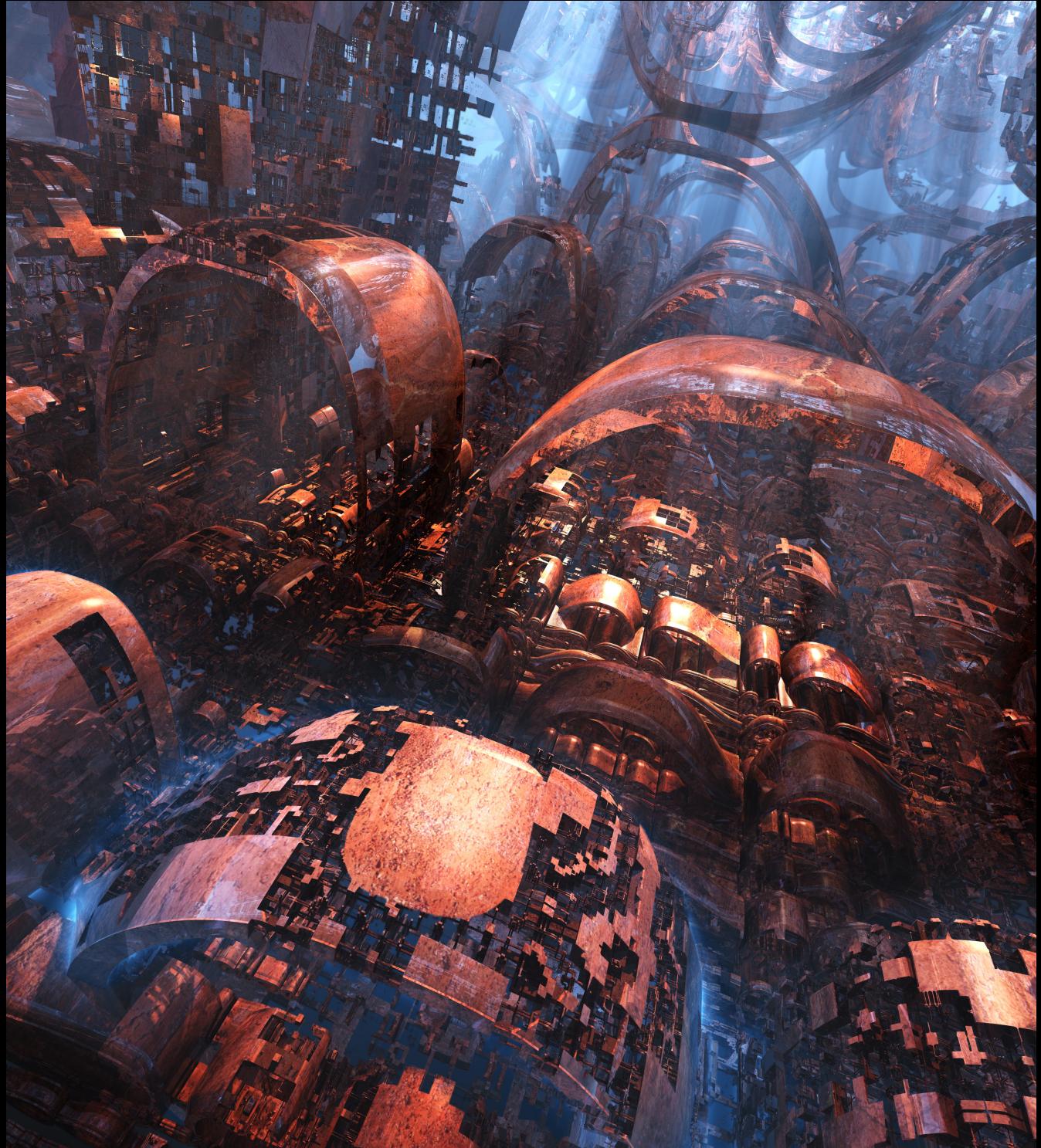


# C++ Data Structures from Scratch, Vol. 1



Robert MacGregor

From databases and operating systems to simulations and graphics, data structures are an integral part of all programming domains. Designed for complete beginners as well as those with prior programming experience, *C++ Data Structures from Scratch, Vol. 1* covers everything from basic C++ language concepts to creating fully functional, STL-style implementations of common data structures and sorting algorithms.

### **Key features:**

- 170+ complete source code files, with detailed line-by-line analysis and diagrams
- 60 sample programs directly illustrating key concepts from each chapter
- Free sample content and online support at the official website, *cppdatastructures.com*

### **Major topics:**

- Step-by-step instructions for setting up an IDE (integrated development environment)
- Thorough coverage of fundamental C++ language concepts:
  - Datatypes, variables, and arithmetic
  - Logic, functions, and program structure
  - Pointers, arrays, and references
  - Object-oriented programming (classes) and operator overloading
  - Template metaprogramming, in the style of the Standard Template Library (STL)
  - Dynamic memory allocation
- A comprehensive, line-by-line guide to implementing:
  - Bubble sort, insertion sort, and quick sort
  - Allocators
  - Dynamic arrays (STL *vector*)
  - Doubly-linked lists (STL *list*)
  - Single-block deques (double-ended queues)
  - Multi-block deques (STL *deque*)
  - Unbalanced binary search trees
  - AVL trees (STL *map*)
  - Bidirectional / random access, const, and reverse iterators for each data structure

### **About the author:**

- Robert MacGregor is the developer of a C++ API for financial market trading systems. He is also a CTA (Commodity Trading Advisor) in the National Futures Association, and a Chartered Market Technician in the CMT Association.

# C++ Data Structures from Scratch, Vol. 1

Robert MacGregor

Copyright 2015 by Robert MacGregor. All rights reserved.

No part of this book may be reproduced or transmitted by any means without the prior written consent of the author.

Although every precaution has been taken to verify the accuracy of the information contained herein, the author and publisher assume no responsibility for errors or omissions. Furthermore, no liability is assumed for any damages resulting from the use of the information or programs contained herein.

Published by South Coast Books

For errata, supplementary material, and contact / purchase information, visit [www.cppdatastructures.com](http://www.cppdatastructures.com)

Cover illustration: *SelfSimilarCircuitry* by Mark J. Brady ([www.markjaybeefractal.com](http://www.markjaybeefractal.com))

ISBN-10: 0-996-2115-1-9

ISBN-13: 978-0-9962115-1-2

1st Printing, March 2015

*Dedicated to Milagros “Mila” Oronce Reyes  
(1935-2014)*



# Table of Contents

## Introduction and Getting Started

### Part 1: Your First Program

1.1: Standard Output, Variables, and Datatypes	1
--	---

### Part 2: Arithmetic Operations and User Input

2.1: Basic Arithmetic	5
2.2: Standard Input	9
2.3: The Increment Operator	11
2.4: The Decrement Operator	15
2.5: Compound Assignment	17

### Part 3: Control Flow

3.1: Relational Operators and Conditional Statements	21
3.2: Logical Operators	27
3.3: Loops	31
3.4: Boolean Variables	41
3.5: Putting It All Together	43

### Part 4: Functions and Scope

4.1: Functions	49
4.2: Namespaces	53
4.3: Scope	59
4.4: Function Overloading	63
4.5: Header Files and Inline Functions	65

### Part 5: Pointers, Arrays, and References

5.1: Pointers	67
5.2: Pass by Reference	71
5.3: Arrays	75
5.4: Pointer Arithmetic	85
5.5: References	99
5.6: Const Correctness	105

## Part 6: Classes and Operator Overloading

6.1: Classes	117
6.2: Operator Overloading	125

## Part 7: Templates and Function Objects

7.1: Function templates	137
7.2: Class templates and Function Objects	145
7.3: Introducing the Array Class Template	155

## Part 8: Dynamic Memory Allocation

8.1: Tracing Object Lifetime	173
8.2: Introducing the Allocator Class Template	183

## Part 9: Dynamic Arrays

9.1: Introducing the Vector Class Template	189
9.2: Reserve, Pop Back, Copy, and Assignment	199
9.3: Insert and Erase	207

## Part 10: Linked Lists

10.1: Introducing the List Class Template	213
10.2: Pop Front, Pop Back, and Clear	227
10.3: Implementing an Iterator	233
10.4: Implementing a Const Iterator	241
10.5: Copy and Assignment	247
10.6: Insert and Erase	253

## Part 11: Reverse Iterators

11.1: Introducing the ReverseIter Class Template	261
11.2: Reverse Iterators for the Array and Vector Classes	269

## Part 12: Single-Block Double-Ended Queues

12.1: Introducing the Ring Class Template	279
12.2: Push Front and Array Subscript	289
12.3: Pop Front, Pop Back, and Clear	295
12.4: Implementing the Iterators	301
12.5: Copy, Assignment, Insert, and Erase	307

## Part 13: Multi-Block Double-Ended Queues

13.1: Introducing the MultiRing Class Template	315
13.2: Push Front and Array Subscript	323
13.3: Iterators, Pop Front, Pop Back, and Clear	327
13.4: Copy, Assignment, Insert, and Erase	331

## Part 14: Unbalanced Binary Trees

14.1: Key-Mapped Pairs	333
14.2: Nodes	337
14.3: Introducing the DemoBinaryTree Class	349
14.4: Recursive In-Order Traversal	355
14.5: Iterative In-Order Traversal	363
14.6: Implementing the Iterators	367
14.7: Retrieving Elements by Key Value	371
14.8: Introducing the BinaryTree Class Template	377
14.9: Insert	379
14.10: Erase	395
14.11: Clear, Copy, and Assignment	417

## Part 15: Balanced Binary Trees

15.1: Rotating Nodes	427
15.2: Introducing the AvlTree Class Template	447
15.3: Insert	451
15.4: Erase	471

## Part 16: Time Complexity

16.1: Iterator Categories	495
16.2: Big O Notation	501

## Index

503



# Introduction and Getting Started

## *Chapter outline*

- *What are data structures, and why is it necessary to know how they work?*
- *The goal of this book*
- *Obtaining the accompanying source code*
- *Setting up an IDE (integrated development environment)*
- *Recommended study approach*
- *A brief overview of the major topics*

The manipulation of data is a fundamental task performed by nearly all types of software. An address book application, for example, alphabetically sorts contact names, while a web browser maintains a chronological history of visited sites.

A “data structure” is a software component that collects and organizes data. Although many types of data structures can perform the same tasks, they do so with varying degrees of efficiency. The inner workings of each type reflect a set of performance trade-offs intrinsic to computer hardware. A thorough understanding of these trade-offs is therefore critical to designing effective software.

The goals of this book are twofold. Parts 1-7 teach fundamental C++ language concepts, culminating in the study of basic sorting algorithms. Parts 8-16 then apply these concepts by building a fully functional set of commonly used data structures, in the style of the STL (Standard Template Library). By mastering the material herein, readers will not only acquire a strong foundation in data structures and C++, but also a solid conceptual framework for programming in a wide variety of languages and domains.

This book is designed for complete beginners as well as those with prior programming experience. To get started, follow the instructions below to obtain the accompanying source code and set up an IDE (integrated development environment).

## Obtaining the accompanying source code:

Each chapter is a line-by-line walk-through of a small program, designed to illustrate key concepts as simply and directly as possible. To obtain the accompanying source code (programs), visit the official website, [www.cppdatastructures.com](http://www.cppdatastructures.com).

The relevant source files and / or folders are listed at the beginning of each chapter. The root folder (*dss*) is omitted. If a folder is listed without specific filenames, it indicates that the chapter uses all of the files in that folder. The listing for Chapter 5.2, for example,

## *Source folders*

*passByReference  
swapInts*

indicates that Chapter 5.2 uses:

- All of the files in the folder *dss/passByReference*
- All of the files in the folder *dss/swapInts*

Similarly, the listing for Chapter 7.3,

*Source files and folders*

*Array/I*  
*Array/common/memberFunctions\_I.h*  
*quickSort*

indicates that Chapter 7.3 uses:

- All of the files in the folder *dss/Array/I*
- The file *memberFunctions\_I.h* in the folder *dss/Array/common*, but not the other files in *dss/Array/common*
- All of the files in the folder *dss/quickSort*

#### Setting up an IDE (integrated development environment):

Creating a C++ program requires three basic tools, the first of which is a word processor. The word processor is used to write “source code,” a set of plain text files with the extension *.h* or *.cpp*. The source code specifies the instructions to be carried out by the computer when the program is run.

Computers, however, do not natively understand C++ in plain text form; the source code (*.h* / *.cpp* files, written by a human) must first be translated into object code (*.obj* files), which is what the computer hardware natively understands. This translation process, called “compilation,” is performed by a separate program called a “compiler.”

A third program called a “linker” then combines the various object (*.obj*) files into a single executable (*.exe*) file, which can be run from the command line or desktop.

An IDE, short for “integrated development environment,” is a program that combines the functionality of a word processor, compiler, and linker. To install and set up an IDE, visit [www.cppdatastructures.com](http://www.cppdatastructures.com).

#### Recommended study approach:

- At the beginning of each chapter, compile the included source code and run the program.
- Read the chapter, following along with the included source code.
- Read the chapter again, recreating the source code from scratch.
- Compile the recreated source code and run the program, verifying the output.

#### A brief overview of this book:

Parts 1-3 introduce basic programming concepts (variables, arithmetic, and logic).

Part 4 introduces the fundamental tools of program organization (functions, namespaces, and header files).

Part 5 introduces the concept of indirection (pointers, arrays, and references) and implements the bubble sort and insertion sort algorithms.

Part 6 introduces object-oriented programming (classes) and operator overloading by implementing a custom datatype.

Part 7 introduces template metaprogramming and function objects. This section also implements the quick sort algorithm, introducing the concept of recursion.

Part 8 introduces dynamic memory allocation and traceable objects, completing the foundation for Parts 9-15.

Parts 9-15 are comprehensive, line-by-line guides to implementing:

- Dynamic arrays (*STL vector*)
- Doubly-linked lists (*STL list*)
- Single-block deques (double-ended queues)
- Multi-block deques (*STL deque*)
- Unbalanced binary search trees
- AVL trees (*STL map*)
- Bidirectional / random access, const, and reverse iterators for each data structure

Part 16 discusses iterator categories and time complexity, summarizing the relative efficiency of the data structures from Parts 9-15.



# C++ Data Structures from Scratch, Vol. 1



# Part 1: Your First Program

## 1.1: Standard Output, Variables, and Datatypes

*Source folder: hello*

*Chapter outline*

- *Displaying messages on the screen*
- *Using variables to store and modify values*
- *Basic datatypes: integers, real numbers, booleans, and character strings*

**Before proceeding, please visit the official website ([www.cppdatastructures.com](http://www.cppdatastructures.com)) to obtain the accompanying source code and set up an IDE (integrated development environment).**

Line 4 of main.cpp,

```
int main()
```

specifies a "function" called main. A function, also called a "subroutine" or "method," is a set of program statements (instructions) to be executed.

A function's statements are collectively referred to as the "function body," which is enclosed in braces. The opening and closing braces of main are located in lines 5 and 36, respectively. A function can be "called," or executed, any number of times.

The type of output value generated by a function is called its "return type." The return type of main is int (line 4), which denotes the integer datatype, more commonly known as "type int." Line 8,

```
cout << "Hello" << endl;
```

generates the output

```
Hello
```

cout, pronounced "c-out," is called the "standard output stream." The << is called the "stream insertion operator." endl, pronounced "end-l" for "newline," represents a carriage return. The statement can be read as "Insert the character string 'Hello' into the standard output stream, followed by a carriage return (new line)." The semicolon indicates the end of the statement. Lines 10-14,

```
int i = 0;
double d = 3.14;
bool b = true;
char c = '?';
string s = "Independence Day";
```

declare and initialize the variables

i, of type int (integer), initialized to 0  
 d, of type double (floating point, or real number), initialized to 3.14  
 b, of type bool (boolean, or true/false value), initialized to true  
 c, of type char (single character), initialized to "?"  
 s, of type string (character string), initialized to "Independence Day"

Line 16,

```
cout << "i is " << i << endl;
```

prints the character string "i is ", followed by the value of i, followed by a new line, generating the output

```
i is 0
```

Similarly, lines 17-20,

```
cout << "d is " << d << endl;
cout << "b is " << b << endl;
cout << "c is " << c << endl;
cout << "s is " << s << endl << endl;
```

generate the output

```
d is 3.14
b is 1
c is ?
s is Independence Day
```

Note that for boolean values, "true" and "false" are printed as the integer values 1 and 0, respectively.  
 Lines 22-26,

```
i = 9;
d = 2.71;
b = false;
c = '!';
s = "July 4th, 1776";
```

assign new values to each variable, and lines 28-33,

```
cout << "i is " << i << endl;
cout << "d is " << d << endl;
cout << "b is " << b << endl;
cout << "c is " << c << endl;
cout << "s is " << s << endl;
cout << "Goodbye\n";
```

generate the output

```
i is 9
d is 2.71
b is 0
c is !
s is July 4th, 1776
Goodbye
```

In line 33, the `\n` denotes a carriage return, making the statement

```
cout << "Goodbye\n";
```

equivalent to

```
cout << "Goodbye" << endl;
```

Note that `\n` is a special character that must be part of a character string (i.e. within quotation marks):

```
cout << "Goodbye\n";
```

is valid, but

```
cout << i\n;
cout << d\n;
cout << b\n;
cout << c\n;
cout << s\n;
```

are not. Also note that the type `char` uses single-quotation marks (lines 13, 25), while the type `string` uses double-quotation marks (lines 14, 26).

Recall that the return type of `main` is `int`. Line 35,

```
return 0;
```

terminates the function and returns a value of 0 to the system. In line 38,

```
// single-line comment
```

all text following the double forward slashes (`//`) to the end of the line is ignored by the compiler. In a multiple-line comment, the compiler ignores all text from the `/*` to the `*/`, as in (lines 40-43)

```
/*
    multiple-line
    comment
*/
```

`cout`, `endl`, and the `string` type are provided by the C++ Standard Library. In order to use Standard Library components, the appropriate "header files" must be included. Lines 1-2,

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

are called "include directives." Before compilation, a separate program called the "preprocessor" replaces each include directive with the code from the specified file. All of the code in the `<iostream>` header is inserted at line 1, followed by the code from the `<string>` header. `<iostream>` provides support for `cout` and `endl`, while `<string>` provides support for the `string` type.

The statement in line 6,

```
using namespace std;
```

is called a "using directive," which will be discussed in a later chapter.

Variables can be declared without being set to specific values, as in

```
int i;
double d;
bool b;
char c;      // The values of i, d, b, and c are undefined

i = 7;
d = 1.618;
b = true;
c = '!';    // The values of i, d, b, and c are now 7, 1.618, true, and '!'
```

Note, however, that variables of type `string` are implicitly (automatically) initialized to a value of "" (empty string):

```
string s;                  // The value of s is ""

s = "Independence Hall"; // The value of s is now "Independence Hall"
```

# Part 2: Arithmetic Operations and User Input

## 2.1: Basic Arithmetic

*Source folder: basicArithmetic*

*Chapter outline*

- Performing addition, subtraction, multiplication, and division
- Calculating a remainder using modulus division

Given (line 7)

```
int a = 1;
```

the expression

```
a + 2
```

returns (evaluates to) 3. The + is called the "addition operator," and the statement (line 8)

```
int b = a + 2;
```

initializes b to 3. Lines 10-11,

```
cout << "a = " << a << endl;
cout << "b = " << b << endl << endl;
```

generate the output

```
a = 1
b = 3
```

The expression

```
7 - b
```

returns (evaluates to) 4. The - is called the "subtraction operator," and the statement (line 13)

```
a = 7 - b;
```

sets a to 4. Lines 15-16,

```
cout << "a = " << a << endl;
cout << "b = " << b << endl << endl;
```

generate the output

```
a = 4
b = 3
```

The expression

```
a * 3
```

returns (evaluates to) 12. The \* is called the "multiplication operator," and the statement (line 18)

```
b = a * 3;
```

sets b to 12. Lines 20-21,

```
cout << "a = " << a << endl;
cout << "b = " << b << endl << endl;
```

generate the output

```
a = 4
b = 12
```

The expression

```
b / 4
```

returns (evaluates to) 3. The / is called the "division operator," and the statement (line 23)

```
a = b / 4;
```

sets a to 3. Lines 25-26,

```
cout << "a = " << a << endl;
cout << "b = " << b << endl << endl;
```

generate the output

```
a = 3
b = 12
```

The expression

```
a % 2
```

returns (evaluates to) the remainder of (a / 2), which is 1. The % is called the "modulo operator," and the statement (line 28)

```
b = a % 2;
```

sets b to 1. Lines 30-31,

```
cout << "a = " << a << endl;
cout << "b = " << b << endl << endl;
```

generate the output

```
a = 3
b = 1
```



## 2.2: Standard Input

*Source folder: standardInput*

*Chapter outline*

- Getting data from the user and processing that data

Lines 8-10,

```
string firstName;
string lastName;
int age;
```

declare the string variables firstName and lastName, followed by the int variable age. Line 12,

```
cout << "Please enter your first name: ";
```

generates the output

```
Please enter your first name:
```

Line 13,

```
cin >> firstName;
```

prompts the user to enter a string value, then writes that value to the variable firstName. cin, pronounced "c-in," is called the "standard input stream." Like cout, cin is provided via the <iostream> header (line 1). The >> is called the "stream extraction operator." Line 15,

```
cout << "Please enter your last name and age (separated by whitespace): ";
```

generates the output

```
Please enter your last name and age (separated by whitespace):
```

Line 16,

```
cin >> lastName >> age;
```

prompts the user to enter a string value followed by an integer value (separated by whitespace), then writes the respective values to the variables lastName and age. Lines 18-19,

```
cout << "\nHello, " << firstName << " " << lastName << ".\n";
cout << "In 5 years, you will be " << age + 5 << " years old.\n";
```

generate the output

```
Hello, <firstName> <lastName>.  
In 5 years, you will be <age + 5> years old.
```

Note that the expression (line 18)

```
cout << "\nHello, "
```

is equivalent to

```
cout << endl << "Hello, "
```

A sample run of the program is

```
Please enter your first name: Alex  
Please enter your last name and age (separated by whitespace): Turner 37
```

```
Hello, Alex Turner.  
In 5 years, you will be 42 years old.
```

## 2.3: The Increment Operator

*Source folder: incrementOperator*

*Chapter outline*

- The difference between l-values and r-values
- The difference between prefix and postfix increment

An “l-value” (short for “left-value”) is a named variable, while an “r-value” (short for “right-value”) is an unnamed value. In lines 7-11,

```
int a;
int b;

a = 7;
b = 3;
```

for example, the named integer variables a and b are l-values, while the unnamed values 7 and 3 are r-values.

An l-value (named variable) can be assigned to:

```
b = 5;      // Set the value of b to 5 (valid)
b = a;      // Set the value of b to the value of a (valid)
```

An r-value (unnamed value), however, cannot be assigned to:

```
5 = b;      // Set 5 to the value of b (invalid)
```

An l-value, in other words, can appear on either the left or right-hand side of an assignment expression, while an r-value can only appear on the right-hand side (as in the above statements). Similarly, given

```
string x = "Debussy";      // l-values x and y
string y = "Ravel";        // r-values "Debussy" and "Ravel"
```

the statements

```
y = "Stravinsky";          // Set the value of y to "Stravinsky"
y = x;                      // Set the value of y to the value of x
```

are valid, while the statement

```
"Stravinsky" = y;           // Set "Stravinsky" to the value of y
```

is not.

The distinction between l-values and r-values is critical to understanding the semantics of the

“increment operator” (`++`). The statement (line 16)

```
++a;           // Increment the value of a (from 7 to 8)
```

for example, is shorthand for

```
a = a + 1;    // Set the value of a to (a + 1) (the r-value 8)
```

When the `++` appears on the left-hand side of a variable (as in line 16), it is called the “prefix increment operator.” A prefix increment expression, in addition to incrementing the value of a variable, returns (evaluates to) an l-value. The expression `++a`, for example, increments the value of `a`, then returns (evaluates to) the l-value (named variable) `a`. A prefix increment expression can therefore appear on either the left or right-hand side of an assignment expression. The statement (line 19)

```
b = ++a;      // Increment the value of a (from 8 to 9),
               // then set the value of b to the value of a (9)
```

for example, is equivalent to

```
++a;
b = a;
```

Similarly, a statement such as

```
++b = 2;       // Increment the value of b (from 9 to 10)
               // then set the value of b to 2
```

though nonsensical, is valid and equivalent to:

```
++b;
b = 2;
```

When the `++` appears on the right-hand side of a variable, it is called the “postfix increment operator.” The statement (line 23)

```
a++;          // Increment the value of a (from 9 to 10)
```

for example, is equivalent to

```
a = a + 1;
```

Note, however, that while a prefix increment expression returns an l-value (named variable), a postfix increment expression returns an r-value (unnamed value). A postfix increment expression can therefore only appear on the right-hand side of an assignment expression. Also note that the r-value returned by a postfix increment expression is equal to the variable's original value (before it was incremented). The statement (line 26)

```
b = a++;      // Set the value of b to the value of a (10),
               // then increment the value of a (from 10 to 11)
```

is therefore equivalent to

```
b = 9;  
a++;
```

Because they can be more error-prone and difficult to understand, statements such as

```
b = ++a;  
b = a++;
```

are largely avoided throughout this book in favor of their long-form counterparts,

```
++a;  
b = a;  
  
b = a;  
a++; // or ++a;
```

The program in this chapter generates the output

```
a = 7      // Initial values of a and b  
b = 3  
  
a = 8      // Result of ++a;  
  
a = 9      // Result of b = ++a;  
b = 9  
  
a = 10     // Result of a++;  
  
a = 11     // Result of b = a++;  
b = 10
```



## 2.4: The Decrement Operator

*Source folder: decrementOperator*

*Chapter outline*

- The difference between prefix and postfix decrement

Given (lines 7-8)

```
int a = 9;
int b = 5;
```

the statement (line 13)

```
--a;           // Decrement the value of a (from 9 to 8)
```

is shorthand for

```
a = a - 1;
```

The -- is called the "decrement operator." When it appears on the left-hand side of a variable (as in line 13), it is called the "prefix decrement operator." Like a prefix increment expression, a prefix decrement expression returns an l-value. The statement (line 16)

```
b = --a;      // Decrement the value of a (from 8 to 7),
               // then set the value of b to the value of a (7)
```

for example, is equivalent to

```
--a;
b = a;
```

When the -- appears on the right-hand side of a variable, it is called the "postfix decrement operator." The statement (line 20)

```
a--;          // Decrement the value of a (from 7 to 6)
```

for example, is shorthand for

```
a = a - 1;
```

Like a postfix increment expression, a postfix decrement expression also returns an r-value. Note, however, that this r-value is equal to the original value of the variable (before it was decremented). The statement (line 23)

```
b = a--;      // Set the value of b the value of a (6),
               // then decrement the value of a (from 6 to 5)
```

is therefore equivalent to

```
b = a;  
a--;
```

The program in this chapter generates the output

```
a = 9      // Initial values of a and b  
b = 5  
  
a = 8      // Result of --a;  
  
a = 7      // Result of b = --a;  
b = 7  
  
a = 6      // Result of a--;  
  
a = 5      // Result of b = a--;  
b = 6
```

## 2.5: Compound Assignment

*Source folder: compoundAssignment*

*Chapter outline*

- *Modifying variables using the compound assignment operators (addition, subtraction, multiplication, division, and modulo)*

The `+=`, `-=`, `*=`, `/=`, and `%=` operators are collectively known as the "compound assignment operators." Given (lines 7-8)

```
int a = 3;
int b = 5;
```

for example, the statement (line 12)

```
a += 8;           // Increment the value of a by 8 (from 3 to 11)
```

is shorthand for

```
a = a + 8;
```

The `+=` is called the "addition assignment operator." An addition assignment expression returns an l-value, so it can appear on either the left or right-hand side of an assignment expression. The statement (line 15)

```
b = (a += 8);    // Increment the value of a by 8 (from 11 to 19),
                  // then set the value of b to the value of a (19)
```

for example, is equivalent to

```
a += 8;
b = a;
```

The statement

```
(a += 8) = b;    // Increment the value of a by 8 (from 19 to 27),
                  // then set the value of a to the value of b (19)
```

though nonsensical, is valid and equivalent to

```
a += 8;
a = b;
```

The statement (line 18)

```
a -= 4;           // Decrement the value of a by 4 (from 19 to 15)
```

is shorthand for

```
a = a - 4;
```

The `=` is called the "subtraction assignment operator." A subtraction assignment expression returns an l-value. The statement (line 21)

```
b = (a -= 4); // Decrement the value of a by 4 (from 15 to 11),  
// then set b to the value of a (11)
```

for example, is equivalent to

```
a -= 4;  
b = a;
```

The statement (line 24)

```
a *= 3; // Multiply the value of a by 3 (from 11 to 33)
```

is shorthand for

```
a = a * 3;
```

The `*=` is called the "multiplication assignment operator." A multiplication assignment expression returns an l-value. The statement (line 27)

```
b = (a *= 3); // Multiply the value of a by 3 (from 33 to 99),  
// then set b to the value of a (99)
```

for example, is equivalent to

```
a *= 3;  
b = a;
```

The statement (line 30)

```
a /= 3; // Divide the value of a by 3 (from 99 to 33)
```

is equivalent to

```
a = a / 3;
```

The `/=` is called the "division assignment operator." A division assignment expression returns an l-value. The statement (line 33),

```
b = (a /= 3); // Divide the value of a by 3 (from 33 to 11),  
// then set b to the value of a (11)
```

for example, is equivalent to

```
a /= 3;
b = a;
```

The statement (line 36)

```
a %= 4;           // Calculate (a % 4) (the remainder of a / 4, which is 3),
                  // then set the value of a to the result (3)
```

is shorthand for

```
a = a % 4;
```

The `%=` is called the "modulo assignment operator." A modulo assignment expression returns an l-value. The statement (line 39)

```
b = (a %= 2);    // Calculate (a % 2) (the remainder of a / 2, which is 1),
                  // set the value of a to the result (1),
                  // then set the value of b to the value of a (1)
```

for example, is equivalent to

```
a %= 2;
b = a;
```

Because they can be more error-prone, statements such as

```
b = (a += 8);
```

are avoided throughout this book in favor of their long-form counterparts,

```
a += 8;
b = a;
```

The program in this chapter generates the output

```
a = 3, b = 5
a = 11          // Result of a += 8;
a = 19, b = 19  // Result of b = (a += 8);
a = 15          // Result of a -= 4;
a = 11, b = 11  // Result of b = (a -= 4);
a = 33          // Result of a *= 3;
a = 99, b = 99  // Result of b = (a *= 3);
a = 33          // Result of a /= 3;
a = 11, b = 11  // Result of a = (a /= 3);
a = 3           // Result of a %= 2;
a = 1, b = 1     // Result of b = (a %= 2);
```



# Part 3: Control Flow

## 3.1: Relational Operators and Conditional Statements

*Source folder: conditionalStatements*

*Chapter outline*

- Comparing values using the relational operators
- Controlling program behavior using the “if” and “else” keywords

Lines 7-10,

```
int x;

cout << "Enter a number (integer): ";
cin >> x;
```

prompt the user to enter an integer value, which is stored in the variable x.

An "if statement" uses the syntax

```
if (condition)
{
    body
}
```

which can be read as "If the condition is true, then execute the body." The body can consist of any number of statements. The if statement in lines 12-15,

```
if (x == 0)
{
    cout << "You entered 0\n";
}
```

for example, can be read as "If x is equal to 0, then print the message 'You entered 0'." The condition is the boolean (true / false) expression (line 12)

```
x == 0
```

which returns (evaluates to) true if and only if the value of x is equal to 0. The “equal to” operator (==) is used to test whether two values are equal. If the user enters 0, the condition (line 12) will return true, causing the body (line 14) to be executed. After execution of the body, the program will proceed to line 17. If, on the other hand, the user enters some value other than 0, the condition (line 12) will return false, causing the program to skip the body (line 14) and proceed directly to line 17.

Because the body (line 14) only consists of a single statement, the enclosing braces can be omitted, as in

```
if (x == 0)
    cout << "You entered 0\n";
```

An "if...else" statement uses the syntax

```
if (condition)
{
    body
}
else
{
    alternateBody
}
```

which can be read as "If the condition is true, then execute the body; otherwise, execute the alternateBody." The if...else statement in lines 17-20,

```
if (x % 2 == 0)
    cout << "You entered an even number\n";
else
    cout << "You entered an odd number\n";
```

for example, can be read as "If the remainder of  $x / 2$  is equal to 0, then print the message 'You entered an even number'; otherwise, print the message 'You entered an odd number'." The condition is the boolean expression (line 17)

```
x % 2 == 0
```

which returns true if and only if  $x$  is divisible by 2 (i.e. if the remainder of  $x / 2$  is equal to 0). If the user enters an even number, the condition (line 17) will return true, causing the body (line 18) to be executed. If, on the other hand, the user enters an odd number, the condition (line 17) will return false, causing the alternate body (line 20) to be executed. In both cases, the program will then proceed to line 22.

An "if..else if" statement uses the syntax

```

if (condition1)
{
    body1
}
else if (condition2)
{
    body2
}
else
{
    body3
}

```

which can be read as “If condition1 is true, then execute body1. Otherwise, if condition2 is true, then execute body2. Otherwise (if neither condition1 nor condition2 is true), execute body3.” The “if...else if” statement in lines 22-27,

```

if (x > 0)
    cout << "You entered a positive number\n";
else if (x < 0)
    cout << "You entered a negative number\n";
else
    cout << "The number you entered is neither positive nor negative\n";

```

for example, can be read as “If x is greater than 0, then print the message ‘You entered a positive number’. Otherwise, if x is less than 0, then print the message ‘You entered a negative number’. Otherwise (if x is neither greater than 0 nor less than 0), print the message ‘The number you entered is neither positive nor negative’.”

If the user enters a positive number, the first condition (line 22) will return true, causing the first body (line 23) to be executed.

If the user enters a negative number, the first condition (line 22) will return false, causing the second condition (line 24) to be evaluated. The second condition will return true, causing the second body (line 25) to be executed.

If the user enters 0, the first condition (line 22) will return false, causing the second condition (line 24) to be evaluated. The second condition will also return false, causing the third body (line 27) to be executed.

An “if...else if” statement can contain multiple “else if” clauses. Additionally, it need not contain an “else” clause at the end. In the statement

```

if (condition1)
{
    body1;
}
else if (condition2)
{
    body2;
}
else if (condition3)
{
    body3;
}

```

for example, body 1, 2, or 3 will be executed, but only if condition 1, 2, or 3 is true. If none of them are true, then no body will be executed.

Conditional statements can be “nested” in order to create more complex program logic. A nested conditional statement is one that exists within the body of another conditional statement, as in lines 29-35:

```

if (x > 15)
{
    if (x < 20)
        cout << "You entered a number greater than 15 but less than 20\n";
    else
        cout << "You entered a number greater than both 15 and 19\n";
}

```

The if...else statement (lines 31-34)

```

if (x < 20)
    cout << "You entered a number greater than 15 but less than 20\n";
else
    cout << "You entered a number greater than both 15 and 19\n";

```

is only evaluated if the outer condition (line 29),

x > 15

returns true. In other words, if the user enters a number greater than 15, the condition (line 31)

x < 20

will be evaluated, then either line 32 or 34 will be executed, depending on the result. If the user enters a number less than 15, however, the program will skip lines 31-34 and proceed directly to line 37.

Some sample runs of the program in this chapter are

```

Enter a number (integer): 0
You entered 0

```

```
You entered an even number  
The number you entered is neither positive nor negative
```

```
Enter a number (integer): 17  
You entered an odd number  
You entered a positive number  
You entered a number greater than 15 but less than 20
```

```
Enter a number (integer): -8  
You entered an even number  
You entered a negative number
```

```
Enter a number (integer): 25  
You entered an odd number  
You entered a positive number  
You entered a number greater than both 15 and 19
```

The following table summarizes the relational (comparison) operators:

Symbol	Operator name
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to



## 3.2: Logical Operators

*Source folder: logicalOperators*

*Chapter outline*

- Using the “and”, “or”, and “not” operators to form compound boolean expressions

As in the previous chapter's program, lines 7-10,

```
int x;

cout << "Enter a number (integer): ";
cin >> x;
```

prompt the user to enter an integer value, storing it in the variable x.

Individual boolean expressions, such as

```
x > 0      // Returns true if the value of x is greater than 0;
            // otherwise, returns false

x < 10     // Returns true if the value of x is less than 10;
            // otherwise, returns false
```

can be combined to form a “compound boolean expression” as in line 12,

```
x > 0 && x < 10    // Returns true if and only if
                    // the value of x is greater than 0 AND less than 10;
                    // otherwise, returns false
```

The `&&` is called the “and operator.” If the users enter a number that is greater than 0 and less than 10, the condition (line 12) will return true, causing line 13,

```
cout << "Your number is positive and less than 10\n";
```

to be executed. If, however, the user enters a number that is either less than 0 or greater than 9, the condition will return false, causing line 15,

```
cout << "Your number is either nonpositive or greater than 9\n";
```

to be executed. In both cases, control will resume at (i.e. the program will proceed to) line 17, which contains another compound boolean expression:

```
x % 2 == 1 || x < 0    // Returns true if
                        // the remainder of x/2 is equal to 1 (x is odd),
                        // OR if x is less than 0
                        // Only returns false if both conditions are false
```

The `||` is called the “or operator.” If the user enters a number that is either odd, less than 0, or both, the condition (line 17) will return true and control (i.e. the program) will proceed to line 18,

```
cout << "Your number is either odd, negative, or both\n";
```

If, however, the user enters a number that is neither odd nor less than 0, the condition (line 17) will return false and control will proceed to line 20,

```
cout << "Your number is neither odd nor negative\n";
```

Parentheses can be used to form compound boolean expressions of any length or complexity, as in line 22:

```
(x < -5 && x > -10) || (x > 5 && x < 10)      // Returns true if
                                                       // x is between -5 and -10,
                                                       // OR if x is between 5 and 10
```

The `||` indicates that this expression returns true if the first sub-expression,

```
x < -5 && x > -10      // x is less than -5 AND greater than -10
```

returns true, or if the second sub-expression,

```
x > 5 && x < 10      // x is greater than 5 AND less than 10
```

returns true. If both sub-expressions return false (i.e. if the user enters a number that is neither between -5 and -10, nor between 5 and 10), then the expression returns false. In line 27,

```
!(x < 0)      // Returns true if and only if
                // x is NOT less than 0 (x is NOT negative)
```

the `!` is called the “not operator,” which negates any boolean expression to which it is applied. In other words, if the expression

```
x < 0      // Is x negative?
```

returns false (i.e. because x is not negative), then the expression

```
!(x < 0)      // Is x not negative?
```

must return true. Similarly, if the expression

```
x < 0      // Is x negative?
```

returns true (i.e. because x is negative), then the expression

```
!(x < 0)      // Is x not negative?
```

must return false.

Some sample runs of the program in this chapter are

```
Enter a number (integer): 3
Your number is positive and less than 10
Your number is either odd, negative, or both
Your number is neither between - 5 and - 10, nor between 5 and 10
Your number is not negative
```

```
Enter a number (integer): -6
Your number is either nonpositive or greater than 9
Your number is either odd, negative, or both
Your number is either between - 5 and - 10, or between 5 and 10
Your number is negative
```

```
Enter a number (integer): -8
Your number is either nonpositive or greater than 9
Your number is either odd, negative, or both
Your number is either between - 5 and - 10, or between 5 and 10
Your number is negative
```

```
Enter a number (integer): 18
Your number is either nonpositive or greater than 9
Your number is neither odd nor negative
Your number is neither between - 5 and - 10, nor between 5 and 10
Your number is not negative
```

```
Enter a number (integer): 0
Your number is either nonpositive or greater than 9
Your number is neither odd nor negative
Your number is neither between - 5 and - 10, nor between 5 and 10
Your number is not negative
```

```
Enter a number (integer): 9
Your number is positive and less than 10
Your number is either odd, negative, or both
Your number is either between - 5 and - 10, or between 5 and 10
Your number is not negative
```



### 3.3: Loops

*Source folder: loops*

*Chapter outline*

- “*for*” loops
- “*while*” loops

A “loop” is a body of statements that is repeatedly executed as long as a particular condition holds true. The “*for* loop” (lines 7-8)

```
for (int i = 0; i != 5; ++i)
    cout << i << endl;
```

for example, prints the values 0 through 4, inclusive. The loop can be read as “Initialize an int *i* to 0. As long as the condition (*i* != 5) returns true, execute the statement (*cout* << *i* << *endl*); then (++*i*). When the condition (*i* != 5) returns false, terminate the loop.”

The syntax of a *for* loop is

```
for (initializer; condition; postBody)
{
    body;
}
```

The individual components are:

The initializer (*int i = 0;*)

Statement to be executed once and only once, at the beginning of the loop.

The condition (*i != 5*)

Boolean expression that is evaluated before each execution, or “iteration,” of the body. If the expression returns true, the body is executed; if it returns false, the loop terminates.

The body (*cout << i << endl;*)

Statement(s) to be executed if the condition returns true.

The postBody (++*i*)

Statement that is executed after each iteration (execution of the body).

Note that the enclosing braces can be omitted if the body consists of a single statement (as in this example). The loop performs a total of 5 iterations:

---

```
initializer:
    int i = 0;           // i is 0
```

```
condition:  
i != 5 // True  
  
body (iteration 1):  
cout << i << endl; // Print the value of i (0) and a newline  
  
postBody:  
++i; // i is 1
```

---

```
condition:  
i != 5 // True  
  
body (iteration 2):  
cout << i << endl; // Print the value of i (1) and a newline  
  
postBody:  
++i; // i is 2
```

---

```
condition:  
i != 5 // True  
  
body (iteration 3):  
cout << i << endl; // Print the value of i (2) and a newline  
  
postBody:  
++i; // i is 3
```

---

```
condition:  
i != 5 // True  
  
body (iteration 4):  
cout << i << endl; // Print the value of i (3) and a newline  
  
postBody:  
++i; // i is 4
```

---

```
condition:  
i != 5 // True  
  
body (iteration 5):  
cout << i << endl; // Print the value of i (4) and a newline  
  
postBody:  
++i; // i is 5
```

---

```
condition:  
i != 5 // False
```

Terminate loop

Also note that any variable declared within a loop is “local” to the loop, i.e. it cannot be referred to outside the loop, as in

```
for (int i = 0; i != 5; ++i)      // i is declared in the loop's
    cout << i << endl;           // initializer, so it can only be
                                // used within the loop

i = 10;                           // Error: Can no longer use i
cout << i << endl;
```

If a variable will be needed both inside and outside of a given loop, it must be declared outside the loop, as in

```
int i;                          // i is declared outside the loop

for (i = 0; i != 5; ++i)
    cout << i << endl;

i = 10;                         // Ok
cout << i << endl;
```

Line 12,

```
int i = 0;
```

declares and initializes a new integer i, and the “while loop” (lines 14-18),

```
while (i != 5)
{
    cout << i << endl;
    ++i;
}
```

prints the values from 0 to 4, inclusive. The loop can be read as “While (i.e. as long as) the condition ( $i \neq 5$ ) returns true, execute the statement (`cout << i << endl;`) then (`++i`). When the condition ( $i \neq 5$ ) returns false, terminate the loop.”

The syntax of a while loop is

```
while (condition)
{
    body;
}
```

The individual components are:

The condition ( $i \neq 5$ )

Boolean expression that is evaluated before each execution, or “iteration,” of the body. If the expression returns true, the body is executed; if it returns false, the loop terminates.

The body (`cout << i << endl; ++i;`)  
 Statement(s) to be executed if the condition returns true.

The loop performs 5 iterations:

```
int i = 0;                                // i is 0 (line 12)



---


condition:  

i != 5                                     // True

body (iteration 1):  

cout << i << endl;                      // Print the value of i (0) and a newline  

++i;                                         // i is 1



---


condition:  

i != 5                                     // True

body (iteration 2):  

cout << i << endl;                      // Print the value of i (1) and a newline  

++i;                                         // i is 2



---


condition:  

i != 5                                     // True

body (iteration 3):  

cout << i << endl;                      // Print the value of i (2) and a newline  

++i;                                         // i is 3



---


condition:  

i != 5                                     // True

body (iteration 4):  

cout << i << endl;                      // Print the value of i (3) and a newline  

++i;                                         // i is 4



---


condition:  

i != 5                                     // True

body (iteration 5):  

cout << i << endl;                      // Print the value of i (4) and a newline  

++i;                                         // i is 5



---


condition:  

i != 5                                     // False
```

Terminate loop

The body of a loop, like that of a conditional statement, can contain both nested conditional statements and loops, as in (lines 22-31)

```

for (int i = 0; i != 5; ++i)           // Main loop
{
    if (i % 2 == 0)                   // Nested conditional statement
        cout << i << " is even\n";
    else
        cout << i << " is odd\n";

    for (int n = i - 1; n >= 0; --n)   // Nested loop
        cout << n << endl;
}

```

For each value of *i* from 0 to 4 inclusive (line 22), the conditional statement (lines 24-27)

```

if (i % 2 == 0)
    cout << i << " is even\n";
else
    cout << i << " is odd\n";

```

is first evaluated. If the condition

```

i % 2 == 0                         // Is the remainder of i/2 equal to 0?
// (Is i divisible by 2?)

```

returns true, then line 25,

```
cout << i << " is even\n";          // Print "<i> is even"
```

is executed; otherwise, line 27,

```
cout << i << " is odd\n";          // Print "<i>" is odd"
```

is executed. The nested loop (lines 29-30)

```

for (int n = i - 1; n >= 0; --n)    // For each integer value n
    cout << " " << n << endl;       // from (i - 1) down to 0 inclusive,
// print " <n>"                  // print " <n>"
```

is then evaluated for the current value of *i*. The main loop performs 5 iterations, and within each of those iterations, the nested loop performs *i* iterations:

---

```
int i = 0;                           // i is 0
```

---

```
i != 5                                // True
```

---

iteration 1 (main loop) :

```

if (i % 2 == 0)
    cout << i << " is even\n";      // Print "0 is even"
else
    cout << i << " is odd\n";

int n = i - 1;                      // n is -1

n >= 0                                // False
Terminate nested loop

++i;                                    // i is 1

```

---

i != 5 // True

iteration 2 (main loop) :

```

if (i % 2 == 0)
    cout << i << " is even\n";
else
    cout << i << " is odd\n";      // Print "1 is odd"

int n = i - 1;                      // n is 0

n >= 0                                // True
iteration 1 (nested loop):
    cout << " " << n << endl;    // Print " 0"
    --n;                                // n is -1

n >= 0                                // False
Terminate nested loop

++i;                                    // i is 2

```

---

i != 5 // True

iteration 3 (main loop) :

```

if (i % 2 == 0)
    cout << i << " is even\n";      // Print "2 is even"
else
    cout << i << " is odd\n";

int n = i - 1;                      // n is 1

n >= 0                                // True
iteration 1 (nested loop):
    cout << " " << n << endl;    // Print " 1"
    --n;                                // n is 0

```

```
n >= 0                                // True
iteration 2 (nested loop):
    cout << " " << n << endl;        // Print " 0"
    --n;                                // n is -1

n >= 0                                // False
Terminate nested loop

++i;                                    // i is 3
```

---

```
i != 5                                // True

iteration 4 (main loop):

if (i % 2 == 0)
    cout << i << " is even\n";
else
    cout << i << " is odd\n";        // Print "3 is odd"

int n = i - 1;                          // n is 2

n >= 0                                // True
iteration 1 (nested loop):
    cout << " " << n << endl;        // Print " 2"
    --n;                                // n is 1

n >= 0                                // True
iteration 2(nested loop):
    cout << " " << n << endl;        // Print " 1"
    --n;                                // n is 0

n >= 0                                // True
iteration 3 (nested loop):
    cout << " " << n << endl;        // Print " 0"
    --n;                                // n is -1

n >= 0                                // False
Terminate nested loop

++i;                                    // i is 4
```

---

```
i != 5                                // True

iteration 5 (main loop):

if (i % 2 == 0)
    cout << i << " is even\n";        // Print "4 is even"
else
    cout << i << " is odd\n";

int n = i - 1;                          // n is 3
```

```

n >= 0                                // True
iteration 1 (nested loop):
    cout << " " << n << endl;        // Print " 3"
    --n;                                // n is 2

n >= 0                                // True
iteration 2 (nested loop):
    cout << " " << n << endl;        // Print " 2"
    --n;                                // n is 1

n >= 0                                // True
iteration 3 (nested loop):
    cout << " " << n << endl;        // Print " 1"
    --n;                                // n is 0

n >= 0                                // True
iteration 4 (nested loop):
    cout << " " << n << endl;        // Print " 0"
    --n;                                // n is -1

n >= 0                                // False
Terminate nested loop

++i;                                    // i is 5


---


i != 5                                // False

Terminate main loop

```

The condition can be omitted from a loop by using the “true” and “break” keywords, as in (lines 35-42)

```

for (int x = 1000; true; ++x)
{
    if (x % 17 == 0)
    {
        cout << x << " is divisible by 17\n";
        break;
    }
}

```

This loop prints the first number greater than or equal to 1000 that is divisible by 17. In this context, the boolean value “true” (line 35) is treated as a boolean expression that always returns true. The “break” keyword (line 40) is used to specify the condition under which the loop will terminate. The loop performs 4 iterations:

```

initializer:
    int x = 1000;                      // x is 1000


---


condition:
    true

```

```

body (iteration 1):
if (x % 17 == 0)                                // False
{
    cout << x << " is divisible by 17\n";
    break;
}

postBody:
++x;                                              // x is 1001

```

---

```

condition:
true

body (iteration 2):
if (x % 17 == 0)                                // False
{
    cout << x << " is divisible by 17\n";
    break;
}

postBody:
++x;                                              // x is 1002

```

---

```

condition:
true

body (iteration 3):
if (x % 17 == 0)                                // False
{
    cout << x << " is divisible by 17\n";
    break;
}

postBody:
++x;                                              // x is 1003

```

---

```

condition:
true

body (iteration 4):
if (x % 17 == 0)                                // True
{
    cout << x << " is divisible by 17\n";   // Print "1003 is divisible by 17"
    break;                                         // Terminate the loop
}

```

This loop can also be written as a while loop:

```

int x = 1000;

while (true)
{
    if (x % 17 == 0)                                // If x is divisible by 17
    {
        cout << x << " is divisible by 17\n";   // Print "<x> is divisible by 17"
        break;                                       // Terminate the loop
    }
    else                                              // Otherwise,
    {
        ++x;                                         // Increment x, then perform
    }                                                 // the next iteration
}

```

Note that without the “break” keyword, the program would become trapped in an infinite (non-terminating) loop, printing the message “1003 is divisible by 17” (executing line 39) indefinitely.

The complete program output is

```

0                                         // for loop (lines 7-8)
1
2
3
4

0                                         // while loop (lines 14-18)
1
2
3
4

0 is even                               // Nested conditional (lines 24-27)
1 is odd                                // Nested conditional (lines 24-27)
0                                         // Nested loop (lines 29-30)
2 is even                               // Nested conditional (lines 24-27)
1                                         // Nested loop (lines 29-30)
0
3 is odd                                // Nested conditional (lines 24-27)
2                                         // Nested loop (lines 29-30)
1
0
4 is even                               // Nested conditional (lines 24-27)
3                                         // Nested loop (lines 29-30)
2
1
0

1003 is divisible by 17      // Iteration 4 (line 39)

```

### 3.4: Boolean Variables

*Source folder: booleanVariables*

*Chapter outline*

- Using boolean variables to increase code readability

Recall from Chapter 1.1 that variables of type `bool` store true / false (boolean) values. The result returned by a boolean expression can therefore be assigned to a boolean variable. Given lines 7-10,

```
int k;

cout << "Enter a number (integer): ";
cin >> k;
```

for example, line 12,

```
bool isPositive = (k > 0);
```

initializes the variable `isPositive` to the result of the expression `(k > 0)`. If the user enters a positive number, `isPositive` will be initialized to true; otherwise, it will be initialized to false. Similarly, in line 13,

```
bool isNegative = (k < 0);
```

`isNegative` will be initialized to either true or false, depending on the result of the expression `(k < 0)`. Line 14,

```
bool isEven = (k % 2 == 0);
```

initializes `isEven` to the result of the expression `(k % 2 == 0)` (is `k` divisible by 2?).

A boolean variable can be used as the condition in an “if”, “else”, or “else if” clause. Lines 16-17,

```
if (isEven)
    cout << "You entered an even number\n";
```

for example, are shorthand for

```
if (isEven == true)
    cout << "You entered an even number\n";
```

Recall from Chapter 3.2 that the not operator (!) negates a boolean expression. Lines 19-20,

```
if (!isEven);                                // "If is not even..."
    cout << "You entered an odd number\n";
```

are shorthand for

```
if (!isEven == true) // if (isEven == false)
    cout << "You entered an odd number\n";
```

Boolean variables can also be used in compound expressions, such as (lines 22-23)

```
if (!isPositive && !isNegative) // "If is not positive and
    cout << "You entered 0\n"; // is not negative"
```

which is shorthand for

```
if ((!isPositive == true) && (!isNegative == true))
    cout << "You entered 0\n";
```

and equivalent to

```
if ((isPositive == false) && (isNegative == false))
    cout << "You entered 0\n";
```

Some sample runs of the program in this chapter are

```
Enter a number (integer): -3
You entered an odd number
```

```
Enter a number (integer): 0
You entered an even number
You entered 0
```

```
Enter a number (integer): 5
You entered an odd number
```

### 3.5: Putting It All Together

*Source folder: retirementAge*

*Chapter outline*

- A small program that illustrates all of the key concepts from Parts 1-3

A sample run of the program in this chapter is

What is your current age? 43

At what age do you plan to retire? 60

In 5 years, you will be 48 years old.  
You will have 12 years before retirement.

In 10 years, you will be 53 years old.  
You will have 7 years before retirement.

In 15 years, you will be 58 years old.  
You will have 2 years before retirement.

In 20 years, you will be 63 years old.  
You will have been retired for 3 years.

In 25 years, you will be 68 years old.  
You will have been retired for 8 years.

This program obtains the user's current age and planned age of retirement, then displays their retirement status for the next 25 years, in 5-year intervals. For each interval, the program first displays the user's age at that point in the future. Then, if the user is retired, the program tells them how many years they have been retired; if not, it tells them how many years they have before retirement.

Lines 7-14,

```
int currentAge;
int retirementAge;

cout << "\nWhat is your current age? ";
cin >> currentAge;

cout << "\nAt what age do you plan to retire? ";
cin >> retirementAge;
```

prompt the user to enter their current age and planned age of retirement, storing the respective values in the int variables currentAge and retirementAge. The loop in lines 16-34 then displays the user's retirement status:

```

for (int futureYears = 5; futureYears <= 25; futureYears += 5)
{
    int futureAge = currentAge + futureYears;
    bool isRetired = (futureAge >= retirementAge);

    cout << "\nIn " << futureYears << " years, you will be " << futureAge <<
        " years old.\n";

    if (isRetired)
    {
        cout << " You will have been retired for " <<
            futureAge - retirementAge << " years.\n";
    }
    else
    {
        cout << " You will have " << retirementAge - futureAge <<
            " years before retirement.\n";
    }
}

```

futureYears (line 16) represents the number of years in the future (5, 10, 15, 20, or 25). For each 5-year interval, futureAge is the user's age at that point in the future (line 18):

```
int futureAge = currentAge + futureYears;
```

Line 19,

```
bool isRetired = (futureAge >= retirementAge);
```

then determines whether or not the user is retired at that point. If so, the expression

```
(futureAge >= retirementAge)      // Is the user's age at that point in the
                                    // future greater than or equal to their
                                    // planned age of retirement?
```

will return true; if not, it will return false. The result is stored in the boolean variable isRetired. Lines 21-22,

```
cout << "\nIn " << futureYears << " years, you will be " << futureAge <<
    " years old.\n";
```

print the user's future age, generating the output

```
In <futureYears> years, you will be <futureAge> years old.
```

The nested conditional statement (lines 24-34),



```

condition:
futureYears <= 25                                // True

body (iteration 2):

int futureAge = currentAge + futureYears;          // futureAge is 53
bool isRetired = (futureAge >= retirementAge);    // isRetired is false

cout << "\nIn " << futureYears << " years, you will be " << futureAge <<
    " years old.\n";

if (isRetired)                                     // False
{
    cout << " You will have been retired for " <<
        futureAge - retirementAge << " years.\n";
}
else                                                 // Execute this branch
{
    cout << " You will have " << retirementAge - futureAge <<
        " years before retirement.\n";
}

postBody:
futureYears += 5                                    // futureYears is 15

```

---

```

condition:
futureYears <= 25                                // True

body (iteration 3):

int futureAge = currentAge + futureYears;          // futureAge is 58
bool isRetired = (futureAge >= retirementAge);    // isRetired is false

cout << "\nIn " << futureYears << " years, you will be " << futureAge <<
    " years old.\n";

if (isRetired)                                     // False
{
    cout << " You will have been retired for " <<
        futureAge - retirementAge << " years.\n";
}
else                                                 // Execute this branch
{
    cout << " You will have " << retirementAge - futureAge <<
        " years before retirement.\n";
}

postBody:
futureYears += 5                                    // futureYears is 20

```

---

```

condition:
futureYears <= 25                                // True

```

```
body (iteration 4):

    int futureAge = currentAge + futureYears;           // futureAge is 63
    bool isRetired = (futureAge >= retirementAge);     // isRetired is true

    cout << "\nIn " << futureYears << " years, you will be " << futureAge <<
        " years old.\n";

    if (isRetired)                                     // True
    {                                                 // Execute this branch
        cout << " You will have been retired for " <<
            futureAge - retirementAge << " years.\n";
    }
    else
    {
        cout << " You will have " << retirementAge - futureAge <<
            " years before retirement.\n";
    }

postBody:
    futureYears += 5                                  // futureYears is 25
```

```
condition:  
    futureYears <= 25                                // True  
  
body (iteration 5):  
  
    int futureAge = currentAge + futureYears;           // futureAge is 68  
    bool isRetired = (futureAge >= retirementAge);      // isRetired is true  
  
    cout << "\nIn " << futureYears << " years, you will be " << futureAge <<  
        " years old.\n";  
  
    if (isRetired)                                     // True  
    {                                                 // Execute this branch  
        cout << " You will have been retired for " <<  
            futureAge - retirementAge << " years.\n";  
    }  
    else  
    {  
        cout << " You will have " << retirementAge - futureAge <<  
            " years before retirement.\n";  
    }  
  
postBody:  
    futureYears += 5                                  // futureYears is 30
```

```
condition:  
    futureYears <= 25 // False
```

Terminate loop



# Part 4: Functions and Scope

## 4.1: Functions

*Source folder: functions*

*Chapter outline*

- Declaring, defining, and calling functions
- The difference between arguments and parameters

Managing complexity is crucial to effective programming. To this end, functions are essential because they provide a mechanism for reducing a large, complex task to a set of smaller, simpler tasks. Functions can also reduce code duplication and facilitate program organization.

The syntax for declaring a function is

```
returnType functionName(parameters);
```

Recall from Chapter 1.1 that the return type of a function is the type of output value generated by that function. “Parameters,” conversely, are input values. A function can have any number of parameters, though the type of each parameter must be specified. Line 3,

```
double salesTax(double price, double rate);
```

for example, declares the function salesTax, which has two parameters of type double (price and rate) and returns a value of type double.

Once a function has been declared, it can be called (executed) any number of times. A function's “definition” contains the actual statements to be executed whenever that function is called. Every function must be defined once and only once. Note, however, that a function definition can be placed anywhere after its corresponding declaration. The syntax for defining a function is

```
returnType functionName(parameters)
{
    body
}
```

where the body contains the statements to be executed whenever the function is called. The salesTax function is defined as (lines 32-35)

```
double salesTax(double price, double rate)
{
    return price * (rate / 100);
}
```

The function, when called, returns the amount of sales tax on an item at the given price and rate. The expression

```
salesTax(24.99, 6.25) // price = $24.99, rate = 6.25%
```

for example, returns 1.561875 ( $24.99 \times 6.25/100$ ). Line 4,

```
double shippingCost(double weight);
```

declares the function `shippingCost`, which has a single parameter of type `double` (`weight`) and returns a value of type `double`. The function definition is (lines 37-47)

```
double shippingCost(double weight)
{
    double cost;

    if (weight < 5.0)
        cost = 7.99;
    else
        cost = 2.00 * weight;

    return cost;
}
```

The function, when called, returns the shipping cost of an item with the given weight. If the item weighs less than 5 pounds, the shipping cost is a flat rate of \$7.99 (line 42); otherwise, the shipping cost is \$2.00 per pound (line 44). The expression

```
shippingCost(2.50) // weight = 2.50 lbs (less than 5.0 lbs)
```

for example, returns 7.99, while

```
shippingCost(8.25) // weight = 8.25 lbs (not less than 5.0 lbs)
```

returns 16.50 ( $2.00 \times 8.25$ ).

Lines 10-21 of main,

```
double retailPrice;
double salesTaxRate;
double itemWeight;

cout << "Enter retail price (e.g. for $24.99, enter 24.99): ";
cin >> retailPrice;

cout << "Enter sales tax rate (e.g. for 6.25%, enter 6.25): ";
cin >> salesTaxRate;

cout << "Enter item weight (e.g. for 2.50 lbs, enter 2.50): ";
cin >> itemWeight;
```

prompt the user to enter the retail price of an item, the sales tax rate, and the weight of the item, storing the values in the variables `retailPrice`, `salesTaxRate`, and `itemWeight`. In lines 23-25,

```
double totalCost = retailPrice +
    salesTax(retailPrice, salesTaxRate) +
    shippingCost(itemWeight);
```

the expression

```
retailPrice +
    salesTax(retailPrice, salesTaxRate) +
    shippingCost(itemWeight)
```

returns the total cost of the item (a value of type `double`), which is then stored in the variable `totalCost`.

The execution of a function is known as a “function call.” In the function call (line 24)

```
salesTax(retailPrice, salesTaxRate)
```

`retailPrice` and `salesTaxRate` are called “arguments,” which are said to be “passed” to the `salesTax` function. Similarly, in the function call (line 25)

```
shippingCost(itemWeight)
```

`itemWeight` is the argument passed to the `shippingCost` function.

Note the distinction between “arguments” and “parameters.” An argument, such as `itemWeight` (line 25), is the value passed to a function in an actual call. A parameter, such as `weight` (line 37), is a separate variable that only exists with the body of a function. Each time a function is called, its parameters are initialized to the values of the corresponding arguments, then destroyed when the function terminates.

Line 27,

```
cout << "\nThe total cost is $" << totalCost << endl;
```

prints the value of `totalCost`, generating the output

```
The total cost is $<totalCost>
```

Some sample runs of the program are

```
Enter retail price (e.g. for $24.99, enter 24.99): 29.99
Enter sales tax rate (e.g. for 6.25%, enter 6.25): 7.50
Enter item weight (e.g. for 2.50 lbs, enter 2.50): 3      // Flat-rate shipping
```

```
The total cost is $40.2293
```

```
Enter retail price (e.g. for $24.99, enter 24.99): 14.99
```

52

Enter sales tax rate (e.g. for 6.25%, enter 6.25): 4.75

Enter item weight (e.g. for 2.50 lbs, enter 2.50) // \$2.00/lb shipping

The total cost is \$32.702

## 4.2: Namespaces

*Source folder: namespaces*

*Chapter outline*

- Declaring and accessing components in namespaces
- The “using” keyword

A “namespace” is a named region in which functions, variables, and classes (discussed later) can be declared. Namespaces aid in program organization by allowing related components to be grouped together.

The program in this chapter is functionally identical to that of the previous chapter. The code, however, has been organized further by creating additional functions and employing namespaces. In lines 3-8,

```
namespace userInterface
{
    double getRetailPrice();
    double getSalesTaxRate();
    double getItemWeight();
}
```

for example, the functions `getRetailPrice`, `getSalesTaxRate`, and `getItemWeight` are declared in namespace `userInterface`. Each function, when called, prompts the user to enter the corresponding value (retail price, sales tax rate, item weight), then returns that value. Their definitions are (lines 33-70)

```
namespace userInterface
{
    double getRetailPrice()
    {
        using namespace std;

        double x;

        cout << "Enter retail price (e.g. for $24.99, enter 24.99): ";
        cin >> x;

        return x;
    }
}
```

```

double getSalesTaxRate()
{
    using namespace std;

    double x;

    cout << "Enter sales tax rate (e.g. for 6.25%, enter 6.25): ";
    cin >> x;

    return x;
}

double getItemWeight()
{
    using namespace std;

    double x;

    cout << "Enter item weight (e.g. for 2.50 lbs, enter 2.50): ";
    cin >> x;

    return x;
}
};

```

The opening and closing braces (lines 4, 8, 33, 70) indicate where the namespace begins and ends. Also note that the closing brace requires a semicolon (lines 8, 70). In lines 10-15,

```

namespace feeCalculator
{
double totalFees(double itemPrice, double salesTaxRate, double itemWeight);
double salesTax(double price, double rate);
double shippingCost(double weight);
};

```

the functions totalFees, salesTax, and shippingCost are declared in namespace feeCalculator. The salesTax and shippingCost functions are identical to those of the previous chapter. The totalFees function calculates the sales tax and shipping cost by calling the corresponding functions, then returns the sum. The definitions are (lines 72-95)

```

namespace feeCalculator
{
double totalFees(double itemPrice, double salesTaxRate, double itemWeight)
{
    return salesTax(itemPrice, salesTaxRate) + shippingCost(itemWeight);
}

double salesTax(double price, double rate)
{
    return price * (rate / 100);
}

```

```

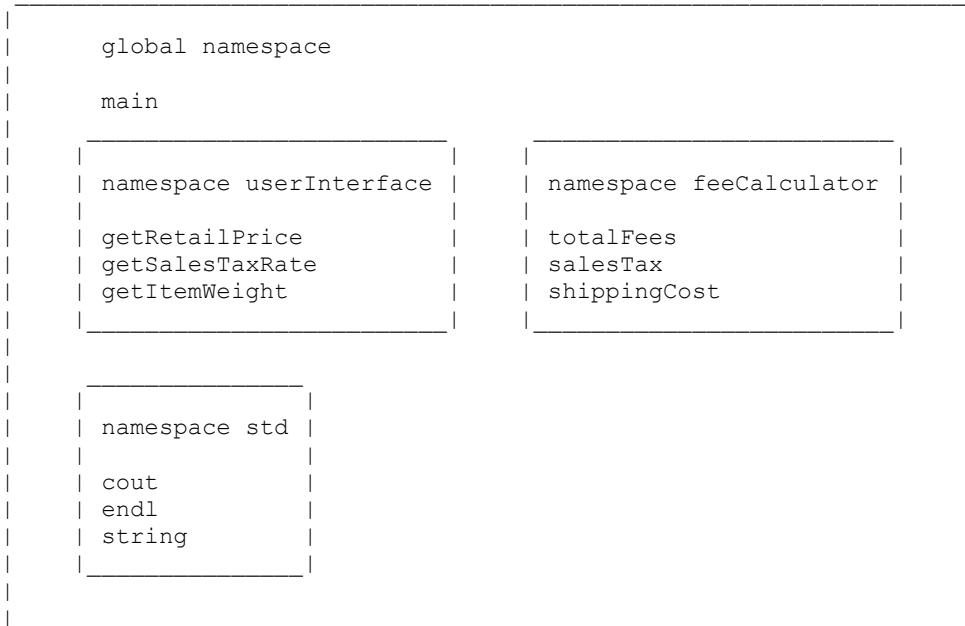
double shippingCost(double weight)
{
    double cost;

    if (weight < 5.0)
        cost = 7.99;
    else
        cost = 2.00 * weight;

    return cost;
}
};

```

`cout`, `endl`, the string datatype, and nearly all other C++ Standard Library components are declared in namespace `std`. The function `main` (lines 17-31) is declared in what is known as the “global,” or outermost, namespace. The global namespace encloses all other namespaces. The program can thus be depicted as



Relative to the global namespace, `userInterface`, `feeCalculator`, and `std` are all inner namespaces. From within the function `main`, anything declared in one of these inner namespaces must therefore be referred to by its fully qualified name, as in lines 21-26,

```

double retailPrice = userInterface::getRetailPrice();
double salesTaxRate = userInterface::getSalesTaxRate();
double itemWeight = userInterface::getItemWeight();

```

```
double totalCost = retailPrice +
    feeCalculator::totalFees(retailPrice, salesTaxRate, itemWeight);
```

The :: is called the “scope resolution operator.”

From within a given namespace, anything declared in that same namespace can be referred to without qualifying its name, as in lines 72-77,

```
namespace feeCalculator
{
    double totalFees(double price, double salesTaxRate, double itemWeight)
    {
        return salesTax(price, salesTaxRate) + shippingCost(itemWeight);
    }

    // ...
}
```

`salesTax` and `shippingCost` are declared in the same namespace as `totalFees`. Within `totalFees`, they can therefore be referred to without qualification (as in line 76), as opposed to

```
return feeCalculator::salesTax(price, salesTaxRate) +
    feeCalculator::shippingCost(itemWeight);

// The feeCalculator:: is unnecessary here
```

Line 19,

```
using namespace std;
```

is a “using directive,” which makes all components declared in namespace `std` available without qualification. Without line 19, `cout` and `endl` would have to be referred to by their fully qualified names, `std::cout` and `std::endl`, as in

```
std::cout << "\nThe total cost is $" << totalCost << std::endl;
```

main, for example, could have been written as

```
using namespace std;                                // Using directive
using namespace userInterface;                      // Using directive
using feeCalculator::totalCost;                     // Using declaration

double retailPrice = getRetailPrice();
double salesTaxRate = getSalesTaxRate();
double itemWeight = getItemWeight();

double totalCost = retailPrice +
    totalFees(retailPrice, salesTaxRate, itemWeight);

cout << "\nThe total cost is $" << totalCost << endl;

return 0;
```

## The using directive

```
using namespace userInterface;
```

allows all of the components declared in namespace userInterface to be referred to without qualification, while the “using declaration”

```
using feeCalculator::totalCost;
```

allows only totalCost to be referred to without qualification. All other components declared in namespace feeCalculator (salesTax, shippingCost) would have to be fully qualified, otherwise a compiler error would result.



## 4.3: Scope

*Source folder: scope*

*Chapter outline*

- How a variable's point of declaration (namespace, function, loop, or block) determines its lifetime within a program

The lifetime of a variable, i.e. when it is created and destroyed, is known as its “scope.” There are 4 main types of scope:

### Namespace scope

A variable declared outside any function has “namespace scope”; it is destroyed when the program terminates. Variables declared in the global namespace are known as “global variables.”

### Function scope

A variable declared in a function, but outside any loop or conditional statement, has “function scope”; it is destroyed when the function terminates. Recall from Chapter 4.1 that this also applies to function parameters.

### Loop scope

A variable declared in the initializer of a loop has “loop scope”; it is destroyed when the loop terminates.

### Block scope

A variable declared in the body of a loop or conditional statement has “block scope”; it is destroyed at the end of the body in which it was declared (i.e. the end of its enclosing block).

Note that all of the programs thus far have used variables with either function, loop, or block scope.

A sample run of the program in this chapter is

```
This program calculates the area and circumference of 3 circles,
as well as running totals of the area and circumference.
```

```
Enter the radius of circle 1 (e.g. for 3.5 m, enter 3.5): 7
```

```
Area of circle 1 = 153.938 sq m
Circumference of circle 1 = 43.9823 m
Sum of areas = 153.938 sq m
Sum of circumferences = 43.9823 m
```

```
Enter the radius of circle 2 (e.g. for 3.5 m, enter 3.5): 4
```

```
Area of circle 2 = 50.2654 sq m
Circumference of circle 2 = 25.1327 m
```

```

Sum of areas = 204.203 sq m
Sum of circumferences = 69.115 m

Enter the radius of circle 3 (e.g. for 3.5 m, enter 3.5): 9

Area of circle 3 = 254.469 sq m
Circumference of circle 3 = 56.5486 m
Sum of areas = 458.672 sq m
Sum of circumferences = 125.664 m

```

When the program is run, line 4,

```
const double pi = 3.14159;
```

creates the global variable `pi`, initialized to 3.14159. `pi` has namespace scope, so it will not be destroyed until the program terminates. The “`const`” keyword indicates that `pi` is a constant, which cannot be modified; attempting to do so will generate a compiler error. The “`const`” keyword will be discussed further in Part 5.

Lines 6-7,

```
double calculateArea(double r);
double calculateCircumference(double r);
```

declare the functions `calculateArea` and `calculateCircumference`, which return the area and circumference of a circle with radius `r`. Their definitions are (lines 44-54)

```
double calculateArea(double r)                      // Uses the formula
{
    return pi * pow(r, 2);                          // Area = pi * radius-squared
}

double calculateCircumference(double r)            // Uses the formula
{
    double diameter = 2 * r;                        // Circumference = pi * diameter
    return pi * diameter;
}
```

The expression (line 46)

```
pow(r, 2)      // returns r to the power of 2
```

calls the function

```
double pow(double base, int exponent);
```

which is provided by the Standard Library header `<cmath>` (line 1).

The function `main` (lines 9-42) is then executed. Lines 16-17,

```
double sumAreas = 0;
double sumCircumferences = 0;
```

create the variables sumAreas and sumCircumferences, which will be used to store the running totals of the area and circumference. Both of these variables have function scope; they will be destroyed when main terminates.

The loop initializer (line 19),

```
int c = 1;
```

then creates the variable c, which represents the number of the current circle (1, 2, or 3). c has loop scope, so it will be destroyed when the loop (lines 19-39) terminates (after iteration 3).

In each iteration of the loop (lines 21-38), lines 21-27,

```
double radius;

cout << "Enter the radius of circle " <<
c <<
"(e.g. for 3.5 m, enter 3.5): ";

cin >> radius;
```

prompt the user to enter the radius of circle c (1, 2, or 3), then store that value in the variable radius. radius has block scope; it is created during each iteration of the loop (in line 21) and destroyed at the end of each iteration (line 39). Lines 29-30,

```
double area = calculateArea(radius);
double circumference = calculateCircumference(radius);
```

create two more block-scoped variables, area and circumference. Like radius, these are created (in lines 29-30) and destroyed (in line 39) a total of 3 times.

Each call to (execution of) calculateArea entails the creation of r (line 44), a function-scoped variable that is destroyed at the end of the call (line 47). Similarly, in each call to calculateCircumference, the function-scoped variables r and diameter are created (lines 49, 51) and destroyed (line 54).

After the area and circumference of the current circle are calculated, lines 32-33,

```
sumAreas += area;
sumCircumferences += circumference;
```

add the area and circumference to their respective running totals, sumAreas and sumCircumferences.

Lines 35-38,

```
cout << "\nArea of circle " << c << " = " << area << " sq m\n";
cout << "Circumference of circle " << c << " = " << circumference << " m\n";
```

```
cout << "Sum of areas = " << sumAreas << " sq m\n";
cout << "Sum of circumferences = " << sumCircumferences << " m\n\n";
```

then print the calculated values, generating the output

```
Area of circle <c> = <area> sq m
Circumference of circle <c> = <circumference> m
Sum of areas = <sumAreas> sq m
Sum of circumferences = <sumCircumferences> m
```

At the end of each iteration (1, 2, and 3), the block-scoped variables radius, area, and circumference are destroyed. When the loop terminates (after iteration 3), the loop-scoped variable c is destroyed. The function main then terminates, resulting in the destruction of the function-scoped variables sumAreas and sumCircumferences. Finally, the program terminates, at which point the global (namespace-scoped) variable pi is destroyed.

When a variable is destroyed, it is said to “go out of scope.” The variables radius, area, and circumference, for example, all go out of scope in line 39, while the variable diameter goes out of scope in line 54.

Minimizing the scope of variables makes code easier to understand and less prone to error. Suppose for example, that c (line 19) was declared as a global variable:

```
const double pi = 3.14159;
int c;

double calculateArea(double r);
double calculateCircumference(double r);

int main()
{
    // ...

    for (c = 1; c <= 3; ++c)
    {
        // ...
    }
}
```

Although this is perfectly valid, the exact purpose of c would not be immediately obvious to a reader at the point of declaration. c's status as a global variable also implies that it is intended to be accessed by multiple functions, which is not actually the case. And worst of all, it would make it easy to create bugs, such as

```
double calculateArea(double r)
{
    --c;                                // This would cause an infinite loop in main
    return pi * pow(r, 2);                // by ensuring that the value of c never
                                            // reaches 4 (to terminate the loop)
}
```

## 4.4: Function Overloading

*Source folder: functionOverloading*

*Chapter outline*

- *Creating multiple functions with the same name*

Creating multiple functions with the same name is known as “function overloading.” Overloaded functions can differ by their number and / or types of parameters. The function area (lines 4-6)

```
double area(double length);
double area(double length, double width);
double area(double length, double width, double height);
```

for example, is overloaded by number of parameters. The first overload returns the area of a square, the second returns the area of a rectangle, and the third returns the area of a cuboid (rectangular prism, or box) (lines 33-48):

```
double area(double length)
{
    return pow(length, 2);
}

double area(double length, double width)
{
    return length * width;
}

double area(double length, double width, double height)
{
    return (2 * length * width) +
        (2 * length * height) +
        (2 * width * height);
}
```

Lines 12-23,

```
double length;
double width;
double height;

cout << "Enter length (e.g. for 2.5 m, enter 2.5): ";
cin >> length;

cout << "Enter width (e.g. for 2.5 m, enter 2.5): ";
cin >> width;

cout << "Enter height (e.g. for 2.5 m, enter 2.5): ";
cin >> height;
```

prompt the user to enter values for length, width, and height.

The process by which the compiler selects the appropriate overload for a given function call is known as “overload resolution.” Lines 25-28,

```
cout << "The area of a square is " << area(length) << " sq m\n";
cout << "The area of a rectangle is " << area(length, width) << " sq m\n";
cout << "The area of a cuboid is " << area(length, width, height) <<
    " sq m\n";
```

for example, print the area of a square, rectangle, and cuboid with the given dimensions. At compile time (i.e. when the program is compiled), the compiler selects the overloads based on the number of arguments passed in each call.

A sample run of the program is

```
Enter length (e.g. for 2.5 m, enter 2.5): 4
Enter width (e.g. for 2.5 m, enter 2.5): 6
Enter height (e.g. for 2.5 m, enter 2.5): 3.5

The area of a square is 16 sq m
The area of a rectangle is 24 sq m
The area of a cuboid is 118 sq m
```

The pow function (line 35) is an example of overloading by parameter type. The Standard Library provides several overloads of pow, including

```
double pow(double base, int exponent);
double pow(double base, double exponent);
```

Given

```
double x = 1.618;
int y = 3;
double z = 4.5;
```

and the expression

```
pow(x, y) // The arguments, x and y, are of type double and int
```

for example, the compiler would select the first overload. Given the expression

```
pow(x, z) // The arguments, x and z, are of type double and double
```

however, the compiler would select the second overload.

## 4.5: Header Files and Inline Functions

*Source folder: headerFiles*

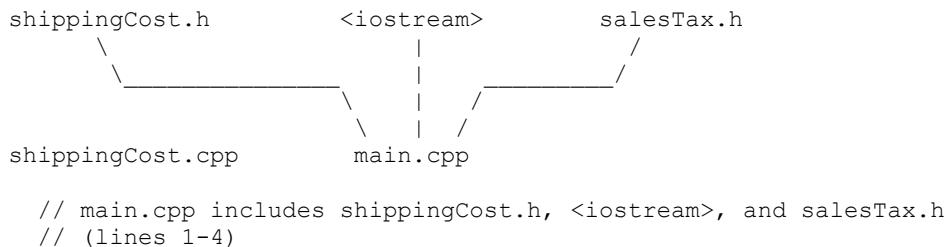
*Chapter outline*

- Translation units, linking, and the one-definition rule

A “translation unit” is the source code in a single .cpp file and all of its included headers. Each translation unit is compiled separately, resulting in a set of object files containing machine code. A separate program called the “linker” then combines all of the machine code and generates an executable file.

Because each translation unit is compiled separately, any translation unit that calls a given function must contain a declaration of that function. Recall from Chapter 4.1, however, that a program can contain only one definition of each function; this is known as the “one-definition rule,” or ODR. By separating a function’s declaration and definition into .h and .cpp files, its declaration can be duplicated any number of times (via #include) without duplicating its definition.

The program in this chapter is functionally identical to that of Chapter 4.1, but has been partitioned into multiple source files:



`shippingCost.h` contains the declaration of the `shippingCost` function, while `shippingCost.cpp` contains the definition. For `main.cpp` to call the `shippingCost` function (line 25), it must contain the declaration, which is provided by the include directive in line 4. The advantage of having multiple translation units is that changes can be made to one .cpp file without requiring recompilation of the others. For large programs, this can drastically reduce build times.

`main.cpp` also requires the declaration of the `salesTax` function for the call in line 24, so `salesTax.h` is included in line 3. Note, however that `salesTax.h` contains both the function declaration (line 4) and definition (lines 6-9):

```

double salesTax(double price, double rate); // Declaration

```

```
inline double salesTax(double price, double rate)           // Definition
{
    return price * (rate / 100);
}
```

The “`inline`” keyword (line 6) indicates that `salesTax` is an “`inline function`.” Designating a function as `inline` instructs the compiler to insert the code from the entire function body at each call site, thus eliminating the time overhead of an actual function call. In lines 23-25 (`main.cpp`),

```
double totalCost = retailPrice +
    salesTax(retailPrice, salesTaxRate) +           // Function call
    shippingCost(itemWeight);                      // Function call
```

for example, the compiler would generate the code

```
double totalCost = retailPrice +
    retailPrice * (salesTaxRate / 100) +           // Inline expansion
    shippingCost(itemWeight);                      // Function call
```

Inline functions are an exception to the ODR: for the compiler to perform inline expansion, the function’s definition must be visible within every translation unit containing calls to that function. The definition, however, must not appear more than once within the same translation unit. To prevent such duplication, any header file containing an inline function must also contain an “`include guard`.” The syntax for creating an include guard is

```
#ifndef HEADER_ID
#define HEADER_ID

// Source code...

#endif
```

where `HEADER_ID` is a unique identifier (customarily in all-caps) for the header file. An include guard prevents the compiler from processing everything between the `#ifndef` and `#endif` more than once within the same translation unit. The include guard for `salesTax.h` is in lines 1, 2, and 11.

Note that the degree to which inline expansion actually occurs is at the discretion of the compiler. Although it can result in faster code (by eliminating function calls), inline expansion can also result in slower code (by increasing code size). Furthermore, there is no definitive rule dictating which functions should be “`inlined`” (designated as `inline`). A general guideline, however, is to only inline functions with particularly short or trivial bodies.

# Part 5: Pointers, Arrays, and References

## 5.1: Pointers

*Source folder: pointers*

*Chapter outline*

- How variables are stored in memory
- The difference between dereferencing a pointer and reseating it

Computer memory can be pictured as a grid of equally-sized cells, or “bytes.” Each byte has a unique address denoting its location within the grid.

At runtime (i.e. during the course of a program's execution), each variable is constructed at a particular address in memory, and occupies a fixed number of bytes. The exact number of bytes used to store a given variable depends on its type (char, int, double, etc.), the hardware of the machine on which the program is run, and the compiler used to create the program. Each int, for example, might require 4 bytes using one architecture (hardware / compiler combination), but 8 bytes using another.

Memory addresses are commonly represented using hexadecimal (base-16) numbers. The base-10 number 10995, for example, is represented as 2AF3 in hexadecimal. For the purposes of illustration, however, consider a hypothetical architecture that uses base-10 numbers to represent memory addresses, and uses 1 byte to store each int. Given lines 7-8,

```
int x = 0;
int y = 1;
```

a possible memory layout at runtime is

01	02	03	04	05	06	07	08	09	10	
11	12	13	14	15	16	17	18	19	20	
21	22	23	24	25	26	27	28	29	30	

x is located at address 03 (with a value of 0) and y is located at address 17 (with a value of 1). Line 10,

```
int* p = nullptr;
```

declares and initializes the variable p, of type int\*. The type int\*, known as “pointer to int,” is read right-to-left, where the “indirection operator” (\*) is read as “pointer to.” A pointer is a type of variable

whose value is a memory address. In other words, while the value of an int variable (such as x) is an integer, the value of an int\* variable (such as p) is the in-memory address (location) of a particular int.

In line 10, p is initialized to the value nullptr ("null pointer"), which is a special value that represents "no address." The current state of p can be described as "p is a null pointer / null / nil," "p points to null / nil," or "p doesn't point to anything." Suppose that the architecture uses 3 bytes to store each int\*. As of line 10, a possible memory layout is

01	02	03	04	05	06	07	08	09	10	
		x(0)								
11	12	13	14	15	16	17	18	19	20	
						y(1)				
21	22	23	24	25	26	27	28	29	30	
			p(nullptr)							

p occupies 3 bytes of memory at address 23, and its value is nullptr. Lines 12-14,

```
cout << "p = nullptr = " << p << endl;
cout << "x = " << x << endl;
cout << "y = " << y << endl;
```

generate the output

```
p = nullptr = 00000000      // Hexadecimal (actual) representation of nullptr
x = 0
y = 1
```

Line 16,

```
p = &x;
```

can be read as "set the value of p to the address of x," or simply "set p to point to x." The & is called the "address-of operator," and the expression

```
&x
```

returns the in-memory address (location) of x. The current state of p can now be described as "p points to x," or "p is a pointer to x." x is said to be the "referent" of p. The memory layout is unchanged, but the value of p is now 03 (the address of x):

01	02	03	04	05	06	07	08	09	10	
		x(0)								
11	12	13	14	15	16	17	18	19	20	
						y(1)				
21	22	23	24	25	26	27	28	29	30	
			p(03)							

In line 17, the expression

`*p`

can be read as "the int pointed to by p," "the int at address p," or simply "the referent of p." The `*` is called the "indirection operator" or "dereference operator." Applying the indirection operator to a pointer is known as "derefencing" that pointer. Dereferencing a pointer returns its referent. Because `p` currently points to `x` (address 03), the expression

`*p`

is equivalent to

`x`

The statement (line 17)

`*p = 5;`

therefore means "set the referent of p (the int pointed to by p) to 5," and is equivalent to

`x = 5;`

Lines 19-22,

```
cout << "p = &x = " << p << endl;
cout << "*p = " << *p << endl;
cout << "x = " << x << endl;
cout << "y = " << y << endl << endl;
```

generate the output

```
p = &x = 0031FCC8      // Hexadecimal (actual) address of x
*p = 5
x = 5
y = 1
```

Line 24,

```
p = &y;      // Set the value of p to the address of y
```

sets `p` to point to `y`. Line 25,

```
*p = 8;      // Set the referent of p (the int pointed to by p) to 8
```

is therefore equivalent to

`y = 8;`

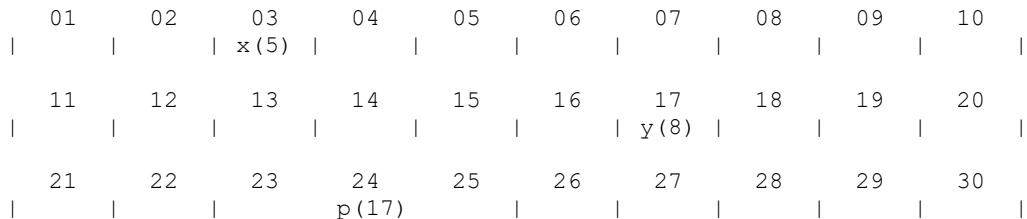
Lines 27-30,

```
cout << "p = &y = " << p << endl;
cout << "*p = " << *p << endl;
cout << "x = " << x << endl;
cout << "y = " << y << endl << endl;
```

generate the output

```
p = &y = 0031FCCC      // Hexadecimal (actual) address of y
*p = 8
x = 5
y = 8
```

As of line 30, the memory layout remains unchanged, but the value of p is now 17 (the address of y). The values of x and y have also been indirectly modified via p (x is now 5 and y is 8).



