors) reaches the limit, HANG_MAX.

The constructor does all of its work in ctors (line 132). The trace comment, now commented out, was helpful during program construction. This is what the Board looks like just after construction:

Puzzle is: <[ _ _ _ _ _ ]>
Letters left--> a b c d e f g h i j k l m n o p q r s t u v w x y z
Bad guesses--> _ _ _ _ _ _

Guess a letter:

The play() function.

• Line 139: Let the player keep making moves until player either loses or wins.
• Lines 140–143: If the player has found all the letters in the puzzle, return 1 to mean “won one game”.
• Otherwise, return 0, indicating a loss and 0 games won.

The move() function. Move is straightforward. It prompts the player for a guess, reads it, and calls guess() to analyze its category. When —it guess() returns, move displays the results to the player.

The guess() function.

• A value of type Status is returned by guess() (Line 182) and used by move() to select a response message for the player. We use an enum class to clarify, simplify, and modularize the code. The resulting two functions are much clearer that the alternatives: cryptic integer codes for the outcomes.
• The return type must be given on line 168 as “Board::Status”, even though it is written simply as “Status” on line 123. This is necessary because the Status type is defined inside the Board class and this function definition is outside the class declaration.

• Given a puzzle, an alphabet, and a guess, there are four possible outcomes. We do a case analysis and use four return statements to simplify the logic by keeping the cases maximally separate from each other.

• If the guess is not in the game-alphabet or it has been guessed previously, the player will not be “charged” for the bad guess. To find out, we call Alphabet::find() (line 170) to search for the guess among the remaining legal letters. The return value is the subscript of the guess in the alphabet, if it is a legal letter, otherwise -1. For example, when ‘e’ is guessed, the subscript 4 will be returned because ‘e’ is the fifth letter in the alphabet.

Puzzle is: <[ e _ _ _ _ ]>

Letters left--> a b c d f g h i j k l m n o p q r s t u v w x y z
Bad guesses--> _ _ _ _

Guess a letter:

• If the letter is valid, we then check (line 172) whether it was previously guessed. To do this, we use the subscript returned by Alphabet::find() to index the mask array that parallels the alphabet array. (A result of false means the letter has been used, true means it is still available.) A duplicate guess causes this display:

Guess a letter: e
You guessed ‘e’ -- but you guessed it once before.

• If the letter is legal and still possible, we set the appropriate position in the Alphabet mask array (line 173) to “false” to indicate that the letter is now used. For example, if we guess ‘e’, we see that alpha.mask[4] is false and ‘e’ has disappeared from the display.

• On line 175, we send control to Hangword::try_letter() to compare the guess to the letters in the puzzle word. It might be a wrong guess, or it might match 1 or several letters in the puzzle word. The result returned by try_word is the number of letters in the puzzle that match the guess (zero or more).

• If the answer is zero, we set the corresponding mask position of the error array to “true”, increment the error-counter, and return (Line 190) with an error code. This will cause the letter to “appear” (in alphabetical order) in the error array the next time it is displayed.

Puzzle is: <[ e _ _ _ _ ]>

Letters left--> a b c d f g h i j k l m n o p q r s t u v w x y z
Bad guesses--> s _ _ _ _

Guess a letter:

• Line 182: The last step is adding the number of matches for the current letter to the number for this round, and returning the Status code.

The print() function.

• Our gameboard displays three arrays of letters:

  – Puzzle, the mystery-word consisting of underscores and correctly guessed letters.
  – Errors, a list of incorrect guesses followed by underscores for additional possible bad guesses.
  – Alpha, the list of letters that have not yet been guessed.

11.6 The Word Classes

A mask marks a subset. A masked data structure is the simplest way to denote a subset of the data that is to be used (or not used) for a particular purpose. One or several masks might be used to mark one or several different subsets. The mask field or fields might be members of the structure or might exist in a parallel data
structure. The mask fields might be type bool (to denote a simple yes/no choice) or any other enumerated type. Masks are most useful if the condition that they represent is not simple to test for, and if the status of an object changes over time or is checked frequently.

A masked structure lets us sort or process data efficiently and easily. For example, suppose a club membership database has one record for each member. At least two sets of masks might be helpful: one for age (juvenile, teenager, adult, senior) and another for dues status (guest, lifetime, paid-up, due, lapsed). Masks can be useful here because age is messy to categorize and dues status is based on the member’s history.

11.6.1 The BaseWord Declaration

The class BaseWord creates a basic masked string, that is, a string with a corresponding array of bools to indicate whether each letter in the string is currently valid or not. If a letter is valid, it is searched and displayed. If not, it is hidden both from sight and from the computations. You could say that the mask controls access to the individual letters in the array. This is an easy and fast way to select a subset of the data stored in an array. To add an element or remove it, simply change the bit from true to false, or vice versa.

```cpp
// ==========================================================================
// Maskable word base class.
// A. Fischer and M. Fischer, August 2022 file: words.hpp
#pragma once
#include "tools.hpp"
// ========================================================================
class BaseWord {

protected:
  const string word; // partially concealed word,
  size_t length; // simple name for word.length().
  bool* show; // show marks which letters to reveal.

public:
  BaseWord(const string st) :
    word(st), length(st.length()), show(new bool[length]){}
  ~BaseWord() { delete[] show; }

  void setAll(bool on_off){ // set false to hide letter, true to expose.
    for (int k=0; k<length; k++) show[k] = on_off;
  }

  bool& showSlot(int k) const { return show[k]; }
  size_t len() const { return length; }
};
```

The BaseWord class. In this class, everything is based on a particular string (the argument to the BaseWord constructor) and its length. The members named word, length and show are constants, so they must be initialized using ctors, after the length of the argument string is known. Class members are initialized in this order:

- The word itself must be brought in from the argument list.
- Having the word, we can get its length.
- Having the length of the word, we can allocate a mask of the same length.

Note that the class members should be written in the same order as the ctors.

Line 208: The class member named word is a constant string, either the alphabet or a word that was selected randomly from the vocabulary and is still stored there. No copy is made of this string because we do not need to modify it. The word, itself, is constant. The mask array is modified during the play, and those modifications determine which letters are displayed.

Line 216: The setAll() function initializes all of the booleans in the mask array, to either true or false, depending on the argument. For the alphabet, all should be initially true. For the list of errors, all should be initially false. When a bit is turned off in the alphabet, it is turned in in either the error list or the puzzle word. In that way, letters move from one line of the display to another.
Line 219: The `showSlot()` function takes a subscript as an argument and returns the boolean value at that position in a mask array. This is used by `Board::guess()` in the process of analyzing the player’s guess.

Line 220: The `len()` function is an accessor. It was defined to provide a short name because it is used in many places.

### 11.7 The Derived Word Classes

Both of these words are masked arrays derived from `BaseWord`. This is non-polymorphic derivation because there are no virtual functions.

- The first line of a class declaration declares the derivation relationships, if any. In this game, `Alphabet` and `HangWord` are both derived from `BaseWord` by public derivation:

  ```cpp
  class Alphabet : public BaseWord { ... };
  class HangWord : public BaseWord { ... };
  ```

- The two new classes are derived from `BaseWord` so that we can have the same structure but different print functions. Also, one class has a `find()` function, the other has a `tryLetter()` function.

```cpp
class Alphabet : public BaseWord {
public:
  Alphabet(const string st, bool on_off) : BaseWord(st) {
    setAll(on_off);
    //cerr << "Constructing Alphabet."
  }
  int find(char c) const; // return index of first c in word
  ostream& print (ostream&); // print an alphabet
};

inline ostream& operator<<(ostream& out, Alphabet& x){ return x.print(out); }
```

```cpp
class HangWord : public BaseWord {
public:
  HangWord(const string st) : BaseWord(st) {
    setAll(false);
    //cerr << "Constructing HangWord. " <<st;
  }
  int tryLetter(char);
  ostream& print (ostream&); // print a hang word
};

inline ostream& operator<<(ostream& out, HangWord& x){ return x.print(out); }
```

- The public/protected/private status of all the inherited members, in `Alphabet` and `HangWord`, is the same as in class `BaseWord`.
- The two derived classes are in the same file because they are both very small and the author believes putting them in one pair of files makes reading and comprehending the code easier.
- The first part of an `Alphabet` or `HangWord` object is a `BaseWord` object.
- The constructors for `Alphabet` and `HangWord` must use ctor initializers to provide arguments for the `BaseWord` constructor.
- Functions in the derived class have access to the members of the base class because public derivation was used.
• The print functions and operator extensions in these classes provide a visual interface that was not provided by the base class. They can freely use the protected members of the base class: len, word, and mask.

```
// ==========================================================================
// Implementation for maskable words.
// A. Fischer, June 4, 2000 file: words_d.cpp

#include "tools.hpp"
#include "words_d.hpp"

// Implementation for maskable words.

// =========================================== Alphabet class functions
int Alphabet::find(char c) const {
    int k;
    for (k=0; k<length && c != word[k]; k++); // Loop body is empty.
    return (k==length) ? -1 : k;
}

ostream& Alphabet::print (ostream& out) {
    for (int k=0; k<len(); k++) if( show[k] ) out << " " << word[k];
    return out;
}

// ================================================= Hangword class functions
int HangWord::tryLetter(char c) {
    int count = 0;
    for (int k=0; k<len(); k++) {
        if (c == word[k]) {
            count++;
            show[k] = true; // show this letter in the puzzle
        }
    }
    return count;
}

ostream& HangWord::print (ostream& out) {
    out << "<[ ";
    for (int k=0; k<len(); k++) out << ' ' << (show[k] ? word[k] : '_') ;
    out << " ]>");
    return out;
}
```

Implementations of the Alphabet and HangWord classes. Three functions deserve comments:

• Alphabet::find() is a sequential search of the alphabet for a copy of the players input. The loop ends if and when the letter is found. The conditional operator in line 281 returns this position or -1, an error code. This is very compact code, but is well within the bounds of readability.

  This is an unusual loop because all the work is done in the control portion of the loop. There IS no loop body. It is a good idea to put a comment on any loop like this to avoid introducing errors during program modification or maintenance.

• HangWord::tryLetter() searches the puzzle word for copies of the player’s guess. If found, the mask is set to true in the corresponding position and the “score” is counted.

• Both print functions use a one-line loop and a conditional operator to check the mask and print either a puzzle letter or a dash. A multi-line if statement could be used, but it
11.8 UML Diagrams: A View of the Class Relationships

A UML diagram gives a static view of the application; it illustrates the data types being used, the protection level of each part of each class, and the possible ways that one class can access or use another.

The classes used in this application fall into two nearly separate subsystems: Game + Board, and the three Word classes. These two subsystems are diagrammed side-by-side.

![UML Diagram](image)

Figure 11.1: UML describes class relationships.

Note that the letter ‘C’ is listed at the right side of some boxes, after a function. This is my way of denoting that a function is constant, that is, it does not modify its implied parameter. Likewise, “c.string” is used to mean “const string”.

Game and Board. In the UML diagram we see that the Board is NOT a member of the Game class; it is instantiated in Game::play() because each round of the Game is based on a different random word. The information needed to instantiate a Board is not available until after play() begins executing.

Putting things in the right class makes programs simpler. In an earlier version of the program, Board was a data member of Game. This led to the use of pointers, the need for a default constructor, and confusion about when and where to delete the board.

The enum class. There is no place in a traditional UML diagram to put a class that is defined within another class. In this diagram, it is listed at the bottom of the class-box, where I usually put additional relevant information.

The Word classes. From the UML, you can see that the BaseWord class is not associated with or aggregated by any other classes and, in fact, is not instantiated by the program. It is used only as a basis for deriving Alphabet and Hangword, which form the basis for the gameboard display. This follows a basic OO-design guideline:

Don’t instantiate a class that you also derive from.

The goal of this rule is to minimize the conflict between the requirements of a base class, which must be clean and general, and a class that must serve the specific needs of an an application.
In Figure 11.1, we see that two classes are derived from BaseWord: an upward-pointing triangle is used to symbolize derivation. The two + signs indicate that public derivation was used in both cases.

BaseWord defines a data structure and a set of functions that implement one basic behavior. From it, we derive two sub-classes that define variations on the basic theme and have different initialization, search, and display rules. Neither derived class has data members – therefore the section of the UML box is empty. None of the functions in the derived classes override anything defined in the base class. This is a very simple example of non-polymorphic derivation.
Chapter 12: Polymorphism and Virtual Functions

From Lewis Carroll, *Through the Looking Glass*:

“When I use a word,” Humpty Dumpty said, in rather a scornful tone, “it means just what I choose it to mean – neither more nor less.”

“The question is,” said Alice, “whether you can make words mean so many different things.”

“The question is,” said Humpty Dumpty, “which is to be master— that’s all.”

12.1 Basic Concepts

In this chapter, we introduce virtual functions and two complex and powerful uses for derived classes that virtual functions support: abstraction and polymorphism.

12.1.1 Definitions

We need to clearly distinguish between the two kinds of derivation:

**Simple derivation.** In simple derivation, two application classes are derived from the same base class. This kind of derivation serves two important purposes:

- To factor out the common parts of two closely related classes so that the common code does not need to be written twice.
- To derive a class from the instantiation of a template with a particular type. If we need a `vector<Item>`, we could define a convenient name for it by deriving a new class from the instantiation of the library class:

  ```cpp
  class PartList : vector<Item>
  ```

- To facilitate reuse of library classes and templates, we can wrap a new class around the library class. The derived class will contain only functions that are useful to the application. It will block access to all other functions in the base class.

**Polymorphic derivation.** Sometimes a class has multiple subclasses and one or more functions that apply to all of them. Now imagine a list of employees. All employees must be paid, so we have a `calculatePay()` function in the Employee class. However, the formula for the pay calculation is different for professional staff and union members, so we want the method for `calculatePay()` to be different for managers than it is for clerks. A function with one name that is implemented by three or more defining methods. In C, one would accomplish this with a switch statement. In C++, classes and virtual functions provide a nicer solution for defining a function with one name that is implemented by multiple defining methods.

For any given call on `calculatePay()`, we want the appropriate method to be used for the calculation. Now suppose we have declared four objects:
Manager M;
Accountant A;
Clerk C, D;

Then if we call A.calculatePay() or M.calculatePay() we want the calculation to be made using the formula for professional staff. If we call C.calculatePay() or D.calculatePay(), we want the union formula used.

Virtual functions solve this problem. The purpose of a virtual function is to have one name, one prototype, and more than one definition so that the function’s behavior can be appropriate for each of the derived classes.

A virtual function is a function in a base class that forms part of the interface for a set of derived classes. It is declared virtual in the base class and may or may not have a definition in that class. It will have definitions in the derived classes if a behavior is needed that differs from the method in the base class.

12.1.2 Virtual functions.

A virtual function defined in the Employee class (Figure 12.1), is shared by the classes in the derivation hierarchy. (Note: In this sketch, three shapes have been drawn on top of the ordinary rectangular class boxes. This is not proper UML; the shapes will be used later to explain polymorphism.)

We create a virtual function when the same task must be done for all objects of all types in the hierarchy, but the method for doing this task depends on the particular representation of an object. For example, suppose the function calculatePay() must be defined for all employees, but the formula for the calculation is different for union members and professional staff. We want a function with one name that is implemented by three or more defining methods. For any given function call, we want the appropriate function to be called. For example, suppose we have declared four objects:

Manager M;
Accountant A;
Clerk C, D;

Then if we call A.calculatePay() or M.calculatePay() we want the calculation to be made using the formula for professional staff. If we call C.calculatePay() or D.calculatePay(), we want the union formula used.

We implement this arrangement by declaring calculatePay() as a virtual function in the Employee class, with or without a general method in that class. If defined, the general method might do the parts of the calculation that are common to all classes. We also define calculatePay() in each derived class to make the calculation appropriate for a specific type of employee. When calculatePay() is called, the system must select one of the methods to use. It does this by looking at the specific type of the object in the function call and using the right method for that type.

Choosing the right method. Choosing the right method is called “dispatching the function call”. It normally happens at compile time, However, for virtual functions a specific method is chosen at run time by looking at the specific type of the argument in the call.

Syntax. For examples, look at print() in Container, Linear and Queue, later in this chapter.

- The prototype for a virtual function starts with the keyword virtual. The rest of the prototype is just like any other function.
- If a virtual function has no definition in the base class, then its prototype ends in =0 instead of a semicolon.
- Any class with one or more virtual functions must have a virtual destructor.

How it works. Every object that could require dynamic dispatch must and will carry a type tag (one or a few bytes) at run time that identifies its relationship to the base class of the class hierarchy. (1st subclass, 2nd subclass, etc.) This is necessary if even one function is declared virtual, either in the class itself or in a parent class. Doing this takes extra space and extra run time.