To summarize the aforementioned notation,

```
x is 5
y is 8
p is 17

&x is 03
&y is 17
&p is 23

*p is y (the int at address 17)
```

Note that dereferencing any pointer without a valid referent is undefined:

```
double* q = nullptr;      // q is null, so
*q = 1.618;              // *q is undefined

string* r;                // r does not have a referent, so
cout << *r;              // *r is undefined
```

Changing the object to which a pointer points is known as “reseating” a pointer. Line 16, for example, reseats p from null to x, while line 24 reseats p from x to y.

## 5.2: Pass by Reference

*Source folders*

*passByReference*  
*swapInts*

*Chapter outline*

- *Creating functions with pointer parameters*
- *The difference between pass-by-reference and pass-by-value*

A sample run of the program in this chapter is

```
Enter an integer value for x: 17
Enter an integer value for y: 43

The value of x is now 17
The value of y is now 43

Enter s to swap the values of x and y, or q to quit (case-sensitive): s

The value of x is now 43
The value of y is now 17

Enter s to swap the values of x and y, or q to quit (case-sensitive): s

The value of x is now 17
The value of y is now 43

Enter s to swap the values of x and y, or q to quit (case-sensitive): q
```

Line 3 (swap.h),

```
void swap(int* a, int* b); // Swap the values of the referents of a and b
```

declares the function swap, in namespace dss (short for “Data Structures from Scratch”). The return type “void” indicates that the function does not return a value. The function definition is (swap.cpp, lines 3-8)

```
void swap(int* a, int* b)
{
    int c = *a; // Use c to store the original value of a's referent
    *a = *b; // Set a's referent to the value of b's referent
    *b = c; // Set b's referent to the original value of a's referent
}
```

Lines 9-16 (main.cpp),

```

int x;
int y;

cout << "Enter an integer value for x: ";
cin >> x;

cout << "Enter an integer value for y: ";
cin >> y;

```

prompt the user to enter values for the variables x and y. In each iteration of the loop (lines 18-34), lines 20-21,

```

cout << "\nThe value of x is now " << x << endl;
cout << "The value of y is now " << y << endl;

```

print the current values of x and y, generating the output

```

The value of x is now <x>
The value of y is now <y>

```

Lines 23-28,

```

char userCommand;

cout << "\nEnter s to swap the values of x and y, or q to quit"
     " (case-sensitive): ";

cin >> userCommand;

```

prompt the user to enter a single character, which is stored in the variable userCommand. If the user enters 's', the addresses of x and y are passed to the swap function, after which the loop performs the next iteration. If the user enters any other character, the loop terminates, after which main terminates (lines 30-33):

```

if (userCommand == 's')
    dss::swap(&x, &y);
else
    break;

```

In the function call (line 31)

```
dss::swap(&x, &y);
```

the parameters, a and b (swap.cpp, line 3), are pointers to x and y, respectively. The function body (lines 5-7),

```

int c = *a;           // Use c to store the original value of a's referent
*a = *b;             // Set a's referent to the value of b's referent
*b = c;              // Set b's referent to the original value of a's referent

```

is therefore equivalent to

```

int c = x;           // Use c to store the original value of x
x = y;             // Set x to the value of y
y = c;             // Set y to the original value of x

```

If the initial values of x and y are 17 and 43, this becomes

```

int c = 17;
x = 43;
y = 17;

```

Passing the address of a variable is known as “pass-by-reference.” In line 31,

```
dss::swap(&x, &y);
```

for example, the addresses of x and y are passed to the swap function, so x and y are said to be “passed by reference.”

Passing the value of a variable is known as “pass-by-value.” Recall the program from Chapter 4.1:

```

double salesTax(double price, double rate);
double shippingCost(double weight);

// ...

double retailPrice;
double salesTaxRate;
double itemWeight;

// ...

double totalCost = retailPrice +
    salesTax(retailPrice, salesTaxRate) +
    shippingCost(itemWeight);

```

In the expression

```
salesTax(retailPrice, salesTaxRate)
```

the values of retailPrice and salesTaxRate are passed to the salesTax function, so retailPrice and salesTaxRate are said to be “passed by value.” Similarly, in the expression

```
shippingCost(itemWeight)
```

the value of itemWeight is passed to the shippingCost function, so itemWeight is said to be passed by value.



## 5.3: Arrays

*Source folders*

*arraySubscript  
swapInts*

*Chapter outline*

- Declaring a sequence of elements
- Accessing elements via the array subscript operator
- Creating functions with array parameters
- Implementing bubble sort

An “array” is a sequence of elements of a given type. The program in this chapter prompts the user to enter 5 integers, stores them in an array, and sorts the array in ascending order. A sample run is

```
Enter a number (integer): 3
Enter a number (integer): 1
Enter a number (integer): 5
Enter a number (integer): 4
Enter a number (integer): 2
```

```
Performing bubble sort...
```

```
3 1 5 4 2
If 3 > 1, then swap
1 3 5 4 2
If 3 > 5, then swap
1 3 5 4 2
If 5 > 4, then swap
1 3 4 5 2
If 5 > 2, then swap
1 3 4 2 5
If 1 > 3, then swap
1 3 4 2 5
If 3 > 4, then swap
1 3 4 2 5
If 4 > 2, then swap
1 3 2 4 5
If 1 > 3, then swap
1 3 2 4 5
If 3 > 2, then swap
1 2 3 4 5
If 1 > 2, then swap
1 2 3 4 5
```

```
The sorted sequence is:
1 2 3 4 5
```

The syntax for declaring an array is

```
T arrayName[size];
```

where T is the element type and size is the number of elements. Lines 15-16 (main.cpp),

```
const int totalNumbers = 5;
int numbers[totalNumbers];
```

for example, declare an array called numbers, which contains 5 elements of type int. Although the declaration could have been written as

```
int numbers[5];
```

the size variable totalNumbers is used to prevent “5” from being repeated throughout the code. Note, however, that when declaring an array in this manner, the size variable must be a constant (recall the “const” keyword, which was introduced in Chapter 4.3).

Each element in an array is a variable, accessed by its index number. Indexing begins at 0, so the elements in numbers are

```
numbers[0]      // The first element ("The element at index 0")
numbers[1]
numbers[2]
numbers[3]
numbers[4]      // The last element ("The element at index 4")
```

The [] is called the “array subscript operator.” Accessing an out-of-bounds element, such as

```
numbers[-1]
numbers[17]
```

is undefined.

A sequence of elements is customarily specified in terms of its first and “one-past-the-last” elements, using set builder notation. The entire set of elements in numbers (elements 0 through 4), for example, is

```
[numbers[0], numbers[5])
```

The left bracket indicates that the set includes element 0, while the right parenthesis indicates that the set does not include element 5. Similarly, the set

```
[numbers[2], numbers[4])
```

includes elements 2 and 3, but not 4.

Once the array has been constructed, the loop (lines 18-22),

```

for (int i = 0; i != totalNumbers; ++i)
{
    cout << "Enter a number (integer): ";
    cin >> numbers[i];
}

```

performs 5 iterations, storing the user's input values in [numbers[0], numbers[5]). Line 25,

```
printArray(numbers, totalNumbers);
```

then prints the entire sequence by calling the function (line 7)

```

void printArray(int a[], int size); // Print each element in an array
                                    // of ints a, containing size elements,
                                    // i.e. print the range [a[0], a[size])

```

The function definition is (lines 48-56)

```

void printArray(int a[], int size)
{
    using namespace std;

    for (int i = 0; i != size; ++i)
        cout << a[i] << ' ';

    cout << endl;
}

```

The loop (lines 52-53),

```
for (int i = 0; i != size; ++i)
    cout << a[i] << ' ';
```

performs 5 iterations, printing the values of [numbers[0], numbers[5]). Note that whenever an array is passed to a function (as in line 25), it is automatically passed by reference; it is not possible to pass an array by value. Furthermore, array parameters (such as a), unlike pointer parameters, do not require dereferencing. The reasons for this will be discussed in the next chapter.

Lines 27-38 employ a “bubble sort” algorithm to rearrange the values of [numbers[0], numbers[5]) from least to greatest. Bubble sorting gets its name from the fact that the largest value gradually rises to the top, like a bubble in a glass of water. Once the largest value reaches the top, the second largest value is moved towards the top, followed by the third largest, etc. until the entire sequence is sorted.

Suppose, for example, that the user enters the sequence {3, 1, 5, 4, 2}:

index	0	1	2	3	4	
value	3	1	5	4	2	

The algorithm works by traversing the sequences

```
[numbers[0], numbers[4]) (elements 0 to 3)
[numbers[0], numbers[3]) (elements 0 to 2)
[numbers[0], numbers[2]) (elements 0 to 1)
[numbers[0], numbers[1]) (element 0)
```

In each pass, the algorithm compares successive pairs of elements and swaps them if the first element is greater than the second:

```
Pass 1: Traverse [numbers[0], numbers[4]) (elements 0 to 3)      // end = 3
  If numbers[0] > numbers[1], then swap them                      // i = 0
  If numbers[1] > numbers[2], then swap them                      // i = 1
  If numbers[2] > numbers[3], then swap them                      // i = 2
  If numbers[3] > numbers[4], then swap them                      // i = 3
```

At the end of Pass 1, numbers[4] will have the largest value.

```
Pass 2: Traverse [numbers[0], numbers[3]) (elements 0 to 2)      // end = 2
  If numbers[0] > numbers[1], then swap them                      // i = 0
  If numbers[1] > numbers[2], then swap them                      // i = 1
  If numbers[2] > numbers[3], then swap them                      // i = 2
```

At the end of Pass 2, numbers[3] will have the second-largest value.

```
Pass 3: Traverse [numbers[0], numbers[2]) (elements 0 to 1)      // end = 1
  If numbers[0] > numbers[1], then swap them                      // i = 0
  If numbers[1] > numbers[2], then swap them                      // i = 1
```

At the end of Pass 3, numbers[2] will have the third-largest value.

```
Pass 4: Traverse [numbers[0], numbers[1]) (element 0)           // end = 0
  If numbers[0] > numbers[1], then swap them                      // i = 0
```

At the end of Pass 4, numbers[1] will have the fourth-largest value. Because there are 5 elements altogether, numbers[0] will be left with the fifth-largest (i.e. the smallest) value.

The algorithm is implemented using a nested loop (lines 27-38):

```
for (int end = totalNumbers - 2; end >= 0; --end)
{
    for (int i = 0; i <= end; ++i)
    {
        cout << "If " << numbers[i] << " > " << numbers[i + 1] << ", then swap\n";
        if (numbers[i] > numbers[i + 1])
            dss::swap(&numbers[i], &numbers[i + 1]);
        printArray(numbers, totalNumbers);
    }
}
```

Below is a walkthrough of how the algorithm sorts the array {3, 1, 5, 4, 2}. The print statements (lines

31, 36) have been omitted:

```
int end = totalNumbers - 2;

    end
index | 0 | 1 | 2 | 3 | 4 |
value | 3 | 1 | 5 | 4 | 2 |

end >= 0                                // True
```

---

Pass 1 (4 iterations, for i = 0, 1, 2, and 3)

```
int i = 0;

    i      end
index | 0 | 1 | 2 | 3 | 4 |
value | 3 | 1 | 5 | 4 | 2 |

i <= end                                // True

if (numbers[i] > numbers[i + 1])
    dss::swap(&numbers[i], &numbers[i + 1]);    // Swap 3 and 1

    i      end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 5 | 4 | 2 |

++i;

    i      end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 5 | 4 | 2 |

i <= end                                // True

if (numbers[i] > numbers[i + 1])
    dss::swap(&numbers[i], &numbers[i + 1]);    // Not executed

++i;

    i   end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 5 | 4 | 2 |

i <= end                                // True

if (numbers[i] > numbers[i + 1])
    dss::swap(&numbers[i], &numbers[i + 1]);    // Swap 5 and 4

    i   end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 4 | 5 | 2 |
```

```

++i;

          i
          end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 4 | 5 | 2 |

i <= end                                // True

if (numbers[i] > numbers[i + 1])
    dss::swap(&numbers[i], &numbers[i + 1]);      // Swap 5 and 2

          i
          end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 4 | 2 | 5 |

++i;

          end   i
          end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 4 | 2 | 5 |

i <= end                                // False

Terminate nested loop
Pass 1 is complete
The largest element (5) is now at the correct position

```

---

```

--end;

          end
          end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 4 | 2 | 5 |

end >= 0                                // True

```

---

```

Pass 2 (3 iterations, for i = 0, 1, and 2)

int i = 0;

          i      end
          end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 4 | 2 | 5 |

i <= end                                // True

if (numbers[i] > numbers[i + 1])
    dss::swap(&numbers[i], &numbers[i + 1]);      // Not executed

++i;

```

```

          i   end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 4 | 2 | 5 |

i <= end                                // True

if (numbers[i] > numbers[i + 1])
    dss::swap(&numbers[i], &numbers[i + 1]);      // Not executed

++i;

          i
          end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 4 | 2 | 5 |

i <= end                                // True

if (numbers[i] > numbers[i + 1])
    dss::swap(&numbers[i], &numbers[i + 1]);      // Swap 4 and 2

          i
          end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 2 | 4 | 5 |

++i;

          end   i
          index | 0 | 1 | 2 | 3 | 4 |
          value | 1 | 3 | 2 | 4 | 5 |

i <= end                                // False

Terminate nested loop
Pass 2 is complete
The second-largest element (4) is now at the correct position


---


--end;

          end
          index | 0 | 1 | 2 | 3 | 4 |
          value | 1 | 3 | 2 | 4 | 5 |

end >= 0                                  // True


---


Pass 3 (2 iterations, for i = 0 and 1)

int i = 0;

```

```

        i   end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 2 | 4 | 5 |

i <= end                                // True

if (numbers[i] > numbers[i + 1])
    dss::swap(&numbers[i], &numbers[i + 1]);      // Not executed

++i;

        i
        end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 3 | 2 | 4 | 5 |

i <= end                                // True

if (numbers[i] > numbers[i + 1])
    dss::swap(&numbers[i], &numbers[i + 1]);      // Swap 3 and 2

        i
        end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 2 | 3 | 4 | 5 |

++i;

        end   i
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 2 | 3 | 4 | 5 |

i <= end                                // False

Terminate nested loop
Pass 3 is complete
The third-largest element (3) is now at the correct position


---


--end;

        end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 2 | 3 | 4 | 5 |

end >= 0                                  // True


---


Pass 4 (1 iteration, for i = 0)

int i = 0;

```

```

    i
  end
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 2 | 3 | 4 | 5 |

i <= end                                // True

if (numbers[i] > numbers[i + 1])
  dss::swap(&numbers[i], &numbers[i + 1]);      // Not executed

++i;

    end   i
index | 0 | 1 | 2 | 3 | 4 |
value | 1 | 2 | 3 | 4 | 5 |

i <= end                                // False

Terminate nested loop
Pass 4 is complete
The fourth-largest element (2) is now at the correct position

Because the sequence contains 5 elements altogether,
the remaining element (i.e. the fifth-largest element, 1) is also
at the correct position

```



## 5.4: Pointer Arithmetic

*Source folders*

```
pointerArithmetic
swapInts
```

*Chapter outline*

- Using pointers to traverse arrays

The program in this chapter is functionally identical to that of the previous chapter, but is implemented using pointer arithmetic.

Recall from Chapter 5.1 that each variable in a program is constructed at a specific memory address and occupies a fixed number of bytes. An array is a block of consecutive bytes, where each element is adjacent to the next one in the sequence. Consider a hypothetical architecture where each int occupies 2 bytes of memory, and addresses are represented using base-10 values. The statement

```
int a[4]; // Construct a, an array of 4 ints
```

might result in the memory layout

01	02	03	04	05	06	07	08	09	10	
11	12	13	14	15	16	17	18	19	20	
		a[0]		a[1]		a[2]		a[3]		

```
// Element a[0] is located at address 12
// Element a[1] is located at address 14
// Element a[2] is located at address 16
// Element a[3] is located at address 18
```

The name of an array, when written without a subscript, is a pointer to the first element in that array. The expression

```
a // address 12
```

for example, returns a pointer to element a[0]. Element a[0] itself can therefore be obtained by dereferencing the pointer, as in

```
*a // Equivalent to a[0] ("The int at address a")
```

A pointer to any other element can be obtained by applying an integer offset, which represents the distance (number of elements) to that element. The expression

```
a + 1 // address 12, offset 1 element (2 bytes) = address 14
```

for example, returns a pointer to element  $a[1]$ , while the expression

```
a + 2      // address 12, offset 2 elements (4 bytes) = address 16
```

returns a pointer to element  $a[2]$ . Again note that the integer offset represents the number of elements, not bytes; the conversion from the number of elements to bytes is performed automatically.

The array subscript operator, introduced in the previous chapter, is actually a shorthand notation for a dereferenced pointer. The expression

```
a[1]          // Element a[1] (1 element from a[0])
```

for example, is shorthand for

```
* (a + 1)    // The int at address a (12), offset 1 element,
// i.e. the int at address 14
```

Similarly, the expression

```
a[2]          // Element a[2] (2 elements from a[0])
```

is shorthand for

```
* (a + 2)    // The int at address a (12), offset 2 elements,
// i.e. the int at address 16
```

Incrementing a pointer will seat it (make it point) to the next element, while decrementing a pointer will seat it to the previous element:

```
int* p = a + 2;      // p points to element a[2]
int* q = a + 1;      // q points to element a[1]

++p;                // p now points to element a[3]
--q;                // q now points to element a[0]
```

Pointers also support the addition assignment and subtraction assignment operators:

```
p -= 2;            // p now points to element a[1]
q += 2;            // q now points to element a[2]
```

Dereferencing a pointer returns the referent element:

```
*p = 37;           // a[1] = 37;
*q = 41;           // a[2] = 41;
```

Address values are comparable. In the above memory layout,

```
nullptr < address 01 < address 02 < address 03 < address 04 < ...
```

Two pointers (of the same type) are therefore directly comparable:

```
p < q           // true (address 14 < address 16)
q > p           // true (address 16 > address 14)

p != nullptr    // true (address 14 != nullptr)
q != nullptr    // true (address 16 != nullptr)

p > nullptr     // true (address 14 > nullptr)
q > nullptr     // true (address 16 > nullptr)

p == nullptr    // false (address 14 != nullptr)
q == nullptr    // false (address 16 != nullptr)

p < nullptr     // false (address 14 > nullptr)
q < nullptr     // false (address 16 > nullptr)
```

Subtracting one pointer from another returns the distance (number of elements) between the referent elements:

```
int distanceFromPToQ = q - p;      // distanceFromPToQ is 1 element
                                    // (address 16 - address 14)

int distanceFromQToP = p - q;      // distanceFromQToP is -1 element
                                    // (address 14 - address 16)
```

A pointer can be incremented or decremented to any address, even one outside of an array's bounds:

```
p += 3;          // p now points to address 20 (out-of-bounds)
q -= 3;          // q now points to address 10 (out-of-bounds)
```

Out-of-bounds pointers cannot be dereferenced, but they are still comparable:

```
p > q           // true (address 20 > address 10)
q < p           // true (address 10 < address 20)

p != nullptr    // true (address 20 != nullptr)
q != nullptr    // true (address 10 != nullptr)

p > nullptr     // true (address 20 > nullptr)
q > nullptr     // true (address 10 > nullptr)

p == nullptr    // false (address 20 != nullptr)
q == nullptr    // false (address 10 != nullptr)

p < nullptr     // false (address 20 > nullptr)
q < nullptr     // false (address 10 > nullptr)
```

Suppose that as of lines 15-16 (main.cpp),

```
const int totalNumbers = 5;
int numbers[totalNumbers];
```

the memory layout is

01	02	03	04	05	06	07	08	09	10	
11	12	13	14	15	16	17	18	19	20	

```
// Element numbers[0] is located at address 11
// Element numbers[1] is located at address 13
// Element numbers[2] is located at address 15
// Element numbers[3] is located at address 17
// Element numbers[4] is located at address 19
```

The loop (lines 18-22),

```
for (int* p = numbers; p != numbers + totalNumbers; ++p)
{
    cout << "Enter a number (integer): ";
    cin >> *p;
}
```

performs 5 iterations:

---

```
int* p = numbers;                                // p points to numbers[0]
                                                // (address 11)
```

---

```
p != numbers + totalNumbers           // p != address 21, true
```

Iteration 1:

```
cout << "Enter a number (integer): ";
cin >> *p;                                     // cin >> numbers[0];

++p                                              // p points to numbers[1]
                                                // (address 13)
```

---

```
p != numbers + totalNumbers           // p != address 21, true
```

Iteration 2:

```
cout << "Enter a number (integer): ";
cin >> *p;                                     // cin >> numbers[1];

++p                                              // p points to numbers[2]
                                                // (address 15)
```

---

```
p != numbers + totalNumbers           // p != address 21, true
```

Iteration 3:

```

cout << "Enter a number (integer): ";
cin >> *p;                                // cin >> numbers[2];

++p                                         // p points to numbers[3]
// (address 17)

```

---

```
p != numbers + totalNumbers           // p != address 21, true
```

Iteration 4:

```

cout << "Enter a number (integer): ";
cin >> *p;                                // cin >> numbers[3];

++p                                         // p points to numbers[4]
// (address 19)

```

---

```
p != numbers + totalNumbers           // p != address 21, true
```

Iteration 5:

```

cout << "Enter a number (integer): ";
cin >> *p;                                // cin >> numbers[4];

++p                                         // p points to numbers[5]
// (address 21, out-of-bounds)

```

---

```
p != numbers + totalNumbers           // p != address 21, false
```

Terminate loop

Line 25,

```
printArray(numbers, totalNumbers);
```

calls the printArray function, passing a pointer to numbers[0] (address 11) and the array size (5). The function has been rewritten as (line 7)

```

void printArray(int* a, int size);          // a is a pointer to the first
                                            // element of an array of ints,
                                            // containing size elements

```

The function is defined in lines 46-56. For the function call in line 25, the loop (lines 52-53)

```

for (int* p = a; p != a + size; ++p)
    cout << *p << ' ';

```

thus becomes

```
for (int* p = numbers; p != numbers + totalNumbers; ++p)
    cout << *p << ' ';
```

which behaves identically to the loop in lines 18-22; the only difference is that in each iteration, this loop simply prints the value of the referent element (\*p).

The bubble sort algorithm has also been reimplemented, using pointers instead of array subscripting (lines 27-38):

```
for (int* end = numbers + totalNumbers - 2; end >= numbers; --end)
{
    for (int* p = numbers; p <= end; ++p)
    {
        cout << "If " << *p << " > " << *(p + 1) << ", then swap\n";
        if (*p > *(p + 1))
            dss::swap(p, p + 1);

        printArray(numbers, totalNumbers);
    }
}
```

As in the previous chapter, the algorithm can be summarized as

Pass 1: Traverse [numbers[0], numbers[4]) (elements 0-3)	// end = numbers + 3
If numbers[0] > numbers[1], then swap them	// p = numbers + 0
If numbers[1] > numbers[2], then swap them	// p = numbers + 1
If numbers[2] > numbers[3], then swap them	// p = numbers + 2
If numbers[3] > numbers[4], then swap them	// p = numbers + 3
Pass 2: Traverse [numbers[0], numbers[3]) (elements 0-2)	// end = numbers + 2
If numbers[0] > numbers[1], then swap them	// p = numbers + 0
If numbers[1] > numbers[2], then swap them	// p = numbers + 1
If numbers[2] > numbers[3], then swap them	// p = numbers + 2
Pass 3: Traverse [numbers[0], numbers[2]) (elements 0-1)	// end = numbers + 1
If numbers[0] > numbers[1], then swap them	// p = numbers + 0
If numbers[1] > numbers[2], then swap them	// p = numbers + 1
Pass 4: Traverse [numbers[0], numbers[1]) (element 0)	// end = numbers + 0
If numbers[0] > numbers[1], then swap them	// p = numbers + 0

Below is a walkthrough of how the algorithm sorts the array {3, 1, 5, 4, 2}. The print statements (lines 31, 36) have been omitted.

```
int* end = numbers + totalNumbers - 2;
                    end
| 3 | 1 | 5 | 4 | 2 |
end >= numbers // True
```

---

```

Pass 1 (4 iterations)

int* p = numbers;

    p      end
| 3 | 1 | 5 | 4 | 2 |

p <= end          // True

if (*p > *(p + 1))
    dss::swap(p, p + 1);           // Swap 3 and 1

    p      end
| 1 | 3 | 5 | 4 | 2 |

++p;

    p      end
| 1 | 3 | 5 | 4 | 2 |

p <= end          // True

if (*p > *(p + 1))
    dss::swap(p, p + 1);           // Not executed

++p;

    p      end
| 1 | 3 | 5 | 4 | 2 |

p <= end          // True

if (*p > *(p + 1))
    dss::swap(p, p + 1);           // Swap 5 and 4

    p      end
| 1 | 3 | 4 | 5 | 2 |

++p;

    p
end
| 1 | 3 | 4 | 5 | 2 |

p <= end          // True

if (*p > *(p + 1))
    dss::swap(p, p + 1);           // Swap 5 and 2

    p
end
| 1 | 3 | 4 | 2 | 5 |

++p;

```

```

    end   p
| 1 | 3 | 4 | 2 | 5 |

p <= end                                // False

Terminate nested loop
Pass 1 is complete
The largest element (5) is now at the correct position


---


--end;

    end
| 1 | 3 | 4 | 2 | 5 |

end >= numbers                            // True


---


Pass 2 (3 iterations)

int* p = numbers;

    p      end
| 1 | 3 | 4 | 2 | 5 |

p <= end                                // True

if (*p > *(p + 1))
    dss::swap(p, p + 1);                  // Not executed

++p;

    p      end
| 1 | 3 | 4 | 2 | 5 |

p <= end                                // True

if (*p > *(p + 1))
    dss::swap(p, p + 1);                  // Not executed

++p;

    p
    end
| 1 | 3 | 4 | 2 | 5 |

p <= end                                // True

if (*p > *(p + 1))
    dss::swap(p, p + 1);                  // Swap 4 and 2

    p
    end
| 1 | 3 | 2 | 4 | 5 |

```

```

++p;

    end   p
| 1 | 3 | 2 | 4 | 5 |

p <= end                                // False

Terminate nested loop
Pass 2 is complete
The second-largest element (4) is now at the correct position

```

---

```

--end;

    end
| 1 | 3 | 2 | 4 | 5 |

end >= numbers                            // True

```

---

```

Pass 3 (2 iterations)

int* p = numbers;

    p   end
| 1 | 3 | 2 | 4 | 5 |

p <= end                                // True

if (*p > *(p + 1))
    dss::swap(p, p + 1);                  // Not executed

++p;

    p
end
| 1 | 3 | 2 | 4 | 5 |

p <= end                                // True

if (*p > *(p + 1))
    dss::swap(p, p + 1);                  // Swap 3 and 2

    p
end
| 1 | 2 | 3 | 4 | 5 |

++p;

    end   p
| 1 | 2 | 3 | 4 | 5 |

p <= end                                // False

```

```

Terminate nested loop
Pass 3 is complete
The third-largest element (3) is now at the correct position

```

---

```

--end;

    end
| 1 | 2 | 3 | 4 | 5 |

end >= numbers           // True

```

---

Pass 4 (1 iteration)

```

int* p = numbers;

    p
end
| 1 | 2 | 3 | 4 | 5 |

p <= end           // True

if (*p > *(p + 1))
    dss::swap(p, p + 1);           // Not executed

++p;

```

```

    end   p
| 1 | 2 | 3 | 4 | 5 |

```

```

p <= end           // False

```

```

Terminate nested loop
Pass 4 is complete
The fourth-largest element (2) is now at the correct position

```

Because the sequence contains 5 elements altogether,  
the remaining element (i.e. the fifth-largest element, 1) is also  
at the correct position

---

```

--end;

    end
|     | 1 | 2 | 3 | 4 | 5 |

end >= numbers           // False

Terminate main loop

```

The semantics of prefix / postfix increment (and decrement, discussed in Chapters 2.3 and 2.4) also apply to pointers. If *p* points to an element in an array, the expression

```
++p      // Prefix increment
```

points p to the next element and returns p. The expression

```
*++p      // Prefix increment and dereference
```

therefore points p to the next element, then accesses that element via the dereference operator (\*). The expression

```
p++      // Postfix increment
```

points p to the next element, but returns a pointer to p's original referent. The expression

```
*p++      // Postfix increment and dereference
```

therefore points p to the next element, but accesses p's original referent.

C++ inherited this syntax from C, where it was originally used to write more concise code when traversing arrays. Because it can be more error-prone and difficult to understand, however, it is largely avoided throughout this book. Consider, for example, the array of ints

```
|p 7 | 4 | 6 | 3 | 8 | 5 | 9 |
```

where p is a pointer (int\*) to the first element, 7. The statement

```
int* q = ++p;
```

is equivalent to

```
++p;
int* q = p;
```

```
|p 7 | 4 | 6 | 3 | 8 | 5 | 9 |
     q
```

Similarly, the statement

```
int* r = p++;
```

is equivalent to

```
int* r = p;
++p;
```

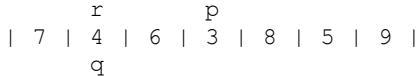
```
|r 7 | 4 | 6 | 3 | 8 | 5 | 9 |
     p
     q
```

### The statement

```
cout << *++p;      // Point p to the next element (3),
// then print the referent of p (3)
```

is equivalent to

```
+p;
cout << *p;
```



### The statement

```
cout << *p++;      // Print the referent of p (3),
// then point p to the next element (8)
```

is equivalent to

```
cout << *p;
++p;
```



### The statement

```
*++p = 0;          // Point p to the next element (5),
// then set the referent of p to 0
```

is equivalent to

```
+p;
*p = 0;
```



### The statement

```
*p++ = 1;          // Set the referent of p to 1,
// then point p to the next element (9)
```

is equivalent to

```
*p = 1;
++p;
```

```
| 7 | 4 | 6 | 3 | 8 | 1 | 9 |
    r   p
    q
```

The loop in the printArray function (lines 52-53),

```
for (int* p = a; p != a + size; ++p)
    cout << *p << ' ';
```

for example, could have been written as

```
int* p = a;
while (p != a + size)
    cout << *p++ << ' ';      // Print the referent of p, point p to the next
                                // element, then print a whitespace
```

As mentioned above, this syntax is largely avoided throughout this book, but a basic understanding of it is necessary to implement custom iterators, which will be discussed in later sections.



## 5.5: References

*Source folder: references*

*Chapter outline*

- How references differ from pointers
- Creating functions with reference parameters

The program in this chapter introduces references and illustrates how they can be used to create functions that modify their arguments. A sample run is

```
Enter player's first name: Michael
Enter player's score: 84
```

```
Enter player's first name: Catherine
Enter player's score: 87
```

```
Enter player's first name: James
Enter player's score: 80
```

```
Enter player's first name: Sarah
Enter player's score: 95
```

```
Enter player's first name: Edward
Enter player's score: 92
```

```
The high score of 95 belongs to Sarah
```

A “reference” is an alias, or alternate name, of an existing variable. Given

```
double d = 3.14159;
```

for example, the statement

```
double& r = d;
```

declares and initializes the variable r, of type double&. The &, which denotes “reference to” is called the “reference operator,” and the type double& is read right-to-left as “reference to double.” While the value of a double is a real number and the value of a double\* is the address of a double, a double& is an alias of an existing double. The above statement declares r as an alias (alternate name) of d. r is said to be a reference to d, and d is the referent of r. The statements

```
r = 1.618;
cout << r << endl;
++r;
double* p = &r;
```

are equivalent to

```
d = 1.618;
cout << d << endl;
++d;
double* p = &d;
```

The following properties of references make them easier to use and less error-prone than pointers, but also less powerful:

- Initializing a reference does not require taking the address of the referent:

```
int i = 7;
int* p = &i;      // p points to i (requires the &)
int& r = i;      // r refers to i (does not require the &)
```

- Accessing the referent of a reference does not require dereferencing:

```
*p = 19;          // Access the referent of p (requires the *)
cout << *p;

r = 23;          // Access the referent of r (does not require the *)
cout << r;
```

- A reference cannot be uninitialized:

```
string* p;      // Declare a pointer without a referent (ok)
string& r;      // Declare a reference without a referent (compiler error)
```

- A reference cannot be null:

```
double* p = nullptr;    // p is a null pointer (ok),
                        // but there is no equivalent "nullref" value for
                        // specifying a "null reference"
```

- A reference cannot be reseated:

```
string a = "Dorian Gray";
string b = "Basil Hallward";

string* p = &a;    // p points to a
p = &b;          // p now points to b

string& r = a;    // r refers to a (r is an alias of a)
                  // r cannot be made to have a new referent

r = b;          // a = b;
                  // Sets the value of a to "Basil Hallward";
                  // does not make r refer to b
```

Lines 10-14,

```
const int totalPlayers = 5;
```

```
string names[totalPlayers];
int scores[totalPlayers];
int highScoreIndex = 0;
```

construct names, an array of 5 strings, scores, an array of 5 ints, and highScoreIndex, an int used to track the index of the highest score. Note that the function (line 4)

```
void getNameAndScore(std::string& name, int& score);
```

uses reference parameters: name is a reference to a string (std::string&) and score is a reference to an int (int&). The function prompts the user to enter a string value and an int value, then stores those values in the referents of name and score (lines 30-39):

```
void getNameAndScore(std::string& name, int& score)
{
    using namespace std;

    cout << "\nEnter player's first name: ";
    cin >> name;

    cout << "Enter player's score: ";
    cin >> score;
}
```

### The function call

```
getNameAndScore(names[2], scores[2]);
```

for example, writes the user's input values to the elements names[2] and scores[2]. name is an alias of the element names[2], and score is an alias of the element scores[2]. If this function were implemented using pointers, the definition would be

```
void getNameAndScore(std::string* name, int* score)
{
    using namespace std;

    cout << "\nEnter player's first name: ";
    cin >> *name;

    cout << "Enter player's score: ";
    cin >> *score;
}
```

and the equivalent function call would be

```
getNameAndScore(&names[2], &scores[2]);      // Pass the addresses of names[2]
                                                // and scores[2]
```

or

```
getNameAndScore(names + 2, scores + 2);
```

## The loop (lines 16-22)

```

for (int i = 0; i != totalPlayers; ++i)
{
    getNameAndScore(names[i], scores[i]);

    if (scores[i] > scores[highScoreIndex])
        highScoreIndex = i;
}

```

performs 5 iterations. Each iteration records the name and score of the current player (line 18). If the current player's score is greater than the high score (line 20), then highScoreIndex is set to the current player's index value (line 21). Suppose, for example, that the user enters the values

names	scores	index
Michael	84	0
Catherine	87	1
James	80	2
Sarah	95	3
Edward	92	4

Michael	84	0
Catherine	87	1
James	80	2
Sarah	95	3
Edward	92	4

```

int highScoreIndex = 0;                                // highScoreIndex is 0 (line 14)
int i = 0;                                            // i is 0

```

---

i != totalPlayers	// True
-------------------	---------

Iteration 1:

getNameAndScore(names[i], scores[i]);	// Record names[0] and scores[0]
if (scores[i] > scores[highScoreIndex])	// Compare scores[0] and scores[0]
highScoreIndex = i;	// Not executed

```

++i;                                              // i is 1

```

---

i != totalPlayers	// True
-------------------	---------

Iteration 2:

getNameAndScore(names[i], scores[i]);	// Record names[1] and scores[1]
if (scores[i] > scores[highScoreIndex])	// Compare scores[1] and scores[0]
highScoreIndex = i;	// Set highScoreIndex to 1

```

++i;                                              // i is 2

```

---

i != totalPlayers	// True
-------------------	---------

Iteration 3:

```
getNameAndScore(names[i], scores[i]);      // Record names[2] and scores[2]
if (scores[i] > scores[highScoreIndex])    // Compare scores[2] and scores[1]
    highScoreIndex = i;                    // Not executed
++i;                                       // i is now 3
```

---

```
i != totalPlayers                         // True
```

Iteration 4:

```
getNameAndScore(names[i], scores[i]);      // Record names[3] and scores[3]
if (scores[i] > scores[highScoreIndex])    // Compare scores[3] and scores[1]
    highScoreIndex = i;                    // Set highScoreIndex to 3
++i;                                       // i is 4
```

---

```
i != totalPlayers                         // True
```

Iteration 5:

```
getNameAndScore(names[i], scores[i]);      // Record names[4] and scores[4]
if (scores[i] > scores[highScoreIndex])    // Compare scores[4] and scores[3]
    highScoreIndex = i;                    // Not executed
++i;                                       // i is 5
```

---

```
i != totalPlayers                         // False
```

Terminate loop

When the loop terminates, highScoreIndex is 3, so lines 24-25,

```
cout << "\nThe high score of " << scores[highScoreIndex] <<
    " belongs to " << names[highScoreIndex] << endl;
```

generate the output

```
The high score of 95 belongs to Sarah
```

The following tables summarize the differences between pointers and references, discussed earlier in this chapter:

	Requires address-taking (&)	Requires dereferencing (*)	
Pointer Reference	Yes No	Yes No	
	Can be declared w/o referent	Can be null	Can be reseated
Pointer Reference	Yes No	Yes No	Yes No

Although references are much less powerful than pointers, they play an important role in classes and operator overloading, which will be discussed in later chapters.

## 5.6: Const Correctness

*Source folder: constCorrectness*

*Chapter outline*

- *Const pointers and const references*
- *When to use const pointer and const reference parameters*
- *Implementing insertion sort*

A “const pointer” behaves identically to a pointer, but cannot be used to modify its referent:

```
double a[5];

// Assume that the values of [a[0], a[5]) have already been set...

const double* cp = a;      // cp, a const pointer to a double, points to
                           // element a[0]

cout << *cp << endl;      // Read the value of cp's referent (ok)

++cp;                      // Point to the next element, a[1] (ok)
--cp;                      // Point to the previous element, a[0] (ok)
cp += 4;                   // Point to element a[4] (ok)
cp -= 2;                   // Point to element a[2] (ok)

*cp = 2.718;                // Compiler error: a const pointer cannot be used
                           // to modify its referent
```

Similarly, a “const reference” behaves identically to a reference, but cannot be used to modify its referent:

```
string s = "ornithology";

const string& cr = s;        // cr, a const reference to a string, refers
                           // to s

cout << cr << endl;        // Read the value of cr's referent (ok)

cr = "entomology";          // Compiler error: a const reference cannot be
                           // used to modify its referent
```

If a pointer or reference parameter does not modify its referent, it should be declared `const`. Adherence to this principle is called “const correctness.” The function

```
void printElement(const std::string& s)    // Print [<s>], followed by a
{                                              // whitespace
    std::cout << '[' << s << ']' << ' ';
```

for example, is said to be “const correct”: the function does not modify the referent of s, so s is declared as a const reference. The function

```
void printElement(std::string& s)
{
    std::cout << '[' << s << ']' << ' ';
```

however, is not const correct: the parameter type, std::string&, implies that the function modifies the referent of s, when in reality it does not. The function

```
void swap(std::string& a, std::string& b)           // Swap the values of the
{                                                 // referents of a and b
    std::string c = a;
    a = b;
    b = c;
}
```

on the other hand, is const correct: the parameter types, std::string&, imply that the function modifies the referents of a and b, and it indeed does.

Though not strictly necessary, const correctness reduces the possibility of programming errors. It allows the compiler to detect bugs, such as

```
void printElement(const std::string& s)      // s is a const reference
{
    std::cout << '[' << s << ']' << ' ';

    s = "bug";                                // Compiler error: cannot modify
                                                // the referent of s

    std::string t = "gotcha";
    swap(s, t);                               // Compiler error: cannot convert
                                                // a const reference (s) to a
                                                // reference (a, the parameter in
                                                // the swap function)
}
```

It also provides a way of documenting code, letting other users know which functions can modify their arguments:

```
void printElement(const std::string& s);      // The parameter type indicates
                                                // that this function is
                                                // guaranteed to not modify its
                                                // argument

void swap(std::string& a, std::string& b); // The parameter types indicate
                                                // that this function can modify
                                                // its arguments
```

The function (line 5)

```
void printArray(const int* begin, const int* end);
```

prints each element in the set [begin, end), where begin is a pointer to the first element and end is a pointer to the one-past-the-last element. Recall from Chapter 5.3 that in set builder notation, the right parenthesis indicates that end is not included in the set. Because the function does not modify any of the elements, its parameters are const pointers (lines 35-42):

```
void printArray(const int* begin, const int* end)
{
    while (begin != end)
    {
        std::cout << *begin << ' ';
        ++begin;
    }
}
```

The swap function, introduced in Chapter 5.2, has been reimplemented using reference parameters (lines 6, 44-49):

```
void swap(int& a, int& b)      // Swap the values of the referents of a and b
{
    int c = a;
    a = b;
    b = c;
}
```

Lines 15-22 of main,

```
const int totalNumbers = 9;
int numbers[totalNumbers];

for (int i = 0; i != totalNumbers; ++i)
{
    cout << "Enter a number (integer): ";
    cin >> numbers[i];
}
```

prompt the user to enter 9 integers, which are stored in the array numbers. Line 25,

```
insertionSort(numbers, numbers + totalNumbers);
```

passes a pointer to the first element (numbers) and one-past-the-last element (numbers + totalNumbers) to the function (line 7)

```
void insertionSort(int* begin, int* end);
```

which uses an algorithm called “insertion sort” to sort the elements in the set [begin, end). Insertion sort is a more efficient (but slightly more complex) algorithm than the bubble sort introduced in Chapter 5.3 (lines 51-66):

```

void insertionSort(int* begin, int* end)
{
    for (int* current = begin; current != end; ++current)      // Outer loop
    {
        int* y = current;
        int* x = current;
        --x;

        while (y != begin && *y < *x)                                // Inner loop
        {
            swap(*y, *x);
            --y;
            --x;
        }
    }
}

```

The algorithm traverses each element in the array, starting with the first element (begin). In each traversal, the current element is sorted by repeatedly shifting it left until it is not less than its preceding element. Suppose, for example, that the user enters the sequence {2, 8, 5, 9, 7, 3, 6, 4, 1}:

```

// Refer to the function definition (lines 51-66)
// B stands for begin (a pointer to the first element)
// E stands for end (a pointer to the one-past-the-last element)
// c stands for current (a pointer to the current element)

Initializer:
int* current = begin;           // current now points to element 0

B                   E
c
2 8 5 9 7 3 6 4 1

```

---

Traversal 1:

```

int* y = current;
int* x = current;
--x;

c
2 8 5 9 7 3 6 4 1
x y

```

The condition ( $y \neq \text{begin}$ ) returns false, so the body of the inner loop is not executed

Post-Body:  
 $\text{++current};$  // current now points to element 1

---

Traversal 2:

```

int* y = current;
int* x = current;
--x;

      C
2 8 5 9 7 3 6 4 1
x y

*y (8) is not less than *x (2), so the body of the inner loop is not
executed

```

Post-Body:  
`++current; // current now points to element 2`

---

Traversal 3:

```

int* y = current;
int* x = current;
--x;

      C
2 8 5 9 7 3 6 4 1
x y

```

Inner loop:

```

swap(*y, *x);
--y;
--x;

      C
2 5 8 9 7 3 6 4 1
x y

*y (5) is not less than *x (2), so the inner loop terminates

```

Post-Body:  
`++current; // current now points to element 3`

---

Traversal 4:

```

int* y = current;
int* x = current;
--x;

      C
2 5 8 9 7 3 6 4 1
x y

```

`*y (9) is not less than *x (8), so the body of the inner loop is not
executed`

---

```
Post-Body:
++current;                                // current now points to element 4
```

---

Traversal 5:

```
int* y = current;
int* x = current;
--x;
```

```
      C
2 5 8 9 7 3 6 4 1
    x   y
```

Inner loop:

```
swap(*y, *x);
--y;
--x;
```

```
      C
2 5 8 7 9 3 6 4 1
    x   y
```

```
swap(*y, *x);
--y;
--x;
```

```
      C
2 5 7 8 9 3 6 4 1
    x   y
```

\*y (7) is not less than \*x (5), so the inner loop terminates

---

```
Post-Body:
++current;                                // current now points to element 5
```

---

Traversal 6:

```
int* y = current;
int* x = current;
--x;
```

```
      C
2 5 7 8 9 3 6 4 1
    x   y
```

Inner loop:

```
swap(*y, *x);
--y;
--x;
```

C  
2 5 7 8 3 9 6 4 1  
x y

```
swap(*y, *x);
--y;
--x;
```

C  
2 5 7 3 8 9 6 4 1  
x y

```
swap(*y, *x);
--y;
--x;
```

C  
2 5 3 7 8 9 6 4 1  
x y

```
swap(*y, *x);
--y;
--x;
```

C  
2 3 5 7 8 9 6 4 1  
x y

\*y (3) is not less than \*x (2), so the inner loop terminates

Post-Body:  
++current; // current now points to element 6

---

Traversal 7:

```
int* y = current;
int* x = current;
--x;
```

C  
2 3 5 7 8 9 6 4 1  
x y

Inner loop:

```
swap(*y, *x);
--y;
--x;
```

C  
2 3 5 7 8 6 9 4 1  
x y

```
swap(*y, *x);
--y;
--x;
```

C  
2 3 5 7 6 8 9 4 1  
x y

```
swap(*y, *x);
--y;
--x;
```

C  
2 3 5 6 7 8 9 4 1  
x y

\*y (6) is not less than \*x (5), so the inner loop terminates

**Post-Body:**  
++current; // current now points to element 7

---

Traversal 8:

```
int* y = current;
int* x = current;
--x;
```

C  
2 3 5 6 7 8 9 4 1  
x y

Inner loop:

```
swap(*y, *x);
--y;
--x;
```

C  
2 3 5 6 7 8 4 9 1  
x y

```
swap(*y, *x);
--y;
--x;
```

C  
2 3 5 6 7 4 8 9 1  
x y

```
swap(*y, *x);
--y;
--x;
```

```

          C
2 3 5 6 4 7 8 9 1
      x  y

swap(*y, *x);
--y;
--x;

          C
2 3 5 4 6 7 8 9 1
      x  y

swap(*y, *x);
--y;
--x;

          C
2 3 4 5 6 7 8 9 1
      x  y

*y (4) is not less than *x (3), so the inner loop terminates

Post-Body:
++current;           // current now points to element 8

```

---

## Traversal 9:

```

int* y = current;
int* x = current;
--x;

```

```

          C
2 3 4 5 6 7 8 9 1
      x  y

```

## Inner loop:

```

swap(*y, *x);
--y;
--x;

```

```

          C
2 3 4 5 6 7 8 1 9
      x  y

```

```

swap(*y, *x);
--y;
--x;

```

```

          C
2 3 4 5 6 7 1 8 9
      x  y

```

```

swap(*y, *x);

```

```
--y;
--x;

      C
2 3 4 5 6 1 7 8 9
    x  y

swap(*y, *x);
--y;
--x;

      C
2 3 4 5 1 6 7 8 9
    x  y

swap(*y, *x);
--y;
--x;

      C
2 3 4 1 5 6 7 8 9
    x  y

swap(*y, *x);
--y;
--x;

      C
2 3 1 4 5 6 7 8 9
    x  y

swap(*y, *x);
--y;
--x;

      C
2 1 3 4 5 6 7 8 9
    x  y

swap(*y, *x);
--y;
--x;

      C
1 2 3 4 5 6 7 8 9
    x  y
```

The condition (`y != begin`) returns false, so the inner loop terminates

---

<code>Post-Body:</code>	<code>++current;</code>	<code>// current now points to element 9</code>
-------------------------	-------------------------	---

```
B           E  
C  
1 2 3 4 5 6 7 8 9
```

The condition (current != end) returns false, so the outer loop terminates

After the array is sorted, lines 27-28,

```
cout << "\nThe sorted sequence is:\n";  
printArray(numbers, numbers + totalNumbers);
```

print the array.

A sample run of the program is

```
Enter a number (integer): 2  
Enter a number (integer): 8  
Enter a number (integer): 5  
Enter a number (integer): 9  
Enter a number (integer): 7  
Enter a number (integer): 3  
Enter a number (integer): 6  
Enter a number (integer): 4  
Enter a number (integer): 1
```

Performing insertion sort...

The sorted sequence is:

```
1 2 3 4 5 6 7 8 9
```



# Part 6: Classes and Operator Overloading

## 6.1: Classes

*Source folder: classes*

*Chapter outline:*

- Creating a custom datatype
- Implementing a default constructor and member functions (const / non-const)
- Calling member functions on an object through a pointer / reference

A “class” is a custom datatype. Consider, for example, the Student class (Student.h, lines 8-26):

```
class Student
{
public:
    Student();

    std::string firstName() const;
    std::string lastName() const;
    int testScore() const;
    char letterGrade() const;

    void setName(const std::string& firstName, const std::string& lastName);
    void setTestScore(int testScore);

private:
    std::string _firstName;
    std::string _lastName;
    int _testScore;
    char _letterGrade;
};
```

The class is declared in namespace dss (lines 6-7, 54). Note that the closing brace of the class definition (lines 8-9, 26) also requires a semicolon. Lines 21-25,

```
private:
    std::string _firstName;
    std::string _lastName;
    int _testScore;
    char _letterGrade;
```

specify that each “object” (variable) of type Student contains 4 variables: \_firstName, \_lastName (of type string), \_testScore (of type int), and \_letterGrade (of type char). The “private” keyword indicates that these variables, known as an object’s “data members,” are only accessible through its public “member functions” declared in lines 10-19:

```

public:
    Student();

    std::string firstName() const;
    std::string lastName() const;
    int testScore() const;
    char letterGrade() const;

    void setName(const std::string& firstName, const std::string& lastName);
    void setTestScore(int testScore);

```

The first member function, declared in line 11,

```
Student();
```

is called the “default constructor,” which is responsible for initializing a Student's data members. This constructor is automatically called whenever a Student object is created without supplying any arguments, as in

```

Student s;           // Construct a single object, s, of type Student
Student a[5];        // Construct an array of 5 Student objects

```

The syntax for defining a constructor is

```

ClassName::ClassName(parameters):
    initializerList
{
    body
}

```

The default constructor does not have any parameters. The “initializer list” specifies the value to which each data member is initialized when an object is constructed. The body contains any other statements to be executed after the data members have been initialized. The default constructor for the Student class is defined as (Student.cpp, lines 5-12)

```

Student::Student():
    _firstName(""),
    _lastName(""),
    _testScore(0),
    _letterGrade(' ')
{
    // ...
}

```

Whenever a Student is created via the default constructor, its `_firstName` and `_lastName` are initialized to empty strings, its `_testScore` is initialized to 0, and its `_letterGrade` is initialized to a blank character. Note that each data member in the initializer list is separated by a comma (lines 6-8).

The member function (Student.h, line 18)

```
void setName(const std::string& firstName, const std::string& lastName);
```

sets a Student's `_firstName` and `_lastName` to the given `firstName` and `lastName`, as in

```
s.setName("Cynthia", "Jackson");           // Set s._firstName to "Cynthia"
// and s._lastName to "Jackson"

a[2].setName("Roland", "Montgomery");    // Set a[2]._firstName to "Roland"
// and a[2]._lastName to "Montgomery"
```

The `.` is called the “member access operator.” In the above statements, the `setName` member function is said to be “called on” the objects `s` and `a[2]`. The syntax for defining a member function is

```
returnType ClassName::functionName(parameters)
{
    body
}
```

The `setName` function is defined as (Student.h, lines 48-53)

```
inline void Student::setName(const std::string& firstName,
    const std::string& lastName)
{
    _firstName = firstName;
    _lastName = lastName;
}
```

Because the function has been inlined, the definition is placed in the class header file (as opposed to the `.cpp` file).

Attempting to access an object's private members directly, as in

```
s._firstName = "Cynthia";
a[2]._lastName = "Montgomery";
```

results in a compiler error.

The member function (line 19)

```
void setTestScore(int testScore);
```

sets a Student's `_testScore` to the given `testScore`, then sets the Student's `_letterGrade` according to the following scale:

<code>_testScore</code>	<code>_letterGrade</code>
90 or above	A
80-89	B
70-79	C
60-69	D
Below 60	F

The function definition is (Student.cpp, lines 14-28)

```
void Student::setTestScore(int testScore)
{
    _testScore = testScore;

    if (_testScore >= 90)
        _letterGrade = 'A';
    else if (_testScore >= 80)
        _letterGrade = 'B';
    else if (_testScore >= 70)
        _letterGrade = 'C';
    else if (_testScore >= 60)
        _letterGrade = 'D';
    else
        _letterGrade = 'F';
}
```

The statement

```
s.setTestScore(92);
```

for example, sets the `_testScore` and `_letterGrade` of Student `s` to 92 and 'A', while the statement

```
a[2].setTestScore(85);
```

sets the `_testScore` and `_letterGrade` of Student `a[2]` to 85 and 'B'.

`setName` and `setTestScore`, which modify the object on which they are called, are known as “non-const member functions.” The remaining member functions (Student.h, lines 13-16),

```
std::string firstName() const;      // Returns the value of _firstName
std::string lastName() const;       // Returns the value of _lastName
int testScore() const;              // Returns the value of _testScore
char letterGrade() const;           // Returns the value of _letterGrade
```

are known as “const member functions” because they do not modify the object on which they are called. Placing the “`const`” keyword after the parameter list designates a member function as `const`. Note that this is also required in the function definition (lines 28-46):

```
inline std::string Student::firstName() const
{
    return _firstName;
}

inline std::string Student::lastName() const
{
    return _lastName;
}
```

```
inline int Student::testScore() const
{
    return _testScore;
}

inline char Student::letterGrade() const
{
    return _letterGrade;
}
```

The statements

```
string n = s.lastName();      // n is "Jackson"
cout << a[2].testScore();    // Print "85"
```

for example, read (but do not modify) the Student objects s and a[2].

A member function can also be called on an object through a pointer or reference to that object, as in

```
Student& r = s;           // r, a reference to a Student, refers to s
Student* p = a + 2;        // p, a pointer to a Student, points to a[2]

cout << r.letterGrade();   // cout << s.letterGrade();
cout << p->firstName();  // cout << a[2].firstName();
```

Note that the expression

```
p->firstName()           // a[2].firstName()
```

is shorthand for

```
(*p).firstName()          // Dereference p to obtain the referent object,
                           // then call the member function firstName on
                           // that object
```

Declaring member functions as `const`, where appropriate, provides a way to document code and reduce the possibility of programming errors. Attempting to modify a data member from within a `const` member function, for example, results in a compiler error:

```
inline std::string Student::firstName() const
{
    _firstName = "Bug";      // Compiler error: Const member functions cannot
                           // modify data members

    return _firstName;
}
```

Attempting to call a non-`const` member function on the referent object of a `const` pointer or `const` reference also results in a compiler error:

```

void printFullName(const Student& s)           // s is a const reference
{
    std::cout << s.firstName() << ' ' <<
        s.lastName() << std::endl;               // Ok: Can call const member
                                                    // functions on the referent of s

    s.setName("Bug", "McBuggerson");            // Compiler error: Cannot call
                                                // non-const member functions
                                                // on the referent of s
}

void printTestResults(const Student* s)          // s is a const pointer
{
    std::cout << s->testScore() << ' ' <<
        s->letterGrade() << std::endl;           // Ok: Can call const member
                                                    // functions on the referent of s

    s->setTestScore(100);                      // Compiler error: Cannot call
                                                // non-const member functions
                                                // on the referent of s
}

```

Lines 16-17 (main.cpp),

```

const int classSize = 5;
Student students[classSize];

```

construct an array, students, containing 5 objects of type Student. The loop (lines 19-32)

```

for (int i = 0; i != classSize; ++i)
{
    string firstName;
    string lastName;
    int testScore;

    cout << "\nEnter first name, last name, and test score\n"
        "(e.g. Owen Harper 87): ";

    cin >> firstName >> lastName >> testScore;

    students[i].setName(firstName, lastName);
    students[i].setTestScore(testScore);
}

```

performs 5 iterations. In each iteration, the user is prompted to enter two strings and an integer, which are stored in the variables firstName, lastName, and testScore (lines 21-28). These values are then stored in the current Student object (students[i]) via the member functions setName and setTestScore (lines 30-31). Recall that setTestScore also sets the Student's letter grade. Line 35,

```
printArray(students, students + classSize);
```

then prints each element in the array via the function (lines 8, 42-55)

```
void printArray(const Student* begin, const Student* end)
{
    using namespace std;

    while (begin != end)
    {
        cout << begin->firstName() << ' ' <<      // Print the referent element's
        begin->lastName() << ' ' <<                // firstName, lastName,
        begin->testScore() << ' ' <<               // testScore, and letterGrade
        begin->letterGrade() << endl;

        ++begin;                                     // Point to the next element
    }
}
```

A sample run of the program is

```
Enter first name, last name, and test score
(e.g. Owen Harper 87): Alexis Tague 84
```

```
Enter first name, last name, and test score
(e.g. Owen Harper 87): Nicholas Carson 93
```

```
Enter first name, last name, and test score
(e.g. Owen Harper 87): Diane Brown 76
```

```
Enter first name, last name, and test score
(e.g. Owen Harper 87): Peter Smith 81
```

```
Enter first name, last name, and test score
(e.g. Owen Harper 87): Gregory Nelson 65
```

```
Alexis Tague 84 B
Nicholas Carson 93 A
Diane Brown 76 C
Peter Smith 81 B
Gregory Nelson 65 D
```



## 6.2: Operator Overloading

*Source folders*

*operatorOverloading  
Student*

*Chapter outline*

- Defining common operators (comparison, I/O, assignment) for objects of a given type
- Implementing a copy constructor

Recall how the program in the previous chapter handled I/O (input / output) of Student objects:

```
Student s;

string firstName;                                // Temporary variables, used
string lastName;                                 // to store the user's input
int testScore;                                    // values

cin >> firstName >> lastName >> testScore;      // Obtain the user's input
                                                // values individually

s.setName(firstName, lastName);                  // Set the object's fields
s.setTestScore(testScore);

cout << s.firstName() << ' ' <<
     s.lastName() << ' ' <<
     s.testScore() << ' ' <<
     s.letterGrade() << endl;                      // Print each field individually
```

Although that works, it is far from ideal: handling I/O for Students should be as easy as it is for built-in types, such as int or double. In other words, users of the Student class ought to be able to simply write the above as

```
Student s;

cin >> s;                                         // Write the user's input values
                                                // to the object

cout << s << endl;                             // Print all of the object's
                                                // fields
```

Also consider how a user might compare two Students, x and y, for equality (assuming that two Students are considered to be equal if they have the same first and last names):

```

if (x.firstName() == y.firstName() && x.lastName() == y.lastName())
    cout << "x and y are equal\n";
else
    cout << "x and y are not equal\n";

```

Comparison, like I/O, should be no more difficult for Students than it is for ints. Users ought to be able to simply write

```

if (x == y)
    cout << "x and y are equal\n";
else
    cout << "x and y are not equal\n";

```

For all of the above to work, however, the stream extraction (`>>`), stream insertion (`<<`), and equality (`==`) operators must be defined, or “overloaded,” for objects of type Student. Each of these operators is implemented as a function. The expression

```
x == y
```

for example, is shorthand for

```
x.operator==(y)
```

where `operator==` is the name of a member function called on `x`, with `y` as the argument. The function, which returns a value of type `bool`, is declared as (Student.h, line 24)

```
bool operator==(const Student& rhs) const;
```

`rhs`, short for “right-hand side,” is a `const` reference to the `Student` on the right-hand side of the operator (`==`). In the expression (`x == y`), for example, `rhs` is a `const` reference to `y`. `x` is the left operand and `y` is the right operand. Also note that the function is declared `const` because it does not modify the object on which it is called. The function definition is (lines 74-77)

```

inline bool Student::operator==(const Student& rhs) const
{
    return _firstName == rhs._firstName && _lastName == rhs._lastName;
}

```

The function returns the result of the expression

```
_firstName == rhs._firstName && _lastName == rhs._lastName
```

which returns true if the expressions

```

(firstName == rhs._firstName) // Is the _firstName of the left operand
                             // equal to that of the right operand?

(lastName == rhs._lastName) // Is the _lastName of the left operand
                           // equal to that of the right operand?

```

both return true; otherwise, it returns false.

Similarly, the expressions

x != y  
x >= y  
x <= y  
x > y  
x < y

are shorthand for

```
x.operator !=(y)  
x.operator >=(y)  
x.operator <=(y)  
x.operator >(y)  
x.operator <(y)
```

where operator!=, operator>=, operator<=, operator>, and operator< are the names of member functions called on x, with y as the argument (lines 25-29):

```
bool operator!=(const Student& rhs) const;
bool operator>=(const Student& rhs) const;
bool operator<=(const Student& rhs) const;
bool operator>(const Student& rhs) const;
bool operator<(const Student& rhs) const;
```

A Student x is said to be “less than” a Student y if x's full name (last, first) comes before y's full name in alphabetical order (Student.cpp, lines 23-31):

```
bool Student::operator<(const Student& rhs) const
{
    if (_lastName < rhs._lastName)
        return true;
    else if (_lastName == rhs._lastName && _firstName < rhs._firstName)
        return true;
    else
        return false;
}
```

A Student named Douglas Atkinson, for example, is less than a Student named Mary Atkinson (because “Douglas” is less than “Mary”), but not less than Stuart Allen (because “Allen” is less than “Atkinson”). The remaining operators ( $\neq$ ,  $\geq$ ,  $\leq$ , and  $>$ ) can then be implemented in terms of  $\equiv$ ,  $\neq$ , and  $<$  (Student.h, lines 79-97):

```
inline bool Student::operator!=(const Student& rhs) const
{
    return !(*this == rhs); // True if this Student (the left operand)
} // is not equal to the right operand
```

```

inline bool Student::operator>=(const Student& rhs) const
{
    return !(*this < rhs);    // True if this Student (the left operand)
}                                // is not less than the right operand

inline bool Student::operator<=(const Student& rhs) const
{
    return !(*this > rhs);    // True if this Student (the left operand)
}                                // is not greater than the right operand

inline bool Student::operator>(const Student& rhs) const
{
    return (*this != rhs) && !(*this < rhs);    // True if this Student (the
}                                // left operand) is not equal
                                         // to the right operand, and
                                         // not less than the right
                                         // operand

```

Within the body of a member function, the “this” keyword is a pointer to the object on which the function was called. The expression

`*this`

which dereferences the “this” pointer, therefore returns the object on which the function was called. The `setName` member function, for example, could have been written as

```

inline void Student::setName(const std::string& firstName,
    const std::string& lastName)
{
    this->_firstName = firstName;    // (*this)._firstName = firstName;
    this->_lastName = lastName;      // (*this)._lastName = lastName;
}

```

As a sidenote, the “this” pointer is actually a hidden parameter in every member function: the compiler implements `setName` as

```

inline void Student::setName(Student* this,           // "this" is a pointer to the
    const std::string& firstName,                  // Student object on which the
    const std::string& lastName)                   // function was called
{
    this->_firstName = firstName;
    this->_lastName = lastName;
}

```

Given

`Student s;`

for example, the member function call

`s.setName("Theodore", "Jenkins");`

is implemented by the compiler as

```
Student::setName(&s, "Theodore", "Jenkins");      // Pass the address of s
                                                    // for "this"
```

Unlike the comparison operators, the stream operators (<< and >>) are implemented as nonmember functions. The expression

```
cout << s
```

for example, is shorthand for

```
operator<<(cout, s)
```

where operator<< is the name of a nonmember function, and cout and s are the arguments. cout is an object of type std::ostream (short for “output stream”). The function is declared as (Student.h, line 11)

```
std::ostream& operator<<(std::ostream& lhs, const Student& rhs);
```

Note that the forward declaration of the Student class (line 9) is necessary because the declaration of the operator<< function (which contains the type name Student) occurs before the class definition. The parameter lhs, short for “left-hand side,” is a reference to the ostream on the left-hand side of the operator (<<), and rhs is a const reference to the Student on the right-hand side. The function’s return type is std::ostream& (reference to an output stream). Also note that the parameter types (ostream&, const Student&) indicate that the function modifies the referent of lhs, but does not modify the referent of rhs. The definition is (lines 44-52)

```
inline std::ostream& operator<<(std::ostream& lhs, const Student& rhs)
{
    lhs << rhs.firstName() << ' ' <<
    rhs.lastName() << ' ' <<
    rhs.testScore() << ' ' <<
    rhs.letterGrade();

    return lhs;
}
```

The function prints each of the fields in rhs, then returns lhs (a reference to an ostream). The expression

```
cout << s
```

for example, uses the ostream object cout to print each of the fields in Student s, then returns a reference to cout. Returning the reference to cout is what makes it possible to write consecutive output expressions in a single statement, such as

```
Student t;
```

```
// ...
```

```
cout << s << t;
```

which is evaluated as

```
(cout << s) << t;
```

The expression (cout << s) prints s and returns a reference to cout. The returned reference to cout is then used to print t.

Similarly, the expression

```
cin >> s
```

is shorthand for

```
operator>>(cin, s)
```

where operator>> is the name of a nonmember function, and cin and s are the arguments. cin is an object of type std::istream (short for “input stream”). Note, however, that although this is a nonmember function, it is still declared in the Student class, as (Student.h, line 35)

```
friend std::istream& operator>>(std::istream& lhs, Student& rhs);
```

The “friend” keyword indicates that this function is a friend of the Student class, which means that it can access the private members of any Student object. Also note that in this function, rhs is of type Student& (as opposed to const Student&) because the function modifies the referent of rhs. The definition is (Student.cpp, lines 43-52)

```
std::istream& operator>>(std::istream& lhs, Student& rhs)
{
    int testScore;

    lhs >> rhs._firstName >> rhs._lastName >> testScore;
    rhs.setTestScore(testScore);

    return lhs;
}
```

Because this function is a friend of the Student class, it can write the user's input strings directly to the private members \_firstName and \_lastName (line 47). Note, however, that the test score must still be set by calling setTestScore (line 49) to ensure that rhs.\_letterGrade is also updated.

Like the operator<< function, operator>> returns a reference to the left operand. The expression

```
cin >> s
```

for example, returns a reference to cin, which is what makes it possible to write consecutive input expressions in a single statement, such as

```
Student t;
// ...
cin >> s >> t;
```

which is evaluated as

```
(cin >> s) >> t;
```

The expression (cin >> s) writes the user's input values to s and returns a reference to cin. The returned reference to cin is then used to write the next set of input values to t.

Line 17 (Student.h),

```
Student (const Student& source);
```

declares a member function known as the “copy constructor,” which constructs an object as a copy of an existing object (of the same type). The parameter, source, is a const reference to the existing (i.e. source) object:

```
Student s;                                // Construct Student s via the default
                                             // constructor

s.setName("Laura", "Wilkinson");
s.setTestScore(97);

Student t(s);                            // Construct Student t as a copy of s
Student u = t;                          // Construct Student u as a copy of t

cout << t << endl;                      // Prints "Laura Wilkinson 97 A"
cout << u << endl;                      // Prints "Laura Wilkinson 97 A"
```

The copy constructor is also used to construct function parameters and return values of type Student:

```
void printStudent(Student x)           // Each time this function is called,
{                                       // the parameter x is constructed as a
    std::cout << x << std::endl;      // copy of the argument (i.e. the Student
}                                       // object passed to the function)

Student getStudentInfoFromUser()       // Each time this function is called,
{                                       // it returns an unnamed object of type
    Student y;                         // Student, constructed as a copy of y

    std::cout << "Enter student info (<firstName> <lastName> <testScore>): ";
    std::cin >> y;

    return y;
}
```

The copy constructor initializes the value of each data member in the new object to that of its

counterpart in the source object (Student.cpp, lines 14-21):

```
Student::Student(const Student& source) :
    _firstName(source._firstName),
    _lastName(source._lastName),
    _testScore(source._testScore),
    _letterGrade(source._letterGrade)
{
    // ...
}
```

To avoid the unnecessary copying of Student objects (which is less efficient), the above functions (printStudent, getStudentInfoFromUser) can be written using pointer / reference parameters:

```
// Print the referent of r
// (instead of copying the argument and printing the copy)

void printStudent(const Student& r)
{
    std::cout << r << std::endl;
}

// Write the user's input values to the referent of p
// (instead of writing to a temporary object and returning a copy of it)

void getStudentInfoFromUser(Student* p)
{
    std::cout << "Enter student info (<firstName> <lastName> <testScore>): ";
    std::cin >> *p;
}
```

The assignment operator (=) can also be overloaded for the Student class. To continue with the above example, where Student s is Laura Wilkinson with a test score of 97:

```
Student v;

v.setName("Daniel", "Sutherland");
v.setTestScore(84);

s = v;                                // Assign the "value" of Student v to
                                         // that of Student s

cout << s << endl;                      // Prints "Daniel Sutherland 84 B"
```

The assignment expression

```
s = v
```

is shorthand for

```
s.operator=(v)
```

where operator= is the name of a member function called on s, with v as the argument. operator=, the assignment operator, is declared as (Student.h, line 34)

```
Student& operator=(const Student& rhs);
```

The function sets the value of each field in the left operand to that of its counterpart in the right operand (rhs), then returns a reference to the left operand (Student.cpp, lines 33-41):

```
Student& Student::operator=(const Student& rhs)
{
    _firstName = rhs._firstName;
    _lastName = rhs._lastName;
    _testScore = rhs._testScore;
    _letterGrade = rhs._letterGrade;

    return *this;                                // Return a reference to the object on
                                                // which the function was called (a
                                                // reference to the left operand)
}
```

Returning a reference to the left operand is what makes it possible to write consecutive assignment expressions in a single statement, such as

```
s = v = u;
```

which is evaluated as

```
s = (v = u);           // Set v equal to u, then set s equal to v
```

The expression (v = u) sets the value of v to that of u, then returns a reference to v. The returned reference then becomes the right operand that completes the expression (s = ).

Note that if no copy constructor / assignment operator is defined for a given class, the compiler automatically generates one. The implicit (compiler-generated) copy constructor simply constructs each new data member as a copy of its counterpart, while the implicit assignment operator simply applies the assignment operator to each data member and its counterpart. For the Student class, both the implicit copy constructor and assignment operator would have sufficed; they have been explicitly defined here for the purpose of illustration only. Classes developed in later chapters, however, will require more complex copy constructors and assignment operators.

The program in this chapter applies the new operators (I/O, comparison, assignment) and the copy constructor. Lines 17-26 (main.cpp) obtain the names and test scores of 5 students, using the stream extraction operator to directly write the user's input values to each Student object:

```
const int classSize = 5;
Student students[classSize];
```

```

for (int i = 0; i != classSize; ++i)
{
    cout << "\nEnter first name, last name, and test score\n"
        "(e.g. Owen Harper 87): ";

    cin >> students[i];      // operator>>(cin, students[i]);
}

```

The stream insertion operator greatly simplifies the printArray function from the previous chapter (lines 7, 39-46):

```

void printArray(const Student* begin, const Student* end)
{
    while (begin != end)
    {
        std::cout << *begin << std::endl;      // operator<<(cout, *begin) << endl;
        ++begin;
    }
}

```

The swap function (lines 8, 48-53) is identical to that of Chapter 5.6, but uses the type Student instead of int:

```

void swap(Student& a, Student& b)
{
    Student c = a;      // c is created via the copy constructor
    a = b;              // a.operator=(b);
    b = c;              // b.operator=(c);
}

```

The insertionSort function (lines 9, 55-70) is also identical to that of Chapter 5.6, but uses the type Student instead of int. Because operator< compares two Student objects according to the alphabetical order of their names (last, first), this function sorts the Students in alphabetical order:

```

void insertionSort(Student* begin, Student* end)
{
    for (Student* current = begin; current != end; ++current)
    {
        Student* y = current;
        Student* x = current;
        --x;

        while (y != begin && *y < *x)          // *y < *x is equivalent to
        {                                         // (*y).operator<(*x), or
            swap(*y, *x);                      // y->operator<(*x)
            --y;
            --x;
        }
    }
}

```

A sample run of the program is

Enter first name, last name, and test score  
(e.g. Owen Harper 87): Alexis Tague 84

Enter first name, last name, and test score  
(e.g. Owen Harper 87): Nicholas Carson 93

Enter first name, last name, and test score  
(e.g. Owen Harper 87): Diane Brown 76

Enter first name, last name, and test score  
(e.g. Owen Harper 87): Peter Smith 81

Enter first name, last name, and test score  
(e.g. Owen Harper 87): Gregory Nelson 65

Performing insertion sort...

The sorted sequence (by lastName, firstName) is:

Diane Brown 76 C

Nicholas Carson 93 A

Gregory Nelson 65 D

Peter Smith 81 B

Alexis Tague 84 B



# Part 7: Templates and Function Objects

## 7.1: Function Templates

*Source folders*

```
functionTemplates
printSequence
Student
swap
```

*Chapter outline*

- *Creating generic functions*
- *The difference between function parameters, function arguments, template parameters, and template arguments*

Recall the swap functions from Chapters 5.6 and 6.2,

```
void swap(int& a, int& b)           // Swap the referents of a and b
{
    int c = a;                      // a and b are references to ints
    a = b;                          // (int&)
    b = c;
}

void swap(Student& a, Student& b)    // Swap the referents of a and b
{
    Student c = a;                 // a and b are references to Students
    a = b;                          // (Student&)
    b = c;
}
```

This function can be defined generically as (swap.h, lines 9-15)

```
template <typename T>
void swap(T& a, T& b)           // Swap the referents of a and b
{
    T c = a;                     // a and b are references to objects of
    a = b;                       // type T (T&)
    b = c;
}
```

The declaration is in lines 6-7,

```
template <typename T>
void swap(T& a, T& b);
```

This version of swap is called a “function template” because it is a template from which the compiler generates the code for a function. a and b are the function parameters, and T is the “template parameter.” Given

```
int k = 5;
int n = 8;
```

for example, the function can be called as

```
swap<int>(k, n);
```

k and n are the function arguments, and int is the “template argument.” Substituting int for the template parameter T, the compiler generates the code

```
void swap(int& a, int& b);

void swap(int& a, int& b)
{
    int c = a;
    a = b;
    b = c;
}
```

This process is called “template instantiation” because the compiler is said to generate an instance of a function from a template. Note that in the template, the function parameters a and b are of the same type (T). In the above function call, the function arguments k and n are also of the same type (int). The compiler can therefore deduce the intended template argument from the type of the function arguments (int). The above function call, in other words, can simply be written as

```
swap(k, n);                                // Equivalent to
                                              // swap<int>(k, n);
```

This is known as “template argument deduction.” Similarly, given

```
double x = 3.14159;
double y = 1.618;

string s = "Freddie Hubbard";
string t = "Lee Morgan";

swap(x, y);                                // Equivalent to
                                              // swap<double>(x, y);

swap(s, t);                                // Equivalent to
                                              // swap<string>(s, t);
```

the compiler generates the code

```
void swap(double& a, double& b);
```

```

void swap(double& a, double& b)
{
    double c = a;
    a = b;
    b = c;
}

void swap(std::string& a, std::string& b);

void swap(std::string& a, std::string& b)
{
    std::string c = a;
    a = b;
    b = c;
}

```

Recall from Chapter 4.5 that in order to prevent ODR violations, non-inline function definitions must be placed in .cpp files. Function templates, however, are an exception to the ODR: their function definitions (whether inline or not) must be placed in include-guarded header files.

The printArray functions from Chapters 5.6 and 6.2,

```

void printArray(const int* begin, const int* end)
{
    while (begin != end)
    {
        std::cout << *begin << ' ';
        ++begin;
    }
}

void printArray(const Student* begin, const Student* end)
{
    while (begin != end)
    {
        std::cout << *begin << std::endl;
        ++begin;
    }
}

```

can be defined generically as (printSequence.h, lines 11-19)

```

template <class Iter>
void printSequence(Iter begin, Iter end)
{
    while (begin != end)
    {
        std::cout << *begin << ' ';
        ++begin;
    }
}

```

Note that when specifying template parameters, the “class” keyword (lines 8, 11) can be used interchangeably with the “typename” keyword:

```
template <class Iter>                                // template <typename Iter>
void printSequence(Iter begin, Iter end);
```

The template parameter, Iter, is short for “iterator.” An iterator is an object used to traverse a sequence of elements. Pointers, which are used to traverse arrays, are the most powerful type of iterator in that they support the broadest set of operations:

- Dereference (\* / ->): Access the referent element
- Equality / inequality comparison (== / !=): Determine whether two iterators point to the same element
- Increment / decrement (++ / --): Point to the next / previous element in the sequence
- Greater / less-than comparison (> / >= / < / <=): Determine whether the referent element of one iterator comes after / before the referent element of another iterator
- Addition / subtraction (+ / += / - / -=): Point to any element, given an integer offset
- Iterator subtraction (-): Obtain the distance (number of elements) between two iterators

An iterator, by definition, must support at least dereference and increment. Not all types of iterators, however, support all of the above operations. More complex data structures, developed in later chapters, will require custom iterators implemented as classes.

The requirements for instantiating the printSequence template are:

- Iter must support inequality comparison, dereference, and increment (lines 14, 16, 17).
- The stream insertion operator (<<) must be defined for the type of the referent elements (line 16)

Using the swap function template, the insertionSort functions from Chapters 5.6 and 6.2,

```
void insertionSort(int* begin, int* end)
{
    for (int* current = begin; current != end; ++current)
    {
        int* y = current;
        int* x = current;
        --x;

        while (y != begin && *y < *x)
        {
            swap(*y, *x);
            --y;
            --x;
        }
    }
}
```

```

void insertionSort(Student* begin, Student* end)
{
    for (Student* current = begin; current != end; ++current)
    {
        Student* y = current;
        Student* x = current;
        --x;

        while (y != begin && *y < *x)
        {
            swap(*y, *x);
            --y;
            --x;
        }
    }
}

```

can be defined generically as (insertionSort.h, lines 11-27)

```

template <class Iter>
void insertionSort(Iter begin, Iter end)
{
    for (Iter current = begin; current != end; ++current)
    {
        Iter y = current;
        Iter x = current;
        --x;

        while (y != begin && *y < *x)
        {
            swap(*y, *x);
            --y;
            --x;
        }
    }
}

```

The requirements for instantiating this template are:

- Iter must support inequality comparison, increment, decrement, and dereference (lines 14, 18, 20, 22-24)
- The type of the referent elements must support less-than comparison (line 20)
- The swap template must be instantiable for the type of the referent elements (line 22)

The program in this chapter demonstrates the insertionSort / swap and printSequence function templates. Lines 12-24 (main.cpp),

```

const int totalElements = 5;

int intArray[totalElements];
Student studentArray[totalElements];

```

```

for (int i = 0; i != totalElements; ++i)
{
    cout << "\nEnter a number (integer): ";
    cin >> intArray[i];

    cout << "Enter student name and test score (e.g. Owen Harper 87): ";
    cin >> studentArray[i];
}

```

populate 2 arrays, one containing 5 ints (intArray) and the other (studentArray) containing 5 Students.  
Lines 28-29,

```

insertionSort(intArray, intArray + totalElements);
insertionSort(studentArray, studentArray + totalElements);

```

are equivalent to

```

insertionSort<int*>(intArray, intArray + totalElements);
insertionSort<Student*>(studentArray, studentArray + totalElements);

```

Template argument deduction allows the template arguments (int\* / Student\*) to be omitted. The compiler generates the functions

```

void insertionSort(int* begin, int* end)
{
    // ...
}

void insertionSort(Student* begin, Student* end)
{
    // ...
}

```

Within each function body, the compiler deduces the template argument for swap based on the type of \*y and \*x (int / Student), and generates the code

```

void swap(int& a, int& b)
{
    // ...
}

void swap(Student& a, Student& b)
{
    // ...
}

```

Lines 33 and 36,

```

printSequence(intArray, intArray + totalElements);
printSequence(studentArray, studentArray + totalElements);

```

are equivalent to

```
printSequence<int*>(intArray, intArray + totalElements);
printSequence<Student*>(studentArray, studentArray + totalElements);
```

The compiler generates the code

```
void printSequence(int* begin, int* end)
{
    // ...
}

void printSequence(Student* begin, Student* end)
{
    // ...
}
```

A sample run of the program is

```
Enter a number (integer): 3
Enter student name and test score (e.g. Owen Harper 87): Alexis Tague 84

Enter a number (integer): 1
Enter student name and test score (e.g. Owen Harper 87): Nicholas Carson 93

Enter a number (integer): 5
Enter student name and test score (e.g. Owen Harper 87): Diane Brown 76

Enter a number (integer): 4
Enter student name and test score (e.g. Owen Harper 87): Peter Smith 81

Enter a number (integer): 2
Enter student name and test score (e.g. Owen Harper 87): Gregory Nelson 65

Performing insertion sort...

The sorted sequences are:
1 2 3 4 5

Diane Brown 76 C Nicholas Carson 93 A Gregory Nelson 65 D Peter Smith 81 B Ale
xis Tague 84 B
```



## 7.2: Class Templates and Function Objects

### *Source folders*

```
classTemplates
insertionSort
predicates
Student
```

### *Chapter outline*

- *Creating generic classes*
- *Overloading the function call operator*
- *Creating functions with function object parameters*

A “class template” is a template from which the compiler generates the code for a class, given one or more type arguments. Consider, for example, the class template (*predicates.h*, lines 9-13)

```
template <class T>
struct Less
{
    bool operator()(const T& a, const T& b) const;
};
```

When defining a class, the “*struct*” keyword specifies that all members are public by default. The above is therefore equivalent to

```
template <class T>
class Less
{
public:
    bool operator()(const T& a, const T& b) const;
};
```

*operator()*, the “function call operator,” is a *const* member function that returns a value of type *bool*. The parameters, *a* and *b*, are *const* references to objects of type *T* (*const T&*). The function returns true if *a* is less than *b*; otherwise, it returns false (lines 27-31):

```
template <class T>
inline bool Less<T>::operator()(const T& a, const T& b) const
{
    return a < b;
}
```

Note that outside the class definition, the full name of the class is *Less<T>* (line 28). The *Less* class template can be instantiated for any type that supports less-than comparison (*<*). Given

```
int k = 13;
```

```

int n = 21;
string s = "Rome";
string t = "Venice";

Less<int> isLessInt;           // Construct isLessInt, an object of type
                               // Less<int>

Less<string> isLessString;     // Construct isLessString, an object of type
                               // Less<string>

```

for example, the compiler uses the template arguments (int / string) to generate the code

```

struct Less<int>
{
    bool operator()(const int& a, const int& b) const;
};

inline bool Less<int>::operator()(const int& a, const int& b) const
{
    return a < b;
}

struct Less<std::string>
{
    bool operator()(const std::string& a, const std::string& b) const;
};

inline bool Less<std::string>::operator()(const std::string& a,
                                         const std::string& b) const
{
    return a < b;
}

```

Objects that overload the function call operator, such as isLessInt and isLessString, are known as “function objects.” Function objects can be called like ordinary functions. The expressions

```

isLessInt(k, n)                  // Is k less than n?
isLessString(s, t)               // Is s less than t?

```

for example, are shorthand for

```

isLessInt.operator()(k, n)        // Call the member function operator() on
                                // the object isLessInt, with k and n as the
                                // arguments

isLessString.operator()(s, t)      // Call the member function operator() on
                                // the object isLessString, with s and t as
                                // the arguments

```

The Greater class template is identical to the Less class template, but uses the greater-than operator (>) (lines 15-19, 33-37):

```

template <class T>
struct Greater
{
    bool operator()(const T& a, const T& b) const;
};

template <class T>
inline bool Greater<T>::operator()(const T& a, const T& b) const
{
    return a > b;
}

```

Function objects are particularly useful as function parameters. Recall from the previous chapter that the insertionSort function uses the less-than operator ( $<$ ), which sorts the sequence in ascending order:

```

template <class Iter>
void insertionSort(Iter begin, Iter end)
{
    for (Iter current = begin; current != end; ++current)
    {
        Iter y = current;
        Iter x = current;
        --x;

        while (y != begin && *y < *x)           // Use the less-than operator to compare
        {                                         // *y and *x (sorts the elements in
            swap(*y, *x);                      // ascending order)
            --y;
            --x;
        }
    }
}

```

Using the greater-than operator ( $>$ ) to compare  $*y$  and  $*x$  would therefore sort the elements in descending order. Hard-coding the operator into the function, however, would require two separate but nearly identical functions, such as

```

template <class Iter>
void insertionSortAscending(Iter begin, Iter end)
{
    // Use the < operator to compare *y and *x
}

template <class Iter>
void insertionSortDescending(Iter begin, Iter end)
{
    // Use the > operator to compare *y and *x
}

```

This duplication can be avoided by adding a template parameter and function parameter (insertionSort.h, lines 8-9):

```
template <class Iter, class Predicate>
void insertionSort(Iter begin, Iter end, Predicate predicate);
```

The new template parameter, `Predicate`, is the type of a function object that compares two elements and returns true or false. The new function parameter, `predicate`, is the function object itself (lines 11-27):

```
template <class Iter, class Predicate>
void insertionSort(Iter begin, Iter end, Predicate predicate)
{
    for (Iter current = begin; current != end; ++current)
    {
        Iter y = current;
        Iter x = current;
        --x;

        while (y != begin && predicate(*y, *x)) // Use the given predicate to
        {                                       // compare *y and *x, where
            swap(*y, *x);                   // predicate(*y, *x)
            --y;                           // is shorthand for
            --x;                           // predicate.operator()(*y, *x)
        }
    }
}
```

Given

```
Student a[5];           // An array of 5 Students
Less<Student> isLess;   // A function object of type Less<Student>
Greater<Student> isGreater; // A function object of type Greater<Student>

// Populate the array...
```

the array can be sorted in ascending or descending order by passing the appropriate function object to `insertionSort`:

```
insertionSort(a, a + 5, isLess);
insertionSort(a, a + 5, isGreater);
```

Without using template argument deduction, the above function calls would be written as

```
insertionSort<Student*, Less<Student>>(a, a + 5, isLess);
insertionSort<Student*, Greater<Student>>(a, a + 5, isGreater);
```

The compiler generates the code

```
struct Less<Student>
{
    bool operator()(const Student& a, const Student& b) const;
};
```

```

inline bool Less<Student>::operator() (const Student& a,
    const Student& b) const
{
    return a < b;      // return a.operator<(b);
}

struct Greater<Student>
{
    bool operator() (const Student& a, const Student& b) const;
};

inline bool Greater<Student>::operator() (const Student& a,
    const Student& b) const
{
    return a > b;      // return a.operator>(b);
}

void insertionSort(Student* begin, Student* end, Less<Student> predicate)
{
    for (Student* current = begin; current != end; ++current)
    {
        Student* y = current;
        Student* x = current;
        --x;

        while (y != begin && predicate(*y, *x))      // predicate(*y, *x) evaluates
        {
            swap(*y, *x);                          // *y < *x
            --y;
            --x;
        }
    }
}

void insertionSort(Student* begin, Student* end, Greater<Student> predicate)
{
    for (Student* current = begin; current != end; ++current)
    {
        Student* y = current;
        Student* x = current;
        --x;

        while (y != begin && predicate(*y, *x))      // predicate(*y, *x) evaluates
        {
            swap(*y, *x);                          // *y > *x
            --y;
            --x;
        }
    }
}

```

Note that it is also possible to pass an unnamed function object, as in

```
insertionSort(a, a + 5, Less<Student>());
```

where the expression

```
Less<Student>()
```

returns an unnamed object of type Less<Student>, constructed via the default constructor. The function parameter in insertionSort (predicate) is then constructed as a copy of this unnamed object. After insertionSort terminates, the unnamed object is destroyed.