The run-time system will select the correct method to use for each call on a virtual function. To do so, it uses the type tag of the implied parameter to subscript the function’s dispatch table. This table has one slot for each class derived from the class that contains the original virtual declaration. The value stored in slot $k$ is
the entry address for the method that should be used for the $k$th subclass derived from the base class. This is slightly slower than static binding because there is one extra memory reference for every call on every virtual function.

12.2 Creating a Polymorphic Container

Overview of the demo program. The major example in this chapter is a polymorphic implementation of linear containers. The class Container is an abstract class from which linear containers (lists, queues) and non-linear containers (trees, hash tables) could be derived. Container supplies a minimal generic interface for container classes (those that can be used to store collections of data items).

In this chapter, we focus on linear containers. The class Linear is the base class for a polymorphic family of list-based containers. From it we derive the data structures Stack and Queue in this chapter, List and Priority Queue come later. All of these conform to the Container interface and Linear implementation strategy but present different insertion and/or deletion rules. We use the containers in this chapter to store objects of class Exam, consisting of a 3-letter name and an integer.

To a great extent, these containers resemble the template library containers. The purpose for presenting them here is to supply concrete code examples to help the student understand abstract classes, abstract function, derivation, and polymorphism.

The classes Linear and Cell should be defined as templates so that they do not depend on the type of the data in the list. Templates were not used here because they would complicate the structure of the program and the important issue here is the structure of a polymorphic class. In a real application, both would be used.

What to look for.

- A virtual function can be initially declared with or without a definition. For example, pop() is declared without a definition in Container, but insert() is declared with a definition in Linear.
- A virtual function can be redefined (directly or indirectly) in a derived class, as Queue::insert() redefines Linear::insert(). When redefinition is used, the derived-class method is often defined in terms of the inherited method, as Stack::print() is defined in terms of Linear::print().
- If a function is declared virtual, then it is virtual in all derived classes.
- When a virtual function is executed on a member of the base class, the method defined for that function in the appropriate derived class is dispatched, if it exists. Thus, when insert() is called from Linear::put() to insert a new Cell into a Queue, Queue::insert() is dispatched, not Linear::insert(). If Linear::insert() were not virtual, Linear::insert() would be executed.

12.2.1 Container: An Abstract Class

```cpp
// -----------------------------------------------------------------
// Abstract Containers
// A. Fischer June, 2023 file: contain.hpp
// -----------------------------------------------------------------
#pragma once
#include "exam.hpp"
#include "cell.hpp"

class Container {
  public: // -----------------------------------------------
    virtual ~Container() {}
    virtual void put(Item*) = 0; // Put Item into the Container.
    virtual Item* pop() = 0; // Remove next Item from Container.
    virtual Item* peek() = 0; // Look but don't remove next Item.
    virtual ostream& print(ostream&) = 0; // Print all Items in Container.
};
```
A container is a place to store and retrieve data items of any sort which have been attached to cells. Each container has its own discipline for organizing the data that is stored in it. A linear container is one that has a beginning and an end, and the data inside it is arranged in a one-dimensional manner. It might be sorted or unsorted.

In this section we develop a generic linear container class from which specialized containers such as stacks or queues can be derived. We use the program to illustrate the concepts, syntax, and interactions among a polymorphic base class and its implementation classes.

The Container class presented here supplies implementation-independent functions that are appropriate for any container and any contents type. Every container must allow a client program to put data items into the container and get them back out. A print() function is often useful and is necessary for debugging.

### 12.2.2 Linear: A Polymorphic Class

Since Linear is derived from Container, it inherits all of the function prototypes of Container. These functions are equally appropriate for use with an array or a linked list, so Linear could be implemented either way in the derived classes.

The class **Linear** defines a simple, unsorted, linear container that is implemented by a linked list, using a helper class named Cell. It defines a set of list-handling function that can be written in a generic way so that they will apply to most or all linked-list linear containers. The derived classes use the functions of **Linear** to implement various list-based data structures.

```cpp
class Linear: public Container {
protected: // -------------------------------------------------
    Cell* head; // This is a dummy header for the list.
    Cell* here; // Cursor for traversing the container.
    Cell* prior; // Trailing pointer for traversal.

public: //-------------------------------------------------
    Item* peek() { reset(); return here->getData(); }
    void put(Item* ep) { if (ep) insert( new Cell(ep) );}
    Item* pop();
    ostream& print( ostream& out );

protected: //--------------------------------------------------
    Linear(): head(new Cell()), here(nullptr), prior( head ){}
    virtual ~Linear ();
    virtual void reset() { prior = head; here = head->next;}
    bool isEnd() const { return here == nullptr; }
    void next();
    virtual void insert( Cell* cp );
    virtual Cell* remove();
    void setPrior(Cell* cp){ prior = cp; here = prior->next; }
};
inline ostream& operator<<(ostream& out, Linear& s)
{ return s.print(out); }
```

Notes on the Linear class declaration.

- **Protected or not?**
  
  - Linear is no longer abstract because all the inherited abstract functions have real definitions here.
12.2. CREATING A POLYMORPHIC CONTAINER

– The constructor and destructor are protected. That means that they can be called from Stack and Queue, but not directly by an application package. This makes sense because the only way a Linear object can be created is by creating a fully-specific derived class. The destructor of Linear is virtual, so it will be run when any derived object is deleted.

– The functions introduced in this class are protected because, if public, they would allow a caller to decide how and where to put things into the container or take things out of it. This level of control is necessary for defining Stack and Queue and is not safe for public use.

• A polymorphic class...

– Linear is polymorphic because it has virtual functions that may be redefined in a derived class. The protected functions are fully defined in this class.

– Linear overrides the four virtual functions inherited from Container: put(), pop(), peek(), and print(). These functions inherited from the interface are public because they are intended for use by the client application. Two of these are 1-liners and are defined in the header file; two are longer and are defined in the implementation file. The definitions in this class “override” the definitions in the base class because the prototypes in the two classes are exactly alike.

– Line 34: The function put() is not virtual. It simply wraps the item in a Cell, then delegates the actual insertion operation to insert(), which is virtual so that the specific and appropriate version of insert() in the derived class (Stack, Queue, etc.) will always be dispatched.

• A linked list.

– We commit to a linked list implementation when we define the Linear constructor (line 39), reset() (line 41), and isEnd() (line 42). All the functions in the .cpp file also rely on a linked list representation for the data.

– Lines 29, 30, and 41: Two data members in this class, together with the functions ++, reset(), and isEnd(), permit us to start at the beginning of the container and visit each member sequentially until we get to the end. The pointer named prior must always point to the cell preceding the pointer named here.

– Line 41: The reset() function sets the pointer named here to the beginning of the linked list. Note that we are using a 2-pointer scan to make program logic shorter and clearer. Many linked-list operations require using two consecutive cells.

– Line 42: The isEnd() function returns true if here has passed the last item in the container.

Notes on the Linear class implementation.

• Lines 57–64: Because there are virtual functions in Linear, there must be a virtual destructor. The for loop steps through the linked lists deleting first the data in each cell, then the cell itself. The 2-pointer implementation allows us to write a clean, understandable, destructor. We delete prior, but there still exists and can be used for list traversal. At the end of the list, there is just one cell left, so we delete its data then delete the cell.

• Lines 76–82, the pop() function: Whether for a stack or a queue, items will be removed from the head end; pop() does not need virtual: it is correct for both target applications. Lines 68–70 call reset() to position the pointers, then remove the first Cell from the list and return nullptr if the list was empty. Lines 71 and 73 remove the Item from the list and return it. Line 72 deletes the empty Cell because it is no longer useful.

• Lines 85–90 walk one step toward the end of the list, keeping the two pointers adjacent.

• The insert() and remove functions both rely on the other class functions to position here and prior before any call. insert inserts the new data between prior and here; prior removes it and frees memory appropriately.

• Lines 101–105 is a normal print function. It prints brackets on the two ends of the data, and between them, it loops through and prints the Items.
12.2.3 Cell: The Helper Class

Tightly coupled classes. The Container class refers to Cells but does not define Cell. From that class, all we know is that Cell is a helper class that must be used when information is placed into the Container. At that stage, a Cell could be the traditional linked-list cell or it could be just a typedef-synonym for Item* or for a
unique pointer to an Item, which would be appropriate for use with an array-based container.

In the derived Linear class, we commit to one representation for Cell as one piece of a linked list. By including item.hpp we commit to a container that holds Exams. The definition given here for Cell is the usual two-part structure containing an Exam* and a Cell*. Nothing in Cell is virtual and everything is inline.

```cpp
108 // ------------------------------------------------------------------
109 // Cell contains a data* and a link. Cells are used to build lists.
110 // ------------------------------------------------------------------
111 #pragma once
112 #include "item.hpp"
113 #include "tools.hpp"
114 // ------------------------------------------------------------------
115 class Cell {
116     friend class Linear;
117     friend ostream& operator<<( ostream& out, Cell& c);
118     
119     private: // ------------------------------------------------------
120         Item* data;
121         Cell* next;
122     
123     Cell(Item* e = nullptr, Cell* p = nullptr ): data(e), next(p){ }
124     
125     ~Cell(){ cerr <<"Deleting Cell 0x" << this << dec <<"..."; }
126     
127     void print(ostream& out) const { // --------------------------
128         if (data) {
129             out << "Cell 0x" << this;
130             out << "[" << *data << ", " << next << "]n";
131         }
132     }
133 }
134 inline ostream& operator<<(ostream& out, Cell& c){c.print(out); return out;}
```

**Friendship.** On line 116, as in previous linked list definitions, this Cell class gives friendship to Linear. A second friend declaration on line 117, is needed for the operator extension on line 134. We need to choose one of three alternatives:

a. Using the friend function declaration,

b. Making the print function public, and

c. Not having an operator extension for class Cell and writing Linear to always handle Exams, instead of leaving the base type open at that point.

In previous versions of this class, we chose strategies (c); This time we choose (a) because it makes allows the Linear class to be independent of the kind of data being stored.

### 12.2.4 Exam: The Actual Data Class

```cpp
135 //=====================================================================
136 // Bind abstract name ITEM to the name of a real class.
137 // A. Fischer August, 2022 file: item.hpp
138 //=====================================================================
139 #pragma once
140 using Item = Exam ;
```

```cpp
160 //=====================================================================
161 // Exam: A student's initials and one exam score.
162 // A. Fischer, June, 2023 file: exam.hpp
163 //=====================================================================
164 #pragma once
165 #include "tools.hpp"
166 //typedef int KeyType; // will be needed in Chapter 14
167 //=====================================================================
```
12.2.5 Class diagram.

Figure 12.2 is a UML diagram for this program, showing the polymorphic nature of Linear and its relation to Cell, Stack, Queue, and Exam.

12.3 Stack: Fully Specific

A Stack is a container that implements a LIFO discipline. In this program, we derive Stack from Linear and, in so doing, we represent a stack by a linear linked list of Cells. From Linear, Stack inherits a head pointer, two scanning pointers (here and prior) and a long list of functions.

Notes on the Stack code.

- The list head pointer is protected, not private in Linear, because some derived classes (for example Queue), need to refer to it. The scanning pointers are private because the derived classes are supposed to use Linear::setPrior() rather than setting the pointers directly. This guarantees that the two scanning pointers are always in the correct relationship to each other, so that insertions and deletions will work
properly. It also saves lines of code in the long run, since they are written once in Linear instead of multiple times in the derived classes.

- The Stack constructor and destructor are both defaulted. The constructor does no initializations. The destructor also appears to do nothing. However, it is virtual, because Stack is inherited from Linear, which has a virtual destructor. When a Stack goes out of scope, its default destructor will call the destructor in Linear, and free all the Cells on the linked list. There are no memory leaks here.

```cpp
// -----------------------------------------------------------
// Stacks, with an inheritance hierarchy
// A. Fischer June, 2023 file: stack.hpp
// -----------------------------------------------------------
#pragma once
#include "linear.hpp"
// -----------------------------------------------------------
class Stack : public Linear {
  public:
    Stack() =default;
    ~Stack() =default;
    void insert( Cell* cp ) { reset(); Linear::insert(cp); }
    ostream& print( ostream& out ){
      out << " The stack contains:
      return Linear::print( out );
    }
};
```

- Only two functions are defined explicitly in Stack: `insert()` and `print()`.
- Linear::put(Exam*) calls Insert, which is defined in both Linear and Stack. Because `insert()` is virtual, the version in Stack will be used to execute the call written in Linear::put(). We want to insert the new Cell at the head of the list because this is a stack. The actual insertion will be done by Linear::insert(), after Stack::insert() positions here and prior to the head of the list by calling reset(). The left side of figure 12.3 illustrates how control passes back and forth between the base class and the derived class during this process.

- The redefinition of `print()` is not absolutely necessary here, but it helps the output to make better sense. All it does is to print a few words ("The stack contains:" ) and call Linear::print() to print the contents.

---

**Figure 12.3:** How the classes collaborate during Stack::put (left) and Stack::print (right).
12.4 Queue: Fully Specific

A queue is a container that implements a FIFO discipline. This Queue is a linear linked list of Cells with a dummy header. In Figure 12.4, note how the dummy header provides a place to attach the head and tail pointers in an empty queue. Using a little extra memory on the header saves a lot of coding for the special case of an empty queue.

Notes on the Queue code.
This declaration of Queue follows much the same pattern as Stack. However, Queue uses the inherited Linear::print(), instead of defining a specific version of its own. This is done for demonstration purposes: we recognize that better and more informative formatting could be achieved by defining a specific local version of print().

- The output (following main) shows that the queue Q was printed by the inherited function, Linear::print().
- Linear has three data members (head, prior and here), and Queue adds a fourth (tail). The constructor of each class must initialize the data members in its own class. When a Queue is constructed, all four are initialized to create an empty linked list with a dummy header, as in Figure 12.4 (left side).
- An important part of the construction is that the constructor of the base class, Linear, is executed first. It creates a dummy cell and attaches it to the head and prior pointers. When control gets to the Queue constructor, the dummy cell will exist and it is easy to attach the tail pointer to it.
- Queue::insert(Item*) sets here and prior to the tail of the list so that insertions will be made after the last list cell. (In contrast, Stack::insert() calls reset().) Linear::insert() then finishes the job and does the actual insertion. Almost no code is duplicated because the Linear and Queue classes collaborate.
12.5. A MAIN PROGRAM AND ITS OUTPUT

When an inherited virtual function (such as push or pop) is called, control passes back and forth between the base class and the derived class, as does control for pop() shown in Figure 12.5.

If Linear::insert() is not virtual, insertion is done incorrectly for queues. I removed the “virtual” property and got the following output:

```
Putting 3 items on the Queue Q: 11, 22, 33.
[11, 22, 33]
```

Compare it to the correct output at the end of this chapter, where Ali comes first in the queue.

**Inter-class collaboration.** The collaboration between the two print() functions and the << operator is instructive. In the main program, we never call the print functions directly. All printing is done by statements like `cerr << Q;` The execution follows these steps:

- Queue does not supply a definition for `operator<<`, but Queue inherits a method from Linear. So the output task is given to the to Linear (line 49).
- This calls Linear::print (lines 36 and 101–106). Since Linear::print(), is virtual, and Q is a queue, the task of printing Q would be dispatched to Queue::print() if that method existed. But it does not exist. So Linear::print() does the job.

In contrast, the instruction `cerr << S;` goes through more steps:

- Stack does not supply a definition for `operator<<`, but Stack inherits a method from Linear. So the output task is given to the to Linear (line 49).
- Since Linear::print() is virtual and S is a stack, it dispatches Linear::print(S) to Stack::print(S).
- Stack::print(S+) prints an output heading.
- Then it calls Linear::print() explicitly to avoid making a recursive call on Stack::print().
- The class name is necessary only in the last call; all other shifts of responsibility are handled by inheritance or virtual dispatching. This activity is diagrammed on the right in Figure 12.3.

### 12.5 A Main Program and its Output

To test the linear container classes, we wrote a meaningless main program that instantiates one Stack and one Queue and moves data onto both and from one to the other. Enough function calls are made to demonstrate that the classes work properly; the contents of S and Q are printed just often enough to see the data move into and out of each container. The code is given first, followed by the output:
• The Queue implements a first-in first-out order.
• The Stack implements a last-in first-out order.
• peek() returns the same thing that pop() would return, but does not remove it from the list.
• Stack::print() is used to print the stack but the inherited Linear::print() prints the queue.
• All dynamically allocated objects are properly deleted.

226 // ---------------------------------------------------------------------------
227 // Demonstration of derived classes with virtual functions.
228 // A. Fischer June, 2023 file: main.cpp
229 // ---------------------------------------------------------------------------
230 #include "tools.hpp"
231 #include "exam.hpp" // Must precede #include for item.hpp.
232 #include "stack.hpp" // Base type is Item == Exam.
233 #include "queue.hpp" // Base type is Item == Exam.
234 // ---------------------------------------------------------------------------
235 int main( void ) {
236     Stack S;
237     Queue Q;
238     cerr << "Putting 3 items on the Stack S: 99, 88, 77.\n" ;
239     S.put( new Exam("Ned", 99) );     //cerr << S << endl;
240     S.put( new Exam("Max", 88) );     //cerr << S << endl;
241     cerr << " Peeking after second insertion: " <<*S.peek() <<"\n";
242     S.put( new Exam("Leo",77) ); cerr << S << endl;
243     cerr << "Putting 3 items on the Queue Q: 11, 22, 33.\n";
244     Q.put( new Exam("Ali",11) ); //cerr << Q << endl;
245     Q.put( new Exam("Bea",22) ); //cerr << Q << endl;
246     cerr << " Peeking after second insertion: " <<*Q.peek() <<"\n";
247     Q.put( new Exam("Cil",33) ); cerr << Q << endl;
248     cerr << "Pop two Exams from Q, put on S. \n";
249     S.put(Q.pop()); S.put(Q.pop()); cerr <<"\n" <<S << endl;
250     cerr << "Put another Exam onto Q: 44.\n";
251     Q.put( new Exam("Dan",44) ); cerr << Q << endl;
252     cerr << "Pop two Exams from S and discard.\n"
253     delete S.pop();
254     delete S.pop(); cerr <<"\n" << S << endl;
255     bye();
256 }

The output:

Putting 3 items on the Stack S: 99, 88, 77.
Peeking after second insertion: Max: 88
The stack contains:
< [ Cell 0x0x0804c988 [Leo: 77 , 0x804c968]
Cell 0x0x0804c968 [Max: 88 , 0x804c948]
Cell 0x0x0804c948 [Ned: 99 , (nil)]
]>

Putting 3 items on the Queue Q: 11, 22, 33.
Peeking after second insertion: Ali: 11
< [ Cell 0x0x0804c9a8 [Ali: 11 , 0x804c9c8]
Cell 0x0x0804c9c8 [Bea: 22 , 0x804c9e8]
Cell 0x0x0804c9e8 [Cil: 33 , (nil)]
]>

Pop two Exams from Q, put on S.
Deleting Cell 0x0x0804c9a8...
Deleting Cell 0x0x0804c9c8...
The stack contains:

```
< [
  Cell 0x0x804c9c8 [Bea: 22, 0x804c9a8]
  Cell 0x0x804c9a8 [Ali: 11, 0x804c988]
  Cell 0x0x804c988 [Leo: 77, 0x804c968]
  Cell 0x0x804c968 [Max: 88, 0x804c948]
  Cell 0x0x804c948 [Ned: 99, (nil)]
]>  
```

Put another Exam onto Q: 44.

```
< [
  Cell 0x0x804c9e8 [Cil: 33, 0x804ca08]
  Cell 0x0x804ca08 [Dan: 44, (nil)]
]>  
```

Pop two Exams from S and discard.

Deleting Cell 0x0x804c9c8... Deleting Score Bea...
Deleting Cell 0x0x804c9a8... Deleting Score Ali...

The stack contains:

```
< [
  Cell 0x0x804c988 [Leo: 77, 0x804c968]
  Cell 0x0x804c968 [Max: 88, 0x804c948]
  Cell 0x0x804c948 [Ned: 99, (nil)]
]>  
```

**Termination.** After printing the termination message, the objects Q and S will go out of scope and be deallocated. An output trace can serve as part of a proof that deallocation is done correctly and fully, without crashing. The output trace from main is given below. You can see how control and deallocation move through the destructors of the various classes.

Normal termination.

Deleting Cell 0x0x804c928... Deleting Score Cil...
Deleting Cell 0x0x804c9e8... Deleting Score Dan...
Deleting Cell 0x0x804ca08...
Deleting Cell 0x0x804c918... Deleting Score Leo...
Deleting Cell 0x0x804c988... Deleting Score Max...
Deleting Cell 0x0x804c968... Deleting Score Ned...
Deleting Cell 0x0x804c948...

### 12.6 Polymorphic Exception Classes

Chapter 7 covered the basic concepts and rules for exceptions and presented a simple exception class. Here, we extend that example by combining derivation and virtual functions for the exception handlers, the number of catch clauses needed can be minimized. This pattern is illustrated by the example below.

Related exceptions can be created by derivation. For example, the BadSuit, and BadSpot classes, below, define two additional exception types that are derived from Bad. All three might be used in a program that plays an interactive card game of cards and reads input from the keyboard. The class Bad is a base class for the others and defines the functionality common to all three classes, including a virtual print function. The UML diagram in Figure 13.1 shows the derivation hierarchy. The gray circle with a V marks a virtual function.

The purpose of the derived classes in this example is to supply more detailed and specific information than the original version of the exception class. When a data set involves more than one input, and either or both could be invalid, this is a significant courtesy to the user.

The **base exception class.** The changes from the original version are minimal.

- The destructor and the print function are now virtual.
- The last two lines of `print()` have been pulled out into a separate function named `pr()`, so that the print functions in the derived classes can also use this code.
CHAPTER 12. POLYMORPHISM AND VIRTUAL FUNCTIONS

The derived exception classes.

---

Notes on the derived exception classes.

- Two derived classes are given here: one for a bad suit, the other for a bad spot.
- Each class has a constructor, a virtual destructor, and a virtual print function the prints an error comment appropriate for one kind of error. This is better for the user than the general error comment in the prior version of the program.
- Finally, both print functions call the shared `pr()` function in the base class.
12.6. POLYMORPHIC EXCEPTION CLASSES

12.6.1 Card class, for the Demo.

The Card class declaration.

```cpp
#pragma once
#include "tools.hpp"
#include "bad.hpp"
#include "badDerived.hpp"
enum class SuitType{ Spades, Hearts, Diamonds, Clubs, Bad }

class Card {
  int spot;
  SuitType suit;
  void translate(char inspot, char insuit);
  static const string spotlabels[16];
  static const string suitlabels[5];
public:
  Card () =default;
  Card (istream& sin) {
    char inspot, insuit;
    sin >> inspot >> insuit;
    if (!sin.good()) fatal( "Low level read error\n");
    translate(inspot, insuit);
  }
  ostream& print(ostream&);
  static void instructions(int n);
};
```

The Card class implementation.

```cpp
#include "cards.hpp"
const string Card::suitlabels[5]="spades","hearts","diamonds","clubs","bad";
const string Card::spotlabels[16]="bad","Ace","2","3","4","5","6","7","8",
                                 "9","10","Jack","Queen","King"
void Card::instructions( int n) {
  cout << "Please enter " << n << " cards.\n"
  << "Spot codes are 2..9, T, J, Q, K, A \n"
  << "Suit codes are S H D C \n";  
}

void Card::translate(char inspot, char insuit) {
  if (inspot>='2' && inspot<='9') spot = inspot - '0';
  else switch( toupper(inspot) ){
    case 'T': spot = 10; break;
    case 'J': spot = 11; break;
    case 'Q': spot = 12; break;
    case 'K': spot = 13; break;
    case 'A': spot = 1; break;
  default : spot = 0;
}
```
CHAPTER 12. POLYMORPHISM AND VIRTUAL FUNCTIONS

```cpp
switch( toupper(insuit) ) {
    case 'S': suit = SuitType::Spades; break;
    case 'H': suit = SuitType::Hearts; break;
    case 'D': suit = SuitType::Diamonds; break;
    case 'C': suit = SuitType::Clubs; break;
    default : suit = SuitType::Bad;
};
if (spot==0 && suit==SuitType::Bad) throw BadCard(inspot, insuit);
if (spot==0) throw BadSpot(inspot, insuit);
if (suit==SuitType::Bad) throw BadSuit(inspot, insuit);
```

The two differences between the versions of the Card class are:

- The new version has one more `#include` because the derived exception classes are in a separate file.
- The new version has two added lines near the end of the `translate()` function to throw the two new kinds of exceptions, when appropriate.

### 12.6.2 The main function.

```cpp
int main( void ) {
    int k;
    Card hand[NCARDS];
    Card::instructions( NCARDS );
    //--------------------------------------------------- Main loop that reads all cards.
    for (k=0; k<NCARDS; ){
        //------------------------- Here is the single line of active code.
        try {
            cout << "\nEnter card # " <<k+1 << " (spot-code suit-code): " ;
            hand[k] = Card(cin); //---------------------- Input one card.
            //-- No exception - we have a good card.
            cout << " Card successfully entered into hand: ";
            hand[k].print(cout);
            ++k;
        }
        //-------------- Check for the three application-specific exceptions.
        catch (BadCard& bs) { bs.print(); } // Will catch all 3 Bad errors.
        //-------------- Now check for general exceptions thrown by system.
        catch (...) {
            fatal( "\nLast-ditch effort to catch exceptions.\n" );
        }
        //--------------- Control comes here after the try/catch is finished.
    }
    cout << "\nHand is complete: " << endl;
    for (k=0; k<NCARDS; ++k) { hand[k].print( cout ); }
• When an exception is caught by line 155, we call `bs.print()` to process it.
• But `BadCard::print()` is virtual and has three defining methods within the BadCard class hierarchy. So when `bs.print()` is called, the system will inspect the run-time type-tag attached to `bs`, the exception object, to find out which class or subclass constructed this particular exception.
• Then `BadCard::print()` will be executed if the object thrown was a `BadCard`. Otherwise, `BadSpot::print()` or `BadSuit::print()` will be called. This is how we get three different kinds of output from one catch clause. (Note the first, second, and third blocks of error comments in the output transcript, below.)
• If `BadCard::print()` were not virtual, the base-class `print()` function would always be used.
• A function in a derived class often calls a function in its super-class. The `pr()` function illustrates this: it prints a message that is appropriate for all three exceptions, and the print functions in all three classes call it.

Output. The output that follows shows two sample runs of this program with different exception handlers active. The first output is from the program as shown:

```
Please enter 3 cards.
Spot codes are 2..9, T, J, Q, K, A
Suit codes are S H D C

Enter card # 1 (spot-code suit-code): 3d
   Card successfully entered into hand: 3 of diamonds

Enter card # 2 (spot-code suit-code): 2s
   Card successfully entered into hand: 2 of spades

Enter card # 3 (spot-code suit-code): kC
   Card successfully entered into hand: King of clubs

Hand is complete:
3 of diamonds
2 of spades
King of clubs
//===========================================

Please enter 3 cards.
Spot codes are 2..9, T, J, Q, K, A
Suit codes are S H D C

Enter card # 1 (spot-code suit-code): kc
   Card successfully entered into hand: King of clubs

Enter card # 2 (spot-code suit-code): qx
   Legal suits are H D C S
       You entered q of x. Please reenter.

Enter card # 2 (spot-code suit-code): wd
   Legal spot values are 2..9, T, J, Q, K, A
       You entered w of d. Please reenter.

Enter card # 2 (spot-code suit-code): df
   Both spot value and suit are wrong

Enter card # 2 (spot-code suit-code): ah
   Card successfully entered into hand: Ace of hearts

Enter card # 3 (spot-code suit-code): qh
   Card successfully entered into hand: Queen of hearts

Hand is complete:
King of clubs
Ace of hearts
Queen of hearts
```
12.7 Dynamic Casts

Having covered virtual functions and polymorphism, it is time to return to the fourth kind of cast that is supported by C++. Dynamic casts are used with polymorphic classes, and can cast either pointers or references. Dynamic casts can move either upward or downward on the derivation tree. A cast from BaseClass to DerivedClass is a downward cast, a cast from DerivedClass toward the BaseClass is an upward cast.

In this chapter, Container is the base class and Linear is derived from it. This is a polymorphic hierarchy because Container is abstract and all derived classes inherit virtual functions. Suppose we had another derived class, Circular.

![Derivation Hierarchy Diagram]

Suppose sp is a pointer to stk, an object of class Stack.

```cpp
Stack stk;
Stack* sp = &stk;
Linear* ip1 = (Linear*) sp;  // Explicit upward cast.
Linear* lp2 = sp;  // Implicit upward cast is OK
```

Then we can use a dynamic cast to view stk as a Linear or sp as a Linear*. This is an upward dynamic cast. Upward casts are always meaningful because any derived-class object includes a base-class object as part of itself.

Now let us add a Queue object and a pointer to it:

```cpp
Queue que;
Queue* qp = &que;
Linear* ip1 = qp;  // Implicit upward cast is OK
sp = (stack*)qp;  // Illegal to cast sideways.
sp = (stack*)lp1;  // Illegal to cast downwards to the wrong type.
qp = (queue*)lp1;  // Legal to cast downwards to the right type.
```

A downward cast is meaningful if the object pointed at by lp (a base-class pointer) is actually an object of the type mentioned in the cast. That is, if we have Linear* lp we can cast it to a Queue* if and only if it actually points to a Queue not a Stack. (Last line of code, above.)

However, a down-cast of the same pointer (second-to-last line) to a Stack* is not legal because it would not be meaningful. Such a cast could cause a run-time crash. To prevent such problems, the legality of every down-cast is checked at run time. A bad down-cast of a pointer will return a null result, which MUST be checked. A bad down-cast of a reference will throw a `bad_cast` exception. The following demo program illustrates all of these issues. The lines that are commented out were erroneous in some way.

**Dynamic cast demo.** The output is shown below the code.

```cpp
#include <iostream>
using namespace std;
// -------------------------------------------------- file: dyncast.cpp
class A { public: double x=20.2; virtual int funA(){ return 35; } };
class B: public A { public: int b2=98; };
class D: public A { public: int d2=67; };