By creating additional function objects, insertionSort can also be used to sort Students by their first names and test scores. The function objects (alternativeStudentPredicates.h, lines 8-16)

```
struct LessByFirstName
{
    bool operator()(const Student& a, const Student& b) const;
};

struct GreaterByFirstName
{
    bool operator()(const Student& a, const Student& b) const;
};
```

compare two Students, a and b, by comparing their first names (lines 28-38):

```
inline bool LessByFirstName::operator()(const Student& a,
    const Student& b) const
{
    return a.firstName() < b.firstName();      // a is less than b if
                                                // a's firstName is less than
                                                // b's firstName

inline bool GreaterByFirstName::operator()(const Student& a,
    const Student& b) const
{
    return a.firstName() > b.firstName();      // a is greater than b if
                                                // a's firstName is greater than
                                                // b's firstName
```

Similarly, the function objects (lines 18-26)

```
struct LessByTestScore
{
    bool operator()(const Student& a, const Student& b) const;
};

struct GreaterByTestScore
{
    bool operator()(const Student& a, const Student& b) const;
};
```

compare two Students, a and b, by comparing their test scores (lines 40-50):

```

inline bool LessByTestScore::operator()(const Student& a,
    const Student& b) const
{
    return a.testScore() < b.testScore();      // a is less than b if
                                                // a's testScore is less than
                                                // b's testScore

inline bool GreaterByTestScore::operator()(const Student& a,
    const Student& b) const
{
    return a.testScore() > b.testScore();      // a is greater than b if
                                                // a's testScore is greater than
                                                // b's testScore

```

Continuing with the above example (where *a* is an array of 5 Students), these function objects can be passed to *insertionSort*:

```

insertionSort(a, a + 5, LessByFirstName());           // Sort by firstName,
                                                       // ascending

insertionSort(a, a + 5, GreaterByFirstName());          // Sort by firstName,
                                                       // descending

insertionSort(a, a + 5, LessByTestScore());            // Sort by testScore,
                                                       // ascending

insertionSort(a, a + 5, GreaterByTestScore());          // Sort by testScore,
                                                       // descending

```

The above function calls are equivalent to

```

insertionSort<Student*, LessByFirstName>(a, a + 5, LessByFirstName());
insertionSort<Student*, GreaterByFirstName>(a, a + 5, GreaterByFirstName());
insertionSort<Student*, LessByTestScore>(a, a + 5, LessByTestScore());
insertionSort<Student*, GreaterByTestScore>(a, a + 5, GreaterByTestScore());

```

The compiler generates the code

```

void insertionSort(Student* begin, Student* end, LessByFirstName predicate)
{
    // ...
}

void insertionSort(Student* begin, Student* end, GreaterByFirstName predicate)
{
    // ...
}

void insertionSort(Student* begin, Student* end, LessByTestScore predicate)
{
    // ...
}

```

```
void insertionSort(Student* begin, Student* end, GreaterByTestScore predicate)
{
    // ...
}
```

The program in this chapter demonstrates all of the aforementioned examples of insertionSort. Lines 14-21 (main.cpp),

```
const int size = 5;
Student a[size];

for (int i = 0; i != size; ++i)
{
    cout << "\nEnter student name and test score (e.g. Owen Harper 87): ";
    cin >> a[i];
}
```

populate an array of 5 Students, and lines 23-45,

```
cout << "\nSorted by lastName, ascending:\n";
insertionSort(a, a + 5, Less<Student>());
printSequence(a, a + 5);

cout << "\n\nSorted by lastName, descending:\n";
insertionSort(a, a + 5, Greater<Student>());
printSequence(a, a + 5);

cout << "\n\nSorted by firstName, ascending:\n";
insertionSort(a, a + 5, LessByFirstName());
printSequence(a, a + 5);

cout << "\n\nSorted by firstName, descending:\n";
insertionSort(a, a + 5, GreaterByFirstName());
printSequence(a, a + 5);

cout << "\n\nSorted by testScore, ascending:\n";
insertionSort(a, a + 5, LessByTestScore());
printSequence(a, a + 5);

cout << "\n\nSorted by testScore, descending:\n";
insertionSort(a, a + 5, GreaterByTestScore());
printSequence(a, a + 5);
```

sort the array using the various predicates. A sample run of the program is

```
Enter student name and test score (e.g. Owen Harper 87): Alexis Tague 84
Enter student name and test score (e.g. Owen Harper 87): Nicholas Carson 93
Enter student name and test score (e.g. Owen Harper 87): Diane Brown 76
Enter student name and test score (e.g. Owen Harper 87): Peter Smith 81
```

```
Enter student name and test score (e.g. Owen Harper 87): Gregory Nelson 65
```

Sorted by lastName, ascending:

```
Diane Brown 76 C Nicholas Carson 93 A Gregory Nelson 65 D Peter Smith 81 B Alexis Tague 84 B
```

Sorted by lastName, descending:

```
Alexis Tague 84 B Peter Smith 81 B Gregory Nelson 65 D Nicholas Carson 93 A Diane Brown 76 C
```

Sorted by firstName, ascending:

```
Alexis Tague 84 B Diane Brown 76 C Gregory Nelson 65 D Nicholas Carson 93 A Peter Smith 81 B
```

Sorted by firstName, descending:

```
Peter Smith 81 B Nicholas Carson 93 A Gregory Nelson 65 D Diane Brown 76 C Alexis Tague 84 B
```

Sorted by testScore, ascending:

```
Gregory Nelson 65 D Diane Brown 76 C Peter Smith 81 B Alexis Tague 84 B Nicholas Carson 93 A
```

Sorted by testScore, descending:

```
Nicholas Carson 93 A Alexis Tague 84 B Peter Smith 81 B Diane Brown 76 C Gregory Nelson 65 D
```

The function (predicates.h, lines 6-7)

```
template <class T, class Predicate>
bool isReflexivelyEqual(const T& a, const T& b, Predicate predicate);
```

determines whether a is equal to b by performing a reflexive comparison using the given predicate.  
Given

```
int k = 34;
int n = 34;
```

for example, the expression

```
isReflexivelyEqual(k, n, Less<int>())      // is k reflexively equal to n,
                                              // using the less-than operator?
```

returns true because k is not less than n, and n is not less than k. Similarly, the expression

```
isReflexivelyEqual(k, n, Greater<int>())    // is k reflexively equal to n,
                                              // using the greater-than operator?
```

returns true because k is not greater than n, and n is not greater than k. The function definition is (lines 21-25)

```
template <class T, class Predicate>
inline bool isReflexivelyEqual(const T& a, const T& b, Predicate predicate)
{
    return !predicate(a, b) && !predicate(b, a);      // Returns true if
                                                       // predicate(a, b) and
                                                       // predicate(b, a) both
                                                       // return false
```

In addition to the classes `Less<int>` and `Greater<int>`, the compiler generates the code

```
inline bool isReflexivelyEqual(const int& a,
    const int& b,
    Less<int> predicate)
{
    return !predicate(a, b) && !predicate(b, a);      // Returns true if
                                                       // a is not less than b and
                                                       // b is not less than a

inline bool isReflexivelyEqual(const int& a,
    const int& b,
    Greater<int> predicate)
{
    return !predicate(a, b) && !predicate(b, a);      // Returns true if
                                                       // a is not greater than b
                                                       // and b is not greater
                                                       // than a
```

`isReflexivelyEqual` is used to search binary trees, which will be discussed in later chapters. It has been included in the same header file as the `Less` / `Greater` templates, however, because the concepts are closely related.

## 7.3: Introducing the Array Class Template

*Source files and folders*

*Array/1  
Array/common/memberFunctions\_1.h  
quickSort*

*Chapter outline*

- Creating aliases of existing types
- Overloading the array subscript operator
- Const overloading
- Implementing quicksort

A major drawback of “primitive” (built-in) arrays, such as

```
int a[5];                      // a, a primitive array of 5 ints
string b[10];                   // b, a primitive array of 10 strings
// ...
// ...
```

is that an array's size (i.e. the number of contained elements) is not embedded into the array itself. This makes it particularly easy to create bugs, such as

```
printSequence(a, a + 10);        // Undefined behavior: the one-past-the-last
                                // element of a is at address (a + 5), not
                                // (a + 10) (accidentally used 10, the size of b)

printSequence(b, b + 5);         // Only prints the first half of the array:
                                // the one-past-the-last element of b is at
                                // address (b + 10), not (b + 5) (accidentally
                                // used 5, the size of a)
```

Storing the size in a separate variable, as in

```
const int aSize = 5;
const int bSize = 10;

int a[aSize];
string b [bSize];

// ...

cout << "a contains " << aSize << " elements\n";
cout << "b contains " << bSize << " elements\n";

printSequence(a, a + aSize);
printSequence(b, b + bSize);
```

makes it somewhat less error-prone, but no less cumbersome.

This chapter introduces the `Array` class template, which, in addition to solving this problem, illustrates some of the operations common to all the data structures developed herein.

An object of type `Array<T, Size>` contains a single private data member, a primitive array of `Size` elements of type `T`. An object of type `Array<double, 5>`, for example, contains a primitive array of 5 doubles. Its member functions simply provide access to the elements in the contained array, as well as information about its size. Using the `Array` class, the above code can simply be written as

```
Array<int, 5> a;           // Construct a, an object of type Array<int, 5>
Array<string, 10> b;        // Construct b, an object of type Array<string, 10>

// ...

cout << "a contains " << a.size() << " elements\n";
cout << "b contains " << b.size() << " elements\n";

printSequence(a.begin(), a.end());
printSequence(b.begin(), b.end());
```

The member function `size` returns the size of the Array, while `begin` / `end` return iterators (pointers) to the first / one-past-the-last elements.

The class is defined in the header file `Array.h` (lines 6-36). The two template parameters (line 6) are `T` (the type of the stored elements) and `Size` (the number of stored elements). Because `Size` is a `const unsigned` integer, attempting to construct an `Array` with a negative `Size` results in a compiler error:

```
Array<double, -20> c; // Compiler error: Size must be nonnegative
```

The private data member `a` (line 35) is a primitive array containing `Size` elements of type `T`.

The “`typedef`” keyword (lines 10-18) is used to create an alias (alternate name) of an existing type. The syntax for declaring a `typedef` (type alias) is

```
typedef existingTypeName alias;
```

Line 18,

```
typedef T value_type;
```

for example, declares that within the `Array` class, `value_type` is an alias of `T`. Outside the class, the full name of the alias is `Array<T, Size>::value_type`. The statements

for example, are equivalent to

```
double d = 3.14159;
Student s;
```

Similarly, lines 10-17,

```
typedef const T* const_iterator;
typedef const T* const_pointer;
typedef const T& const_reference;
typedef T* iterator;
typedef T* pointer;
typedef T& reference;
typedef int difference_type;
typedef unsigned int size_type;
```

declare that

Array<T, Size>::const\_iterator is an alias of the type const T\* (const pointer to a T)  
 Array<T, Size>::const\_pointer is an alias of the type const T\*  
 Array<T, Size>::const\_reference is an alias of the type const T& (const reference to a T)  
 Array<T, Size>::iterator is an alias of the type T\* (pointer to a T)  
 Array<T, Size>::pointer is an alias of the type T\*  
 Array<T, Size>::reference is an alias of the type T& (reference to a T)  
 Array<T, Size>::difference\_type is an alias of the type int (type used to represent the distance  
     (number of elements) between two elements)  
 Array<T, Size>::size\_type is an alias of the type unsigned int (type used to represent the size  
     of an Array)

The statements

```
Array<double, 5>::const_iterator p;
Array<Student, 10>::const_pointer q;

Array<int, 15>::iterator r;
Array<string, 20>::pointer s;

Array<bool, 25>::difference_type t;
Array<double, 5>::size_type u;
```

for example, are equivalent to

```
const double* p;
const Student* q;

int* r;
string* s;

int t;
unsigned int u;
```

Similarly, `Array<double, 5>::const_reference` is an alias of the type `const double&` (`const` reference to a `double`), while `Array<Student, 10>::reference` is an alias of the type `Student&` (`reference` to a `Student`).

The aliases declared in lines 10-18 are collectively known as “member types” of the `Array` class. Member types, which provide a mechanism for forwarding type information about a class, play a fundamental role in the creation of generic components. Type forwarding will be applied in later chapters.

The member functions (lines 20-32) provide an interface to the private array `_a`. The aforementioned `typedefs` make their declarations equivalent to

```

bool empty() const;           // Returns true if the Array is empty (i.e. if
                             // its Size is 0); otherwise, returns false

unsigned int size() const;    // Returns the number of elements stored in
                             // the Array (i.e. its Size)

const T* begin() const;       // Returns a const_iterator pointing to the
                             // first element

const T* end() const;         // Returns a const_iterator pointing to the
                             // one-past-the-last element

const T& front() const;       // Returns a const_reference to the first
                             // element

const T& back() const;        // Returns a const_reference to the last
                             // element

const T& operator[](unsigned int index) const; // Returns a const_reference
                                              // to the element with the
                                              // given index

T* begin();                  // Returns an iterator pointing to the first
                             // element

T* end();                    // Returns an iterator pointing to the one-
                             // past-the-last element

T& front();                 // Returns a reference to the first element

T& back();                  // Returns a reference to the last element

T& operator[](unsigned int index); // Returns a reference to the
                                 // element with the given
                                 // index

```

The member function definitions have been placed in a separate header file (`memberFunctions_1.h`), which will prevent code duplication as the class is expanded upon in later chapters. This header file is included in line 39 of `Array.h`.

The first member function, `empty`, is defined as (lines 3-7)

```
template <class T, const unsigned int Size>
inline bool Array<T, Size>::empty() const
{
    return Size == 0;
}
```

where the full name of the class is `Array<T, Size>`. The definition of the size function is (lines 9-13)

```
template <class T, const unsigned int Size>
inline typename Array<T, Size>::size_type Array<T, Size>::size() const
{
    return Size;
}
```

where the return type is

```
typename Array<T, Size>::size_type // unsigned int
```

The “`typename`” keyword is required in this context because `size_type` is a member type of a class template (`Array`). This also applies to the return types of the remaining member functions (lines 16, 22, 28, 34, 40, 47, 53, 59, 65, 71),

```
typename Array<T, Size>::const_iterator // const T*
typename Array<T, Size>::const_reference // const T&
typename Array<T, Size>::iterator // T*
typename Array<T, Size>::reference // T&
```

Recall from Chapter 6.1 that a `const` member function does not modify the object on which it is called, while a non-`const` member function does. Overloading a member function based on whether or not it is `const` is known as “`const` overloading.” The `begin` function, for example, is said to be “`const` overloaded” (`Array.h`, lines 22, 28):

```
const_iterator begin() const; // const version
iterator begin(); // non-const version
```

The `const` version, which returns a `const_iterator` (`const T*`), cannot be used to modify the elements in an `Array`. The non-`const` version, which returns an `iterator` (`T*`), can be used to modify the elements in an `Array`.

When a `const`-overloaded function is called, the compiler selects the appropriate overload based on how the call was made. Given

```
typedef Array<double, 5> DoubleArray; // DoubleArray is an alias of the
                                         // type Array<double, 5>

DoubleArray x; // Construct x, an object of type
               // Array<double, 5>
```

```

DoubleArray* p = &x;                                // p, a pointer to an
                                                       // Array<double, 5>, points to x

DoubleArray& r = x;                                // r, a reference to an
                                                       // Array<double, 5>, refers to x

```

for example, the expressions

```

x.begin()      // Return an Array<double, 5>::iterator (a double*)
p->begin()    // pointing to the first element (x._a[0])
r.begin()

```

call the non-const version of begin (because x is a non-const object, p is a non-const pointer, and r is a non-const reference). Given

```

const DoubleArray* cp = &x;                         // cp, a const pointer to an
                                                       // Array<double, 5>, points to x

const DoubleArray& cr = x;                         // cr, a const reference to an
                                                       // Array<double, 5>, refers to x

```

however, the expressions

```

cp->begin()    // Return an Array<double, 5>::const_iterator (a const double*)
cr.begin()      // pointing to the first element (x._a[0])

```

call the const version of begin (because cp is a const pointer and cr is a const reference).

The member function end, which returns an iterator pointing to the one-past-the-last element, is also const-overloaded: the non-const version returns an Array<T, Size>::iterator (T\*), while the const version returns an Array<T, Size>::const\_iterator (const T\*).

begin and end are defined as (memberFunctions\_1.h, lines 15-25, 46-56):

```

template <class T, const unsigned int Size>
inline typename Array<T, Size>::const_iterator Array<T, Size>::begin() const
{
    return _a;           // return &_a[0];
}

template <class T, const unsigned int Size>
inline typename Array<T, Size>::const_iterator Array<T, Size>::end() const
{
    return _a + size(); // return &_a[size()];
}

template <class T, const unsigned int Size>
inline typename Array<T, Size>::iterator Array<T, Size>::begin()
{
    return _a;           // return &_a[0];
}

```

```
template <class T, const unsigned int Size>
inline typename Array<T, Size>::iterator Array<T, Size>::end()
{
    return _a + size();      // return &_a[size()];
}
```

The member functions front and back return references / const references to the first / last elements in \_a (lines 27-37, 58-68):

```
// Const versions of front / back (return const_references)

template <class T, const unsigned int Size>
inline typename Array<T, Size>::const_reference Array<T, Size>::front() const
{
    return *_a;                // Return a const reference to element _a[0]
}

template <class T, const unsigned int Size>
inline typename Array<T, Size>::const_reference Array<T, Size>::back() const
{
    return *(_a + size() - 1);   // Return a const reference to element
                                // _a[size() - 1]

// Non-const versions of front / back (return non-const references)

template <class T, const unsigned int Size>
inline typename Array<T, Size>::reference Array<T, Size>::front()
{
    return *_a;                // Return a reference to element _a[0]
}

template <class T, const unsigned int Size>
inline typename Array<T, Size>::reference Array<T, Size>::back()
{
    return *(_a + size() - 1);   // Return a reference to element
                                // _a[size() - 1]
```

Continuing with the above example (where x is an object of type `Array<double, 5>`), the expression

```
x[2]           // Element 2
```

is shorthand for

```
x.operator[](2) // Returns a reference to x._a[2]
```

where `operator[]` (the array subscript operator) is the name of a member function called on x, with 2 as the argument. The const version of the function returns a const reference to the element at the given index, while the non-const version returns a non-const reference (lines 39-44, 70-75):

```

template <class T, const unsigned int Size>
inline typename Array<T, Size>::const_reference Array<T, Size>::operator[](
    size_type index) const
{
    return _a[index];      // Return a const reference to element _a[index]
}

template <class T, const unsigned int Size>
inline typename Array<T, Size>::reference Array<T, Size>::operator[](
    size_type index)
{
    return _a[index];      // Return a reference to element _a[index]
}

```

In addition to demonstrating these member functions, the program in this chapter introduces a new sorting algorithm called “quicksort.” Lines 11-12 (main.cpp),

```

typedef Array<int, 7> Array;
Array a;

```

declare `Array` as an alias of the type `Array<int, 7>` (`Array` containing 7 ints), then construct an `Array<int, 7> a`. The compiler generates the class definition

```

class Array<int, 7>
{
public:
    typedef const int* const_iterator;
    typedef const int* const_pointer;
    typedef const int& const_reference;
    typedef int* iterator;
    typedef int* pointer;
    typedef int& reference;
    typedef int difference_type;
    typedef unsigned int size_type;
    typedef int value_type;

    bool empty() const;
    size_type size() const;
    const_iterator begin() const;
    const_iterator end() const;
    const_reference front() const;
    const_reference back() const;
    const_reference operator[](size_type index) const;

    iterator begin();
    iterator end();
    reference front();
    reference back();
    reference operator[](size_type index);

private:
    int _a[7];
};

```

as well as the corresponding member function definitions (when those member functions are called), such as

```
inline Array<int, 7>::size_type Array<int, 7>::size() const
{
    return 7;
}

inline Array<int, 7>::iterator Array<int, 7>::begin()
{
    return _a;
}
```

Lines 14-20,

```
a[0] = 5;      // a.operator[](0) = 5;      // a._a[0] = 5;
a[1] = 7;      // a.operator[](1) = 7;      // a._a[1] = 7;
a[2] = 6;      // a.operator[](2) = 6;      // a._a[2] = 6;
a[3] = 1;      // a.operator[](3) = 1;      // a._a[3] = 1;
a[4] = 3;      // a.operator[](4) = 3;      // a._a[4] = 3;
a[5] = 2;      // a.operator[](5) = 2;      // a._a[5] = 2;
a[6] = 4;      // a.operator[](6) = 4;      // a._a[6] = 4;
```

set the values of the elements [a[0], a[7]) via the operator[] member function. Lines 22-27,

```
cout << "a contains " << a.size() << " elements:\n";
printSequence(a.begin(), a.end());
cout << endl << endl;

cout << "The front element is " << a.front() << endl;
cout << "The back element is " << a.back() << endl << endl;
```

demonstrate the remaining member functions, generating the output

```
a contains 7 elements:
5 7 6 1 3 2 4

The front element is 5
The back element is 4
```

Lines 29-33,

```
cout << "Performing quicksort...\n";
quickSort(a.begin(), a.end(), Less<Array::value_type>());

cout << "\nThe sorted sequence is:\n";
printSequence(a.begin(), a.end());
```

then sort and print the Array, generating the output

```
Performing quicksort...
```

The sorted sequence is:  
 1 2 3 4 5 6 7

Note that the expression (line 30)

```
Less<Array::value_type>()      // Construct an unnamed temporary object
                                // of type Less<Array::value_type>
```

is equivalent to

```
Less<int>()
```

because Array::value\_type is an alias of the type int.

The quickSort function template is declared as (quickSort.h, lines 8-9)

```
template <class Iter, class Predicate>
void quickSort(Iter begin, Iter end, Predicate predicate);
```

Like the insertionSort function from Chapter 7.1, quickSort has 2 template parameters, Iter (the type of iterator used to traverse the sequence) and Predicate (the type of function object used to compare elements).

The quicksort algorithm, developed by Tony Hoare, is more efficient than insertion sort, but also more complex. Quicksort is a “recursive algorithm,” i.e. an algorithm that is defined in terms of itself. It works by dividing a sequence of elements into progressively smaller sub-sequences until there are no more remaining sub-sequences. Consider, for example, the sequence

```
5 7 6 1 3 2 4      // Sequence S
```

The process by which a sequence of elements is divided into 2 sub-sequences is called “partitioning.” To partition a given sequence, its last element is selected as the “pivot element.” The goal of the partition operation is to rearrange the sequence such that all elements less than the pivot are to the left of the pivot, while all elements not less than the pivot are to the right of the pivot. Sequence S is partitioned by selecting its last element (4) as the pivot:

```
5 7 6 1 3 2 4      // Sequence S
p                  // The pivot element (p) is 4
                  // Place all elements less than p to the left of p,
                  // and all elements not less than p to the right of p
```

Partitioning sequence S results in 2 sub-sequences, A and B:

```
1 3 2 4 7 6 5      // Sub-sequence A is {1,3,2} (elements less than p)
p                  // Sub-sequence B is {7,6,5} (elements not less than p)
```

Note that order of the elements in each sub-sequence is not important, only that they are all less than (or not less than) the pivot element.

The sub-sequences, A and B, are then partitioned. Sub-sequence A is partitioned by selecting its last element (2) as the pivot:

```
1 3 2          // Sub-sequence A
p              // The pivot element (p) is 2
               // Place all elements less than p to the left of p,
               // and all elements not less than p to the right of p
```

Partitioning sub-sequence A divides it into 2 sub-sequences, D and E:

```
1 2 3          // Sub-sequence D is {1} (elements less than p)
p              // Sub-sequence E is {3} (elements not less than p)
```

The next sub-sequences, D and E, are then partitioned. Because each consists of a single element, however, no further action is required. The original sequence, S, is now

```
1 2 3 4 7 6 5
```

Sub-sequence B is then partitioned by selecting its last element (5) as the pivot:

```
7 6 5          // Sub-sequence B
p              // The pivot element (p) is 5
               // Place all elements less than p to the left of p,
               // and all elements not less than p to the right of p
```

Partitioning sub-sequence B divides it into 2 sub-sequences, F and G:

```
5 6 7          // Sub-sequence F is {} (elements less than p)
p              // Sub-sequence G is {6,7} (elements not less than p)
```

No further action is required for sub-sequence F because it contains 0 elements. Sub-sequence G is then partitioned by selecting its last element (7) as the pivot:

```
6 7          // Sub-sequence G
p              // The pivot element (p) is 7
               // Place all elements less than p to the left of p,
               // and all elements not less than p to the right of p
```

Partitioning sub-sequence G divides it into 2 sub-sequences, H and I:

```
6 7          // Sub-sequence H is {6} (elements less than p)
p              // Sub-sequence I is {} (elements not less than p)
```

No further action is required for sub-sequence H because it contains 1 element, and no further action is required for sub-sequence I because it contains 0 elements. There are no more remaining sub-sequences, so the sorting is complete. The above example can be summarized with the pseudocode

```

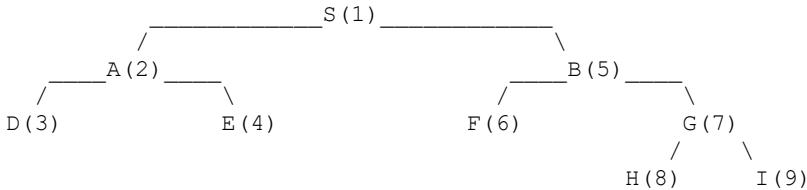
Partition S into A and B
{
    Partition A into D and E
    {
        Partition D (1 element, no sub-sequences)
        Partition E (1 element, no sub-sequences)
    }

    Partition B into F and G
    {
        Partition F (0 elements, no sub-sequences)

        Partition G into H and I
        {
            Partition H (1 element, no sub-sequences)
            Partition I (0 elements, no sub-sequences)
        }
    }
}

```

and the diagram



where the numbers (1-9) indicate the order in which the sequences are partitioned (S is partitioned first, followed by A, D, E, etc.). A is the “left sub-sequence” of S and B is the “right sub-sequence” of S. Similarly, D is the left sub-sequence of A and E is the right sub-sequence of A. The pseudocode for quickSort is

```

template <class Iter, class Predicate>
void quickSort(Iter begin, Iter end, Predicate predicate)
{
    if the sequence [begin, end) contains 1 or more elements
    {
        Partition the sequence [begin, end) and obtain an Iter
        pointing to the pivot element (pivot);

        quickSort the left sub-sequence, [begin, pivot);           // Recursive call
        quickSort the right sub-sequence, [++pivot, end);         // Recursive call
    }
}

```

In the above example, the original sequence S is

```

b           e      // b is begin, e is end
5 7 6 1 3 2 4

```

S contains 1 or more elements, so it is partitioned into

```
b      e    // p is pivot
1 3 2 4 7 6 5 // The left sub-sequence, [begin, pivot), is {1,3,2} (A)
                p // The right sub-sequence, [++pivot, end), is {7,6,5} (B)
```

quickSort is then recursively called for A (until it can no longer be partitioned), followed by B (until it can no longer be partitioned). The function (quickSort.h, lines 11-12)

```
template <class Iter, class Predicate>
Iter _partition(Iter begin, Iter end, Predicate predicate);
```

partitions the sequence [begin, end), using the given predicate to compare elements. The function returns an Iter pointing to the pivot element (the element dividing the original sequence into left and right sub-sequences) (lines 29-50):

```
template <class Iter, class Predicate>
Iter _partition(Iter begin, Iter end, Predicate predicate)
{
    Iter pivot = end;
    Iter x = end;

    --pivot;

    for (Iter y = begin; y != pivot; ++y)
    {
        if (predicate(*y, *pivot))
        {
            x = _xNextElement(x, begin, end);
            swap(*x, *y);
        }
    }

    x = _xNextElement(x, begin, end);
    swap(*x, *pivot);

    return x;
}
```

Given the sequence

```
b      e    // b is begin, e is end
5 7 6 1 3 2 4
```

and a less-than predicate, for example, the function performs the following operations:

```
Iter pivot = end;
Iter x = end;

--pivot;
```

```

5 7 6 1 3 2 4      p
                     x

```

---

Iteration 1:

```

5 7 6 1 3 2 4      p
y                  x
                     // predicate(*y, *pivot) returns false
                     // (5 is not less than 4)

++y;

```

Iteration 2:

```

5 7 6 1 3 2 4      p
y                  x
                     // predicate(*y, *pivot) returns false
                     // (7 is not less than 4)

++y;

```

Iteration 3:

```

5 7 6 1 3 2 4      p
y                  x
                     // predicate(*y, *pivot) returns false
                     // (6 is not less than 4)

++y;

```

Iteration 4:

```

5 7 6 1 3 2 4      p
y                  x
                     // predicate(*y, *pivot) returns true
                     // (1 is less than 4)

x = _xNextElement(x, begin, end);    // If x currently points to the end,
                                         // then point x to the first element;
                                         // otherwise, point x to the next
                                         // element

5 7 6 1 3 2 4      p
x      y

swap(*x, *y);

1 7 6 5 3 2 4      p
x      y
                     // predicate(*y, *pivot) returns false
                     // (1 is not less than 4)

++y;

```



```

    p
1 3 2 5 7 6 4
      x     y

swap(*x, *pivot);

b      p e
1 3 2 4 7 6 5
      x     y
                                         // The sequence is now partitioned
                                         // into the sub-sequences {1 3 2}
                                         // (elements less than 4) and {7 6 5}
                                         // (elements not less than 4)
                                         // Element 4 is the pivot element

return x;                                // Return an Iter pointing to the
                                         // pivot element

```

The statement (lines 41, 46)

```

x = _xNextElement(x, begin, end);          // If x currently points to the end,
                                             // then point x to the first element;
                                             // otherwise, point x to the next
                                             // element

```

is equivalent to

```

if (x == end)
    x = begin;
else
    ++x;

```

The `_xNextElement` function, used primarily to improve the readability of `_partition`, is defined as (lines 52-59)

```

template <class Iter>
Iter _xNextElement(Iter x, Iter begin, Iter end)
{
    if (x == end)
        return begin;
    else
        return ++x;
}

```

As a sidenote, using `_xNextElement` could have been circumvented entirely by simply initializing `x` to point to the one-before-the-first element, then simply incrementing `x` whenever necessary:

```

template <class Iter, class Predicate>
Iter _partition(Iter begin, Iter end, Predicate predicate)
{
    Iter pivot = end;
    Iter x = begin;

    --pivot;
    --x;                                // x points to the one-before-the-
                                         // first element
    for (Iter y = begin; y != pivot; ++y)
    {
        if (predicate(*y, *pivot))
        {
            ++x;                          // Point x to the next element
            swap(*x, *y);
        }
    }

    ++x;                                // Point x to the next element
    swap(*x, *pivot);

    return x;
}

```

`_partition` was not implemented this way because according to the C++ standard, decrementing an iterator at the beginning of a sequence is undefined, though it (usually) works for ordinary pointers (`int*`, `const string*`, etc.).

Once the `_partition` method is implemented, defining the `quickSort` function is relatively easy (lines 17-27):

```

template <class Iter, class Predicate>
void quickSort(Iter begin, Iter end, Predicate predicate)
{
    if (begin != end)      // If the sequence contains 1 or more elements...
    {
        Iter pivot = _partition(begin, end, predicate); // pivot points to the
                                                       // element dividing the
                                                       // original sequence

        quickSort(begin, pivot, predicate);   // quickSort the left sub-sequence
        quickSort(++pivot, end, predicate);  // quickSort the right sub-sequence
    }
}

```

Recall from the first example that it is unnecessary to partition / sort a sub-sequence containing 1 element because any 1-element sequence, by definition, is already sorted. Changing the condition (line 20) to check for 2 or more elements would therefore make the function more efficient:

```
template <class Iter, class Predicate>
void quickSort(Iter begin, Iter end, Predicate predicate)
{
    if (end - begin > 1)      // If the sequence contains 2 or more elements...
    {
        Iter pivot = _partition(begin, end, predicate);

        quickSort(begin, pivot, predicate);
        quickSort(++pivot, end, predicate);
    }
}
```

quickSort was not implemented this way, however, to make it compatible with both “random access iterators” and “bidirectional iterators,” which will be discussed in later chapters.

# Part 8: Dynamic Memory Allocation

## 8.1: Tracing Object Lifetime

*Source folders*

*Traceable  
tracingObjectLifetime*

*Chapter outline*

- *Implicit conversion*
- *Implementing a destructor*
- *Tracing an object's life cycle (construction and destruction) in real-time*

A major drawback of both primitive arrays and Arrays is that their sizes must be set at compile time. It is not possible, in other words, to dynamically set the size based on the user's input at runtime:

```
int size;
cout << "Enter desired array size: ";
cin >> size;

double a[size];           // Compiler error: The size of a primitive array
Array<string, size> b;   // (or an Array) must be a compile-time constant
```

The data structures developed from Part 9 onwards are dynamically resizable, but their implementations will require explicit (manual) memory management. This chapter introduces the Traceable class, which will be used to trace and verify the correctness of explicit memory management.

An object of type Traceable<T> contains a single data member of type T:

```
Traceable<int> i;          // i, a Traceable<int>, contains a single data
                           // member of type int

Traceable<double> d;       // d, a Traceable<double>, contains a single data
                           // member of type double

Traceable<string> s;        // s, a Traceable<string>, contains a single data
                           // member of type string
```

Each object of type Traceable<T> prints a message when it is:

- Constructed (via the default constructor, copy constructor, or from an object of type T)
- Assigned a new value (via the assignment or stream extraction operators)
- Destroyed

The class definition is (Traceable.h, lines 17-44)

```
template <class T>
class Traceable
{
public:
    typedef T value_type;

    Traceable();
    Traceable(const Traceable& source);
    Traceable(const T& source);
    ~Traceable();

    bool operator==(const Traceable& rhs) const;
    bool operator!=(const Traceable& rhs) const;
    bool operator>=(const Traceable& rhs) const;
    bool operator<=(const Traceable& rhs) const;
    bool operator>(const Traceable& rhs) const;
    bool operator<(const Traceable& rhs) const;

    void printValue() const;

    Traceable& operator=(const Traceable& rhs);

    friend std::ostream& operator<< <>(std::ostream& lhs, const Traceable& rhs);
    friend std::istream& operator>> <>(std::istream& lhs, Traceable& rhs);

private:
    T _value;
};
```

value\_type (line 21) is an alias of T, the type of the underlying value (line 43). The default constructor, declared in line 23, is defined as (lines 46-51)

```
template <class T>
inline Traceable<T>::Traceable():
    _value(T())
{
    std::cout << "d " << _value << std::endl;
}
```

The expression (line 48)

```
T()
```

default-constructs a temporary, unnamed object of type T, after which \_value is constructed as a copy of the unnamed object. The function body (line 50),

```
std::cout << "d " << _value << std::endl;
```

then prints the message

```
d <_value>
```

indicating that a Traceable object was (d)efault-constructed.

The copy constructor, declared in line 24, constructs a Traceable<T> as a copy of an existing Traceable<T>. The const reference parameter, source, refers to the source object (lines 53-58):

```
template <class T>
inline Traceable<T>::Traceable(const Traceable& source):
    _value(source._value)
{
    std::cout << "c " << _value << std::endl;
}
```

\_value is constructed as a copy of its counterpart in the source object (line 55). The function body (line 57) then prints the message

```
c <_value>
```

indicating that a Traceable object was (c)opied.

The “value constructor,” declared in line 25, constructs a Traceable<T> from a value of the underlying type (T). The parameter, value, is a const reference to the source value (lines 60-65):

```
template <class T>
inline Traceable<T>::Traceable(const T& value):
    _value(value)
{
    std::cout << "v " << _value << std::endl;
}
```

\_value is constructed as a copy of the source value (line 62). The function body (line 64) then prints the message

```
v <_value>
```

indicating that a Traceable object was constructed from a (v)alue of the underlying type (T). The statements

```
Traceable<int> w;           // Default-construct w
Traceable<int> x(w);        // Copy-construct x from w
Traceable<int> y = w;        // Copy-construct y from w
Traceable<int> z(7);         // Value-construct z from 7
```

for example, generate the output

```
d 0 // An object with a value of 0 was (d)eault-constructed
c 0 // An object with a value of 0 was (c)opy-constructed
c 0 // An object with a value of 0 was (c)opy-constructed
v 7 // An object with a value of 7 was (v)alue-constructed
```

Line 26,

```
~Traceable();
```

declares a member function called the “destructor,” which destroys an object. The destructor is automatically called on an automatically-allocated object when it goes out of scope (recall the different types of scope from Chapter 4.3). All of the objects in the programs thus far have been automatically-allocated. Dynamically-allocated objects, which must be destroyed by manually calling their destructors, will be introduced in the next chapter.

If a destructor is not defined for a given class, the compiler automatically generates one. Note, however, that whether the destructor is implicit or user-defined, it automatically calls the destructor for each data member. The destructor for the Traceable class, for example, is defined as (lines 67-71)

```
template <class T>
inline Traceable<T>::~Traceable()
{
    std::cout << "~ " << _value << std::endl;
}
```

The function body (line 70) prints the message

~ < value>

indicating that a Traceable object was destroyed. The destructor for the class T is then automatically called on the data member value.

The comparison operators, declared in lines 28-33, compare a Traceable<T> with another Traceable<T> by simply comparing the respective values (lines 73-107).

The assignment operator, declared in line 37, assigns the `_value` of the right operand to that of the left operand, and prints a message indicating that the `_value` of a Traceable object was changed (lines 109-117). The statements

for example, generate the output

Given

```
Traceable<int> a;
```

the statement

```
a = 2;
```

is shorthand for

```
a.operator=(2);
```

Note, however, that the operator= function requires an argument of type Traceable<int>, not int (line 37). The compiler therefore actually generates the code

```
a.operator=(Traceable<int>(2));
```

where the expression

```
Traceable<int>(2)
```

value-constructs a temporary, unnamed Traceable<int> from the int 2. This unnamed Traceable<int> is then passed as the argument to the operator= function. After the function call, the unnamed object goes out of scope, at which point its destructor is automatically called. The statement

```
a = 2;
```

therefore generates the output

```
v 2          // (V)alue-construct a temporary, unnamed argument
0 -> 2      // a.operator=(unnamed argument)
~ 2          // Destroy the unnamed argument

// a._value is now 2
```

The automatic call to the value constructor in the above statement is known as “implicit conversion.” Similarly, the boolean expression

```
a == 9          // Is a._value equal to 9?
```

is shorthand for

```
a.operator==(Traceable<int>(9))    // Value-construct a temporary, unnamed
                                         // argument to pass to operator==
```

because the operator== function requires an argument of type Traceable<int>, not int (line 28).

The printValue member function, declared in line 35, simply prints \_value (lines 119-123). This function will be used to test the iterators developed in later chapters.

The stream operators are overloaded as nonmember functions. Note that because these functions are templates, they must be declared before the class definition (lines 8-15):

```
template <class T>
class Traceable;

template <class T>
std::ostream& operator<<(std::ostream& lhs, const Traceable<T>& rhs);

template <class T>
std::istream& operator>>(std::istream& lhs, Traceable<T>& rhs);
```

The forward declaration of the Traceable class (lines 8-9) is required because both functions have `Traceable<T>` parameters (lines 11-15). Both functions also require access to the private member `_value`, so they are declared as friends of the Traceable class (lines 39-40):

```
friend std::ostream& operator<< <>(std::ostream& lhs, const Traceable& rhs);
friend std::istream& operator>> <>(std::istream& lhs, Traceable& rhs);
```

Note that in the friend declarations, the `<>` are also required because these functions are templates.

The stream insertion operator simply prints `_value` and returns a reference to the left operand (lines 125-131):

```
template <class T>
std::ostream& operator<<(std::ostream& lhs, const Traceable<T>& rhs)
{
    lhs << rhs._value;

    return lhs;
}
```

The stream extraction operator writes the user's input value to `_value` and prints a message indicating that the `_value` of a Traceable object was changed (similar to the assignment operator) (lines 133-143):

```
template <class T>
std::istream& operator>>(std::istream& lhs, Traceable<T>& rhs)
{
    T originalValue = rhs._value;

    lhs >> rhs._value;

    std::cout << " " << originalValue << " -> " << rhs._value << std::endl;

    return lhs;
}
```

Given the statements

```
Traceable<int> i;
cout << "Enter an integer: ";
```

```
cin >> i;
cout << "You entered " << i;
```

for example, suppose that the user enters 6. The resultant output would be

```
d 0          // An object with a value of 0 was (d)efault-constructed
Enter an integer: 6
 0 -> 6      // The value of an object was changed from 0 to 6
You entered 6
```

The program in this chapter demonstrates the Traceable class by applying insertionSort to an Array of 5 Traceable<int>. Lines 14-15 (main.cpp),

```
typedef Array<Traceable<int>, 5> Array;
Array a;
```

construct the Array, where Array is an alias of the type Array<Traceable<int>, 5> (Array of 5 Traceable<int>). a is constructed via Array's default constructor (line 15), which default-constructs the elements [a.\_a[0], a.\_a[5]]. The resultant output is

```
d 0          // Default-construct a._a[0]
d 0          // Default-construct a._a[1]
d 0          // Default-construct a._a[2]
d 0          // Default-construct a._a[3]
d 0          // Default-construct a._a[4]
```

The loop (lines 17-21),

```
for (Array::size_type i = 0; i != a.size(); ++i)
{
    cout << "\nEnter an integer: ";
    cin >> a[i];
}
```

prompts the user to enter an integer value for each element. Assuming the user enters the values {3, 1, 5, 4, 2}, the program generates the output

```
Enter an integer: 3
 0 -> 3          // The _value of an object was changed from
                  // 0 to 3 (operator>>) (Traceable.h, line 140)
Enter an integer: 1
 0 -> 1

Enter an integer: 5
 0 -> 5

Enter an integer: 4
 0 -> 4

Enter an integer: 2
 0 -> 2
```

Line 24 (main.cpp),

```
insertionSort(a.begin(), a.end(), Less<Array::value_type>());
```

is equivalent to

```
insertionSort<Array::iterator, Less<Array::value_type>>(a.begin(),
    a.end(),
    Less<Array::value_type>());
```

or

```
insertionSort<Traceable<int>*, Less<Traceable<int>>>(a.begin(),
    a.end(),
    Less<Traceable<int>>());
```

Recall that each execution of the swap function entails the construction of a temporary object, c, which is constructed as a copy of a (swap/swap.h, line 12). insertionSort therefore generates the output

```
c 1      // Copy-construct an object (swap.h, line 12 / Traceable.h, line 57)
1 -> 3  // operator= (swap.h, line 13 / Traceable.h, line 112)
3 -> 1  // operator= (swap.h, line 14 / Traceable.h, line 112)
~ 1      // Destroy temporary object (swap.h, line 15 / Traceable.h, line 70)
c 4
4 -> 5
5 -> 4
~ 4
c 2
2 -> 5
5 -> 2
~ 2
c 2
2 -> 4
4 -> 2
~ 2
c 2
2 -> 3
3 -> 2
~ 2
```

The call to printSequence (main.cpp, line 27),

```
printSequence(a.begin(), a.end());
```

uses the stream insertion operator (Traceable.h, lines 128-130), generating the output

```
1 2 3 4 5
```

When main terminates, a's destructor automatically calls the destructors on each of its data members, in the opposite order in which they were constructed. The resultant output is

```
~ 5 // Destroy a._a[4] (Traceable.h, line 70)
~ 4 // Destroy a._a[3]
~ 3 // Destroy a._a[2]
~ 2 // Destroy a._a[1]
~ 1 // Destroy a._a[0]
```



## 8.2: Introducing the Allocator Class Template

*Source folders*

*Allocator  
introducingAllocator*

*Chapter outline*

- *The difference between automatically and dynamically-allocated objects*
- *Managing dynamically-allocated objects*

Recall from the previous chapter that there are two types of objects, automatically-allocated objects (also known as “automatic objects”) and dynamically-allocated objects. All of the programs thus far have used automatic objects exclusively.

The memory for an automatic object, as the name implies, is automatically allocated (reserved) in a region called the “stack.” At the end of its scope, the object is automatically destroyed and the memory used to hold it is automatically deallocated (freed).

The memory for a dynamically-allocated object, however, must be reserved manually in a separate region called the “heap” (also known as the “free store”). A dynamically-allocated object has unlimited scope; it can exist even after the program terminates. It must therefore be manually destroyed when no longer needed, and the memory used to hold it must be manually freed.

Failing to properly deallocate heap memory results in a “memory leak” (i.e. a leak in the pool of available memory). Because leaked memory cannot be reused by the system, preventing leaks is critical.

An Array (Chapter 7.3), which is a fixed-size data structure, contains automatic objects (elements). The resizable data structures developed from Part 9 onwards, however, will all contain dynamically-allocated objects. This chapter introduces the Allocator class template, which will be used to manage dynamically-allocated objects.

The class is defined in the header file Allocator.h (lines 9-31). An object of type Allocator<T> contains no data members; it simply provides a set of member functions for managing dynamically-allocated objects of type T. An object of type Allocator<double>, for example, manages dynamically-allocated doubles, an Allocator<string> manages dynamically-allocated strings, etc.

The aliases declared in lines 13-19 are identical to those of the Array class. Note that all of the member functions (lines 21-30) are const because there are no data members that can be modified.

The member function max\_size, declared in line 21, returns the maximum possible size (number of elements) of a dynamically-allocated array of T (lines 33-37):

```
template <class T>
inline typename Allocator<T>::size_type Allocator<T>::max_size() const
{
    return std::numeric_limits<size_type>::max() / sizeof(T);
}
```

The expression

```
std::numeric_limits<size_type>::max()
```

returns the largest possible value that can be represented by the type size\_type (unsigned int, line 18), using the given architecture (compiler / hardware). The Standard Library component numeric\_limits is provided by including the header <limits> (line 4). The expression

```
sizeof(T)
```

returns the number of bytes required to hold a single object of type T, using the given architecture. If the largest possible value of an unsigned int, for example, is 100,000,000, and each object of type T requires 8 bytes, then max\_size returns  $100,000,000 / 8$ , or 12,500,000 (the maximum possible size of a dynamically-allocated array of T is 12,500,000 elements).

The overloaded member function address, declared in lines 23-24, simply returns the address of the given object (lines 39-51):

```
template <class T>
inline typename Allocator<T>::const_pointer Allocator<T>::address(
    const_reference object) const
{
    return &object; // Return a const pointer to the given object
}

template <class T>
inline typename Allocator<T>::pointer Allocator<T>::address(
    reference object) const
{
    return &object; // Return a (non-const) pointer to the given object
}
```

The member function allocate, declared in line 26, reserves a block of memory large enough to hold the given number of total objects, then returns a pointer to the beginning of the block (lines 53-58):

```
template <class T>
inline typename Allocator<T>::pointer Allocator<T>::allocate(
    size_type totalObjects) const
{
    return static_cast<T*>(operator new(sizeof(T) * totalObjects));
}
```

Note that this function only reserves memory; it does not actually construct any objects. The expression

```
sizeof(T) * totalObjects
```

returns the number of bytes required to hold the given number of total objects of type T. The expression

```
operator new(sizeof(T) * totalObjects)
```

reserves a block of memory of (sizeof(T) \* totalObjects) bytes, and returns a pointer to the beginning of the block (the address of the block). The returned pointer is of type void\* (pointer to void), which represents a pointer to an unknown type. The expression

```
static_cast<T*>(operator new(sizeof(T) * totalObjects));
```

converts that pointer from the type void\* (pointer to void) to the the type T\* (pointer to T).

The member function construct, declared in line 29, constructs a copy of the given source object at the given address p. p must point to memory already reserved via the allocate member function (lines 66-71):

```
template <class T>
inline void Allocator<T>::construct(pointer p,
    const_reference sourceObject) const
{
    new(p) T(sourceObject);
}
```

The expression

```
T(sourceObject)
```

constructs an unnamed object of type T as a copy of sourceObject, and the statement

```
new(p) T(sourceObject);
```

constructs that unnamed object at address p. Similarly, the statement

```
new(p) T();
```

default-constructs an unnamed object of type T at address p. Constructing (placing) a new object at a specific address (as in the above statements) is called “placement new.”

The member function destroy, declared in line 30, destroys the object at the given address p by calling its destructor. p must point to an object constructed via the construct member function (lines 73-77):

```
template <class T>
inline void Allocator<T>::destroy(pointer p) const
{
    p->~T();      // Dereference p and call the destructor of the class T
}                  // on the referent object
```

The member function deallocate, declared in line 27, frees the memory block at the given address p. p must point to the beginning of a memory block reserved via the allocate member function (lines 60-64):

```
template <class T>
inline void Allocator<T>::deallocate(pointer p) const
{
    operator delete(p); // Free the memory block at address p
}
```

The program in this chapter manages a dynamically-allocated array of Traceable<int>. Line 12 (main.cpp),

```
typedef Allocator<Traceable<int>> Allocator;
```

declares Allocator as an alias of the type Allocator<Traceable<int>> (Allocator of Traceable<int>). Line 13,

```
typedef Allocator::pointer Pointer;
```

is therefore equivalent to

```
typedef Allocator<Traceable<int>>::pointer Pointer;
```

or

```
typedef Traceable<int>* Pointer; // Pointer is an alias of the type
// Traceable<int>* (pointer to
// Traceable<int>)
```

Lines 15-19,

```
Allocator::size_type totalObjects; // unsigned int totalObjects;
cout << "\nEnter number of objects to create: ";
cin >> totalObjects;
```

```
Allocator allocator;
```

prompt the user for the desired number of totalObjects, then construct the allocator. Line 22,

```
Pointer a = allocator.allocate(totalObjects);
```

allocates a memory block large enough to hold the desired number of totalObjects, and stores its address in the Pointer (Traceable<int>\*) a. The loop (lines 25-26)

```
for (Pointer p = a; p != a + totalObjects; ++p)
    allocator.construct(p, Allocator::value_type());
```

then constructs the objects. The expression

```
Allocator::value_type() // Traceable<int>()
```

default-constructs an unnamed Traceable<int>, which is passed as the argument to the construct function. The function then constructs a copy of this object at address p. If the user enters 5 for totalObjects, for example, the loop generates the output

```
d 0    // Default-construct argument (passed to construct function)
c 0    // Construct a copy of the argument at address p (a)
~ 0    // Destroy argument (after termination of construct function)
d 0    // Default-construct argument
c 0    // Construct a copy of the argument at address p (a + 1)
~ 0    // Destroy argument
d 0    // Default-construct argument
c 0    // Construct a copy of the argument at address p (a + 2)
~ 0    // Destroy argument
d 0    // Default-construct argument
c 0    // Construct a copy of the argument at address p (a + 3)
~ 0    // Destroy argument
d 0    // Default-construct argument
c 0    // Construct a copy of the argument at address p (a + 4)
~ 0    // Destroy argument
```

Once the objects are constructed, they can be used. The loop (lines 28-32)

```
for (Pointer p = a; p != a + totalObjects; ++p)
{
    cout << "\nEnter an integer value: ";
    cin >> *p;
}
```

traverses the array, prompting the user to enter a value for each Traceable<int>. If the user enters the values {7, 9, 11, 13, 15}, for example, the loop generates the output

```
Enter an integer value: 7
0 -> 7                                // cin >> *p; (Element a[0])

Enter an integer value: 9
0 -> 9                                // cin >> *p; (Element a[1])

Enter an integer value: 11
0 -> 11                               // cin >> *p; (Element a[2])

Enter an integer value: 13
0 -> 13                               // cin >> *p; (Element a[3])

Enter an integer value: 15
0 -> 15                               // cin >> *p; (Element a[4])
```

Line 35,

```
printSequence(a, a + totalObjects);
```

which is equivalent to

```
printSequence<Traceable<int>*>(a, a + totalObjects);
```

generates the output

```
7 9 11 13 15
```

Once the objects are no longer needed, they must be destroyed and the memory block must be released. The loop (lines 38-39),

```
for (Pointer p = a; p != a + totalObjects; ++p)
    allocator.destroy(p);
```

traverses the array, destroying each object. The resultant output is

```
~ 7      // Destroy *p (element a[0])
~ 9      // Destroy *p (element a[1])
~ 11     // Destroy *p (element a[2])
~ 13     // Destroy *p (element a[3])
~ 15     // Destroy *p (element a[4])
```

Line 42,

```
allocator.deallocate(a);
```

then frees the memory block, allowing the system to reuse it later.

# Part 9: Dynamic Arrays

## 9.1: Introducing the Vector Class Template

*Source files and folders*

*Vector/1*  
*Vector/common/memberFunctions\_1.h*

*Chapter outline*

- Default constructor
- Accessor functions (*empty / size / capacity, begin / end, front / back, operator[]*)
- *push\_back*
- *Destructor and clear*

This chapter introduces the `Vector` class, a runtime-resizable array. Unlike an `Array<T, Size>`, which stores its elements in an automatically-allocated array, a `Vector<T>` stores its elements in a dynamically-allocated array. The size of a `Vector` can therefore expand and contract as needed.

The class is defined in the header file `Vector.h` (lines 8-50). The member aliases (lines 12-20) are identical to those of `Array`. The data members (lines 46-49) are

`_block`, a pointer to the dynamically-allocated memory block, where the elements are stored  
`_size`, the number of elements currently stored in the block  
`_capacity`, the maximum number of elements that can be stored in the block  
`_alloc`, an Allocator used to manage the dynamically-allocated elements

The default constructor (line 22), which constructs an empty `Vector`, initializes `_block` to `null`, and `_size` and `_capacity` to 0. No special initialization is required for `_alloc` (`memberFunctions_1.h`, lines 3-10):

```
template <class T>
Vector<T>::Vector():
    _block(nullptr),
    _size(0),
    _capacity(0)
{
    // ...
}
```

The member functions `size` and `capacity` (`Vector.h`, lines 26-27) return the `Vector`'s current `_size` and `_capacity` (`memberFunctions_1.h`, lines 25-35):

```

template <class T>
inline typename Vector<T>::size_type Vector<T>::size() const
{
    return _size;
}

template <class T>
inline typename Vector<T>::size_type Vector<T>::capacity() const
{
    return _capacity;
}

```

The member functions empty, begin / end, front / back, and operator[] (Vector.h, lines 25, 28-38) are identical to those of Array (memberFunctions\_1.h, lines 19-23, 37-96).

The member function (Vector.h, line 39)

```
void push_back(const T& source);
```

inserts a copy of the given source object at the end of the Vector, thereby increasing its size by 1. The newly inserted element becomes the back element. If the memory block is full (i.e. its current size is equal to its capacity), the block is reallocated (replaced with a higher-capacity block) before inserting the new element. The capacity is increased according to the function

$$\text{newCapacity} = 1.5 \times (\text{currentCapacity} + 2)$$

The following table illustrates how the capacity of a Vector increases for the first 5 reallocations:

currentCapacity	newCapacity
0	3
3	7
7	13
13	22
22	36

Given

```
Vector<char> v;      // Default-construct v, a Vector of char
                      // v.size() and v.capacity() are 0
                      // v._block is null
```

for example, the statement

```
v.push_back('a');
```

inserts the element 'a' at the end of v. Because v is full (v.size() is equal to v.capacity()), however, its current block is first replaced with a block of capacity 3. The new element is then inserted at the end of the block (i.e. the location of the one-past-the-last element):

```
| a |   |   |           // v.size() is now 1
// v.capacity() is now 3
```

The statement

```
v.push_back('b');
```

inserts the element 'b' at the end. The block is not full (v.size() is not equal to v.capacity()), so no reallocation is necessary:

```
| a | b |   |           // v.size() is now 2
// v.capacity() is now 3
```

Similarly, the statement

```
v.push_back('c');
```

inserts the element 'c' at the end, without reallocation:

```
| a | b | c |           // v.size() is now 3
// v.capacity() is now 3
```

In the next call to push\_back,

```
v.push_back('d');
```

v is full, so the current block must first be reallocated. The new block has a capacity of 7 elements:

```
| a | b | c | d |   |   |           // v.size() is now 4
// v.capacity() is now 7
```

3 more elements can then be pushed back without reallocation, as in

```
v.push_back('e');
v.push_back('f');
v.push_back('g');
```

```
| a | b | c | d | e | f | g |           // v.size() is now 7
// v.capacity() is now 7
```

On the next call to push\_back, however, the current block will be replaced with a new block large enough to hold 13 elements:

```
v.push_back('h');

| a | b | c | d | e | f | g | h |   |   |   |   |
// v.size() is now 8
// v.capacity() is now 13
```

The private member function (Vector.h, line 43)

```
void _reallocate(size_type newCapacity);
```

replaces the Vector's current memory block with a new block, where `newCapacity` is the capacity of the new block. Reallocation entails the following steps:

- Create a new block large enough to hold `newCapacity` elements
  - Copy each element in the current block to its corresponding location in the new block
  - Destroy each element in the current block, then deallocate (release) the current block
  - Reseat the Vector's internal pointer (`_block`) to point to the new block and update the `_capacity`

The function definition is (memberFunctions\_1.h, lines 115-129)

```
template <class T>
void Vector<T>::_realloc(size_type newCapacity)
{
    T* newBlock = _alloc.allocate(newCapacity);

    for (size_type i = 0; i != size(); ++i)
    {
        _alloc.construct(newBlock + i, _block[i]);
        _alloc.destroy(_block + i);
    }

    _alloc.deallocate(_block);
    _block = newBlock;
    _capacity = newCapacity;
}
```

Consider, for example, a full `Vector<char>` containing 3 elements:

```
B           // _size is now 3
| a | b | c | // _capacity is now 3

                           // B denotes _block, which points to the first element
                           // of the current block
```

`newCapacity` is  $1.5 \times (\_capacity + 2)$ , which yields 7.5. The decimal portion is truncated, so `newCapacity` is 7. The function call

```
realloc(7);
```

entails the following operations:

```

N // N denotes newBlock
| | | | | | | |
// Copy each element in the current block to its corresponding location in
// the new block

Iteration 1 (i = 0):
_alloc.construct(newBlock + i, _block[i]); // Construct a copy of element
// _block[0] at address
// (newBlock + 0)
B
| a | b | c |
N
| a | | | | | | |
_alloc.destroy(_block + i); // Destroy the element at
// address (_block + 0)
B
| | b | c |
N
| a | | | | | | |

Iteration 2 (i = 1):
_alloc.construct(newBlock + i, _block[i]); // Construct a copy of element
// _block[1] at address
// (newBlock + 1)
B
| | b | c |
N
| a | b | | | | | |
_alloc.destroy(_block + i); // Destroy the element at
// address (_block + 1)
B
| | | c |
N
| a | b | | | | | |

Iteration 3 (i = 2):
_alloc.construct(newBlock + i, _block[i]); // Construct a copy of element
// _block[2] at address
// (newBlock + 2)
B
| | | c |
N
| a | b | c | | | | |
_alloc.destroy(_block + i); // Destroy the element at
// address (block + 2)

```

```

    B
    |   |   |   |

    N
    | a | b | c |   |   |   |

// Release the current block, then reseat _block to point to the new block

_alloc.deallocate(_block);
_block = newBlock;

    B
    | a | b | c |   |   |   |

// Update the _capacity

_capacity = newCapacity;                                // _capacity is now 7

```

Once the `_reallocate` method has been implemented, defining the `push_back` method is relatively easy (memberFunctions\_1.h, lines 98-106):

```

template <class T>
void Vector<T>::push_back(const T& source)
{
    if (size() == capacity())
        _reallocate(static_cast<size_type>(1.5 * (capacity() + 2)));

    _alloc.construct(end(), source);
    ++_size;
}

```

If the Vector is full (line 101), the block is reallocated (line 102). The expression

```
1.5 * (capacity() + 2)
```

returns the new capacity as a floating-point value (decimal number), and the expression

```
static_cast<size_type>(1.5 * (capacity() + 2))
```

truncates the decimal portion of that value (e.g. truncates the floating-point value 7.5 to 7, 13.5 to 13, etc.). More specifically, it converts the floating-point value returned by the expression

```
1.5 * (capacity() + 2)
```

to a value of type `size_type` (`unsigned int`). Note that without the use of `static_cast`, the value will be implicitly (automatically) converted, but the compiler will issue a warning:

```

_reallocate(1.5 * (capacity() + 2));

// Compiler warning: _reallocate's function parameter (newCapacity) is
// of type size_type (unsigned int)

```

```
// The argument being passed is a floating-point value
// Possible loss of data (e.g. 7.5 will become 7, 13.5 will become 13, etc.)
```

Line 104 constructs a copy of the source object at the address of the one-past-the-last element, and line 105 updates the `_size`.

The private member function (Vector.h, line 44)

```
void _destroyAllElements();
```

destroys each element in the block (memberFunctions\_1.h, lines 131-136):

```
template <class T>
void Vector<T>::_destroyAllElements()
{
    for (size_type i = 0; i != size(); ++i) // Destroy each element in the
        _alloc.destroy(_block + i);           // range [_block, _block + _size)
}
```

The member function (Vector.h, line 40)

```
void clear();
```

destroys all of the elements in the Vector and resets its size to 0 (memberFunctions\_1.h, lines 108-113):

```
template <class T>
inline void Vector<T>::clear()
{
    _destroyAllElements();
    _size = 0;
}
```

Note, however, that it does not release the block, and the capacity is unchanged. This ensures that subsequent calls to `push_back` will require fewer reallocations. Given the `Vector<char> v`

```
| a | b | c | d | e | f | g | h |   |   |   |   |
```

```
// v.size() is 8
// v.capacity() is 13
```

for example, the statement

```
v.clear();
```

results in

```
|   |   |   |   |   |   |   |   |   |   |
```

```
// v.size() is 0
// v.capacity() is 13
// 13 elements can be pushed back without reallocation
```

The destructor (`Vector.h`, line 23) destroys all of the elements and releases the memory block (`memberFunctions_1.h`, lines 12-17):

```
template <class T>
inline Vector<T>::~Vector()
{
    _destroyAllElements();
    _alloc.deallocate(_block);
}
```

The program in this chapter demonstrates the `push_back` method. Line 12 (`main.cpp`),

```
Vector<Traceable<char>> v;
```

constructs `v`, a `Vector` of `Traceable<char>`. Each iteration of the loop (lines 14-36) prompts the user to enter a single-character command, 'p' (to `push_back` a new element) or 'q' (to quit) (lines 16-18).

If the user enters the command 'p' (line 20), the program prompts the user to enter a character, which is then pushed back into the `Vector` (lines 22-26). All of the elements are then displayed via `printSequence` (lines 28-30), after which the next iteration begins.

If the user enters any other command (line 32), the loop is exited (line 34), after which `main` terminates and `v` (an automatic object) is automatically destroyed. `v`'s destructor destroys all of its elements and releases its memory block.

A sample run of the program is

```
(p)ush_back (q)uit: p

Enter a character to push_back: a
v a                                         // Construct argument passed to push_back
c a                                         // Construct copy of argument in block
~ a                                         // Destroy argument

v contains 1 element(s):
a

(p)ush_back (q)uit: p

Enter a character to push_back: b
v b                                         // Construct argument passed to push_back
c b                                         // Construct copy of argument in block
~ b                                         // Destroy argument

v contains 2 element(s):
a b

(p)ush_back (q)uit: p

Enter a character to push_back: c
v c                                         // Construct argument passed to push_back
```

```

c c                                // Construct copy of argument in block
~ c                                // Destroy argument

v contains 3 element(s):
a b c

(p)ush_back (q)uit: p

Enter a character to push_back: d
v d                                // Construct argument passed to push_back
c a                                // Construct copy of 'a' in new block
~ a                                // Destroy 'a' in current block
c b                                // Construct copy of 'b' in new block
~ b                                // Destroy 'b' in current block
c c                                // Construct copy of 'c' in new block
~ c                                // Destroy 'c' in current block
c d                                // Construct copy of argument in new block
~ d                                // Destroy argument

v contains 4 element(s):
a b c d

(p)ush_back (q)uit: p

Enter a character to push_back: e
v e                                // Construct argument passed to push_back
c e                                // Construct copy of argument in block
~ e                                // Destroy argument

v contains 5 element(s):
a b c d e

(p)ush_back (q)uit: p

Enter a character to push_back: f
v f                                // Construct argument passed to push_back
c f                                // Construct copy of argument in block
~ f                                // Destroy argument

v contains 6 element(s):
a b c d e f

(p)ush_back (q)uit: p

Enter a character to push_back: g
v g                                // Construct argument passed to push_back
c g                                // Construct copy of argument in block
~ g                                // Destroy argument

v contains 7 element(s):
a b c d e f g

(p)ush_back (q)uit: p

```

```
Enter a character to push_back: h
v h // Construct argument passed to push_back
c a // Construct copy of 'a' in new block
~ a // Destroy 'a' in current block
c b // Construct copy of 'b' in new block
~ b // Destroy 'b' in current block
c c // Construct copy of 'c' in new block
~ c // Destroy 'c' in current block
c d // Construct copy of 'd' in new block
~ d // Destroy 'd' in current block
c e // Construct copy of 'e' in new block
~ e // Destroy 'e' in current block
c f // Construct copy of 'f' in new block
~ f // Destroy 'f' in current block
c g // Construct copy of 'g' in new block
~ g // Destroy 'g' in current block
c h // Construct copy of argument in new block
~ h // Destroy argument

v contains 8 element(s):
a b c d e f g h

(p)ush_back (q)uit: q
~ a
~ b
~ c
~ d
~ e
~ f
~ g
~ h

// v's destructor destroys all of its
// elements, then releases its memory
// block
```

## 9.2: Reserve, Pop Back, Copy, and Assignment

*Source files and folders*

```
printContainer
Vector/2
Vector/common/memberFunctions_2.h
```

*Chapter outline*

- *reserve*
- *pop\_back*
- *Copy constructor and assignment operator*

Recall from the previous chapter that inserting an element into a full Vector involves a computationally-expensive reallocation: before the new element can be inserted, all of the existing elements must first be copied to a new, higher-capacity memory block. These reallocations can be avoided, however, by allocating a higher-capacity memory block in advance. The member function (Vector.h, line 41)

```
void reserve(size_type newCapacity);
```

reallocates the Vector's current block to a new block with the given capacity. If the newCapacity is less than or equal to the Vector's current capacity, the function does nothing (memberFunctions\_2.h, lines 38-43)

```
template <class T>
void Vector<T>::reserve(size_type newCapacity)
{
    if (newCapacity > capacity())      // Reallocate to the newCapacity, only if
        _reallocate(newCapacity);       // it exceeds the current capacity
}
```

Given

```
Vector<Student> v;
v.reserve(100);
```

for example, 100 elements can be inserted before the Vector must be reallocated.

The member function (Vector.h, line 43)

```
void pop_back();
```

removes the back element from the Vector, decreasing its size by 1. The element is destroyed by calling its destructor. Given the Vector

```
| a | b | c | d | e | f |   |   | // The back element is 'f'
// _size is 6
// _capacity is 9
```

for example, calling pop\_back results in

```
| a | b | c | d | e |   |   |   | // The back element is 'e'
// _size is 5
// _capacity is 9
```

Note that the \_capacity is unchanged (memberFunctions\_2.h, lines 45-50):

```
template <class T>
inline void Vector<T>::pop_back()
{
    _alloc.destroy(&back());
    --_size;
}
```

Recall that the member function back returns a reference to the back element. The expression &back() therefore returns the address of the back element, which is passed to the Allocator's destroy method.

The copy constructor (Vector.h, line 23),

```
Vector(const Vector& source);
```

constructs a Vector as a copy of an existing Vector. The parameter, source, is a const reference to the source object. The newly constructed Vector contains the same elements as the source Vector; more specifically, it contains copies of the elements in the source Vector (memberFunctions\_2.h, lines 3-13):

```
template <class T>
Vector<T>::Vector(const Vector& source):
    _block(nullptr),
    _size(0),
    _capacity(0)
{
    _reallocate(source.capacity());

    for (const iterator i = source.begin(); i != source.end(); ++i)
        push_back(*i);
}
```

Lines 5-7 construct the new Vector as an empty Vector (like the default constructor). Line 9 reserves the same capacity in the new Vector as that of the source Vector. The loop (lines 11-12) then inserts (a copy of) each element from the source Vector into the new Vector. Because sufficient capacity has already been reserved, none of these calls to push\_back will require reallocation.

The assignment operator (Vector.h, line 40)

```
Vector& operator=(const Vector& rhs);
```

sets the value of each element in the left operand to that of its counterpart in the right operand. If the left and right operands are of unequal size, elements are pushed / popped as necessary. Given

a   b   c   d   e   f     P   Q   R	// Vector<char> x // Vector<char> y
--	--

for example, the statement

```
x = y; // x.operator=(y);
```

results in

P   Q   R     P   Q   R	// Vector<char> x // Vector<char> y
----------------------------	--

x.size() is greater than y.size(), so the excess elements {d, e, f} are popped off of x. The remaining elements in x {a, b, c} are then assigned the values {P, Q, R}. Similarly, given

a   b   c     P   Q   R   S   T   U	// Vector<char> x // Vector<char> y
--	--

the statement

```
x = y;
```

results in

P   Q   R   S   T   U     P   Q   R   S   T   U	// Vector<char> x // Vector<char> y
--	--

x.size() is less than y.size(), so the existing elements in x {a, b, c} are assigned the values {P, Q, R}. The remaining elements in y {S, T, U} are then pushed back into x. The function is defined in lines 15-36 (memberFunctions\_2.h). Lines 18-25,

```
if (size() >= rhs.size())
{
    while (size() != rhs.size())
        pop_back();

    for (size_type i = 0; i != size(); ++i)
        (*this)[i] = rhs[i];
}
```

handle the first case. If the left operand is larger than (or the same size as) the right operand, elements are popped off the left operand until it is the same size as the right operand (lines 20-21). Each element in the left operand is then assigned the value of its counterpart in the right operand (lines 23-24). Lines 26-33,

```

else
{
    for (size_type i = 0; i != size(); ++i)
        (*this)[i] = rhs[i];

    for (size_type i = size(); i != rhs.size(); ++i)
        push_back(rhs[i]);
}

```

handle the second case. If the left operand is smaller (contains fewer elements) than the right operand, each existing element in the left operand is assigned the value of its counterpart in the right operand (lines 28-29). The remaining elements in the right operand are then pushed back into the left operand (lines 31-32).

The function (printContainer.h, lines 10-11)

```

template <class Container>
void printContainer(const Container& container);

```

prints each element in the range [container.begin(), container.end()) by calling printSequence, then prints a new line (lines 16-21):

```

template <class Container>
inline void printContainer(const Container& container)
{
    printSequence(container.begin(), container.end());
    std::cout << std::endl;
}

```

Because the parameter, container, is a const reference, the compiler selects the const versions of container's begin and end methods. The statement

```
printSequence(container.begin(), container.end());
```

is therefore equivalent to

```
printSequence<Container::const_iterator>(container.begin(), container.end());
```

Given

```

Vector<int> v;

// ...

printContainer(v);

```

for example, the compiler generates the code

```
inline void printContainer(const Vector<int>& container)
{
    printSequence<const int*>(container.begin(), container.end());
    std::cout << std::endl;
}
```

Similarly, the function `printContainerReverse` (lines 13-14, 23-28) prints each element in the range `[container.rbegin(), container.rend()]`. The member functions `rbegin` (“reverse begin”) and `rend` (“reverse end”) will be implemented in a later section.

The program in this chapter demonstrates each of the new member functions. `main` simply calls the functions (`main.cpp`, lines 10-13, 20-23)

```
void testReserveAndPopBack();
void testCopyConstructor();
void testAssignmentOperatorCase1();
void testAssignmentOperatorCase2();
```

Line 30 declares `_Vector` as an alias of the type `Vector<Traceable<char>>`. The first function, `testReserveAndPopBack`, constructs a `_Vector` `v` and reserves enough space for 8 elements (lines 38-39). Lines 41-48 then push back the elements `{a, b, c, d, e, f, g, h}`. The resultant output,

```
v a      // v.push_back('a');
c a
~ a
v b      // v.push_back('b');
c b
~ b
v c      // v.push_back('c');
c c
~ c
v d      // v.push_back('d');
c d
~ d
v e      // v.push_back('e');
c e
~ e
v f      // v.push_back('f');
c f
~ f
v g      // v.push_back('g');
c g
~ g
v h      // v.push_back('h');
c h
~ h
```

verifies that all 8 elements were inserted without reallocating the memory block. The loop (lines 52-53) then pops the back element off of `v` until `v` is empty, generating the output

```
~ h      // v.pop_back()
```

```

~ g    // v.pop_back()
~ f    // v.pop_back()
~ e    // v.pop_back()
~ d    // v.pop_back()
~ c    // v.pop_back()
~ b    // v.pop_back()
~ a    // v.pop_back()

```

The next function, testCopyConstructor, constructs a `_Vector` `v`, inserts the elements `{a, b, c, d}`, then constructs a new `_Vector` `w` as a copy of `v` (lines 64-72). Constructing `w` results in the output

```

c a    // Copy source element 'a' from v
c b    // Copy source element 'b' from v
c c    // Copy source element 'c' from v
c d    // Copy source element 'd' from v

```

Because the copy constructor initializes `w` with the same capacity as `v`, no reallocations take place. Line 75 verifies that `w` contains the elements `{a, b, c, d}` by calling `printContainer`. When the function terminates, `w` is automatically destroyed, followed by `v`, generating the output

```

~ a    // w's destructor
~ b
~ c
~ d
~ a    // v's destructor
~ b
~ c
~ d

```

The next function, `testAssignmentOperatorCase1`, demonstrates the assignment operator when the left operand is larger (contains more elements than) the right operand. `_Vector` `x` contains the elements `{A, B, C, D, E}` and `_Vector` `y` contains the elements `{1, 2, 3}` (lines 85-97). The statement (line 100)

```
x = y;
```

generates the output

```

~ E          // Pop the excess elements {D,E} off of x
~ D
A -> 1    // Assign the remaining elements {A,B,C} in x the values of the
B -> 2    // corresponding elements in y
C -> 3

```

Line 103 prints `x`, verifying that it contains the same elements as `y`. When the function terminates, `x` and `y` go out of scope, and their destructors generate the output

```

~ 1    // y's destructor
~ 2
~ 3
~ 1    // x's destructor
~ 2

```

```
~ 3
```

The last function, `testAssignmentOperatorCase2`, demonstrates the assignment operator when the left operand is smaller (contains fewer elements than) the right operand. `_Vector x` contains the elements `{A, B, C}` and `_Vector y` contains the elements `{1, 2, 3, 4, 5}` (lines 113-125). The statement (line 128)

```
x = y;
```

generates the output

```
A -> 1      // Assign the existing elements in x {A,B,C} the values of the
B -> 2      // corresponding elements in y
C -> 3
c 1          // Push back the remaining elements in y {4,5} into x
~ 1          //   The existing elements in x {1,2,3} are first reallocated
c 2          //   because x's memory block is full
~ 2
c 3
~ 3
c 4          //   Construct a copy of element '4' in x
c 5          //   Construct a copy of element '5' in y
```

Line 131 verifies that `x` contains the same elements as `y`, `{1, 2, 3, 4, 5}`. When the function terminates, both `_Vectors` are destroyed, generating the output

```
~ 1          // y's destructor
~ 2
~ 3
~ 4
~ 5
~ 1          // x's destructor
~ 2
~ 3
~ 4
~ 5
```



## 9.3: Insert and Erase

*Source files and folders*

*Vector/3*

*Vector/common/memberFunctions\_3.h*

*Chapter outline*

- *insert*
- *erase*

Inserting an element in the middle of a Vector (i.e. anywhere besides the end) involves inserting it at the end (via `push_back`), then shifting it to the desired insertion point. The shifting procedure is similar to that of insertion sort (Chapter 5.6), where a given element is shifted towards the beginning of a sequence by repeatedly swapping it with its preceding element. Consider, for example, the Vector

```
a b c d e f
    i           // i is in iterator to element d
```

To insert a new element X before d, X is first pushed back:

```
a b c d e f X
    i           // p is an iterator to X, the new back element
```

X is then repeatedly swapped with its preceding element until it reaches the desired insertion point:

Iteration 1:

```
swap(*p, *(p - 1));
```

```
a b c d e p X f
    i
```

```
--p;
```

```
a b c d e X p f
    i
```

Iteration 2:

```
swap(*p, *(p - 1));
```

```
a b c d X p e f
    i
```

```
--p;
      p
a b c d X e f
      i

Iteration 3:

swap(*p, *(p - 1));

      p
a b c X d e f
      i

--p;

      p
a b c X d e f
      i

// p has reached the desired insertion point (i), so the insertion is
// complete
```

The member function (Vector.h, line 42)

```
iterator insert(const_iterator insertionPoint, const T& source);
```

inserts a copy of the given source object before insertionPoint, then returns an iterator to the newly inserted element (memberFunctions\_3.h, lines 5-19):

```
template <class T>
typename Vector<T>::iterator Vector<T>::insert(const_iterator insertionPoint,
    const T& source)
{
    size_type insertionDistance = insertionPoint - begin();

    push_back(source);

    iterator i = begin() + insertionDistance;

    for (iterator p = end() - 1; p != i; --p)
        swap(*p, *(p - 1));

    return i;
}
```

If the block is at full capacity, push\_back (line 11) will incur a reallocation, which will invalidate insertionPoint (because its referent element will be moved to a different address). It is therefore necessary to first obtain the distance (number of elements) from the beginning of the sequence to the insertionPoint (line 9). Initializing i in terms of this distance (line 13) guarantees that it will point to the correct address, even if a reallocation occurred on push\_back:

```

a b c d e f      // Original sequence
      ^           // insertionPoint (^) refers to element d
      // insertionDistance (the distance from a to d) is 3

push_back('X');

a b c d e f X  // New sequence
      // If a reallocation occurred, insertionPoint no longer
      // refers to the correct address

iterator i = begin() + insertionDistance;    // i = begin() + 3;

a b c d e f X  // Shift X to location i...
      ^           i

```

Erasing an element from the middle involves overwriting it with its proceeding element until the back of the sequence is reached, then removing the last element via `pop_back`. Consider, for example, the Vector

```

a b c X d e f
      p           // p is an iterator to element X

```

To erase element X, each element from X to e (inclusive) is overwritten with the value of its proceeding element:

Iteration 1:

```

*p = *(p + 1);

a b c d d e f
      p

++p;

a b c d d e f
      p

```

Iteration 2:

```

*p = *(p + 1);

a b c d e e f
      p

++p;

a b c d e e f
      p

```

Iteration 3:

```

*p = *(p + 1);

```

```

a b c d e f f
    p

++p;

a b c d e f f
    p

// p has reached the back of the sequence
// Remove the last (redundant) element via pop_back

a b c d e f
    p

```

The member function (Vector.h, line 43)

```
iterator erase(const_iterator erasurePoint);
```

erases the element at `erasurePoint`, then returns an iterator to the element proceeding the erased element (memberFunctions\_3.h, lines 21-32):

```

template <class T>
typename Vector<T>::iterator Vector<T>::erase(const_iterator erasurePoint)
{
    size_type erasureDistance = erasurePoint - begin();

    for (iterator p = begin() + erasureDistance; p != end() - 1; ++p)
        *p = *(p + 1);

    pop_back();

    return begin() + erasureDistance;
}

```

`erasureDistance` (line 24) is the distance (number of elements) from the beginning of the sequence to the `erasurePoint`:

```

a b c X d e f
    ^          // erasurePoint (^) is an iterator to element X
    p          // erasureDistance (the distance from a to X) is 3
              // p = begin() + erasureDistance;

```

`p` must be initialized in terms of `erasureDistance` (line 26) because an iterator (`T*`) cannot be constructed from a `const_iterator` (`const T*`). In other words, attempting to write

```
iterator p = erasurePoint;      // Construct an iterator (p) from a
                               // const_iterator (erasurePoint)
```

results in a compiler error. It is, however, possible to construct a `const_iterator` from an iterator, as in

```
iterator i = begin();
```

```
const_iterator ci = i;           // Construct a const_iterator (ci) from an
                                // iterator (i) (ok)
```

The program in this chapter demonstrates the aforementioned examples. Lines 12-21 (main.cpp) construct the Vector (Vector<Traceable<char>>)

```
a b c d e f
```

Line 28,

```
Vector::iterator newElement = v.insert(v.begin() + 3, 'X');
```

inserts a new element X at v.begin() + 3 (before d), after which newElement refers to X. The resultant output is

```
v X          // Construct argument passed to insert
c X          // Construct copy of argument at end of block (push_back)
c X          // swap(*p, *(p - 1));
    X -> f
    f -> X
~ X
c X          // swap(*p, *(p - 1));
    X -> e
    e -> X
~ X
c X          // swap(*p, *(p - 1));
    X -> d
    d -> X
~ X
~ X          // Destroy argument passed to insert
```

Lines 34-36 print the newly inserted element (X), followed by its preceding and proceeding elements (c and d). Line 39,

```
Vector::iterator proceedingElement = v.erase(newElement);
```

erases X, after which proceedingElement refers to d. The resultant output is

```
x -> d      // *p = *(p + 1);
d -> e      // *p = *(p + 1);
e -> f      // *p = *(p + 1);
~ f          // pop_back();
```

Lines 45-49 print the proceeding and preceding elements (d and c). When main terminates, v's destructor destroys the remaining elements, {a, b, c, d, e, f}, then releases the memory block.



# Part 10: Linked Lists

## 10.1: Introducing the List Class Template

*Source files and folders*

*List/1*  
*List/common/ListNode.h*  
*List/common/memberFunctions\_1.h*

*Chapter outline*

- *The ListNode class*
- *Default constructor*
- *Accessor functions (empty / size, front / back)*
- *push\_back / push\_front*
- *Destructor*

Recall from the previous chapter that inserting or erasing an element in the middle of a Vector can be a computationally-expensive operation. In the worst-case scenario, the required number of swaps / overwrites is directly proportional to the size of the container. Similarly, whenever a reallocation occurs, every single element must be copied and destroyed. In the Vector

a b c d e f

for example, inserting an element before 'a' requires 6 (size) swaps, while erasing element 'a' requires 5 (size - 1) overwrites. Reallocating the memory block requires 6 (size) copy-constructions and destructions.

This chapter introduces the List class template, which addresses these issues. A List is a type of deque (pronounced “deck,” short for “double-ended queue”). A deque is a container that supports the efficient insertion and removal of elements at both ends, front and back.

Unlike a Vector, which stores all of its elements in a single memory block, a List stores each element in a separate node. A “node” is an object that contains a single element and two pointers: one to the preceding node (“left”), and one to the proceeding node (“right”). The leftmost node (the node containing the front element) is called the “head,” while the rightmost node (the node containing the back element) is called the “tail.”

The ListNode class is defined as (ListNode.h, lines 6-15)

```

template <class T>
struct ListNode
{
    ListNode(const T& sourceElement);

    T element;                                // The contained element of type T

    ListNode* left;                            // Pointer to the preceding (left) node
    ListNode* right;                           // Pointer to the proceeding (right) node
};

```

The constructor (line 9) constructs a `ListNode`, initializing its element as a copy of the given `sourceElement`, and its left / right pointers to null (lines 17-24):

```

template <class T>
inline ListNode<T>::ListNode(const T& sourceElement):
    element(sourceElement),
    left(nullptr),
    right(nullptr)
{
    // ...
}

```

Consider, for example, a `List<char>` containing the elements {A, B, C, D, E}:

```

      _head                               _tail
|-'A' |   |-'B' |   |-'C' |   |-'D' |   |-'E' |
x <-| left | <-| left | <-| left | <-| left | <-| left |
     | right | -->| right | -->| right | -->| right | -->| right | -> x
// x denotes null

```

The node containing element 'A' is the head, and the node containing element 'E' is the tail. The List contains two pointers, `_head` (a pointer to the head) and `_tail` (a pointer to the tail). The expressions

```

_head->element
_tail->element

```

for example, return elements 'A' and 'E'. The expression

```

_head->right

```

returns a pointer to the node containing element 'B'. The expression

```

_head->right->element

```

therefore returns element 'B'. Similarly, the expression

```

_tail->left

```

returns a pointer to the node containing element 'D', and the expression

```
_tail->left->element
```

returns element 'D'. The expressions

```
_head->right->right  
_tail->left->left
```

both return pointers to the node containing element 'C', and the expressions

```
_head->right->right->element  
_tail->left->left->element
```

both return the element 'C'. The pointers

```
_head->left  
_tail->right
```

are both null.

The List class is defined in the header file List.h (lines 9-45). The aliases declared in lines 13-19 are identical to those of Vector. Line 35 declares Node as an alias of the type ListNode<T>. The data members are declared in lines 41-44,

```
Node* _head;           // ListNode<T>* _head; (Pointer to the head)
Node* _tail;           // ListNode<T>* _tail; (Pointer to the tail)
size_type _size;        // Number of elements currently stored
Allocator<Node> _alloc; // Allocator<ListNode<T>> _alloc; (Allocator used
                        // to manage the Nodes)
```

The default constructor (line 21) constructs an empty List by initializing \_head and \_tail to null, and \_size to 0 (memberFunctions\_1.h, lines 3-10).

The member functions empty and size (List.h, lines 24-25) are identical to those of Vector (memberFunctions\_1.h, lines 18-28).

The member functions front and back (List.h, lines 26-27, 29-30) return references to the head and tail elements (memberFunctions\_1.h, lines 30-52).

The private member function (List.h, line 37)

```
Node* _createNode(const T& source);
```

creates a new Node, initializing its element as a copy of the given source object. The function returns a pointer to the new Node (memberFunctions\_1.h, lines 96-103):

```

template <class T>
typename List<T>::Node* List<T>::_createNode(const T& source)
{
    Node* newNode = _alloc.allocate(1);
    _alloc.construct(newNode, Node(source));

    return newNode;
}

```

Line 99 reserves a block of memory large enough to hold a single Node and stores the address in the pointer `newNode`. Line 100 constructs a temporary, unnamed Node,

```
Node(source)
```

and passes it to the allocator's `construct` method, which constructs a copy of this Node in the newly reserved memory block. After the `construct` method terminates, the temporary Node is destroyed. Line 102 then returns a pointer to the newly created Node.