int main( void ) {
  A* ap;
  B bb;  // Base class is A.
```
B* bp = new B;
D dd;  // Base class is A.
//------------------------------------------------------ up casts
ap = &dd;  // No cast needed for upcasts with public derivation.
ap = dynamic_cast<A*>(&dd); // But Low->high dynamic cast is OK.
ap = (A*)(&dd);  // This means the same thing.
cout << "*ap= " <<ap->x <<endl; // Every D object inherits A’s parts.
cout <<" dd= " <<dd.d2 <<endl; // d2 exists and is initialized.
  // cout <<"*ap= " <<ap->d2 <<endl; // but it is not accessible from A’s level.

//---------------------------------------------------- down casts
cout <<" bp->b2=" << bp->b2 <<endl; // bp and b2 are initialized.
bp = dynamic_cast<B*>(&dd); // Can’t cast a D to a B. bp will= nullptr
  // cout << " bp->b2=" << bp->b2 <<endl; // Can’t dereference a nullptr
  cout <<" bb.funA()=" <<bb.funA() <<endl; // bp can access A’s functions.
cout <<" bp->b2=" <<bp->b2 <<endl; // but low-level members can’t be seen.
ap = &bb;  // Initialize ap using default up-cast.
bp = dynamic_cast<B*>(ap);; // Should be OK
  cout <<" bp->b2=" << bp->b2 <<endl; // Yes!
}  

/* Output -------------------------------------
*ap= 20.2
dd= 67
bp->b2=98
-----------------------------
bb.funA()=35
-----------------------------
bp->b2=98
*/

Notes on the dynamic casts.

• Three classes are declared on lines 4–6. A is the base class, the other two are derived from A. These classes have only 1 or 2 members each, barely enough to illustrate the principles. In all cases, we rely on default constructors, default initializations, and default destructors.

• Lines 9–12 declare objects, of which two are pointers.

• Lines 14–16 show three legal statements. Line 14 uses the default up-cast. The other lines use two different syntaxes to do the same thing.

• Line 19 is commented out because you cannot access the parts of the derived class using a base-class pointer.

• Line 22 proves the pointers and objects are working normally.

• Line 23 is a cross-cast: neither up nor down. Cross casts are always meaningless, even though they are legal. The g++ compiler gives a warning about this. The result of this cast is nullptr.

• Line 25 shows that a B object inherits all of A’s functions, and can call them.

• Line 30 makes ap point at a B object. No problem.

• Line 31 then uses a down-cast to change the value back to a B pointer. No problem; it works and you can print it.

When should I use a dynamic-cast? You never need to use an explicit dynamic cast to move up a derivation tree. The dynamic cast is done automatically whenever you use a derived-class object in a base-class context, or set a base-class pointer to point at a derived-class object. For this reason, it is hard to find a good use for an explicit upward cast in a simple program.
The need for down-casts is also rare. Most of the purposes for which you might wish to use a downcast are served best by using virtual functions instead!

The rules for using dynamic casts are complicated by private parts, private derivation, and multiple inheritance. Some examples are given in the next chapter to illustrate these issues and show what kind of dynamic casts are and are not permitted in a multiple-inheritance situation.
Chapter 13: Abstract Classes and Multiple Inheritance


Train up a child in the way he should go, and when he is old he will not depart from it.

An abstract class declares a set of behaviors (function prototypes) that all of its descendants must follow (or implement). Documentation accompanying the abstract class must explain the purpose of each function and how each interacts with other parts of the class. Derived classes must implement all of the functions, and should obey the guidelines explained in the documentation.

13.1 An Abstract Class Defines Expectations

Abstract classes. When a base class includes even one prototype for a pure virtual function, it is an abstract class which cannot be used to create objects. However, such classes do have a purpose and they are important in the process of developing a large system. An abstract class lets us define and enforce a common interface, or behavior, for a set of related classes. Class derivation, combined with methods defined in the derived class(es) make the abstraction useful.

Large systems are developed by teams of programmers in this way. The first development step is to identify what the major modules should be and what interface each module should provide. Then the interfaces for these modules are specified as abstract classes.

The abstract class forms a contract for the programming team that will develop the component, and for different people (or teams) who will work on the subsystems simultaneously. When the code for a subsystem is compiled, it will include the abstract class for its foundational module. The compiler will ensure that no necessary part is forgotten, and that all functions in the subsystem conform to the prototypes that were promised.

An abstract class specifies a set of virtual representation-dependent function prototypes for which definitions will be required in future derived classes. By doing so, it enables a large system to be developed in a top-down style.

In the previous chapter, we defined an abstract class named Container, a foundation module named Linear, derived from Container, and two classes that implement different linear data structures: Stack and Queue.

Since all the prototypes provided by Container are defined, a developer can work on the derived class, Linear. Then programmers can simultaneously begin to write Linear, Stack, and Queue, relying on the prototypes already defined. When the modules are complete, the subsystems can be easily interchanged because the established prototypes guarantee that all modules conform to a uniform framework.

In this chapter we go one step further: we add two new linear data structures and a second abstract class that defines requirements for the types that can be stored in these data structures.

Definition and rules.

- The opposite of abstract is concrete. A concrete class can have virtual functions, but all of those functions must have methods defined within the class itself or its ancestor classes.
- Any class that has one or more pure virtual functions is called an abstract class. An abstract class cannot be instantiated, that is, used to construct any objects.

13.2 Abstraction Example: an Ordered Type

Two abstract classes are used in this chapter’s program: Container (from the previous chapter) and Ordered.
• **Container** defines the interface that should be presented by any container class: a way to put data into the container (put()), and find it when needed (get()), take it out of the container (remove()), write the contents of the container to a stream (print()).

• **Ordered** defines prototypes for functions that are needed when you sort data items: comparison functions, sentinels, and a way to access the key field of the data.

**Specifying and enforcing requirements.** The purpose of a container is to store a collection of items. The data stored in an item is not important; we use the class name `Item` as a representative of any kind of object that a containers might store. However, a few properties of an Item are essential for use with a sorted container:

• The Item must contain a key field and a key() function that returns the key.

• The operators `<` and `==` must be defined to compare two Items. Items will be compared using one or both of these operators. They will be stored in the container in ascending order, as defined by the operator `<`.

• A programmer who creates an Ordered class must supply the appropriate typedef for `KeyType` and appropriate definitions for the operators and sentinels that define the minimum and maximum possible values for a `KeyType` object.

The first two properties can be specified by defining an abstract class, Ordered, that gives prototypes (but no definitions) for the three required functions. We can enforce these requirements in a data class by deriving the data class from Ordered. When we do this, we instruct the compiler to guarantee that the derived data class does implement every function listed by Ordered. If one of the functions is missing, the compiler will give an error comment.

13.3 Multiple Inheritance

A class may be derived from more than one parent class. (We must include the header files for each parent class.) The purpose of such *multiple inheritance* is:

• Simple form: to inherit properties from one parent and constraints from another.

• General form: inherit properties from two parent classes

The syntax is a simple extension of ordinary derivation. We use it here (line 30) to combine the properties of the Exam class from the previous chapter with the abstract Ordered class. The new class is a wrapper for Exam that provides more functions than the original class but does not duplicate the ones that Exam supplies (Print() and operator<<).

### 13.3.1 Item: The Data Class

In the previous chapter, we used a typedef to make `Item` a synonym for the `Exam` class. In this chapter, we do more with `Item`: we use multiple inheritance to add constraints and functionality to the original `Exam` class. The first parent of `Item` is `Exam`, the second parent is `Ordered`. The `Item` class defines all the abstract functions from ordered. Then the combination of the functionality in `Exam` and the functionality from `Ordered` supplies everything necessary to make ordered lists of exams in `List` and `PQueue`.

```cpp
// Bind abstract name ITEM to a real class.
// A. Fischer August, 2022 file: item.hpp
#pragma once
#include <limits>
#include "exam.hpp"
#include "ordered.hpp"
```

15 //===========================================================================
16 // Bind abstract name ITEM to a real class.
17 // A. Fischer August, 2022 file: item.hpp
18 //===========================================================================
19 #pragma once
20 #include <limits>
21 #include "exam.hpp"
22 #include "ordered.hpp"
23 //===========================================================================
24 //----------------------------------------------------------------------------
13.4 TWO MORE LINEAR CONTAINER

Notes on the Item class.
Two comparison operators and the key() function are required for Item because it was derived from Ordered. We define these three functions (lines 33–35) in such a way that score (inherited from Exam) is the key field and the exams will be sorted in ascending order by Score. A third comparison function (line 36) is added for the convenience of client classes.

The Item class also defines two constants that are often needed for sorting algorithms: the maximum and minimum values of type KeyType. Line 24 tells us that KeyType is a synonym for int; the int values used here are supplied by the header file <limits> in the standard library.

The Item constructor does nothing but pass its arguments through to the Exam constructor because it has no variables of its own that need initialization. The virtual destructor in the Ordered class is necessary to avoid warning comments in the Item class. For example, without that seemingly useless function, we get a warning comment:

```
item.hpp:32: warning:
'class Item' has virtual functions but non-virtual destructor
```

With these definitions, Item fulfills all the inherited obligations, the class is concrete and can be used normally. This class will be used with Container, Linear, and Cell from the prior chapter to build two new container classes: List and Priority Queue.

13.4 Two More Linear Container

In this chapter we develop two new container classes from Linear. In a stack or a queue, all insertions and deletions are at one of the ends of the container; we never need to locate a spot in the middle. In contrast, a
priority queue requires all insertions to be made in sorted order, and a simple sorted list requires both insertions in the middle and a search whenever an item is removed. To develop these classes in a general way, we assume that each Cell will contain an Item that is derived from Ordered, and we use the functions promised by Ordered to define necessary new functions in the Item class:

```
KeyType getKey() const { return score; }
bool operator==(const KeyType& k) const { return getKey() == k; }
bool operator<( const KeyType& k ) const { return getKey() < k; }
bool operator<( const Item& s ) const { return getKey() < s.getKey(); }
```

These functions allow us to search or sort a linear container according to the key field of the Item.

### 13.4.1 PQueue: a Sorted Linear Container

In a priority queue, each new item must be inserted so that it is greater than the prior item and less than the following item. The first item is always the highest priority and is the only one ever removed.

```cpp
// ------------------------------------------------------------------------
// Priority queues: derived from Container<--Linear<--PQueue
// A. Fischer July, 2023 file: pqueue.hpp
// ------------------------------------------------------------------------
#pragma once
#include "linear.hpp"

class PQueue : public Linear {
public: // -----------------------------------------------------
    PQueue() =default;
    ~PQueue()=default;
    // ------------------------ Insert new Cell in ascending sorted order.
    void insert( Cell* cp ) {
        for (reset(); !isEnd(); next() ) { // locate insertion spot.
            if ( !(here->getKey() < cp->getKey()) ) break;
        }
        Linear::insert( cp ); // insert at end or at here.
    }
};
```

**Notes on the PQueue code**

- Confusingly, lower numbers usually indicate higher priority. You will see this in the output.
- Items are deleted from a priority queue at the head of the list, just like an ordinary queue. Preparation for a deletion is easy because Linear provides the reset() function to position its pointers at the head of the list.
- Items are inserted in priority order in the list, which means that the insertion spot must be found. In a linked list, that means we must scan the list to locate the correct spot.
- The loop on lines 51–53 performs such a scan using getKey(), the < operator and the list traversal functions (reset(), isEnd(), and next() ) that are provided by Linear and inherited by PQueue. When the right place is found, control is returned to Linear to do the actual insertion.
- The only other functions here are a default constructor and a default destructor. Neither is necessary because the compiler will supply them by default. (Compare this class to List, below, in which the constructor and destructor have been omitted.) However, it is good style to write explicit definitions.

**Testing the PQueue**

This main program's only purpose is to test the pqueue class. It puts meaningless data into the container, takes some back out, and prints the results. The test results shown on the next page prove that insertions into the PQueue are implemented in the correct order, and that removal from the middle of the list works correctly. Other test runs verified that removal from head and tail of list also work properly. Trace comments after termination are omitted here, but show that all remaining Cells and Items were deleted by the destructors.
The output:

Multiple~> multi
Putting 4 items onto PQueue P: 22, 11, 44.

<[
Cell 0x0x6000000480060 [Ali: 11 , 0x6000000480080]
Cell 0x0x6000000480080 [Edy: 20 , 0x6000000480050]
Cell 0x0x6000000480050 [Bea: 22 , 0x6000000480070]
Cell 0x0x6000000480070 [Dan: 44 , 0x0]
]

Dequeuing one item from P and discard it:
Deleting Cell 0x0x6000000480060...0x6000000084270 Peek at P: Edy: 20
13.4.2 List: An Unordered List

The List class provides a container with no special rules for insertion or internal order. Unlike the priority queue, Items are not stored in sorted order. Insertions can be in any order and are always made at one end of the data structure. However, removal of a list item requires a search.

```cpp
// Unsorted list: derived from Container|--Linear|--List
// A. Fischer July, 2023 file: list.hpp

#pragma once
#include "linear.hpp"
#include "item.hpp"

class List : public Linear {
public:
    void insert( Cell* cp ) { reset(); Linear::insert(cp); }
    void remove( KeyType k ) { find(k); Linear::remove(); }
    void find( KeyType k ) { // Sequential search of linked list.
        for (reset(); !isEnd(); next()) if ( here->getKey() == k) break;
        if (isEnd()) cerr << " Item with key " << k << " not found in list.\n";
    }
};
```

13.4.3 Testing the Unordered List

This main program’s only purpose is to test the list class. It puts meaningless data into the list, takes some back out, and prints the results. The test results show that insertions into the List are implemented improperly, and that removal from the middle of the list works correctly. Other test runs verified that removal from head and tail of list, and attempted removal of a key that did not exist, also work properly.

```cpp
int main( void ) {
    List L;
    cerr << " Print the empty List L.\n" << L;
    cerr << "\n Putting 4 items onto List L: 29, 37, 18, 22.\n";
    L.put( new Item("Ned", 29) ); //cerr << L << endl;
    L.put( new Item("Leo", 37) ); //cerr << L << endl;
```
13.4. TWO MORE LINEAR CONTAINER

The output:
```
Multiple-> multi
  Print the empty List L.
<[
 ]>

Putting 4 items onto List L: 29, 37, 18, 22.
<[
Cell 0x0x600000b44080 [Bea: 22 , 0x600000b44070]
Cell 0x0x600000b44070 [Edy: 18 , 0x600000b44060]
Cell 0x0x600000b44060 [Leo: 37 , 0x600000b44050]
Cell 0x0x600000b44050 [Ned: 29 , 0x0]
]>}

Remove 35 from L: Item with key 35 not found in list.
Remove 37 from L: <[
Cell 0x0x600000b44080 [Bea: 22 , 0x600000b44070]
Cell 0x0x600000b44070 [Edy: 18 , 0x600000b44060]
Cell 0x0x600000b44060 [Leo: 37 , 0x600000b44050]
Cell 0x0x600000b44050 [Ned: 29 , 0x0]
]>}

Remove last item on list: <[
Cell 0x0x600000b44080 [Bea: 22 , 0x600000b44070]
Cell 0x0x600000b44070 [Edy: 18 , 0x0]
]>}

Push a new item onto L. The list now contains:
<[
Cell 0x0x600000b44090 [Una: 46 , 0x600000b44080]
Cell 0x0x600000b44080 [Bea: 22 , 0x600000b44070]
Cell 0x0x600000b44070 [Edy: 18 , 0x0]
]>}

Notes on the List code

• Since the order of items in the list does not matter, we use the easiest possible insertion method: insertion
  at the head, as in Stack.
• However, removing an item creates two new problems: how can we specify which item to remove, and
  how can we find it?
• The removal function required by Container does not have a parameter, but to remove an item from a List,
  we must know the key of the desired item. We solve the problem here by adding a method for remove(
KEYTYPE) to the List class (Line 70).

- Since the list is unsorted, we must search the entire list before we know whether or not the key is in the list. So remove() calls a search function, find(KeyType) to locate the item. If the key value is not found, user information is displayed.
- In either case, Linear::remove() is called to remove the item from the data structure. This does nothing if the key value was not found because the list pointers are already at the end of the list.

### 13.5 Derivation from Two Classes with Data Members

If a class is polymorphic or is derived from a polymorphic class, the true type of every class object will be stored as part of the object at run time. I call this a “type tag”. Whenever a virtual function is called, this type tag is used to select the most appropriate method for the function. If class is not part of a polymorphic family, no run-time type tag is attached to its objects, and no run-time function dispatching happens.

As long as derivation is used singly—so that a derived class has only one parent with data members—inheritance works smoothly, the storage model is easy to implement, and it is not hard to understand how it works. However, when a class inherits data members from two parents, two serious problems can occur:

- The this pointer points at the beginning of the entire object, consisting of the parts of the first parent, followed by the parts of the second, and finally, the parts of the derived class. To apply a function inherited from the second class, the compiler must compute where this should point for that part of the object. Dynamic casts are related to this problem.

- Three generations. Suppose class A exists, and classes B and C are derived from it. Then suppose class D is derived from both B and C. If A (the grandparent) has data members, then B and C inherit them. But what about D? D inherits all of B’s parts and all of C’s parts. So does D have two copies of each data member from A? Virtual inheritance exists to prevent this mess: if the derivation is virtual, only one copy of the grandparent’s data will be inherited. (Note: When used in this way, the word virtual has nothing to do with virtual functions.)

![Class Diagram](image_url)

**Figure 13.3: Double (donut) inheritance.**

**Naming and visibility rules.** These rules govern the meaning of a name in a donut situation:

- A class can inherit two actually different members with the same name from different parent classes. When this happens, the name is ambiguous and you must use the :: to denote which one you want. For example, within class D, you would write B::a to refer to the member named ‘a’ inherited from B or C::a for the member inherited from C.
13.5. DERIVATION FROM TWO CLASSES WITH DATA MEMBERS

- If there are more than three levels in an inheritance hierarchy, the virtual property must be redeclared at each level that has multiple inheritance.
- A is a parent class of B, and both define a member named x. In classes A and C, only A::x is visible, so writing x in these contexts will always mean A::x. For functions in classes B and D, both A::x and B::x are visible, but B::x dominates A::x because it is “closer” to these classes. The dominant member will be used when x is written without the double colon. To refer to the non-dominant member, the full name A::x must be used.

13.5.1 Virtual Inheritance

```cpp
// ------------------------------------------------- File: "donut.hpp"
#pragma once
#include "tools.hpp"

class A { //----------------------------------------- Grandparent Class
    public:
        double x = 11.1;
        A() =default;
        virtual ~A(){};
        virtual void dump(){ cerr <<" A::x = " << x; }
};

class B: virtual public A { //----------------------- First Parent of D
    public:
        int a =20;
        double x =22.2;
        B() =default;
        virtual ~B(){};
        virtual void dump(){
            A::dump();
            cerr <<" B::x = " << x <<" " <<"B::a = " << a;
        }
};

class C: virtual private A { //--------------------- Second Parent of D
    public:
        int a =30;
        C() =default;
        virtual ~C(){};
        virtual void dump(){ A::dump(); cerr <<" C::a = " << a <<"\n"; }
};

class D: public B, public C { //---------------------- Grandchild Class
    float f =44.4;
    public:
        D() =default;
        void dump(){
            cerr <<"The A part of dd:\n"; A::dump(); cerr <<endl;
            cerr <<"The B part of dd:\n"; B::dump(); cerr <<endl;
            cerr <<"The C part of dd:\n"; C::dump();
            cerr <<"dd's own parts:\n" <<" D::f = " << f <<"\n\n";
        }
};
```

The keyword virtual can be used in a derivation declaration to prevent inheriting the same member from two parents. It is only relevant when multiple inheritance will be used, and both parents are derived from the same class. In these cases, there is more than one path through the UML diagram from the earliest ancestor class that has data members to some derived class. When virtual derivation is used, an object of a derived class will have exactly one copy of the members of each ancestor. If derivation is not virtual, each derivation path will produce its own copy of any common ancestor, resulting in two data members with the same full name in the derived-class object. This is dysfunctional!

The simplest situation that illustrates the rules for virtual derivation is a class hierarchy in the shape of a “donut”. The set of four classes defined above creates the “donut” shown in Figure 13.3. Each class has either
CHAPTER 13. ABSTRACT CLASSES AND MULTIPLE INHERITANCE

one or two data members, a constructor, and a dump function. Data is made public to make it easier to show what is going on.