The member function (List.h, line 32)

```
void push_back(const T& source);
```

inserts a copy of the given source object at the end of the List, increasing its size by 1. The procedure for pushing back an element is

```

Create a new node containing the given element;

If the List is empty
{
    Set _head and _tail to point to the new node;
}
Otherwise
{
    Set the new node's left link to point to the tail;

    Update the tail's right link to point to the new node;
    Update _tail to point to the new node;
}

Increment the _size;

```

Consider, for example, an empty `List<char>`. To push back the first element A:

```

// Create a new node containing the given element

Node* newNode = _createNode('A');      // newNode is a pointer to the new node

        newNode
        | 'A' |
null <- | left  |
        | right | -> null

```

```
// Set _head and _tail to point to the new node

_head = newNode;
_tail = newNode;

    _head
    _tail
    | 'A' |
null <- | left |
        | right | -> null
```

To push back a second element B:

```
// Create a new node containing the given element

Node* newNode = _createNode('B');

    _head
    _tail          newNode
    | 'A' |          | 'B' |
null <- | left |  null <- | left |
        | right | -> null  | right | -> null

// Set the new node's left link to point to the tail

newNode->left = _tail;

    _head
    _tail          newNode
    | 'A' |          | 'B' |
null <- | left | <----- | left |
        | right | -> null  | right | -> null

// Update the tail's right link to point to the new node

_tail->right = newNode;

    _head
    _tail          newNode
    | 'A' |          | 'B' |
null <- | left | <----- | left |
        | right | -----> | right | -> null

// Update _tail to point to the new node

_tail = newNode;

    _head          _tail
    | 'A' |          | 'B' |
null <- | left | <----- | left |
        | right | -----> | right | -> null
```

To push back a third element C:

```

// Create a new node containing the given element

Node* newNode = _createNode('C');

    | _head           | _tail           | newNode
    | 'A'   |           | 'B'   |           | 'C'   |
null <- | left  | <----- | left  | null <- | left  |
        | right | -----> | right | -> null | right | -> null

// Set the new node's left link to point to the tail

newNode->left = _tail;

    | _head           | _tail           | newNode
    | 'A'   |           | 'B'   |           | 'C'   |
null <- | left  | <----- | left  | <----- | left  |
        | right | -----> | right | -> null | right | -> null

// Update the tail's right link to point to the new node

_tail->right = newNode;

    | _head           | _tail           | newNode
    | 'A'   |           | 'B'   |           | 'C'   |
null <- | left  | <----- | left  | <----- | left  |
        | right | -----> | right | -----> | right | -> null

// Update _tail to point to the new node

_tail = newNode;

    | _head           |           | _tail
    | 'A'   |           | 'B'   |           | 'C'   |
null <- | left  | <----- | left  | <----- | left  |
        | right | -----> | right | -----> | right | -> null

```

The full definition of push\_back is (memberFunctions\_1.h, lines 75-94)

```

template <class T>
void List<T>::push_back(const T& source)
{
    Node* newNode = _createNode(source);

    if (empty())
    {
        _head = newNode;
        _tail = newNode;
    }
    else
    {
        newNode->left = _tail;
        _tail->right = newNode;
        _tail = newNode;
    }

    ++_size;
}

```

The member function (List.h, line 31)

```
void push_front(const T& source);
```

inserts a copy of the given source object at the front of the List, increasing its size by 1. The procedure is the mirror image of push\_back:

```

Create a new node containing the given element;

If the List is empty
{
    Set _head and _tail to point to the new node;
}
Otherwise
{
    Set the new node's right link to point to the head;

    Update the head's left link to point the new node;
    Update _head to point to the new node;
}

Increment the _size;

```

To push front the first element A into an empty List, for example:

```
// Create a new node containing the given element

Node* newNode = _createNode('A');
```

```

        newNode
        | 'A' |
null <- | left |
        | right | -> null

// Set _head and _tail to point to the new node

_head = newNode;
_tail = newNode;

        _head
        _tail
        | 'A' |
null <- | left |
        | right | -> null

```

To push front a second element B:

```

// Create a new node containing the given element

Node* newNode = _createNode('B');

        newNode
        | 'B' |           _head
                           _tail
null <- | left |   null <- | left |
        | right | -> null | right | -> null

// Set the new node's right link to point to the head

newNode->right = _head;

        newNode
        | 'B' |           _head
                           _tail
null <- | left |   null <- | left |
        | right | -----> | right | -> null

// Update the head's left link to point to the new node

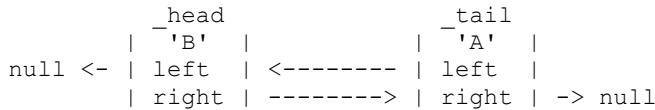
_head->left = newNode;

        newNode
        | 'B' |           _head
                           _tail
null <- | left | <----- | left |
        | right | -----> | right | -> null

// Update _head to point to the new node

_head = newNode;

```



To push front a third element C:

```

// Create a new node containing the given element

Node* newNode = _createNode('C');

    newNode           _head           _tail
    | 'C'  |           | 'B'  |           | 'A'  |
null <- | left  |   null <- | left  | <----- | left   |
        | right | -> null     | right | -----> | right  | -> null

// Set the new node's right link to point to the head

newNode->right = _head;

    newNode           _head           _tail
    | 'C'  |           | 'B'  |           | 'A'  |
null <- | left  |   null <- | left  | <----- | left   |
        | right | -----> | right | -----> | right  | -> null

// Update the head's left link to point to the new node

_head->left = newNode;

    newNode           _head           _tail
    | 'C'  |           | 'B'  |           | 'A'  |
null <- | left  | <----- | left  | <----- | left   |
        | right | -----> | right | -----> | right  | -> null

// Update _head to point to the new node

_head = newNode;

    _head           _tail
    | 'C'  |           | 'A'  |
null <- | left  | <----- | left  | <----- | left   |
        | right | -----> | right | -----> | right  | -> null

```

The full definition of `push_front` is (memberFunctions\_1.h, lines 54-73)

```

template <class T>
void List<T>::push_front(const T& source)
{
    Node* newNode = _createNode(source);

    if (empty())
    {
        _head = newNode;
        _tail = newNode;
    }
    else
    {
        newNode->right = _head;
        _head->left = newNode;
        _head = newNode;
    }

    ++_size;
}

```

The private member function (List.h, line 38)

```
void _destroyNode(Node* n);
```

destroys the given Node and releases its memory block (memberFunctions\_1.h, lines 105-110):

```

template <class T>
inline void List<T>::_destroyNode(Node* n)
{
    _alloc.destroy(n);           // Destroy the node
    _alloc.deallocate(n);       // Release its memory block
}

```

The private member function (List.h, line 39)

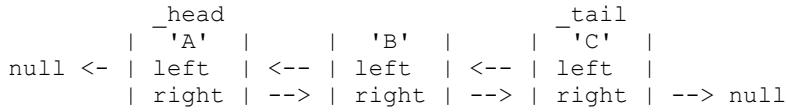
```
void _destroyAllNodes();
```

destroys all Nodes in the List and releases their memory blocks (memberFunctions\_1.h, lines 112-124):

```
template <class T>
void List<T>::_destroyAllNodes()
{
    Node* n = _head;
    Node* next;

    while (n != nullptr)
    {
        next = n->right;
        _destroyNode(n);
        n = next;
    }
}
```

Given the List

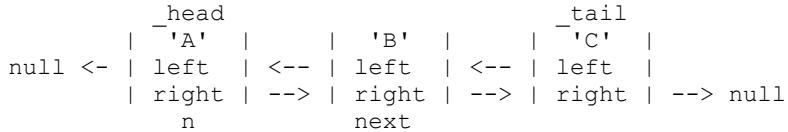


for example, `_destroyAllNodes` performs the following operations:

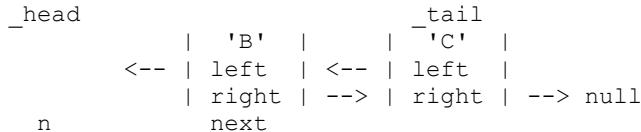
```
Node* n = _head;
Node* next;
```

Iteration 1:

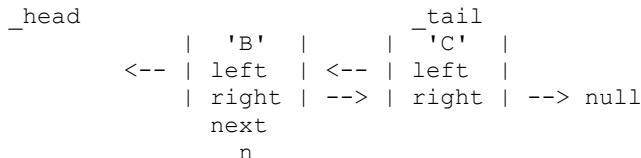
```
next = n->right;
```



```
_destroyNode(n);
```

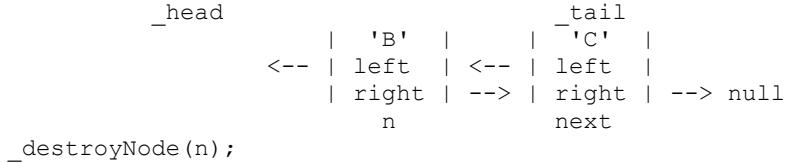


```
n = next;
```



Iteration 2:

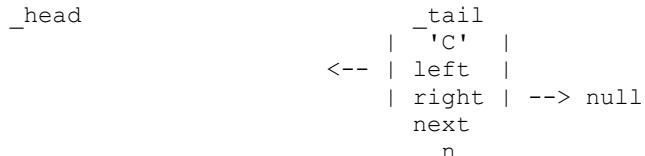
```
next = n->right;
```



```
_destroyNode(n);
```

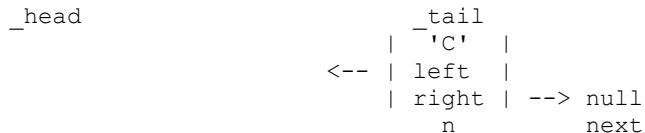


```
n = next;
```



Iteration 3:

```
next = n->right;
```



```
_destroyNode(n);
```



```
n = next;
```



```
// n is null, so the loop terminates
```

The destructor (List.h, line 22) destroys the List by simply calling `_destroyAllNodes` (memberFunctions\_1.h, lines 12-16).

The program in this chapter constructs two empty Lists, then pushes some elements to the front and back of each one. Lines 11-14 (main.cpp) construct the Lists s and t. Lines 16-23 insert the sequence {Z, Y, X, A, B, C} into s. The resultant output is

```

v A      // Construct argument passed to push_back
c A      // Construct argument passed to _alloc.construct
c A      // Construct copy of argument in List
~ A      // Destroy argument passed to _alloc.construct
~ A      // Destroy argument passed to push_back
v B      // Repeat for 'B'
c B
c B
~ B
~ B
v C      // Repeat for 'C'
c C
c C
~ C
~ C

v X      // Construct argument passed to push_front
c X      // Construct argument passed to _alloc.construct
c X      // Construct copy of argument in List
~ X      // Destroy argument passed to _alloc.construct
~ X      // Destroy argument passed to push_front
v Y      // Repeat for 'Y'
c Y
c Y
~ Y
~ Y
v Z      // Repeat for 'Z'
c Z
c Z
~ Z
~ Z

```

Similarly, lines 26-33 insert the sequence {C, B, A, X, Y, Z} into t.

When main terminates, t is destroyed, followed by s, generating the output

```

~ C      // Destroy t
~ B
~ A
~ X
~ Y
~ Z
~ Z      // Destroy s
~ Y
~ X

```

~ A  
~ B  
~ C

## 10.2: Pop Front, Pop Back, and Clear

*Source files and folders*

*List/2*  
*List/common/memberFunctions\_2.h*

*Chapter outline*

- *clear*
- *pop\_back / pop\_front*

The member function *clear* (*List.h*, line 35) removes all the elements from the List (*memberFunctions\_2.h*, lines 33-41):

```
template <class T>
void List<T>::clear()
{
    _destroyAllNodes();

    _head = nullptr;
    _tail = nullptr;
    _size = 0;
}
```

The member function *pop\_back* (*List.h*, line 34) removes the back element, decreasing the size of the List by 1. The procedure is

```
Obtain a pointer to the new tail (the current tail's left neighbor);

If the new tail is not null,
    update its right link to point to null;
Otherwise,
    the List is about to become empty, so update _head to point to null;

Destroy the current tail;
Update _tail to point to the new tail;
Decrement the _size;
```

Given the List

```
      _head          _tail
      | 'A' |          | 'B' |
null <- | left | <-- | left |          |
      | right | --> | right | --> null
```

for example, *pop\_back* performs the following operations:

```
// Obtain a pointer to the new tail (the current tail's left neighbor)
```

```

Node* newTail = _tail->left;

    newTail
      _head           _tail
    | 'A'  |           | 'B'  |
null <- | left  | <----- | left  |
      | right | -----> | right | -> null

// The new tail is not null, so update its right link to point to null

newTail->right = nullptr;

    newTail
      _head           _tail
    | 'A'  |           | 'B'  |
null <- | left  | <----- | left  |
      | right | -> null  | right | -> null

// Destroy the current tail

_destroyNode(_tail);

    newTail
      _head           _tail
    | 'A'  |           | 'B'  |
null <- | left  |           | right | -> null

// Update _tail to point to the new tail

_tail = newTail;

    _head
    _tail
  | 'A'  |
null <- | left  |
      | right | -> null

```

To pop back the next element:

```

// Obtain a pointer to the new tail (the current tail's left neighbor)

Node* newTail = _tail->left;

newTail   _head
          _tail
        | 'A'  |
null <- | left  |
          | right | -> null

// The new tail is null, so the List is about to become empty
// Update _head to point to null

_head = nullptr;

```

```

newTail
_head      _tail
|   |-'A'   |
null <-| left   |
| right  | -> null

// Destroy the current tail

_destroyNode(_tail);

newTail
_head      _tail
null

// Update _tail to point to the new tail

_tail = newTail;

_head           // _head and _tail are now null
_tail

null

```

`pop_back` is defined as (`memberFunctions_2.h`, lines 18-31)

```

template <class T>
void List<T>::pop_back()
{
    Node* newTail = _tail->left;

    if (newTail != nullptr)
        newTail->right = nullptr;
    else
        _head = nullptr;

    _destroyNode(_tail);
    _tail = newTail;
    --_size;
}

```

The member function `pop_front` (`List.h`, line 33) removes the front element. The procedure is the mirror image of `pop_back`:

```

Obtain a pointer to the new head (the current head's right neighbor);

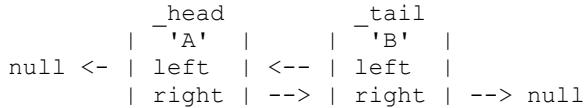
If the new head is not null,
    update its left link to point to null;
Otherwise,
    the List is about to become empty, so update _tail to point to null;

Destroy the current head;
Update _head to point to the new head;

```

```
Decrement the _size;
```

Given the List



for example, `pop_front` performs the following operations:

```
// Obtain a pointer to the new head (the current head's right neighbor)

Node* newHead = _head->right;

                                newHead
    _head      _tail
    | 'A' |     | 'B' |
null <- | left | <----- | left |
         | right| -----> | right| -> null

// The new head is not null, so update its left link to point to null

newHead->left = nullptr;

                                newHead
    _head      _tail
    | 'A' |     | 'B' |
null <- | left |     null <- | left |
         | right| -----> | right| -> null

// Destroy the current head

_destroyNode(_head);

                                newHead
    _head      _tail
    | 'B' |     |
null <- | left |
         | right| -> null

// Update _head to point to the new head

_head = newHead;

                                _head
                                _tail
                                | 'B' |
null <- | left |
         | right| -> null
  
```

To pop front the next element:

```

// Obtain a pointer to the new head (the current head's right neighbor)

Node* newHead = _head->right;

          _head      newHead
          | tail
          | 'B'   |
null <- | left   |
          | right  | -> null

// The new head is null, so the List is about to become empty
// Update _tail to point to null

_tail = nullptr;

          _head      newHead
          | tail
          | 'B'   |
null <- | left   |
          | right  | -> null

// Destroy the current head

_destoryNode(_head);

          _head      newHead
          | tail
          | null

// Update _head to point to the new head

_head = newHead;

          _head      // _head and _tail are now
          | tail      // null

          | null

```

pop\_front is defined as (memberFunctions\_2.h, lines 3-16)

```

template <class T>
void List<T>::pop_front()
{
    Node* newHead = _head->right;

    if (newHead != nullptr)
        newHead->left = nullptr;
    else
        _tail = nullptr;

    _destroyNode(_head);
    _head = newHead;
    --_size;
}

```

The program in this chapter demonstrates the `pop_front` and `pop_back` member functions. Lines 11-21 (`main.cpp`) construct a `List` `s` containing the elements {Z, Y, X, A, B, C}. The loop in lines 24-25 pops the back element from `s` until `s` is empty, generating the output

```

~ C
~ B
~ A
~ X
~ Y
~ Z

```

Lines 28-34 repopulate `s` with the same elements, {Z, Y, X, A, B, C}. The loop in lines 37-38 pops the front element from `s` until `s` is empty, generating the output

```

~ Z
~ Y
~ X
~ A
~ B
~ C

```

## 10.3: Implementing an Iterator

*Source files and folders*

*List/3*

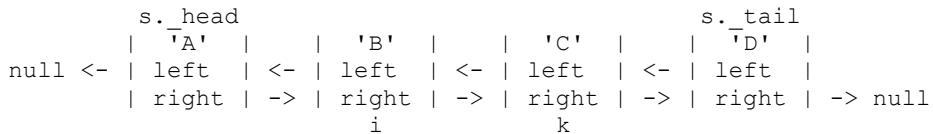
*List/common/ListIter.h*

*List/common/memberFunctions\_3.h*

*Chapter outline*

- The ListIter class
- begin / end (non-const)

This chapter introduces the ListIter class, an iterator that traverses the elements in a List. A ListIter object contains two data members, *\_list* (a pointer to the underlying List) and *\_node* (a pointer to the node containing the referent element). Consider, for example, the following List<char> s,



where *i* is an iterator (ListIter) to element B and *k* is an iterator to element C.

*i.\_list* is a pointer (List<char>\*) to s

*i.\_node* is a pointer (ListNode<char>\*) to the Node containing element B

*k.\_list* is a pointer (List<char>\*) to s

*k.\_node* is a pointer (ListNode<char>\*) to the Node containing element C

The expressions

```

i == k      // Equality (Return true if i and k point to the same element)
i != k      // Inequality (Return true if i and k point to different elements)

*i          // Dereference (Return the referent of i, element B)

++i         // Prefix increment (Point i to the next element, C,
            // then return i)

--i         // Prefix decrement (Point i to the previous element, A,
            // then return i)

i++         // Postfix increment (Point i to the next element, C,
            // but return an iterator to the original element, B)
  
```

```
i--      // Postfix decrement (Point i to the previous element, A,
       // but return an iterator to the original element, B)
```

are shorthand for

```
i.operator==(k)
i.operator!=(k)

i.operator*()

i.operator++()
i.operator--()

i.operator++(0)
i.operator--(0)
```

where the operators ==, !=, \*, ++, and -- are member functions of the ListIter class.

The ListIter class is defined in the header file ListIter.h (lines 8-42). The template parameter, List (line 8), is the type of the iterator's underlying List. In the above example, i and k are objects of type ListIter<List<char>> (“ListIter for a List<char>”). Line 12,

```
friend List;
```

declares the underlying List class as a friend of the ListIter class. The class List<char>, for example, is a friend of ListIter<List<char>>, while the class List<int> is a friend of the class ListIter<List<int>>. This friend declaration allows all member functions of List to access the private members of ListIter.

Lines 14-17 and 33 declare pointer, reference, difference\_type, value\_type, and Node as aliases of the corresponding types in the underlying List. For a ListIter<List<char>>, for example, the compiler generates the code

```
typedef List<char>::pointer pointer;
typedef List<char>::reference reference;
typedef List<char>::difference_type difference_type;
typedef List<char>::value_type value_type;
typedef List<char>::Node Node;

// ListIter<List<char>>::pointer is an alias of the type char*
// ListIter<List<char>>::reference is an alias of the type char&
// ListIter<List<char>>::difference_type is an alias of the type int
// ListIter<List<char>>::value_type is an alias of the type char
// ListIter<List<char>>::Node is an alias of the type ListNode<char>
```

Line 18,

```
typedef std::bidirectional_iterator_tag iterator_category;
```

declares iterator\_category as an alias of the type std::bidirectional\_iterator\_tag. bidirectional\_iterator\_tag is defined in the Standard Library header <iterator>, which is included in line

4. This indicates that the “iterator category” of a `ListIter` is “bidirectional iterator.” Iterator categories will be applied in a later chapter.

The data members `_list` (pointer to the underlying List) and `_node` (pointer to the Node containing the iterator's referent element) are declared in lines 40-41. The private constructor (line 35)

```
ListIter(List* list, Node* node);
```

simply initializes both data members using the given parameters (lines 110-116). The begin and end member functions of List will use this constructor to construct their return values (iterators pointing to the first / one-past-the-last elements). Because this (non-default) constructor has been defined, the compiler will not automatically generate a default constructor. The default constructor must therefore be explicitly defined even though it does nothing (lines 20, 44-48).

The comparison operators (lines 22-23) compare two ListIter objects by determining whether they point to the same Node (lines 50-60).

The dereference operator (line 25) returns a reference to the referent element (lines 68-72), while the member access operator (line 24) returns a pointer to the referent element (lines 62-66).

The private member function `_pointToNextNode` (line 37) reseats `_node` to point to the next Node in the List (lines 118-122):

```
template <class List>
inline void ListIter<List>::_pointToNextNode()
{
    _node = _node->right;
}
```

Similarly, the private member function `_pointToPreviousNode` (line 38) reseats `_node` to point to the previous Node in the List (lines 124-131):

```
template <class List>
void ListIter<List>::_pointToPreviousNode()
{
    if (_node != nullptr)          // If _node is not null,
        _node = _node->left;      //   point it to its left neighbor;
    else                          // Otherwise,
        _node = _list->_tail;    //   _node points to the one-past-the-last
}                                //   node (null), so point it to the tail
```

This allows `_node` to be reseated to the previous Node, even if it points to the end (one-past-the-last) Node. Revisiting the example at the beginning of the chapter.

```

s._head           s._tail
| 'A' |   | 'B' |   | 'C' |   | 'D' |
null <- | left | <- | left | <- | left | <- | left |
         | right| -> | right| -> | right| -> | right| -> null
                     i                 k

```

calling `_pointToPreviousNode` on `i` will execute line 128, while calling `_pointToPreviousNode` on `k` will execute line 130.

The implementation of the increment / decrement operators ensures that a `ListIter` behaves like an ordinary pointer. Recall from the end of Chapter 5.4 that given a pointer `p`, the expression

```
++p      // Prefix increment p
```

points `p` to the next element and returns `p`, while the expression

```
p++      // Postfix increment p
```

points `p` to the next element but returns a separate pointer to `p`'s original referent. The prefix increment / decrement operators (lines 27-28),

```
ListIter& operator++();
ListIter& operator--();
```

point the `ListIter` to the next / previous Node in the List and return (a reference to) the `ListIter` itself (lines 74-88):

```
template <class List>
inline ListIter<List>& ListIter<List>::operator++()
{
    _pointToNextNode();

    return *this;      // Return a reference to the object (ListIter) on which
                      // the function was called

template <class List>
inline ListIter<List>& ListIter<List>::operator--()
{
    _pointToPreviousNode();

    return *this;
}
```

Given a `ListIter` `i`, for example, the expression

```
++i      // i.operator++()
```

points `i` to the next element and returns (a reference to) `i`.

The postfix increment / decrement operators (lines 29-30),

```
ListIter operator++(int n);
ListIter operator--(int n);
```

point the `ListIter` to the next / previous Node, but return an iterator to the original Node (lines 90-108):

```

template <class List>
inline ListIter<List> ListIter<List>::operator++(int n)
{
    ListIter iterToOriginalNode(*this);      // Construct iterToOriginalNode as a
                                              // copy of *this;
    _pointToNextNode();                      // this->_pointToNextNode();

    return iterToOriginalNode;
}

template <class List>
inline ListIter<List> ListIter<List>::operator--(int n)
{
    ListIter iterToOriginalNode(*this);      // Construct iterToOriginalNode as a
                                              // copy of *this;
    _pointToPreviousNode();                  // this->_pointToPreviousNode();

    return iterToOriginalNode;
}

```

The parameter n exists solely to distinguish the postfix operator++ / operator-- from the prefix versions; it is not actually used by either function. Given a ListIter i, for example, the expression

```
i++    // i.operator++(0)
```

points i to the next element but returns a separate iterator to i's original referent.

Once the ListIter class is implemented, the begin and end member functions can be defined for the List class. Line 14 (List.h),

```
friend class ListIter<List>;
```

declares the class ListIter<List> as a friend of List. This allows all member functions of ListIter to access the private members of List. The class ListIter<List<char>>, for example, is a friend of the class List<char>, while the class ListIter<List<int>> is a friend of the class List<int>. This is necessary because the ListIter member function \_pointToPreviousNode requires access to the private List member \_tail (ListIter.h, line 130).

Line 18,

```
typedef ListIter<List> iterator;
```

declares iterator as an alias of the type ListIter<List>. This makes List<T>::iterator an alias of the type ListIter<List<T>>. List<int>::iterator, for example, is an alias of the type ListIter<List<int>>.

The member functions begin and end (lines 33-34) can then be defined as (memberFunctions\_3.h, lines 3-13)

```

template <class T>
inline typename List<T>::iterator List<T>::begin()
{
    return iterator(this, _head);           // return ListIter<List>(this, _head);
}

template <class T>
inline typename List<T>::iterator List<T>::end()
{
    return iterator(this, nullptr);        // return ListIter<List>(this, nullptr);
}

```

begin returns a ListIter pointing to the head Node (line 6), while end returns a ListIter pointing to the end (one-past-the-last) Node (line 12). Both functions construct their return values using ListIter's private constructor (ListIter.h, line 35). Recall that they can access this constructor because List is a friend of ListIter (ListIter.h, line 12).

The program in this chapter demonstrates the begin / end member functions as well as the ListIter class. Lines 13-21 (main.cpp) construct a List<int> s containing the elements {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5}. Line 23,

```
printSequence(s.begin(), s.end());
```

is equivalent to

```
printSequence<List<int>::iterator>(s.begin(), s.end());
```

or

```
printSequence<ListIter<List<int>>>(s.begin(), s.end());
```

The compiler generates the code

```

void printSequence(ListIter<List<int>> begin, ListIter<List<int>> end)
{
    while (begin != end)           // while (begin.operator!=(end))
    {
        std::cout << *begin << ' ';   // std::cout << begin.operator*() << ' ';
        ++begin;                   // begin.operator++();
    }
}

```

printSequence generates the output

```
-5 -4 -3 -2 -1 0 1 2 3 4 5
```

Lines 26-31 then traverse the List in reverse order, generating the output

```
5 4 3 2 1 0 -1 -2 -3 -4 -5
```

Lines 34-35 construct a List<Traceable<string>> t, containing a single element, {macadamia}. Line 38,

```
List<Traceable<string>>::iterator x = t.begin();
```

initializes an iterator x, pointing to that element. Line 39,

```
x->printValue(); // (*x).printValue();
// Dereference x, then call the printValue member
// function on the referent element (a Traceable<string>)
```

is shorthand for

```
x.operator->()>printValue();
```

The expression

```
x.operator->() // Call the operator-> member function on x (an object  
// of type ListIter<List<Traceable<string>>>)
```

returns a pointer to the iterator's referent element, an object of type `Traceable<string>`. The

->printValue()

then calls the `printValue` member function through this pointer, generating the output

## macadamia

Note that `List<T>::iterator` is also compatible with the `insertionSort` (Chapter 7.2) and `quickSort` (Chapter 7.3) functions. Given

```
List<int> x;
List<Student> y;

// Insert some elements into x and y...
```

for example, the statements

are equivalent to

```
insertionSort<List<int>::iterator, Less<int>>(x.begin(),  
    x.end(),  
    Less<int>());
```

```
quickSort<List<Student>::iterator, Greater<Student>>(y.begin(),  
    y.end(),  
    Greater<Student>());
```

## 10.4: Implementing a Const Iterator

*Source files and folders*

*ConstIter/ConstIter.h*  
*ConstIter/memberFunctions\_1.h*  
*List/4*  
*List/common/memberFunctions\_4.h*

*Chapter outline*

- *The ConstIter class*
- *Using iterator\_traits to obtain type information about a given iterator class*
- *begin / end (const)*

This chapter introduces the ConstIter class template, which will be used to implement not only *List<T>::const\_iterator*, but also the *const\_iterators* for all the remaining data structures in this book.

The ConstIter class is defined in the header file ConstIter.h (lines 16-57). The template parameter Container (line 16) is the type of the ConstIter's underlying container (data structure). A ConstIter<List<int>>, for example, is a const iterator for a List<int>.

Line 20,

```
friend Container;
```

declares the underlying container class as a friend of ConstIter. The class List<int>, for example, is a friend of the class ConstIter<List<int>>, which allows all member functions of List<int> to access the private members of ConstIter<List<int>>.

Line 21,

```
typedef typename Container::iterator Iter;
```

declares Iter as an alias of the underlying container's iterator type (Container::iterator). Similarly, lines 32-33,

```
typedef typename Container::const_pointer pointer;
typedef typename Container::const_reference reference;
```

declare pointer and reference as aliases of the underlying container's const\_pointer and const\_reference types. For a ConstIter<List<int>>, for example, the compiler generates the code

```

class ConstIter<List<int>>                                // Const iterator for a
{                                                               // List<int>
public:
    friend class List<int>;
    typedef List<int>::iterator Iter;

    // ...

    typedef List<int>::const_pointer pointer;                // const int*
    typedef List<int>::const_reference reference;           // const int&

    // ...

private:
    Iter _i;                                                 // List<int>::iterator _i;
};

```

The Standard Library provides the class template iterator\_traits, which is used to obtain type information about a given iterator class. iterator\_traits is defined in the header <iterator> as

```

namespace std
{
template <class Iterator>
struct iterator_traits
{
    typedef typename Iterator::pointer pointer;
    typedef typename Iterator::reference reference;
    typedef typename Iterator::difference_type difference_type;
    typedef typename Iterator::value_type value_type;
    typedef typename Iterator::iterator_category iterator_category;
};
};

```

Given an iterator class I, the expressions

```

std::iterator_traits<I>::pointer
std::iterator_traits<I>::reference
std::iterator_traits<I>::difference_type
std::iterator_traits<I>::value_type
std::iterator_traits<I>::iterator_category

```

are equivalent to

```

I::pointer
I::reference
I::difference_type
I::value_type
I::iterator_category

```

The expressions

```
std::iterator_traits<List<int>::iterator>::pointer
```

```
std::iterator_traits<List<double>::iterator>::reference  
std::iterator_traits<List<char>::iterator>::difference_type  
std::iterator_traits<List<bool>::iterator>::value_type  
std::iterator_traits<List<string>::iterator>::iterator_category
```

for example, are therefore equivalent to

```
List<int>::iterator::pointer           // int*
List<double>::iterator::reference      // double&
List<char>::iterator::difference_type   // int
List<bool>::iterator::value_type        // bool
List<string>::iterator::iterator_category // std::bidirectional_iterator_tag
```

Lines 26-30 (ConstIter.h),

```
typedef typename std::iterator_traits<Iter>::value_type value_type;
typedef typename std::iterator_traits<Iter>::difference_type difference_type;

typedef typename std::iterator_traits<Iter>::iterator_category
iterator_category;
```

use `iterator_traits` to obtain `Iter`'s `value_type`, `difference_type`, and `iterator_category`. For a `ConstIter<List<int>>`, for example (where `Iter` is an alias of the type `List<int>::iterator`), the compiler generates the code

```
typedef std::iterator_traits<List<int>::iterator>::value_type value_type;
typedef std::iterator_traits<List<int>::iterator>::difference_type difference_type;
typedef std::iterator_traits<List<int>::iterator>::iterator_category iterator_category;
```

The class definition of ConstIter<List<int>> thus becomes

ConstIter's default constructor (line 23) implicitly initializes `_i` (the contained iterator, line 56) via its own default constructor (memberFunctions\_1.h, lines 3-7).

The constructor (ConstIter.h, line 24)

```
ConstIter(const Iter& i);
```

constructs a ConstIter from the given Iter `i`. This also allows for the implicit conversion of an Iter to a ConstIter (the same way that a pointer can be implicitly converted to a const pointer) (memberFunctions\_1.h, lines 9-14):

```
template <class Container>
inline ConstIter<Container>::ConstIter(const Iter& i):
    _i(i)
{
    // ...
}
```

The member functions `operator==` and `operator!=` (ConstIter.h, lines 35-36) compare two ConstIter by comparing their contained Iters (memberFunctions\_1.h, lines 16-26).

The member functions (ConstIter.h, lines 41-42)

```
pointer operator->() const;    // Container::const_pointer operator->() const;
reference operator*() const;   // Container::const_reference operator*() const;
```

return a const pointer / const reference to the referent element by calling the respective member function on `_i` (memberFunctions\_1.h, lines 28-40). Note that the expressions (lines 32, 39)

```
_i.operator->()
*_i           // _i.operator*()
```

return a (non-const) pointer / reference because `_i` is an object of type Container::iterator (non-const iterator); this pointer / reference is then implicitly converted to a const pointer / const reference.

The prefix increment / decrement operators (ConstIter.h, lines 48-49) point the ConstIter to the next / previous element by prefix incrementing / decrementing `_i`. Both functions return a reference to the ConstIter on which the function was called (memberFunctions\_1.h, lines 42-56).

The postfix increment / decrement operators (ConstIter.h, lines 50-51) point the ConstIter to the next / previous element, but return a separate ConstIter pointing to the original element. This is achieved by applying postfix increment / decrement to `_i` and returning the result (memberFunctions\_1.h, lines 58-68). The expressions (lines 61, 67)

```
_i++
_i--
```

return an Iter (Container::iterator) because `_i` is an object of type Iter. This Iter is then implicitly

converted to a ConstIter via ConstIter's constructor (ConstIter.h, line 24).

The remaining member functions (ConstIter.h, lines 37-40, 43-46, 52-53) and nonmember operator+ function (lines 11-14) are not required to implement List<T>::const\_iterator. These functions, which are defined in the header files memberFunctions\_2.h and nonMemberFunctions.h (included in ConstIter.h, lines 61-62), will be discussed in a later chapter.

Once the ConstIter class is implemented, the const versions of begin and end can be defined for the List class. Line 17 (List.h),

```
typedef ConstIter<List> const_iterator;
```

declares List<T>::const\_iterator as an alias of the type ConstIter<List<T>>. List<int>::const\_iterator, for example, is an alias of the type ConstIter<List<int>>. The const versions of begin / end (lines 32-33) are then defined as (memberFunctions\_4.h, lines 3-13)

```
template <class T>
inline typename List<T>::const_iterator List<T>::begin() const
{
    return const_cast<List*>(this)->begin();
}

template <class T>
inline typename List<T>::const_iterator List<T>::end() const
{
    return const_cast<List*>(this)->end();
}
```

Within the body of a const member function, the “this” pointer is a const pointer to the object on which the function was called. In lines 6 and 12, the “this” pointer is a const pointer to the List on which the function was called. The expressions

```
begin()      // this->begin()
end()        // this->end()
```

will therefore call the const versions of begin / end by default. To call the non-const versions of begin / end, a non-const “this” pointer must be explicitly obtained. The expression

```
const_cast<List*>(this)      // Explicitly obtain a List* (non-const pointer)
                           // from "this" (a const pointer)
```

returns a non-const “this” pointer (a List\*). The expressions

```
const_cast<List*>(this)->begin()
const_cast<List*>(this)->end()
```

will therefore call the non-const versions of begin / end, which return List<T>::iterators. These iterators can then be used to construct the required List<T>::const\_iterators. The statements

```
return const_cast<List*>(this)->begin();
return const_cast<List*>(this)->end();
```

construct the return values via ConstIter's constructor (ConstIter.h, line 24), and are equivalent to

```
return const_iterator(const_cast<List*>(this)->begin());
return const_iterator(const_cast<List*>(this)->end());

// Return a const_iterator constructed from the iterator returned by the
// non-const version of begin / end
```

The printContainer function template, which calls the const versions of a given container's begin / end functions, can now be used with the List class. Lines 11-14 (main.cpp) construct a List<int> s containing the elements {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. In line 16,

```
printContainer(s);      // printContainer<List<int>>(s);
```

the compiler generates the code

```
inline void printContainer(const List<int>& container)
{
    printSequence(container.begin(), container.end());           // Calls the const
    std::cout << std::endl;                                         // versions of begin
}                                                               // and end

void printSequence(List<int>::const_iterator begin,
                  List<int>::const_iterator end)
{
    while (begin != end)
    {
        std::cout << *begin << ' ';
        ++begin;
    }
}
```

The call to printSequence generates the output

```
0 1 2 3 4 5 6 7 8 9
```

As mentioned at the beginning of this chapter, the ConstIter class will be reused to implement the const\_iterators for all the remaining data structures in this book.

## 10.5: Copy and Assignment

*Source files and folders*

*List/5*  
*List/common/memberFunctions\_5.h*

*Chapter outline*

- *Copy constructor and assignment operator*

The copy constructor (*List.h*, line 28) constructs a *List* as a copy of an existing *List*. Its implementation is nearly identical to that of *Vector* (Chapter 9.2) (*memberFunctions\_5.h*, lines 3-11). Lines 5-7 initialize the *List* as an empty container, then the loop in lines 9-10 inserts a copy of each element from the source *List*.

The implementation of the assignment operator (*List.h*, line 42) is also nearly identical to that of *Vector* (Chapter 9.2). Because *List* does not support subscripting (*operator[]*), however, it must use an iterator to set the new value of each element. The pseudocode is

```
template <class T>
List<T>& List<T>::operator=(const List& rhs)
{
    if (size is greater than or equal to rhs.size)
    {
        pop_back until size is equal to rhs.size;

        for each element e in this List
            set e to the value of its corresponding element c in rhs;
    }
    else
    {
        for each element e in this List
            set e to the value of its corresponding element c in rhs;

        for each remaining element c in rhs
            push_back(c);
    }

    return a reference to this List;
}
```

The function is defined in lines 13-40 (*memberFunctions\_5.h*). Recall from Chapters 5.4 and 10.3 that given an iterator *i*, the expression

*\*i++*

points *i* to the next element, but returns *i*'s original referent element. The loop (lines 24-25)

```

while (e != end())
    *e++ = *c++;
                // Set e's referent to the value of c's referent,
                // then point e and c to the next elements in their
                // respective sequences

```

is therefore equivalent to

```

while (e != end())
{
    *e = *c;

    ++e;
    ++c;
}

```

Similarly, the loop (lines 35-36)

```

while (c != rhs.end())
    push_back(*c++);
                // Push back the referent of c, then point c to
                // the next element in the sequence

```

is equivalent to

```

while (c != rhs.end())
{
    push_back(*c);
    ++c;
}

```

Given the Lists

a   b   c   d   e   f	// List<char> x
P   Q   R	// List<char> y

for example, the statement

```
x = y;                                // x.operator=(y);
```

performs the following operations:

```

while (size() != rhs.size())
    pop_back();

iterator e = begin();
const_iterator c = rhs.begin();

| a | b | c |
   e

| P | Q | R |
   c

```

Iteration 1:  
`*e++ = *c++;`

| P | b | c |  
   e

| P | Q | R |  
   c

Iteration 2:  
`*e++ = *c++;`

| P | Q | c |  
   e

| P | Q | R |  
   c

Iteration 3:  
`*e++ = *c++;`

| P | Q | R |  
   e

| P | Q | R |  
   c

Given the Lists

a   b   c	// List<char> x
P   Q   R   S   T   U	// List<char> y

the statement

`x = y;` // `x.operator=(y);`

performs the following operations:

`iterator e = begin();`  
`const_iterator c = rhs.begin();`

| a | b | c |  
   e

| P | Q | R | S | T | U |  
   c

Iteration 1:  
`*e++ = *c++;`

| P | b | c |  
   e

```
| P | Q | R | S | T | U |
  c
```

Iteration 2:  
`*e++ = *c++;`

```
| P | Q | c |
  e
```

```
| P | Q | R | S | T | U |
  c
```

Iteration 3:  
`*e++ = *c++;`

```
| P | Q | R |
  e
```

```
| P | Q | R | S | T | U |
  c
```

Iteration 1:  
`push_back(*c++);`

```
| P | Q | R | S |
```

```
| P | Q | R | S | T | U |
  c
```

Iteration 2:  
`push_back(*c++);`

```
| P | Q | R | S | T |
```

```
| P | Q | R | S | T | U |
  c
```

Iteration 3:  
`push_back(*c++);`

```
| P | Q | R | S | T | U |
```

```
| P | Q | R | S | T | U |
  c
```

The program in this chapter demonstrates the above examples. The function `testAssignmentOperatorCase1` (`main.cpp`, lines 9, 25-52) demonstrates the first, while the function `testAssignmentOperatorCase2` (lines 10, 54-81) demonstrates the second.

The call to `testAssignmentOperatorCase1` (line 15) generates the output

```
~ f          // pop_back();
~ e          // pop_back();
```

```

~ d          // pop_back();
a -> P      // *e++ = *c++;
b -> Q      // *e++ = *c++;
c -> R      // *e++ = *c++;

P Q R       // printContainer(x);

~ P          // Destroy y
~ Q
~ R
~ P          // Destroy x
~ Q
~ R

```

testAssignmentOperatorCase2 (line 16) generates the output

```

a -> P      // *e++ = *c++;
b -> Q      // *e++ = *c++;
c -> R      // *e++ = *c++;
c S         // push_back(*c++);
c S
~ S
c T         // push_back(*c++);
c T
~ T
c U         // push_back(*c++);
c U
~ U

P Q R S T U // printContainer(x);

~ P          // Destroy y
~ Q
~ R
~ S
~ T
~ U
~ P          // Destroy x
~ Q
~ R
~ S
~ T
~ U

```



## 10.6: Insert and Erase

*Source files and folders*

*List/6*  
*List/common/memberFunctions\_6.h*

*Chapter outline*

- insert and erase

The member function (List.h, line 43)

```
iterator insert(const_iterator insertionPoint, const T& source);
```

inserts a copy of the given source object before insertionPoint, then returns an iterator to the newly inserted element.

Recall from Chapter 9.3 that the efficiency of Vector's insert method is directly related to the relative position of the insertionPoint: the closer it is to the front of the Vector, the greater the number of swaps required. In a List, however, a new element can be inserted anywhere by simply creating a new node and updating the links (left / right pointers) of at most 2 other nodes.

The insert member function is defined in lines 3-36 (memberFunctions\_6.h). The algorithm consists of 3 branches (cases).

Case 1: Inserting an element at the front of the List (insertionPoint == begin()) (lines 7-12)

Given a List<char> s

```

    _head      _tail
    | 'A' |      | 'B' |
null <- |left| <- |left|
          |right| -> |right| -> null

```

the function call

```
s.insert(s.begin(), 'x');
```

inserts the element 'x' before element 'A', then returns an iterator to element 'x'. This is equivalent to inserting 'x' via push\_front, then returning an iterator to the head:

```
push_front('x');
```

```

    _head           _tail
    |'x' |     | 'A' |     |'B' |
null <- |left| <- |left| <- |left|
        |right| -> |right| -> |right| -> null

return iterator(this, _head);

```

Case 2: Inserting an element at the end of the List (insertionPoint == end()) (lines 13-18)

```

    _head           _tail
    |'x' |     | 'A' |     |'B' |
null <- |left| <- |left| <- |left|
        |right| -> |right| -> |right| -> null

```

The function call

```
s.insert(s.end(), 'z');
```

inserts the element 'z' before the one-past-the-last element (s.end()), then returns an iterator to element 'z'. This is equivalent to inserting 'z' via push\_back, then returning an iterator to the tail:

```

push_back('z');

    _head           _tail
    |'x' |     | 'A' |     | 'B' |     |'z' |
null <- |left| <- |left| <- |left| <- |left|
        |right| -> |right| -> |right| -> |right| -> null

return iterator(this, _tail);

```

Case 3: Inserting an element between 2 existing elements (insertionPoint is neither begin() nor end()) (lines 19-35)

```

    _head           _tail
    |'x' |     | 'A' |     | 'B' |     |'z' |
null <- |left| <- |left| <- |left| <- |left|
        |right| -> |right| -> |right| -> |right| -> null
                           i

// i is a List<char>::iterator pointing to element 'B'

```

The function call

```
s.insert(i, 'y');
```

inserts the element 'y' before element 'B', then returns an iterator to element 'y'. The function performs the following operations:

```

// Create a new node containing the given element

Node* newNode = _createNode('y');

```

```

newNode
| 'y' |
null <- |left |
|right| -> null

// Let leftNeighbor and rightNeighbor be pointers to the new left / right
// neighbors of the new node

Node* leftNeighbor = insertionPoint._i._node->left;
Node* rightNeighbor = insertionPoint._i._node;

            _head                                _tail
| 'x' |     | 'A' | <----- | 'B' |     | 'z' |
null <- |left | <- |left | <----- |left | <- |left |
|right| -> |right| -----> |right| -> |right| -> null
                leftNeighbor                  rightNeighbor
                                         insertionPoint

// Recall that:
//   insertionPoint is a const_iterator
//   insertionPoint._i is the iterator contained in insertionPoint
//   insertionPoint._i._node is the Node* contained in insertionPoint._i

// These private members are accessible because List is a friend of ConstIter
// and ListIter

// Update leftNeighbor's right link and rightNeighbor's left link to point to
// the new node

leftNeighbor->right = newNode;
rightNeighbor->left = newNode;

            _head                                newNode
| 'x' |     | 'A' |           | 'y' |           | 'B' |     _tail
null <- |left | <- |left | null <- |left | <----- |left | <- |left |
|right| -> |right| -----> |right| null -> |right| -> |right| -> null
                leftNeighbor                  rightNeighbor

// Update newNode's left link to point to leftNeighbor, and its right link
// to point to rightNeighbor

newNode->left = leftNeighbor;
newNode->right = rightNeighbor;

            _head                                newNode
| 'x' |     | 'A' |           | 'y' |           | 'B' |     _tail
null <- |left | <- |left | <----- |left | <----- |left | <- |left |
|right| -> |right| -----> |right| -----> |right| -> |right| -> null
                leftNeighbor                  rightNeighbor

// Increment the size and return an iterator to the newly inserted element,
// 'y'

++_size;

```

```
return iterator(this, newNode);
```

The member function (List.h, line 44)

```
iterator erase(const_iterator erasurePoint);
```

removes the element at erasurePoint and returns an iterator to the next element in the List. Like insert, the function is divided into 3 branches (memberFunctions\_6.h, lines 38-67):

Case 1: Erasing the head (erasurePoint == begin()) (lines 41-46)

```
head           tail
|-'x' |     | 'A' |     | 'y' |     | 'B' |     |-'z' |
null <- |left| <- |left| <- |left| <- |left| <- |left|
         |right| -> |right| -> |right| -> |right| -> |right| -> null
erasurePoint
```

The function call

```
s.erase(s.begin());
```

removes the element 'x' then returns an iterator to the next element, 'A'. This is equivalent to removing 'x' via pop\_front, then returning an iterator to the head:

```
pop_front();
```

```
head           tail
|-'A' |     | 'y' |     | 'B' |     |-'z' |
null <- |left| <- |left| <- |left| <- |left|
         |right| -> |right| -> |right| -> |right| -> null
```

```
return iterator(this, _head);
```

Case 2: Erasing the tail (erasurePoint.\_i.\_node == \_tail) (lines 47-52)

```
head           tail
|-'A' |     | 'y' |     | 'B' |     |-'z' |
null <- |left| <- |left| <- |left| <- |left|
         |right| -> |right| -> |right| -> |right| -> null
                           i
```

```
// i is a List<char>::iterator pointing to element 'z'
```

The function call

```
s.erase(i);
```

removes the element 'z' via pop\_back and returns an iterator to the the next element, the end:

```
pop_back();
```

```

    _head           _tail
    | 'A' |     | 'y' |     | 'B' |
null <- |left| <- |left| <- |left|
          |right| -> |right| -> |right| -> null

return end();

```

Case 3: Erasing an element between 2 existing elements (erasurePoint is neither the head nor tail) (lines 53-66)

```

    _head           _tail
    | 'A' |     | 'y' |     | 'B' |
null <- |left| <- |left| <- |left|
          |right| -> |right| -> |right| -> null
                           i

// i is a List<char>::iterator pointing to element 'y'

```

The function call

```
s.erase(i);
```

removes the element 'y' and returns an iterator to the next element, 'B':

```

// erasurePoint points to element 'y'
// Let trash be a pointer to the node at erasurePoint (the node to be erased)
// Let leftNeighbor and rightNeighbor be pointers to the left / right
// neighbors of trash

Node* trash = erasurePoint._i._node;
Node* leftNeighbor = trash->left;
Node* rightNeighbor = trash->right;

    _head           _tail
    | 'A' |     | 'y' |     | 'B' |
null <- |left| <- |left| <- |left|
          |right| -> |right| -> |right| -> null
                           L       T       R

// T denotes trash
// L and R denote leftNeighbor / rightNeighbor

_destroyNode(trash);
--_size;

    _head           _tail
    | 'A' |           | 'B' |
null <- |left|           <- |left|
          |right| ->           |right| -> null
                           L           R

// Reconnect leftNeighbor and rightNeighbor by updating their respective
// links

```

```

leftNeighbor->right = rightNeighbor;
rightNeighbor->left = leftNeighbor;

           _head          _tail
           | 'A' |          | 'B' |
null <- |left| <----- |left|
           |right| -----> |right| -> null
           L             R

// Return an iterator to the next element, 'B'

return iterator(this, rightNeighbor);

```

The program in this chapter demonstrates the above examples. Lines 12-17 (main.cpp) construct a List<Traceable<char>> s containing the elements {A, B}. Lines 20 and 23,

```

s.insert(s.begin(), 'x');
s.insert(s.end(), 'z');

```

insert 'x' and 'z' at the beginning and end of the List, generating the output

```

v x    // Construct argument passed to insert
c x    // Construct argument passed to _alloc.construct
c x    // Construct copy of argument (new element in List)
~ x    // Destroy argument passed to _alloc.construct
~ x    // Destroy argument passed to insert

v z    // Similar output when inserting 'z'...
c z
c z
~ z
~ z    // s now contains the elements {x, A, B, z}

```

Line 26,

```
List::iterator i = +++++s.begin();      // i points to element B
```

is equivalent to

```

List::iterator i = s.begin();
++i;
++i;

```

Line 28,

```
i = s.insert(i, 'y');
```

inserts 'y' before element 'B', after which i points to the newly inserted element 'y'. The insertion generates the output

```
v y    // Construct argument passed to insert
```

```
c y    // Construct argument passed to _alloc.construct
c y    // Construct copy of argument (new element in List)
~ y    // Destroy argument passed to _alloc.construct
~ y    // Destroy argument passed to insert
```

Line 31 then prints the contents of s, generating the output

```
x A y B z
```

Lines 34-36,

```
cout << "Inserted element " << *i << endl;
cout << "Preceding element is " << *--i << endl;
cout << "Proceeding element is " << *++i << endl << endl;
```

which are equivalent to

```
cout << "Inserted element " << *i << endl;
--i;
cout << "Preceding element is " << *i << endl;
++i;
++i;
cout << "Proceeding element is " << *i << endl << endl;
```

generate the output

```
Inserted element y
Preceding element is A
Proceeding element is B
```

Lines 38 and 41 erase the head and tail elements, 'x' and 'z', generating the output

```
~x
~z    // s now contains the elements {A, y, B}
```

Line 44 decrements i, pointing it to element 'y'. Line 47,

```
i = s.erase(i);
```

erases element 'y', after which i points to the next element, 'B'. Lines 50-54 then print the contents of s as well as the proceeding / preceding elements, generating the output

```
A B
```

```
Proceeding element is B
Preceding element is A
```



# Part 11: Reverse Iterators

## 11.1: Introducing the ReverseIter Class Template

*Source files and folders*

```
List/7
List/common/memberFunctions_7.h
printContainer
ReverseIter/ReverseIter.h
ReverseIter/memberFunctions_1.h
```

*Chapter outline*

- The ReverseIter class
- Implementing rbegin and rend for List
- The printContainerReverse function template

A “reverse iterator” is an iterator that traverses a sequence in reverse order. Consider, for example, a `List<int>` containing the elements {0, 1, 2, 3, 4}:

```
begin          end
0             1             2             3             4
rend          rbegin        p
// begin / end are the first / one-past-the-last elements in the List
// p is a List<int>::reverse_iterator pointing to element 4
```

The reverse sequence is {4, 3, 2, 1, 0}, where `rbegin` (“reverse begin”) is the first element and `rend` (“reverse end”) is the one-past-the-last element. Incrementing `p` points it to the next element of the reversed sequence:

```
++p;
begin          end
0             1             2             3             4
rend          rbegin        p
```

Similarly, decrementing `p` points it to the previous element of the reversed sequence:

```
--p;

begin          end
0           1           2           3           4
rend          rbegin      p
```

Although p points to element 4, it does not simply contain a pointer to the Node containing element 4; rather, it actually contains a `List<int>::iterator _i` pointing to the end:

```
begin          end
0           1           2           3           4
rend          rbegin      p           p._i

// p is a List<int>::reverse iterator pointing to element 4
// p._i is a List<int>::iterator pointing to end
```

The expression

```
*p // Element 4
```

returns the element to the left of `p._i`. The expression

```
++p // p.operator++()
```

points p to element 3 by decrementing `p._i`:

```
begin          end
0           1           2           3           4
rend          rbegin      p           p._i

// p is a List<int>::reverse_iterator pointing to element 3
// p._i is a List<int>::iterator pointing to element 4
// *p returns the element to the left of p._i (element 3)
```

Similarly, the expression

```
--p // p.operator--()
```

points p back to element 4 by incrementing `p._i`:

```
begin          end
0           1           2           3           4
rend          rbegin      p           p._i

// p is a List<int>::reverse_iterator pointing to element 4
// p._i is a List<int>::iterator pointing to end
// *p returns the element to the left of p._i (element 4)
```

This chapter introduces the ReverseIter class template, which will be used to implement not only `List<T>::reverse_iterator` and `List<T>::const_reverse_iterator`, but also the reverse iterators for all of the data structures in this book.

The ReverseIter class is defined in the header file `ReverseIter.h` (lines 15-55). The template parameter Iter (line 15) is the type of the contained iterator `_i` (line 54). Lines 19-24 declare difference\_type, pointer, reference, value\_type, and iterator\_category as aliases of Iter's corresponding types. For a `ReverseIter<List<int>::iterator>`, for example, the compiler generates the code

```
class ReverseIter<List<int>::iterator>      // A reversed List<int>::iterator
{
public:
    typedef List<int>::iterator::difference_type difference_type;
    typedef List<int>::iterator::pointer pointer;
    typedef List<int>::iterator::reference reference;
    typedef List<int>::iterator::value_type value_type;
    typedef List<int>::iterator::iterator_category iterator_category;

    // ...

    explicit ReverseIter<List<int>::iterator>(const List<int>::iterator& i);

    // ...

    List<int>::iterator base() const;

    // ...

private:
    List<int>::iterator _i;
};
```

The default constructor (line 26) implicitly initializes `_i` via its own default constructor (`memberFunctions_1.h`, lines 3-7). The constructor (line 27)

```
explicit ReverseIter(const Iter& i);      // Construct a ReverseIter<Iter>
                                            // from an Iter i
```

constructs a ReverseIter by initializing `_i` as a copy of the given Iter `i` (`memberFunctions_1.h`, lines 9-14). The “`explicit`” keyword indicates that this constructor may only be called explicitly; it disallows the implicit conversion of an Iter to a `ReverseIter<Iter>`. This will be discussed more later in the chapter.

The member function (`ReverseIter.h`, line 32)

```
Iter base() const;
```

returns a copy of `_i`, the “base” iterator (`memberFunctions_1.h`, lines 24-28).

### The constructor (ReverseIter.h, lines 29-30)

```
template <class OtherIter>
ReverseIter(const ReverseIter<OtherIter>& source);
```

constructs a `ReverseIter<Iter>` from a `ReverseIter<OtherIter>` by constructing `_i` from `source._i` (`memberFunctions_1.h`, lines 16-22):

```
template <class Iter>
template <class OtherIter>
inline ReverseIter<Iter>::ReverseIter(const ReverseIter<OtherIter>& source) :
    _i(source.base())
{
    // ...
}

// source is an object of type ReverseIter<OtherIter>
// source.base() returns source._i, an object of type OtherIter
// This OtherIter is then used to construct _i, an object of type Iter
```

This constructor will also be discussed more later in the chapter.

The comparison operators (`ReverseIter.h`, lines 33-34) compare two `ReverseIter`s by simply comparing their contained `Iter`s (`memberFunctions_1.h`, lines 30-40).

As mentioned earlier, the dereference operator (`ReverseIter.h`, line 40) returns (a reference to) the element to the left of `_i` (`memberFunctions_1.h`, lines 52-60):

```
template <class Iter>
inline typename ReverseIter<Iter>::reference
ReverseIter<Iter>::operator*() const
{
    Iter temp = _i;
    --temp;           // temp points to the element to the left of _i
    return *temp;     // Return (a reference to) that element
}
```

Similarly, the member access operator (`ReverseIter.h`, line 39) returns a pointer to (the address of) the element to the left of `_i` (`memberFunctions_1.h`, lines 42-50).

The increment / decrement operators (`ReverseIter.h`, lines 46-49) simply apply the reverse operation to `_i` (`memberFunctions_1.h`, lines 62-88).

The remaining member functions (`ReverseIter.h`, lines 35-38, 41-44, 50-51) and nonmember function operator`+` (lines 11-13) are not required to implement `List<T>::reverse_iterator` and `List<T>::const_reverse_iterator`. These functions, which are defined in the header files `ReverseIter/memberFunctions_2.h` and `ReverseIter/nonMemberFunctions.h`, will be discussed in the next chapter.

Once the ReverseIter class is implemented, List<T>::reverse\_iterator and List<T>::const\_reverse\_iterator can be defined. Line 23 (List.h),

```
typedef ReverseIter<iterator> reverse_iterator;
```

declares reverse\_iterator as an alias of the type ReverseIter<iterator> (a reversed List<T>::iterator). Similarly, line 19,

```
typedef ReverseIter<const_iterator> const_reverse_iterator;
```

declares const\_reverse\_iterator as an alias of the type ReverseIter<const\_iterator> (a reversed List<T>::const\_iterator).

The member functions (lines 38-39, 45-46)

```
const_reverse_iterator rbegin() const;
const_reverse_iterator rend() const;

reverse_iterator rbegin();
reverse_iterator rend();
```

return reverse iterators to the first and one-past-the-last elements of the reverse sequence. rbegin returns a ReverseIter in which \_i points to the end, while rend returns a ReverseIter in which \_i points to begin (memberFunctions\_7.h, lines 3-25):

```
template <class T>
inline typename List<T>::const_reverse_iterator List<T>::rbegin() const
{
    return const_reverse_iterator(end());
}

template <class T>
inline typename List<T>::const_reverse_iterator List<T>::rend() const
{
    return const_reverse_iterator(begin());
}

template <class T>
inline typename List<T>::reverse_iterator List<T>::rbegin()
{
    return reverse_iterator(end());
}

template <class T>
inline typename List<T>::reverse_iterator List<T>::rend()
{
    return reverse_iterator(begin());
}
```

A List can then be traversed backward by incrementing a reverse iterator from rbegin to rend. Lines 11-14 (main.cpp) construct a List<int> s containing the elements {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Line 16,

```
printContainerReverse(s);
```

is equivalent to

```
printContainerReverse<List<int>>(s);
```

The `printContainerReverse` function template is defined as (`printContainer.h`, lines 23-28)

```
template <class Container>
inline void printContainerReverse(const Container& container)
{
    printSequence(container.rbegin(), container.rend());
    std::cout << std::endl;
}
```

The compiler generates the code

```
inline void printContainerReverse(const List<int>& container)
{
    printSequence(container.rbegin(), container.rend());
    std::cout << std::endl;
}
```

`container` is a `const` reference, so the expressions

```
container.rbegin()
container.rend()
```

call the `const` versions of `rbegin` and `rend` (which return `const_reverse_iterator`s). The compiler therefore generates the `printSequence` function

```
void printSequence(List<int>::const_reverse_iterator begin,
                  List<int>::const_reverse_iterator end)
{
    while (begin != end)
    {
        std::cout << *begin << ' ';
        ++begin;
    }
}
```

The call to `printContainerReverse` generates the output

```
9 8 7 6 5 4 3 2 1 0
```

Recall the “`explicit`” keyword in the constructor (`ReverseIter.h`, line 27)

```
explicit ReverseIter(const Iter& i);
```

Without the “`explicit`” keyword, a `List<T>::reverse_iterator` could be automatically constructed from a `List<T>::iterator`, even when unintended:

```

// Subtle bug (Accidentally using s.begin() instead of s.rbegin())
for (List<int>::reverse_iterator i = s.begin(); i != s.rend(); ++i)
{
    // ...
}

// List<int>::reverse_iterator i is implicitly constructed from the
// List<int>::iterator returned by s.begin()

// i is therefore initialized to point to the one-past-the-last element of
// the reversed sequence (rend)

// The loop condition (i != s.rend()) will thus always be false, so the
// loop body will never be executed

```

With the “explicit” keyword, however, the above code results in a compiler error because the “explicit” keyword disallows the implicit (automatic) construction of a List<int>::reverse\_iterator (i) from a List<int>::iterator (s.begin()). For the above code to compile, the constructor would have to be called explicitly:

```

// Explicitly construct a List<int>::reverse_iterator (i) from the
// List<int>::iterator s.begin() (ok) (explicitly call ReverseIter's
// constructor)

List<int>::reverse_iterator i = List<int>::reverse_iterator(s.begin());

```

#### The template constructor (ReverseIter.h, lines 29-30)

```

// Construct a ReverseIter<Iter> from a ReverseIter<OtherIter>

template <class OtherIter>
ReverseIter(const ReverseIter<OtherIter>& source);

```

is what allows a const\_reverse\_iterator to be constructed from a reverse\_iterator, as in

```

List<int> s;

// ...

List<int>::reverse_iterator p = s.rbegin();

// Construct q (a List<int>::const_reverse_iterator) from p (a
// List<int>::iterator)

List<int>::const_reverse_iterator q = p;

```

p is a ReverseIter<List<int>::iterator>, but q is a ReverseIter<List<int>::const\_iterator>. The template constructor constructs q by constructing q.\_i (a List<int>::const\_iterator) from p.base() (a List<int>::iterator) (memberFunctions\_1.h, line 19).



## 11.2: Reverse Iterators for the Array and Vector Classes

*Source files and folders*

```
Array/2
Array/common/memberFunctions_2.h
ReverseIter
Vector/4
Vector/common/memberFunctions_4.h
```

*Chapter outline*

- Using the ReverseIter class with ordinary pointers
- Implementing rbegin and rend for Array and Vector

Recall from the previous chapter that ReverseIter's member types are all obtained via iterator\_traits (ReverseIter.h, lines 19-24):

```
typedef typename std::iterator_traits<Iter>::difference_type difference_type;
typedef typename std::iterator_traits<Iter>::pointer pointer;
typedef typename std::iterator_traits<Iter>::reference reference;
typedef typename std::iterator_traits<Iter>::value_type value_type;
typedef typename std::iterator_traits<Iter>::iterator_category
    iterator_category;
```

Suppose, however, that the contained Iter \_i is an ordinary pointer, such as an int\*. The compiler would generate the code

```
typedef std::iterator_traits<int*>::difference_type difference_type;
typedef std::iterator_traits<int*>::pointer pointer;
typedef std::iterator_traits<int*>::reference reference;
typedef std::iterator_traits<int*>::value_type value_type;
typedef std::iterator_traits<int*>::iterator_category iterator_category;
```

which, ordinarily, would fail to compile. Because int\* is a built-in type and not a class, it does not (and cannot) have its own member types (difference\_type, value\_type, etc.). The expressions

```
std::iterator_traits<int*>::difference_type
std::iterator_traits<int*>::pointer
std::iterator_traits<int*>::reference
std::iterator_traits<int*>::value_type
std::iterator_traits<int*>::iterator_category
```

are therefore invalid by default. The Standard Library solves this problem by using a language feature called “template specialization.” A template specialization is an alternate definition of a class template, to be used for a specific set of template arguments. Recall from Chapter 10.4 that the default definition of iterator\_traits is

```

namespace std
{
template <class Iterator>
struct iterator_traits
{
    typedef typename Iterator::pointer pointer;
    typedef typename Iterator::reference reference;
    typedef typename Iterator::difference_type difference_type;
    typedef typename Iterator::value_type value_type;
    typedef typename Iterator::iterator_category iterator_category;
};
};

```

For pointers and const pointers, the Standard Library defines the specializations

```

namespace std
{
template <class T>
struct iterator_traits<T*>           // Pointer specialization
{
    typedef typename T* pointer;
    typedef typename T& reference;
    typedef typename ptrdiff_t difference_type;
    typedef typename T value_type;
    typedef typename random_access_iterator_tag iterator_category;
};

template <class T>
struct iterator_traits<const T*>      // Const pointer specialization
{
    typedef typename const T* pointer;
    typedef typename const T& reference;
    typedef typename ptrdiff_t difference_type;
    typedef typename T value_type;
    typedef typename random_access_iterator_tag iterator_category;
};
};

```

`ptrdiff_t`, short for “pointer difference type,” is a Standard Library alias of an integer type. `random_access_iterator_tag` is used to indicate that the iterator category of a pointer / const pointer is “random access iterator.” Like the `bidirectional_iterator_tag` introduced in Chapter 10.3, this will be applied in a later chapter.

These specializations of `iterator_traits` allow the compiler to generate an appropriate `iterator_traits` class for a pointer or const pointer. Given the expressions

```

std::iterator_traits<int*>::pointer
std::iterator_traits<int*>::reference
std::iterator_traits<int*>::difference_type
std::iterator_traits<int*>::value_type
std::iterator_traits<int*>::iterator_category

```

```
std::iterator_traits<const double*>::pointer
std::iterator_traits<const double*>::reference
std::iterator_traits<const double*>::difference_type
std::iterator_traits<const double*>::value_type
std::iterator_traits<const double*>::iterator_category
```

for example, the compiler generates the classes

```
struct iterator_traits<int*>
{
    typedef typename int* pointer;
    typedef typename int& reference;
    typedef typename ptrdiff_t difference_type;
    typedef typename int value_type;
    typedef typename random_access_iterator_tag iterator_category;
};

struct iterator_traits<const double*>
{
    typedef typename const double* pointer;
    typedef typename const double& reference;
    typedef typename ptrdiff_t difference_type;
    typedef typename double value_type;
    typedef typename random_access_iterator_tag iterator_category;
};
```

When `int*` and `const double*` are used with the `ReverseIter` class template, the compiler can then generate the code

```
class ReverseIter<int*> // A reversed int*
{
public:
    typedef std::ptrdiff_t difference_type;
    typedef int* pointer;
    typedef int& reference;
    typedef int value_type;
    typedef std::random_access_iterator_tag iterator_category;

    // ...

private:
    int* _i;
};
```

```

class ReverseIter<const double*> // A reversed const double*
{
public:
    typedef std::ptrdiff_t difference_type;
    typedef const double* pointer;
    typedef const double& reference;
    typedef double value_type;
    typedef std::random_access_iterator_tag iterator_category;

    // ...

private:
    const double* _i;
};

```

Consider an array of 5 ints containing the elements {0, 1, 2, 3, 4},

0	1	2	3	4
		p	p._i	

The reverse sequence is {4, 3, 2, 1, 0}, where p is a ReverseIter<int\*> pointing to element 2. p contains an int\* \_i, pointing to element 3. The expression

```
p += 2 // Increment p by 2 elements (Point p to element 0)
```

is shorthand for

```
p.operator+=(2)
```

0	1	2	3	4
p	p._i			

The member function (ReverseIter.h, line 50)

```
ReverseIter& operator+=(difference_type offset);
```

increments the ReverseIter by the given offset by simply decrementing the contained Iter \_i by the given offset (memberFunctions\_2.h, lines 55-61).

The expression

```
p -= 2 // Decrement p by 2 elements (Point p back to element 2)
```

is shorthand for

```
p.operator+=(2)
```

0	1	2	3	4
		p	p._i	

The member function (ReverseIter.h, line 51)

```
ReverseIter& operator==(difference_type offset);
```

decrements the ReverseIter by the given offset by simply incrementing `_i` by the given offset (memberFunctions\_2.h, lines 63-69).

The statement

```
ReverseIter<int*> q = p + 1; // q points to +1 elements from p
```

is shorthand for

```
ReverseIter<int*> q = p.operator+(1);
```

0	1	2	3	4
		<code>q</code>	<code>q._i</code>	
				<code>p</code>
			<code>p._i</code>	

The member function (ReverseIter.h, line 42)

```
ReverseIter operator+(difference_type offset) const;
```

returns a ReverseIter in which the given offset is subtracted from `_i` (memberFunctions\_2.h, lines 34-39):

```
template <class Iter>
inline ReverseIter<Iter> ReverseIter<Iter>::operator+
(difference_type offset) const
{
    return ReverseIter(_i - offset);
}
```

The function call

```
p.operator+(1)
```

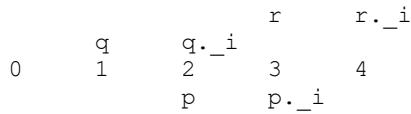
for example, returns a ReverseIter in which `_i` is initialized to  $(p._i - 1)$ , i.e. a ReverseIter in which `_i` points to element 2. `q` is then constructed from that ReverseIter.

The statement

```
ReverseIter<int*> r = p - 1; // r points to -1 elements from p
```

is shorthand for

```
ReverseIter<int*> r = p.operator-(1);
```



The member function (ReverseIter.h, line 43)

```
ReverseIter operator-(difference_type offset) const;
```

returns a ReverseIter in which the given offset is added to `_i` (memberFunctions\_2.h, lines 41-46):

```
template <class Iter>
inline ReverseIter<Iter> ReverseIter<Iter>::operator-
(difference_type offset) const
{
    return ReverseIter(_i + offset);
}
```

The function call

```
p.operator-(1)
```

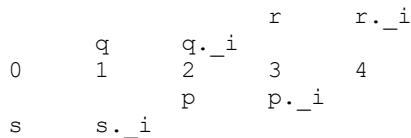
for example, returns a ReverseIter in which `_i` is initialized to  $(p._i + 1)$ , i.e. a ReverseIter in which `_i` points to element 4. `r` is then constructed from that ReverseIter.

The statement

```
ReverseIter<int*> s = 1 + q; // s points to +1 elements from q
```

is shorthand for

```
ReverseIter<int*> s = operator+(1, q);
```



The nonmember function (ReverseIter.h, lines 11-13)

```
template <class Iter>
ReverseIter<Iter> operator+(typename ReverseIter<Iter>::difference_type lhs,
const ReverseIter<Iter>& rhs);
```

returns a ReverseIter in which the given offset (the left operand) is added to the given ReverseIter (the right operand) (nonmemberFunctions.h, lines 3-9):

```
template <class Iter>
inline ReverseIter<Iter> operator+
    typename ReverseIter<Iter>::difference_type lhs,
    const ReverseIter<Iter>& rhs)
{
    return rhs + lhs;      // return rhs.operator+(lhs);
}
```

The expression

```
1 + q      // operator+(1, q)
```

for example, therefore becomes

```
q + 1      // q.operator+(1)
```

The reverse sequence in the above diagram is  $\{4, 3, 2, 1, 0\}$ , so  $q$  is greater than  $p$ , and  $p$  is greater than  $r$ . Similarly,  $r$  is less than  $p$ , and  $p$  is less than  $q$ . The expressions

```
q >= r      // Is q greater than or equal to r?
r <= q      // Is r less than or equal to q?
q > r       // Is q greater than r?
r < q       // Is r less than q?
```

are shorthand for

```
q.operator>=(r)
r.operator<=(q)
q.operator>(r)
r.operator<(q)
```

The comparison operators (ReverseIter.h, lines 35-38)

```
bool operator>=(const ReverseIter& rhs) const;
bool operator<=(const ReverseIter& rhs) const;
bool operator>(const ReverseIter& rhs) const;
bool operator<(const ReverseIter& rhs) const;
```

compare two ReverseIter objects by performing the reverse comparison on their contained Iters (memberFunctions\_2.h, lines 3-25).

The expressions

```
q - r      // Returns 2 (the distance from r to q)
r - q      // Returns -2 (the distance from q to r)
```

are shorthand for

```
q.operator-(r)
r.operator-(q)
```

The member function (ReverseIter.h, line 44)

```
difference_type operator-(const ReverseIter& rhs) const;
```

performs iterator subtraction by subtracting the Iter of the left operand from that of the right operand (memberFunctions\_2.h, lines 48-53). In the expressions

```
q - r      // q.operator-(r)
r - q      // r.operator-(q)
```

for example, the function computes

```
r._i - q._i
q._i - r._i
```

Given

0	1	2	3	4
p			p._i	

the expressions

```
p[2]      // Element 0 (the element at location p, offset 2)
p[-2]     // Element 4 (the element at location p, offset -2)
```

are shorthand for

```
p.operator[](2)
p.operator[](−2)
```

The array subscript operator (ReverseIter.h, line 41)

```
reference operator[](difference_type offset) const;
```

returns a reference to the element with the given offset (memberFunctions\_2.h, lines 27-32):

```
template <class Iter>
inline typename ReverseIter<Iter>::reference ReverseIter<Iter>::operator[](
    difference_type offset) const
{
    return _i[-offset - 1];      // return *(_i - offset - 1);
```

In the expressions

```
p[2]
p[-2]
```

for example, the function returns references to elements

```
p._i[-3]      // p._i[-2 - 1], or *(p._i - 2 - 1)
p._i[1]       // p._i[2 - 1], or *(p._i + 2 - 1)
```

Now that the ReverseIter class supports the same operations as an ordinary pointer, it can be used to implement the reverse iterators for Array and Vector. Lines 14 and 18 (Vector.h),

```
typedef ReverseIter<const_iterator> const_reverse_iterator;
typedef ReverseIter<iterator> reverse_iterator;
```

declare const\_reverse\_iterator and iterator as aliases of the types ReverseIter<const T\*> and ReverseIter<T\*>. The member functions rbegin and rend (lines 34-35, 42-43) are then defined identically to those of List (Vector/common/memberFunctions\_4.h, lines 3-25).

For the Array class, the implementation of the reverse iterators and rbegin / rend is identical to that of Vector (Array.h, lines 13, 17) (Array/common/memberFunctions\_2.h, lines 3-27).

The program in this chapter demonstrates reverse iteration over a Vector. Lines 10-13 (main.cpp) construct a Vector<int> v containing the elements {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Line 15,

```
cout << "\nv contains " << v.rend() - v.rbegin() << " elements\n";
```

demonstrates ReverseIter's subtraction operator, and is equivalent to

```
cout << "\nv contains " << v.rend().operator-(v.rbegin()) << " elements\n";
```

The resultant output is

```
v contains 10 elements
```

Lines 16-21 print the elements in reverse order by applying the array subscript operator to a const\_reverse\_iterator i, generating the output

```
The reverse sequence is:
9 8 7 6 5 4 3 2 1 0
```

Line 24-25,

```
cout << "The 3rd element is " << *(i + 2) << endl;
cout << "The 4th element is " << *(3 + i) << endl;
```

demonstrate the member / nonmember operator+ functions, and are equivalent to

```
cout << "The 3rd element is " << * (i.operator+(2)) << endl;
cout << "The 4th element is " << * (operator+(3, i)) << endl;
```

The resultant output is

```
The 3rd element is 7
The 4th element is 6
```



# Part 12: Single-Block Double-Ended Queues

## 12.1: Introducing the Ring Class Template

*Source files and folders*

*Ring/1*  
*Ring/common/memberFunctions\_1.h*

*Chapter outline*

- Default constructor / destructor
- Accessor functions (*empty / size / capacity, front / back*)
- *push\_back / reserve*

Vectors excel at element access: because the elements are stored in a single block of memory, any given element can be quickly retrieved via simple pointer arithmetic. *push\_back* is also extremely fast, as long as the block is not full (in which case a time-consuming reallocation is required). *pop\_back*, which simply destroys the back element and updates the Vector's size, is always efficient. *insert* and *erase*, however, are inefficient because they require a loop of swaps or overwrites.

Lists, on the other hand, excel at *insert* / *erase*, which simply involves creating / destroying a node, then updating the links of at most 3 nodes (the new node and left / right neighbors). Element access, however, is inefficient: because the nodes are stored at non-contiguous memory addresses, the only way to retrieve a given element is to traverse the List one node at a time.

This chapter introduces the Ring (“ring buffer”) class template. Ring is a hybrid of List and Vector, supporting efficient push / pop at both ends as well as fast element access via pointer arithmetic. A Ring stores all of its elements in a single memory block (like a Vector) while maintaining pointers to the head and tail elements (like a List). The block is treated as if it were connected end-to-end, i.e. as if it formed a contiguous “ring” of memory. Suppose, for example, that the sequence of ints

1 2 3 4 5 6 7

is stored in a Ring with a capacity of 12 elements, and that

B is a pointer to the beginning of the memory block  
E is a pointer to the end of the memory block  
H is a pointer to the head element (1)  
T is a pointer to the tail element (7)

Possible memory layouts for the above sequence include:

```

H          T
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |   |   |   |   |           // Layout 1
B           E

```

```

          H          T
|   |   |   | 1 | 2 | 3 | 4 | 5 | 6 | 7 |   |   |           // Layout 2
B           E

```

```

          T          H
| 4 | 5 | 6 | 7 |   |   |   |   | 1 | 2 | 3 |           // Layout 3
B           E

```

The elements are traversed left-to-right, from the head (H) to the tail (T). If the end of the block (E) is reached, the sequence resumes at the beginning of the block (B). Each of the above layouts can thus be depicted as a ring, traversed clockwise beginning at the head:

```

          | 4 |
          | 3 |   | 5 |
          | 6 |   | 7 |           // Layout 1
H   | 2 |
| 1 |
B   |   |
E   |   |
      |   |

```

```

          H
          | 1 |
          | 2 |   | 3 |
          | 4 |   | 5 |
          | 6 |   | 7 |           // Layout 2
B   |   |
E   |   |
      |   |

```

```

          T
          | 7 |
          | 6 |   |   |
          | 5 |   |   |
          | 4 |   |   |
          | 3 |   |   |
          E   | 2 |   H   |   |
          | 1 |           // Layout 3

```

The Ring class is defined in the header file Ring.h (lines 8-45). The template parameter T (line 8) represents the type of elements stored in the Ring. The member types (lines 12-18) are identical to those of List and Vector. The data members are (lines 38-44)

```

T* _block;           // Pointer to the memory block storing the elements (B)
T* _endOfBlock;     // Pointer to the end of the block (E)
T* _head;           // Pointer to the first element in the sequence (H)
T* _tail;           // Pointer to the last element in the sequence (T)
size_type _size;    // Number of elements currently stored in the block

```

```
size_type _capacity; // Maximum number of elements storable in the block
Allocator<T> _alloc; // Allocates memory and constructs / destroys elements
```

The member functions empty, size, and capacity (Ring.h, lines 23-25 / memberFunctions\_1.h, lines 21-37) are identical to those of Vector (Chapter 9.1).

The member functions front and back (Ring.h, lines 26-27, 29-30) return references to the head and tail elements (memberFunctions\_1.h, lines 39-61).

The default constructor (Ring.h, line 20) allocates a block large enough to hold 3 elements. Because the Ring is empty, the head and tail pointers are initialized to null (memberFunctions\_1.h, lines 3-12):

```
template <class T>
Ring<T>::Ring():
    _head(nullptr),
    _tail(nullptr),
    _size(0),
    _capacity(3)
{
    _block = _alloc.allocate(_capacity);
    _endOfBlock = _block + _capacity - 1;
}
```

The constructor thus creates a Ring with the memory layout



Like a Vector, a Ring automatically reallocates its memory block as needed. The private member function (line 35)

```
void _realloc(size_type newCapacity);
```

replaces the Ring's current memory block with a new block, where newCapacity is the capacity of the new block. Reallocation entails the following steps:

- Create a new block large enough to hold newCapacity elements
- Copy each element in the current block to the new block
- Destroy each element in the current block, then deallocate (release) the current block
- Adopt the new block by updating the Ring's internal pointers (\_block, \_endOfBlock, \_head, \_tail)
- Update the \_capacity

The function is defined in lines 95-126 (memberFunctions\_1.h). Consider, for example, a full Ring<int> containing the elements {1, 2, 3}, with the memory layout

```
T      H          // _size is 3
| 3 | 1 | 2 |      // _capacity is 3
B      E
```

To calculate the capacity of the new block, Ring uses the same formula as Vector (Chapter 9.1):

$$\begin{aligned} \text{newCapacity} &= 1.5 \times (\text{currentCapacity} + 2) \\ &= 7 \end{aligned}$$

The `reallocate` function performs the following operations:

```

// Allocate a new block with the given newCapacity (7)

T* newBlock = _alloc.allocate(newCapacity);

size_type totalElementsCopied = 0;           // totalElementsCopied is 0
T* sourceElement = _head;                   // sourceElement points to element 1
T* newElement = newBlock;                   // newElement points to newBlock

// N denotes newBlock
// s denotes sourceElement
// n denotes newElement

      T      H
| 3 | 1 | 2 |
 B          E
      s

|   |   |   |   |   |   |
      N
      n

```

## Iteration 1:

```

// Copy the current source element to the new block, then destroy it

_alloc.construct(newElement, *sourceElement);
_alloc.destroy(sourceElement);
    T      H
    | 3   |     | 2  |
    B          E
        S
| 1 |     |     |     |     |     |
N
n

++totalElementsCopied;           // totalElementsCopied is 1
++newElement;                   // newElement points to (newBlock + 1)

// Get the next source element

if (sourceElement == _endOfBlock)
    sourceElement = _block;
else
    ++sourceElement;           // sourceElement points to element 2

```

The diagram consists of a series of vertical bars of increasing height from left to right. Above the first bar is the label 'T'. Above the second bar is 'H'. Above the third bar is 'B'. Above the fourth bar is 'E'. Above the fifth bar is 'S'. Above the sixth bar is '1'. Above the seventh bar is 'N'. Below the eighth bar is the label 'n'.

## Iteration 2:

```

// Copy the current source element to the new block, then destroy it

_alloc.construct(newElement, *sourceElement);
_alloc.destroy(sourceElement);

      T      H
      |      |
      3      |      |
      B          E
              S

      | 1  | 2  |      |      |      |      |
      N
      n

++totalElementsCopied;           // totalElementsCopied is 2
++newElement;                   // newElement points to (newBlock + 2)

// Get the next source element

if (sourceElement == _endOfBlock)
    sourceElement = _block;           // sourceElement points to element 3
else
    ++sourceElement;

      T      H
      |      |
      3      |      |
      B          E
              S

      | 1  | 2  |      |      |      |      |
      N
      n

```

### Iteration 3:

```
// Copy the current source element to the new block, then destroy it
_alloc.construct(newElement, *sourceElement);
_alloc.destroy(sourceElement);
```

The member function reserve (Ring.h, line 31, memberFunctions\_1.h, lines 63-68) is identical to that of Vector (Chapter 9.2).

The member function (Ring.h, line 32)

```
void push_back(const T& source);
```

inserts a copy of the given source object at the end of the Ring. The function consists of 3 main branches (memberFunctions 1.h, lines 70-93).

Case 1: The Ring is empty (lines 73-77, 91-92)

Suppose that a new element, 1, is to be pushed back into an empty Ring<int> with a capacity of 3:

```

|   |   |   |
B       E

// Point the _head and _tail to the beginning of the block

_head = _block;
_tail = _block;

H
T
|   |   |   |
B       E

// Construct the new element at the tail and update the size

_alloc.construct(_tail, source);
++_size;

H
T
| 1 |   |   |
B       E

```

Case 2: The Ring is full (`size() == capacity()`) (lines 78-82, 91-92)

Consider pushing back a new element, 4, into a full Ring with a capacity of 3:

```

T   H
| 3 | 1 | 2 |
B       E

// Reallocate the block to make room for the new element

_reallocate(static_cast<size_type>(1.5 * (capacity() + 2)));

H   T
| 1 | 2 | 3 |   |   |   |
B           E
N

// Point the _tail to the one-past-the-last element

++_tail;

H           T
| 1 | 2 | 3 |   |   |   |
B           E

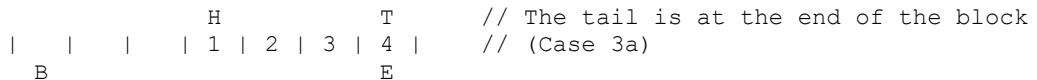
```

```
// Construct the new element at the tail and update the size
_alloc.construct(_tail, source);
++_size;
```



Case 3: The Ring is neither empty nor full (lines 83-89, 91-92)

This branch consists of 2 sub-cases: the tail is at the end of the block (Case 3a), or it is not (Case 3b). Given the Ring



for example, suppose that a new element 5 is to be pushed back.

```
// If the tail is at the end of the block, then point _tail to the
// beginning of the block; otherwise, point _tail to the one-past-the-last
// element

if (_tail == _endOfBlock)
    _tail = _block;                      // Point _tail to the beginning of the block
else
    ++_tail;

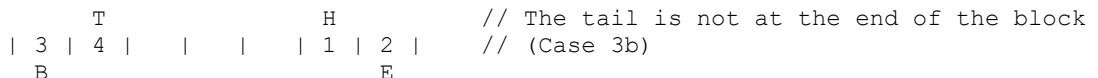
      T          H
|   |   |   | 1 | 2 | 3 | 4 |
      B          E

// Construct the new element at the tail and update the size

_alloc.construct(_tail, source);
++_size;

      T          H
| 5 |   |   | 1 | 2 | 3 | 4 |
      B          E
```

Given the Ring



suppose a that new element 5 is to be pushed back.

```

// If the tail is at the end of the block, then point _tail to the
// beginning of the block; otherwise, point _tail to the one-past-the-last
// element

if (_tail == _endOfBlock)
    _tail = _block;
else
    ++_tail;                                // Point _tail to the one-past-the-last element

    T          H
| 3 | 4 |   |   | 1 | 2 |
B           E

// Construct the new element at the tail and update the size

_alloc.construct(_tail, source);
++_size;

    T          H
| 3 | 4 | 5 |   |   | 1 | 2 |
B           E

```

The private member function `_destroyAllElements` (Ring.h, line 36) traverses each element from head to tail, destroying it along the way (memberFunctions\_1.h, lines 128-145). The traversal is similar to that of the `_reallocate` function (lines 101-117); the only difference is that instead of counting the `totalElementsCopied`, it counts the `totalElementsDestroyed`.

The destructor (Ring.h, line 21) simply calls `_destroyAllElements` and releases the memory block (memberFunctions\_1.h, lines 14-19).

The program in this chapter demonstrates a series of calls to `push_back` on a `Ring<Traceable<int>>` `r`. Line 11 (main.cpp) constructs the Ring, and the loop (lines 13-17) pushes back the elements {0, 1, 2, 3, 4, 5, 6, 7, 8}.

The resultant output is

```

v 0      H          // r.push_back(0);
c 0      | 0 |   |   |
~ 0      T

v 1      H      T          // r.push_back(1);
c 1      | 0 | 1 |   |
~ 1

v 2      H          T          // r.push_back(2);
c 2      | 0 | 1 | 2 |
~ 2

v 3      // r.push_back(3);
c 0      // _reallocate (newCapacity = 7)
~ 0

```

```

c 1
~ 1      H      T
c 2      | 0 | 1 | 2 |   |   |   |
~ 2      H      T
c 3      | 0 | 1 | 2 | 3 |   |   |   |
~ 3

v 4      H      T          // r.push_back(4);
c 4      | 0 | 1 | 2 | 3 | 4 |   |   |
~ 4

v 5      H      T          // r.push_back(5);
c 5      | 0 | 1 | 2 | 3 | 4 | 5 |   |
~ 5

v 6      H      T          // r.push_back(6);
c 6      | 0 | 1 | 2 | 3 | 4 | 5 | 6 |   |
~ 6

v 7      // r.push_back(7);
c 0      // _realloc (newCapacity = 13)
~ 0

c 1
~ 1
c 2
~ 2
c 3
~ 3
c 4
~ 4
c 5
~ 5      H      T
c 6      | 0 | 1 | 2 | 3 | 4 | 5 | 6 |   |
~ 6      H      T
c 7      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |   |
~ 7

v 8      H      T          // r.push_back(8);
c 8      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |   |
~ 8

```

When main terminates, r's destructor generates the output

	H	T
~ 0	1   2   3   4   5   6   7   8	
~ 1	2   3   4   5   6   7   8	
~ 2	3   4   5   6   7   8	
~ 3	4   5   6   7   8	
~ 4	5   6   7   8	
~ 5	6   7   8	
~ 6	7   8	
~ 7	8	
~ 8	8	

## 12.2: Push Front and Array Subscript

*Source files and folders*

*Ring/2  
Ring/common/memberFunctions\_2.h*

*Chapter outline*

- *push\_front*
- *operator[]*

The member function (Ring.h, line 34)

```
void push_front(const T& source);
```

inserts a copy of the given source object at the front of the Ring (memberFunctions\_2.h, lines 16-39). The function, which consists of 3 main branches, is the mirror image of *push\_back*.

Case 1: The Ring is empty (lines 19-23, 37-38)

Suppose that a new element, 1, is to be pushed to the front of an empty Ring<int> with a capacity of 3:

```
|   |   |   |
 B   E

// Point the _head and _tail to the beginning of the block

_head = _block;
_tail = _block;

H
T
|   |   |   |
B   E

// Construct the new element at the head and update the size

_alloc.construct(_head, source);
++_size;

H
T
| 1 |   |   |
B   E
```

Case 2: The Ring is full (*size() == capacity()*) (lines 24-28, 37-38)

Consider pushing front a new element, 4, into a full Ring with a capacity of 3:

```

T      H
| 1 | 3 | 2 |
B      E

// Reallocate the block to make room for the new element

_reallocate(static_cast<size_type>(1.5 * (capacity() + 2)));

H      T
| 3 | 2 | 1 |   |   |   |
B           E

// Point the _head to the end of the block

_head = _endOfBlock;

H      T      H
| 3 | 2 | 1 |   |   |   |
B           E

// Construct the new element at the head and update the size

_alloc.construct(_head, source);
++_size;

H      T      H
| 3 | 2 | 1 |   |   |   | 4 |
B           E

```

Case 3: The Ring is neither empty nor full (lines 29-35, 37-38)

This branch consists of 2 sub-cases: the head is at the beginning of the block (Case 3a), or it is not (Case 3b). Given the Ring

```

H      T      // The head is at the beginning of the
| 4 | 3 | 2 | 1 |   |   |   // block (Case 3a)
B           E

```

for example, suppose that a new element 5 is to be pushed front.

```

// If the head is at the beginning of the block, then point _head to the
// end of the block; otherwise, point _head to the one-before-the-first
// element

if (_head == _block)
    _head = _endOfBlock;    // Point _head to the end of the block
else
    --_head;

```

```

        T          H
| 4 | 3 | 2 | 1 |   |   |
B           E

// Construct the new element at the head and update the size

_alloc.construct(source, _head);
++_size;

        T          H
| 4 | 3 | 2 | 1 |   |   | 5 |
B           E

```

Given the Ring

```

        T          H          // The head is not at the beginning of
| 2 | 1 |   |   | 4 | 3 |          // the block (Case 3b)
B           E

```

suppose that a new element 5 is to be pushed front.

```

// If the head is at the beginning of the block, then point _head to the
// end of the block; otherwise, point _head to the one-before-the-first
// element

if (_head == _block)
    _head = _endOfBlock;
else
    --_head;                      // Point _head to the one-before-the-first element

        T          H
| 2 | 1 |   |   | 4 | 3 |
B           E

// Construct the new element at the head and update the size

_alloc.construct(source, _head);
++_size;

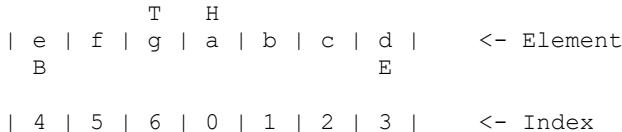
        T          H
| 2 | 1 |   |   | 5 | 4 | 3 |
B           E

```

The private member function (Ring.h, line 38)

```
reference _element(size_type index) const;
```

returns a reference to the element at the given index (offset from the head). Consider, for example, a Ring<char> containing the sequence {a, b, c, d, e, f, g}



The expressions

```

_element(0)
_element(2)
_element(5)

```

return references to elements a, c, and f respectively. The function is defined in lines 41-50 (memberFunctions\_2.h). The function first calculates headToEndDistance, the distance (number of elements) from the head to the end of the block:

```
size_type headToEndDistance = _endOfBlock - _head;
```

In the above diagram, headToEndDistance is 3. If the given index is less than or equal to headToEndDistance, the function simply returns

```
_head[index]
```

Elements a, b, c, and d, for example, are

```

_head[0]          // *(_head + 0)
_head[1]          // *(_head + 1)
_head[2]          // *(_head + 2)
_head[3]          // *(_head + 3)

```

If the given index is greater than headToEndDistance, the function returns

```
_block[index - headToEndDistance - 1]
```

Elements e, f, and g, for example, are

```

_block[4 - 3 - 1] // _block[0]    // *(_block + 0)
_block[5 - 3 - 1] // _block[1]    // *(_block + 1)
_block[6 - 3 - 1] // _block[2]    // *(_block + 2)

```

The array subscript operator (Ring.h, lines 28, 32),

```

const_reference operator[](size_type index) const;
reference operator[](size_type index);

```

returns a reference to the element at the given index by simply calling `_element` (memberFunctions\_2.h, lines 3-14). Note that in both cases, the call to `_element` returns a (non-const) reference. In the `const` version of `operator[]` (line 7), this reference is implicitly converted to a `const_reference`.

The program in this chapter demonstrates `push_front` and the array subscript operator. Line 11 (`main.cpp`) constructs a `Ring<Traceable<int>>` `r`, and the loop (lines 13-17) pushes the elements {0, 1, 2, 3, 4, 5, 6, 7, 8} to the front. The resultant output is

```

v 0      H          // r.push_front(0);
c 0      | 0 |      |
~ 0      T

v 1      T          H          // r.push_front(1);
c 1      | 0 |      | 1 |
~ 1

v 2      T  H          // r.push_front(2);
c 2      | 0 | 2 | 1 |
~ 2

v 3      // r.push_front(3);
c 0      // _reallocate (newCapacity = 7)
~ 0
c 1
~ 1      H          T
c 2      | 2 | 1 | 0 |      |      |      |
~ 2      T          H
c 3      | 2 | 1 | 0 |      |      | 3 |
~ 3

v 4      T          H          // r.push_front(4);
c 4      | 2 | 1 | 0 |      |      | 4 | 3 |
~ 4

v 5      T          H          // r.push_front(5);
c 5      | 2 | 1 | 0 |      | 5 | 4 | 3 |
~ 5

v 6      T  H          // r.push_front(6);
c 6      | 2 | 1 | 0 | 6 | 5 | 4 | 3 |
~ 6

v 7      // r.push_front(7);
c 0      // _reallocate (newCapacity = 13)
~ 0
c 1
~ 1
c 2
~ 2
c 3
~ 3
c 4
~ 4
c 5
~ 5      H          T
c 6      | 6 | 5 | 4 | 3 | 2 | 1 | 0 |      |      |      |      |

```

```

~ 6                                     T
c 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |   |   |   |   |   |   | 7 |
~ 7

v 8                                     T
c 8 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |   |   |   |   |   | 8 | 7 |
~ 8

```

Lines 19-27 then construct a Ring<char> s with the layout

```

T   H
| e | f | g | a | b | c | d |   <- Element
B           E

| 4 | 5 | 6 | 0 | 1 | 2 | 3 |   <- Index

```

Lines 29-32 traverse s using the array subscript operator, generating the output

```
s contains 7 elements:
a b c d e f g
```

When main terminates, s is destroyed, followed by r. r's destructor generates the output

```

~ 8
~ 7
~ 6
~ 5
~ 4
~ 3
~ 2
~ 1
~ 0

```

## 12.3: Pop Front, Pop Back, and Clear

*Source files and folders*

*Ring/3*  
*Ring/common/memberFunctions\_3.h*

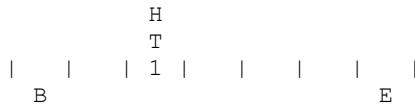
*Chapter outline*

- *pop\_front / pop\_back*
- *clear*

The member function *pop\_back* (*Ring.h*, line 37) removes the tail element from the Ring (*memberFunctions\_3.h*, lines 25-45). The function consists of 3 branches.

Case 1: The Ring contains a single element (*size() == 1*) (lines 28, 30-34, 44)

Given the Ring



*pop\_back* performs the following operations:

```
// Destroy the tail element
_alloc.destroy(_tail);

          H
          T
|   |   |   |   |   |   |
B           E

// Point _head and _tail to null, then update the size
_head = nullptr;
_tail = nullptr;
--_size;

|   |   |   |   |   |   |
B           E
```

Case 2: The Ring contains more than 1 element, and the tail is at the beginning of the block (*\_tail == \_block*) (lines 28, 35-38, 44)

```

T          H          // The tail is at the beginning of the
| 4 |     |     | 1 | 2 | 3 |          // block
B           E

// Destroy the tail element

_alloc.destroy(_tail);

T          H
|     |     | 1 | 2 | 3 |
B           E

// Point _tail to the end of the block, then update the size

_tail = _endOfBlock;
--_size;

T          H          T
|     |     | 1 | 2 | 3 |
B           E

```

Case 3: The Ring contains more than 1 element, and the tail is not at the beginning of the block (lines 28, 39-42, 44)

```

T          H          // The tail is not at the beginning of
| 3 | 4 |     |     | 1 | 2 |          // the block
B           E

// Destroy the tail element

_alloc.destroy(_tail);

T          H
| 3 |     |     | 1 | 2 |
B           E

// Point _tail to the last element in the sequence (the previous element in
// the block), then update the size

--_tail;
--_size;

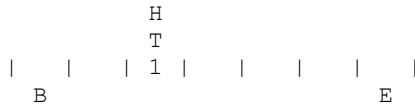
T          H
| 3 |     |     | 1 | 2 |
B           E

```

`pop_front` (Ring.h, line 36), which removes the head element from the Ring (memberFunctions\_3.h, lines 3-23), is the mirror image of `pop_back`. The function consists of 3 branches.

Case 1: The Ring contains a single element (`size() == 1`) (lines 6, 8-12, 22)

Given the Ring



for example, `pop_front` performs the following operations:

```
// Destroy the head element
_alloc.destroy(_head);

H
T
| | | | | | | |
B E

// Point _head and _tail to null, then update the size

_head = nullptr;
_tail = nullptr;
--_size;

| | | | | | | |
B E
```

Case 2: The Ring contains more than 1 element, and the head is at the end of the block  
`(_head == _endOfBlock)` (lines 6, 13-16, 22)

```
T H // The head is at the end of the block
| 3 | 2 | 1 | | | | 4 |
B E

// Destroy the head element
_alloc.destroy(_head);

T H
| 3 | 2 | 1 | | | | |
B E

// Point _head to the beginning of the block, then update the size

_head = _block;
--_size;

H T
| 3 | 2 | 1 | | | | |
B E
```

Case 3: The Ring contains more than 1 element, and the head is not at the end of the block (lines 6, 17-20, 22)

```

    T          H          // The head is not at the end of
| 2 | 1 |     |     | 4 | 3 |          // the block
    B          E

// Destroy the head element

_alloc.destroy(_head);

    T          H
| 2 | 1 |     |     | 3 |
    B          E

// Point _head to the first element in the sequence (the next element in the
// block), then update the size

++_head;
--_size;

    T          H
| 2 | 1 |     |     | 3 |
    B          E

```

The member function clear (Ring.h, line 38) removes all elements from the Ring by calling `_destroyAllElements`, pointing `_head` and `_tail` to null, and resetting the `_size` to 0 (memberFunctions\_3.h, lines 47-55).

The program in this chapter demonstrates `pop_front` and `pop_back`. Lines 11-18 (main.cpp) construct 2 Rings of `Traceable<int>`, `r` and `s`, with the layout:

```

    T          H
| 6 | 5 | 4 | 3 | 2 | 1 | 0 |     |     |     | 8 | 7 |
    B          E

```

The loop in lines 22-23 calls `pop_back` on `r` until it is empty, generating the output

```

~ 0      | 6 | 5 | 4 | 3 | 2 | 1 |     |     |     | 8 | 7 |
                    T          H
~ 1      | 6 | 5 | 4 | 3 | 2 |     |     |     | 8 | 7 |
                    T          H
~ 2      | 6 | 5 | 4 | 3 |     |     |     | 8 | 7 |
                    T          H
~ 3      | 6 | 5 | 4 |     |     |     | 8 | 7 |
                    T          H
~ 4      | 6 | 5 |     |     |     | 8 | 7 |
                    T          H
~ 5      | 6 |     |     |     | 8 | 7 |

```

The loop in lines 27-28 then calls `pop_front` on `s` until it is empty, generating the output

~ 8		6		5		4		3		2		1		0							H
~ 7		H		6		5		4		3		2		1		T					
~ 6			H		5		4		3		2		1		0						
~ 5				H		4		3		2		1		0							
~ 4					H		3		2		1		0		T						
~ 3						H		2		1		0			T						
~ 2							H		1		0				T						
~ 1								H							T						
~ 0									H							T					



## 12.4: Implementing the Iterators

*Source files and folders*

*ConstIter*  
*Ring/4*  
*Ring/common/memberFunctions\_4.h*  
*RingIter*

*Chapter outline*

- *The RingIter class*
- *Implementing the remaining member functions of the ConstIter class*
- *Implementing begin / end / rbegin / rend for the Ring class*

The following tables summarize the iterators developed thus far:

Type	Description
ConstIter<Container>	<p>Contains an object <code>_i</code> of type <code>Container::iterator</code></p> <p>pointer and reference are aliases of  <code>Container::const_pointer</code> and  <code>Container::const_reference</code></p> <p>Overloaded operators apply the corresponding operation to <code>_i</code></p>
ReverseIter<Iter>	<p>Contains an object <code>_i</code> of type <code>Iter</code></p> <p>Overloaded operators apply the reverse operation to <code>_i</code></p>

Type	Alias of
<code>Array&lt;T&gt;::iterator</code>	<code>T*</code>
<code>Array&lt;T&gt;::const_iterator</code>	<code>const T*</code>
<code>Array&lt;T&gt;::reverse_iterator</code>	<code>ReverseIter&lt;Array&lt;T&gt;::iterator&gt;</code>
<code>Array&lt;T&gt;::const_reverse_iterator</code>	<code>ReverseIter&lt;Array&lt;T&gt;::const_iterator&gt;</code>
<code>Vector&lt;T&gt;::iterator</code>	<code>T*</code>
<code>Vector&lt;T&gt;::const_iterator</code>	<code>const T*</code>
<code>Vector&lt;T&gt;::reverse_iterator</code>	<code>ReverseIter&lt;Vector&lt;T&gt;::iterator&gt;</code>
<code>Vector&lt;T&gt;::const_reverse_iterator</code>	<code>ReverseIter&lt;Vector&lt;T&gt;::const_iterator&gt;</code>
<code>List&lt;T&gt;::iterator</code>	<code>ListIter&lt;List&lt;T&gt;&gt;</code>
<code>List&lt;T&gt;::const_iterator</code>	<code>ConstIter&lt;List&lt;T&gt;::iterator&gt;</code>
<code>List&lt;T&gt;::reverse_iterator</code>	<code>ReverseIter&lt;List&lt;T&gt;::iterator&gt;</code>
<code>List&lt;T&gt;::const_reverse_iterator</code>	<code>ReverseIter&lt;List&lt;T&gt;::const_iterator&gt;</code>

A Ring<T>::iterator contains two data members, `_ring` (a pointer to the underlying Ring) and `_index` (the index value of the referent element). Consider, for example, a Ring<char> `x`

T	H
e   f   g   a   b   c   d	// Element
4   5   6   0   1   2   3	// Index
k	i

where `i` is a Ring<char>::iterator pointing to element `c`, and `k` is a Ring<char>:: iterator pointing to element `f`.

```
i._ring is a pointer to x, and i._index is 2
k._ring is a pointer to x, and k._index is 5
```

### The expressions

```
i == k // Equality (Return true if i and k point to the same element)
i != k // Inequality (Return true if i and k point to different elements)
i > k // Greater than (Return true if i's referent proceeds k's referent)
i < k // Less than (Return true if i's referent precedes k's referent)
i >= k // Greater than or equal to
i <= k // Less than or equal to

*i // Dereference (Return the referent of i, element c)
i[2] // Array subscript (Return the element with an offset of 2, element e)

i + 2 // Addition (Return an iterator pointing to element e)
2 + i // Addition (Return an iterator pointing to element e)
i - 2 // Subtraction (Return an iterator pointing to element a)
k - i // Subtraction (Return the distance (# of elements) from i to k, 3)

++i // Prefix increment (Point i to the next element, d, and return i)
--i // Prefix decrement (Point i to the previous element, b, and return i)

i++ // Postfix increment (Point i to d, but return an iterator to c)
i-- // Postfix decrement (Point i to b, but return an iterator to c)

i += 4 // Addition assignment (Increment i by 4, so that it points to g)
i -= 2 // Subtraction assignment (Decrement i by 2, so that it points to a)
```

are shorthand for

```
i.operator==(k) // Return true if (i._index == k._index)
i.operator!=(k) // Return true if (i._index != k._index)
i.operator>(k) // Return true if (i._index > k._index)
i.operator<(k) // Return true if (i._index < k._index)
i.operator>=(k) // Return true if (i._index >= k._index)
i.operator<=(k) // Return true if (i._index <= k._index)

i.operator*() // Return a reference to
// Return i._ring->_element(i._index), which is x._element(2)
```

```

i.operator[](2) // Return a reference to element e
                // Return i._ring->_element(i._index + 2), which is
                // x._element(2 + 2)

i.operator+(2) // Return an iterator in which _ring and _index are
                // initialized to i._ring (&x) and (i._index + 2) (4)

operator+(2, i) // Return (i + 2), or i.operator+(2)

i.operator-(2) // Return an iterator in which _ring and _index are
                // initialized to i._ring (&x) and (i._index - 2) (0)

k.operator-(i) // Return k._index - i._index (5 - 2)

i.operator++() // Increment i._index
i.operator--() // Decrement i._index
i.operator++(0) // Create a copy of i, increment i._index, return the copy
i.operator--(0) // Create a copy of i, decrement i._index, return the copy
i.operator+=(4) // Increment i._index by 4
i.operator==(2) // Decrement i._index by 2

```

Ring<T>::iterator is implemented as the RingIter class template (RingIter.h, lines 15-54). The template parameter Ring (line 15) represents the type of the underlying Ring. For the class RingIter<Ring<char>>, for example, the compiler generates the the code

```

class RingIter<Ring<char>>
{
public:
    friend class Ring<char>;

    typedef Ring<char>::pointer pointer;
    typedef Ring<char>::reference reference;
    typedef Ring<char>::difference_type difference_type;
    typedef Ring<char>::value_type value_type;
    typedef std::random_access_iterator_tag iterator_category;

    // ...

private:
    RingIter<Ring<char>>(const Ring<char>* ring, int index); // Constructor

    const Ring<char>* _ring; // Pointer to the underlying Ring<char>
    int _index; // Index of the referent element
};

```

The default constructor (line 27) constructs a RingIter without an underlying Ring or index value (lines 63-67).

The private constructor (line 50) initializes \_ring and \_index using the given parameters (lines 195-201). This constructor will be used by Ring's member functions, so Ring must be declared as a friend of RingIter (line 19).

The comparison operators (lines 29-34) compare two RingIters by simply comparing their `_index` values (lines 69-103).

The member access / dereference operators (lines 35-36) retrieve the referent element by calling the underlying Ring's `_element` function (lines 105-115).

The array subscript operator (line 37) retrieves the element with the given offset by adding it to the `_index`, then passing that value to `_element` (lines 117-122).

The addition / subtraction operators (lines 38-39) return a RingIter in which the `_index` is initialized with the given offset (lines 124-134).

The iterator subtraction operator (line 40) subtracts the `_index` of the right operand from that of the left operand (lines 136-141).

The increment / decrement / compound assignment operators (lines 42-47) simply apply the corresponding operation to the `_index` (lines 143-193).

The nonmember addition operator (lines 11-13) adds the left operand to the right operand via the right operand's `operator+ member function` (lines 56-61):

```
template <class Ring>
inline RingIter<Ring> operator+(typename RingIter<Ring>::difference_type lhs,
    const RingIter<Ring>& rhs)
{
    return rhs + lhs;      // return rhs.operator+(lhs);
```

The remaining member functions of the ConstIter class (`ConstIter.h`, lines 37-40, 43-46, 52-53),

```
bool operator>=(const ConstIter& rhs) const;
bool operator<=(const ConstIter& rhs) const;
bool operator>(const ConstIter& rhs) const;
bool operator<(const ConstIter& rhs) const;

reference operator[](difference_type offset) const;
ConstIter operator+(difference_type offset) const;
ConstIter operator-(difference_type offset) const;
difference_type operator-(const ConstIter& rhs) const;

ConstIter& operator+=(difference_type offset);
ConstIter& operator-=(difference_type offset);
```

are used to implement `Ring<T>::const_iterator`. Recall from Chapter 10.4 that `ConstIter`'s member functions simply performed the corresponding operation on the contained Iter `_i`. The same is true of these member functions (`ConstIter/memberFunctions_2.h` (lines 3-69)).

The nonmember `operator+` function (`ConstIter.h`, lines 11-14)

```
template <class Container>
ConstIter<Container> operator+
    typename ConstIter<Container>::difference_type lhs,
    const ConstIter<Container>& rhs);
```

simply adds the left operand to the right operand via the right operand's operator+ member function (ConstIter/nonmemberFunctions.h, lines 3-8):

```
template <class Iter>
inline ConstIter<Iter> operator+
    typename ConstIter<Iter>::difference_type lhs,
    const ConstIter<Iter>& rhs)
{
    return rhs + lhs;      // return rhs.operator+(lhs);
}
```

Line 15 (Ring.h),

```
friend class RingIter<Ring>;
```

declares RingIter as a friend of the Ring class. This allows RingIter's member functions operator->, operator\*, and operator[] to call the private member function \_element on the underlying Ring. Lines 21-22,

```
typedef RingIter<Ring> iterator;
typedef ReverseIter<iterator> reverse_iterator;
```

declare Ring<T>::iterator and Ring<T>::reverse\_iterator as aliases of the types RingIter<Ring<T>> and ReverseIter<Ring<T>::iterator>. Lines 17-18,

```
typedef ConstIter<Ring> const_iterator;
typedef ReverseIter<const_iterator> const_reverse_iterator;
```

declare Ring<T>::const\_iterator and Ring<T>::const\_reverse\_iterator as aliases of the types ConstIter<Ring<T>> and ReverseIter<Ring<T>::const\_iterator>.

The member functions begin, end, rbegin, and rend (lines 35-38, 43-46) are defined in the header file Ring/common/memberFunctions\_4.h (lines 3-49).

The non-const version of begin (line 30) returns an iterator pointing to the head element (index 0), while the non-const version of end (line 36) returns an iterator pointing to the one-past-the-last element (a Ring's size() is equivalent to the index of its one-past-the-last element).

The const versions of begin / end (lines 6 and 12) follow the same design pattern as that of List (Chapter 10.4).

The implementation of rbegin / rend (lines 18, 24, 42, 48) is identical to that of List and Vector (Chapters 11.1 and 11.2).

The program in this chapter demonstrates the RingIter class. Lines 10-13 (main.cpp) construct a Ring<int> r containing the elements {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Line 15,

```
cout << "\nr contains " << r.end() - r.begin() << " elements:\n";
```

which is equivalent to

```
cout << "\nr contains " << r.end().operator-(r.begin()) << " elements:\n";
```

generates the output

```
r contains 10 elements:
```

The loop in lines 19-20 traverses r by applying the array subscript operator to a const\_iterator i, generating the output

```
0 1 2 3 4 5 6 7 8 9
```

Lines 23-34,

```
cout << "The 3rd element is " << *(i + 2) << endl;
cout << "The 4th element is " << *(3 + i) << endl;
```

which are equivalent to

```
cout << "The 3rd element is " << * (i.operator+(2)) << endl;
cout << "The 4th element is " << * (operator+(3, i)) << endl;
```

generate the output

```
The 3rd element is 2
The 4th element is 3
```

The RingIter class will be reused with the MultiRing class, developed in the next section.

## 12.5: Copy, Assignment, Insert, and Erase

*Source files and folders*

*Ring/5*  
*Ring/common/memberFunctions\_5.h*

*Chapter outline*

- Copy constructor / Assignment operator
- insert / erase

The copy constructor (*Ring.h*, line 30) is nearly identical to that of *Vector* (Chapter 9.2) (*memberFunctions\_5.h*, lines 5-21):

```
template <class T>
Ring<T>::Ring(const Ring& source):
    _head(nullptr),
    _tail(nullptr),
    _size(0),
    _capacity(source._capacity)
{
    _block = _alloc.allocate(_capacity);
    _endOfBlock = _block + _capacity - 1;

    for (const_iterator sourceElement = source.begin();
        sourceElement != source.end();
        ++sourceElement)
    {
        push_back(*sourceElement);
    }
}
```

Lines 7-13 construct a new, empty *Ring* with the same capacity as the source *Ring*. This allows the loop in lines 15-20 to *push\_back* each source element into the new *Ring* without reallocation.

The implementation of the assignment operator (*Ring.h*, line 51) is identical to that of *Vector* (Chapter 9.2) (*memberFunctions\_5.h*, lines 23-44).

The member function (*Ring.h*, line 53)

```
iterator insert(const_iterator insertionPoint, const T& source);
```

inserts a copy of the given source object before *insertionPoint*, then returns an iterator to the newly inserted element (*memberFunctions\_5.h*, lines 46-69). Its implementation is similar to that of *Vector* (Chapter 9.3), but is optimized around *Ring*'s ability to efficiently push an element to the front as well as the back.

Given a Ring containing the elements {a, b, c, d, e, f, g, h, i, j, k}, for example, the function first obtains midpoint, an iterator to the middle element:

```
const_iterator midpoint = begin() + size()/2;

          M           // M denotes midpoint
a   b   c   d   e   f   g   h   i   j   k
```

The function then splits into 2 branches.

Case 1: The insertion point is closer to the front than it is to the back (insertionPoint < midpoint) (lines 50-51, 53-59, 68)

Suppose that a new element Y is to be inserted before d:

```
          P       M           // insertionPoint (P) is closer to the
a   b   c   d   e   f   g   h   i   j   k   // front than it is to the back

// Calculate insertionDistance, the distance from the front to the insertion
// point
```

```
size_type insertionDistance = insertionPoint - begin();
```

```
// insertionDistance is 3
```

```
push_front(source);
iterator n = begin();

Y   a   b   c   d   e   f   g   h   i   j   k
n
```

Iteration 1:

```
swap(*n, *(n + 1));

a   Y   b   c   d   e   f   g   h   i   j   k
n

++n;

a   Y   b   c   d   e   f   g   h   i   j   k
n
```

Iteration 2:

```
swap(*n, *(n + 1));

a   b   Y   c   d   e   f   g   h   i   j   k
n

++n;
```

```
a b Y c d e f g h i j k
n
```

Iteration 3:

```
swap(*n, *(n + 1));

a b c Y d e f g h i j k
n

++n;

a b c Y d e f g h i j k
n

// n is equal to begin() + insertionDistance, so the loop terminates

// Return an iterator to the newly inserted element

return iterator(this, insertionDistance);
```

Case 2: The insertion point is closer to the back than it is to the front (lines 50-51, 60-66, 68)

Suppose that a new element Z is to be inserted before i:

```
M P // insertionPoint (P) is closer to the
a b c d e f g h i j k // back than it is to the front

// Calculate insertionDistance, the distance from the front to the insertion
// point

size_type insertionDistance = insertionPoint - begin();

// insertionDistance is 8

push_back(source);
iterator n = end() - 1;

a b c d e f g h i j k Z
n
```

Iteration 1:

```
swap(*n, *(n - 1));

a b c d e f g h i j Z k
n

--n;

a b c d e f g h i j Z k
n
```

Iteration 2:

```

swap(*n, *(n - 1));

a b c d e f g h i z j k
      n

--n;

a b c d e f g h i z j k
      n

```

Iteration 3:

```

swap(*n, *(n - 1));

a b c d e f g h z i j k
      n

--n;

a b c d e f g h z i j k
      n

// n is equal to begin() + insertionDistance, so the loop terminates

// Return an iterator to the newly inserted element

return iterator(this, insertionDistance);

```

The member function (Ring.h, line 54)

```
iterator erase(const_iterator erasurePoint);
```

removes the element at erasurePoint, then returns an iterator to the next element in the sequence (memberFunctions\_5.h, lines 71-93). Its implementation is also similar to that of Vector (Chapter 9.3), but is optimized around whether the erasurePoint is closer to the front or the back.

The function first calculates erasureDistance (the distance from the front to the erasurePoint) and obtains midpoint, an iterator to the middle element:

```
size_type erasureDistance = erasurePoint - begin();
const_iterator midpoint = begin() + size()/2;
```

Case 1: The erasure point is closer to the front than it is to the back (erasurePoint < midpoint) (lines 74-75, 77-83, 92)

```

P           M           // erasurePoint (P) is closer to the
a b c Y d e f g h i j k   // front than it is to the back

// Calculate erasureDistance, the distance from the front to the erasure
// point

```

```

size_type erasureDistance = erasurePoint - begin();

// erasureDistance is 3

iterator n = begin() + erasureDistance;

a b c Y d e f g h i j k
      n

Iteration 1:

*n = *(n - 1);

a b c c d e f g h i j k
      n

--n;

a b c c d e f g h i j k
      n

Iteration 2:

*n = *(n - 1);

a b b c d e f g h i j k
      n

--n;

a b b c d e f g h i j k
      n

Iteration 3:

*n = *(n - 1);

a a b c d e f g h i j k
      n

--n;

a a b c d e f g h i j k
      n

// n is equal to begin(), so the loop terminates

pop_front();

a b c d e f g h i j k

// Return an iterator to the next element in the sequence (d)

return iterator(this, erasureDistance);

```

Case 2: The erasure point is closer to the back than it is to the front (lines 74-75, 84-90, 92)

```

        M      P          // erasurePoint (P) is closer to the
a b c d e f g h Z i j k  // front than it is to the back

```

```

// Calculate erasureDistance, the distance from the front to the erasure
// point

```

```
size_type erasureDistance = erasurePoint - begin();
```

```
// erasureDistance is 8
```

```
iterator n = begin() + erasureDistance;
```

```

a b c d e f g h Z i j k
        n

```

Iteration 1:

```
*n = *(n + 1);
```

```

a b c d e f g h i i j k
        n

```

```
++n;
```

```

a b c d e f g h i i j k
        n

```

Iteration 2:

```
*n = *(n + 1);
```

```

a b c d e f g h i j j k
        n

```

```
++n;
```

```

a b c d e f g h i j j k
        n

```

Iteration 3:

```
*n = *(n + 1);
```

```

a b c d e f g h i j k k
        n

```

```
++n;
```

```

a b c d e f g h i j k k
        n

```

```
// n is equal to end() - 1, so the loop terminates
```

```

pop_back();

a b c d e f g h i j k

// Return an iterator to the next element in the sequence (i)

return iterator(this, erasureDistance);

```

The program in this chapter demonstrates the insert and erase member functions. Lines 14-26 (main.cpp) construct a Ring<Traceable<char>> x containing the elements {a, b, c, d, e, f, g, h, i, j, k}.

Lines 29-31 insert the element Y before d, generating the output

```

v Y           // Construct argument passed to insert
c Y           // push_front(source);
c Y           // swap(*n, *(n + 1));
    Y -> a
    a -> Y
~ Y
c Y           // swap(*n, *(n + 1));
    Y -> b
    b -> Y
~ Y
c Y           // swap(*n, *(n + 1));
    Y -> c
    c -> Y
~ Y
~ Y           // Destroy argument passed to insert

Inserted element Y:
a b c Y d e f g h i j k

```

Lines 34-36 insert the element Z before i, generating the output

```

v Z           // Construct argument passed to erase
c Z           // push_back(source);
c Z           // swap(*n, *(n - 1));
    Z -> k
    k -> Z
~ Z
c Z           // swap(*n, *(n - 1));
    Z -> j
    j -> Z
~ Z
c Z           // swap(*n, *(n - 1));
    Z -> i
    i -> Z
~ Z
~ Z           // Destroy argument passed to erase

Inserted element Z:
a b c Y d e f g h Z i j k

```

Lines 39-43 erase element Y, generating the output

```
Erasing element Y...
Y -> c           // *n = *(n - 1);
c -> b           // *n = *(n - 1);
b -> a           // *n = *(n - 1);
~ a              // pop_front();
```

```
Proceeding element is d:
a b c d e f g h Z i j k
```

Lines 46-50 erase element Z, generating the output

```
Erasing element Z...
Z -> i           // *n = *(n + 1);
i -> j           // *n = *(n + 1);
j -> k           // *n = *(n + 1);
~ k              // pop_back();
```

```
Proceeding element is i:
a b c d e f g h i j k
```

When main terminates, x's destructor destroys the remaining elements, generating the output

```
~ a
~ b
~ c
~ d
~ e
~ f
~ g
~ h
~ i
~ j
~ k
```

# Part 13: Multi-Block Double-Ended Queues

## 13.1: Introducing the MultiRing Class Template

*Source files and folders*

*MultiRing/I  
MultiRing/common/memberFunctions\_I.h*

*Chapter outline*

- Default constructor / destructor
- Accessor functions (empty / size, front / back)
- push\_back

Recall from Chapter 12.1 that the time cost of reallocating a Ring is directly proportional to the size of the Ring. Reallocations therefore become more and more expensive as a Ring expands.

This chapter introduces the MultiRing class template, which retains the key features of a Ring (efficient push / pop at both ends, fast random element access) while greatly reducing the cost of reallocation.

Internally, a MultiRing<T> (MultiRing containing elements of type T) is a Ring<Ring<T>\*> (Ring of pointers to Ring<T>). Each pointer refers to a dynamically-allocated Ring<T>, or “page” of elements, and each page has the same capacity.

Consider, for example, a full Ring<char> x and a full MultiRing<char> y, containing the elements {a, b, c, d, e, f, g, h, i}:

```
// Ring<char> x
| a | b | c | d | e | f | g | h | i |
// x.size() is 9
// x.capacity() is 9

// MultiRing<char> y
| _pages[0] | _pages[1] | _pages[2] |
// y.size() is 9
```

y contains a data member, `_pages`, of type Ring<Ring<char>\*> (Ring of pointers to Ring<char>):

```

.pages.size() is 3

._pages[0] is a pointer (Ring<char>*) to the Ring<char> containing {a,b,c}
._pages[1] is a pointer (Ring<char>*) to the Ring<char> containing {d,e,f}
._pages[2] is a pointer (Ring<char>*) to the Ring<char> containing {g,h,i}

._pages.front() (_pages[0]), is a pointer to the "front page"
._pages.back() (_pages[2]), is a pointer to the "back page"

```

For x, pushing back a new element j involves a reallocation of all 9 existing elements. For y, however, the same operation simply involves the creation of a new page. The new element can then be pushed back into the new page without affecting any of the other elements:

```

| a | b | c |       | d | e | f |       | g | h | i |       | j |   |   |
| _pages[0]     |   _pages[1]     |   _pages[2]     |   _pages[3]     |

```

Note that if `_pages` is full, creating a new page involves a reallocation of the pointers, but the elements themselves are never reallocated. In the above example, the worst-case scenario involves reallocating the 3 pointers in `_pages`, as opposed to all 9 elements.

The page capacity can be arbitrarily large, as long as it is kept constant for each page. Consider, for example, a Ring x containing 300 elements:

```

| x[000]-x[299] |

// x.size() is 300
// Elements x[000]-x[299] are stored in a single block of memory

```

The same elements can be stored in a MultiRing y, distributed over 3 pages of 100 elements each:

```

| y[000]-y[099] |       | y[100]-y[199] |       | y[200]-y[299] |
| _pages[0]     |       _pages[1]     |       _pages[2]     |

```

// y.size() is 300  
// \_pages.size() is 3

// \_pages[0] is a pointer to the Ring containing elements y[000]-y[099]  
// \_pages[1] is a pointer to the Ring containing elements y[100]-y[199]  
// \_pages[2] is a pointer to the Ring containing elements y[200]-y[299]

The worst-case scenario for pushing an element into x involves the reallocation of all 300 elements. The worst-case scenario for pushing an element into y, however, simply involves the creation of a new page and the reallocation of 3 pointers.

The MultiRing class is defined in the header file MultiRing.h (lines 9-45). The template parameter T (line 9) represents the type of elements stored in the MultiRing. The member types `const_pointer`, `const_reference`, `pointer`, `reference`, `difference_type`, `size_type`, and `value_type` (lines 13-19) are identical to those of Ring.

```
typedef Ring<T> Page;
```

declares Page as an alias of the type Ring<T> (a Page is a Ring of elements of type T). The data members are (lines 42-44)

```
Ring<Page*> _pages;           // Ring of pointers to Pages
size_type _size;              // Number of elements currently stored
Allocator<Page> _alloc;      // Handles the creation / destruction of Pages
```

Line 40,

```
static const size_type _pageCapacity = 5;
```

declares the static data member \_pageCapacity, of type const size\_type (const unsigned int), with a value of 5. The “static” keyword indicates that \_pageCapacity is shared among all MultiRings of a given type. All MultiRing<int> objects, for example, share a single \_pageCapacity variable, while all MultiRing<double> objects share another \_pageCapacity variable.

The default constructor (line 21) simply initializes \_size to 0 (memberFunctions\_1.h, lines 3-8). \_pages and \_alloc are implicitly initialized via their own default constructors.

The member functions empty and size (MultiRing.h, lines 24-25) are identical to those of Ring (memberFunctions\_1.h, lines 16-26).

The member functions front and back (MultiRing.h, lines 26-27, 29-30) return references to the front and back elements (memberFunctions\_1.h, lines 28-50):

```
template <class T>
inline typename MultiRing<T>::const_reference MultiRing<T>::front() const
{
    return _pages.front()->front();
}

template <class T>
inline typename MultiRing<T>::const_reference MultiRing<T>::back() const
{
    return _pages.back()->back();
}

template <class T>
inline typename MultiRing<T>::reference MultiRing<T>::front()
{
    return _pages.front()->front();
}

template <class T>
inline typename MultiRing<T>::reference MultiRing<T>::back()
{
    return _pages.back()->back();
}
```

## The expressions

```
_pages.front()
(pages.back())
```

return pointers to the front / back pages (i.e. the front / back Ring<T>). The expression

```
_pages.front() ->front()
```

therefore returns (a reference to) the front element of the front page, while the expression

```
_pages.back() ->back()
```

returns (a reference to) the back element of the back page.

## The private member function (MultiRing.h, line 36)

```
Page* _createPage();
```

creates a new Page (Ring<T>) of capacity `_pageCapacity`, then returns a pointer to the new Page (memberFunctions\_1.h, lines 62-71):

```
template <class T>
typename MultiRing<T>::Page* MultiRing<T>::_createPage()
{
    Page* newPassword = _alloc.allocate(1); // Allocate memory for 1 Page
    _alloc.construct(newPage, Page()); // Construct the Page

    newPassword->reserve(_pageCapacity); // Prevents the need to reallocate
                                         // elements within the Page
    return newPassword; // Return a pointer to the new Page
}
```

## The member function (MultiRing.h, line 31)

```
void push_back(const T& source);
```

inserts a copy of the given source object at the end of the MultiRing. The procedure is

```
If the MultiRing is empty or the back page is full,
create a new back page;
```

```
Push back the new element into the back page
Update the size;
```

and the function definition is (memberFunctions\_1.h, lines 52-60)

```
template <class T>
void MultiRing<T>::push_back(const T& source)
{
    if (empty() || _pages.back()->size() == _pages.back()->capacity())
        _pages.push_back(_createPage());

    _pages.back()->push_back(source);
    ++_size;
}
```

The expression

```
_pages.back()->size() == _pages.back()->capacity()
```

determines whether or not the back page is full. In the statement

```
_pages.push_back(_createPage());
```

`_createPage` creates a new page and returns a pointer to it. The pointer is then pushed back into `_pages`, making the newly created page the back page.

The private member function (MultiRing.h, line 37)

```
void _destroyPage(Page* p);
```

destroys the given Page and releases its memory (memberFunctions\_1.h, lines 73-78):

```
template <class T>
inline void MultiRing<T>::_destroyPage(Page* p)
{
    _alloc.destroy(p);
    _alloc.deallocate(p);
}
```

The statement

```
_alloc.destroy(p);
```

destroys the page by simply calling its destructor, i.e. the destructor of the Ring class. Recall that this destroys all of the elements, releases the memory block used to hold the elements, then destroys the data members (`_block`, `_endOfBlock`, `_head`, `_tail_size`, `_capacity`, `_alloc`). The statement

```
_alloc.deallocate(p);
```

then releases the memory that was used to hold the page.

The private member function (MultiRing.h, line 38)

```
void _destroyAllPages();
```

traverses the pointers in `_pages`, destroying each Page (memberFunctions\_1.h, lines 80-85):

```
template <class T>
void MultiRing<T>::_destroyAllPages()
{
    for (Ring<Page*>::iterator p = _pages.begin(); p != _pages.end(); ++p)
        _destroyPage(*p);
}
```

`p` is a `Ring<Page*>::iterator` (an iterator pointing to a `Page*`), so the expression

```
*p      // Dereference the iterator p
```

returns a `Page*` (pointer to a Page), which is then passed to `_destroyPage`.

The destructor (MultiRing.h, line 22) simply calls `_destroyAllPages` (memberFunctions\_1.h, lines 10-14). After calling `_destroyAllPages`, the destructor automatically destroys `_alloc`, `_size`, and `_pages`.

Line 11 (main.cpp) constructs an empty `MultiRing<Traceable<int>>` `m`. The loop in lines 13-14 then pushes back the elements `{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}`, generating the output

```
v 0      |      _pages[0]      |      // Construct argument passed to push_back
c 0      | 0  |                          // Destroy argument passed to push_back
~ 0

v 1      |      _pages[0]      |
c 1      | 0  | 1  |
~ 1

v 2      |      _pages[0]      |
c 2      | 0  | 1  | 2  |
~ 2

v 3      |      _pages[0]      |
c 3      | 0  | 1  | 2  | 3  |
~ 3

v 4      |      _pages[0]      |
c 4      | 0  | 1  | 2  | 3  | 4  |
~ 4

v 5      |      _pages[0]      |      _pages[1]      |
c 5      | 0  | 1  | 2  | 3  | 4  | 5  |
~ 5

v 6      |      _pages[0]      |      _pages[1]      |
c 6      | 0  | 1  | 2  | 3  | 4  | 5  | 6  |
~ 6

v 7      |      _pages[0]      |      _pages[1]      |
c 7      | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
~ 7
```

```

v 8      |     _pages[0]      |     _pages[1]      |
c 8      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
~ 8

v 9      |     _pages[0]      |     _pages[1]      |
c 9      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
~ 9

v 10     |     _pages[0]      |     _pages[1]      |     _pages[2]      |
c 10     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
~ 10

v 11     |     _pages[0]      |     _pages[1]      |     _pages[2]      |
c 11     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
~ 11

```

When main terminates, m is destroyed, generating the output

```

|     _pages[0]      |     _pages[1]      |     _pages[2]      |
~ 0     | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
~ 1     |  | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
~ 2     |  |  | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
~ 3     |  |  |  | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
~ 4     |  |  |  |  | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
~ 5     |  |  |  |  | 6 | 7 | 8 | 9 | 10 | 11 |
~ 6     |  |  |  |  |  | 7 | 8 | 9 | 10 | 11 |
~ 7     |  |  |  |  |  |  | 8 | 9 | 10 | 11 |
~ 8     |  |  |  |  |  |  |  | 9 | 10 | 11 |
~ 9     |  |  |  |  |  |  |  |  | 10 | 11 |
~ 10    |  |  |  |  |  |  |  |  |  | 11 |
~ 11

```



## 13.2: Push Front and Array Subscript

*Source files and folders*

*MultiRing/2  
MultiRing/common/memberFunctions\_2.h*

*Chapter outline*

- *push\_front*
- *operator[]*

The member function (MultiRing.h, line 33)

```
void push_front(const T& source);
```

is the mirror image of *push\_back*. The procedure is

```
If the MultiRing is empty or the front page is full,  
create a new front page;
```

```
Push front the new element into the front page  
Update the size;
```

and the function definition is (memberFunctions\_2.h, lines 17-25)

```
template <class T>  
void MultiRing<T>::push_front(const T& source)  
{  
    if (empty() || _pages.front()->size() == _pages.front()->capacity())  
        _pages.push_front(_createPage());  
  
    _pages.front()->push_front(source);  
    ++_size;  
}
```

The private member function (MultiRing.h, line 39)

```
reference _element(size_type index) const;
```

returns a reference to the element at the given index, like that of *Ring* (Chapter 12.2). Its implementation, however, is slightly more complex (memberFunctions\_2.h, lines 27-42). Consider, for example, a *MultiRing<char>* containing 16 elements (the letters a through p), distributed over 4 pages:

```

Element      a   b   c   d   e   f   g   h   i   j   k   l   m   n   o   p
index        0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15
pageNumber   0   0   0   1   1   1   1   2   2   2   2   2   2   3   3   3
cellNumber  0   1   2   0   1   2   3   4   0   1   2   3   4   0   1   2

// index denotes the index number of a given element, relative to all of
// the elements in the MultiRing (0-15)

// _pages[0] is a pointer to the Page containing {a,b,c}
// _pages[1] is a pointer to the Page containing {d,e,f,g,h}
// _pages[2] is a pointer to the Page containing {i,j,k,l,m}
// _pages[3] is a pointer to the Page containing {n,o,p}

// pageNumber denotes the number of the page in which the element is stored
// (0, 1, 2, 3)

// cellNumber denotes the index number of a given element, relative to its
// page (element f is located at index 2 of page 1, element n is located at
// index 0 of page 3, etc.)

```

The function consists of 2 branches.

Case 1: The given index refers to an element in the front page ( $\text{index} < \text{_pages.front()}->\text{size}()$ ) (lines 30-33)

```

Element      a   b   c
index        0   1   2

```

In this case, the function can simply obtain the element from the front page using the given index:

```
return (*_pages.front())[index]; // return (*_pages[0])[index];
```

The expression

```
(*_pages.front()) // Dereference the pointer _pages.front()
```

returns a reference to the front Page. The desired element is then obtained via the array subscript operator.

Case 2: The given index refers to an element beyond the front page ( $\text{index} \geq \text{_pages.front()}->\text{size}()$ ) (lines 34-41)

```

// Determine the pageNumber and cellNumber of the desired element

size_type offset = index - _pages.front()>>size();
size_type pageNumber = 1 + offset/_pageCapacity;
size_type cellNumber = offset % _pageCapacity;

```

Element	d	e	f	g	h	i	j	k	l	m	n	o	p
index	3	4	5	6	7	8	9	10	11	12	13	14	15
offset	0	1	2	3	4	5	6	7	8	9	10	11	12
pageNumber	1	1	1	1	1	2	2	2	2	3	3	3	3
cellNumber	0	1	2	3	4	0	1	2	3	4	0	1	2

```
// Obtain the element via its pageNumber and cellNumber
return (*_pages[pageNumber])[cellNumber];
```

To retrieve element 15 (p), for example:

```
offset = index - _pages.front()->size();
= 15 - 3;
= 12;

pageNumber = 1 + offset/_pageCapacity;
= 1 + 12/5;
= 1 + 2;           // The decimal portion of 12/5 (2.4) is truncated
= 3;

cellNumber = offset % _pageCapacity;
= 12 % 5;
= 2;

return (*_pages[pageNumber])[cellNumber];    // return (*_pages[3])[2]
                                            // (page 3, cell 2)
```

Note that in the expression (line 37)

```
offset/_pageCapacity
```

the decimal portion of the result is automatically truncated because offset and \_pageCapacity are of type size\_type (unsigned int).

The array subscript operator (MultiRing.h, lines 28, 32) can then be defined in terms of the \_element function (memberFunctions\_2.h, lines 3-15), as it is in the Ring class (Chapter 12.2).

Lines 13-22 (main.cpp) construct a MultiRing<Traceable<char>> m containing the elements {a, b, c, d, e, f, g, h}, generating the output

```
v h      |      _pages[0]      |
c h      | h | 
~ h

v g      |      _pages[0]      |
c g      | g | h |
~ g

v f      |      _pages[0]      |
c f      | f | g | h |
```

```

~ f

v e      |      _pages[0]      |
c e      | e | f | g | h |
~ e

v d      |      _pages[0]      |
c d      | d | e | f | g | h |
~ d

v c      |      _pages[0]      |      _pages[1]      |
c c      |          | c | d | e | f | g | h |
~ c

v b      |      _pages[0]      |      _pages[1]      |
c b      |          | b | c | d | e | f | g | h |
~ b

v a      |      _pages[0]      |      _pages[1]      |
c a      |          | a | b | c | d | e | f | g | h |
~ a

```

The layout of m is

Element index	a	b	c
0	1	2	

Element index	d	e	f	g	h
3	4	5	6	7	

offset	0	1	2	3	4
pageNumber	1	1	1	1	1

cellNumber	0	1	2	3	4
------------	---	---	---	---	---

The loop in lines 25-26 prints the elements [m[0], m[8]) via the array subscript operator, generating the output

a b c d e f g h

When main terminates, m is destroyed, generating the output

```

|      _pages[0]      |      _pages[1]      |
~ a      |          | b | c | d | e | f | g | h |
~ b      |          | | c | d | e | f | g | h |
~ c          | d | e | f | g | h |
~ d          | | e | f | g | h |
~ e          | | | f | g | h |
~ f          | | | | g | h |
~ g          | | | | | h |
~ h

```

### 13.3: Iterators, Pop Front, Pop Back, and Clear

*Source files and folders*

*MultiRing/3  
MultiRing/common/memberFunctions\_3.h*

*Chapter outline*

- *begin / end / rbegin / rend*
- *pop\_front / pop\_back / clear*

Recall from Chapter 12.4 that `Ring<T>::iterator` was implemented using the `RingIter` class template:

Type	Alias of
<code>Ring&lt;T&gt;::iterator</code>	<code>RingIter&lt;Ring&lt;T&gt;&gt;</code>
<code>Ring&lt;T&gt;::const_iterator</code>	<code>ConstIter&lt;Ring&lt;T&gt;::iterator&gt;</code>
<code>Ring&lt;T&gt;::reverse_iterator</code>	<code>ReverseIter&lt;Ring&lt;T&gt;::iterator&gt;</code>
<code>Ring&lt;T&gt;::const_reverse_iterator</code>	<code>ReverseIter&lt;Ring&lt;T&gt;::const_iterator&gt;</code>

Note, however, that `RingIter` is also compatible with any container class C that:

- Defines the member types `pointer`, `reference`, `difference_type`, and `value_type`
- Defines a member function `_element`, taking a single `int` argument and returning a `C::reference`
- Declares `RingIter` as a friend class (if C's `_element` function is private)

`MultiRing` fulfills these requirements, so its iterators can also be implemented in terms of `RingIter`:

Type	Alias of
<code>MultiRing&lt;T&gt;::iterator</code>	<code>RingIter&lt;MultiRing&lt;T&gt;&gt;</code>
<code>MultiRing&lt;T&gt;::const_iterator</code>	<code>ConstIter&lt;MultiRing&lt;T&gt;::iterator&gt;</code>
<code>MultiRing&lt;T&gt;::reverse_iterator</code>	<code>ReverseIter&lt;MultiRing&lt;T&gt;::iterator&gt;</code>
<code>MultiRing&lt;T&gt;::const_reverse_iterator</code>	<code>ReverseIter&lt;MultiRing&lt;T&gt;::const_iterator&gt;</code>

Line 15 (`MultiRing.h`) declares `RingIter` as a friend of the `MultiRing` class. Lines 17-18 and 21-22 then declare the above aliases.

The implementation of `begin`, `end`, `rbegin`, and `rend` (lines 34-37, 42-45) is identical to `Ring`; the only difference is that the `const_cast` converts the “this” pointer to a `MultiRing*` instead of a `Ring*` (`memberFunctions_3.h`, lines 3-50).

The member functions (`MultiRing.h`, lines 51-52)

```
void pop_front();
void pop_back();
```

are mirror images of each other. Their respective procedures are

```

pop_front
{
    Pop the front element off of the front page;

    If the front page is empty,
        remove the front page;

    Update the size;
}

pop_back
{
    Pop the back element off of the back page;

    If the back page is empty,
        remove the back page;

    Update the size;
}

```

and their function definitions are (memberFunctions\_3.h, lines 52-78)

```

template <class T>
void MultiRing<T>::pop_front()
{
    _pages.front()->pop_front();

    if (_pages.front()->empty())
    {
        _destroyPage(_pages.front());           // Destroy the front page
        _pages.pop_front();                   // Remove the pointer to the front page
    }

    --_size;
}

template <class T>
void MultiRing<T>::pop_back()
{
    _pages.back()->pop_back();

    if (_pages.back()->empty())
    {
        _destroyPage(_pages.back());           // Destroy the back page
        _pages.pop_back();                   // Remove the pointer to the back page
    }

    --_size;
}

```

The member function clear (MultiRing.h, line 53) simply destroys all the pages, removes the pointers

to those pages, and resets the size to 0 (memberFunctions\_3.h, lines 80-86).

Lines 14-21 (main.cpp) construct two MultiRing<Traceable<int>>, x and y, containing the elements

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}      // x
{11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0}      // y
```

Lines 23-27 demonstrate MultiRing<T>::const\_iterator by calling printContainer on x and y.

The loop in lines 31-32 calls pop\_back on x until it is empty, generating the output

```
~ 11 |   _pages[0]   |   _pages[1]   |   _pages[2]   |
~ 10 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |   |
~ 9  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |   |
~ 8  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |   |
~ 7  | 0 | 1 | 2 | 3 | 4 | 5 | 6 |   |
~ 6  | 0 | 1 | 2 | 3 | 4 | 5 |   |
~ 5  | 0 | 1 | 2 | 3 | 4 |
~ 4  | 0 | 1 | 2 | 3 |
~ 3  | 0 | 1 | 2 |
~ 2  | 0 | 1 |
~ 1  | 0 |   |
~ 0
```

Lines 36-37 then call pop\_front on y until it is empty, generating the output

```
~ 11 |   _pages[0]   |   _pages[1]   |   _pages[2]   |
~ 10 |   | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
~ 9  |   |   | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
~ 8  |   |   |   | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
~ 7  |   |   |   |   | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
~ 6  |   |   |   |   |   | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
~ 5  |   |   |   |   |   |   | 4 | 3 | 2 | 1 | 0 |
~ 4  |   |   |   |   |   |   |   | 3 | 2 | 1 | 0 |
~ 3  |   |   |   |   |   |   |   |   | 2 | 1 | 0 |
~ 2  |   |   |   |   |   |   |   |   |   | 1 | 0 |
~ 1  |   |   |   |   |   |   |   |   |   |   | 0 |
~ 0
```



## 13.4: Copy, Assignment, Insert, and Erase

*Source files and folders*

*MultiRing/4  
MultiRing/common/memberFunctions\_4.h*

*Chapter outline*

- Copy constructor / assignment operator
- insert / erase

The copy constructor (MultiRing.h, line 30) is nearly identical to that of Ring (memberFunctions\_4.h, lines 3-15):

```
template <class T>
MultiRing<T>::MultiRing(const MultiRing& source):
    _size(0)
{
    _pages.reserve(source._pages.capacity());

    for (const_iterator sourceElement = source.begin();
        sourceElement != source.end();
        ++sourceElement)
    {
        push_back(*sourceElement);
    }
}
```

The new container's size is initialized to 0 (line 5), and each element in the source container is pushed back into the new container (lines 9-14). Line 7,

```
_pages.reserve(source._pages.capacity());
```

ensures that all of the source elements can be pushed back without reallocating the pointers in `_pages`.

The remaining member functions, `operator=`, `insert`, and `erase` (MultiRing.h, lines 50-52) are identical to those of Ring (memberFunctions\_4.h, lines 17-88).



# Part 14: Unbalanced Binary Trees

## 14.1: Key-Mapped Pairs

*Source folder: Pair*

*Chapter outline*

- Storing pairs of key / mapped values

A binary tree is a type of data structure in which the elements are automatically sorted. Each element is an object containing a “key value” and a “mapped value.” The elements are sorted and accessed by their key values. The mapped value of a given element is the data associated with (i.e. mapped to) the corresponding key. A binary tree, for example, could contain a set of name-age pairs, where each element contains a person's name (the key value) and their age (the mapped value).

This chapter introduces the Pair class template, which will be used to store pairs of key / mapped values. An object of type `Pair<A, B>` simply contains two public data members: first (an object of type A) and second (an object of type B). The Pair class is defined in the header file `Pair.h` (lines 12-26):

```
template <class A, class B>                                // An A-B pair
struct Pair
{
    typedef A first_type;
    typedef B second_type;

    Pair();                                         // Default constructor
    Pair(const A& _first, const B& _second);      // Value constructor

    template <class X, class Y>                      // Template constructor
    Pair(const Pair<X, Y>& source);

    A first;                                         // Key value
    B second;                                        // Mapped value
};
```

`first_type` and `second_type` (lines 15-16) are aliases of the types A and B respectively. Suppose that the aforementioned name-age pairs are represented using `Pair<std::string, int>`. The compiler would generate the code

```

struct Pair<std::string, int>                                // A string-int pair
{
    typedef std::string first_type;
    typedef int second_type;

    Pair<std::string, int>();                                // Default constructor

    Pair<std::string, int>(const std::string& _first,        // Value constructor
                           const int& _second);

    template <class X, class Y>                                // Template
    Pair<std::string, int>(const Pair<X, Y>& source);      // constructor

    std::string first;                                         // Key value
    int second;                                               // Mapped value
};

```

The default constructor (line 18) simply constructs first / second as copies of a default-constructed A / B (lines 34-40).

The value constructor (line 19) constructs first and second as copies of the given parameters (lines 42-48).

The template constructor (lines 21-22) constructs a Pair<A, B> from a Pair<X, Y> (lines 50-57):

```

template <class A, class B>
template <class X, class Y>
inline Pair<A, B>::Pair(const Pair<X, Y>& source):
    first(source.first),
    second(source.second)
{
    // ...
}

```

A Pair<A, B> can be constructed from a Pair<X, Y> as long as an object of type A can be constructed from an object of type X, and an object of type B can be constructed from an object of type Y:

```

// q.first / q.second (doubles) are constructed from p.first / p.second
// (ints)

Pair<int, int> p;
Pair<double, double> q = p;

// s.first (const string) is constructed from r.first (string)
// s.second (Traceable<int>) is constructed from r.second (int)

Pair<string, int> r;
Pair<const string, Traceable<int>> s = r;

```

The nonmember function makePair (lines 9-10) returns a Pair<A, B> in which first / second are initialized to the given values (lines 28-32). This function provides a shorthand alternative to using the

value constructor. Given a `Pair<int, double>` p, for example the statement

```
p = Pair<int, double>(7, 0.618);      // Template arguments (int, double) must
                                         // be supplied when using the value
                                         // constructor
```

can be written as

```
p = makePair(7, 0.618);                // p = makePair<int, double>(7, 0.618);
                                         // Template argument deduction allows
                                         // template arguments to be omitted
```

The header file `PairIo.h` (“Pair with I/O (input/output)”) overloads the stream insertion / extraction operators for the `Pair` class.

The stream insertion operator (lines 10-11, 16-22) generates the output

```
(<first>, <second>)
```

where `<first>` and `<second>` are the values of first and second.

The stream extraction operator (lines 13-14, 24-30) writes the input values to the right operand's data members (first and second).

The program in this chapter demonstrates the `Pair` class. Lines 13-15 (`main.cpp`) declare `Record` as an alias of the type `Pair<FirstName, Age>`, or `Pair<string, int>`. Line 17 constructs a `Vector<Record>` `v`, and the loop in lines 19-27 performs 3 iterations. Each iteration prompts the user for a first name and age, writes the values to a `Record` `r`, then pushes it back into `v`. Lines 29-30 then print the contents of `v`. A sample run of the program is

```
Enter first name and age (separated by whitespace): Goichi 47
Enter first name and age (separated by whitespace): Shinji 49
Enter first name and age (separated by whitespace): Hideo 51

You entered:
(Goichi,47) (Shinji,49) (Hideo,51)
```



## 14.2: Nodes

## *Source files and folders*

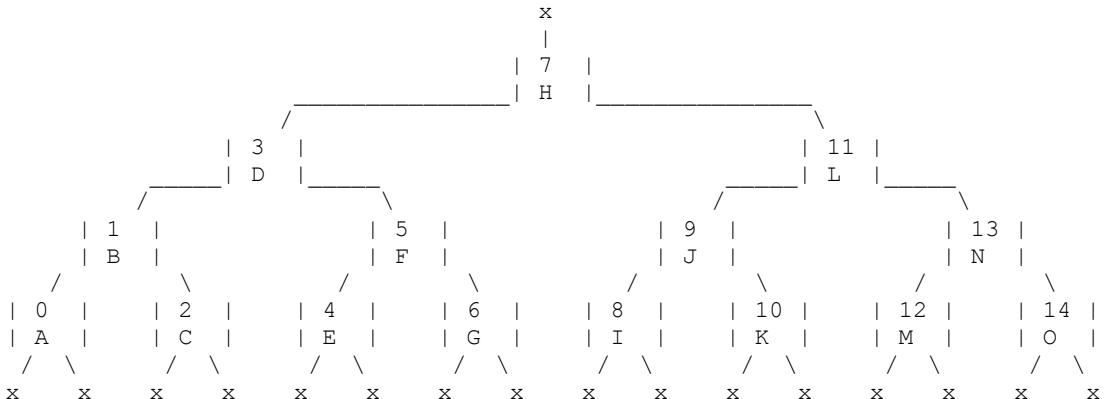
*BinaryTreeNode*  
*bt/key.h*  
*bt/relatives.h*

## *Chapter outline*

- The `BinaryTreeNode` class
  - Implementing functions to conveniently access the relatives and key value of a given node

The following diagram illustrates a binary tree containing 15 elements of type `Pair<const int, char>`:

// x denotes null

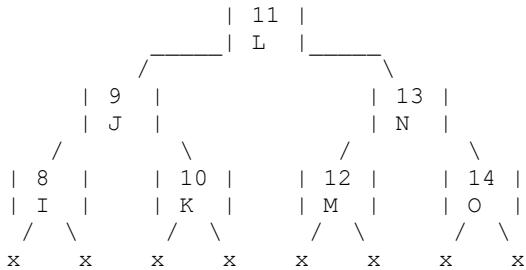


The tree consists of 15 nodes. Each node contains a single element (Pair<const int, char>) as well as links (pointers) to its left and right "child nodes." The node containing the Pair (7,H), for example, has two children: its "left child" is the node containing the Pair (3,D), and its "right child" is the node containing the Pair (11,L). Similarly, the node containing (9,J) has two children: its left link points to the node containing (8,I), and its right link points to the node containing (10,K). The node containing (2,C) has no children: its left and right links (pointers) are both null. The nodes containing (0,A), (2,C), (4,E), (6,G), (8,I), (10,K), (12,M), and (14,O) are called "leaf nodes" because they have no children.

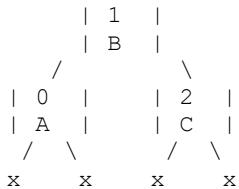
Each node also contains a link to its parent node. The parent link of the node containing (4,E), for example, points to the node containing (5,F). Similarly, the parent link of the node containing (9,J) points to the node containing (11,L). The node containing (7,H) is called the "root node" because it has no parent (its parent link is null).

Given a node  $n$ ,  $n$ 's "left subtree" is the tree rooted at  $n$ 's left child, and  $n$ 's "right subtree" is the tree

rooted at n's right child. Node (7,H)'s right subtree, for example, is



Similarly, node (3,D)'s left subtree is



and node (5,F)'s right subtree is



The "levels" of a tree are numbered from the top down, beginning at 0. The maximum number of nodes that can occupy a given level is  $2^{\text{level}}$ . In the above tree, each level is full:

Level 0 contains 1 ( $2^0$ ) node: (7,H)

Level 1 contains 2 ( $2^1$ ) nodes: (3,D), (11,L)

Level 2 contains 4 ( $2^2$ ) nodes: (1,B), (5,F), (9,J), (13,N)

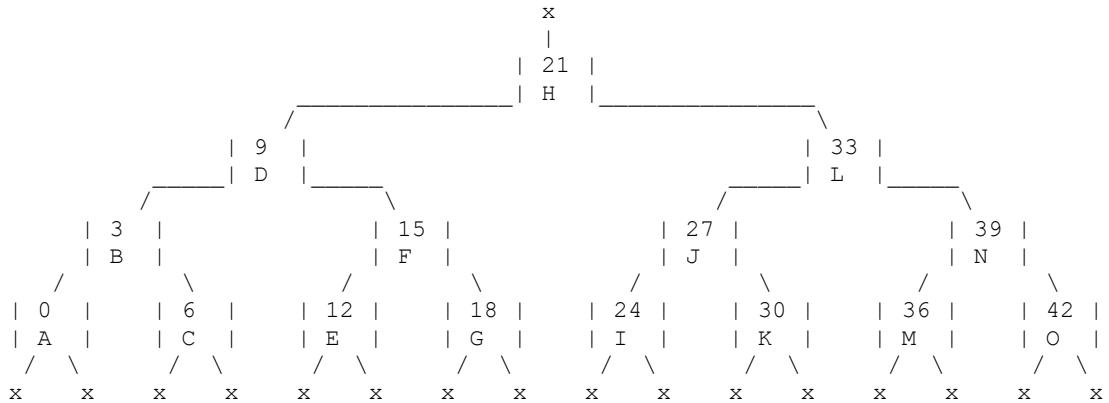
Level 3 contains 8 ( $2^3$ ) nodes: (0,A), (2,C), (4,E), (6,G), (8,I), (10,K), (12,M), (14,O)

The "height" of the tree, or number of levels, is 4.

No two elements in a tree can have the same key value (duplicate keys are not allowed).

Recall from Chapter 7.2 that the Less class is a predicate (function object) used to determine whether one value is less than another. The above tree is sorted using a predicate of type `Less<int>`: for any given node n, the key of n's left child is less than that of n, and the key of n's right child is greater than that of n.

These properties make binary trees ideal for looking up elements by key value. Given the tree



for example, any element can be located by traversing 4 nodes or less.

To retrieve the element with a key value of 12:

Begin at the root, node (21,H);

12 is less than 21, so go left, to node (9,D);

12 is greater than 9, so go right, to node (15,F);

12 is less than 15, so go left, to node (12,E);

To retrieve the element with a key value of 30:

Begin at the root, node (21,H);

30 is greater than 21, so go right, to node (33,L);

30 is less than 33, so go left, to node (27,J);

30 is greater than 27, so go right, to node (30,K);

To retrieve the element with a key value of 5:

Begin at the root, node (21,H);

5 is less than 21, so go left, to node (9,D);

5 is less than 9, so go left, to node (3,B);

5 is greater than 3, so go right, to node (6,C);

5 is less than 6, so go left, to null;

No element with a key value of 5 exists in the tree;

Arranged in ascending order of their key values, the nodes form the “in-order” sequence

0	3	6	9	12	15	18	21	24	27	30	33	36	39	42
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O

Given a node n, n's “in-order predecessor” is the previous node in the sequence, and n's “in-order successor” is the next node in the sequence. The in-order predecessor of node (21,H), for example, is

node (18,G); the in-order successor of node (21,H) is node (24,I). In addition to the parent and child links, each node also contains a link to its in-order predecessor and in-order successor:

```

 0   3   6   9   12   15   18   21   24   27   30   33   36   39   42
 A   B   C   D   E   F   G   H   I   J   K   L   M   N   O
 x <-p <-p
 s-> x

```

Node (21,H)'s predecessor link (*p*), for example, is a pointer to node (18,G); its successor link (*s*) is a pointer to node (24,I). Node (0,A)'s predecessor link and node (42,O)'s successor link are null. These links allow the tree to be traversed in-order, as if it were a sorted linked list. The first node in the sequence (the leftmost node of the tree) is called the “head,” while the last node in the sequence (the rightmost node of the tree) is called the “tail.”

The `BinaryTreeNode` class is defined in the header file `BinaryTreeNode.h` (lines 8-25). The template parameters `Key` and `Mapped` (line 8) represent the types of the element's key / mapped values. `key_type` and `mapped_type` (lines 11-12) are simply aliases of those types.

`value_type` (line 13) is an alias of the element's type, `Pair<const Key, Mapped>`. In the above diagrams, for example, each node is of type `BinaryTreeNode<int, char>`, and contains an element of type `Pair<const int, char>`. The key values must be immutable (const) in order to preserve the sorted structure of the tree.

The element is declared in line 17, followed by the links to the node's parent, children, predecessor, and successor (lines 19-24).

The constructor (line 15) constructs a node from the given `sourceElement` and initializes all of its links to null (lines 27-37).

The header file `relatives.h` contains a set of functions for obtaining information about a node's relatives and providing convenient access to those relatives. The argument passed to each of these functions (a pointer to a node) must not be null. The namespace `bt` (`relatives.h`, line 6) is short for “binary tree.”

The function (lines 8-9)

```
template <class Node>
bool isRoot(const Node* n);
```

returns true if *n* is the root of the tree (if *n* has no parent) (lines 38-42):

```
template <class Node>
inline bool isRoot(const Node* n)
{
    return n->parent == nullptr;
}
```

If *p* is a `BinaryTreeNode<int, char>*` (pointer to a `BinaryTreeNode<int, char>`), for example, the expression

```
isRoot(p)      // Is p the root of the tree?
```

is equivalent to

```
isRoot<BinaryTreeNode<int, char>>(p)
```

The compiler generates the code

```
inline bool isRoot(const BinaryTreeNode<int, char>* n)
{
    return n->parent == nullptr;
}
```

The functions (lines 14-18)

```
template <class Node>
bool isLeftChild(const Node* n);

template <class Node>
bool isRightChild(const Node* n);
```

return true if n is the left / right child of its parent. If n is the root, the functions return false (lines 50-60):

```
template <class Node>
inline bool isLeftChild(const Node* n)
{
    return !isRoot(n) && (n == n->parent->left);
}

template <class Node>
inline bool isRightChild(const Node* n)
{
    return !isRoot(n) && (n == n->parent->right);
}
```

Note that if n is the root of the tree, the expression

```
n->parent->left
```

is undefined because the root's parent link (pointer) is null. In the statement (line 53)

```
return !isRoot(n) && (n == n->parent->left);
```

the second expression,

```
(n == n->parent->left)
```

is only evaluated if the first expression,

```
!isRoot(n)
```

returns true. In other words, the statement

```
return !isRoot(n) && (n == n->parent->left);
```

is equivalent to

```
if (!isRoot(n))                                // If n is not the root,
    return n == n->parent->left;                //   return true / false depending on
else                                            //   whether n is the left child of its
    return false;                                //   parent;
                                                // Otherwise (if n is the root),
                                                //   return false;
```

Similarly, the statement (line 59)

```
return !isRoot(n) && (n == n->parent->right);
```

is equivalent to

```
if (!isRoot(n))                                // If n is the root,
    return n == n->parent->right;               //   is n the right child of its parent?
Else                                            // Otherwise (if n is the root),
    return false;                                //   return false;
```

The functions (lines 20-24)

```
template <class Node>
bool hasLeftChild(const Node* n);

template <class Node>
bool hasRightChild(const Node* n);
```

return true if n has a left / right child (lines 62-72):

```
template <class Node>
inline bool hasLeftChild(const Node* n)
{
    return n->left != nullptr;      // If n's left link does not point to null,
}                                         // return true; otherwise, return false

template <class Node>
inline bool hasRightChild(const Node* n)
{
    return n->right != nullptr;     // If n's right link does not point to null,
}                                         // return true; otherwise, return false
```

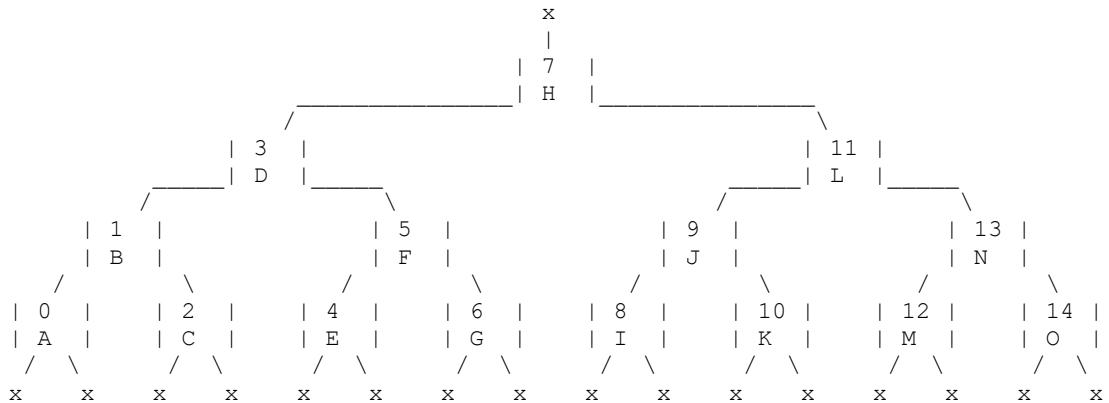
The function (lines 11-12)

```
template <class Node>
bool isLeaf(const Node* n);
```

returns true if n is a leaf (i.e. if n has no children) (lines 44-48):

```
template <class Node>
inline bool isLeaf(const Node* n)
{
    return !hasLeftChild(n) && !hasRightChild(n);
}
```

Given the tree



and a node pointer n, for example,

```
If n points to node (7,H),
isRoot(n) returns true
isLeftChild(n) returns false
isRightChild(n) returns false
hasLeftChild(n) returns true
hasRightChild(n) returns true
isLeaf(n) returns false
```

```
If n points to node (3,D),
isRoot(n) returns false
isLeftChild(n) returns true
isRightChild(n) returns false
hasLeftChild(n) returns true
hasRightChild(n) returns true
isLeaf(n) returns false
```

```
If n points to node (13,N),
isRoot(n) returns false
isLeftChild(n) returns false
isRightChild(n) returns true
hasLeftChild(n) returns true
hasRightChild(n) returns true
isLeaf(n) returns false
```

```
If n points to node (4,E),
isRoot(n) returns false
isLeftChild(n) returns true
```

```

isRightChild(n) returns false
hasLeftChild(n) returns false
hasRightChild(n) returns false
isLeaf(n) returns true

```

The functions (lines 26-30)

```

template <class Node>
bool hasPredecessor(const Node* n);

template <class Node>
bool hasSuccessor(const Node* n);

```

return true if n has a predecessor / successor (if n's predecessor / successor link is not null) (lines 74-84):

```

template <class Node>
inline bool hasPredecessor(const Node* n)
{
    return n->predecessor != nullptr;
}

template <class Node>
inline bool hasSuccessor(const Node* n)
{
    return n->successor != nullptr;
}

```

The function (lines 32-33)

```

template <class Node>
Node* leftmostNode(Node* n);

```

descends the tree leftward beginning at n, then returns a pointer to the leftmost non-null node. If n does not have a left child, the function simply returns n (lines 86-93):

```

template <class Node>
Node* leftmostNode(Node* n)
{
    while (n->left != nullptr)      // While n has a left child,
        n = n->left;                //     point n to its left child

    return n;
}

```

Similarly, the function (lines 35-36)

```

template <class Node>
Node* rightmostNode(Node* n);

```

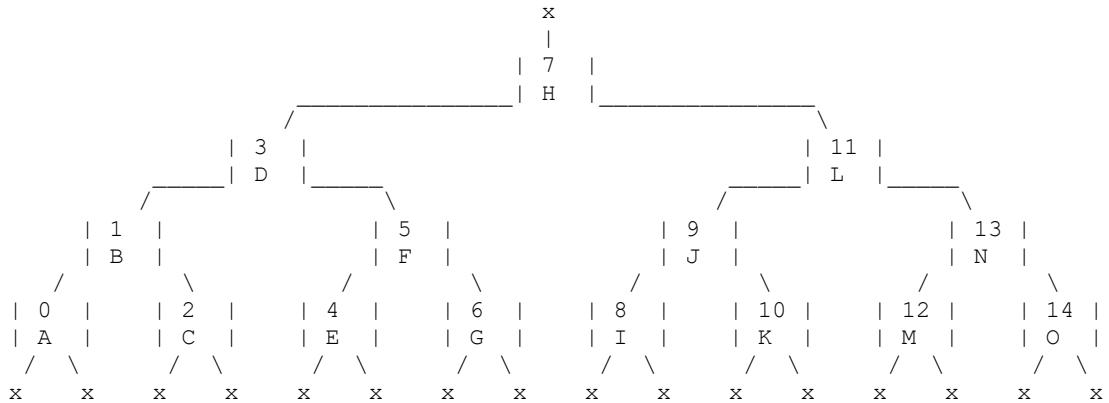
descends the tree rightward beginning at n, then returns a pointer to the rightmost non-null node. If n

does not have a right child, the function simply returns n (lines 95-102):

```
template <class Node>
Node* rightmostNode(Node* n)
{
    while (n->right != nullptr)      // While n has a right child,
        n = n->right;                //   point n to its right child

    return n;
}
```

Given the tree



and a node pointer n, for example,

```
If n points to node (7,H),
hasPredecessor(n) returns true
hasSuccessor(n) returns true
leftmostNode(n) returns a pointer to node (0,A)
rightmostNode(n) returns a pointer to node (14,O)
```

```
If n points to node (11,L),
hasPredecessor(n) returns true
hasSuccessor(n) returns true
leftmostNode(n) returns a pointer to node (8,I)
rightmostNode(n) returns a pointer to node (14,O)
```

```
If n points to node (5,F),
hasPredecessor(n) returns true
hasSuccessor(n) returns true
leftmostNode(n) returns a pointer to node (4,E)
rightmostNode(n) returns a pointer to node (6,G)
```

```
If n points to node (12,M),
hasPredecessor(n) returns true
hasSuccessor(n) returns true
leftmostNode(n) returns a pointer to node (12,M)
```

```

rightmostNode(n) returns a pointer to node (12,M)

If n points to node (0,A),
hasPredecessor(n) returns false
hasSuccessor(n) returns true
leftmostNode(n) returns a pointer to node (0,A)
rightmostNode(n) returns a pointer to node (0,A)

If n points to node (14,O),
hasPredecessor(n) returns true
hasSuccessor(n) returns false
leftmostNode(n) returns a pointer to node (14,O)
rightmostNode(n) returns a pointer to node (14,O)

```

The header file key.h contains functions for conveniently obtaining the key value of a given node or Pair. The function (key.h, lines 8-9)

```
template <class Node>
const typename Node::key_type& key(const Node* node);
```

returns a const reference to the key of the given node (lines 14-18):

```
template <class Node>
inline const typename Node::key_type& key(const Node* node)
{
    return node->element.first;
}
```

Given a `BinaryTreeNode<int, char>*` n, for example, the expression

```
key(n) // Return a const reference to the int n->element.first
```

is equivalent to

```
key<BinaryTreeNode<int, char>>(n)
```

The compiler generates the code

```
inline const BinaryTreeNode<int, char>::key_type& key(
    const BinaryTreeNode<int, char>* node)
{
    return node->element.first;
}
```

The function's return type is

```
const BinaryTreeNode<int, char>::key_type& // const int&
```

The main purpose of this function is to improve code readability: the expressions

```
key(n)
```

```
n->element.first
```

both return the key value of n, but the former is more concise and readable.

Similarly, the function (lines 11-12)

```
template <class Pair>
const typename Pair::first_type& key(const Pair& pair);
```

returns a const reference to the key of the given Pair (lines 20-24):

```
template <class Pair>
inline const typename Pair::first_type& key(const Pair& pair)
{
    return pair.first;
}
```

Given an element p of type `Pair<const int, char>`, for example, the expression

```
key(p) // Return a const reference to p.first
```

is equivalent to

```
key<Pair<const int, char>>(p)
```

The compiler generates the code

```
inline const Pair<const int, char>::first_type& key(
    const Pair<int, char>& pair)
{
    return pair.first;
}
```

The function's return type is

```
const Pair<const int, char>::first_type& // const int& const
                                            // Const reference to a const int
```

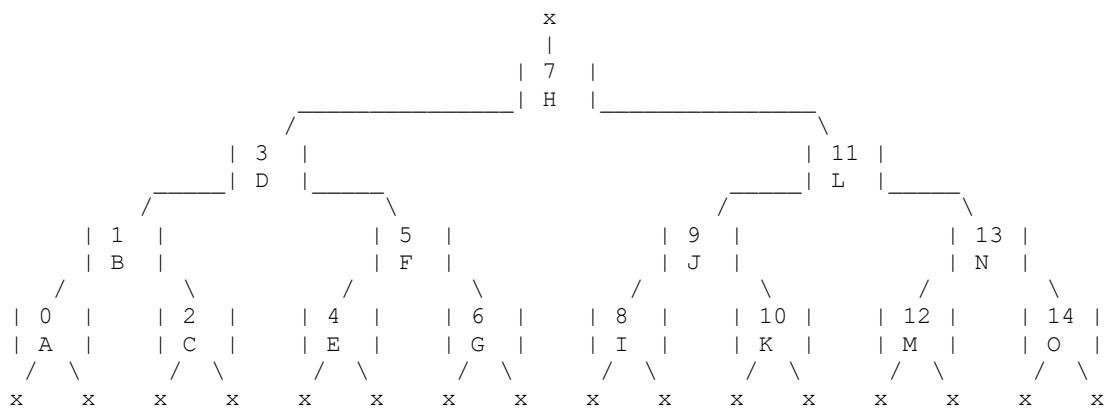
This function, like the other key function (lines 8-9) primarily exists to improve code readability: in the context of a binary tree, the expression

```
key(p) // The key value of element p
```

is more descriptive than

```
p.first
```

Given the tree



and a node pointer n, for example,

If n points to node (7,H),  
 key(n) and key(n->element)  
 both return a const reference to the const int n->element.first (7)

If n points to node (3,D),  
 key(n) and key(n->element)  
 both return a const reference to the const int n->element.first (3)

## 14.3: Introducing the DemoBinaryTree Class

## *Source files and folders*

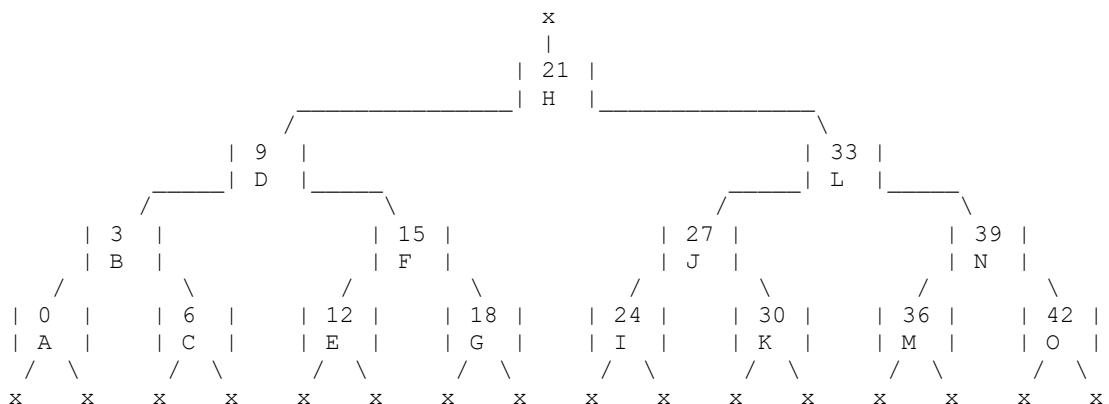
## *DemoBinaryTree/1* *DemoBinaryTree/common/memberFunctions\_1.h*

## *Chapter outline*

- Implementing a test platform for the binary tree traversal and search algorithms

This chapter introduces the `DemoBinaryTree` class, which will be used to demonstrate traversal and search algorithms only. Insertion, erasure, copy, and assignment will be implemented with the `BinaryTree` class template, developed later in this section.

A DemoBinaryTree is simply a fixed-size tree of 15 elements with the layout



Each node is a `BinaryTreeNode<int, char>`, containing an element of type `Pair<const int, char>`.

The class is defined in the header file DemoBinaryTree.h (lines 13-61). Lines 16-18 declare the aliases key type, mapped type, and value type (int, char, Pair<const int, char>). Lines 19 and 44,

```
typedef Less<key_type> key_compare;  
typedef key_compare Predicate;
```

declare `key_compare` and `Predicate` as aliases of the type `Less<int>`. `difference_type` and `size_type` (lines 20-21) are identical to those of the containers developed in previous sections. Lines 22-23,

```
typedef const value_type* const_pointer;
typedef const value_type& const_reference;
```

declare const pointer and const reference as aliases of the types

```
const Pair<const int, char>*      // Const pointer to a Pair<const int, char>
const Pair<const int, char>&      // Const reference to a Pair<const int, char>
```

Similarly, lines 24-25,

```
typedef value_type* pointer;
typedef value_type& reference;
```

declare pointer and reference as aliases of the types

```
Pair<const int, char>*          // Pointer to a Pair<const int, char>
Pair<const int, char>&          // Reference to a Pair<const int, char>
```

Line 26 declares Node as an alias of the type `BinaryTreeNode<int, char>`.

The data members are (lines 55-60)