The data diagrams on the right of the diagram show how storage would be allocated for objects of each of the four classes. The data member of A is inherited by classes B and C and becomes the first member of objects bb and cc. class D inherits all of the data members of B and all of the data members of C. Nonetheless, dd has only one sub-object of class A, because virtual derivation was used to derive B and C from A. The brief donut program illustrates how naming, visibility, and dynamic casts work.

Dynamic Casts on the Donut

```cpp
#include "donut.hpp"

int main( void ) {
    A aa;  // Data member: x
    C cc;  // Data members: A::x, C::a
    D dd;  // Data members: A::x, B::a, B::x, C::a, D::f

    cerr <<"Dumping aa\n"; aa.dump();
    cerr <<"Dumping cc\n"; cc.dump();
    cerr <<"Cast dd to a B* and dump\n"; dynamic_cast<B*>(&dd)->dump();

    // Upward casts. -----------------------------------------------
    B* bp = &dd;  // Explicit Low->high reference cast not needed.
    B& br = dd;   // Explicit Low->high reference cast not needed.
    cerr <<"Dumping br\n"; br.dump();  // Use the reference variable to dump.

    A* ap = dynamic_cast<A*>(bp);  // Upward cast IS OK if derivation is public.
    cerr <<"Dumping ap\n"; ap->dump(); // Show result of dynamic cast.

    // Downward casts. ---------------------------------------------
    cerr <<"Dumping after down casts, D->B->A->D .\n";
    D* dp = dynamic_cast<D*>(bp);  // Can dynamic_cast downward where type is true.
    dp->dump();                    // Start with D and return to a D.

    cerr <<"Dumping after down casts, D->B->A->C .\n";
    C* cp = dynamic_cast<C*>(ap);  // Can dynamic_cast downward where type is true.
    cp->dump();                    // A D object has all the parts of a C object.

    cerr <<"Dumping after down casts, A->C .\n";
    ap = &aa;

    //cp = dynamic_cast<C*>(&aa);  // Cannot cast A object to larger C type.
    cp = dynamic_cast<C*>(ap);    // cp is a more complex type than &aa.
    cerr <<"Result of cast is: " <<cp <<" "; // Bad cast result is nullptr.
    cp->dump();                    // You can’t dereference nullptr.
}
```

Notes on the donut diagram example:

- We define 4 classes that form a donut diagram. A is the base class. B and C are derived from A by virtual public derivation. D is similarly derived from both B and C.
- Two classes (A and B) have a data member named x. You can see from the output that B has BOTH copies. To access A’s x from B, you must use the full name, A::x But B’s x can be accessed without the double colon.
- Two of the classes (B, C) have a data member named a. Thus, dd has two different data members named a. Both exist and can be used by D, but only by using the full names B::a and C::a
- Line 67 is a legal down-cast.
- Lines 76 and 77 are not legal down-casts.; 76 is commented-out, line 77 causes a segfault and ends execution.
• In this example, both B and C are derived virtually from A. If we omit the “virtual” from either one of the lines 163 and 174, or from both, we get this error comment.

donut.cpp:16:13: error: ambiguous conversion from derived class 'D' to base class 'A':
   class D -> class B -> class A
   class D -> class C -> class A
   A* ap = &d;           // default up-cast

The output:

Dumping aa
   A::x = 11.1
   A::x = 11.1
   A::x = 11.1
   C::a = 30

Cast dd to a B* and dump
   The A part of dd:
      A::x = 11.1
   The B part of dd:
      A::x = 11.1
      B::x = 22.2
      B::a = 20
   The C part of dd:
      A::x = 11.1
      C::a = 30
   dd's own parts:
      D::f = 44.4

Dumping br
   The A part of dd:
      A::x = 11.1
   The B part of dd:
      A::x = 11.1
      B::x = 22.2
      B::a = 20
   The C part of dd:
      A::x = 11.1
      C::a = 30
   dd's own parts:
      D::f = 44.4

Dumping ap
   The A part of dd:
      A::x = 11.1
   The B part of dd:
      A::x = 11.1
      B::x = 22.2
      B::a = 20
   The C part of dd:
      A::x = 11.1
      C::a = 30
   dd's own parts:
      D::f = 44.4

Dumping dp after up and down casts, D->B->A->D .
   The A part of dd:
      A::x = 11.1
   The B part of dd:
      A::x = 11.1
      B::x = 22.2
      B::a = 20
   The C part of dd:
      A::x = 11.1
      C::a = 30
   dd's own parts:
      D::f = 44.4

Dumping cp after up and down casts, D->B->A->C .
   The A part of dd:
      A::x = 11.1
   The B part of dd:
      A::x = 11.1
      B::x = 22.2
      B::a = 20
   The C part of dd:
      A::x = 11.1
      C::a = 30
   dd's own parts:
      D::f = 44.4
Dumping after down casts, A->C.
Result of cast is: 0x0  Segmentation fault
Chapter 14: Templates

It is hard to overestimate the importance of the standard Template Library: the STL can simplify life for any C++ programmer who knows how to use it well. Ay, there’s the rub.

Two kinds of templates are part of the standard library: template functions and template data structures. We discuss the data structures first.

Templates are the archetypes behind the “container” data structures whose processing methods depend only on the structuring method and not on the type of the data contained in the structure. The C++ standard library supports a set of pre-defined templates that implement stacks, queues, trees, hash tables, and more. A programmer might use the template library structures as they are, extend them, adapt them, or define entirely new template classes.

A template consists of declarations and functions with one or more type parameters or non-type parameters. Type parameters are used to represent the base type of a data structure. Non-type are generally integers and are used to define such things as array lengths. Both kinds of parameters can be given default values.

By itself, a template is not compilable code. Before it can be compiled, actual types must be supplied to replace the type parameters. The replacement process is called instantiation and happens near the beginning of compile time. The result of instantiation is a class declaration plus compilable definitions of the class functions.

14.1 The Standard Template Library

The Standard Template Library was designed with extreme care to be as complete and portable and as safe as possible within the context of standard C++. Among the design goals were:

• To provide standardized and efficient template implementations of common data structures, of algorithms that operate on these structures, and of algorithms that are data-independent.

• To produce efficient code. Instantiation of the generic container class is done at compile-time, producing code that is both correct and efficient at run time. This contrasts with derivation and polymorphism which can be used to achieve the same ends. Polymorphism involves code executed at run-time to determine which method to use; it is much less efficient that template expansion to do the same task.

• To unify array and linked list concepts, terminology, and interface syntax. Code can be written and partially debugged before making a commitment to one kind of implementation or to another. This permits code to be designed and built in a truly top-down manner.

There are three major kinds of components in the STL:

• Containers manage a set of storage objects (list, tree, hashtable, etc). Twelve basic kinds are defined, and each kind has a corresponding allocator that manages storage for it.

• Iterators are pointer-like objects that provide a way to traverse through a container.

• Algorithms are computational procedures (sort, find, set_union, etc.) that use iterators to act on containers. The STL algorithms are used in a broad range of applications.

In addition to these components, STL has several kinds of objects that support containers. These include pairs (key–value pairs, for the associative containers), allocators (to support dynamic allocation and deallocation) and function-objects (to “wrap” a function in an object). Function objects can be used in algorithms instead of function pointers.
14.2 Containers

The definition of each container class consists of template code, of course, but that code is not part of the standard. Instead, the standard gives a complete definition of the functional properties and time/space requirements that characterize the container. Two groups of containers are supported: sequence containers (lists, vectors, queues, etc.) and sorted associative containers (maps, sets, etc). The intention is that a programmer will select a class based on the functions it supports and its performance characteristics. Although natural implementations of each container are suggested, the actual implementations are not standardized: anything that is operationally equivalent to the model code is permitted.

Big-O notation is used to describe performance characteristics. In the following descriptions, an algorithm that is defined as time $O(n)$, is never worse than $O(n)$ but may often be better.

**Member operations.** Some member functions are defined for all containers. These include:

- **Constructors:** A null constructor, a constructor with one parameter of the container type, and a copy constructor. The latter two constructors operate in linear time.
- **Destructor:** It will be applied to every element of the container and all memory will be returned. Takes linear time.
- **Traversal initialization:** `begin()`, `end()`, `rbegin()`, `rend()`. These mark beginning and ending points for a traversal or reverse-traversal.
- **The object’s state:** `size()` – current fill level, `max_size()` – allocation size, `empty()` – true or false.
- **Assignment:** `=` Assign one container to another. Linear time.
- **Equality:** `a == b`, `a != b` – Returns true or false. Two containers are equal when the sequences of elements in both are element-wise equal (using the definition of operator== on the element type. Otherwise they are not equal. Both take linear time.
- **Order:** `<`, `<=, >, >=` Lexicographic comparisons; linear time.
- **Misc:** `a.swap(b)` – swaps two containers of the same type. Constant time.

**Sequence Containers**

These are all sequence containers: `array`, `vector`, `deque`, `forward_list`, `list`. There are also container adaptors, implemented on top of these basic containers: `stack`, `queue`, and `priority_queue`. These classes all support the following requirements:

- **Constructor with two parameters, int n and base-type element t:** Construct a sequence with n copies of t.
- **Constructor with two forward-iterator parameters, j and k:** Construct a sequence equal to the contents of the range `[j, k)`.
- **Traversal initialization:** `begin()`, `end()`, `rbegin()`, `rend()`. These mark beginning and ending points for a traversal or reverse-traversal or for many algorithms.
- **The object’s state:** `size()` – current fill level, `max_size()` – allocation size, `empty()` – true or false.
- **Assignment:** `=` Assign one container to another. Linear time.
- **Equality:** `a == b`, `a != b` – Returns true or false. Two containers are equal when the sequences of elements in a and b are element-wise equal (using the definition of operator== on the element type. Otherwise they are not equal. Both take linear time.
- **Order:** `<`, `<=, >, >=` Lexicographic comparisons; linear time.
- **Misc:** `a.swap(b)` – swaps two containers of the same type. Constant time.
14.3 Iterators

An iterator provides a general way to access the objects in a container, unifying and replacing both subscripts and pointers. It allows the traversal of all elements in a container in a uniform syntax that does not depend on the implementation of the container or on its content-type.

Iterators are used generally for all the containers and for ordinary arrays. Each kind of container, has an associated iterator-type that is aware of the structure of the container and how to traverse it in order. For example, this declares an iterator for a vector with base type $T$: `vector<T>::iterator it;`

There are iterators, used with all STL containers, that mark the logical beginning and the end of a data structure:

- `begin`: Points at the logically first element in a container.
- `end`: Points one slot past the last data stored in a container and is normally used to terminate loops.

14.3.1 The Identity of a Type

Class `type_info` The class `type_info` stores information about a type. An instance is automatically created each time a type is defined. Members include:

- A printable name: `cout << typeid(a).name;`
- A hashcode (type `size_t`) that identifies the type:
  `cout << typeid(a).hash_code;`
  This code is implementation dependent and may vary between executions.
- For primitive types, the name is part of the language definition.
- For template library types, it is a long string of letters and symbols drawn from the type definition.
- For user-defined types, the ID uses the class or enum name and the next unused integer code for classes or for enums.

Functions defined by `type_info` A unique identifier for a type.

- `operator ==:`
  Compare the types of two expressions for equality
- `operator !=:`
  Compare the types of two expressions for inequality
- `name()`: The printable name of the type.
- `hash_code()`: An integer that uniquely identifies the type during the current execution.
- `before()`: used by the compiler, an implementation-dependent order of definitions.
**Why is this needed?** Its purpose is to permit the runtime system:

- To compare the types of two objects about which it knows little or nothing.
- To declare and allocate a temporary to hold an object of any type. This is essential for creating temporaries in some template functions.
- It is probably used as part of a runtime type-check that is necessary for implementing a dynamic cast.

Although the `typeid` can be printed, there is never a need to “understand” the printed result. This feature exists because the run-time system needs it. However, it could also be used by a programmer doing something unusual.

The three STL classes covered here are vector, string, and map. Vector is a sequence container and Map is a sorted associative container. String is derived from Vector, but is presented separately here because of its great usefulness, and because it supports many functions for string processing.

### 14.4 Using Simple STL Containers.

You have been using strings and vectors for a long time. The purpose of presenting them here is to present the most useful functions supported by the two class. In addition, a substantial program that uses vectors is presented.

#### 14.4.1 Vector

This example demonstrates several important parts of C++ including:

- The for-all loop used with an STL container.
- vectors and vector-iterators.
- How to use iterators.
- Two ways to subscript a vector.
- Several functions in the vector class.

**Vector demo:** Please note: An STL vector is more general than a Flexarray and has more functions. You should use it (not Flex) because it is part of the standard and presents the same interface as the other STL sequence container classes. It is safer than home-built code because it has been debugged under virtually all circumstances. The Flex array, however, is simpler and better for teaching how growing arrays work. It is also easy to use for what it implements: storing an indefinite amount of input data.

This demo is contained entirely within a main program and a print() function. It makes a vector then demonstrates how to do several things with it.

**Printing.**

- We use a range-based for loop when using a basic for loop, the programmer must handle incrementing and testing for loop-exit time. In contrast, a range-based loop deals with each item stored in a container, in order. This is possible because every STL container knows how many items are stored in it, and how to access each one. Line 10 shows this simple loop syntax: `for (int k : v) ...` The type of the variable `k` should be the same as the type of elements stored in the vector, `v`.

- When writing template code, which is used to store any and all types of data, the generic type `auto` can be used for the type of the data. The meaning of `auto` will be deduced by the compiler. However, `auto` should not be used for other purposes just because you are too lazy to write the correct type. If you do so, you will be throwing away much of the help that a compiler can give you. Typechecking relies on redundancy and it is important for achieving stable code.

---

1. This is sometimes called a “for-each” loop in C++ and a “for-all” loop in Java. We illustrate the simplest form of this loop.
2. Warning: This is an unsafe practice. Do’t do it..
14.4. USING SIMPLE STL CONTAINERS.

```cpp
#include <iostream> // for cout and <<
#include <vector> // for the container class and its functions.
#include <algorithm> // for sort
using namespace std; // To avoid the need to write std:: dozens of times.

void print(vector<int>& v) { // print out the elements of the vector
    int idx = 0;
    for (int k : v) cout << "Vector[" << idx++ << "] = " << k << endl;
}

int main( void ) {
    vector<int> numbers(\{11, 82, 24, 56, 6\});
    numbers.push_back(19);
    // Test subscript and at().
    cout << "Element 4 = " << numbers[4] << endl; // No bounds check
    cout << "Element 4 = " << numbers.at(4) << endl; // With bounds check
    cout << "Before sorting: " << endl;
    print(numbers); // print vector elements
    sort(numbers.begin(), numbers.end()); // sort vector elements
    cout << "After sorting: " << endl;
    print(numbers); // print elements again
    // search the vector for the number 3
    vector<int>::iterator pos;
    pos = find(numbers.begin(), numbers.end(), 3);
    if (pos == numbers.end())
        cout << "The value 3 was not found" << endl;
    cout << "First element in vector is " << numbers.front() << endl;
    cout << "Now remove last element and element [2] " << endl;
    numbers.pop_back(); // remove last element (efficient) &
    pos = find(numbers.begin(), numbers.end(), 24);
    if (pos != numbers.end())
        numbers.erase(pos); // remove element from middle (slow)
    print(numbers);
}

Declaration, allocation, and deallocation.

- On line 15 we construct and initialize a vector of integers. Among other parts, a vector has a pointer to a
dynamically allocated array that will grow when needed, a current allocation length, and the number of
data items stored in the vector.
- All the STL container classes are self-managing. The destructor inside the class will properly free any
dynamic storage used when the object goes out-of-scope. The programmer never sees or calls these
destructors.
- Line 16 shows another way to put elements into the vector. It is placed at the end of the filled portion of
the vector’s array.

Vector functions. Lines 18 and 19 show two ways, both useful, to access an element in a vector.

- The subscript operator is defined for the vector class and it works in the usual way. It does not perform
a bounds check.
- The at() function is also defined. It works similarly but first performs a bounds check to be sure the
subscript is legal and defined for the current size of the vector.
CHAPTER 14. TEMPLATES

- Which should you use? Use the operator if the subscript is very local and has been restricted to the proper range and and the loop ends at the size of the vector. Otherwise use the function. Use the function if there is any possibility that the subscript is invalid. Did the subscript come from Input? Then the use the operator.
- Why? The function is safer but slower. But, at some, point, you must trust the compiler to correctly compile the code in the program. There is no need to check everything all the time.

Iterators. The STL algorithms use iterators for most things

- Each STL container defines its own kind of iterators that “know” the structure of the container and are able to traverse its elements, visiting each element once.
- An iterator is like a pointer, but when incremented, it goes to the next unvisited item in the container.
- The functions begin() and end(), are defined for all STL containers. They return iterators associated with the first and last elements in the container. (end() is actually a pointer to the first vector slot past the end of the data.)

Algorithms and Iterators. The STL algorithms use iterators for most things.

- Line 23 calls sort(), one of the most useful template algorithms. The implementation is a quicksort that is very fast. STL also defines stable_sort() which does a mergesort.
- Lines 22 and 25 print the vector before and after sorting.
- Line 23: Two random-access iterators are used to call the template-algorithm sort: one for the beginning and the other for the end of the portion of the vector to be sorted.
- Line 28 declares an iterator variable of the right type for vector and Line 29 uses it to store the position at which a specific element is found. We can use this iterator later to access that element.
- Line 29: the find() function searches the part of the vector that is between the two iterators for the given key value (the third argument).
- Line 36: removing the last element from the vector is very efficient (constant time). Removing an element from any other position in a vector (line 37) is inefficient and should be avoided if possible. (All the data after the removal slot must be moved one slot toward the beginning, and any pointers into the array, past the removal point, are invalidated.)
- Lines 30 and 38 test for the iterator value end(), which is returned by the find function to signal failure to find the key value.
- Line 39: If find() did find the key element in the vector on line 48, then we remove it from the vector here. The gap will be closed up and the size of the vector will decrease. As mentioned, this is not an efficient operation and doing this it often will slow down execution.

| Element 4 = 6 | Vector[2] = 19 |
| Element 4 = 6 | Vector[3] = 24 |
| Vector[1] = 82 | The value 3 was not found |
| Vector[2] = 24 | |

The output:

| Vector[0] = 11 |
| Vector[0] = 6 |
| Vector[1] = 11 |

After sorting:

| Vector[0] = 6 |
| Vector[2] = 19 |
| Vector[3] = 56 |
14.4.2 Using Vectors

The Heap class Template. A heap is a balanced binary tree stored in an array, without pointers. Insert and remove operations are both $O(\log(n))$ complexity.