```
Node* _root;                  // Pointer to the root node
Node* _head;                  // Pointer to the leftmost node, which contains
                             // the front element (first element of the in-order
                             // sequence)
Node* _tail;                  // Pointer to the rightmost node, which contains
                             // the back element (last element of the in-order
                             // sequence)
Predicate _predicate;         // Function object used for sorting and searching
Array<Node*, 15> _n;          // Array of 15 pointers to nodes
Allocator<Node> _alloc;       // Handles the creation / destruction of nodes
```

The member function `empty` (line 31) simply returns false because the `DemoBinaryTree` has a fixed size of 15 elements (`memberFunctions_1.h`, lines 8-11). The member function `size` (`DemoBinaryTree.h`, line 32) simply returns the size of `_n`, 15 (`memberFunctions_1.h`, lines 13-16).

The member functions `front` / `back` (`DemoBinaryTree.h`, lines 33-34, 40-41) return references to the front / back elements (`memberFunctions_1.h`, lines 18-26, 48-56).

The member function `key_comp` (`DemoBinaryTree.h`, line 35) returns a copy of the tree's predicate (`memberFunctions_1.h`, lines 28-31).

The member functions `root` / `head` / `tail` (`DemoBinaryTree.h`, lines 36-38) return const pointers to the root, head, and tail nodes (`memberFunctions_1.h`, lines 33-46).

The private member function (`DemoBinaryTree.h`, line 48)

```
Node* _createNode(const value_type& sourceElement);
```

creates a new `Node`, initializing its element as a copy of the given source element. The function returns

a pointer to the new Node. Its implementation (memberFunctions\_1.h, lines 58-65) is identical to that of the List class (Chapter 10.1):

```
DemoBinaryTree::Node* DemoBinaryTree::_createNode(
    const value_type& sourceElement)
{
    Node* newNode = _alloc.allocate(1);           // Allocate memory for 1
                                                // Node

    _alloc.construct(newNode, Node(sourceElement)); // Construct a Node
                                                // at that address

    return newNode;                                // Return a pointer to
                                                // the new Node
}
```

The private member function `_destroyNode` (DemoBinaryTree.h, line 51) destroys the given Node and releases the memory used to hold it. The implementation (memberFunctions\_1.h, lines 157-161) is also identical to that of List (Chapter 10.1).

The default constructor (DemoBinaryTree.h, line 28) creates 15 nodes, storing pointers to those nodes in the Array `_n`. It then connects the nodes by setting their parent and child links. Once the links have been set, the constructor initializes the remaining data members, `_root` / `_head` / `_tail` (DemoBinaryTree.h, lines 63-71):

```
DemoBinaryTree::DemoBinaryTree()
{
    _createAllNodes();
    _setParentAndChildLinks();

    _root = _n[7];
    _head = bt::leftmostNode(_root);
    _tail = bt::rightmostNode(_root);
}
```

The private member function `_createAllNodes` (DemoBinaryTree.h, line 49) creates the 15 nodes (memberFunctions\_1.h, lines 67-84). In the statement (line 69)

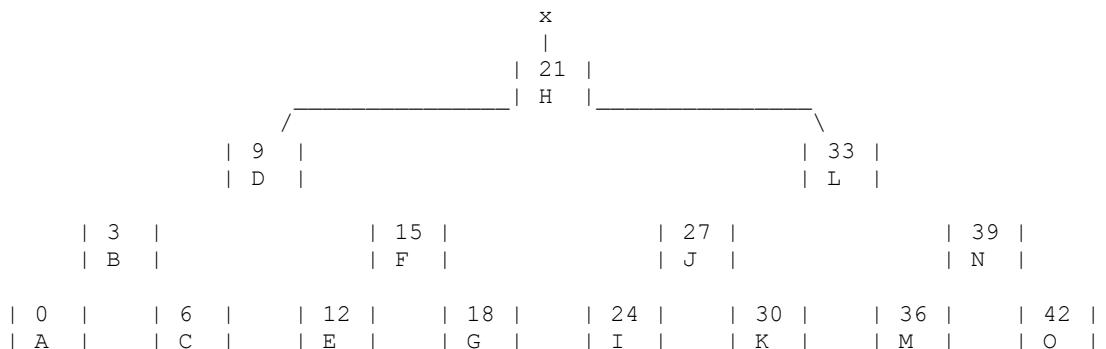
```
_n[0] = _createNode(value_type(0, 'A'));
```

for example, the expression

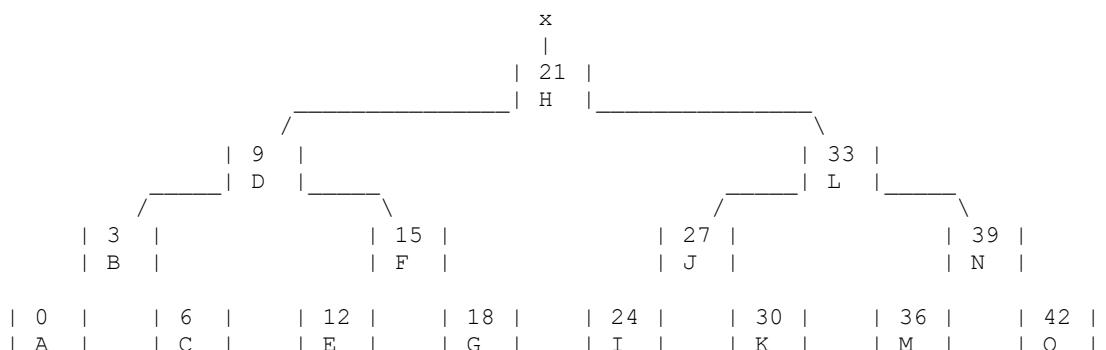
```
value_type(0, 'A') // Pair<const int, char>(0, 'A')
```

constructs the Pair (0,A) via Pair's value constructor (Pair.h, line 19). A new Node is then created from this Pair, after which the address of the new Node is stored in the pointer `_n[0]`. Once all the nodes have been created, the layout of the tree is

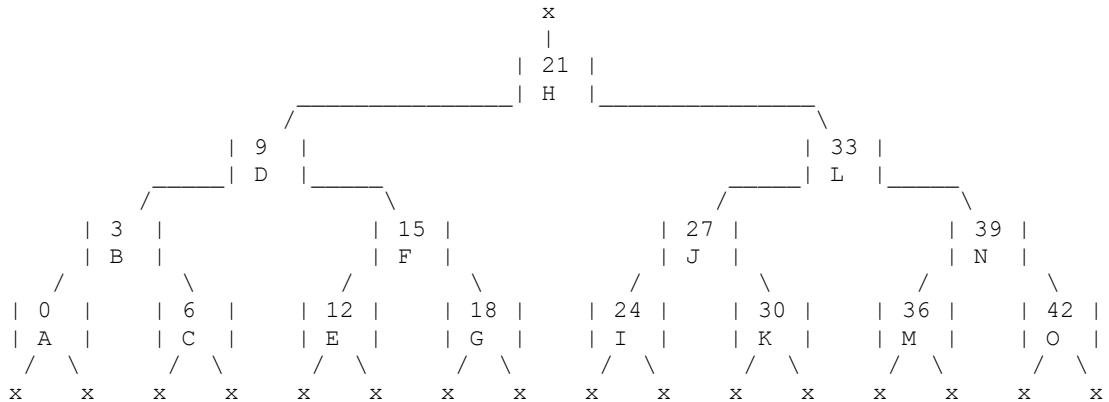
The private member function `_setParentAndChildLinks` (`DemoBinaryTree.h`, line 50) then connects the nodes (`memberFunctions_1.h`, lines 86-155). Lines 90-92 connect the root (node (21,H)) to its parent (null) and children, nodes (9,D) and (33,L):



Lines 96-98 connect node (9,D) to its parent (node (21,H)) and children (nodes (3,B) and (15,F)). Similarly, lines 100-102 connect node (33,L) to its parent (node (21,H)) and children (nodes (27,J) and (39,N)):



Lines 106-120 and 124-154 then connect the remaining nodes, on levels 2 and 3:



After connecting the nodes, the constructor initializes the `_root`, `_head`, and `_tail` pointers to nodes (21,H), (0,A), and (42,O) respectively (`DemoBinaryTree.h`, lines 68-70). Recall from the previous chapter that the expressions

```
bt::leftmostNode(_root)  
bt::rightmostNode( root)
```

return pointers to the leftmost / rightmost nodes, beginning from the `_root`.

The private member function `_destroyAllNodes` (`DemoBinaryTree.h`, line 52) destroys all of the nodes by calling `_destroyNode` for each pointer in `_n` (`DemoBinaryTree.h`, lines 73-77).

The destructor (`DemoBinaryTree.h`, line 29) destroys the tree by simply calling `_destroyAllNodes` (`memberFunctions 1.h`, lines 3-6).

As mentioned at the beginning of this chapter, DemoBinaryTree will not support copy and assignment, so the copy constructor and assignment operator have been disabled by declaring them as private (DemoBinaryTree.h, lines 46, 53). Attempting to perform either of these operations will result in a compiler error.



## 14.4: Recursive In-Order Traversal

*Source files and folders*

*inOrderRecursive  
PrintBtNode*

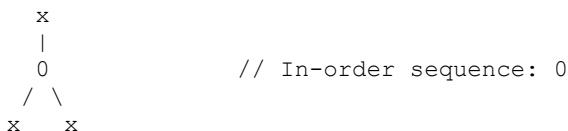
*Chapter outline*

- Verifying the structure of a tree using recursive in-order traversal

Recall from Chapter 7.3 that a recursive function is a function defined in terms of itself. Given a tree and a pointer to the root n, the procedure for performing an in-order traversal of the tree can be defined recursively as

```
traverseInOrder(n)
{
    If n is not null
    {
        traverseInOrder(n->left);      // Traverse n's left subtree
        visit(n);                      // Perform an operation on n
        traverseInOrder(n->right);     // Traverse n's right subtree
    }
}
```

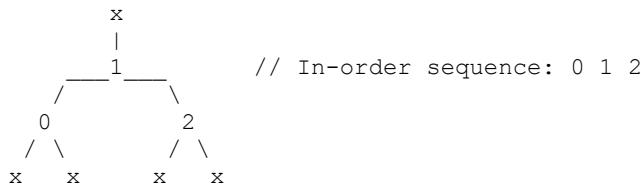
Traversing the tree



for example, entails the following operations:

```
Traverse node 0's left subtree, null
{
    // ...
}
Visit node 0;
Traverse node 0's right subtree, null
{
    // ...
}
```

Traversing the tree

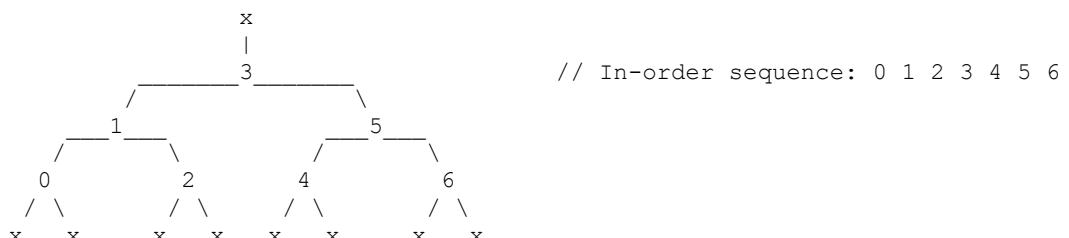


entails the following operations:

```

Traverse 1->left, node 0
{
  Traverse 0->left, null
  {
    // ...
  }
  Visit node 0;
  Traverse 0->right, null
  {
    // ...
  }
}
Visit node 1;
Traverse 1->right, node 2
{
  Traverse 2->left, null
  {
    // ...
  }
  Visit node 2;
  Traverse 2->right, null
  {
    // ...
  }
}
  
```

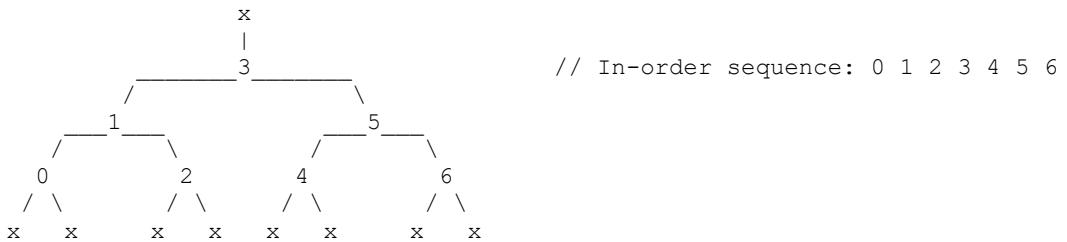
Traversing the tree



entails the following operations:

```
Traverse 3->left, node 1
{
    Traverse 1->left, node 0
    {
        Traverse 0->left, null
        {
            // ...
        }
        Visit node 0;
        Traverse 0->right, null
        {
            // ...
        }
    }
    Visit node 1;
    Traverse 1->right, node 2
    {
        Traverse 2->left, null
        {
            // ...
        }
        Visit node 2;
        Traverse 2->right, null
        {
            // ...
        }
    }
}
Visit node 3;

Traverse 3->right, node 5 (continued on next page)
```



```

Traverse 3->right, node 5
{
  Traverse 5->left, node 4
  {
    Traverse 4->left, null
    {
      // ...
    }
    Visit node 4;
    Traverse 4->right, null
    {
      // ...
    }
  }
  Visit node 5;
  Traverse 5->right, node 6
  {
    Traverse 6->left, null
    {
      // ...
    }
    Visit node 6;
    Traverse 6->right, null
    {
      // ...
    }
  }
}
  
```

The function (inOrderRecursive.h, lines 6-7)

```

template <class Node, class Function>
void traverseInOrder(const Node* n, Function visit);
  
```

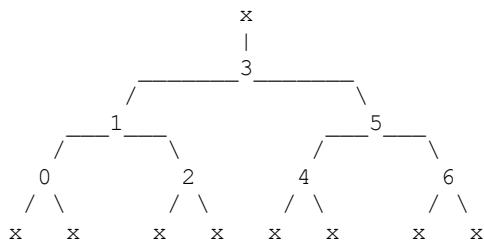
performs a recursive in-order traversal of the tree rooted at n, calling the given visit function for each node (lines 9-18):

```
template <class Node, class Function>
void traverseInOrder(const Node* n, Function visit)
{
    if (n != nullptr)
    {
        traverseInOrder(n->left, visit);
        visit(n);
        traverseInOrder(n->right, visit);
    }
}
```

The function object (PrintBtNode.h, lines 11-15)

```
template <class Node>
struct PrintBtNode // Print binary tree node
{
    void operator()(const Node* n) const;
};
```

visits the given node n by printing the key values of n and n's parent, left child, and right child, where applicable (lines 17-33). Given the tree



for example, visiting node 5 generates the output

```
node 5
parent 3
left 4
right 6
```

while visiting node 2 generates the output

```
node 2
parent 1
```

The program in this chapter performs a recursive in-order traversal of a DemoBinaryTree, using PrintBtNode to visit each node. Line 13 (main.cpp) constructs a DemoBinaryTree t, and line 14 constructs printNode, a function object of type PrintBtNode<DemoBinaryTree::Node>. Recall from the previous chapter that DemoBinaryTree::Node is an alias of the type BinaryTreeNode<int, char>. The compiler generates the class

```
struct PrintBtNode<DemoBinaryTree::Node>
{
    void operator()(const DemoBinaryTree::Node* n) const;
};
```

Line 16,

```
traverseInOrder(t.root(), printNode);
```

traverses the tree in-order, calling printNode for nodes 0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, and 42. The function call (line 16) is equivalent to

```
traverseInOrder<DemoBinaryTree::Node, PrintBtNode<DemoBinaryTree::Node>>(
    t.root(),
    printNode);
```

The compiler generates the function

```
void traverseInOrder(const DemoBinaryTree::Node* n,
PrintBtNode<DemoBinaryTree::Node> visit)
{
    if (n != nullptr)
    {
        traverseInOrder(n->left, visit);
        visit(n); // visit.operator()(n);
        traverseInOrder(n->right, visit);
    }
}
```

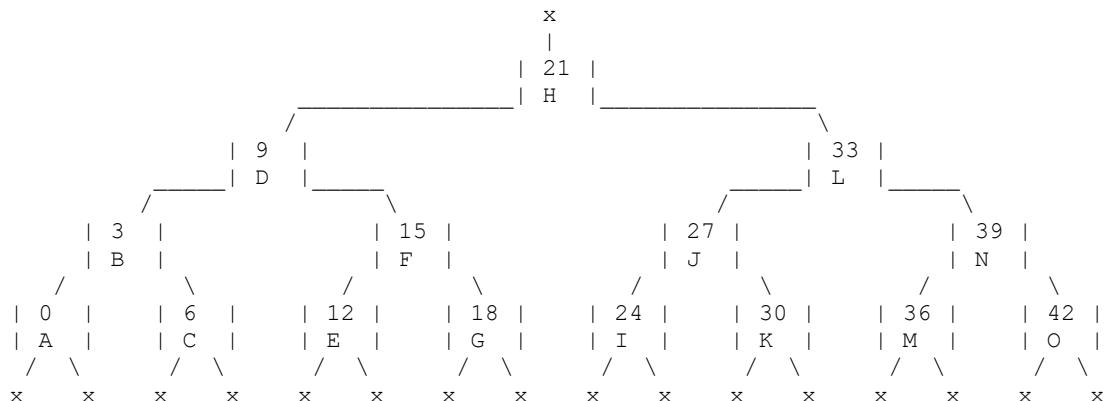
The call to traverseInOrder verifies the structure of the tree, generating the output

```
node 0          // Refer to the diagram on the next page
  parent 3
node 3
  parent 9
    left 0
    right 6
node 6
  parent 3
node 9
  parent 21
    left 3
    right 15
node 12
  parent 15
node 15
  parent 9
    left 12
    right 18
node 18
  parent 15
node 21
```

```

left 9
right 33
node 24
  parent 27
node 27
  parent 33
  left 24
  right 30
node 30
  parent 27
node 33
  parent 21
  left 27
  right 39
node 36
  parent 39
node 39
  parent 33
  left 36
  right 42
node 42
  parent 39

```





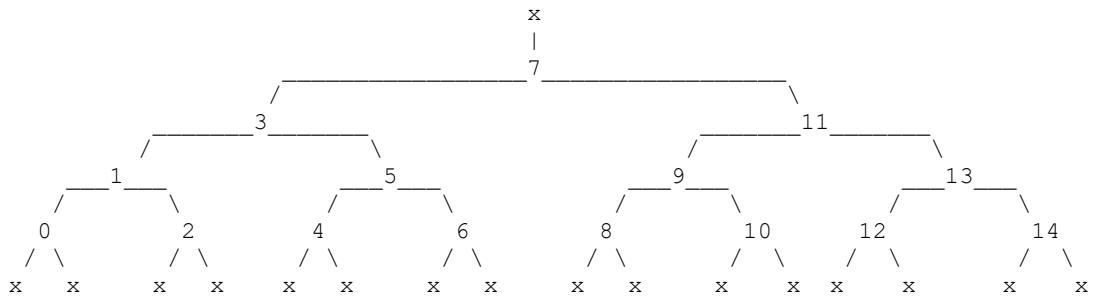
## 14.5: Iterative In-Order Traversal

*Source folder: inOrderIterative*

*Chapter outline*

- Locating the in-order predecessor / successor of a given node, using iteration

Consider the tree



The function (inOrderIterative.h, lines 8-9)

```
template <class Node>
Node* inOrderPredecessor(Node* n);
```

returns a pointer to the in-order predecessor of the given node n. The function consists of 3 branches (lines 14-35).

Case 1: n has a left child (lines 19-22)

n's predecessor is the rightmost node of n's left subtree:

```
The predecessor of node 1 is rightmostNode(1->left), 0
The predecessor of node 5 is rightmostNode(5->left), 4
The predecessor of node 9 is rightmostNode(9->left), 8
The predecessor of node 13 is rightmostNode(13->left), 12
The predecessor of node 3 is rightmostNode(3->left), 2
The predecessor of node 11 is rightmostNode(11->left), 10
The predecessor of node 7 is rightmostNode(7->left), 6
```

Case 2: n does not have a left child, but n is the right child of its parent (lines 23-26)

n's predecessor is n's parent:

```
The predecessor of node 2 is 2->parent, 1
The predecessor of node 6 is 6->parent, 5
The predecessor of node 10 is 10->parent, 9
```

The predecessor of node 14 is 14->parent, 13

Case 3: n does not have a left child, and n is not the right child of its parent (lines 27-34)

The predecessor of  $n$  is located by finding the nearest parent  $p$  that is a right child.  $n$ 's predecessor is  $p$ 's parent.

If no such parent exists, then  $n$  is the first node in the sequence and  $n$ 's predecessor is null.

To locate the predecessor of node 0, for example:

To locate the predecessor of node 4:

```
p = n->parent; // p points to node 5

Iteration 1:
isRightChild(p) returns true,
so return p->parent (a pointer to node 3)
```

To locate the predecessor of node 8:

```
p = n->parent;                                // p points to node 9

Iteration 1:
    isRightChild(p) returns false;
    p = p->parent;                            // p points to node 11

Iteration 2:
    isRightChild(p) returns true,
        so return p->parent (a pointer to node 7)
```

To locate the predecessor of node 12:

```
p = n->parent; // p points to node 13
```

```

Iteration 1:
isRightChild(p) returns true,
so return p->parent (a pointer to node 11)

```

The function (lines 11-12)

```

template <class Node>
Node* inOrderSuccessor(Node* n);

```

returns a pointer to the in-order successor of the given node n. The mirror image of inOrderPredecessor, this function also consists of 3 branches (lines 37-58).

Case 1: n has a right child (lines 42-45)

n's successor is the leftmost node of n's right subtree:

```

The successor of node 1 is leftmostNode(1->right), 2
The successor of node 5 is leftmostNode(5->right), 6
The successor of node 9 is leftmostNode(9->right), 10
The successor of node 13 is leftmostNode(13->right), 14
The successor of node 3 is leftmostNode(3->right), 4
The successor of node 11 is leftmostNode(11->right), 12
The successor of node 7 is leftmostNode(7->right), 8

```

Case 2: n does not have a right child, but n is the left child of its parent (lines 46-49)

n's successor is n's parent:

```

The successor of node 0 is 0->parent, 1
The successor of node 4 is 4->parent, 5
The successor of node 8 is 8->parent, 9
The successor of node 12 is 12->parent, 13

```

Case 3: n does not have a right child, and n is not the left child of its parent (lines 50-57)

The successor of n is located by finding the nearest parent p that is a left child. n's successor is p's parent.

If no such parent exists, then n is the last node in the sequence and n's successor is null.

To locate the successor of node 2, for example:

```

p = n->parent;                                // p points to node 1

Iteration 1:
isLeftChild(p) returns true,
so return p->parent (a pointer to node 3)

```

To locate the successor of node 6:

```

p = n->parent;                                // p points to node 5

Iteration 1:
isLeftChild(p) returns false;
p = p->parent;                                // p points to node 3

Iteration 2:
isLeftChild(p) returns true,
so return p->parent (a pointer to node 7)

```

To locate the successor of node 10:

```

p = n->parent;                                // p points to node 9

Iteration 1:
isLeftChild(p) returns true,
so return p->parent (a pointer to node 11)

```

To locate the successor of node 14:

```

p = n->parent;                                // p points to node 13

Iteration 1:
isLeftChild(p) returns false;
p = p->parent;                                // p points to node 11

Iteration 2:
isLeftChild(p) returns false;
p = p->parent;                                // p points to node 7

Iteration 3:
isLeftChild(p) returns false;
p = p->parent;                                // p is null...
                                                 // ...so node 14's successor is null
return nullptr;

```

The program in this chapter uses iteration to traverse a DemoBinaryTree. Line 12 (main.cpp) constructs the tree, and the loop in lines 14-19 prints each element from head to tail. n is initialized to point to the head. In each iteration, the loop prints n's element then points n to its in-order successor. The loop performs 15 iterations, generating the output

```
(0,A) (3,B) (6,C) (9,D) (12,E) (15,F) (18,G) (21,H) (24,I) (27,J) (30,K) (33,L)
) (36,M) (39,N) (42,O)
```

Similarly, the loop in lines 23-28 prints the sequence in reverse order, generating the output

```
(42,O) (39,N) (36,M) (33,L) (30,K) (27,J) (24,I) (21,H) (18,G) (15,F) (12,E) (
9,D) (6,C) (3,B) (0,A)
```

## 14.6: Implementing the Iterators

*Source files and folders*

*BinaryTreeIter  
DemoBinaryTree/2  
DemoBinaryTree/common/memberFunctions\_2.h*

*Chapter outline*

- *The BinaryTreeIter class*
- *Implementing begin / end / rbegin / rend for the DemoBinaryTree class*

The BinaryTreeIter class, an iterator for a binary tree, follows the exact same design pattern as the ListIter class from Chapter 10.3. The class is defined in the header file BinaryTreeIter.h (lines 8-42). The template parameter Tree (line 8) is the type of the iterator's underlying tree. A BinaryTreeIter contains two pointers, one to the underlying tree (\_tree) and one to the node containing the referent element (\_node) (lines 40-41).

The only difference between BinaryTreeIter and ListIter is that the private member functions \_pointToNextNode / \_pointToPreviousNode (lines 37-38, 120-133) point \_node to the successor / predecessor (instead of the right / left neighbor).

BinaryTreeIter is compatible with any binary tree class that:

- Defines the member types pointer, reference, difference\_type, and value\_type (lines 14-17)
- Defines a member type, Node, which contains a predecessor and successor link (lines 33, 123, 130)
- Contains a data member \_tail, which is a pointer to the last (rightmost) Node in the tree (line 132)
- Declares BinaryTreeIter as a friend (if \_tail is a private data member)

Line 18 (DemoBinaryTree.h) declares the class `BinaryTreeIter<DemoBinaryTree>` as a friend of DemoBinaryTree. This allows BinaryTreeIter's member function \_pointToPreviousNode to access DemoBinaryTree's private data member \_tail (BinaryTreeIter.h, line 132).

Line 30 (DemoBinaryTree.h),

```
typedef BinaryTreeIter<DemoBinaryTree> iterator;
```

declares `DemoBinaryTree::iterator` as an alias of the type `BinaryTreeIter<DemoBinaryTree>`. DemoBinaryTree's remaining iterator types are declared in lines 31 and 26-27:

```
typedef ReverseIter<iterator> reverse_iterator;
typedef ConstIter<DemoBinaryTree> const_iterator;
typedef ReverseIter<const_iterator> const_reverse_iterator;
```

The implementation of begin, end, rbegin, and rend (DemoBinaryTree.h, lines 41-44, 52-55 /

`memberFunctions_2.h`, lines 24-62) is identical to that of List (Chapters 10.3, 10.4, 11.1).

The function definition of the constructor has been updated and moved from DemoBinaryTree.h to memberFunctions\_2.h (lines 6-22). After creating the nodes, setting the parent / child links, and initializing the \_root / \_head / \_tail, it then sets the predecessor / successor links of each node (lines 15-21):

```
Node* n = _head;
while (n != nullptr)
{
    n->predecessor = inOrderPredecessor(n);
    n->successor = inOrderSuccessor(n);
    n = n->successor;
}
```

The loop performs 15 iterations:

```
Node* n = _head;    // n points to node (0,A)  
                    // p / s denote the node's predecessor / successor links
```

n  
0  
A  
p  
s

## Iteration 1:

n	
0	3
A	B
x <-p	p
s->	s

```
n = n->successor; // n points to node (3, B)
```

	n
0	3
A	B
x <-p	p
	s-> s

## Iteration 2:

```
n->predecessor = inOrderPredecessor(n); // Point n's predecessor link  
// to node (0, A)
```

```

n->successor = inOrderSuccessor(n);           // Point n's successor link to
                                               // node (6,C)

      n
      0   3   6
      A   B   C
x <-p <-p   p
s-> s-> s

n = n->successor;                          // n points to node (6,C)

```

```

      n
      0   3   6
      A   B   C
x <-p <-p   p
s-> s-> s

```

Iteration 3:

```

n->predecessor = inOrderPredecessor(n);       // Point n's predecessor link
                                               // to node (3,C)

n->successor = inOrderSuccessor(n);           // Point n's successor link to
                                               // node (9,D)

      n
      0   3   6   9
      A   B   C   D
x <-p <-p <-p   p
s-> s-> s-> s

n = n->successor;                          // n points to node (9,D)

```

```

      n
      0   3   6   9
      A   B   C   D
x <-p <-p <-p   p
s-> s-> s-> s

```

// Iterations 4-15 set the remaining links, after which the layout becomes

```

      0   3   6   9   12   15   18   21   24   27   30   33   36   39   42
      A   B   C   D   E   F   G   H   I   J   K   L   M   N   O
x <-p <-p
s-> x

```

Now that the predecessor / successor links are intact, the private member function `_destroyAllNodes` can destroy the tree as if it were a linked list. The function definition of `_destroyAllNodes` has also been moved from `DemoBinaryTree.h` to `memberFunctions_2.h` (lines 64-75). The new implementation is nearly identical to that of List (Chapter 10.1): the only difference is that it uses each node's successor link (instead of the right neighbor link) to obtain the next node in the sequence.

The program in this chapter demonstrates `DemoBinaryTree::const_iterator` and

DemoBinaryTree::const\_reverse\_iterator. Line 12 (main.cpp) constructs a DemoBinaryTree t, and lines 14-18,

```
cout << "t contains " << t.size() << " elements:\n";
printContainer(t);

cout << "\nThe reverse sequence is:\n";
printContainerReverse(t);
```

generate the output

```
t contains 15 elements:
(0,A) (3,B) (6,C) (9,D) (12,E) (15,F) (18,G) (21,H) (24,I) (27,J) (30,K) (33,L)
) (36,M) (39,N) (42,O)

The reverse sequence is:
(42,O) (39,N) (36,M) (33,L) (30,K) (27,J) (24,I) (21,H) (18,G) (15,F) (12,E) (
9,D) (6,C) (3,B) (0,A)
```

The BinaryTreeIter class will be reused with the BinaryTree and AvlTree classes, developed in later chapters.

## 14.7: Retrieving Elements by Key Value

*Source files and folders*

*bt/findNode.h*  
*DemoBinaryTree/3*  
*DemoBinaryTree/common/memberFunctions\_3.h*

*Chapter outline*

- *Binary tree search*

The function (*findNode.h*, lines 10-13)

```
template <class Node, class Predicate>
Node* findNode(Node* root,
    const typename Node::key_type& searchKey,
    Predicate predicate);
```

returns a pointer to the node containing the given *searchKey*. The function begins at the root and traverses the tree downward, using the given predicate to compare the *searchKey* with the key value of each node. If the *searchKey* is not found, the function returns a null pointer (lines 15-31):

```
template <class Node, class Predicate>
Node* findNode(Node* root,
    const typename Node::key_type& searchKey,
    Predicate predicate)
{
    for (Node* n = root; n != nullptr;)
    {
        if (predicate(searchKey, key(n)))           // Case 1
            n = n->left;
        else if (predicate(key(n), searchKey))       // Case 2
            n = n->right;
        else                                         // Case 3
            return n;
    }

    return nullptr;
}
```

The loop consists of 3 branches. Suppose, for example, that the given predicate returns true if the first argument is less than the second. The expression

```
predicate(searchKey, key(n))
```

will therefore return true if the *searchKey* is less than the key value of the current node, and the expression

```
predicate(key(n), searchKey)
```

will return true if the searchKey is greater than the key value of the current node. If neither of those expressions return true, then the key value of the current node must be equal to the searchKey. The loop can thus be described using the following pseudocode:

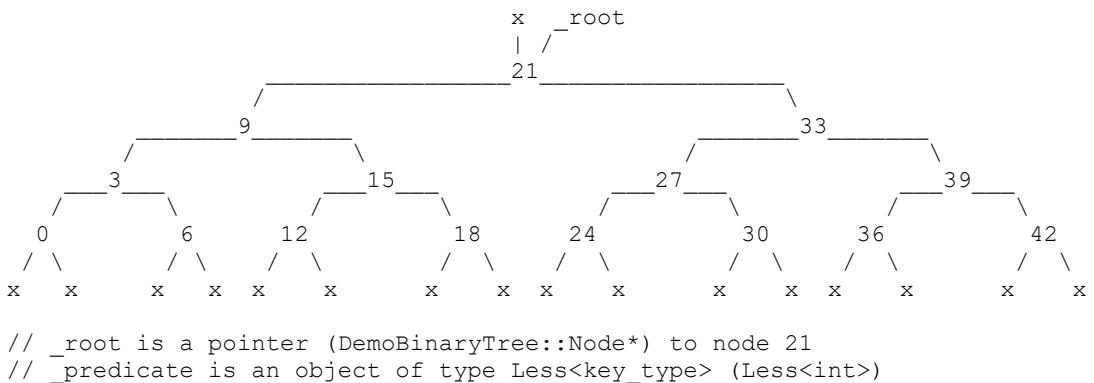
```
If the searchKey is less than the key value of the current node,  
proceed to the left child (Case 1);
```

```
Otherwise, if the searchKey is greater than the key value of the current node,  
proceed to the right child (Case 2);
```

```
Otherwise, the key value of the current node is equal to the searchKey,  
so return a pointer to the current node (Case 3);
```

If the searchKey is never found, the loop will eventually terminate when n becomes null (i.e. when the bottom of the tree is reached), at which point the function returns a null pointer (line 30).

Consider, for example, a DemoBinaryTree (mapped values are not shown):



The function call

```
findNode(_root, 12, _predicate)      // Return a pointer to the node containing
                                         // a key value of 12
```

is equivalent to

```
findNode<DemoBinaryTree::Node, Less<int>>(_root, 12, _predicate)
```

The compiler generates the function

```
DemoBinaryTree::Node* findNode(DemoBinaryTree::Node* root,
  const DemoBinaryTree::Node::key_type& searchKey,
  Less<DemoBinaryTree::Node::key_type> predicate);
```

which is equivalent to

```
BinaryTreeNode<int, char>* findNode(BinaryTreeNode<int, char>* root,
const int& searchKey,
Less<int> predicate);
```

The function call

```
findNode(_root, 12, _predicate) // searchKey is 12
```

performs the following operations:

```
Node* n = root; // n points to node 21
```

Iteration 1:

```
if (predicate(searchKey, key(n)))
    n = n->left;
else if (predicate(key(n), searchKey))
    n = n->right;
else
    return n;
```

Iteration 2:

```
if (predicate(searchKey, key(n)))
    n = n->left;
else if (predicate(key(n), searchKey))
    n = n->right; // n points to node 15
else
    return n;
```

Iteration 3:

```
if (predicate(searchKey, key(n)))
    n = n->left;
else if (predicate(key(n), searchKey))
    n = n->right;
else
    return n;
```

Iteration 4:

```
if (predicate(searchKey, key(n)))
    n = n->left;
else if (predicate(key(n), searchKey))
    n = n->right;
else
    return n; // Return a pointer to node 12
```

The function call

```
findNode(_root, 28, _predicate) // searchKey is 28
```

performs the following operations:

```

Node* n = root;                                // n points to node 21

Iteration 1:

if (predicate(searchKey, key(n)))
    n = n->left;
else if (predicate(key(n), searchKey))
    n = n->right;                            // n points to node 33
else
    return n;

Iteration 2:

if (predicate(searchKey, key(n)))
    n = n->left;                            // n points to node 27
else if (predicate(key(n), searchKey))
    n = n->right;
else
    return n;

Iteration 3:

if (predicate(searchKey, key(n)))
    n = n->left;
else if (predicate(key(n), searchKey))
    n = n->right;                            // n points to node 30
else
    return n;

Iteration 4:

if (predicate(searchKey, key(n)))
    n = n->left;                            // n is null
else if (predicate(key(n), searchKey))
    n = n->right;
else
    return n;

// Terminate loop

return nullptr;                                // Return a null pointer,
                                                // indicating that the searchKey
                                                // was not found

```

The const-overloaded member function (DemoBinaryTree.h, lines 47, 59)

```

const_iterator find(const key_type& searchKey) const;
iterator find(const key_type& searchKey);

```

finds the element containing the given searchKey and returns an iterator pointing to that element. If the searchKey is not found, the function returns an iterator to the end (one-past-the-last element).

The non-const version uses findNode to retrieve a pointer to the desired node, then returns an iterator

containing that pointer (`memberFunctions_3.h`, lines 11-15):

```
inline DemoBinaryTree::iterator DemoBinaryTree::find(
    const key_type& searchKey)
{
    return iterator(this, bt::findNode(_root, searchKey, _predicate));
}
```

The const version is implemented using the same design pattern as the const versions of begin and end (lines 5-9):

```
inline DemoBinaryTree::const_iterator DemoBinaryTree::find(
    const key_type& searchKey) const
{
    return const_cast<DemoBinaryTree*>(this)->find(searchKey);
}
```

The expression

```
const_cast<DemoBinaryTree*>(this)
```

returns a non-const “this” pointer, which is used to call the non-const version of `find`:

```
->find(searchKey)
```

The non-const version of `find` returns an iterator, which is then used to construct the `const_iterator` returned by the function. The statement

```
return const_cast<DemoBinaryTree*>(this)->find(searchKey);
```

is equivalent to

```
return const_iterator(
    const_cast<DemoBinaryTree*>(this)->find(searchKey));

// Return a const_iterator constructed from the iterator returned by
// the non-const version of find
```

The program in this chapter demonstrates `DemoBinaryTree`'s `find` method. Lines 12-13 (`main.cpp`) construct a `DemoBinaryTree t` and print its contents. Lines 15-17 prompt the user to enter the key value of a desired element (`searchKey`, of type `int`). Line 19 then obtains an iterator `i` pointing to the desired element, at which point `main` splits into 2 branches.

If the desired element was found (lines 21-29), the program prints the element, prompts the user to enter a new mapped value for that element, and prints the contents of `t` once again.

If, however, the desired element was not found (lines 30-33), the program informs the user accordingly.

Some sample runs of the program are

```
// Case 1

(0,A) (3,B) (6,C) (9,D) (12,E) (15,F) (18,G) (21,H) (24,I) (27,J) (30,K) (33,L)
) (36,M) (39,N) (42,O)

Enter key value (int) of desired element: 12

Found element (12,E)
Enter new mapped value (char): x      // (12,E) becomes (12,x)

(0,A) (3,B) (6,C) (9,D) (12,x) (15,F) (18,G) (21,H) (24,I) (27,J) (30,K) (33,L)
) (36,M) (39,N) (42,O)

// Case 2

(0,A) (3,B) (6,C) (9,D) (12,E) (15,F) (18,G) (21,H) (24,I) (27,J) (30,K) (33,L)
) (36,M) (39,N) (42,O)

Enter key value (int) of desired element: 28

No element with a key value of 28 exists.
```

## 14.8: Introducing the BinaryTree Class Template

*Source files and folders*

*BinaryTree/1  
BinaryTree/common/memberFunctions\_1.h*

*Chapter outline*

- *Template parameters and member types*
- *Default constructor / destructor*
- *Accessor functions*

This chapter introduces the `BinaryTree` class, a fully functional binary tree.

The class is defined in the header file `BinaryTree.h` (lines 14-72). The first two template parameters, `Key` and `Mapped` (line 14), are the types of the key / mapped values stored in each element (key-mapped pair). The third template parameter, `Predicate`, is the type of predicate used to sort and search the elements. In line 14, the

```
= Less<Key>
```

specifies `Less<Key>` as a default template argument for `Predicate`: if no predicate is specified, the compiler will automatically use the type `Less<Key>`. The statements

```
BinaryTree<int, char, Less<int>> t;  
  
// Construct t, a BinaryTree of int-char Pairs, sorted and searched using a  
// predicate of type Less<int>  
  
BinaryTree<string, double, Less<string>> u;  
  
// Construct u, a BinaryTree of string-double Pairs, sorted and searched  
// using a predicate of type Less<string>
```

for example, can simply be written as

```
BinaryTree<int, char> t;  
BinaryTree<string, double> u;
```

As demonstrated in the previous chapters, using a less-than predicate will sort the elements in ascending order of their key values. Using a greater-than predicate, as in

```
BinaryTree<int, char, Greater<int>> t;  
BinaryTree<string, double, Greater<string>> u;
```

will sort the elements in descending order of their key values.

BinaryTree's key\_type and mapped\_type are simply aliases of the template parameters Key and Mapped (lines 20-21). key\_compare (line 23) is an alias of the template parameter Predicate. The remaining member types (lines 22, 24-34) are identical to those of DemoBinaryTree.

The data members are (lines 66-71)

```
Node* _root;           // Pointer to the root node
Node* _head;           // Pointer to the head node
Node* _tail;           // Pointer to the tail node
size_type _size;        // Number of contained elements (key-mapped pairs)
Predicate _predicate;   // Function object used for sorting and searching
Allocator<Node> _alloc; // Handles creation / destruction of nodes
```

All of these (except for \_size) are identical to their counterparts in DemoBinaryTree.

As discussed in Chapter 14.6, the friend declaration (line 18) allows BinaryTreeIter's \_pointToPreviousNode function to access the \_tail.

The default constructor (line 36) constructs an empty BinaryTree, initializing the \_root, \_head, and \_tail to null, and the \_size to 0 (memberFunctions\_1.h, lines 5-13).

The member function empty (BinaryTree.h, line 39) returns true if the \_size is 0; otherwise, it returns false (memberfunctions\_1.h, lines 21-25).

The member function size (BinaryTree.h, line 40) returns the \_size (memberfunctions\_1.h, lines 27-32). The return type is

```
typename BinaryTree<Key, Mapped, Predicate>::size_type
```

and the full name of the function is

```
BinaryTree<Key, Mapped, Predicate>::size() const
```

The implementations of the remaining member functions (BinaryTree.h, lines 37, 41-64 / memberFunctions\_1.h, lines 15-19, 34-189) are identical to those of DemoBinaryTree.

The remaining chapters in this section will implement the insertion, erasure, copy, and assignment algorithms.

## 14.9: Insert

*Source files and folders*

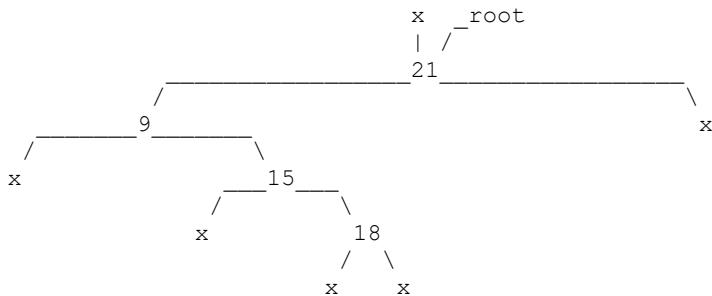
*BinaryTree/2*  
*BinaryTree/common/memberFunctions\_2.h*  
*bt/attachNewNode.h*  
*bt/findInsertionPoint.h*

*Chapter outline*

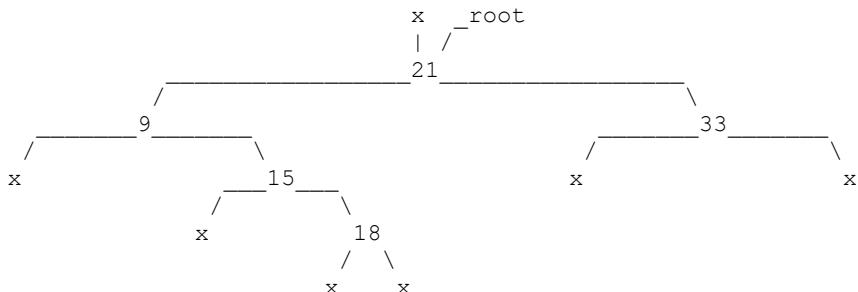
- Locating the insertion point for a new node
- Attaching a new node
- Implementing *BinaryTree*'s insert method

Inserting a new node into an empty tree simply involves creating the node and pointing the `_root`, `_head`, and `_tail` to that node.

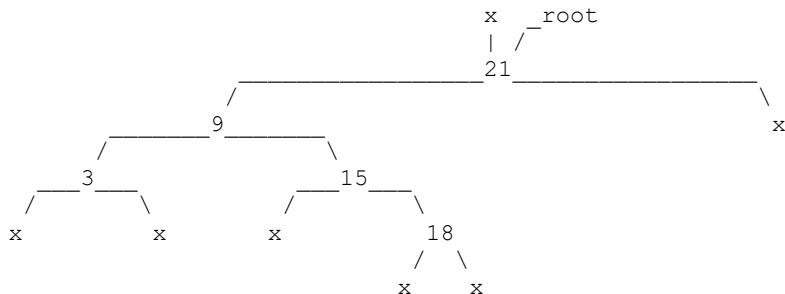
In a non-empty tree, however, the new node is attached to an existing node called the “insertion point.” Consider, for example, the tree



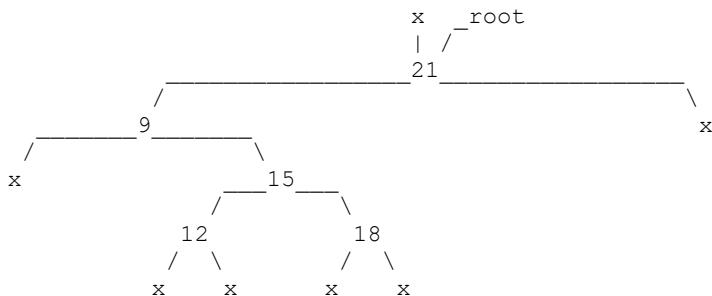
When inserting a new node with a key value of 33, the insertion point is node 21:



When inserting a new node with a key value of 3, the insertion point is node 9:



When inserting a new node with a key value of 12, the insertion point is node 15:



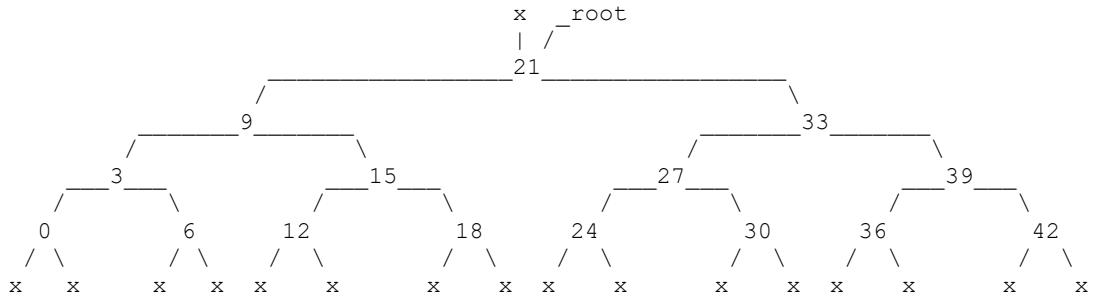
The function (findInsertionPoint.h, lines 10-13)

```

template <class Node, class Predicate>
Node* findInsertionPoint(Node* root,
    const typename Node::key_type& newKey,
    Predicate predicate);
  
```

returns a pointer to the insertion point for a new node with a key value of newKey. If a node with newKey already exists in the tree, the function returns a pointer to that node.

The function locates the insertion point by searching for newKey as if it already existed in the tree. The implementation of findInsertionPoint is nearly identical to that of findNode (Chapter 14.7); the only difference is that it uses an additional pointer, insertionPoint, which “trails” n in the search for newKey (lines 15-35). Consider, for example, a `BinaryTree<int, int, Less<int>>` (mapped values are not shown):



When inserting a new node with a key value of 28, the insertion point for the new node is located via the function call

```
findInsertionPoint(_root, 28, _predicate);
```

The function performs the following operations:

```
// newKey is 28

Node* insertionPoint = root;           // insertionPoint points to node 21
Node* n = root;                      // n points to node 21

Iteration 1:

insertionPoint = n;                  // insertionPoint points to node 21

if (predicate(newKey, key(n)))
    n = n->left;
else if (predicate(key(n), newKey))
    n = n->right;                 // n points to node 33
else
    return n;

Iteration 2:

insertionPoint = n;                  // insertionPoint points to node 33

if (predicate(newKey, key(n)))
    n = n->left;
else if (predicate(key(n), newKey))
    n = n->right;                // n points to node 27
else
    return n;

Iteration 3:

insertionPoint = n;                  // insertionPoint points to node 27
```

```

if (predicate(newKey, key(n)))
    n = n->left;
else if (predicate(key(n), newKey))
    n = n->right;                                // n points to node 30
else
    return n;

```

## Iteration 4:

When inserting a new node with a key value of 12, the insertion point for the new node is located via the function call

```
findInsertionPoint( root, 12, predicate);
```

The function performs the following operations:

```
// newKey is 12
```

```
Node* insertionPoint = root; // insertionPoint points to node 21  
Node* n = root; // n points to node 21
```

## Iteration 1:

```

insertionPoint = n;                                // insertionPoint points to node 21

if (predicate(newKey, key(n)))
    n = n->left;                               // n points to node 9
else if (predicate(key(n), newKey))
    n = n->right;
else
    return n;

```

## Iteration 2:

```
insertionPoint = n; // insertionPoint points to node 9
```

```

if (predicate(newKey, key(n)))
    n = n->left;
else if (predicate(key(n), newKey))
    n = n->right;
else
    return n;

```

Iteration 3:

```

insertionPoint = n;                                // insertionPoint points to node 15

if (predicate(newKey, key(n)))
    n = n->left;
else if (predicate(key(n), newKey))
    n = n->right;
else
    return n;

```

Iteration 4:

```

insertionPoint = n;                                // insertionPoint points to node 12

if (predicate(newKey, key(n)))
    n = n->left;
else if (predicate(key(n), newKey))
    n = n->right;
else
    return n;                                     // Return a pointer to node 12

```

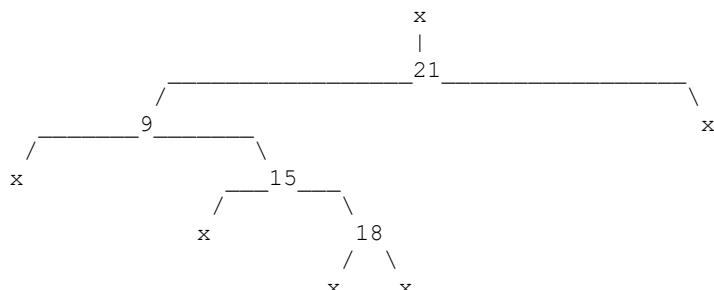
Once the insertion point is located, the new node is attached by updating the appropriate links. The function (attachNewNode.h, lines 12-13)

```

template <class Node, class Predicate>
void attachNewNode(Node* newNode, Node* insertionPoint, Predicate predicate);

```

attaches the new node to the insertion point, using the given predicate (lines 15-34). Consider, for example, the tree



```

9   15   18   21
x <-p <-p <-p <-p
s-> s-> s-> s-> x

```

When inserting a new node with a key value of 12, the insertion point is node 15. attachNewNode performs the following operations:

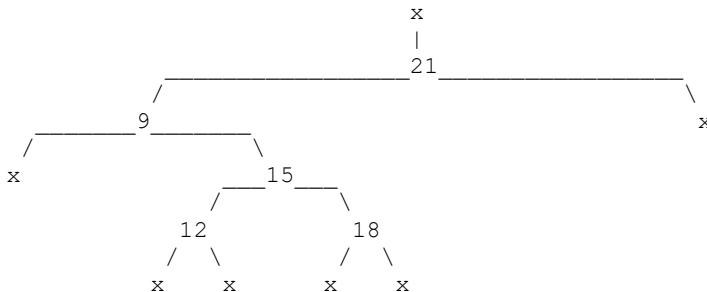
```
// newNode points to node 12
// insertionPoint points to node 15

// Point node 12's parent link to node 15

newNode->parent = insertionPoint;

// Determine whether node 12 should become the left or right child of
// node 15, and update node 15's left or right link accordingly

if (predicate(key(newNode), key(insertionPoint)))
    insertionPoint->left = newNode;                                // Point node 15's left
else                                         // link to node 12
    insertionPoint->right = newNode;
```



```
// Point node 12's predecessor link to node 9

newNode->predecessor = inOrderPredecessor(newNode);

// Point node 9's successor link to node 12

if (hasPredecessor(newNode))
    newNode->predecessor->successor = newNode;

// Point node 12's successor link to node 15

newNode->successor = inOrderSuccessor(newNode);

// Point node 15's predecessor link to node 12

if (hasSuccessor(newNode))
    newNode->successor->predecessor = newNode;

9   12   15   18   21
x <-p <-p <-p <-p <-p
s-> s-> s-> s-> s-> x
```

When inserting a new node with a key value of 33, the insertion point is node 21. attachNewNode

performs the following operations:

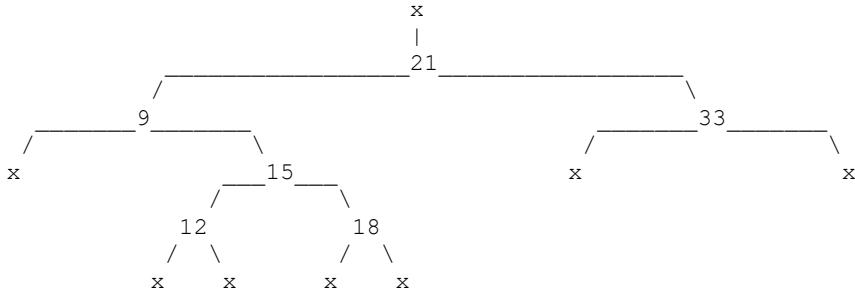
```
// newNode points to node 33
// insertionPoint points to node 21

// Point node 33's parent link to node 21

newNode->parent = insertionPoint;

// Determine whether node 33 should become the left or right child of
// node 21, and update node 21's left or right link accordingly

if (predicate(key(newNode), key(insertionPoint)))
    insertionPoint->left = newNode;
else
    insertionPoint->right = newNode;                      // Point node 21's right
                                                               // link to node 33
```



```
// Point node 33's predecessor link to node 21

newNode->predecessor = inOrderPredecessor(newNode);

// Point node 21's successor link to node 33

if (hasPredecessor(newNode))
    newNode->predecessor->successor = newNode;

// Point node 33's successor link to null

newNode->successor = inOrderSuccessor(newNode);

// Node 33 does not have a successor (its successor is null),
// so it is neither necessary nor possible to update its successor's
// predecessor link

if (hasSuccessor(newNode))
    newNode->successor->predecessor = newNode;      // Not executed

9   12   15   18   21   33
x <-p <-p <-p <-p <-p
  s-> s-> s-> s-> s-> x
  
```

The member function (BinaryTree.h, line 60)

```
Pair<iterator, bool> insert(const value_type& sourceElement);
```

inserts a copy of the given source element into the tree and returns a Pair<iterator, bool> p.

If the tree does not already contain an element with the same key value as the source element, then p.first will point to the newly inserted element and p.second will be true, indicating that a new element was inserted.

If, however, the tree already contains an element with the same key value as the source element, then p.first will point to the pre-existing element and p.second will be false, indicating that a new element was not inserted.

The procedure can be described using the following pseudocode:

```
If the tree is empty
{
    Create a new node from the sourceElement;
    Point _root, _head, and _tail to the new node;
    Update the _size;

    Return an iterator pointing to the root and "true";
}
Otherwise
{
    Find the insertion point for the new key value;

    If the insertion point's key value is the same as that of the
        sourceElement
    {
        Return an iterator pointing to the insertion point and "false";
    }
Otherwise
{
    Create a new node from the sourceElement, and attach it to the
        insertion point;

    If the new node is the left child of the head,
        point _head to the new node;
    Otherwise, if the new node is the right child of the tail,
        point _tail to the new node;

    Update the _size;
    Return an iterator pointing to the new node and "true";
}
```

The function is defined in the header file memberFunctions\_2.h (lines 8-52). Consider, for example, an empty BinaryTree<int, int, Less<int>>. Inserting the first element (21,0), performs the following operations (mapped values are not shown, only key values):

```

// The tree is empty (lines 14-23)

// Create a new node from the sourceElement (21,0),
// and point _root to the new node (21)

_root = _createNode(sourceElement);

          x   _root
          |   /
          21
         /   \
        x     x

21
x <-p
s-> x

// Point _head and _tail to the new node (21)

_head = _root;
_tail = _root;

          x   _root/_head/_tail
          |   /
          21
         /   \
        x     x

// Update the _size;

++_size;           // _size is 1

// Return an iterator pointing to the new node (21) and "true"

return makePair(iterator(this, _root), true);

```

Inserting a new element (9,0) performs the following operations:

```

// The tree is not empty (lines 24-51)

// Find the insertion point for the new key value (9)

Node* insertionPoint = bt::findInsertionPoint(_root,
    key(sourceElement),
    _predicate);

// insertionPoint points to node 21

// insertionPoint's key value (21) is not equal to sourceElement's key
// value (9), so insert a new node (lines 36-50)

// Create a new node from the sourceElement (9,0), and attach it to the
// insertion point (node 21);

```

```

Node* newNode = _createNode(sourceElement);
bt::attachNewNode(newNode, insertionPoint, _predicate);

          x  _root/_head/_tail
          | 
          21 _____
         /   \ 
        9    x
       / \ 
      x  x

      9  21
x <-p <-p
  s-> s-> x

// If the new node (9) is the left child of the head (node 21),
// point _head to the new node;
// Otherwise, if the new node (9) is the right child of the tail (node 21),
// point _tail to the new node;

if (newNode == _head->left)
    _head = newNode;                      // Point _head to node 9
else if (newNode == _tail->right)
    _tail = newNode;

          x  _root/_tail
          | / 
          21 _____
         / \ 
        -head  9
       / \ 
      x  x

// Update the _size;
++_size;    // _size is 2

// Return an iterator pointing to the new node (9) and "true"
return makePair(iterator(this, newNode), true);

```

Inserting a new element (33,0) performs the following operations:

```

// The tree is not empty (lines 24-51)

// Find the insertion point for the new key value (33)

Node* insertionPoint = bt::findInsertionPoint(_root,
    key(sourceElement),
    _predicate);

// insertionPoint points to node 21

```

```

// insertionPoint's key value (21) is not equal to sourceElement's key
// value (33), so insert a new node (lines 36-50)

// Create a new node from the sourceElement (33,0), and attach it to the
// insertion point (node 21);

Node* newNode = _createNode(sourceElement);
bt::attachNewNode(newNode, insertionPoint, _predicate);

          x  _root/_tail
          |  /
         21  _____ \
         |   |
         33  _____ \
         |   |
         x   x

_head   \ /   21   _____ \
      \ /   |   |   \
      9   33  |   |
      |   |   |   |
      x   x   x   x

x 9  21  33
x <-p <-p <-p
      s-> s-> s-> x

// If the new node (33) is the left child of the head (node 9),
// point _head to the new node;
// Otherwise, if the new node (33) is the right child of the tail (node 21),
// point _tail to the new node;

if (newNode == _head->left)
    _head = newNode;
else if (newNode == _tail->right)
    _tail = newNode;           // Point _tail to node 33

          x  _root
          |  /
         21  _____ \
         |   |
         tail  _____ \
         |   |
         33  _____ \
         |   |
         x   x

_head   \ /   21   _____ \
      \ /   |   |   \
      9   33  |   |
      |   |   |   |
      x   x   x   x

// Update the _size;

++_size;    // _size is 3

// Return an iterator pointing to the new node (33) and "true"

return makePair(iterator(this, newNode), true);

```

Inserting a new element (15,0) performs the following operations:

```

// The tree is not empty (lines 24-51)

// Find the insertion point for the new key value (15)

```

```

Node* insertionPoint = bt::findInsertionPoint(_root,
    key(sourceElement),
    _predicate);

// insertionPoint points to node 9

// insertionPoint's key value (9) is not equal to sourceElement's key
// value (15), so insert a new node (lines 36-50)

// Create a new node from the sourceElement (15,0), and attach it to the
// insertion point (node 9);

Node* newNode = _createNode(sourceElement);
bt::attachNewNode(newNode, insertionPoint, _predicate);

          x   root
          |   /
        21   9
        |   |
      head   15
        |   |
        x   x
        |   |
      9   15   21   33
x <-p <-p   <-p   <-p
      s-> s->   s->   s-> x

// If the new node (15) is the left child of the head (node 9),
// point _head to the new node;
// Otherwise, if the new node (15) is the right child of the tail (node 33),
// point _tail to the new node;

if (newNode == _head->left)
    _head = newNode;
else if (newNode == _tail->right)
    _tail = newNode;           // _head / _tail are unchanged

// Update the _size;

++_size;      // _size is 4

// Return an iterator pointing to the new node (15) and "true"

return makePair(iterator(this, newNode), true);

```

Inserting a new element (39,0) performs the following operations:

```

// The tree is not empty (lines 24-51)

// Find the insertion point for the new key value (39)

```

```

Node* insertionPoint = bt::findInsertionPoint(_root,
    key(sourceElement),
    _predicate);

// insertionPoint points to node 33
// insertionPoint's key value (33) is not equal to sourceElement's key
// value (39), so insert a new node (lines 36-50)

// Create a new node from the sourceElement (39,0), and attach it to the
// insertion point (node 33);

Node* newNode = _createNode(sourceElement);
bt::attachNewNode(newNode, insertionPoint, _predicate);

          x   _root
          |   /
         21
         |   \
         \   tail
        /   /
       33   39
      /   \
      x   39
      |   \
      x   x

      9   15   21   33   39
x <-p <-p <-p <-p <-p
      s-> s-> s-> s-> s-> x

// If the new node (39) is the left child of the head (node 9),
// point _head to the new node;
// Otherwise, if the new node (39) is the right child of the tail (node 33),
// point _tail to the new node;

if (newNode == _head->left)
    _head = newNode;
else if (newNode == _tail->right)
    _tail = newNode;           // Point _tail to node 39

          x   _root
          |   /
         21
         |   \
         \   tail
        /   /
       33   39
      /   \
      x   39
      |   \
      x   x

// Update the _size

++_size;    // _size is 5

// Return an iterator pointing to the new node (39) and "true"

```

```
    return makePair(iterator(this, newNode), true);
```

Attempting to insert an element with a duplicate key, such as (9,1), performs the following operations:

```
// The tree is not empty (lines 24-51)

// Find the insertion point for the new key value (9)

Node* insertionPoint = bt::findInsertionPoint(_root,
    key(sourceElement),
    _predicate);

// insertionPoint points to node 9

// insertionPoint's key value (9) is equal to sourceElement's key value
// (9), so do not insert a new node (lines 30-35)

// Return an iterator pointing to the existing node (9) and "false"

return makePair(iterator(this, insertionPoint), false);
```

The program in this chapter demonstrates the above examples by inserting the elements (21,0), (9,0), (33,0), (15,0), and (39,0) into a `BinaryTree<Traceable<int>, int>`. Line 17 (main.cpp) constructs the tree `t`, and lines 19-31 insert the elements, generating the output

```
v 21 // Construct argument passed to makePair (Traceable<int> from int)
c 21 // Construct argument passed to _alloc's construct method
c 21 // Construct element (key value) in tree
~ 21 // Destroy argument passed to _alloc's construct method
~ 21 // Destroy argument passed to makePair

v 9
c 9
c 9
~ 9
~ 9

v 33
c 33
c 33
~ 33
~ 33

v 15
c 15
c 15
~ 15
~ 15

v 39
c 39
c 39
~ 39
```

~ 39

Note that each of these calls to insert returns an iterator-bool Pair in which first points to the newly inserted element and second is “true.”

Line 34,

```
Pair<BinaryTree::iterator, bool> p = t.insert(makePair(9, 1));
```

attempts to insert a new element (9,1) and stores the result in the iterator-bool Pair p. Because the tree already contains an element with a key value of 9 (element (9,0)), the new element is not inserted and the resultant output is

```
v 9      // Construct argument passed to makePair (Traceable<int> from int)
~ 9      // Destroy argument passed to makePair
```

The call to insert returns a Pair in which first points to the existing element, (9,0), and second is false. Lines 37-38,

```
if (p.second == false)
    cout << "The tree already contains element " << *p.first << endl;
```

therefore generate the output

```
The tree already contains element (9,0)
```

Lines 40-44 verify the predecessor / successor links by iterating through the tree in forward and reverse order, generating the output

```
t contains 5 elements:
(9,0) (15,0) (21,0) (33,0) (39,0)
```

```
The reverse sequence is:
(39,0) (33,0) (21,0) (15,0) (9,0)
```

In line 47,

```
traverseInOrder(t.root(), PrintBtNode<BinaryTree::Node>());
```

the expression

```
PrintBtNode<BinaryTree::Node>()
```

constructs an unnamed object of type

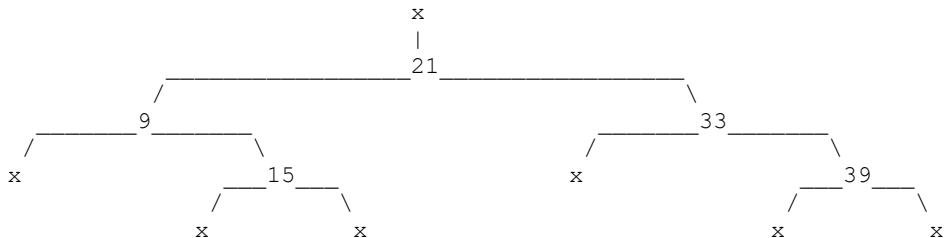
```
PrintBtNode<BinaryTree<Traceable<int>, int>::Node>
```

which is passed to traverseInOrder. The compiler generates the function

```
void traverseInOrder(const BinaryTree<Traceable<int>, int>::Node* root,
PrintBtNode<BinaryTree<Traceable<int>, int>::Node> visit);
```

The call to `traverseInOrder` verifies the remaining structure of the tree, generating the output

```
node 9
  parent 21
  right 15
node 15
  parent 9
node 21
  left 9
  right 33
node 33
  parent 21
  right 39
node 39
  parent 33
```



When main terminates, t is destroyed. t's destructor calls `_destroyAllNodes`, generating the output

```
~ 9    // Destroy key value (const Traceable<int> 9)
~ 15   // Destroy key value (const Traceable<int> 15)
~ 21   // Destroy key value (const Traceable<int> 21)
~ 33   // Destroy key value (const Traceable<int> 33)
~ 39   // Destroy key value (const Traceable<int> 39)
```

## 14.10: Erase

*Source files and folders*

```
BinaryTree/3
BinaryTree/common/memberFunctions_3.h
bt/detachLeafNode.h
bt/detachOneChildNode.h
checkBtIntegrity
```

*Chapter outline*

- Detaching a leaf, 1-child, or 2-child node
- Implementing *BinaryTree*'s *erase* method

Before a node can be destroyed, it must first be detached from the tree by updating the appropriate links. The procedure for detaching a node n depends on whether n has 0, 1, or 2 children.

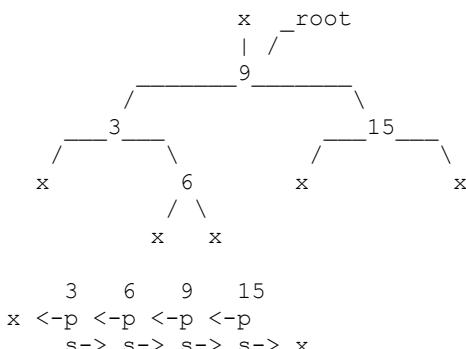
Recall from Chapter 14.2 that a node with 0 children is called a “leaf node.” The function (*detachLeafNode.h*, lines 10-11)

```
template <class Node>
void detachLeafNode(Node* n, Node** root);
```

detaches the leaf node n. If n is the root of the tree, the function also updates the tree's root pointer by setting it to null (lines 13-30). The parameter

```
Node** root // Pointer to a Node*
```

is a pointer to the tree's root pointer. Consider, for example, the tree



If d is a pointer to node 6, the function call

```
detachLeafNode(d, &root);
```

performs the following operations:

```
// n points to node 6
// root points to the pointer _root, not node 9

          x   /   root <-root
          |   /
          9
      /   \   \
     3     15
    / \   / \
   x   x x   x

3   6   9   15
x <-p <-p <-p <-p
s-> s-> s-> s-> x

if (hasPredecessor(n))
    n->predecessor->successor = n->successor;           // Point node 3's successor
                                                               // link to node 9
if (hasSuccessor(n))
    n->successor->predecessor = n->predecessor;         // Point node 9's predecessor
                                                               // link to node 3
3   9   15
x <-p <-p <-p
s-> s-> s-> x

if (isLeftChild(n))
    n->parent->left = nullptr;
else if (isRightChild(n))
    n->parent->right = nullptr;                          // Point node 3's right link
                                                               // to null
else
    *root = nullptr;

n->parent = nullptr;                                     // Point node 6's parent link
                                                               // to null

          x   /   root <-root
          |   /
          9
      /   \   \
     3     15
    / \   / \
   x   x x   x

          x
          |
          6 <-n
    / \
   x   x

// Node 6 can now be destroyed
```

To detach node 3, the function performs the following operations:

```
// n points to node 3

    x   root <-root
    |   /
  n   9
  \ /   \
  3   15
  / \   / \
x   x   x   x

3   9   15
x <-p <-p <-p
s-> s-> s-> x

if (hasPredecessor(n))
    n->predecessor->successor = n->successor;      // Not executed

if (hasSuccessor(n))
    n->successor->predecessor = n->predecessor;    // Point node 9's predecessor
                                                       // link to null
    9   15
x <-p <-p
s-> s-> x

if (isLeftChild(n))
    n->parent->left = nullptr;                      // Point node 9's left link
else if (isRightChild(n))                           // to null
    n->parent->right = nullptr;
else
    *root = nullptr;

n->parent = nullptr;                                // Point node 3's parent link
                                                       // to null

    x   root <-root
    |   /
  9
  / \   \
  /   \   \
x       15
         / \
         x   x

// Node 3 can now be destroyed

    x   n
    |   /
  3
  / \   \
x       x
```

To detach node 15, the function performs the following operations:

```
// n points to node 15
```

```

x   _root <-root
|   /
9   n
/   \
x   15
|   \
x   x

9   15
x <-p <-p
s-> s-> x

if (hasPredecessor(n))
    n->predecessor->successor = n->successor;      // Point node 9's successor
                                                       // link to null
if (hasSuccessor(n))
    n->successor->predecessor = n->predecessor; // Not executed

9
x <-p
s-> x

if (isLeftChild(n))
    n->parent->left = nullptr;
else if (isRightChild(n))
    n->parent->right = nullptr;                      // Point node 9's right link
                                                       // to null
else
    *root = nullptr;

n->parent = nullptr;                                // Point node 15's parent link
                                                       // to null

x   _root <-root
|   /
9
/   \
x   x

x   n
|   /
15
/   \
x   x

```

// Node 15 can now be  
// destroyed

To detach node 9, the function performs the following operations:

```

// n points to node 9

x   _root <-root
|   /
9
/   \
x   x

```

```

9
x <-p
s-> x

if (hasPredecessor(n))
    n->predecessor->successor = n->successor;      // Not executed

if (hasSuccessor(n))
    n->successor->predecessor = n->predecessor;    // Not executed

if (isLeftChild(n))
    n->parent->left = nullptr;
else if (isRightChild(n))
    n->parent->right = nullptr;
else
    *root = nullptr;                                // Point _root to null

n->parent = nullptr;                            // Point node 9's parent link
                                                // null

    x <- _root <-root

        x   n
        |   /
        9
        /   —————— \
        x           x

```

Note that the expression (line 27)

```
*root // The referent of root
```

accesses the tree's root pointer (\_root), not node 9. The statement

```
*root = nullptr;
```

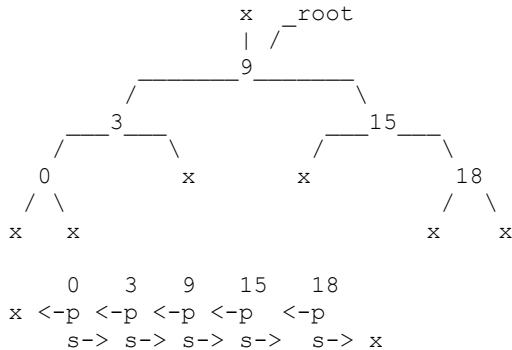
is therefore equivalent to

```
_root = nullptr;
```

The function (detachOneChildNode.h, lines 10-11)

```
template <class Node>
void detachOneChildNode(Node* n, Node** root);
```

detaches the 1-child node n. If n is the root of the tree, the function also updates the tree's root pointer by pointing it to the new root (lines 13-41). Consider, for example, the tree



If d is a pointer to node 3, the function call

```
detachOneChildNode(d, & root);
```

performs the following operations:

```

// n points to node 3
// root points to the pointer _root, not node 9

if (hasPredecessor(n))
    n->predecessor->successor = n->successor;           // Point node 0's successor
                                                               // link to node 9

if (hasSuccessor(n))
    n->successor->predecessor = n->predecessor;         // Point node 9's predecessor
                                                               // link to node 0

    0   9   15   18
x <-p <-p <-p   <-p
    s-> s-> s-> s-> x

Node* nOrphan;

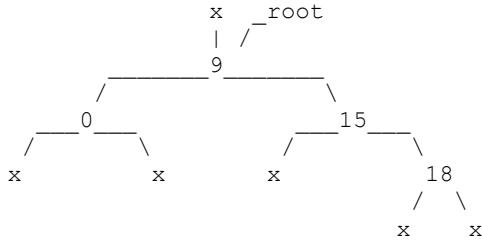
if (hasLeftChild(n))
    nOrphan = n->left;                                         // Node 3's orphan is node 0
else
    nOrphan = n->right;

nOrphan->parent = n->parent;                                // Point node 0's parent link
                                                               // to node 9

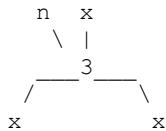
if (isLeftChild(n))
    n->parent->left = nOrphan;                               // Point node 9's left link
else if (isRightChild(n))
    n->parent->right = nOrphan;
else
    *root = nOrphan;

n->parent = nullptr;                                         // Point node 3's parent and
n->left = nullptr;                                           // child links to null
n->right = nullptr;

```



```
// Node 3 can now be destroyed
```



To detach node 15, the function performs the following operations:

```

// n points to node 15

if (hasPredecessor(n))
    n->predecessor->successor = n->successor;      // Point node 9's successor
                                                       // link to node 18

if (hasSuccessor(n))
    n->successor->predecessor = n->predecessor;    // Point node 18's predecessor
                                                       // link to node 9

    0      9      18
x <-p <-p <-p
      s-> s-> s-> x

Node* nOrphan;

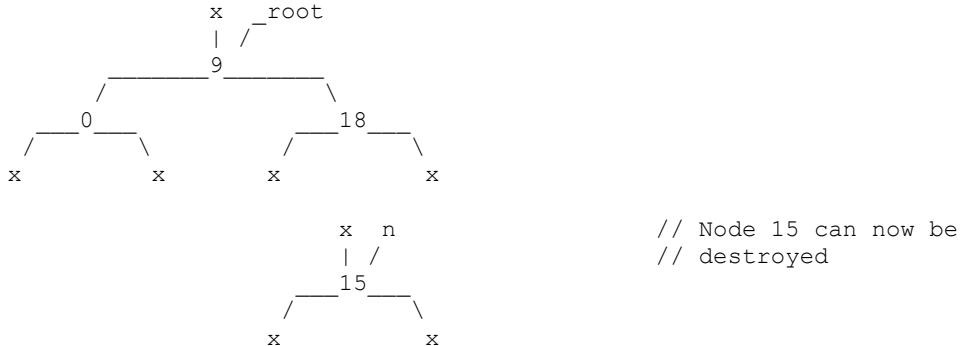
if (hasLeftChild(n))
    nOrphan = n->left;
else
    nOrphan = n->right;                           // Node 15's orphan is node 18

nOrphan->parent = n->parent;                     // Point node 18's parent link
                                                       // to node 9

if (isLeftChild(n))
    n->parent->left = nOrphan;
else if (isRightChild(n))
    n->parent->right = nOrphan;                  // Point node 9's right link
else
    *root = nOrphan;

n->parent = nullptr;                             // Point node 15's parent and
n->left = nullptr;                            // child links to null
n->right = nullptr;

```



Node 18, which is now a leaf, can be detached via `detachLeafNode`:



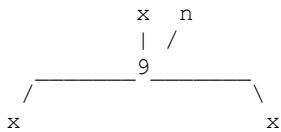
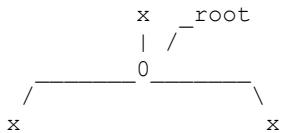
To detach node 9, which now has 1 child, `detachOneChildNode` performs the following operations:

```

if (isLeftChild(n))
    n->parent->left = nOrphan;
else if (isRightChild(n))
    n->parent->right = nOrphan;
else
    *root = nOrphan;                                // Point _root to node 0

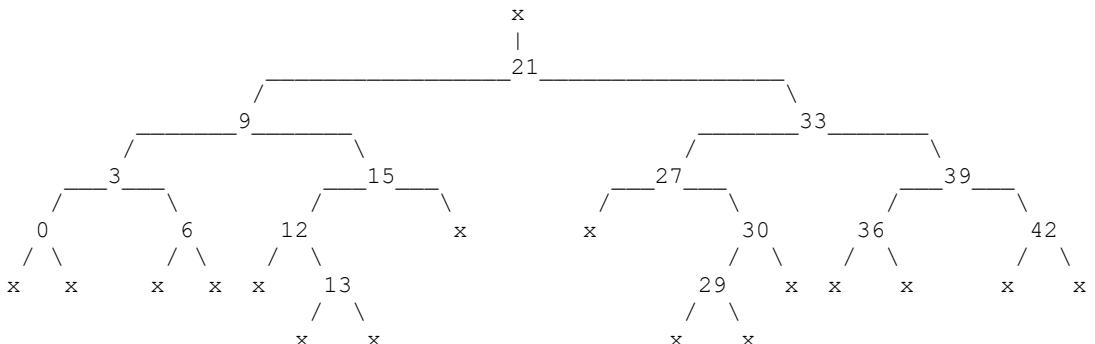
n->parent = nullptr;
n->left = nullptr;
n->right = nullptr;                               // Point node 9's parent and
                                                // child links to null

```



// Node 9 can now be destroyed

If a node has 2 children, its in-order predecessor (or successor) is guaranteed to be either a leaf or 1-child node:



```

// Node 3 has 2 children
//   Its predecessor (node 0) is a leaf
//   Its successor (node 6) is a leaf

// Node 9 has 2 children
//   Its predecessor (node 6) is a leaf
//   Its successor (node 12) has 1 child

// Node 21 has 2 children
//   Its predecessor (node 15) has 1 child
//   Its successor (node 27) has 1 child

```

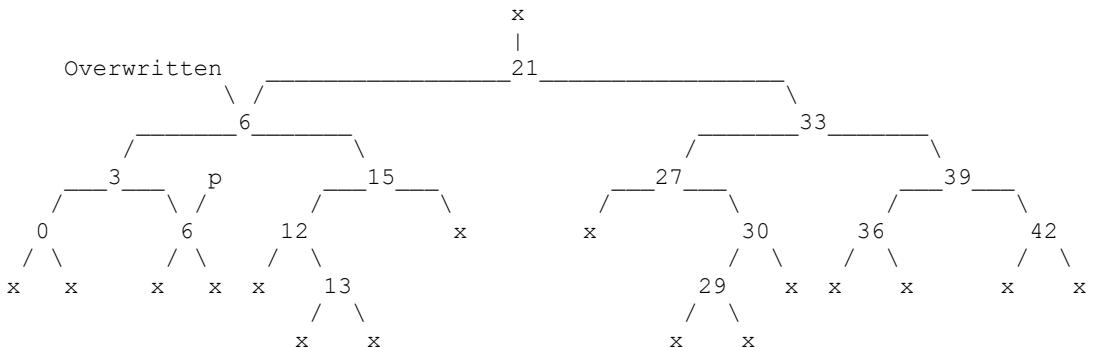
```
// Node 33 has 2 children
//   Its predecessor (node 30) has 1 child
//   Its successor (node 36) is a leaf

// Node 39 has 2 children
//   Its predecessor (node 36) is a leaf
//   Its successor (node 42) is a leaf
```

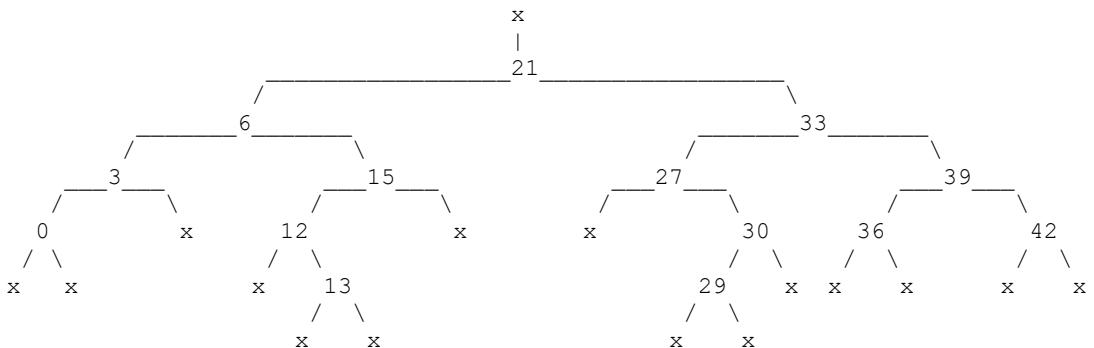
A 2-child node n can therefore be detached by overwriting its element with that of its predecessor p, then detaching p via detachLeafNode / detachOneChildNode.

To detach node 9, for example:

```
// Overwrite node 9's element with that of its predecessor p (node 6)
```

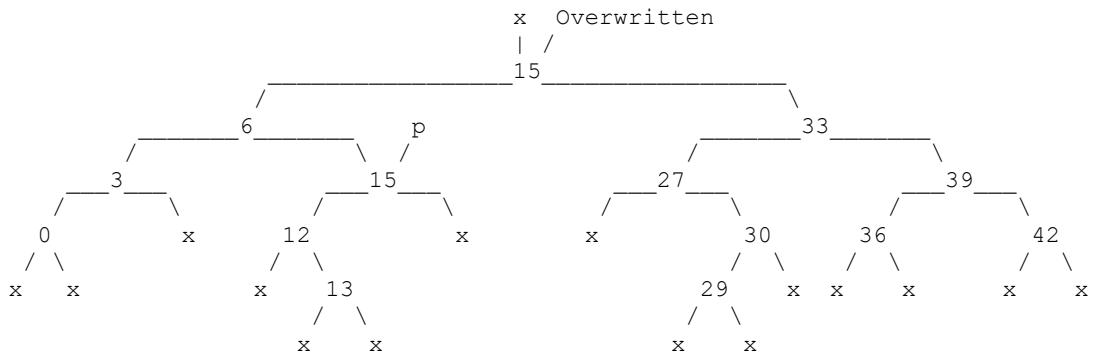


```
// Detach p via detachLeafNode
```

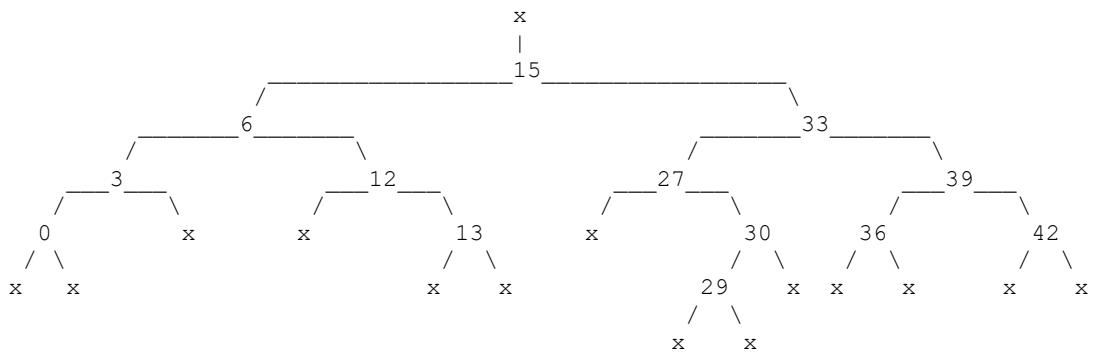


To detach node 21:

```
// Overwrite node 21's element with that of its predecessor p (node 15)
```

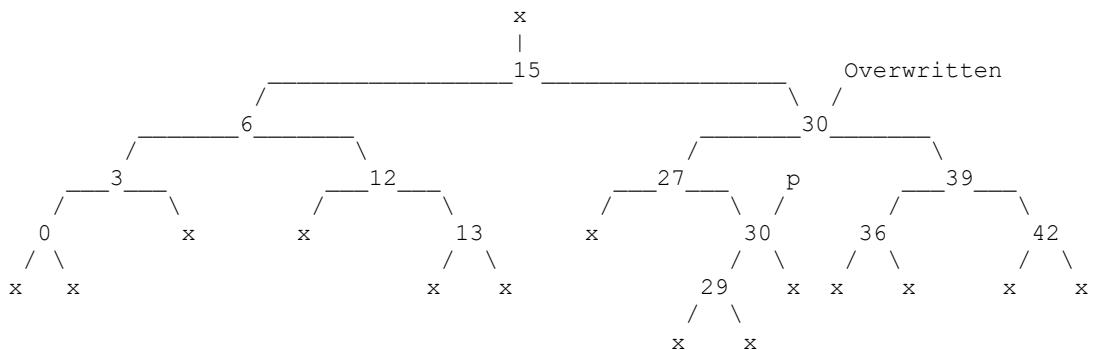


```
// Detach p via detachOneChildNode
```

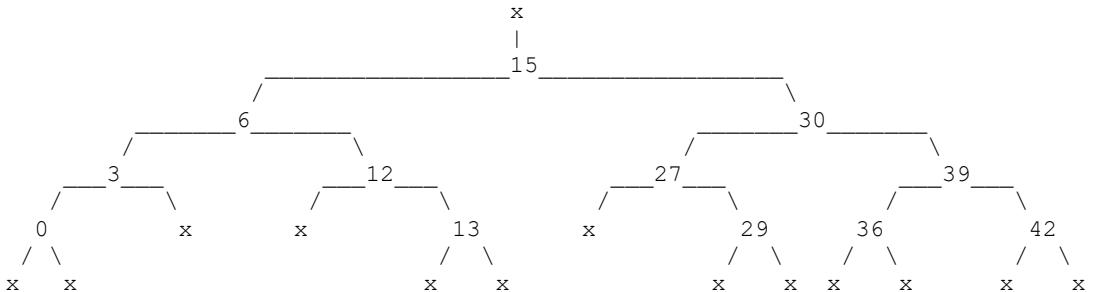


To detach node 33:

```
// Overwrite node 33's element with that of its predecessor p (node 30)
```

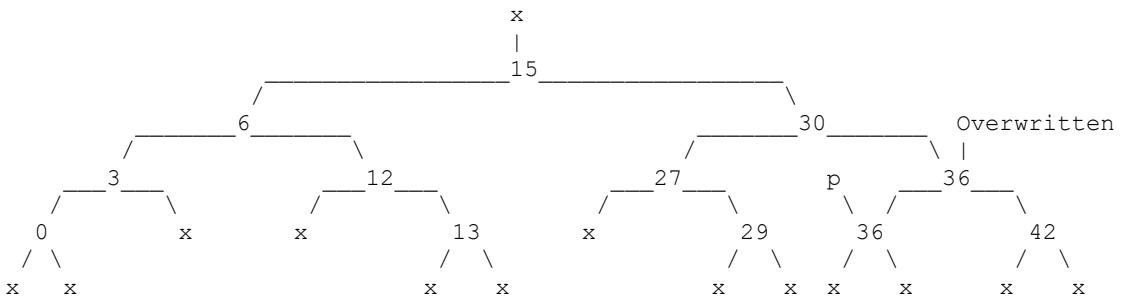


```
// Detach p via detachOneChildNode
```

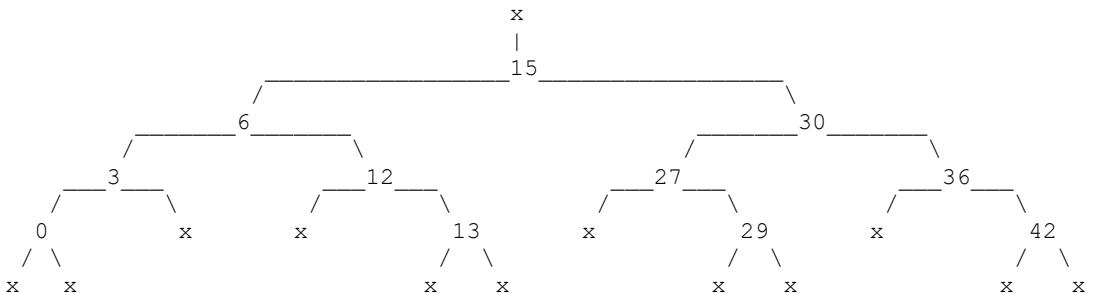


To detach node 39:

```
// Overwrite node 39's element with that of its predecessor p (node 36)
```



```
// Detach p via detachLeafNode
```



The private member function (BinaryTree.h, line 67)

```
void _overwriteElement(Node* n, const value_type& sourceElement);
```

overwrites n's element (key-mapped pair) with the given sourceElement. Note, however, that because key values are const, the function cannot simply assign the new key value to n:

```
n->element.second = sourceElement.second; // Set n's mapped value to that
// of the source element
```

```
// Ok: n->element.second is not const, so its value can be modified
```

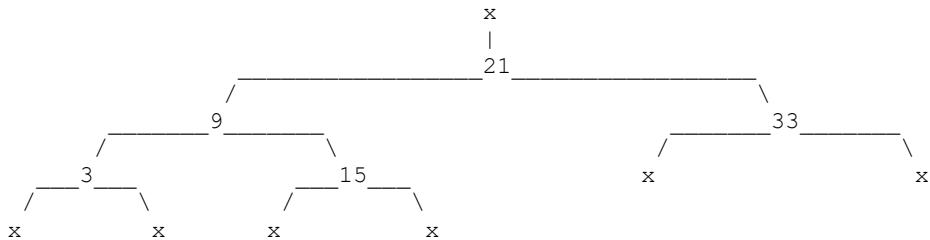
```

n->element.first = sourceElement.first;      // Set n's key value to that of
                                              // the source element

// Compiler error: n->element.first is const, so its value cannot be modified

```

To overwrite the node's key value, the function must therefore replace the entire node with a new node containing the new key value (memberFunctions\_3.h, lines 39-53). Consider, for example, the tree



If p is a pointer to node 9, the function call

```
_overwriteElement(p, p->predecessor->element);
```

overwrites node 9's element with that of node 3 by performing the following operations:

```

// n points to node 9
// sourceElement is node 3's element

// Construct a temporary node containing the new element...