```cpp
#include "tools.hpp"

class Heap {
private:
    vector<BT>& d; // caller's data (in-out parameter)
    vector<BT> h; // our heap
    void downHeap( int start, BT key );
    void upHeap( int start, BT key );
    void heapify();
    void print();

public:
    Heap( vector<BT>& data ) : d(data) {
        h.push_back( BT() ); // Move v[0].
        for (BT item : d) h.push_back(item);
        //cout << "Ready to heapify\n";
        heapify();
    }

    void heapSort();
};

template <class BT> void
Heap<BT>:: heapSort() {
    for (int k = (h.size()-1) / 2; k >= 1; k--) {
        //cout << "downheap from " <<k << " with " <<h[k] <<endl;
        downHeap( k, h[k] );
    }
}

template <class BT> void
Heap<BT>:: print() {
    for (int k=1; k < h.size(); ++k) cout << h[k] <<endl;
    cout <<"\n";
}
```
Notes on the Heap class Template. Please note that every function has extensive documentation. This takes space and time but must be done for a program that will be used professionally.

- The Heap class has two vectors as data members. In line 16, a vector is passed as a reference parameter to avoid the copying that would happen if it were passed by value.
- Line 18 declares a second vector that will be constructed with a quirky property: slot 0 is left blank. This enables the heap algorithms to be written without dozens of repetitions of -1.
- The heap constructor copies or moves the first data item into the last slot of the array. This assumes that BT is a copyable type or that it defines move assignment.
- The task of the heapify() function is to rearrange the data elements into heap order. It is a separate function because it is a standard heap operation and its operation is complex. It is a private function because it is only called once, from the constructor.
- print() omits slot 0, which will never contain data. This function is for debugging, not for the client.

```cpp
59 // ----------------------------------------------------------------------
60 // Pre: The data elements are in heap order.
61 // Post: The maximum element has been removed from the heap and returned.
62 template <class BT> BT
63 Heap<BT>:: remove() {
64     BT answer = h[1];
65     BT key = h.back();
66     h.pop_back();
67     downHeap( 1, key );
68     return answer;
69 }
70 // ----------------------------------------------------------------------
71 // Pre: The heap has n elements in it. The root position, v[start],
72 // is empty, and the key is to be stored in the heap. this->v.size()
73 // is the last occupied slot and the number of items in the vector.
74 // Post: The elements on every path from root to leaf are sorted.
75 template <class BT> void
76 Heap<BT>:: downHeap( int start, BT key ) {
77     int father = start;
78     int son, rson;
79     for (;;) {
80         son = 2*father;
81         rson = son + 1;
82         if (son >= h.size()) break; // This is off the end of the tree.
83         if (rson < h.size() && h[son] < h[rson]) son = rson;
84         if ( h[son] < key ) break;
85         h[father] = h[son];
86         father = son;
87     }
88     h[father] = key;
89 }
90 // ----------------------------------------------------------------------
91 // Pre: The data elements are in heap order.
92 // Post: The key has been inserted and put in its proper place.
93 template <class BT> void
94 Heap<BT>:: insert( BT key ) {
95     cout << "inserting " << key << endl;
96     h.push_back(key);
97     upHeap( h.size()-1, key );
98 }
99 // ----------------------------------------------------------------------
100 // Pre: A new key is to be stored in the heap. this->v.size() is the last
101 // occupied slot and is also the number of BT items in the vector.
```
14.4. USING SIMPLE STL CONTAINERS.

// Post: The new key is stored in the heap and the heap is in heap order.

template <class BT> void
Heap<BT>:: upHeap( int start, BT key ) {
    int son = start;
    int father;
    while (son>1) {
        father = son/2;
        if ( key <= h[father] ) break;
        h[son] = h[father];
        son = father;
    }
    h[son] = key;
    // print();
}

//---------------------------------------------------------------------
// Sort the array into ascending order.
// It must start in heap order.
// The sorted array elements will end up in v[1] through v[v.size()].

template <class BT> void
Heap<BT>:: heapSort() {
    d.erase(d.begin(), d.end()); // Sorted data will be put here later.
    for (int k = h.size()-1; k > 0; k--) { // Start process at middle of heap.
        BT temp = remove(); // Remove item in slot 1; replace by v[k].
        d.push_back( temp ); // Store the removed item in the now-empty slot.
        // cout << ">>> Pulled " << temp << endl;
    }
}

Notes on the Heapsort implementation. These implement the ordinary heap operations.

- **remove()**: First, the value at the root of the stack is removed and returned. Then the value at the end of the vector is moved to the top, and finally, downheap is called to put that value into its correct place in the tree.

- **downheap()** is called by **heapify()** and by **remove()**. It starts at the root of the heap with a data value that needs to be placed properly, and goes downward toward the leaves, moving either right or left depending on comparisons with heap elements at each step, and moving the heap element upward one step. When the new value is greater than both of its sons, it is put into the hole.

- **insert()**: Store the new key value at the end of the vector and call **upheap()**.

- **upheap()**: For a max-heap, start at the slot that contains the new key. Then compare the key to each data value, as you move up the tree. If the key is greater than the tree-value, move the tree-value down. Stop and insert the key when the value in the tree is greater.

- **heapsort()** repeatedly removes the element at the root and stores in in the caller’s data vector. This continues until the heap is empty.

Notes on main(). This main program tests the heap operations and heapsort.

- Line 154 instantiates a vector with the type **Item**, defined below. There is no mention of Items in the heapsort code.

- **getData()**: brings the test data from a file into main’s vector. Two files were used: one with 16 items in it, one per row, and an empty file. The heap implementation handles both appropriately.

- Lines 155–156 open an input stream and make sure it is open, before calling **getData()**.

- Lines 158–162: When testing new code, it is always a good idea to print it out. Since it is in a vector, we can use a range-based loop.

- Line 164 creates a heap using the data vector, and prints it.

- Lines 166–168 exercise the heap operations. All but **insert()** are called by **heapsort**.
void getData(ifstream& source, vector<Item>& store){
    string name;
    int score;
    for(;;){
        source >> name >> score;
        if (source.eof()) break;
        store.push_back(Item(name, score));
    }
}

int main (void) {
    vector<Item> mydata;
    ifstream input ("accounts.txt");
    if (! input.is_open()) fatal("Cannot open accounts.txt for input.
    getData(input, mydata);
    cout << "Output from main, which calls the heap constructor and print.
    There are " << mydata.size() " items in the array."
    cout << "Before arranging in heap order:
    for (Item it : mydata) cout <<it <<endl;
    cout << "===================
    Heap<Item> h( mydata ); // Heapify the data in the vector.
    h.print();
    h.insert( Item("Ricky", 28));
    h.insert( Item("Sasha", 99));
    h.heapSort( );
    cout << "After finishing the sort:
    for (Item it: mydata) cout << it <<endl;
}

The Item class.

string name; // Array of char for student name
int account; // Integer to hold score
Item(string init, int sc): name(init), account(sc) {}
Item(): name("dummy"), account(0) {}
• This is a very simple class. It could be replaced by anything. However, the instantiation parameter for the heap must either be copyable or implement move assignment.
• Line 186, the default constructor, is needed by Heap to make a properly formed dummy value for slot 0 of the heap.
• Lines 188 and 189 define comparison operators that are needed by Heap.
• These requirements need to be explained in the documentation for heap.

14.4.3 String

The string class was originally derived from `vector<char>`, so it inherited the functions in `vector`. Then many functions that echo the C string library were added to support string processing. Vector and string share the ability to grow and share many functions.

String demo:

```cpp
// STL string example. A. Fischer, March 2003, Revised August 2021
#include <string> // header for C++ strings
#include <iostream>
using namespace std;

int main( void ) {
    string str1 = "This is string number one.; // Allocate and initialize.
    // The length of a string
    cout << "String str1 is: \" \" << str1.c_str() <<\"\". "
    << " Its length is: " << str1.size() << "\n\n";
    cout << "Get a substring of six letters starting at subscript 8: ";
    string str2 = str1.substr(8,6);
    cout << str2 << endl;
    // search first string for last instance of the letter 'e'
    int idx = str1.find_last_of("e");
    if (idx != str1.length())
        cout << "The last 'e' in string str1 is at subscript " << idx << endl;
    else cout << "No char 'e' found in string str1" << endl;
    // search second string for first instance of the letter 'x'.
    idx = str1.find_first_of("x");
    if (idx != std::string::npos)
        cout << "The first 'x' in string str2 is at subscript " << idx << endl;
    else cout << "No char 'x' found in string str2\n\n";
    cout << "Now replace "string" with "xxxyyyxxx"."
    idx = str1.find("string");
    if (idx != std::string::npos)
        str1.replace(idx, string("string").length(), "xxxyyyxxx");
    cout << "str1 with replacement is: " << str1.c_str() << endl;
    return 0;
}
```

The output:

String str1 is: "This is string number one.". Its length is: 26
Get a substring of six letters starting at subscript 8: string
The last 'e' in string str1 is at subscript 24
No char 'x' found in string str2
Now replace "string" with "xxxyyyxxx".
str1 with replacement is: This is xxxyyyxxx number one.
Notes on the string demo:

- Line 61: the header command needed for this class.
- Whichever kind of string you are using (C string or C++ string), there are a few times when you want the other kind. It is good to memorize these two methods. On line 68 we see one way to get a C-style string out of a C++ string: `str1.c_str()`. The other way is: `str1.data()`.
- Line 69: There are two ways to get the length of the string: `str.size()` (line 69) and `str1.length` (line 77). Why this duplication: “length()” is the terminology used in C, while “size” is a general function defined for all C++ containers. The results are the same, so use whichever one comes to mind.
- Line 72 creates a new string as a copy of part of another string.
- Line 73: `operator <<` is defined for C++ strings.
- Lines 76, 82, and 88 demonstrate three different `find` functions for searching a string for a substring. The C++ string library has several find functions, including one similar to C’s `strchr()` that searches a string for one letter.
- Lines 77 and 83 show two ways to test whether the search failed. `npos` is defined as −1, an illegal subscript, and is used to signal the failure of the search operation.
- Line 90: To replace a substring with another string, you must supply a subscript where the substitution should start, the length of the substring being replaced, and the replacement. Note that the old and new substrings do not need to be the same length.

14.4.4 Map

A map is an associative container, a collection of key/value pairs. When a value is stored into a map, it is stored using one of its fields, called the key field. To retrieve that value, you use the key field to locate it.

```cpp
100 // A. Fischer, March 2003, Revised August 2021
101 #include <map>
102 #include <iostream>
103 #include <string>
104 using namespace std;
105 using numStr = pair<int, string>;
106
107 int main( void ) { // ---------------------------------------------
108     map<int, string> myMap; // create a map
109     map<int, string>::iterator it;
110 
111     myMap[4] = "Andrea"; // insert element into the map
112     myMap.insert(numStr(2, "Barbara")); // another way.
113     
114     // print all elements ------------------------------------------
115     for (numStr pr : myMap)
116         cout <<"Key = " <<pr.first <<", Value = " <<pr.second <<endl;
117 
118     // try some operations -----------------------------------------
119     it = myMap.find(2);
120     if (it == myMap.end()) cout << "\nKey value 2 not found" << endl;
121     else cout <<\nValue for key 2 = " << it->second << endl;
122 
123     it = myMap.find(3);
124     if (it == myMap.end()) cout << "Value for key 3 not found\n";
125     
126     // get # of elements in map -------------------------------------
127     cout << "\nThe number of elements in myMap is " << myMap.size() << endl;
128     cout << "Now erase one element from map.\n";
129     myMap.erase(2);
130     cout << "The number of elements in myMap is "
131         << myMap.size() << endl;
132 }
```
14.4. USING SIMPLE STL CONTAINERS.

Notes on the map code.

- Line 101 shows the header file needed for this class.
- Line 105 defines a convenient shorthand for \texttt{pair<int, string>}
- Lines 108 and 109 create a map object and an appropriate iterator. Note that both templates require two parameters: the type of the key field and the type items stored in the container.
- Lines 111 implicitly creates the pair \texttt{4, Andrea} and inserts it into the map.
- Line 12 constructs a pair within the argument list of the insert function, and inserts it into the map.
- Line 115 uses the type name defined on line 105 in a range-based loop that iterates through the container and prints each element.
- In a range-based for loop, the parenthesized part must give the typename of the base type of the container or array, and an arbitrary parameter name to be used in the code. Line 115 uses the name \texttt{pr} and gives the type \texttt{numStr} (a shorthand for \texttt{pair<int, string>}), because a map contains a set of pairs.
- Line 116 Each pair has two members, named \texttt{first} and \texttt{second}, these member names are used to access the key and the data, respectively.
- Lines 119 and 123 call \texttt{map::find()}, supplying the key. Compare this to the call on \texttt{vector::find()} in line 37: the required parameters are different but both return an iterator that points to the found item.
- Lines 120 and 124 test for success of the find operation by comparing the result to \texttt{myMap.end()}. This is like the test on line 38 in the vector example. STL provides a common framework for using all the collections.
- Line 127 calls \texttt{myMap.size()} to get the number of pairs stored in the map.
- Line 129 removes a specific pair from the map, given its key value.

The output:

\begin{verbatim}
  Key = 2, Value = Barbara
  Key = 4, Value = Andrea

  Value for key 2 = Barbara
  Value for key 3 not found

  The number of elements in myMap is 2
  Now erase one element from map.
  The number of elements in myMap is 1
\end{verbatim}

Implementations. A map can be implemented as a hash table or a balanced search tree. Both make sense and serve the purpose. We do not actually know which is used, since that is not dictated by the standard. The standard guarantees performance properties, but not implementation. However, I believe that a map is implemented by a balanced tree and an unordered\_map is implemented by a hash table. All classes, including template classes, should encapsulate their members and provide public functions for all essential tasks. This leads to a design conflict: should a template definition include every possible function that someone might someday need? Or should the class be clean and focus all functions on its primary purpose?

An STL map can hold only unique keys; if more than one item with the same key can exist, you must use a multimap instead. If there is no need, ever, to enumerate or print the elements in order, one can use an \texttt{unordered\_map}, for which retrieval is more efficient.

Conclusion The three STL classes introduced here are among the simplest and most useful. But these examples are only “the tip of the iceberg”. The capabilities of these and other STL classes and algorithms go far beyond what is shown here. The three examples are just a starting point from which the student can continue to learn and master this important aspect of modern programming practice.
14.5 Adapting a Template

Sometimes it is useful to have two or more templates available for variations on one basic data structure. For example, there are three variations on the map template: ordered map, unordered map, and multi map. Derivation can be used to achieve this kind of flexibility with minimal effort and minimal duplicated code.

The basic template should be defined with the fewest possible restrictions, then template classes with restrictions can be derived from it to support specialized applications. Taking the STL map as an example, an unordered map can be used with more data base-types than an ordered map, for which a base type needs to support the comparison operators. In such a situation, the solution is to derive another template that includes the `<` function and other functions that depend on a defined order. In this way, one can have a template that works either with or without the restrictions.

It is common to wish to change a class or a template by removing functionality, adding functionality, or changing the names of functions. This is so common that it has been captured in a design pattern called “class adaptor”. A few details should be noted:

• If derivation is involved, the data parts of the Flex class must be protected, not privates.
• Public derivation was used to build the derived template so that all the functionality of Flex will remain available to a client program that derives from FlexFind.
• When a class is derived from a template, its constructor must have an initializer list that supplies a type parameter for the template class, plus any other parameters the template constructor needs.

Adding functionality.
The code for Flex<T>. However, he must write code that works in a broad range of situations, and this must be done without breaking class encapsulation. The solution is to require a potential type-parameter class to provide one or more public functions. The template code is then written to use those functions. In this case, any type F that is used with this template must provide a definition for `print(ostream&)`.

A class name and :: written before the name of a class member avoids ambiguity in situations where a name might otherwise be ambiguous or is not yet fully defined, as during template instantiation.

Removing functionality from a class. Sometimes, when you derive from an existing class or template, it is desirable to eliminate some of the functionality that the base class/template has. This is straightforward: delete it. For example, the subscript operator is defined for Flex, but it should not be defined for Stack because random access to a Stack should not be permitted. Here is the code that removes the subscript operator from the Stack class (line 43): `void operator[] (int x) =delete; // Prevent use of Flex’s subscript`. In place of subscript, Stack defined push and pop.

Example.
In order to illustrate template adaptation, we will use the Flex class from the Dictionary example in Chapter 10. We are not using vector because many sorts of functions are built into that class and there is nothing simple left to add.

14.5.1 Deriving from a Template

A precedence parser starts with code, analyzes it, and produces an expression tree. Every C++ compiler and desktop calculator is based on a precedence parser. This application implements a simple parser (no parentheses, no unary operators). It is moderately complex code that does careful detection of logic errors. For that reason, the logic gets quite complex.

A precedence parser uses two stacks to interpret the meaning of an arithmetic expression. To illustrate template derivation, we derive a Stack template from the Flex template. To begin, the template class from Chapter 10 was modified by making the private parts protected, and by adding the const qualifier to functions that do not change this.
Notes on changes to the Flex template.

- Line 6 is not necessary; the program compiles and works perfectly well without the std::. However, without it, my IDE produces a warning comment each time I use “move”. To get rid of the meaningless error comments, I added line 6. This tells the compiler that move should be replaced by std::move everywhere in the module.

- The private members of the Chapter 10 template have now become protected because we intend to use this class as a base class for derivation.

- The length() and print() functions on lines 20 and 22 now are const functions. The compiler will enforce this restriction, which is important in a large development project. This const should always be used when the code of a function does not modify this. It should have been used in Chapter 10. Now, trying to derive another template from Flex, the const is required because the derived template uses it.

Flex is the base template for the derived template.

```
1  // Class declaration for a flexible array of base type T.
2  // A. Fischer, A. Fischer, July, 2022 file: flex.hpp
3  #pragma once
4  #include "tools.hpp"
5  #define move std::move
6  #define START 4   // Default length for initial array.
7  // ____________________________________________________________
8  template <class T>
9  class Flex{
10     protected: //-------------------------------------------------------
11         int max = START;  // Current allocation size.
12         int n = 0;      // Number of array slots that contain data.
13         T* data;       // Pointer to dynamic array.
14         void grow();
15     public: //-----------------------------------------------------
16         Flex() : data(new T[START]) {}
17         T* operator[]( int k );
18         int length() const { return n; } // Read-only access to array length.
19         T& operator[] ( int k );
20         ostream& print(ostream& out) const {
21             for ( int k=0; k<n; ++k) out << data[k] << " ";
22             return out;
23         }
24     };
25  }
26  template <class T> void Flex<T>::: //-----------------------------
27  grow(){
28         T* temp = data;       // hang onto old data array.
29         max *= 2;              // Double the allocation length.
30         data = new T[max];   // allocate a bigger one.
31         for(int k=0; k<n; ++k) data[k] = move(temp[k]);
32         delete[] temp;
33     };
34  template <class T> T& Flex<T>::: //-----------------------------
35  operator[]( int k ){
36      if ( k>=n ) fatal(string("Flexarray bounds error.
37      return data[k];    // Return & of desired array slot.
38  }
39  template <class T> int Flex<T>::: //-----------------------------
40  push( T dt ) {
41      if ( n==max ) grow();    // Make more space if needed.
42      data[n] = move(dt);    // Store current data.
43      return n++;            // Return index of stored item.
44  }
45  template <class T> inline ostream& //-----------------------------
46  operator<<(ostream& out, Flex<T>& F){ return F.print(out); }
```
The stack is a familiar data structure with restrictions on access. The rule for pushing an item onto a stack fits well with the append-on-the-end rule of the Flex. However, a Stack does not support random access or sequential search. Because of the basic similarity, we can use a Flex to implement a stack. In this example, new functions provide the familiar stack interface, and private derivation is used to hide the inherited Flex class members.

### 14.5.2 Modifying a template: Stack

```cpp
#pragma once

// ---------------------------------------------------------------------------
// Template class definition for a stack of T objects
// Alice E. Fischer June 10, 2000, August 2022 file: stackT.hpp
// ---------------------------------------------------------------------------

#include "flexT.hpp"

template <class T> //-----------------------------------------------------------
template <class T> inline //---------------------------------------------
ostream& operator<<( ostream& out, Stack<T>& s ){ return s.print(out); }
```

Notes on the derived template.

- Lines 56–57 declare a new template class named Stack. The colon denotes class derivation; the template Flex<T> is the base type. It is essential that the same parameter name, T, is used on lines 57 and 59.
- A derived template or class, by default, contains all the data members and functionality of the base class, but the visibility (or not) of base class members depends on the style of derivation declared on the first line of the class. Here we use public derivation, meaning that everything Stack inherits will have the same privacy level as it does in Flex.
- Protected class members of a base class are visible to all derived classes when either public derivation or protected derivation is used. Private class members are not visible in the derived class, but can be used through inherited public functions. Accordingly, the `grow()` function cannot be called from the Stack class, it will be called only from the push function.
14.6. A PRECEDENCE PARSER: INSTANTIATION AND USE OF STACK

- This Stack class template uses protected derivation (line 57). That means that private members will remain private, while the public and protected members become protected in Stack<T>. Public derivation would also work.

Special functions for Stack.
- The constructor (line 59) is defined as =default. This relies on a default destructor in the Flex template (line 17). If that did not exist, the Stack constructor would need to use a ctor to explicitly call the Flex constructor.
- The destructor is defaulted because there is no work for it to do. All the data members of Stack are inherited from Flex. One of those (the array) is dynamically allocated, but freeing it is fully handled by the Flex destructor. We say that Flex is a "self managing" class.
- The copy constructor and copy assignment operator are both deleted (lines 61–62). The move constructor and move assignment operator are both deleted (lines 63–64) so you also cannot move a stack or collect these stacks in a larger data structure.
- This is a radical thing to do, but in this case, the author of the Stack template cannot see any compelling purpose why a program would ever need to copy or move a stack. However, deleting the copy and move constructors does mean that a Stack must always be passed by reference when a function has a Stack parameter.
- Line 65 deletes the inherited subscript operator to implement the semantic nature of a stack: the only accessible part of the stack should be the top.
- Lines 67–71 and 80 all use Flex<T>:: preceding the names of the inherited members of the Flex class. This would not be necessary if we were deriving a non-template. It is necessary for the compiler in this situation because of the order in which templates are expanded during compilation.
- The five traditional stack functions are defined on lines 67–71. Two of them simply rename Flex members (add() and depth()). Two functions (pop() and peek()) are accessors that implement the LIFO restriction on accessing stacks and replace of a subscript function.
- The print function (lines 77-83) “wraps” the output from Flex::print() with a header and other info that mark the bottom and top of the stack.

14.6 A Precedence Parser: Instantiation and Use of Stack

Instantiation is the process of using a template, with actual arguments, to create compilable code for a class declaration and implementation. This is done early in compile time, before parsing, when the compiler first sees an instantiation command. In this example, we instantiate the derived template class, Stack<T> twice to make two different kinds of stacks that are used side-by-side to implement an infix expression evaluator. This simplified evaluator could easily be extended to handle parentheses and non-binary operators, but our purpose here is to explore the use of C+++, not to write a general and powerful evaluator. The Eval class is presented first, followed by a small Operator class, the main program, and some output.

Using the Stack template.
- To instantiate a template, the programmer writes a declaration or a call on new using the class name with an argument list in angle brackets. Two examples are given here, lines 185 and 186:
  Stack<Operator> ators{}; // Stack of pending operators.
  Stack<double> ands{}; // Pending operands and intermediate results.
- Even though the keyword class is used in the template declaration, a non-class type such as double may be used as an argument. A struct type such as Operator may also be used.
- The name of the resulting class includes the <> and the type argument, and each instantiation with a different parameter creates a new class with a unique name. It also creates a new module of compilable code. Here, we create two complete and separate Stack classes, Stack<double> and Stack<Operator>. Each Stack function is compiled twice, once for Stack<double>, then again for Stack<Operator>.
• If a class template is instantiated twice with the same arguments, a compilable module is produced the first time and reused the second time. Suppose we added a third instantiation: `Stack<double> Results`. This third call would refer to the class created by the prior instantiation, `Stack<double> Ands`. So `Results` and `Ands` would be objects of the same class.

### 14.6.1 Evaluation using precedence.

Precedence and associativity are used to define the meaning of operators in most modern languages. The evaluate function implements both. Basically, operators are kept on one stack and operands on another.

- When each operator is read, a decision must be made whether to evaluate the previous operator or stack the new one.
- If the precedence of the incoming operator is greater than the precedence of the stack-top operator, it is not yet time to evaluate either one, and the incoming operator is added to the stack.
- If the precedence of the two operators is equal, the rule for associativity comes into play. Arithmetic operators need left-to-right associativity, so we evaluate the leftmost, which is the one on the top of the stack. So we pop the stacked operator and evaluate it. Then we compare the incoming operator to the next one on the stack.
- Lower incoming precedence also means that the stacked operator should be popped and evaluated immediately (its operands are at the top of the `Ands` stack). Once that is done, the incoming operator must be compared to the new top-of-stack, and so on.
- Lines `xxx` `xxx` handle the end of the expression. In this situation, all operands have been read and stacked, and all operators that are still on the stack must be evaluated, so we force everything with precedence greater than 0 to be dispatched. If there are no errors in the expression, the value that remains on the operand stack is the answer.

```cpp
90 // ========================================================================
92 // Parse and evaluate a prefix expression. All operators are binary;
93 // they are: +, -, *, /, % (a mod b) and ^ (a to the power b)
94 // Operands must start with a digit and may have a decimal point.
95 // Operands must not exceed 30 keystrokes.
96 //
97 #pragma once
98 #include "tools.hpp"
99 #include "stackT.hpp"
100 #include "operator.hpp"
101
class Eval {
102 private: // -----------------------------------------------
103 enum class Intype { bad, number, op, end };
104 Stack<Operator> ators{}; // Stack of pending operators.
105 Stack<double> ands{}; // Pending operands and intermediate results.
106 Intype classify( char ch );
107 void force( int rprec );
108 void dispatch();
109 double expError();
110
class Eval {
111 Eval() =default;
112 ~Eval() =default;
113 static void instructions( void );
114 double evaluate( istream& in );
115 ostream& print( ostream& out );
116 };
```
A Private Type and Private Functions. None of these things should be visible or used outside this class.

- A private enum class is declared (line 104) and used in classify() to simplify the input and parsing operations. One enumeration symbol is listed for each legal kind of keystroke and one for bad data. The enum symbols are returned by the function classify() and used in evaluate().
- The classify function (lines 109 and 127–139). The type of the value returned by classify() is referred to within the class as simply Intype, to declare local variables in class functions (line 185). However, the return type of this function (lines 129 and 136–38) must be written as Eval::Intype because: the definition of the enum class is inside the Eval class and the encapsulated enum type is not visible to the compiler unless you qualify it with the class name.
- The force function (lines 110 and 142–145) is the heart of a precedence parser. It is responsible for comparing the precedence of the incoming operator with the precedence of the operator on the top of the stack and evaluating as many operators as are appropriate, according to precedence. To evaluate an operator, it calls the dispatch function.
- The dispatch function (lines 111 and 147–162) pops two operands from the Ands stack and one operator from the Ators stack, interprets the operator, calls the appropriate C operator or library function, and puts the result back on the Ands stack.
- The expError function is handles errors with undefined expression syntax.

```cpp
121 // ============================================================================================
122 // A. Fischer, June 9, 2002 file: eval.cpp
123 #include "eval.hpp"
124 #include "operator.hpp"
125
126 //---- Decide whether next input char is an operator, a semicolon, the beginning
127 Eval::Intype Eval:: // of an operand, or garbage.
128 classify( char ch ){
129     if (isdigit( ch )) return Intype::number;
130     switch(ch){
131         case '+':
132         case '-':
133         case '*':
134         case '/':
135         case '%': return Intype::op;
136         case ';': return Intype::end;
137         default : return Intype::bad;
138     }
139 }
140 // --------------------------------------------------------------------------------
141 // Evaluate all higher precedence operators on stack.
142 void Eval::
143     force( int rprec ) {
144         while( ators.depth()>0 && ators.peek().precedence() >= rprec ) dispatch();
145     }
146 // --------------------------------------------------------------------------------
147 void Eval::
148     dispatch() {
149         double result;
150         double right = ands.pop();
151         double left = ands.pop();
152         Operator op = ators.pop();
153         switch (op.symbol()) {
154             case '+': result = left + right; break;
155             case '-': result = left - right; break;
156             case '*': result = left * right; break;
157             case '/': result = left / right; break;
158             case '%': result = fmod(left, right); break;
159             case '^': result = pow (left, right); break;
160         }
161         ands.push( result );
162     }
```
// ------------------------------------------------------------- Error comments.

double Eval::
expError()
{
    cerr << "Illegal expression.\n";
    print(cerr);
    return HUGE_VAL;
}
// ---------------------------------- (static) Instructions for the operator.
void Eval::instructions( void ){
    cout
<< "This is a simple infix expression evaluator.\n"
<< "* Operands start with a digit and may or may not have a decimal point.\n"
<< "* All operators are binary operators. Parentheses are not supported.\n"
<< "* Operators are: +, -, *, % (a mod b) and ^ (a to the power b).\n"
<< "* End each expression with a newline.\n"
<< "* To QUIT, type semicolon (;) instead of an expression.\n";
}
// ----------------------------------------------------------- Read input and evaluate expression.
double Eval::
evaluate( istream& in ) {
    Intype next; // Classification of next input character.
    Operator inOp; // Operator object constructed from inSymbol.
    double inNum; // Read input operands into this.
    char ch;

    in >> skipws;
    for(;;) {
        in >> ch;
        if (in.eof()) next = Intype::end;
        else next = classify( ch );
        switch( next ){
            case Intype::number:
                in.putback(ch); // Undo the last read.
                in >> inNum;
                if ( ands.depth() != ators.depth() ) return expError();
                ands.push( inNum );
                break;
            case Intype::op:
                inOp = Operator(ch);
                if ( ands.depth() != ators.depth()+1 ) return expError();
                force( inOp.precedence() );
                ators.push( inOp );
                break;
            case Intype::end:
                if (ands.depth() != ators.depth()+1) return expError();
                force( 0 );
                return ands.pop();
                break;
            case Intype::bad:
                default: return expError();
                }
        }
    }
    // ----------------------------------------------------------- Print the stacks.
    ostream& Eval::
print( ostream& out ){
        out << "Remaining contents of operator stack: " <<ators;
        out << "Remaining contents of operand stack: " <<ands;
        return out;
    }
}
Public functions and notable things. There are only three public functions: the static function for supplying user instructions, print(), and evaluate() the function that performs the precedence parse.

- Lines 117 and 171–179 declare a static class function named instructions. Static class functions cannot use any class members but they can be called even when no object of the class type exists. This one is static because we want main to be able to call this function before creating the first Eval object. It is a class function (not global) because it gives expert instructions about the usage and requirements of the Eval class.
- Note that the word static is used in the declaration on line 117 but not in the remote definition on line 172.
- Lines 194–199 handle numeric inputs that are not restricted to single digits. Number conversion is a complex and picky process that is normally done by the I/O system. DO NOT ever try to do your own number conversion. Use the built-in facilities.
- The problem here is that we need to read the first digit of a number (line 192) to identify that it is a number. If so, we call putback(ch) (line 195). This function does not really modify the input stream! It simply moves the input pointer back one character in the input buffer so that the character will be reread. Then, in line 196, it uses the usual input operator (and the built-in number conversion) to turn the input chars into a binary number. This is much simpler and nicer than using strtod to do the conversion to double.
- The rest of the evaluate function handles operators, newline, and illegal chars.
- HUGE_VAL (line 168) is defined in cmath. It is the largest representable floating-point value, and is used here to signify an error.
- Input error checking is done in the classify function (lines 127–139) and throughout the evaluate function (lines 197, 203, 209, and 215) so that unsupported operators and ill-formed expressions are caught as soon as possible.

14.6.2 The Operator Class

Notes on the operator class. This class exists to store the operator char and its precedence conveniently together. Specific values for precedence are not important; the relationships among the precedences of different operators matter greatly.
CHAPTER 14. TEMPLATES

• We represent an operator as a two-member object: the symbol used to denote the operator, and the precedence of the operator with respect to other operators that are supported. The switch statement in the Operator constructor (lines 262–267) defines the precedence.

• The precedence is used on line 144, where we move down the stack, popping and evaluating all operators that are higher precedence than the new input.

• The symbol is used on line 153–160 to select the C++ operator to use to evaluate the current subexpression.

• We use simple “get” functions (lines 245–246) to make these private members of Operator available to the application in a read-only form.

14.6.3 The Main Program

I have written this algorithm in C and Pascal, also. The C++ version is considerably cleaner, simpler, easier to write, and easier to understand. The improvement is amazing to me, and is made possible by having classes with their own constructors, print functions, error handling, etc.

Overall operation.

• We start by calling the instructions function. The instructions could be written as part of main, but since they are very closely tied to the capabilities of the Eval class, it is better to define an instructions function there and call it from main.

• Main contains the loop that interacts directly with the user. It reads a line of input and quits if it finds the end-of-job sentinel. Otherwise, it creates and uses a new expression evaluator and prints the result.

• We declare the evaluator as a local object (line 269) for each new expression. Using local declarations for the evaluator and the stringstream (line 268) makes initialization very simple and avoids any possibility of having the remains of one expression affect the evaluation of the next one.