Node temp(sourceElement);

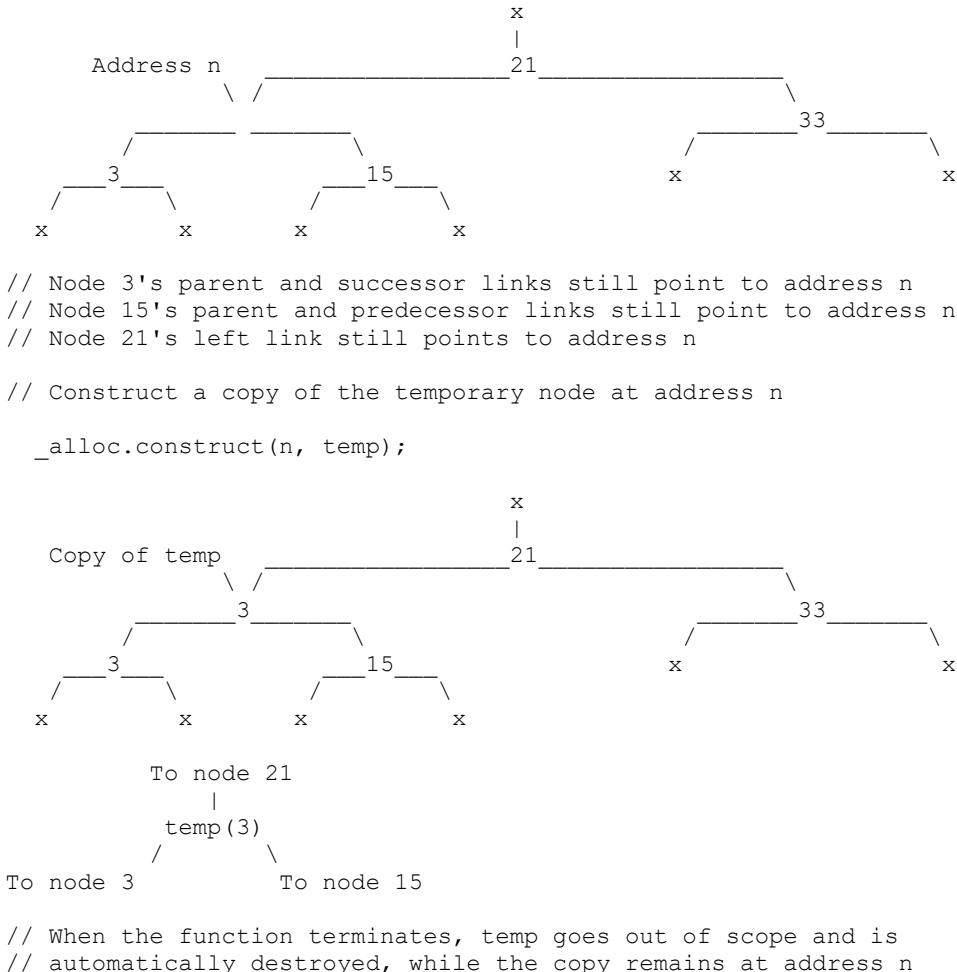
// ...and point all of its links to the same nodes as n's links

temp.parent = n->parent;
temp.left = n->left;
temp.right = n->right;
temp.predecessor = n->predecessor;
temp.successor = n->successor;

To node 21
|
temp(3)
/
To node 3      To node 15
          temp(3)
To node 3  <-p
          s->      To node 15

// Destroy the node at address n

_alloc.destroy(n);
  
```



detachLeafNode, detachOneChildNode, and \_overwriteElement can now be combined to implement BinaryTree's erase method. The member function (BinaryTree.h, line 61)

```
iterator erase(const_iterator erasurePoint);
```

removes the element at erasurePoint and returns an iterator pointing to the next element in the sequence (the element proceeding the erased element). The pseudocode for the function is

```

template <class Key, class Mapped, class Predicate>
typename BinaryTree<Key, Mapped, Predicate>::iterator
BinaryTree<Key, Mapped, Predicate>::erase(const_iterator erasurePoint)
{
    Node* trash = a pointer to the node to be erased;
    Node* trashSuccessor = a pointer to the next node;

    If trash has 2 children
    {
        Overwrite trash->element with that of trash->predecessor;

        Point trash to its predecessor, which is guaranteed to be either a
            leaf or 1-child node;
    }

    If trash is the head,
        Point _head to _head's successor, which will become the new head;

    If trash is the tail,
        Point _tail to _tail's predecessor, which will become the new tail;

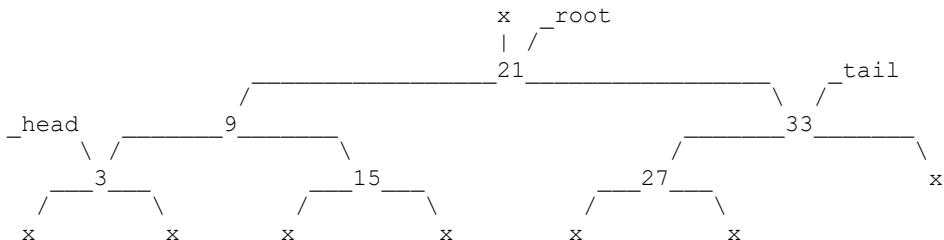
    If trash is a leaf node,
        detach it via detachLeafNode;
    Otherwise,
        detach it via detachOneChildNode;

    Destroy trash and update the _size;

    Return an iterator pointing to the next element (an iterator containing
        trashSuccessor);
}

```

The function is defined in lines 7-37 (memberFunctions\_3.h). Consider, for example, the tree



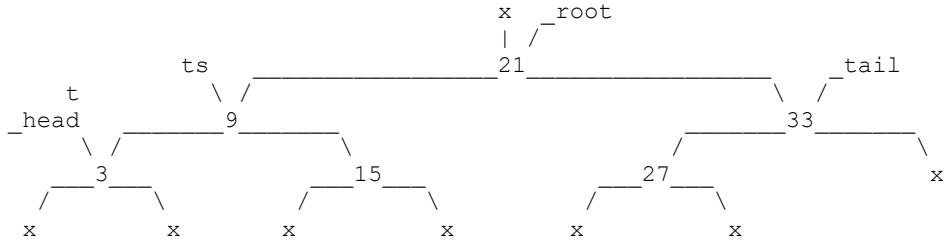
The function erases node 3 by performing the following operations:

```

Node* trash = erasurePoint._i._node;           // trash points to node 3
Node* trashSuccessor = trash->successor;      // trashSuccessor points to
                                               // node 9

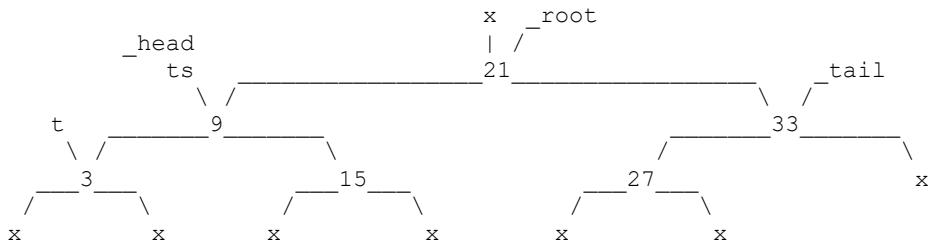
// t denotes trash
// ts denotes trashSuccessor

```



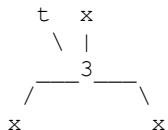
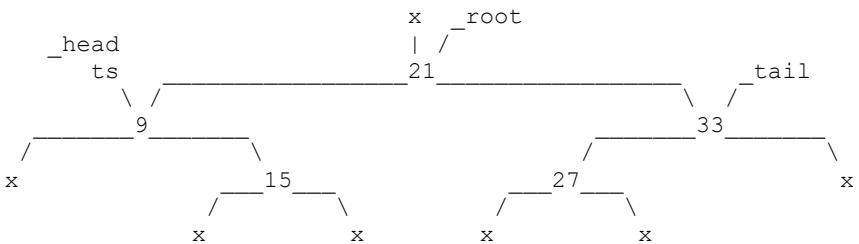
```
// If trash is the head, point _head to _head's successor, which will  
// become the new head
```

```
if (trash == _head)
    _head = _head->successor;
```



```
// If trash is a leaf, detach it via detachLeafNode
```

```
if (isLeaf(trash))  
    detachLeafNode(trash, &_root);
```



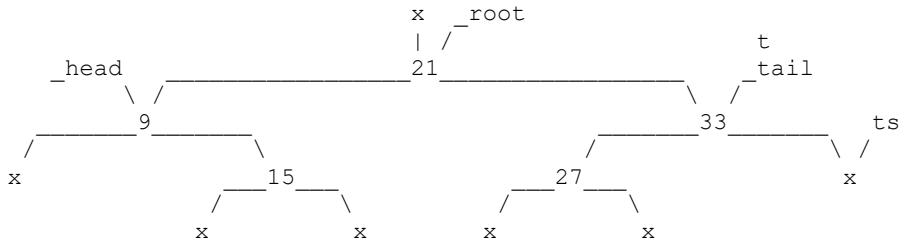
```
// Destroy trash and update the _size;  
// Return an iterator pointing to the next element
```

```
_destroyNode(trash);  
-- size; // size is 5
```

```
return iterator(this, trashSuccessor); // Points to node 9
```

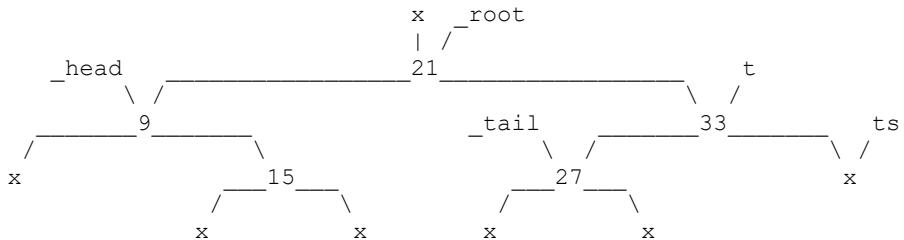
The function erases node 33 by performing the following operations:

```
Node* trash = erasurePoint._i._node;           // trash points to node 33
Node* trashSuccessor = trash->successor;       // trashSuccessor points to
// null
```



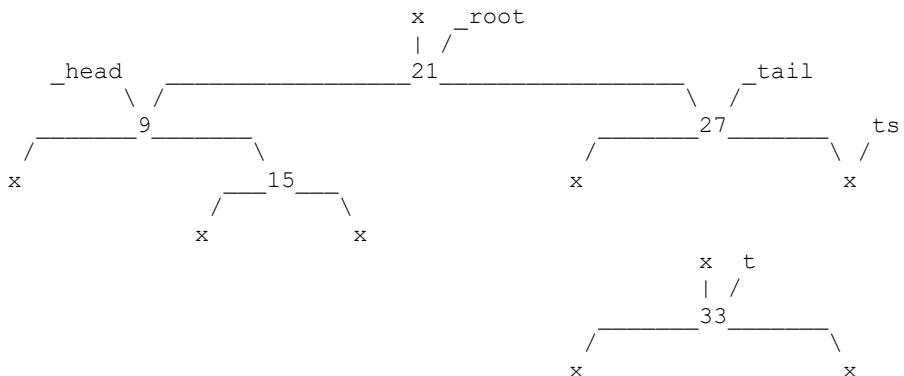
```
// If trash is the tail, point _tail to _tail's predecessor, which will
// become the new tail
```

```
if (trash == _tail)
    _tail = _tail->predecessor;
```



```
// If trash is a leaf, detach it via detachLeafNode; otherwise, detach it
// via detachOneChildNode
```

```
if (isLeaf(trash))
    detachLeafNode(trash, &_root);
else
    detachOneChildNode(trash, &_root);
```



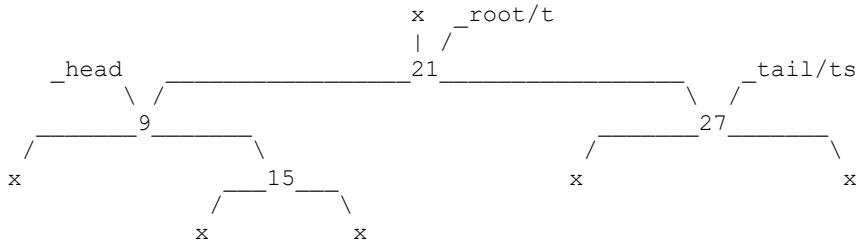
```
// Destroy trash and update the _size;
// Return an iterator pointing to the next element

    _destroyNode(trash);
    --_size;                                // _size is 4

    return iterator(this, trashSuccessor);    // Points to null (the one-past-
                                                // the-last element)
```

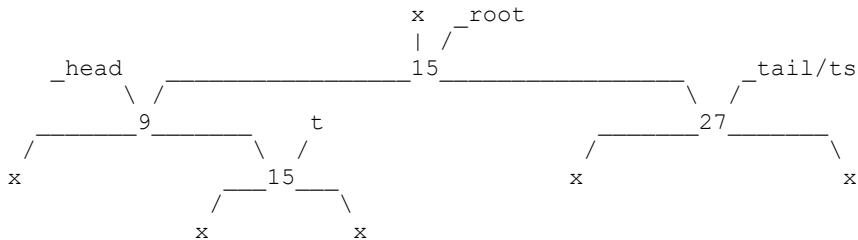
The function erases node 21 by performing the following operations:

```
Node* trash = erasurePoint.i._node;          // trash points to node 21
Node* trashSuccessor = trash->successor;     // trashSuccessor points to
                                              // node 27
```



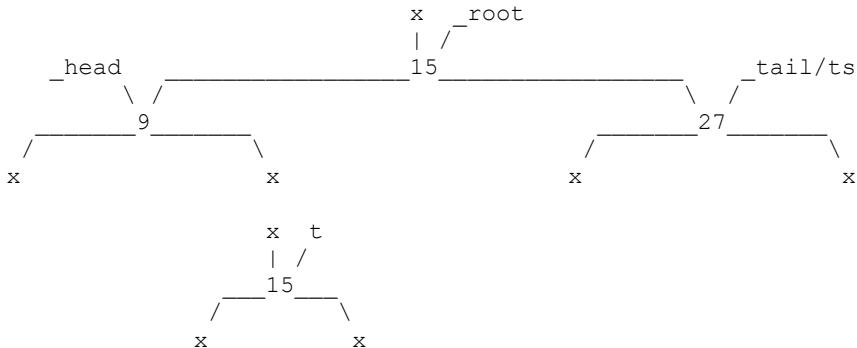
```
// Overwrite trash->element with that of trash->predecessor
// Point trash to trash->predecessor, which is guaranteed to be either
// a leaf or 1-child node
```

```
if (hasLeftChild(trash) && hasRightChild(trash))
{
    _overwriteElement(trash, trash->predecessor->element);
    trash = trash->predecessor;
}
```



```
// If trash is a leaf, detach it via detachLeafNode
```

```
if (isLeaf(trash))
    detachLeafNode(trash, &_root);
```



```

    _destroyNode(trash);
    --_size;                                // _size is 3

    return iterator(this, trashSuccessor);     // Points to node 27

```

The function (checkBtIntegrity.h, lines 11-12)

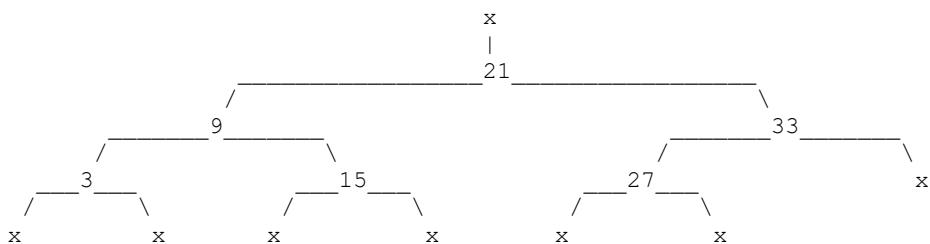
```

template <class Tree, class Function>
void checkBtIntegrity(const Tree& tree, Function printNode);

```

checks the integrity of the given binary tree by printing each element in forward and reverse order, then printing each node using the given function object (lines 14-25).

The program in this chapter demonstrates the above examples. Lines 16-18 (main.cpp) construct a `BinaryTree<Traceable<int>, int>` `t`, an iterator `i`, and a `PrintBtNode` function object `printNode`. Lines 20-25 insert the elements `{(21,0), (9,0), (33,0), (3,0), (15,0), (27,0)}`, resulting in the layout



Lines 28-31 erase element (3,0), generating the output

```

Erasing element (3,0)...
~ 3                                         // Destroy node 3
The next element is (9,0)

(9,0) (15,0) (21,0) (27,0) (33,0)      // checkBtIntegrity
(33,0) (27,0) (21,0) (15,0) (9,0)

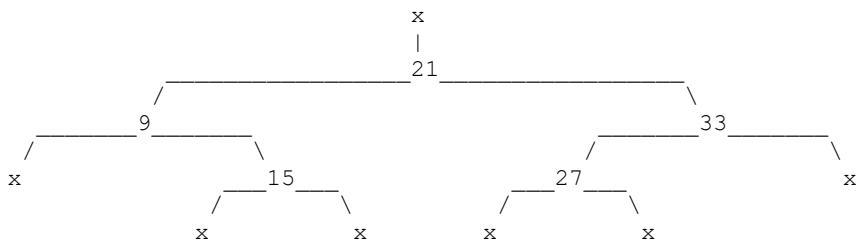
node 9                                         // Refer to the diagram on the next page
parent 21

```

```

    right 15
node 15
    parent 9
node 21
    left 9
    right 33
node 27
    parent 33
node 33
    parent 21
left 27

```



Lines 33-40 erase element (33,0), generating the output

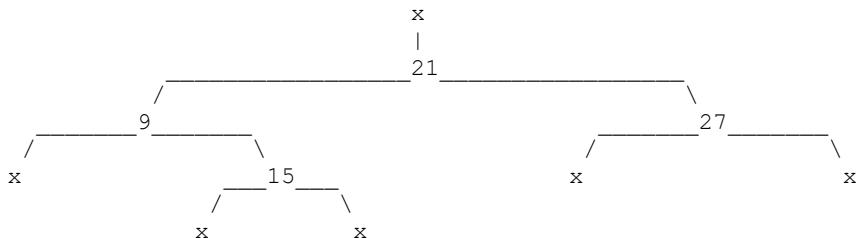
```

v 33                                // Construct argument passed to find
~ 33                                // Destroy argument
Erasing element (33,0)...
~ 33                                // Destroy node 33
The next element is null

(9,0) (15,0) (21,0) (27,0)      // checkBtIntegrity
(27,0) (21,0) (15,0) (9,0)

node 9
    parent 21
    right 15
node 15
    parent 9
node 21
    left 9
    right 27
node 27
    parent 21

```



Lines 42-46 erase element (21,0), generating the output

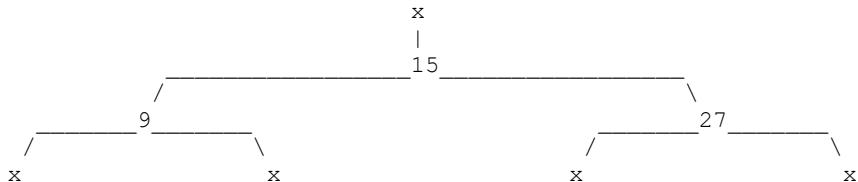
```

v 21                      // Construct argument passed to find
~ 21                      // Destroy argument
Erasing element (21,0)...
c 15                      // Construct temp from predecessor's element
~ 21                      // Destroy node 21
c 15                      // Construct copy of temp in node 21's place
~ 15                      // Destroy temp
~ 15                      // Destroy the predecessor (node 15)
The next element is (27,0)

(9,0) (15,0) (27,0)          // checkBtIntegrity
(27,0) (15,0) (9,0)

node 9
  parent 15
node 15
  left 9
  right 27
node 27
  parent 15

```



When main terminates, t is destroyed. t's destructor destroys the remaining nodes, generating the output

```

~ 9
~ 15
~ 27

```



## 14.11: Clear, Copy, and Assignment

*Source files and folders*

*BinaryTree/4*  
*BinaryTree/common/memberFunctions\_4.h*

*Chapter outline*

- Using recursion to create a copy of an existing tree
- Implementing *BinaryTree*'s clear method, copy constructor, and assignment operator

The member function (*BinaryTree.h*, lines 64)

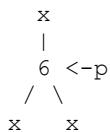
```
void clear();
```

removes all elements from the tree. The function simply calls *\_destroyAllNodes*, points the *\_root*, *\_head*, and *\_tail* to null, and resets the *\_size* to 0 (*memberFunctions\_4.h*, lines 45-54).

The private member function (*BinaryTree.h*, line 67)

```
Node* _copyNode(Node* sourceNode, Node* newParent);
```

creates a copy of the given *sourceNode*, where *newParent* is the parent of the new node (i.e. the parent of the copy). The function then returns a pointer to the new node. The procedure, which is defined recursively, creates a copy of not only the source node, but the entire tree rooted at the source node (*memberFunctions\_4.h*, lines 56-74). Consider, for example, the tree



where *p* is a pointer to node 6 (the source node). The function call

```
Node* q = _copyNode(p, nullptr); // Create a copy of the tree rooted at
                                // p, and point q to the root of the new
                                // tree
```

performs the following operations:

```
// Create a new node from sourceNode->element
Node* newNode = _createNode(sourceNode->element);
```

```

x
|
6 <-n      6 <-s    // n denotes newNode, s denotes sourceNode
 / \
x   x

// Point the new node's parent link to newParent (null)

newNode->parent = newParent;

x      x
|      |
6 <-n      6 <-s
 / \
x   x

// Point the new node's left link to a copy of the source node's left child

newNode->left = _copyNode(sourceNode->left, newNode);

        // Create a copy of sourceNode->left
        // newNode is the parent of the copy

        // Within this call to _copyNode, sourceNode is null,
        // so the function simply returns null, after which
        // newNode->left is set to null

x      x
|      |
6 <-n      6 <-s
 / \     / \
x       x   x

// Point the new node's right link to a copy of the source node's right child

newNode->right = _copyNode(sourceNode->right, newNode);

        // Create a copy of sourceNode->right
        // newNode is the parent of the copy

        // Within this call to _copyNode, sourceNode is null,
        // so the function simply returns null, after which
        // newNode->right is set to null

x      x
|      |
6 <-n      6 <-s
 / \     / \
x   x     x   x

// Return a pointer to the the new node 6

return newNode;

```

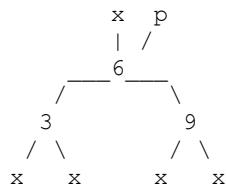
The above example can be summarized as

```
q = _copyNode(source6, nullptr)
{
    new6 = _createNode(source6->element);

    new6->parent = nullptr;
    new6->left = _copyNode(source6->left, new6)
    {
        return nullptr;
    }
    new6->right = _copyNode(source6->right, new6)
    {
        return nullptr;
    }

    return new6;
}
```

Given the tree

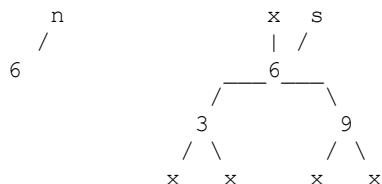


the function call

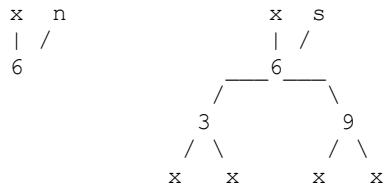
```
Node* q = _copyNode(p, nullptr);
```

performs the following operations:

```
Node* newNode = _createNode(sourceNode->element);
```



```
newNode->parent = newParent;
```



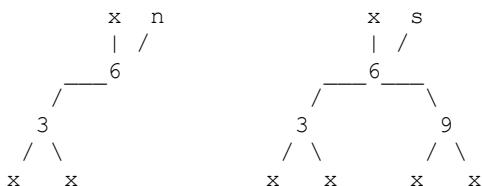
```

newNode->left = _copyNode(sourceNode->left, newNode);

    // Within this call to _copyNode, sourceNode is node 3 and
    // newParent is the new node 6

    // This call creates a new node 3, generates 2 more recursive
    // calls (one for each of the original node 3's null
    // children), then returns a pointer to the new node 3

    // The new node 6's left link (newNode->left) is then pointed
    // to the new node 3
  
```



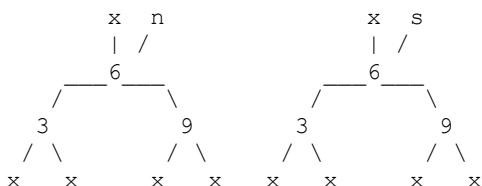
```

newNode->right = _copyNode(sourceNode->right, newNode);

    // Within this call to _copyNode, sourceNode is node 9 and
    // newParent is the new node 6

    // This call creates a new node 9, generates 2 more recursive
    // calls (one for each of the original node 9's null
    // children), then returns a pointer to the new node 9

    // The new node 6's right link (newNode->right) is then
    // pointed to the new node 9
  
```



```
return newNode; // Returns a pointer to the new node 6
```

The above example can be summarized as

```

q = _copyNode(source6, nullptr)
{
    new6 = _createNode(source6->element);

    new6->parent = nullptr;
    new6->left = _copyNode(source6->left, new6)
    {
        new3 = _createNode(source3->element);

        new3->parent = new6;
        new3->left = _copyNode(source3->left, new3)
        {
            return nullptr;
        }
        new3->right = _copyNode(source3->right, new3)
        {
            return nullptr;
        }

        return new3;
    }
    new6->right = _copyNode(source6->right, new6)
    {
        new9 = _createNode(source9->element);

        new9->parent = new6;
        new9->left = _copyNode(source9->left, new9)
        {
            return nullptr;
        }
        new9->right = _copyNode(source9->right, new9)
        {
            return nullptr;
        }

        return new9;
    }

    return new6;
}

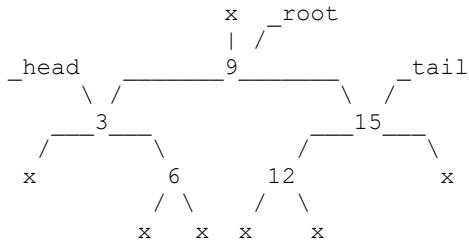
```

The `_copyNode` function is used to implement `BinaryTree`'s copy constructor (`BinaryTree.h`, line 37 / `memberFunctions_4.h`, lines 7-26). Line 10,

```
_root = _copyNode(source._root, nullptr);
```

points `_root` to a copy of the entire source tree. Lines 11-12 then point `_head` / `_tail` to the leftmost / rightmost nodes of the new tree, and line 13 initializes the `_size` to the same value as that of the source object.

Lines 15-25 then set the predecessor and successor links of each node in the new tree, from head to tail. Given the tree



for example, the loop performs 5 iterations:

```

Node* currentNode = _head;           // currentNode is 3
Node* previousNode = nullptr;       // previousNode is null

```

Iteration 1:

```

currentNode->predecessor = previousNode;
currentNode->successor = inOrderSuccessor(currentNode);

```

```

3   6
x <-p   p
s-> s

```

```

previousNode = currentNode;          // previousNode is 3
currentNode = currentNode->successor; // currentNode is 6

```

Iteration 2:

```

currentNode->predecessor = previousNode;
currentNode->successor = inOrderSuccessor(currentNode);

```

```

3   6   9
x <-p <-p   p
s-> s-> s

```

```

previousNode = currentNode;          // previousNode is 6
currentNode = currentNode->successor; // currentNode is 9

```

Iteration 3:

```

currentNode->predecessor = previousNode;
currentNode->successor = inOrderSuccessor(currentNode);

```

```

3   6   9   12
x <-p <-p <-p   p
s-> s-> s-> s

```

```

previousNode = currentNode;          // previousNode is 9
currentNode = currentNode->successor; // currentNode is 12

```

Iteration 4:

```

currentNode->predecessor = previousNode;

```

```

currentNode->successor = inOrderSuccessor(currentNode);

      3   6   9   12   15
x <-p <-p <-p <-p     p
      s-> s-> s-> s-> s

previousNode = currentNode;           // previousNode is 12
currentNode = currentNode->successor; // currentNode is 15

```

Iteration 5:

```

currentNode->predecessor = previousNode;
currentNode->successor = inOrderSuccessor(currentNode);

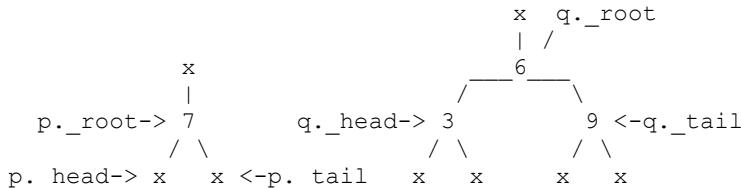
      3   6   9   12   15
x <-p <-p <-p <-p <-p
      s-> s-> s-> s-> s-> x

previousNode = currentNode;           // previousNode is 15
currentNode = currentNode->successor; // currentNode is null

// Terminate loop

```

BinaryTree's assignment operator (BinaryTree.h, line 60) is implemented using the copy constructor (memberFunctions\_4.h, lines 28-43). Consider, for example, the trees p and q



The expression

```
p = q    // p.operator=(q)
```

performs the following operations:

```

// Clear the left operand (p)
clear();

// Construct a temporary BinaryTree as a copy of the right operand (q),
// using the copy constructor

BinaryTree temp(rhs);           // temp is a copy of rhs (q)

```

```

x   temp._root           x   q._root
|   /                         |   /
6   \                         6   \
temp._head-> 3             q._head   \ /   q._tail
/ \   / \                     / \   / \
x   x   x   x                 x   x   x   x
p._root-> x     temp._head-> 3   9 <-temp._tail   3   9
p._head-> x           / \         / \           / \   / \
p._tail-> x           x   x       x   x           x   x   x

// "Steal" temp's elements by swapping the root, head, and tail pointers
// of the left operand (p) and temp

swap(_root, temp._root);    // Swap the addresses to which p._root and
                            // temp._root point

swap(_head, temp._head);    // Swap the addresses to which p._head and
                            // temp._head point

swap(_tail, temp._tail);    // Swap the addresses to which p._tail and
                            // temp._tail point

x   p._root           x   q._root
|   /                         |   /
6   \                         6   \
temp._root-> x     p._head-> 3   9 <-p._tail   3   9
temp._head-> x           / \         / \           / \   / \
temp._tail-> x           x   x       x   x           x   x   x

// Update the _size of the left operand

_size = temp._size;        // Set p._size to 3

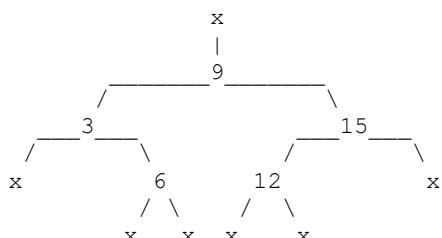
// Return a reference to the left operand (p)

return *this;

```

When the function terminates, temp goes out of scope and its destructor is automatically called. Setting temp's pointers to null (lines 36-38) is necessary to prevent temp's destructor from destroying the stolen elements (which are now “owned” by the left operand, p).

The program in this chapter demonstrates copy and assignment. Lines 18-23 (main.cpp) construct the `BinaryTree<Traceable<int>, int>` a,



Line 26 constructs another tree b as a copy of a, generating the output

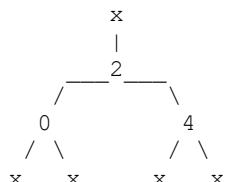
```
c 9      // Construct argument passed to _alloc.construct
c 9      // Construct copy of a's node 9
~ 9      // Destroy argument passed to _alloc.construct
c 3
c 3
~ 3
c 6
c 6
~ 6
c 15
c 15
~ 15
c 12
c 12
~ 12
```

Line 28 then checks the integrity of b, generating the output

```
(3,0) (6,0) (9,0) (12,0) (15,0)
(15,0) (12,0) (9,0) (6,0) (3,0)
```

```
node 3
  parent 9
  right 6
node 6
  parent 3
node 9
  left 3
  right 15
node 12
  parent 15
node 15
  parent 9
  left 12
```

Lines 30-33 construct the tree c,



and line 36 sets a equal to c, generating the output

```
~ 3      // Clear a
~ 6
~ 9
```

```

~ 12
~ 15
c 2      // Construct temp as a copy of c...
c 2
~ 2
c 0
c 0
~ 0
c 4
c 4
~ 4
        // ...after which a takes ownership of temp's elements by swapping
        // the pointers (_root, _head, _tail) of a and temp

```

Line 38 then checks the integrity of a, generating the output

```
(0,0) (2,0) (4,0)
(4,0) (2,0) (0,0)
```

```

node 0
    parent 2
node 2
    left 0
    right 4
node 4
    parent 2

```

When main terminates, the BinaryTrees are destroyed, generating the output

```

~ 0      // c's destructor
~ 2
~ 4
~ 3      // b's destructor
~ 6
~ 9
~ 12
~ 15
~ 0      // a's destructor
~ 2
~ 4

```

# Part 15: Balanced Binary Trees

## 15.1: Rotating Nodes

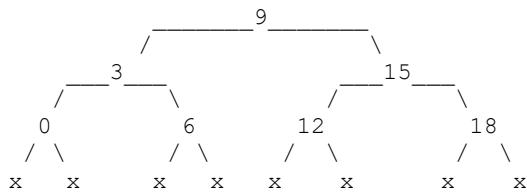
*Source files and folders*

*BinaryTree/5  
BinaryTree/common/memberFunctions\_5.h  
bt/rotate.h*

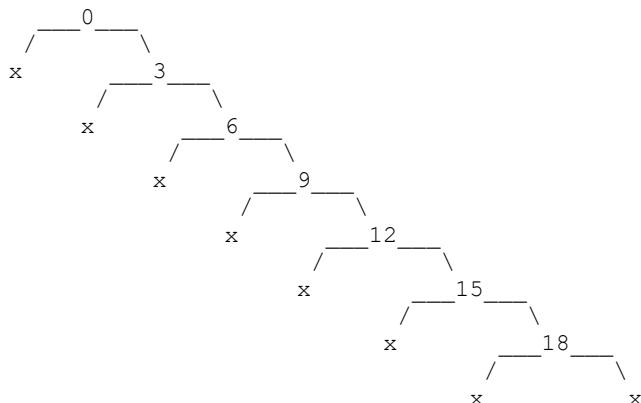
*Chapter outline*

- Reducing the height of a tree by rotating nodes

The order in which elements are inserted into a tree can significantly affect the efficiency of subsequent search operations. Consider, for example, the key values  $\{0, 3, 6, 9, 12, 15, 18\}$ . Inserting them in the order  $\{9, 15, 3, 12, 0, 18, 6\}$  creates a tree that is 3 levels high:

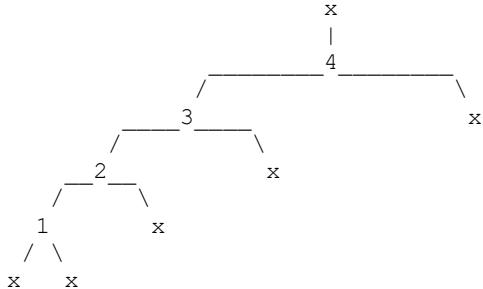


Any element can therefore be retrieved by traversing at most 3 nodes. Inserting the same key values in the order  $\{0, 3, 6, 9, 12, 15, 18\}$ , however, creates a tree that is 7 levels high:

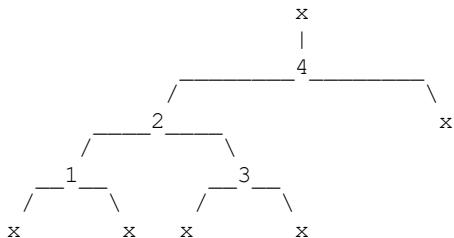


Retrieving an element from this tree can therefore require up to 7 traversals.

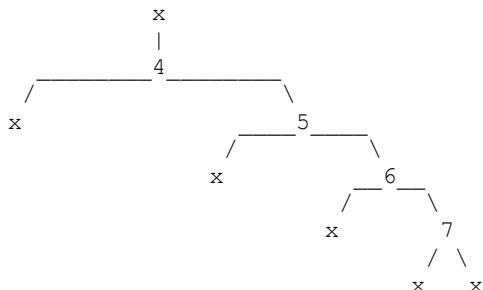
The height of a tree can be reduced by rotating one or more nodes. Rotating a non-root node n moves n up one level and n's parent down one level. Consider, for example, the tree



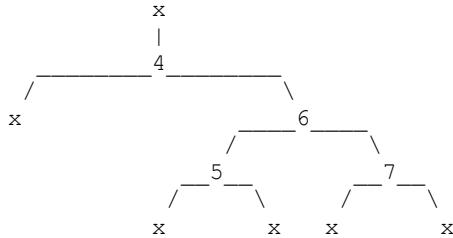
which is 4 levels high. Node 2 is the left child of its parent, so it can be rotated right (clockwise). This moves node 2 up one level, node 3 down one level, and reduces the height of the tree to 3 levels:



Similarly, given the tree

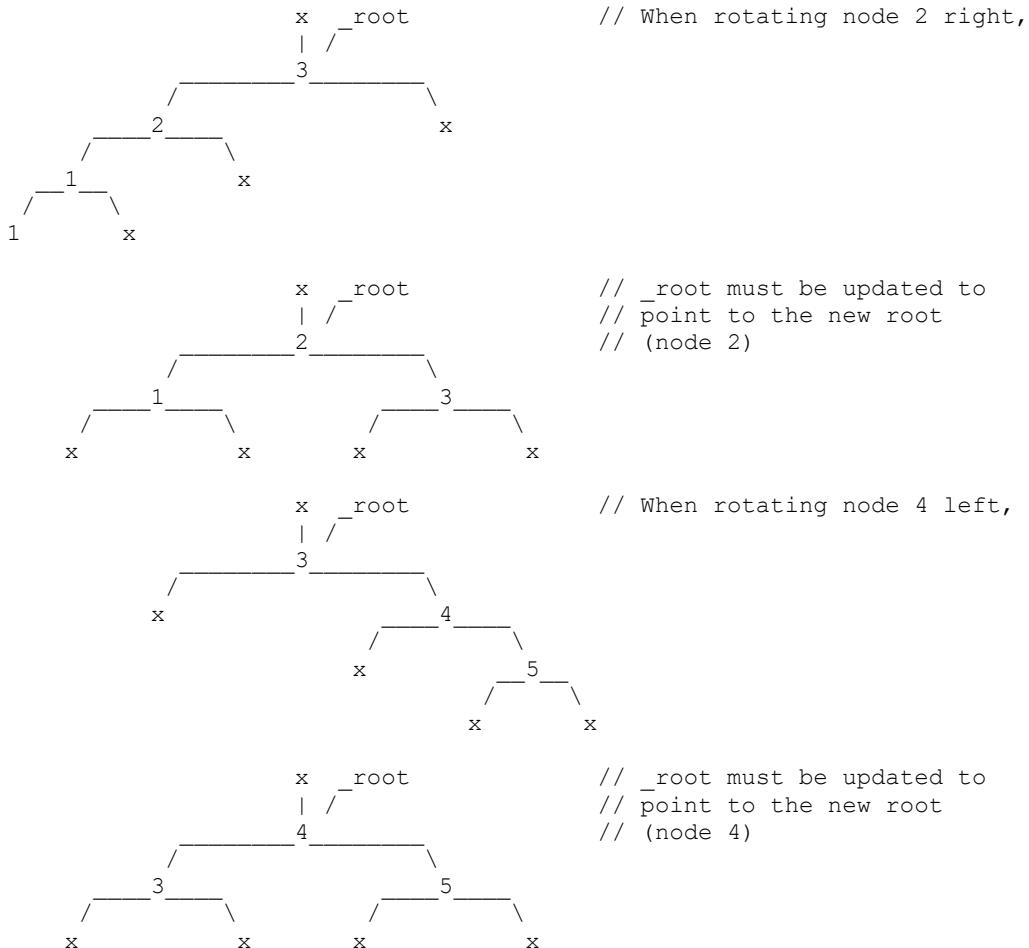


node 6 is the right child of its parent, so it can be rotated left (counterclockwise). This moves node 6 up one level, node 5 down one level, and reduces the height of the tree to 3 levels:

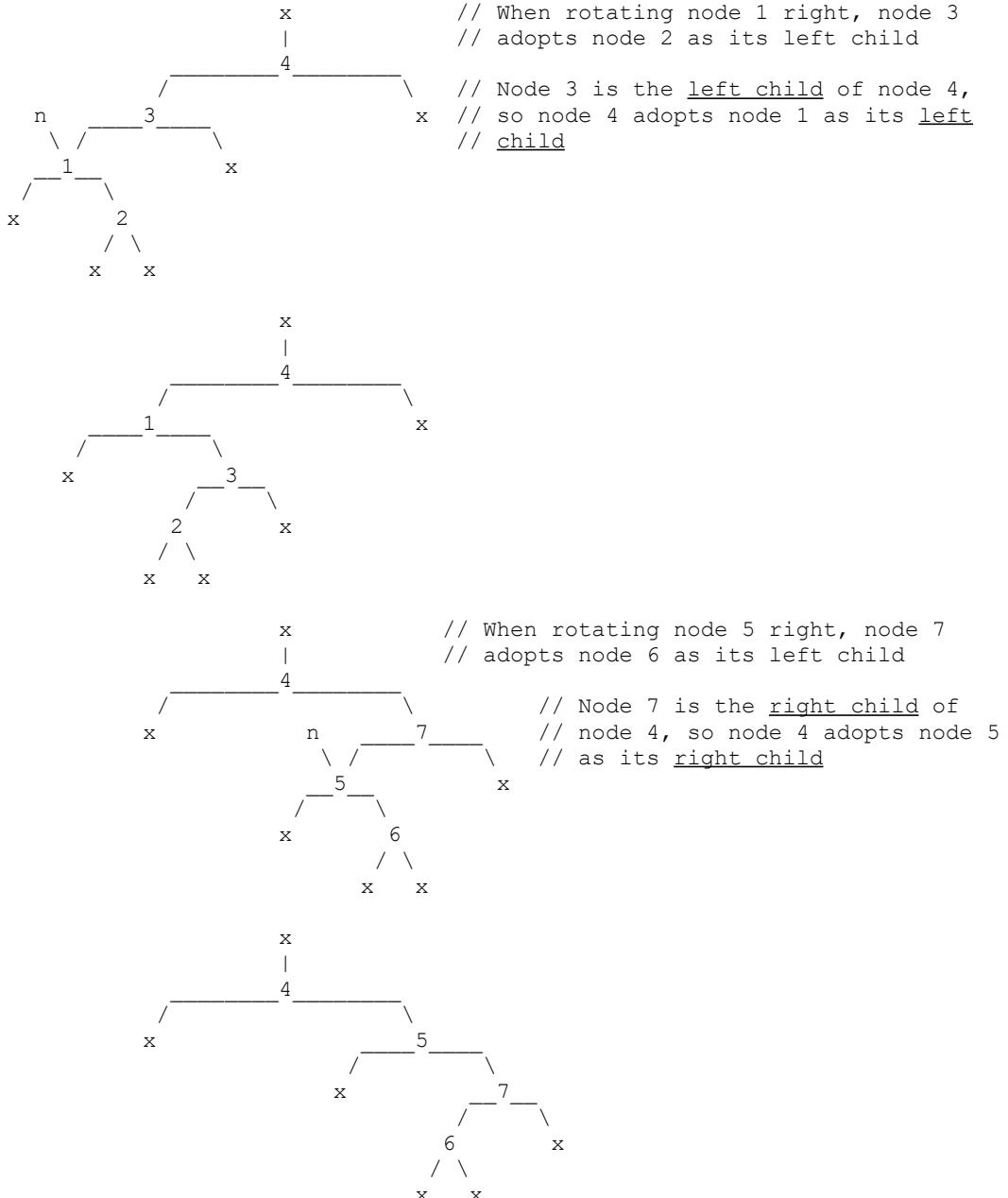


Rotating a node does not affect the in-order sequence of a tree; the predecessor and successor links are therefore unaffected.

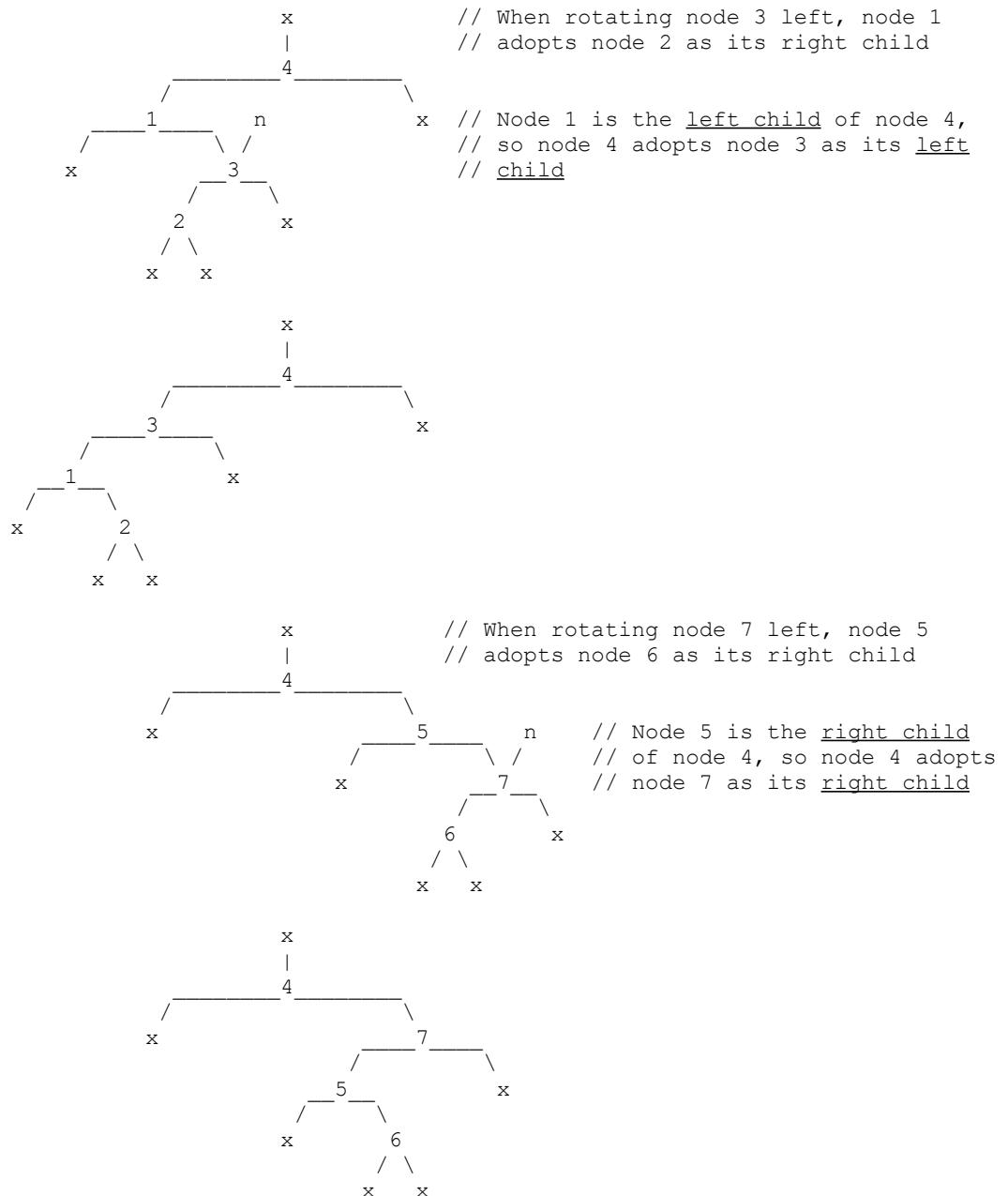
Note, however, that when rotating the left / right child of the root, the tree's root pointer must be updated:



Also note that when performing a right rotation on a node n, n's parent adopts n's right child and n's grandparent adopts n:



Similarly, when performing a left rotation on a node n, n's parent adopts n's left child and n's grandparent adopts n:



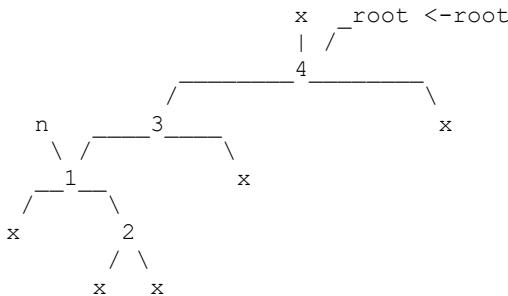
The function (rotate.h, lines 13-14)

```
template <class Node>
void rotateRight(Node* n, Node** root);
```

performs a right rotation on the node n, which must be the left child of its parent. The parameter

```
Node** root
```

is a pointer to the tree's root pointer. If n is the left child of the root, the function updates the tree's root pointer accordingly (lines 40-62). Consider, for example, the tree



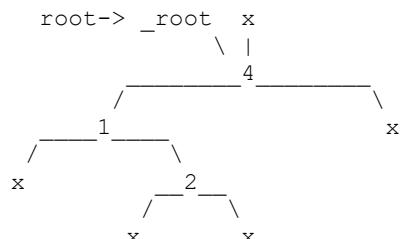
The function rotates node 1 by performing the following operations:

```
// root points to (the pointer) _root, not node 4

Node* n1 = n;                      // n1 points to node 1
Node* n2 = n1->right;              // n2 points to node 2
Node* n3 = n1->parent;             // n3 points to node 3
Node* n4 = n1->parent->parent;     // n4 points to node 4
```

```
// Move n1 up a level and update the root pointer if necessary
```

```
n1->parent = n4;                  // Connect n1 to n4
if (isLeftChild(n3))               // Connect n4 to n1
    n4->left = n1;
else if (isRightChild(n3))
    n4->right = n1;
else
    *root = n1;
```



```
// Move n3 down a level
```

```
n3->parent = n1;                  // Connect n3 to n1
n1->right = n3;                  // Connect n1 to n3
```

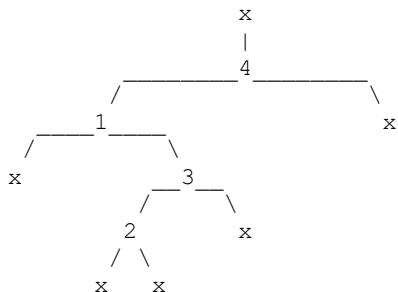
```

root-> _root  x
          \ |
           4
         /   \
        1     x
       / \   |
      x   3   x
             / \
            x   x

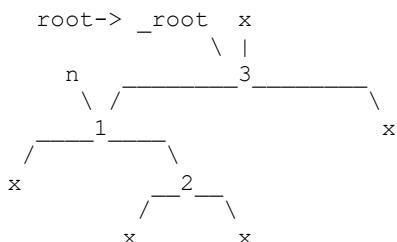
// Make n3 adopt n2

n3->left = n2;                      // Connect n3 to n2
if (n2 != nullptr)                   // Connect n2 to n3
    n2->parent = n3;

```



Given the tree



the function rotates node 1 by performing the following operations:

```

Node* n1 = n;                      // n1 points to node 1
Node* n2 = n1->right;              // n2 points to node 2
Node* n3 = n1->parent;             // n3 points to node 3
Node* n4 = n1->parent->parent;    // n4 is null

// Move n1 up a level and update the root pointer if necessary

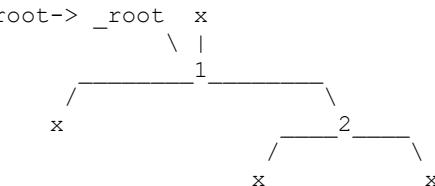
n1->parent = n4;                  // Connect n1 to n4

```

```

if (isLeftChild(n3))
    n4->left = n1;
else if (isRightChild(n3))
    n4->right = n1;
else
    *root = n1;                                // Point _root to node 1

```

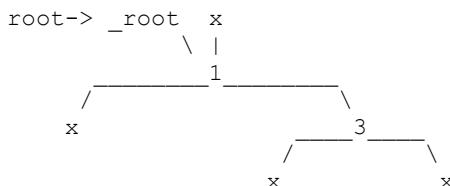


```
// Move n3 down a level
```

```

n3->parent = n1;                          // Connect n3 to n1
n1->right = n3;                           // Connect n1 to n3

```

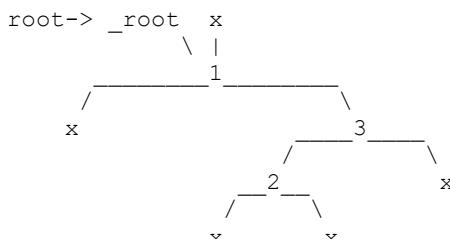


```
// Make n3 adopt n2
```

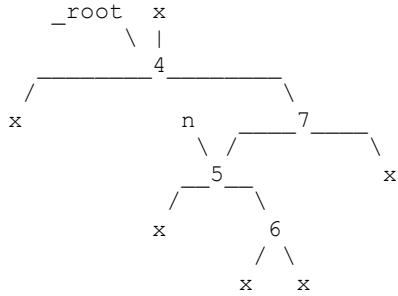
```

n3->left = n2;                           // Connect n3 to n2
if (n2 != nullptr)
    n2->parent = n3;                     // Connect n2 to n3

```

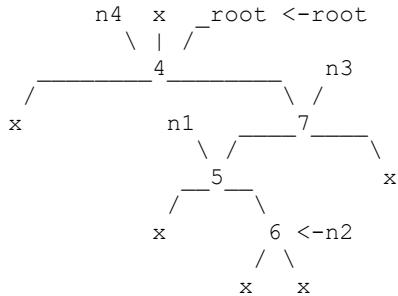


Given the tree



the function rotates node 5 by performing the following operations:

```
Node* n1 = n;                                // n1 points to node 5  
Node* n2 = n1->right;                         // n2 points to node 6  
Node* n3 = n1->parent;                          // n3 points to node 7  
Node* n4 = n1->parent->parent;                 // n4 points to node 4
```

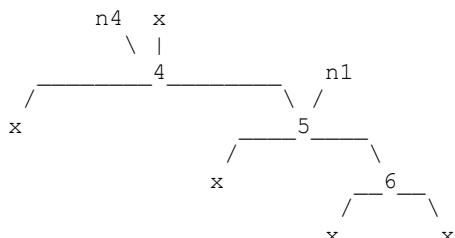


```
// Move node 5 up a level and update the root pointer if necessary
```

```

n1->parent = n4;                                // Connect node 5 to node 4
if (isLeftChild(n3))
    n4->left = n1;
else if (isRightChild(n3))
    n4->right = n1;                            // Connect node 4 to node 5
else
    *root = n1;

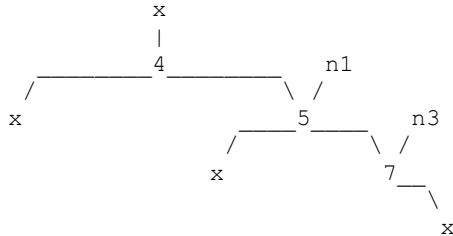
```



```
// Move node 7 down a level
```

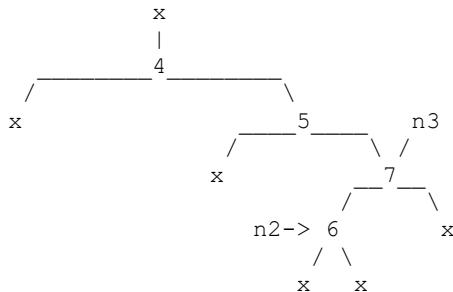
```
n3->parent = n1; // Connect node 7 to node 5
```

```
n1->right = n3; // Connect node 5 to node 7
```



```
// Make node 7 adopt node 6
```

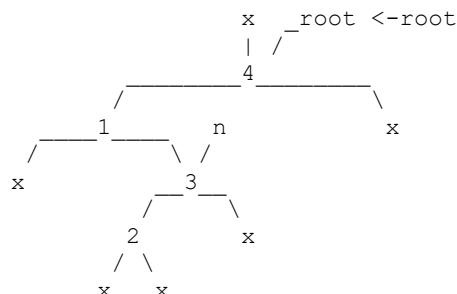
```
n3->left = n2; // Connect node 7 to node 6
if (n2 != nullptr)
    n2->parent = n3; // Connect node 6 to node 7
```



The function (lines 10-11)

```
template <class Node>
void rotateLeft(Node* n, Node** root);
```

performs a left rotation on the node n, which must be the right child of its parent. If n is the right child of the root, the function updates the tree's root pointer accordingly (lines 16-38). This function is the mirror image of `rotateRight`. Consider, for example, the tree



The function rotates node 3 left by performing the following operations:

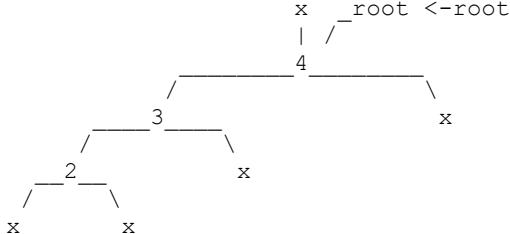
```

Node* n3 = n;                      // n3 points to node 3
Node* n2 = n3->left;              // n2 points to node 2
Node* n1 = n3->parent;            // n1 points to node 1
Node* n4 = n3->parent->parent;    // n4 points to node 4

// Move n3 up a level and update the root pointer if necessary

n3->parent = n4;                  // Connect n3 to n4
if (isLeftChild(n1))              // Connect n4 to n3
    n4->left = n3;
else if (isRightChild(n1))
    n4->right = n3;
else
    *root = n3;

```

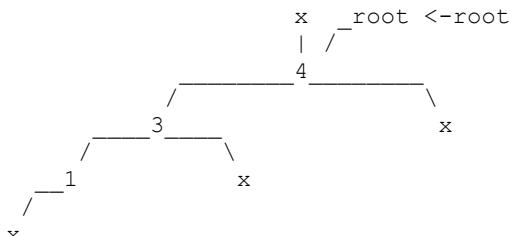


```
// Move n1 down a level
```

```

n1-parent = n3;                    // Connect n1 to n3
n3->left = n1;                   // Connect n3 to n1

```

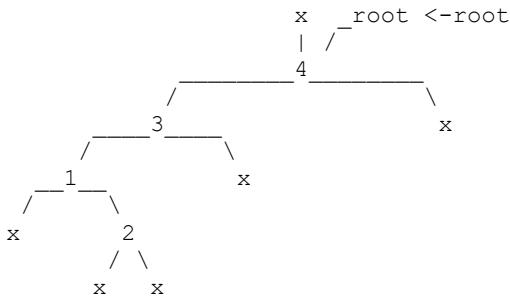


```
// Make n1 adopt n2
```

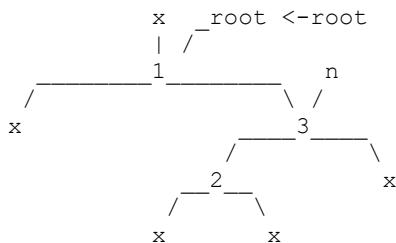
```

n1->right = n2;                  // Connect n1 to n2
if (n2 != nullptr)
    n2->parent = n1;             // Connect n2 to n1

```



Given the tree



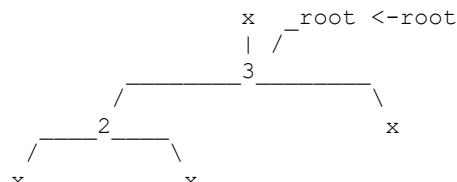
the function rotates node 3 by performing the following operations:

```

Node* n3 = n;                      // n3 points to node 3
Node* n2 = n3->left;               // n2 points to node 2
Node* n1 = n3->parent;             // n1 points to node 1
Node* n4 = n3->parent->parent;     // n4 is null

// Move n3 up a level and update the root pointer if necessary

n3->parent = n4;                  // Connect n3 to n4
if (isLeftChild(n1))
    n4->left = n3;
else if (isRightChild(n1))
    n4->right = n3;
else
    *root = n3;                   // Point _root to node 3
  
```



```
// Move n1 down a level
```

```
n1->parent = n3;                 // Connect n1 to n3
n3->left = n1;                  // Connect n3 to n1
```

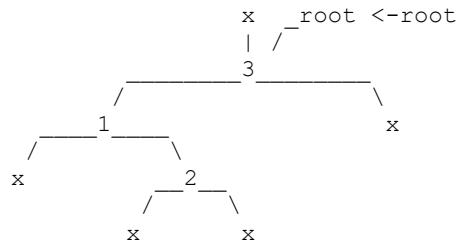
```

x   /--> root <-root
|   |
3   /   \
/   \   \
1   x
/
x

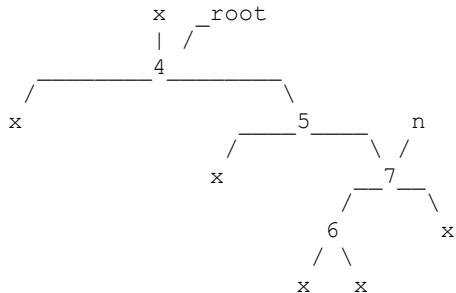
// Make n1 adopt n2

n1->right = n2;           // Connect n1 to n2
if (n2 != nullptr)
    n2->parent = n1;       // Connect n2 to n1

```



Given the tree

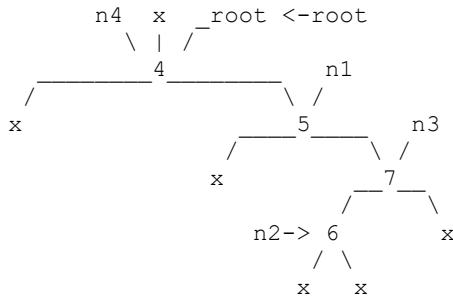


the function rotates node 7 by performing the following operations:

```

Node* n3 = n;           // n3 points to node 7
Node* n2 = n3->left;     // n2 points to node 6
Node* n1 = n3->parent;   // n1 points to node 5
Node* n4 = n3->parent->parent; // n4 points to node 4

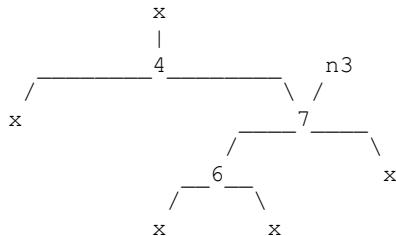
```



```
// Move node 7 up a level and update the root pointer if necessary
```

```

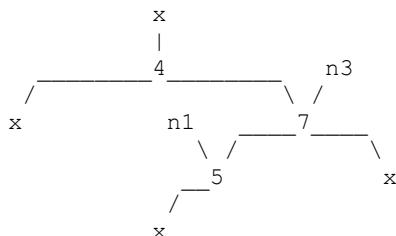
n3->parent = n4;                      // Connect node 7 to node 4
if (isLeftChild(n1))
    n4->left = n3;
else if (isRightChild(n1))
    n4->right = n3;                  // Connect node 4 to node 7
else
    *root = n3;
  
```



```
// Move node 5 down a level
```

```

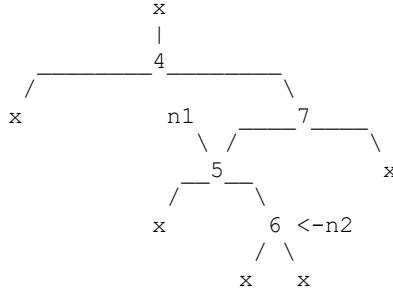
n1-parent = n3;                      // Connect node 5 to node 7
n3->left = n1;                      // Connect node 7 to node 5
  
```



```
// Make node 5 adopt node 6
```

```

n1->right = n2;                     // Connect node 5 to node 6
if (n2 != nullptr)
    n2->parent = n1;                // Connect node 6 to node 5
  
```



The member function (BinaryTree.h, line 65)

```
void rotate(const_iterator rotationPoint);
```

rotates the node at rotationPoint and updates the tree's root pointer accordingly. If the node at rotationPoint is the root, the function does nothing (memberFunctions\_5.h, lines 6-17).

Lines 12-13 (main.cpp) declare BinaryTree and Iter as aliases of the types `BinaryTree<int, int>` and `BinaryTree<int, int>::iterator`. Lines 15-16 construct a tree `t` and a `PrintBtNode` function object `printNode`.

Line 18,

```
Iter i5 = t.insert(makePair(5, 0)).first;
```

inserts the element (5,0) and initializes the iterator `i5` to point to that element. Recall from Chapter 14.9 that the expression

```
t.insert(makePair(5, 0))
```

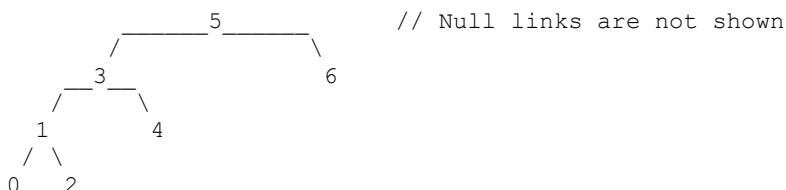
returns a `Pair<iterator, bool>`. The expression

```
t.insert(makePair(5, 0)).first
```

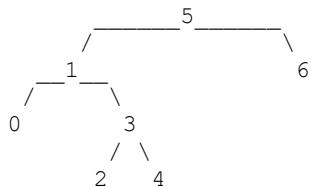
therefore denotes the iterator returned by `insert`. Similarly, line 19,

```
Iter i3 = t.insert(makePair(3, 0)).first;
```

inserts the element (3,0) and initializes the iterator `i3` to point to that element. Lines 20-24 insert the elements (6,0), (1,0), (4,0), (0,0), and (2,0), after which the layout of `t` is



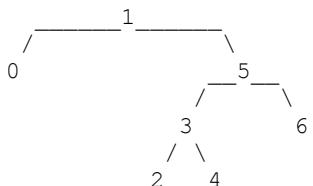
Lines 26-27 perform a right rotation on node 1, generating the output



```

node 0
  parent 1
node 1
  parent 5
  left 0
  right 3
node 2
  parent 3
node 3
  parent 1
  left 2
  right 4
node 4
  parent 3
node 5
  left 1
  right 6
node 6
  parent 5
  
```

Lines 30-31 perform another right rotation on node 1, generating the output



```

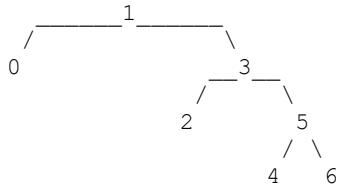
node 0
  parent 1
node 1
  left 0
  right 5
node 2
  parent 3
node 3
  parent 5
  left 2
  right 4
node 4
  parent 3
  
```

```

node 5
  parent 1
  left 3
  right 6
node 6
  parent 5

```

Lines 34-35 perform a right rotation on node 3, generating the output

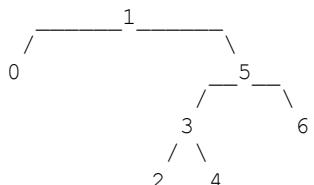


```

node 0
  parent 1
node 1
  left 0
  right 3
node 2
  parent 3
node 3
  parent 1
  left 2
  right 5
node 4
  parent 5
node 5
  parent 3
  left 4
  right 6
node 6
  parent 5

```

Lines 38-39 perform a left rotation on node 5, generating the output



```

node 0
  parent 1
node 1
  left 0
  right 5
node 2

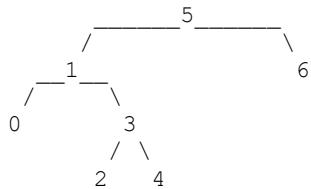
```

```

parent 3
node 3
  parent 5
  left 2
  right 4
node 4
  parent 3
node 5
  parent 1
  left 3
  right 6
node 6
  parent 5

```

Lines 42-43 perform another left rotation on node 5, generating the output

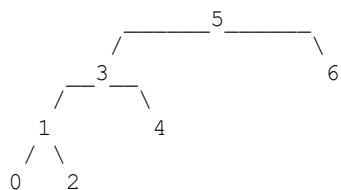


```

node 0
  parent 1
node 1
  parent 5
  left 0
  right 3
node 2
  parent 3
node 3
  parent 1
  left 2
  right 4
node 4
  parent 3
node 5
  left 1
  right 6
node 6
  parent 5

```

Lines 46-47 perform a left rotation on node 3, generating the output



```
node 0
    parent 1
node 1
    parent 3
    left 0
    right 2
node 2
    parent 1
node 3
    parent 5
    left 1
    right 4
node 4
    parent 3
node 5
    left 3
    right 6
node 6
    parent 5
```



## 15.2: Introducing the AvlTree Class Template

*Source files and folders*

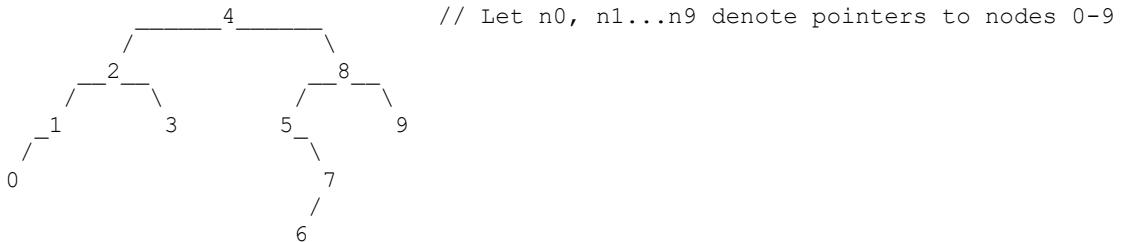
```
AvlTree/1
AvlTree/common/AvlTreeNode.h
AvlTree/common/memberFunctions_1.h
AvlTree/common/memberFunctions_2.h
PrintAvlNode
```

*Chapter outline*

- AVL nodes
- Verifying the structure of an AVL tree
- Private member functions (`_copyNode` / `_overwriteElement`)
- All remaining member functions (except `insert` and `erase`)

This chapter introduces the `AvlTree` class, a self-balancing binary tree. "Balancing" is the process by which the height of a tree is reduced by rotating nodes. The AVL tree is named after its inventors, the Soviet mathematicians G.M. Adelson-Velskii and E.M. Landis.

An `AvlTreeNode` is identical to a `BinaryTreeNode`, with the addition of a "balance factor" data member. The balance factor of a node  $n$  is equal to the height of  $n$ 's right subtree minus that of its left subtree. A positive balance factor indicates a taller right subtree, while a negative balance factor indicates a taller left subtree:



```

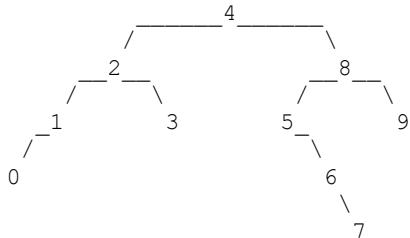
n0->balanceFactor = 0 - 0 = 0
n1->balanceFactor = 0 - 1 = -1
n2->balanceFactor = 1 - 2 = -1
n3->balanceFactor = 0 - 0 = 0
n4->balanceFactor = 4 - 3 = 1
n5->balanceFactor = 2 - 0 = 2
n6->balanceFactor = 0 - 0 = 0
n7->balanceFactor = 0 - 1 = -1
n8->balanceFactor = 1 - 3 = -2
n9->balanceFactor = 0 - 0 = 0

```

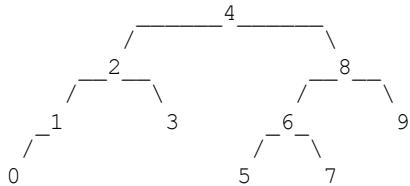
For any given node  $n$ , the heights of  $n$ 's left and right subtrees can differ by at most 1 level; in other

words, the balance factor of every node must be 0, -1, or 1. The above tree is therefore unbalanced at nodes 5 and 8. Balance can be restored, however, by performing two rotations on node 6:

```
// Rotate node 6 right
```



```
// Rotate node 6 left
```



```

n4->balanceFactor = 3 - 3 = 0
n5->balanceFactor = 0 - 0 = 0
n7->balanceFactor = 0 - 0 = 0
n8->balanceFactor = 1 - 2 = -1
  
```

```
// The balance factors of the remaining nodes are unchanged
```

The algorithms for rebalancing a tree will be discussed and implemented in the next two chapters.

The AvlTreeNode class is defined in the header file AvlTreeNode.h (lines 8-27). The default constructor (lines 15, 29-40) constructs a node from the given source element, initializes all of its links to null, and its balance factor to 0.

The function object (PrintAvlNode.h, lines 10-18)

```

template <class Node>
class PrintAvlNode
{
public:
    void operator()(const Node* n) const;

private:
    PrintBtNode<Node> _printBtNode;
};
```

visits the given node n by printing the key values of n and its parent / children, followed by n's balance

factor (lines 20-28):

```
template <class Node>
void PrintAvlNode<Node>::operator()(const Node* n) const
{
    using namespace std;

    _printBtNode(n);

    cout << " balanceFactor " << n->balanceFactor << endl;
}
```

Like the PrintBtNode class (Chapter 14.4), PrintAvlNode will be used with the traverseInOrder function to verify the structure of an AVL tree.

The AvlTree class is defined in the header file AvlTree.h (lines 14-77). The class definition is nearly identical to that of BinaryTree; the only differences are the member types (lines 34, 30, 26)

```
typedef AvlTreeNode<key_type, mapped_type> Node;
typedef BinaryTreeIter<AvlTree> iterator;
typedef ConstIter<AvlTree> const_iterator;
```

and the friend declaration (line 18)

```
friend class BinaryTreeIter<AvlTree>;
```

The insert and erase methods, which are not included, will be developed in the next two chapters. Also note that unlike BinaryTree, AvlTree does not have a public rotate method because its rebalancing algorithms are responsible for performing all rotations.

The header file memberFunctions\_1.h contains the definitions of all member functions except for `_copyNode` and `_overwriteElement`, which are placed in the the header file memberFunctions\_2.h. These two methods are nearly identical to those of BinaryTree; the only difference is that they have been updated to copy the `balanceFactor` from the source node (lines 14, 35). The definitions of all the remaining member functions are identical to those of BinaryTree.



## 15.3: Insert

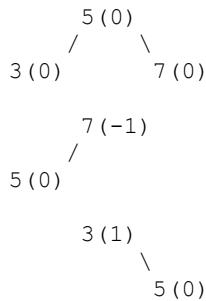
*Source files and folders*

*AvlTree/2*  
*AvlTree/common/memberFunctions\_3.h*

*Chapter outline*

- Correcting balance violations upon the insertion of a node
- Implementing AvlTree's insert method

Recall from the previous chapter that a tree is balanced when the balance factor of every node is either 0, -1, or 1. The following trees, for example, are all balanced (balance factors are shown in parentheses):



An AvlTree uses the same procedure as a BinaryTree to insert a new node. Following the insertion of a new node n, the balance factors of n's ancestors are updated by climbing the tree one node at a time, beginning at n.

Inserting a new node n changes the balance factor of its parent p. If n is the left child of p, then the height of p's left subtree has increased by 1 level; p's balance factor therefore decreases by 1. Consider, for example, the single-node tree

5(0)

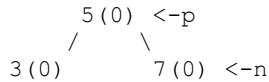
Inserting a new node 3 increases the height of node 5's left subtree by 1 level, so node 5's balance factor decreases by 1:

```

5(-1) <-p (The parent of the new node)
/
3(0) <-n (The new node)
  
```

Conversely, if the new node is the right child of its parent p, then the height of p's right subtree has increased by 1 level, which increases p's balance factor by 1. Inserting a new node 7, for example, increases the height of node 5's right subtree by 1 level; node 5's balance factor therefore increases

from -1 to 0:



A balance violation occurs when the balance factor of a node becomes -2 or 2. There are 4 types of balance violations that can occur upon the insertion of a new node:

- Left-left
- Right-right
- Left-right
- Right-left

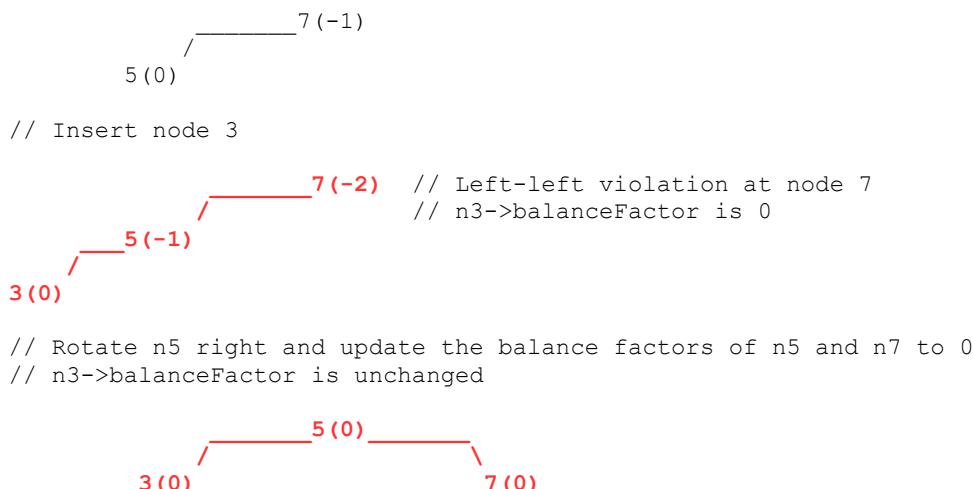
Left-left and right-right violations are corrected by performing a single rotation, while left-right and right-left violations are corrected by performing a double rotation.

The private member function (AvlTree.h, line 72)

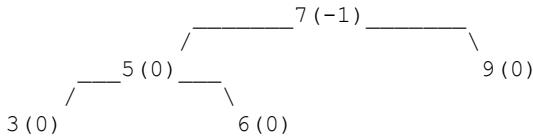
```
void _fixLeftLeft(Node* n);
```

corrects a left-left violation at node n (memberFunctions\_3.h, lines 95-105). There are 3 cases in which an insertion can cause a left-left violation. The procedure for correcting the violation, however, is the same for all 3 cases.

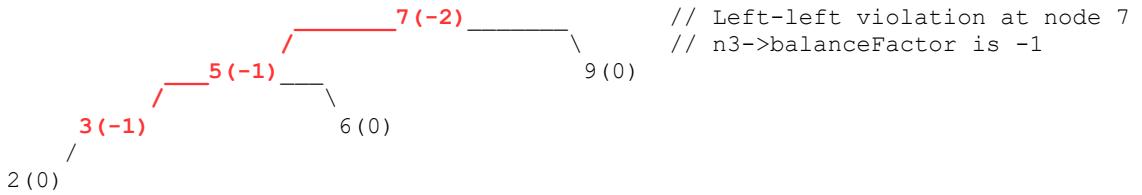
Case 1: Node 3's balance factor is 0 (The new node is node 3)



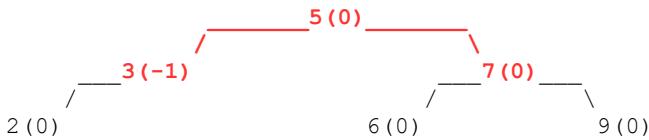
Case 2: Node 3's balance factor is -1 (The new node was inserted into node 3's left subtree)



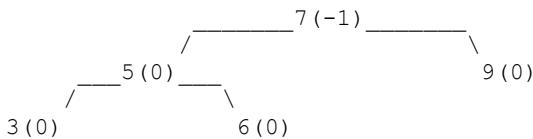
// Insert node 2



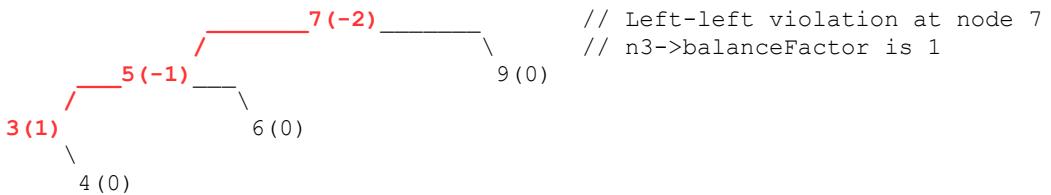
// Rotate n5 right and update the balance factors of n5 and n7 to 0  
// n3->balanceFactor is unchanged



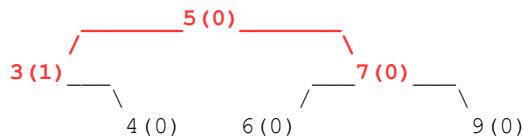
Case 3: Node 3's balance factor is 1 (The new node was inserted into node 3's right subtree)



// Insert node 4



// Rotate n5 right and update the balance factors of n5 and n7 to 0  
// n3->balanceFactor is unchanged

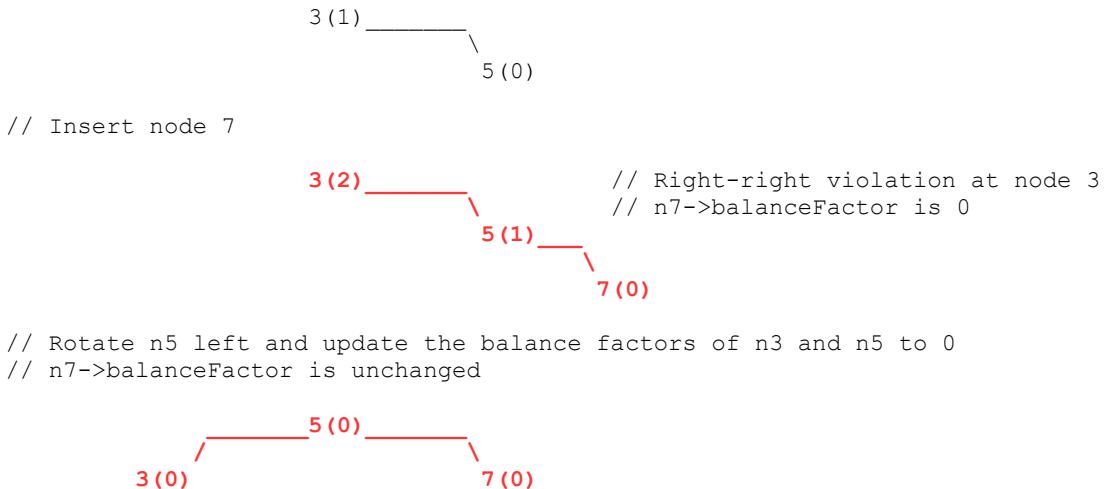


The private member function (AvlTree.h, line 74)

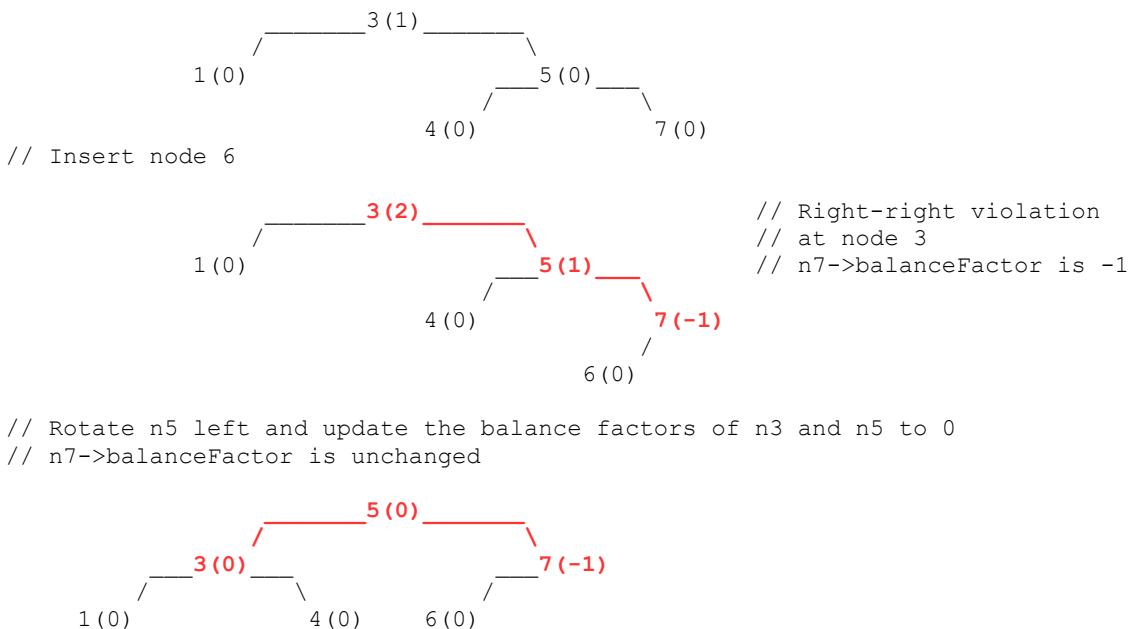
```
void _fixRightRight(Node* n);
```

which corrects a right-right violation at node n, is the mirror image of `_fixLeftLeft` (`memberFunctions_3.h`, lines 136-146).

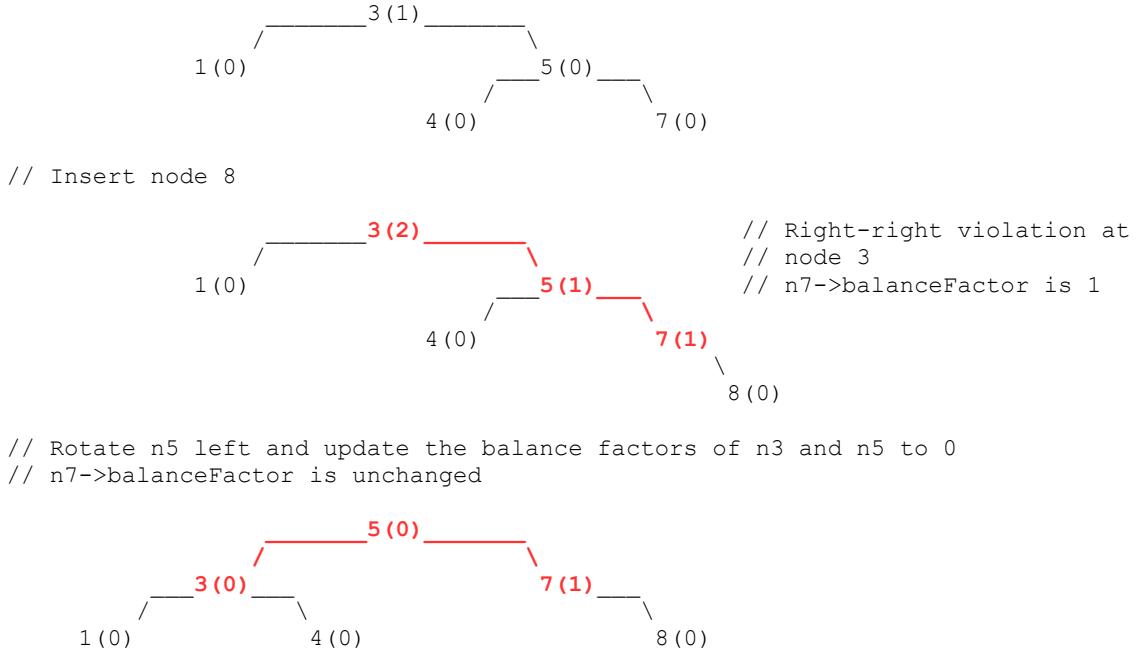
Case 1: Node 7's balance factor is 0 (The new node is node 7)



Case 2: Node 7's balance factor is -1 (The new node was inserted into node 7's left subtree)



Case 3: Node 7's balance factor is 1 (The new node was inserted into node 7's right subtree)

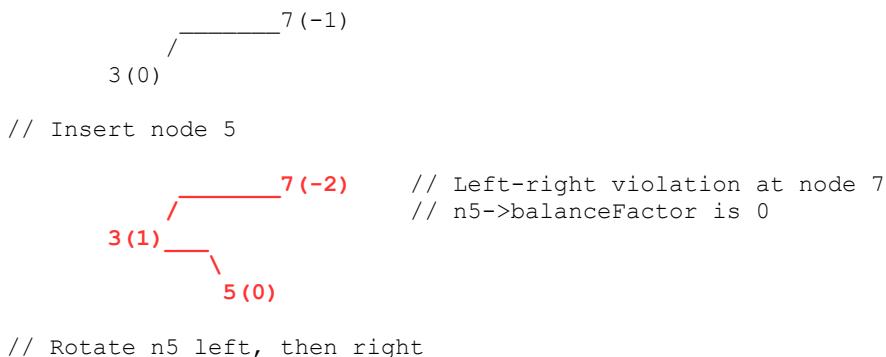


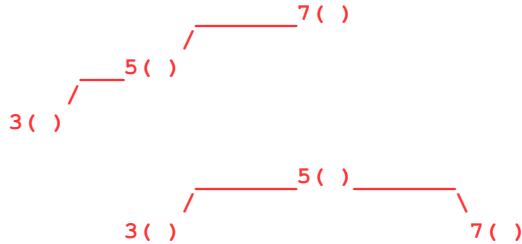
The private member function (AvlTree.h, line 73)