```cpp
#include "eval.hpp"
#include "tools.hpp"

int main( void ) {
    string buf;  // Buffer for one line of input.
    double answer = 0;
    banner();
    Eval::instructions();
    for(;;){
        cout <<"\nEnter an expression (; to quit): ";
        cin >> ws;
        getline(cin, buf);
        if ( buf[0] == ';') break;
        istringstream inst( buf);
        Eval E;
        answer = E.evaluate( inst );
        cout << buf << " = ";
        if (answer == HUGE_VAL) cout << "[eval error]" << endl;
        else cout << answer <<endl;
    }
    bye();
}
```

String-streams.

• Line 268 declares an input string-stream. The istringstream class wraps the interface of an input stream around an existing string, and uses the string as its source of data instead of an input file. We initialize the string-stream to the line of input that was read on line 266.
14.7 UML for Templates

We represent a template class in UML by drawing an ordinary class rectangle with a small dotted rectangle covering its upper-right corner. The names of the template parameters are written in the dotted rectangle. The diagram shows two template classes, FlexArray and Stack. A derived template class is diagrammed like any other derived class, with the template part added. In this diagram, the Stack template is joined to the FlexArray template with an ordinary derivation symbol.

In a UML diagram, an instantiation of a parameterized class is called a bound element. It is diagrammed as a class box with a name like Stack<char>. The type to which the template parameter is bound is given in a <bind> notation on a dotted arrow joining the instantiated class to the box for the template class.

Instantiating a template with two different types produces two separate bound elements. For example, our program that evaluates infix expressions uses two stacks: one for operators, the other for values, so we instantiate the Stack template twice, once as Stack<Operator> and once as Stack<double>. The instantiations create two new classes that are fully bound and can be compiled.

The output.

This is a simple infix expression evaluator.
* Operands start with a digit and may or may not have a decimal point.
* All operators are binary operators. Parentheses are not supported.
* Operators are: +, -, *, /, % (a mod b) and ^ (a to the power b).
* End each expression with a newline.
* To QUIT, type semicolon (;) instead of an expression.

Enter an expression (; to quit): 3+5 = 8
Illegal expression.
Remaining contents of operator stack:
Stack has 1 items: bottom<Symbol: + Precedence: 1
]>top
Remaining contents of operand stack:
Stack has 2 items: bottom<[3 5 ]>top
3+5 = 8 = [eval error]

Enter an expression (; to quit): 3 + 5
3 + 5 = 8

Enter an expression (; to quit): 3+5
3+5 = 8

Enter an expression (; to quit): ;
Normal termination.
14.7.1 Instantiate and derive.

It is quite common to instantiate a template and derive a class from it in one step. This lets the programmer give a simple and meaningful name to the resulting class. Suppose we were to instantiate the Stack template using type `int`, to make a `Stack<int>`. That is an awkward name to be writing repeatedly in a program. We can then derive a class with a meaningful name from it, for example, `Scores`. Happily, both steps can be done in one unified declaration:

```
class Scores : public Stack<int> { ... }
```

The UML diagram would look like one of these. On the left is a 2-step diagram, on the right, a diagram that combines the steps:
Chapter 15: Design Patterns

Design patterns are elegant, adaptable, and reusable solutions to everyday software development problems. Each pattern includes a description of a commonly occurring type of problem, a design for a set of classes and class relationships that solve that problem, and reasons why the given solution is wise.

15.1 Definitions and General OO Principles

15.1.1 Definitions

1. Subclass: X is a subclass of Y if X is derived from Y directly or indirectly.
2. Collaboration: two or more objects that participate in a client/server relationship in order to provide a service.
3. Coupling: A dependency between program elements (such as classes) typically resulting from collaboration between them to provide a service. Classes X and Y are coupled if...
   - X has a function with parameter or local variable of class Y.
   - X has a data member that points at something of class Y.
   - X is a subclass of Y.
   - X implements an interface for class Y (Y gives friendship to X).

[Example:] If the Key class calculates the hash-table index, it must know the length of the hash table. But this couples two classes that would not otherwise be coupled.

4. Cohesion: This is a measure of how strongly related and focused the responsibilities of a class are. A class with high cohesion is a “specialist” with narrow power.

5. System event: A high-level event generated by an external actor; an external input event. For each system event, there is a corresponding operation. For example, when a word-processor user hits the “spell check” button, he is generating a system event indication “perform spell check”.

6. Use case: The sequence of events and actions that occur when a user participates in a dialog with a system during a meaningful process.

15.2 General OO Principles

1. Encapsulation. Data members should be private. Public accessing functions should be defined only when absolutely necessary. [Why] This minimizes the possibility of getting inconsistent data in an object and minimizes the ways in which one class can depend on the representation of another.

2. Narrow interface. Keep the interface (set of public functions) as simple as possible; include only those functions that are of direct interest to client classes. Utility functions that are used only to implement the interface should be kept private. [Why] This minimizes the chance for information to leak out of the class or for a function to be used inappropriately.

3. Delegation: a class that is called upon to perform a task often delegates that task (or part of it) to one of its members who is an expert. [Example:] HashTable::find selects one list and delegates the searching task to List::find.
15.3 Patterns

A pattern is a design issue or communication problem... with a solution based on class structure... and guidance on how to apply the solution in a variety of contexts.

15.3.1 GRASP: General Responsibility Assignment Software Patterns

- High cohesion is desirable. [Why?] It makes a class easier to comprehend, easier to maintain, and easier to reuse. The class will also be less affected by change in other classes. [Example:] Cohesion is low if a HashTable class contains code to extract fields from a data record. Cohesion is low if the data class computes a hash index. Cohesion is high if each class does part of the process.

- Low coupling is desirable. Which class should be given responsibility for a task? [A] Assign a responsibility so that its placement does not increase coupling. [Why?] High coupling makes a class harder to understand, harder to reuse, and harder to maintain because changes in related classes force local changes.

- Expert. Who should do what? [A] Each class should do for itself actions that involve its data members. Each class should “take care of” itself and handle its own emergencies. [Why] This minimizes coupling.

- Creator. Who should create (allocate) an object? [A] The class that composes, aggregates or contains it. [Why] This minimizes coupling.

Who should delete (deallocate) an object? [A] The class that created it. [Why] To minimize confusion, and because, often, nothing else is possible.

- Don’t Talk to Strangers. That is, don’t “send messages” to objects that are not close to you. Non-strangers are:
  - your own data members.
  - elements of a collection which is one of your own data members.
  - a parameter of the current function.
  - a locally-created object.
  - this (but using this is rarely the right thing to do).

Delegate the operation [Why] This is a generalization of the old rule, don’t use globals. It maximizes the locality of every reference and avoids unnecessary coupling between classes.

[Example] Don’t deal directly with a component of one of your own members. Suppose a hardware store class composes an object of class Inventory, and the Inventory is a fvector of Item pointers, as shown above. The retail store would be talking to a stranger if its sell function called the sell function in the Item class directly, like this: `Inv.find(currentKey)->sell(5);` The preferred design is to work through the intermediate class. That is, Inventory should provide a function such as `sell(key, int)` that can be called by RetailStore, and `Inventory::sell(key, int)` should call `Inventory::find(key)` followed by `Item::sell(int)`. Evaluation: In the first design, a change in the Item class might force a change in both RetailStore and Inventory. Using the second design, only Inventory is affected.

15.4 More Complex Design Patterns

15.4.1 Adapter

Sometimes a toolkit class is not reusable because its interface does not match the domain-specific interface an application requires. [Solution] Define an adapter class that can add, subtract, or override functionality, where necessary. There are two ways to do this; on the left is a class adapter, on the right an object adapter.
15.4.2 Indirection

This pattern is used to decouple the application from the implementation where an implementation depends on
the interface of some low-level device. [Why] To make the application stable, even if the device changes.

15.4.3 Proxy

This pattern is like Indirection, and is used when direct access to a component is not desired or possible. What
to do? [Solution:] Provide a placeholder that represents the inaccessible component to control access to it and
interact with it. The placeholder is a local software class. Give it responsibility for communicating with the
real component. [Special cases:] Device proxy, remote proxy. In Remote Proxy, the system must communicate
with an object in another address space.

15.4.4 Polymorphism

In an application where the abstraction has more than one implementation, define an abstract base class and one
or more subclasses. Let the subclasses implement the abstract operations. [Why] to decouple the implementation
from the abstraction and allow multiple implementations to be introduced, as needed.
15.4.5 Controller

Who should be responsible for handling a system event? A controller class. The controller should coordinate the work that needs to be done and keep track of the state of the interaction. It should delegate all other work to other classes.

Factors such as the number of events to be handled, cohesion and coupling should be used to decide among the three kinds of controllers described below and to decide how many controllers there should be. A controller class represents one of the following choices:

- The overall application, business, or organization (facade controller).
- Something in the real world that is active that might be involved in the task (role controller). [Example:] a menu handler.
- An artificial handler of all system events involved in a given use case (use-case controller). [Example:] A retail system might have separate controllers for BuyItem and ReturnItem.

15.4.6 Bridge

This pattern is a generalization of the Indirection pattern, used when both the application class and the implementation class are (or might be) polymorphic. The bridge decouples the application from the polymorphic implementation, greatly reducing the amount of code that must be written, and making the application much easier to port to different implementation environments. In the diagram below, we show that there might be several kinds of windows, and the application might be implemented on two operating systems. The bridge provides a uniform pattern for doing the job.

15.4.7 Subject-Observer or Publish-Subscribe

Your application program has many classes and many objects of some of those classes. You need to maintain consistency among the objects so that when the state of one changes, its dependents are automatically notified. You do not want to maintain this consistency by using tight coupling among the classes.

[Example:] An OO spreadsheet application contains a data object, several presentation “views” of the data, and some graphs based on the data. These are separate objects. But when the data changes, the other objects should automatically change.

[Solution:] In the following discussion, the SpreadsheetData class is the subject, the views and graphs are the observers. The basic Spreadsheet class composes an observer list and provides an interface for attaching and detaching Observer objects from its list. Observer objects may be added to this list, as needed, and all will be notified when the subject (SpreadsheetData) changes. We derive a concrete subject class (SpreadsheetData) from the Spreadsheet class. It will communicate with the observers through a get_state() function, that returns a copy of its state.

The ObserverList class defines an updating interface for objects that should be notified of changes in a subject. The Observer class provides an abstract public function called update() which will be called by ObserverList whenever updateall() is called. This abstract function must be implemented in each concrete observer class.
15.4. **MORE COMPLEX DESIGN PATTERNS**

When the state of the SpreadsheetData subject changes, it executes its inherited notify() function, which calls **ObserverList::updateall()**, which notifies all of the observers. Each one, in turn, executes its update function, which calls the subject’s get_state function. Changes can then be made locally that reflect the change in the subject’s state.

### 15.4.8 Singleton

Suppose you need exactly one instance of a class, and objects in all parts of the application need a single point of access to that instance. [Solution:] A single object may be made available to all objects of class C by making the singleton a static member of class C. A class method can be defined that returns a reference to the singleton if access is needed outside its defining class.

```
static member StringStore& StringStore::getStore(){
    if (instance==NULL) instance = new StringStore;
    return instance;
}
```

[Example] Suppose there were several parts of a program that could use a StringStore. We might define StringStore as a singleton class. The **StringStore::put** function would be made static and would become a global access point to the class, while maintaining full protection for the class members.

### 15.4.9 Decorator

Suppose you make a basic product, but a customer may order a variety of add-ons, in any order and in any combination. This is the pattern for you! Decorating can give objects individualized properties at run-time.

![Diagram of Singleton]

![Diagram of Decorator]
/* -----------------------------------------------------------------------
* Decorator Demo Decorator.cpp
* Created on: Jun 15, 2014 Modified June 2015
* Adapted from http://en.wikipedia.org/wiki/Decorator_pattern
* */

#include <iostream>
#include <string>
#include <iomanip>
using namespace std;

//========================================================================
// Abstract base class
// Define the common functionality of all varieties of Coffee.
class Coffee {
public:
    virtual ~Coffee(){ cout << "Destroying Coffee " <<\n};
    virtual double getCost() = 0; // Returns the cost of the coffee
    virtual string getIngredients() = 0; // Returns ingredients of product
    virtual ostream& print( ostream& out) {
        return cout <<fixed <<setprecision(2) <<\nCost: $" <<getCost()
        <<"; With: " <<getIngredients();
    }
};

//========================================================================
// The basic product without any extras
// Plain black coffee
class SimpleCoffee : public Coffee{
public:
    virtual ~SimpleCoffee(){ cout << "Destroying SimpleCoffee "; }
    double getCost() { return 2.00; }
    string getIngredients() { return "Coffee"; }
    virtual ostream& print( ostream& out) {
        return Coffee::print( cout );
    }
};
inline ostream& operator<< (ostream& out, SimpleCoffee c){ return c.print(out);}
15.4. MORE COMPLEX DESIGN PATTERNS

---

63 //===============================================
64 // Decoration Caramel that adds caramel flavoring to the drink.
65 class AddCaramel : public Decorator {
66 public:
67    AddCaramel (Coffee* someCoffee) : Decorator(someCoffee) {}
68    double getCost() { return cfp->getCost() + 0.75; }
69    string getIngredients() { return cfp->getIngredients() + ", " + "Caramel"; }
70    
71    inline ostream& operator <<( ostream& out, AddCaramel c){ return c.print(out); }
72 
73 //===============================================
74 //===============================================
75 // Test program
76 int main( void ) {
77    SimpleCoffee s;
78    AddMilk* m = new AddMilk(&s);
79    AddCaramel c1(&s);
80    AddCaramel c2(m);  // Note that you can stack decorators:
81    cout <<s <<*m <<c1 <<c2 <<"\n"
82    
83    Coffee* c = m;
84    delete c;
85 }

This pattern is used to extend the functionality of an object when there are several independent ways of doing so.

- The common functions, shared by the product and the decorations, are declared by an abstract base class.
- They are implemented by the product and each decoration.
- The Decorator class provides a way to link all the decorations to each other and to the product.
- Application: Think of an online purchase where the price might be affected by any combination of the base price, tax, shipping, employee discounts, coupons and special sales. Business logic demands that we support any combination of these factors, in any order.

---

15.4.10 Factory Method used for a Framework

**Factory Method Usage** Using a Factory method decouples the client from the details of object creation.

- The Factory class has one static virtual method, which may or may not have a default definition.
- Derived from it are one or more classes that create objects of different styles, but the same general functionality.
- A client calls the CreateProduct() function in the base factory class when it wants an instance of the product.
- This allows a client to select, at run time, the style of product that is wanted. The client does not need to know at compile time what styles are available or any details about how the products are constructed and initialized.
Use when a Framework knows when a new product is needed, but not how to create that product.

```cpp
void newProduct() {
    p.add( create() );
}
+ appFunc1()
+ add()
+ create(): Product* = 0
+ newProduct(): void
```

- int k
- Product* p[10]

More patterns? The ten patterns presented here are some of the earliest and most useful that have been developed. But many, many more design patterns have been identified and published. A professional working in either C++ or Java would do well to study some of the available literature.