```
void _fixLeftRight(Node* n);
```

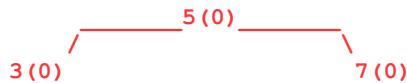
corrects a left-right violation at node n (memberFunctions\_3.h, lines 107-134). There are 3 cases in which an insertion can cause a left-right violation. All 3 cases are corrected by performing a double rotation on the same node, but they differ in how the balance factors are updated.

Case 1: Node 5's balance factor is 0 (The new node is node 5)

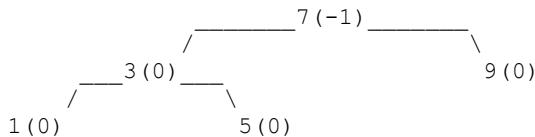




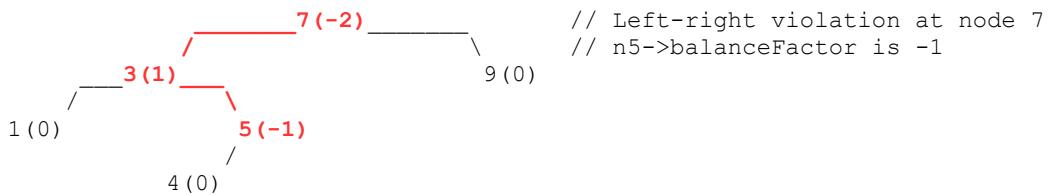
// Update the balance factors of n3 and n7 to 0  
 // n5->balanceFactor is unchanged



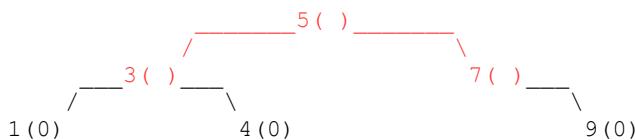
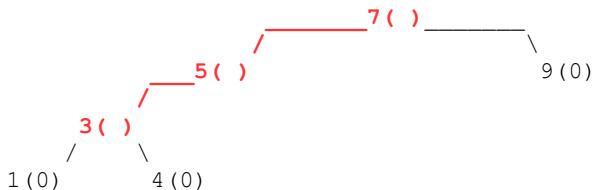
Case 2: Node 5's balance factor is -1 (The new node was inserted into node 5's left subtree)



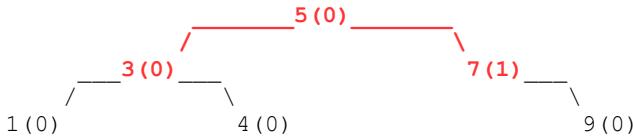
// Insert node 4



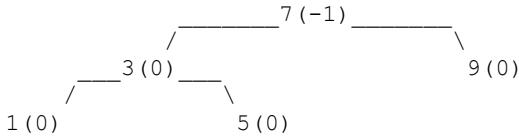
// Rotate n5 left, then right



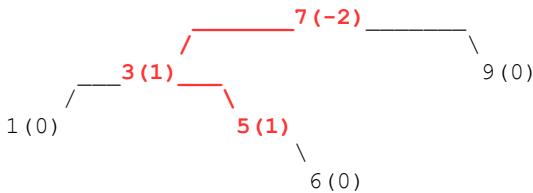
// Update the balance factors of n3, n5, and n7 to 0, 0, and 1



Case 3: Node 5's balance factor is 1 (The new node was inserted into node 5's right subtree)

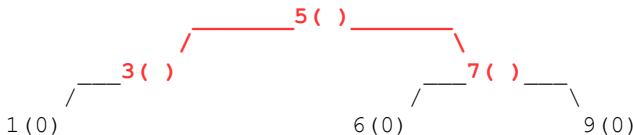
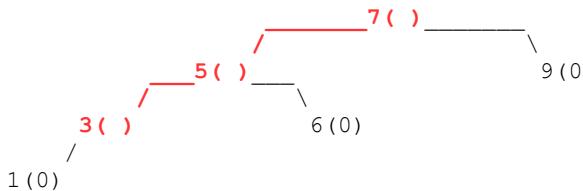


// Insert node 6

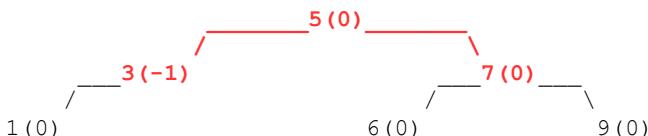


// Left-right violation at node 7  
// n5->balanceFactor is 1

// Rotate n5 left, then right



// Update the balance factors of n3, n5, and n7 to -1, 0, and 0



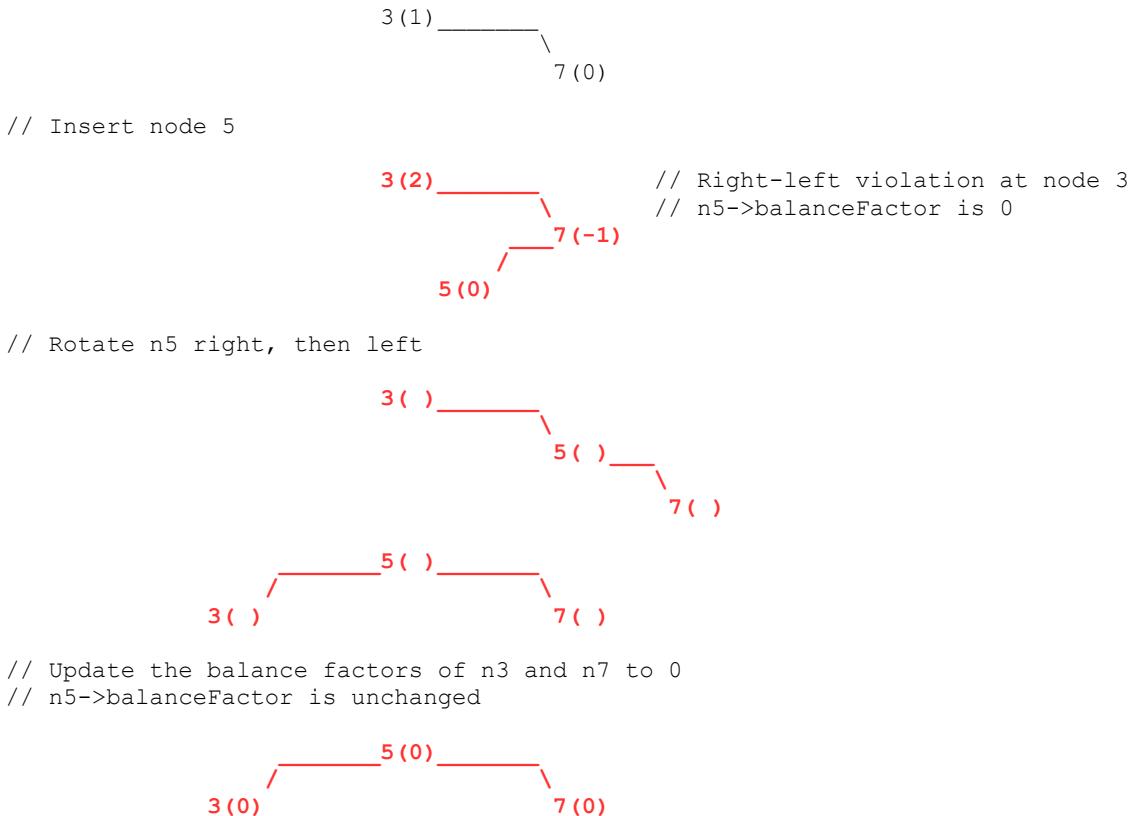
The private member function (AvlTree.h, line 75)

```
void _fixRightLeft(Node* n);
```

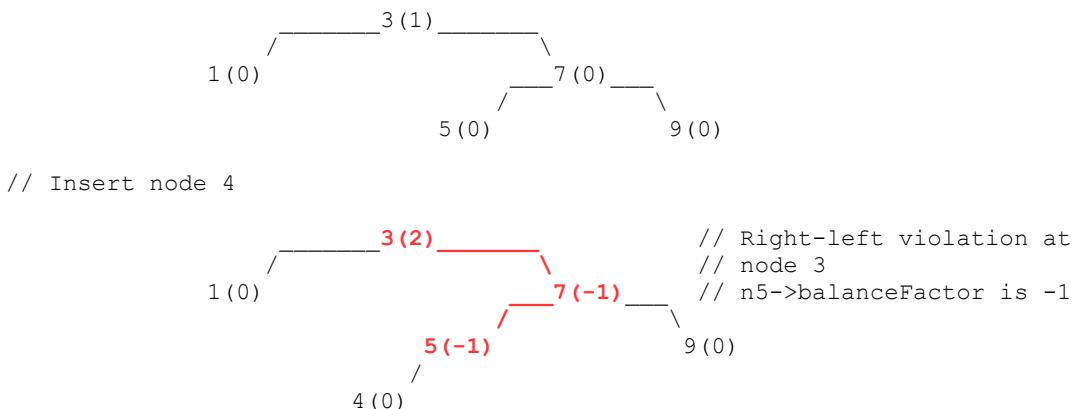
which corrects a right-left violation at node n, is the mirror image of `_fixLeftRight`

(memberFunctions\_3.h, lines 148-175).

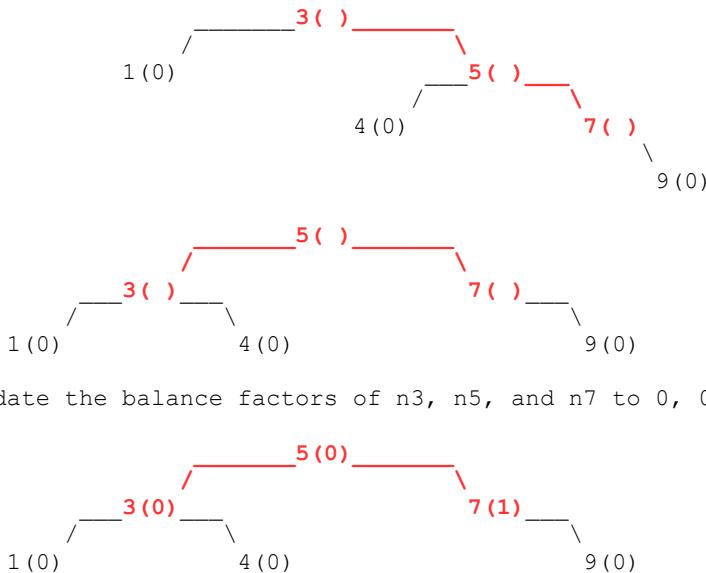
Case 1: Node 5's balance factor is 0 (The new node is node 5)



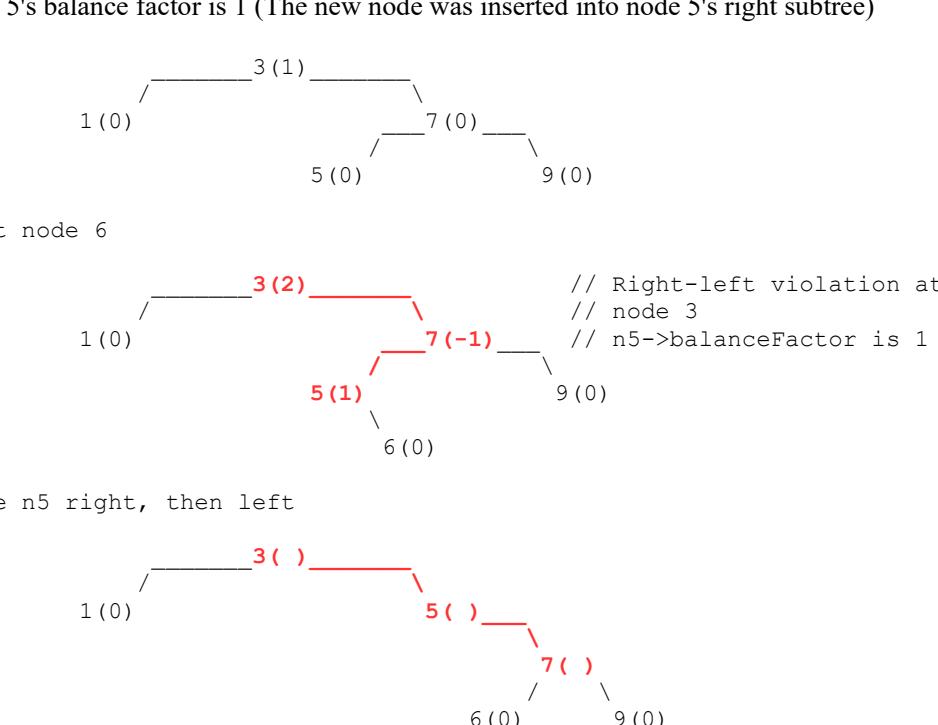
Case 2: Node 5's balance factor is -1 (The new node was inserted into node 5's left subtree)

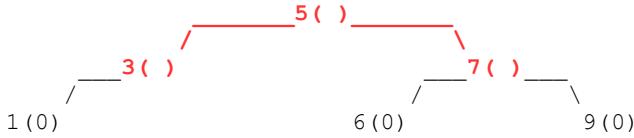


// Rotate n5 right, then left

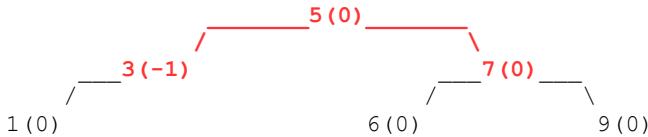


// Update the balance factors of n3, n5, and n7 to 0, 0, and 1



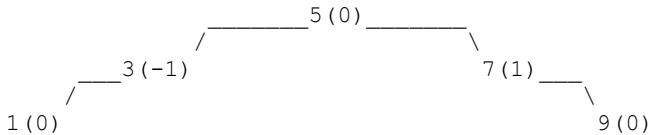


// Update the balance factors of n3, n5, and n7 to -1, 0, and 0

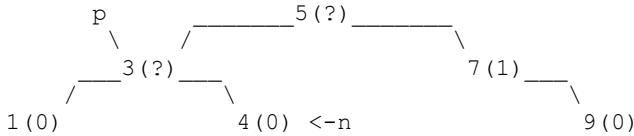


As mentioned at the beginning of the chapter, upon the insertion of a new node n, the balance factors of n's ancestors are updated by climbing the tree one node at a time, beginning at n.

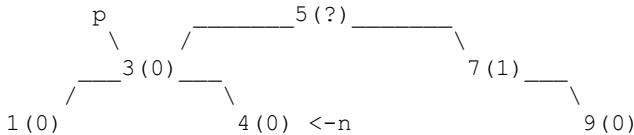
If the balance factor of an ancestor p becomes 0, then the height of p did not change. The balance factors of the remaining ancestors are therefore unchanged, and no further balancing is necessary:



// Insert node 4

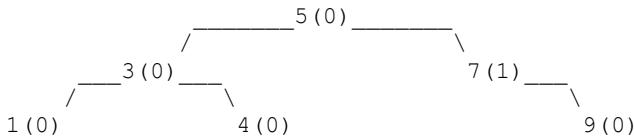


// n is the right child of p, so p's balance factor increases by 1,  
// becoming 0

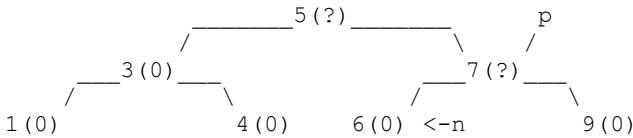


// The height of p (2 levels) therefore did not change

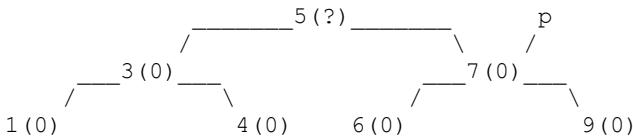
// The balance factor of the remaining ancestor (node 5) is therefore  
// unchanged, so no further balancing is required



```
// Insert node 6
```

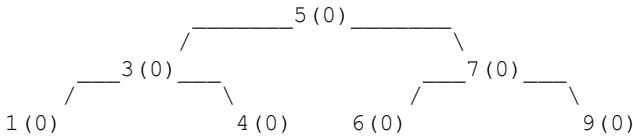


// n is the left child of p, so p's balance factor decreases by 1,  
// becoming 0

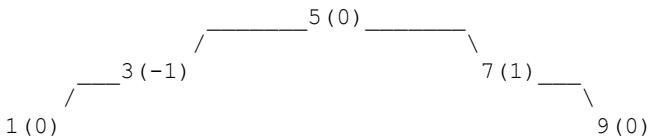


// The height of p (2 levels) therefore did not change

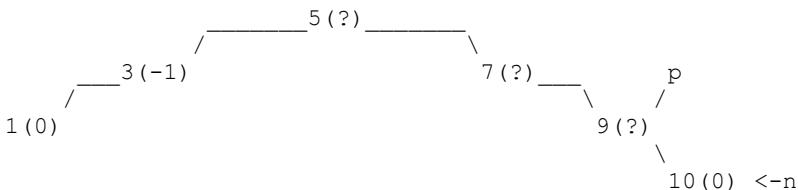
// The balance factor of the remaining ancestor (node 5) is therefore  
// unchanged, so no further balancing is required



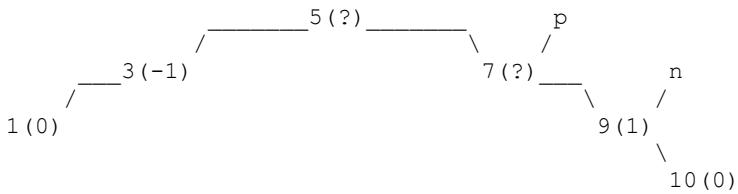
If the balance factor of an ancestor p becomes -2 or 2, the violation is corrected. After the correction is made, the node that took p's place in the tree is guaranteed to have a balance factor of 0. The balance factors of the remaining ancestors are therefore unchanged, and no further balancing is necessary:



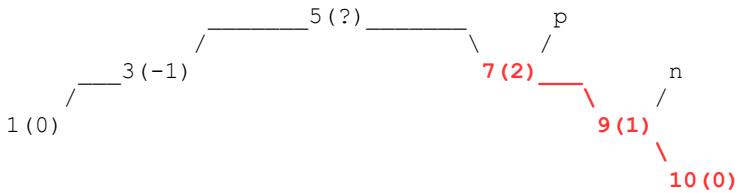
```
// Insert node 10
```



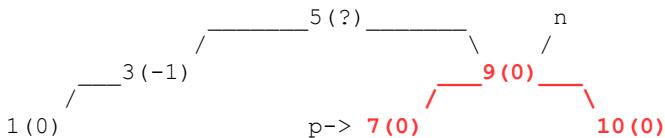
// n is the right child of p, so p's balance factor increases by 1,  
// becoming 1



// n is the right child of p, so p's balance factor increases by 1,  
// becoming 2

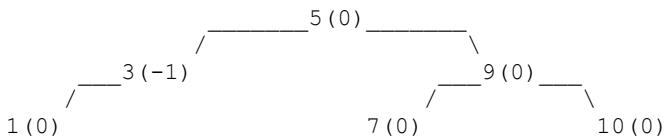


// Correct the right-right violation at p

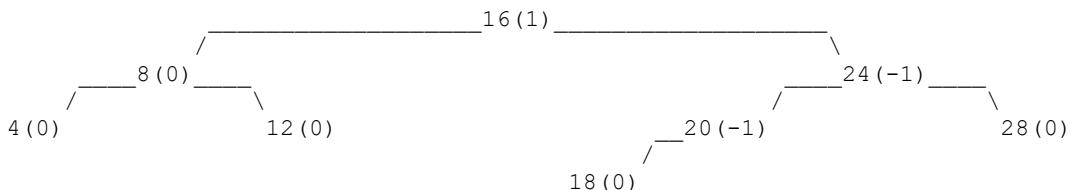


// The balance factor of the node that took p's place (node 9) is 0

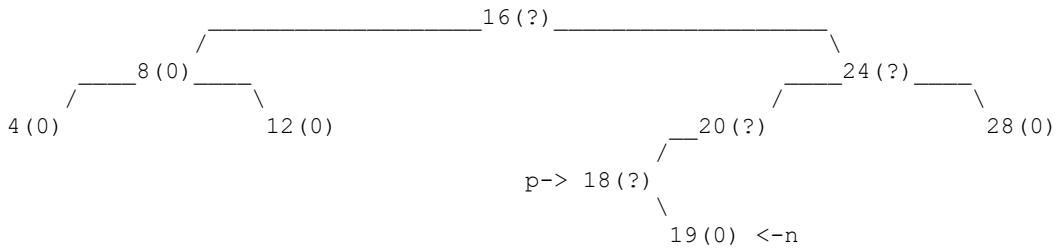
// The balance factor of the remaining ancestor (node 5) is therefore  
// unchanged, so no further balancing is required



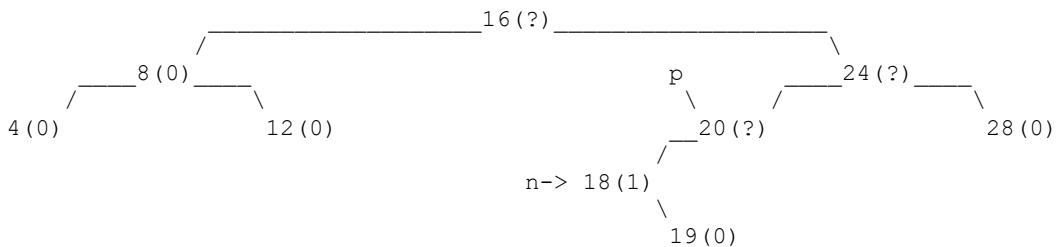
// Similarly, consider the tree



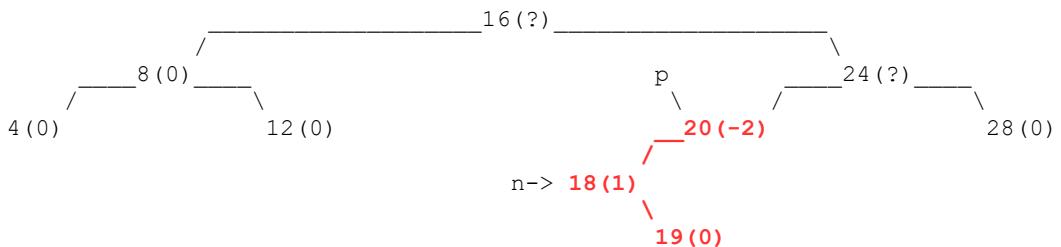
// Insert node 19



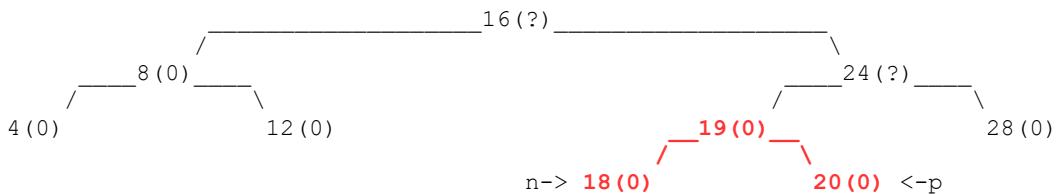
// n is the right child of p, so p's balance factor increases by 1,  
// becoming 1



// n is the left child of p, so p's balance factor decreases by 1,  
// becoming -2

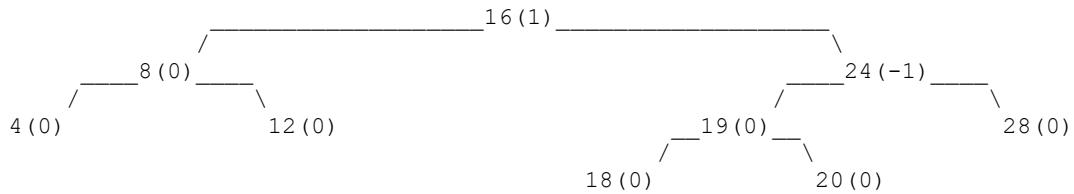


// Correct the left-right violation at p



// The balance factor of the node that took p's place (node 19) is 0

// The balance factors of the remaining ancestors (nodes 24 and 16) are  
// therefore unchanged, so no further balancing is required



The private member function (AvlTree.h, line 71)

```
void _balanceOnInsertion(Node* newNode);
```

balances the tree following the insertion of newNode. The pseudocode for the function is

```

template <class Key, class Mapped, class Predicate>
void AvlTree<Key, Mapped, Predicate>::_balanceOnInsertion(Node* newNode)
{
    For each ancestor p of newNode
    {
        Update the balance factor of p;

        If p's balance factor is 0
        {
            p's height is unchanged;
            The balance factors of the remaining ancestors are therefore
            unchanged, so no further balancing is necessary;
        }
        Otherwise, if p's balance factor is -2
        {
            Determine whether a left-left or left-right violation has occurred,
            and fix it accordingly;

            The balance factor of the node that took p's place in the tree is 0;
            the balance factors of the remaining ancestors are therefore
            unchanged, so no further balancing is necessary;
        }
        Otherwise, if p's balance factor is 2
        {
            Determine whether a right-right or right-left violation has occurred,
            and fix it accordingly;

            The balance factor of the node that took p's place in the tree is 0;
            The balance factors of the remaining ancestors are therefore
            unchanged, so no further balancing is necessary;
        }
    }
}
  
```

The function is defined in lines 58-93 (memberFunctions\_3.h).

The implementation of the insert method (AvlTree.h, line 62 / memberFunctions\_3.h, lines 10-56) is nearly identical to that of BinaryTree; the only difference is that it calls `_balanceOnInsertion` after

inserting a new node (`memberFunctions_3.h`, line 51).

Lines 14-15 (`main.cpp`) construct an `AvlTree<int, int> t` and a `PrintAvlNode` function object. The loop in lines 17-48 then repeats continuously. In each iteration, lines 19-21 prompt the user for a single-character command: i (for insert), p (for print), c (for clear), or q (for quit).

If the user enters i (lines 23-30), the program prompts the user to enter a key value and inserts it into the tree.

If the user enters p (lines 31-35), the program prints the structure of the tree by passing the `PrintAvlNode` function object to `traverseInOrder`.

If the user enters c (lines 36-39), the program clears the tree.

If the user enters q (lines 40-43), the program terminates the loop.

If the user enters some other command, the program prints the message “Invalid command” and begins the next iteration of the loop.

For the following sample run, refer to the diagrams on the corresponding pages. Note, however, that the tree must be cleared before inserting each set of key values.

Inserting the sets of key values (pp. 452-453)

```
7, 5, 3           // Left-left violation (Case 1)
7, 5, 9, 3, 6, 2 // Left-left violation (Case 2)
7, 5, 9, 3, 6, 4 // Left-left violation (Case 3)
```

generates the output

```
node 3           // Case 1
  parent 5
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor 0
node 7
  parent 5
  balanceFactor 0

node 2           // Case 2
  parent 3
  balanceFactor 0
node 3
  parent 5
  left 2
  balanceFactor -1
node 5
```

```

left 3
right 7
balanceFactor 0
node 6
  parent 7
  balanceFactor 0
node 7
  parent 5
  left 6
  right 9
  balanceFactor 0
node 9
  parent 7
  balanceFactor 0

node 3           // Case 3
  parent 5
  right 4
  balanceFactor 1
node 4
  parent 3
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor 0
node 6
  parent 7
  balanceFactor 0
node 7
  parent 5
  left 6
  right 9
  balanceFactor 0
node 9
  parent 7
  balanceFactor 0

```

Inserting the sets of key values (pp. 454-455)

```

3, 5, 7           // Right-right violation (Case 1)
3, 1, 5, 4, 7, 6 // Right-right violation (Case 2)
3, 1, 5, 4, 7, 8 // Right-right violation (Case 3)

```

generates the output

```

node 3           // Case 1
  parent 5
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor 0

```

```

node 7
  parent 5
  balanceFactor 0

node 1           // Case 2
  parent 3
  balanceFactor 0
node 3
  parent 5
  left 1
  right 4
  balanceFactor 0
node 4
  parent 3
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor 0
node 6
  parent 7
  balanceFactor 0
node 7
  parent 5
  left 6
  balanceFactor -1

node 1           // Case 3
  parent 3
  balanceFactor 0
node 3
  parent 5
  left 1
  right 4
  balanceFactor 0
node 4
  parent 3
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor 0
node 7
  parent 5
  right 8
  balanceFactor 1
node 8
  parent 7
  balanceFactor 0

```

Inserting the sets of key values (pp. 455-457)

```

7, 3, 5           // Left-right violation (Case 1)
7, 3, 9, 1, 5, 4 // Left-right violation (Case 2)

```

```
7, 3, 9, 1, 5, 6      // Left-right violation (Case 3)
```

generates the output

```
node 3                  // Case 1
  parent 5
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor 0
node 7
  parent 5
  balanceFactor 0

node 1                  // Case 2
  parent 3
  balanceFactor 0
node 3
  parent 5
  left 1
  right 4
  balanceFactor 0
node 4
  parent 3
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor 0
node 7
  parent 5
  right 9
  balanceFactor 1
node 9
  parent 7
  balanceFactor 0

node 1                  // Case 3
  parent 3
  balanceFactor 0
node 3
  parent 5
  left 1
  balanceFactor -1
node 5
  left 3
  right 7
  balanceFactor 0
node 6
  parent 7
  balanceFactor 0
node 7
  parent 5
```

```

left 6
right 9
balanceFactor 0
node 9
parent 7
balanceFactor 0

```

### Inserting the sets of key values (pp. 458-460)

```

3, 7, 5           // Right-left violation (Case 1)
3, 1, 7, 5, 9, 4 // Right-left violation (Case 2)
3, 1, 7, 5, 9, 6 // Right-left violation (Case 3)

```

### generate the output

```

node 3           // Case 1
parent 5
balanceFactor 0
node 5
left 3
right 7
balanceFactor 0
node 7
parent 5
balanceFactor 0

node 1           // Case 2
parent 3
balanceFactor 0
node 3
parent 5
left 1
right 4
balanceFactor 0
node 4
parent 3
balanceFactor 0
node 5
left 3
right 7
balanceFactor 0
node 7
parent 5
right 9
balanceFactor 1
node 9
parent 7
balanceFactor 0

node 1           // Case 3
parent 3
balanceFactor 0
node 3

```

```
parent 5
left 1
balanceFactor -1
node 5
left 3
right 7
balanceFactor 0
node 6
parent 7
balanceFactor 0
node 7
parent 5
left 6
right 9
balanceFactor 0
node 9
parent 7
balanceFactor 0
```

## 15.4: Erase

*Source files and folders*

*AvlTree/3  
AvlTree/common/memberFunctions\_4.h*

*Chapter outline*

- Correcting balance violations upon the erasure of a node
- Implementing AvlTree's erase method

Recall that BinaryTree's erasure algorithm erases either a leaf or 1-child node; if a node has 2 children, it takes the element of its in-order predecessor (which is guaranteed to be either a leaf or 1-child node), after which the predecessor is erased.

An AvlTree uses the same erasure algorithm. Following the erasure of a node t, the balance factors of t's ancestors are updated by climbing the tree one node at a time, beginning at t's parent.

Erasing a node t changes the balance factor of its parent p. If t was the left child of p, then the height of p's left subtree decreased by 1 level (i.e. p's left subtree was “pruned” by 1 level). The balance factor of p therefore increases by 1:

```

    3(0) <-p
    /   \
1(0) <-t  5(0)

// Erase node 1

    3(1) <-p
      \
      5(0)

// p's left subtree was pruned by 1 level
// p's balance factor therefore increases by 1, to 1

```

Conversely, if t was the right child of p, then the height of p's right subtree decreased by 1 level (i.e. p's right subtree was “pruned” by 1 level). The balance factor of p therefore decreases by 1:

```

    3(1) <-p
      \
      5(0) <-t

// Erase node 5

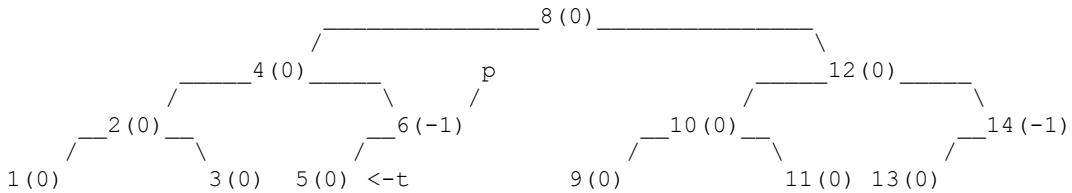
```

```
3(0) <-p
```

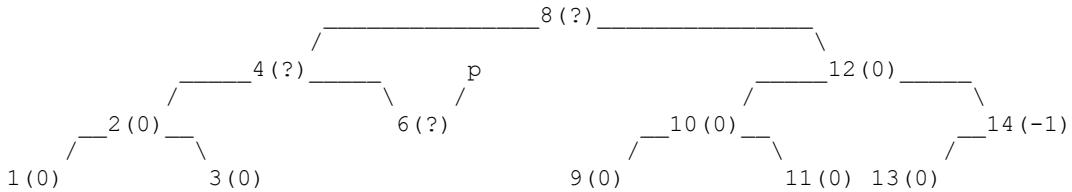
```
// p's right subtree was pruned by 1 level
// p's balance factor therefore decreases by 1, to 0
```

If the balance factor of an ancestor p becomes 0, then the height of p was reduced by the erasure of t. The balance factor of the next ancestor (p's parent) must therefore be updated accordingly.

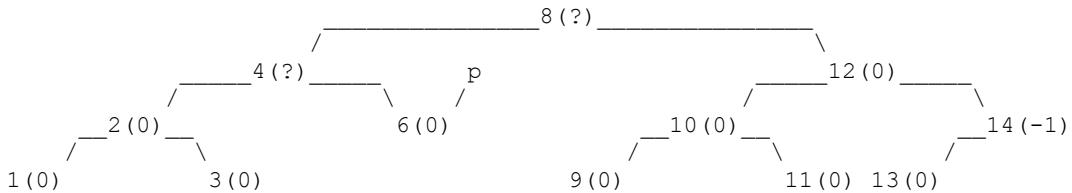
If the balance factor of an ancestor p becomes -1 or 1, then the height of p was unchanged by the erasure of t. The balance factors of the remaining ancestors are therefore unchanged, so no further balancing is necessary.



```
// Erase n5
```

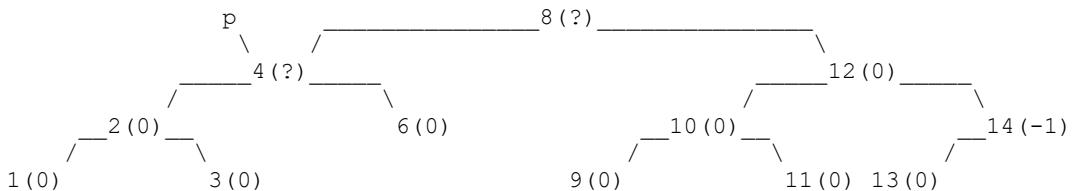


```
// n6's left subtree was pruned (reduced in height) by 1 level, so
// n6->balanceFactor increases by 1
```

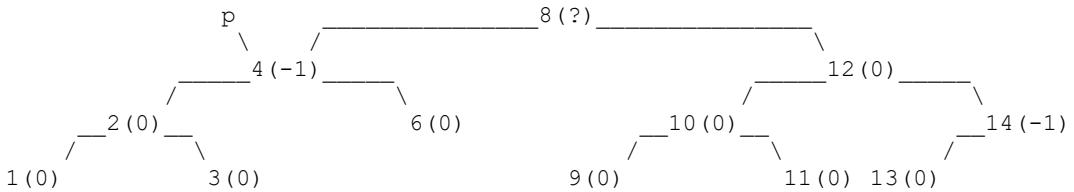


```
// n6->balanceFactor became 0, indicating that the height of n6 was
// reduced by the erasure of n5
```

```
// The balanceFactor of the next ancestor (n4) must therefore be updated
```

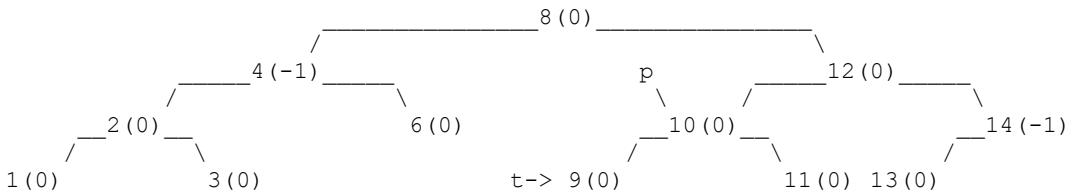


// n4's right subtree was pruned (reduced in height) by 1 level, so  
// n4->balanceFactor decreases by 1

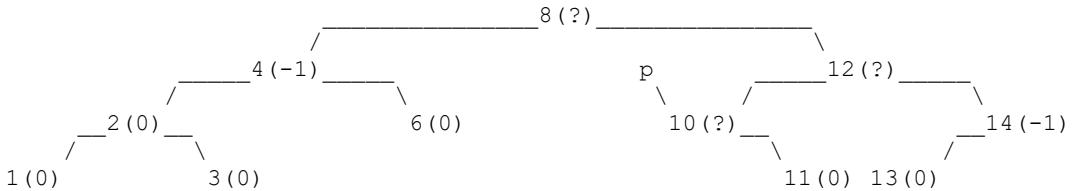


// n4->balanceFactor became -1, indicating that the height of n4 was  
// unchanged by the erasure of n5

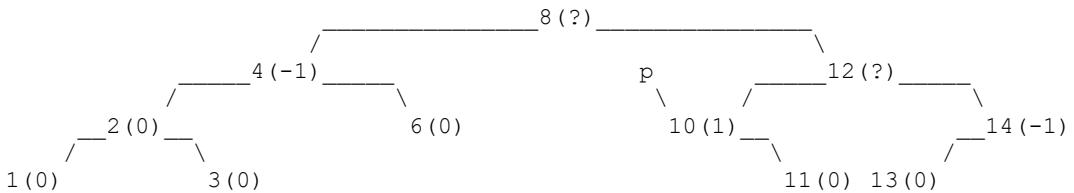
// The balanceFactor of the remaining ancestor (n8) is therefore unchanged,  
// so no further balancing is necessary



// Erase n9

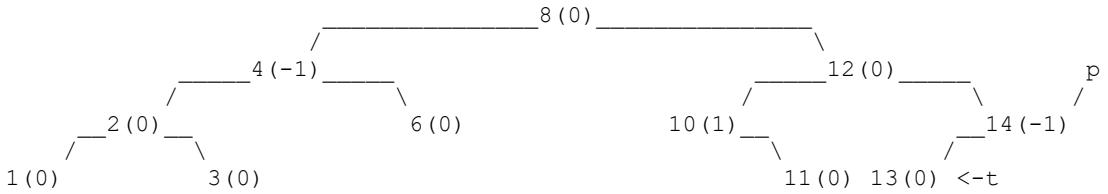


// n10's left subtree was pruned (reduced in height) by 1 level, so  
// n10->balanceFactor increases by 1

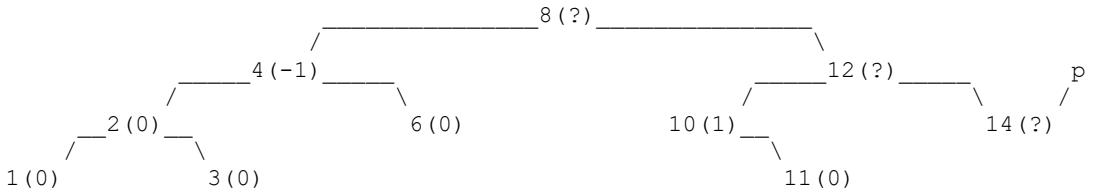


// n10->balanceFactor became 1, indicating that the height of n10 was  
// unchanged by the erasure of n9

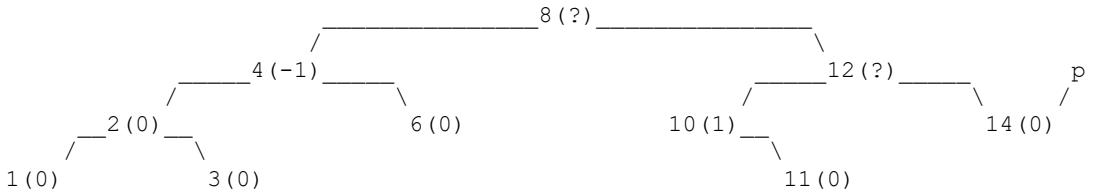
// The balanceFactors of the remaining ancestors (n12 and n8) are therefore  
// unchanged, so no further balancing is necessary



// Erase n13

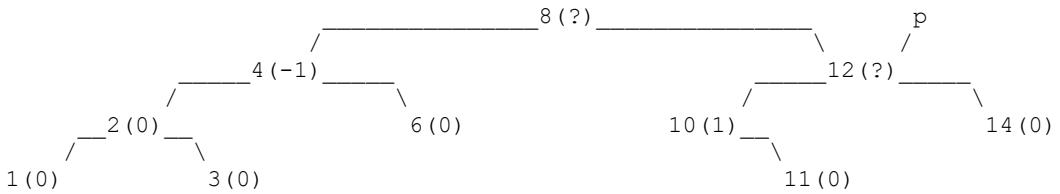


// n14's left subtree was pruned (reduced in height) by 1 level, so  
// n14->balanceFactor increases by 1

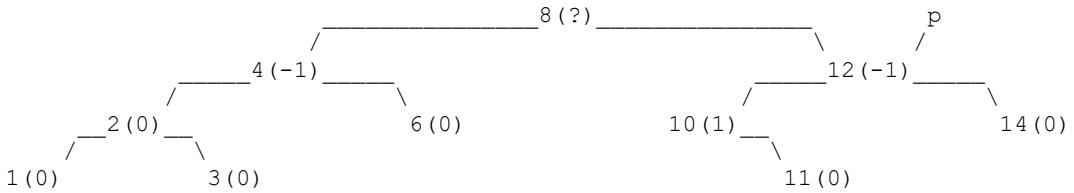


// n14->balanceFactor became 0, indicating that the height of n14 was  
// reduced by the erasure of n13

// The balanceFactor of the next ancestor (n12) must therefore be updated



// n12's right subtree was pruned (reduced in height) by 1 level, so  
// n12->balanceFactor decreases by 1



```
// n12->balanceFactor became -1, indicating that the height of n12 was
// unchanged by the erasure of n13

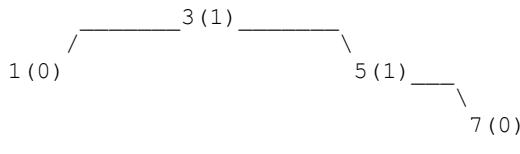
// The balanceFactor of the remaining ancestor (n8) is therefore unchanged,
// so no further balancing is necessary
```

As is the case with insertion, a balance violation occurs when the balance factor of a node becomes -2 or 2.

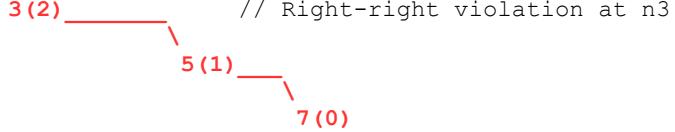
Pruning the left subtree of a node n can result in 3 types of balance violations:

- Right-right (identical to insertion)
- Right-left (identical to insertion)
- Right-zero

The member function `_fixRightRight` (implemented in the previous chapter) corrects a right-right violation:

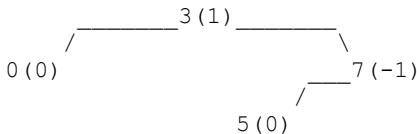


`// Erase n1 (Prune n3's left subtree)`

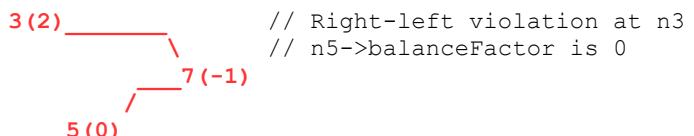


The member function `_fixRightLeft` (also implemented in the previous chapter) corrects a right-left violation:

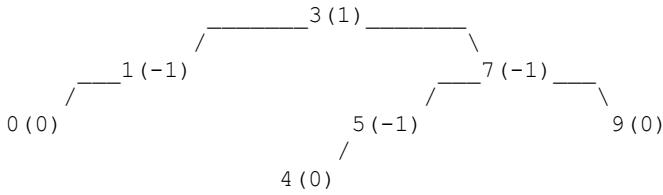
`// Case 1`



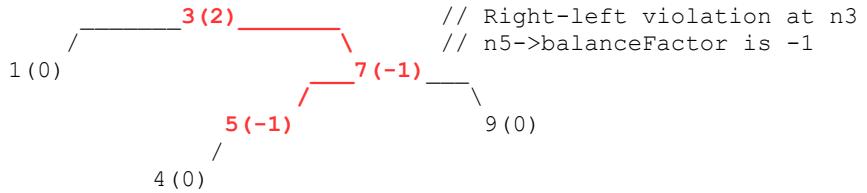
`// Erase n0 (Prune n3's left subtree)`



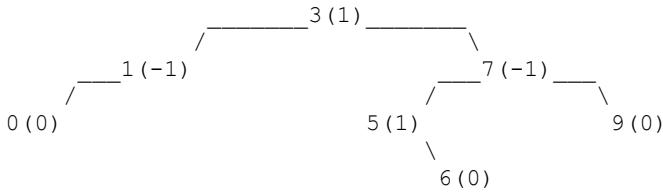
// Case 2



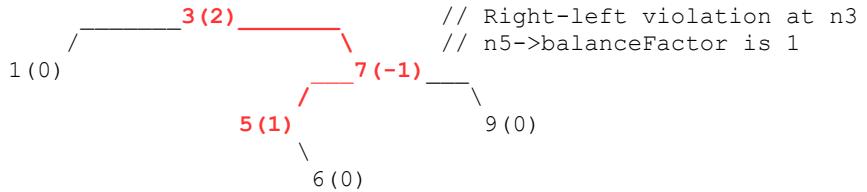
// Erase n0 (Prune n3's left subtree)



// Case 3



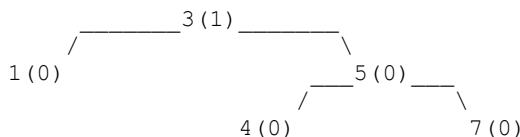
// Erase n0 (Prune n3's left subtree)



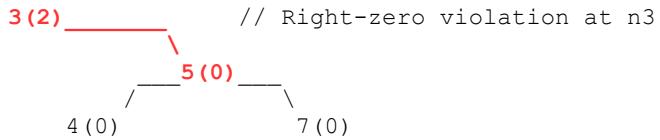
The private member function (AvlTree.h, line 81)

```
void _fixRightZero(Node* n);
```

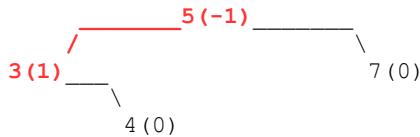
corrects a right-zero violation at node n (memberFunctions\_4.h, lines 120-130). Like a right-right violation, a right-zero violation is corrected by performing a single rotation:



```
// Erase n1 (Prune n3's left subtree)
```



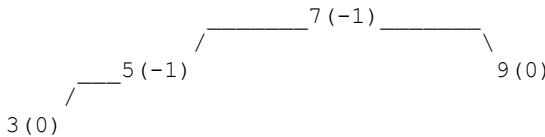
```
// Rotate n5 left and update the balance factors of n3 and n5 to 1 and -1
// The balance factors of n4 and n7 are unchanged
```



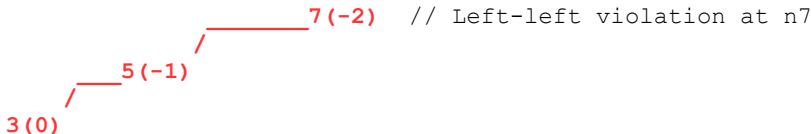
Pruning the right subtree of a node n can result in mirror images of the above violations:

- Left-left (identical to insertion)
- Left-right (identical to insertion)
- Left-zero

The member function `_fixLeftLeft` (implemented in the previous chapter) corrects a left-left violation:

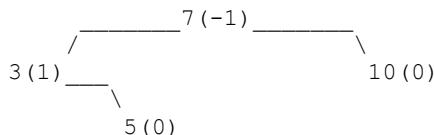


```
// Erase n9 (Prune n7's right subtree)
```

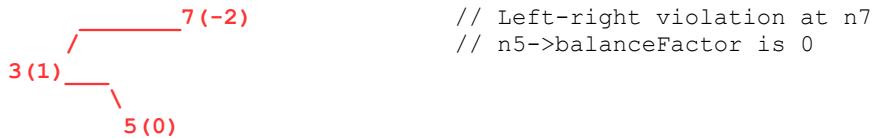


The member function `_fixLeftRight` (also implemented in the previous chapter) corrects a left-right violation:

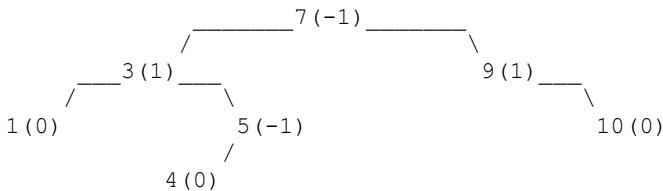
```
// Case 1
```



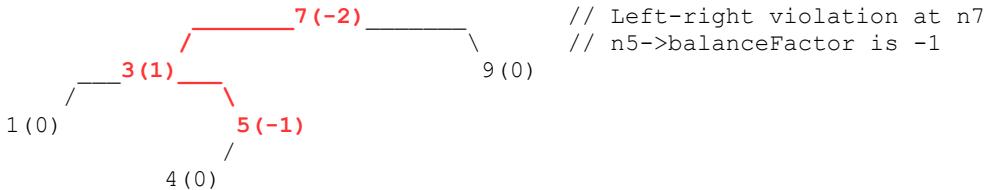
```
// Erase n10 (Prune n7's right subtree)
```



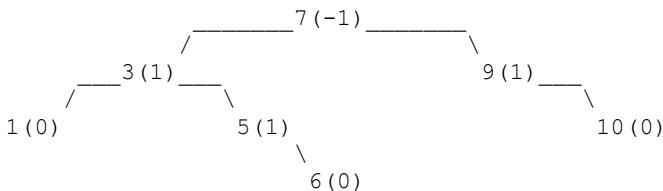
// Case 2



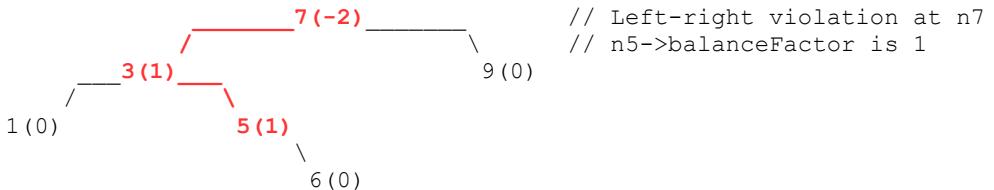
// Erase n10 (Prune n7's right subtree)



// Case 3



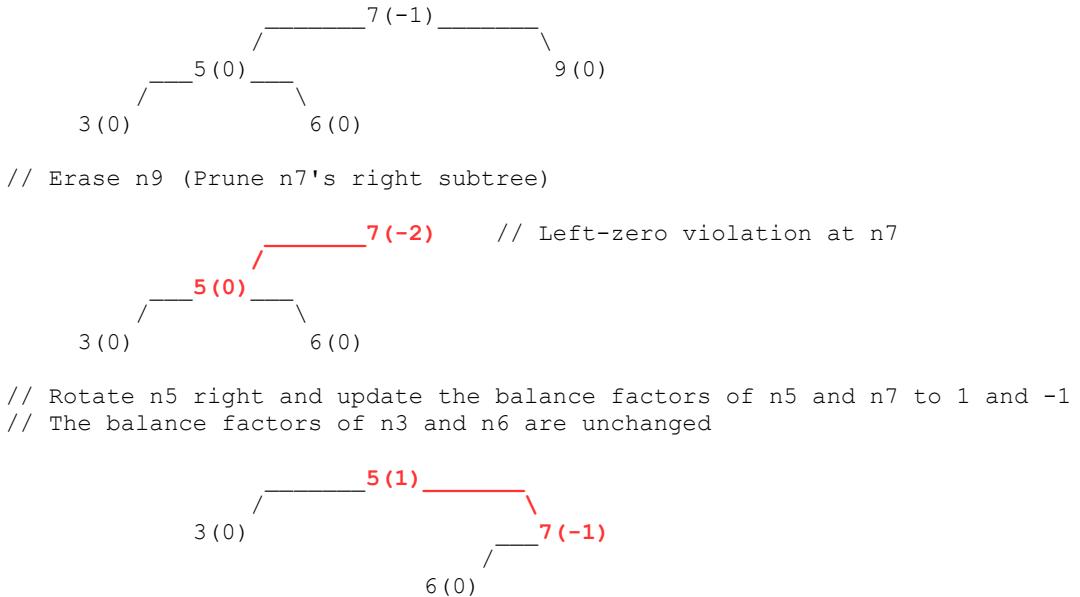
// Erase n10 (Prune n7's right subtree)



The private member function (AvlTree.h, line 78)

```
void _fixLeftZero(Node* n);
```

which corrects a left-zero violation at node n, is the mirror image of `_fixRightZero` (memberFunctions\_4.h, lines 108-118):



The private member function (AvlTree.h, line 74)

```
Node* _balanceOnLeftPruning(Node* p);
```

balances the node p following the pruning (reduction in height of) its left subtree. The pseudocode for the function is

```

template <class Key, class Mapped, class Predicate>
typename AvlTree<Key, Mapped, Predicate>::Node*
AvlTree<Key, Mapped, Predicate>::_balanceOnLeftPruning(Node* p)
{
    Increase p's balance factor by 1 (because p's left subtree was pruned);

    If p's balance factor is 2
    {
        Determine which violation has occurred (right-right, right-left, or
        right-zero) and fix it accordingly;

        Return a pointer to the node that took p's place in the tree;
    }
    Otherwise
    {
        Return p;
    }
}

```

The member function (AvlTree.h, line 75)

```
Node* _balanceOnRightPruning(Node* p);
```

balances the node p following the pruning (reduction in height of) its right subtree. This procedure is the mirror image `_balanceOnLeftPruning`:

```
template <class Key, class Mapped, class Predicate>
typename AvlTree<Key, Mapped, Predicate>::Node*
AvlTree<Key, Mapped, Predicate>::_balanceOnRightPruning(Node* p)
{
    Decrease p's balance factor by 1 (because p's right subtree was pruned);

    If p's balance factor is -2
    {
        Determine which violation has occurred (left-left, left-right, or
        left-zero) and fix it accordingly;

        Return a pointer to the node that took p's place in the tree;
    }
    Otherwise
    {
        Return p;
    }
}
```

Both functions are defined in lines 62-106 (memberFunctions\_4.h). Consider, for example

```

          8 (0)
         /   \
        4 (-1)   12 (-1)
       / \     / \
      2 (0) 6 (0) 10 (1) 14 (0)
     / \   / \
    1 (0) 3 (0) t-> 11 (0)

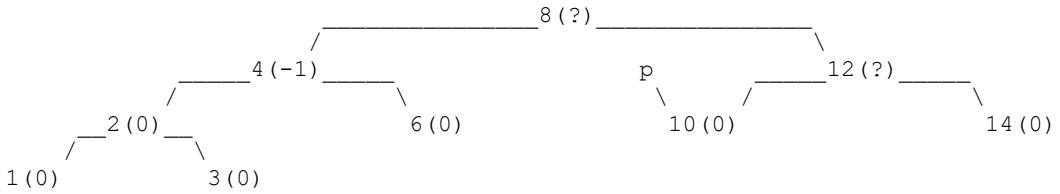
// Erase n11

          8 (?)
         /   \
        4 (-1)   12 (?)
       / \     / \
      2 (0) 6 (0) 10 (?) 14 (0)
     / \   / \
    1 (0) 3 (0)

// p's right subtree was pruned, so balance p via _balanceOnRightPruning
p = _balanceOnRightPruning(p);

// The function decrements n10->balanceFactor (which becomes 0) and returns
// a pointer to n10

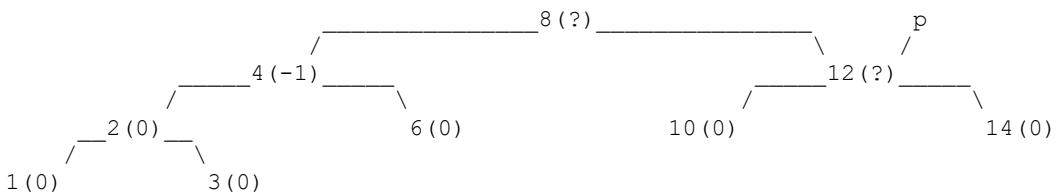
// p still points to n10
```



```

// p->balanceFactor became 0, indicating that the height of p was
// reduced by the erasure of t

// The balanceFactor of the next ancestor (n12) must therefore be updated
  
```



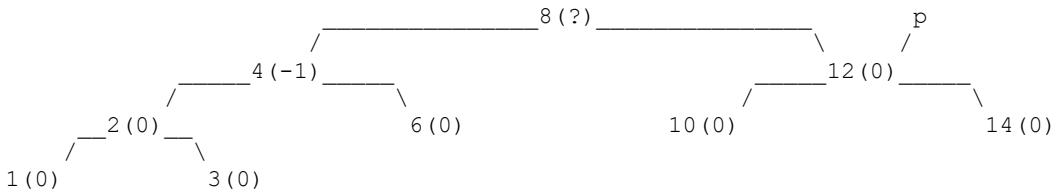
```

// p's left subtree was pruned, so balance p via _balanceOnLeftPruning

p = _balanceOnLeftPruning(p);

// The function increments n12->balanceFactor (which becomes 0) and returns
// a pointer to n12

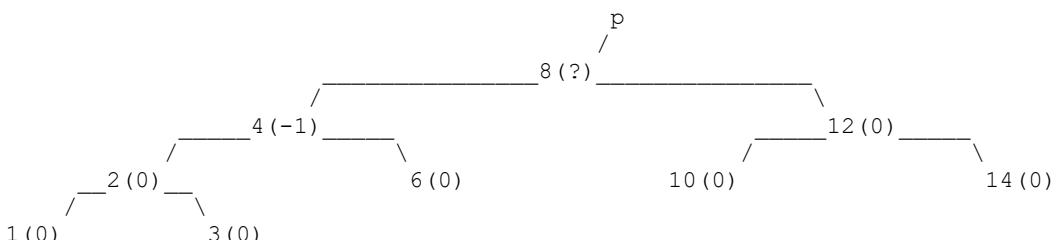
// p still points to n12
  
```



```

// p->balanceFactor became 0, indicating that the height of p was
// reduced by the erasure of t

// The balanceFactor of the next ancestor (n8) must therefore be updated
  
```



```

// p's right subtree was pruned, so balance p via _balanceOnRightPruning
  
```

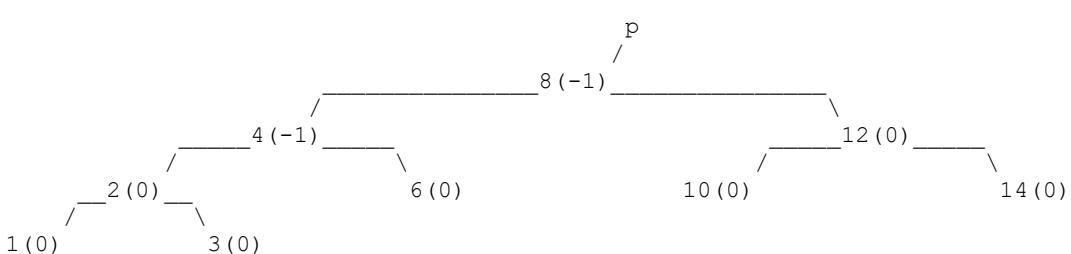
```

p = _balanceOnRightPruning(p);

// The function decrements n8->balanceFactor (which becomes -1) and returns
// a pointer to n8

// p still points to n8

```



```

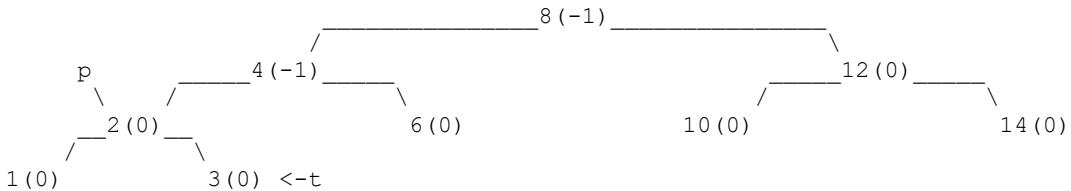
// p->balanceFactor became -1, indicating that the height of p was
// unchanged by the erasure of t

```

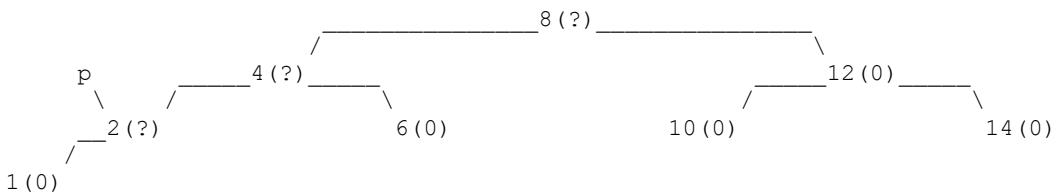
```

// The balanceFactors of the remaining ancestors (n/a) are therefore
// unchanged, so no further balancing is necessary

```



```
// Erase n3
```



```
// p's right subtree was pruned, so balance p via _balanceOnRightPruning
```

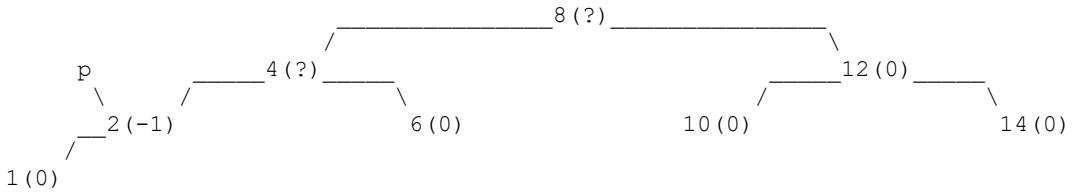
```
p = _balanceOnRightPruning(p);
```

```

// The function decrements n2->balanceFactor (which becomes -1) and returns
// a pointer to n2

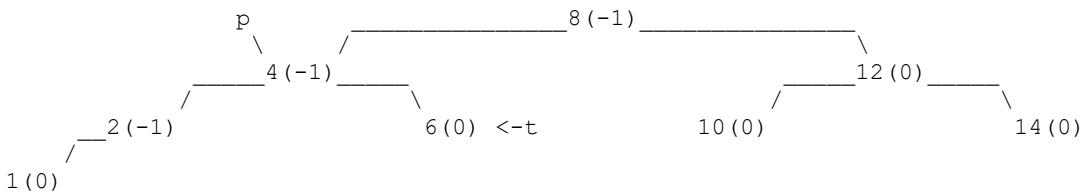
```

```
// p still points to n2
```

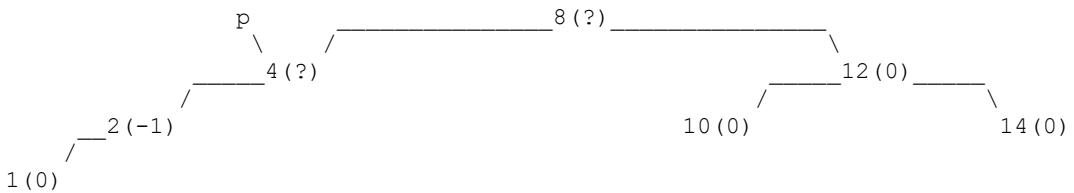


// p->balanceFactor became -1, indicating that the height of p was  
// unchanged by the erasure of t

// The balanceFactors of the remaining ancestors (n4 and n8) are therefore  
// unchanged, so no further balancing is necessary



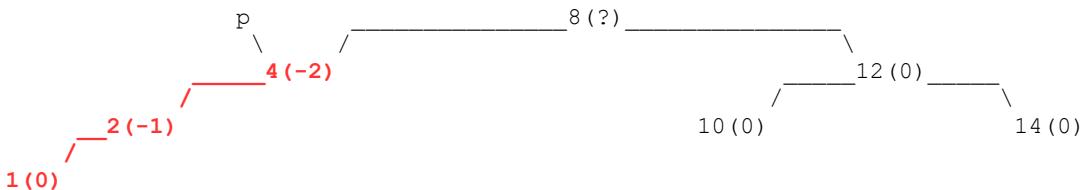
// Erase n6



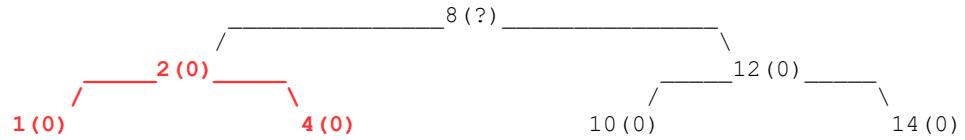
// p's right subtree was pruned, so balance p via `_balanceOnRightPruning`

`p = _balanceOnRightPruning(p);`

// The function decrements n4->balanceFactor (which becomes -2)...

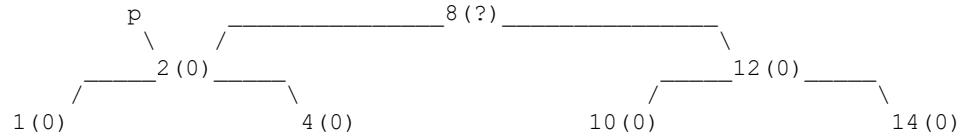


// ...corrects the resulting left-left violation at n4...



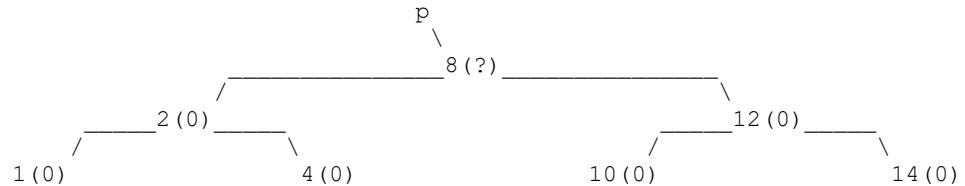
// ...and returns a pointer to the node that took n4's place in  
// the tree

// p now points to n2



// p->balanceFactor became 0, indicating that the height of p was  
// reduced by the erasure of t

// The balanceFactor of the next ancestor (n8) must therefore be updated

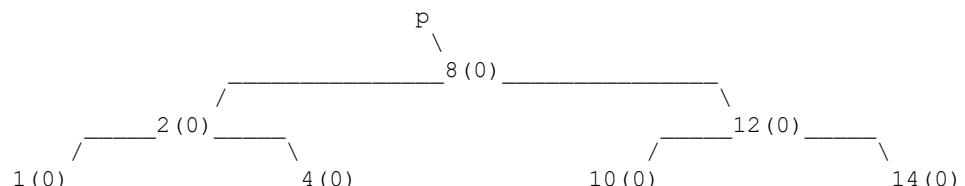


// p's left subtree was pruned, so balance p via \_balanceOnLeftPruning

p = \_balanceOnLeftPruning(p);

// The function increments n8->balanceFactor (which becomes 0) and returns  
// a pointer to n8

// p still points to n8



// p->balanceFactor became 0, indicating that the height of p was  
// reduced by the erasure of t

// The balanceFactor of the next ancestor must therefore be updated,  
// but there are no remaining ancestors so the balancing is complete

The member function erase (AvlTree.h, line 63 / memberFunctions\_4.h, lines 7-42) is nearly identical

to that of BinaryTree. The only difference is that before detaching the trash node, it saves a pointer to trash's parent and a boolean flag indicating whether the parent's left or right child is being erased (lines 28-29):

```
Node* trashParent = trash->parent;
bool leftSubtreeWasPruned = isLeftChild(trash);
```

If trash is the left child of its parent (trashParent), then leftSubtreeWasPruned will be true, indicating that trashParent's left subtree is being pruned.

If trash is the right child of its parent (trashParent), then leftSubtreeWasPruned will be false, indicating that trashParent's right subtree is being pruned.

Line 39 then rebalances the tree:

```
_balanceOnErasure(trashParent, leftSubtreeWasPruned);
```

The private member function (AvlTree.h, line 73)

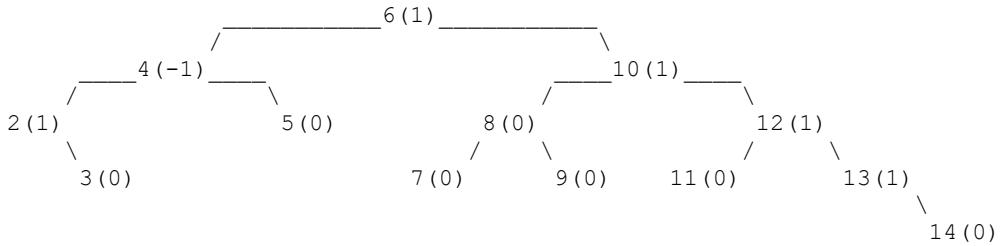
```
void _balanceOnErasure(Node* trashParent, bool leftSubtreeWasPruned);
```

balances the entire tree following the erasure of a node, where trashParent is the parent of the erased node. leftSubtreeWasPruned is a boolean flag indicating whether the left subtree of trashParent was pruned (true) or the right subtree of trashParent was pruned (false). The pseudocode for the function is

```
template <class Key, class Mapped, class Predicate>
void AvlTree<Key, Mapped, Predicate>::_balanceOnErasure(Node* trashParent,
    bool leftSubtreeWasPruned)
{
    For each ancestor p of the erased node, beginning at trashParent
    {
        If the left subtree of p was pruned,
            p = _balanceOnLeftPruning(p);
        Otherwise
            p = _balanceOnRightPruning(p);

        If p's balance factor is -1 or 1
        {
            p's height is unchanged;
            The balance factors of the remaining ancestors are therefore unchanged,
                so no further balancing is necessary;
        }
        Otherwise
        {
            p's balance factor is 0, indicating that p's height has decreased by
                1 level;
            The balance factor of the next ancestor must therefore be updated, so
                continue to p's parent;
        }
    }
}
```

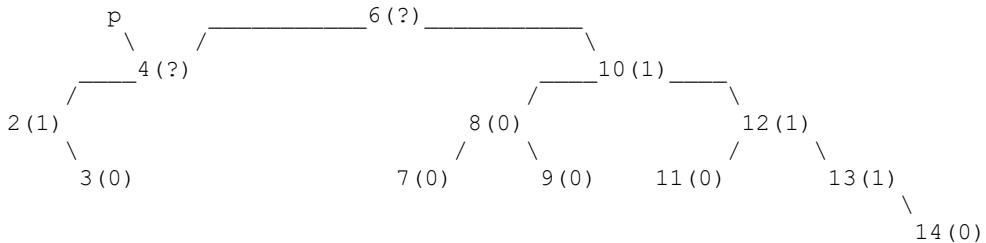
The function is defined in lines 44-60 (memberFunctions\_4.h). Given the tree



for example, suppose that node 5 is erased. trashParent points to node 4 and leftSubtreeWasPruned is initialized to false (because the trash node, 5, was the right child of its parent). `_balanceOnErasure` performs the following operations:

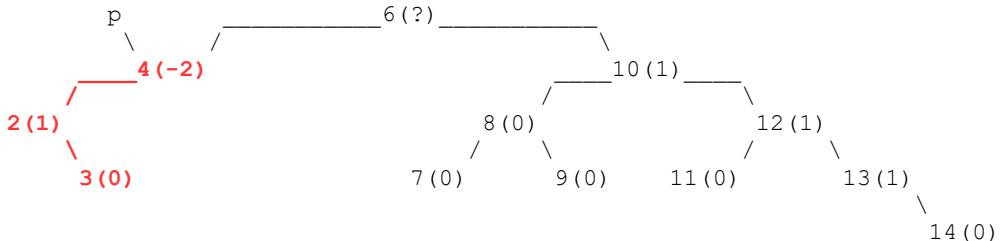
```
// leftSubtreeWasPruned is false
```

Iteration 1:

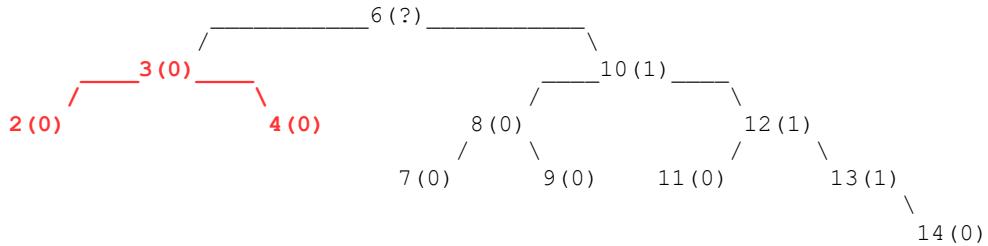


```
p = _balanceOnRightPruning(p);
```

```
// _balanceOnRightPruning decrements p->balanceFactor...
```

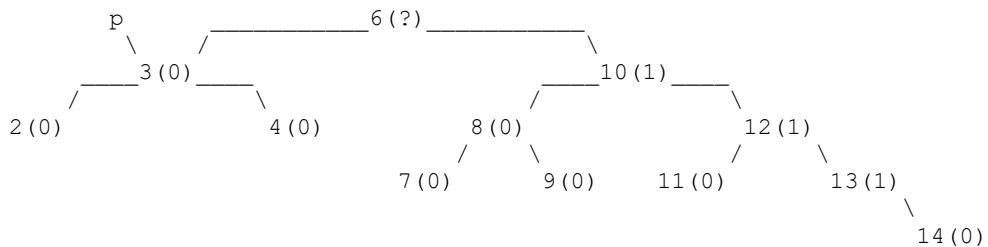


```
// ...corrects the resulting left-right violation at n4...
```



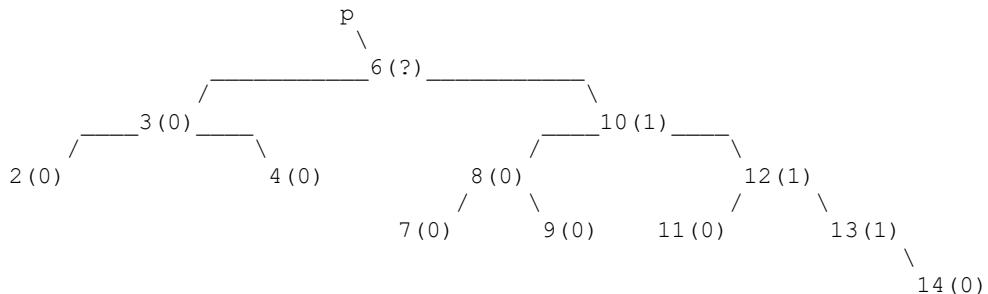
```

// ...and returns a pointer to the node that took n4's place in the tree
// p now points to n3
  
```



```

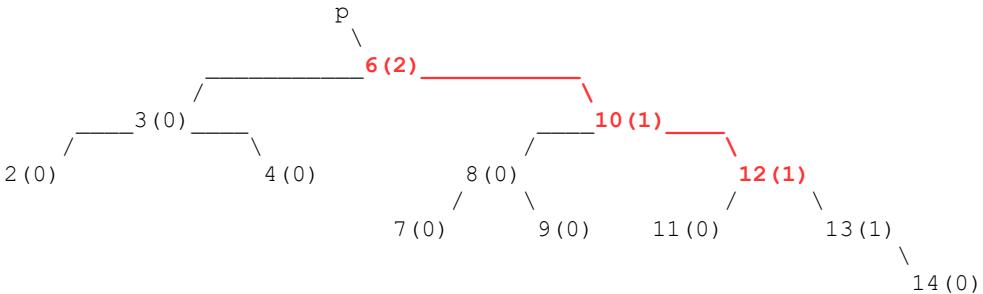
leftSubtreeWasPruned = bt::isLeftChild(p);      // leftSubtreeWasPruned is
// true
p = p->parent;
  
```



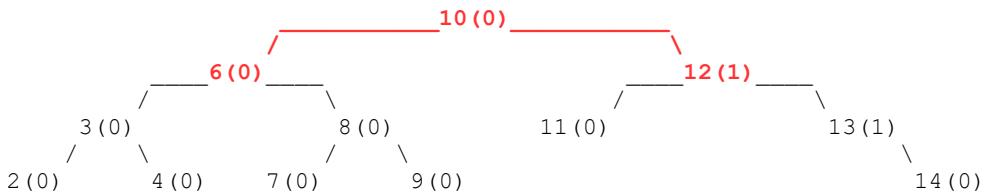
Iteration 2:

```

p = _balanceOnLeftPruning(p);
// _balanceOnLeftPruning increments p->balanceFactor...
  
```

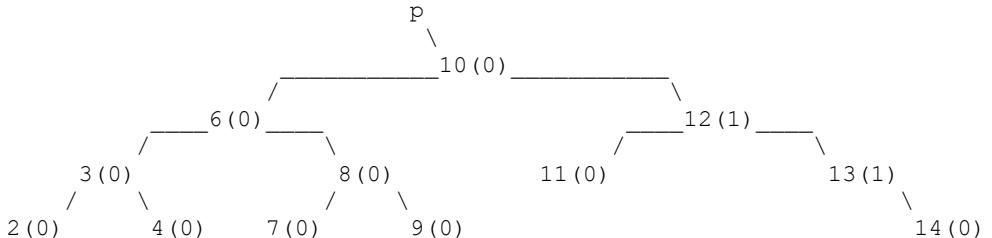


// ...corrects the resulting right-right violation at n6...



// ...and returns a pointer to the node that took n6's place in the tree

// p now points to n10



leftSubtreeWasPruned = bt::isLeftChild(p); // leftSubtreeWasPruned is  
// false

p = p->parent;

// p is null, so the loop terminates

The function (main.cpp, line 9)

```
char getCommandFromUser();
```

prompts the user to enter a single-character command and returns the user's input value (lines 63-70).

The function (line 10)

```
int getKeyValueFromUser();
```

prompts the user to enter an integer key value and returns the user's input value (lines 72-79).

main (lines 13-59) is nearly identical to that of the previous chapter, but now includes an additional branch that handles erasure. If the user enters e (lines 31-38), the program prompts the user for the desired key value and locates it in the tree. If the key value was found, it is erased from the tree; otherwise, the program displays the message “Key value not found”.

For the following sample run, refer to the diagrams on the corresponding pages. Note, however, that the tree must be cleared before inserting each set of key values.

Inserting the keys {3, 1, 5, 7} and erasing node 1 (p. 475) generates the output (p. 454)

```

node 3           // Right-right violation corrected
  parent 5
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor 0
node 7
  parent 5
  balanceFactor 0

```

Inserting the keys {3, 0, 7, 5} and erasing node 0 (p. 475) generates the output (p. 458)

```

node 3           // Right-left violation (Case 1) corrected
  parent 5
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor 0
node 7
  parent 5
  balanceFactor 0

```

Inserting the keys {3, 1, 7, 0, 5, 9, 4} and erasing node 0 (p. 476) generates the output (pp. 458-459)

```

node 1           // Right-left violation (Case 2) corrected
  parent 3
  balanceFactor 0
node 3
  parent 5
  left 1
  right 4
  balanceFactor 0
node 4
  parent 3
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor 0

```

```

node 7
  parent 5
  right 9
  balanceFactor 1
node 9
  parent 7
  balanceFactor 0

```

Inserting the keys {3, 1, 7, 0, 5, 9, 6} and erasing node 0 (p. 476) generates the output (pp. 459-460)

```

node 1           // Right-left violation (Case 3) corrected
  parent 3
  balanceFactor 0
node 3
  parent 5
  left 1
  balanceFactor -1
node 5
  left 3
  right 7
  balanceFactor 0
node 6
  parent 7
  balanceFactor 0
node 7
  parent 5
  left 6
  right 9
  balanceFactor 0
node 9
  parent 7
  balanceFactor 0

```

Inserting the keys {3, 1, 5, 4, 7} and erasing node 1 (pp. 476-477) generates the output

```

node 3           // Right-zero violation corrected
  parent 5
  right 4
  balanceFactor 1
node 4
  parent 3
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor -1
node 7
  parent 5
  balanceFactor 0

```

Inserting the keys {7, 5, 9, 3} and erasing node 9 (p. 477) generates the output (p. 452)

```

node 3           // Left-left violation corrected

```

```

parent 5
balanceFactor 0
node 5
left 3
right 7
balanceFactor 0
node 7
parent 5
balanceFactor 0

```

Inserting the keys {7, 3, 10, 5} and erasing node 10 (pp. 477-478) generates the output (pp. 455-456)

```

node 3           // Left-right violation (Case 1) corrected
parent 5
balanceFactor 0
node 5
left 3
right 7
balanceFactor 0
node 7
parent 5
balanceFactor 0

```

Inserting the keys {7, 3, 9, 1, 5, 10, 4} and erasing node 10 (p. 478) generates the output (pp. 456-457)

```

node 1           // Left-right violation (Case 2) corrected
parent 3
balanceFactor 0
node 3
parent 5
left 1
right 4
balanceFactor 0
node 4
parent 3
balanceFactor 0
node 5
left 3
right 7
balanceFactor 0
node 7
parent 5
right 9
balanceFactor 1
node 9
parent 7
balanceFactor 0

```

Inserting the keys {7, 3, 9, 1, 5, 10, 6} and erasing node 10 (p. 478) generates the output (p. 457)

```

node 1           // Left-right violation (Case 3) corrected
parent 3
balanceFactor 0

```

```

node 3
  parent 5
  left 1
  balanceFactor -1
node 5
  left 3
  right 7
  balanceFactor 0
node 6
  parent 7
  balanceFactor 0
node 7
  parent 5
  left 6
  right 9
  balanceFactor 0
node 9
  parent 7
  balanceFactor 0

```

Inserting the keys {7, 5, 9, 3, 6} and erasing node 9 (p. 479) generates the output

```

node 3          // Left-zero violation corrected
  parent 5
  balanceFactor 0
node 5
  left 3
  right 7
  balanceFactor 1
node 6
  parent 7
  balanceFactor 0
node 7
  parent 5
  left 6
  balanceFactor -1

```

Inserting the keys {6, 4, 10, 2, 5, 8, 12, 3, 7, 9, 11, 13, 14} and erasing node 5 (p. 486) generates the output (p. 488)

```

node 2          // Left-right and right-right violations corrected
  parent 3
  balanceFactor 0
node 3
  parent 6
  left 2
  right 4
  balanceFactor 0
node 4
  parent 3
  balanceFactor 0
node 6
  parent 10

```

```
left 3
right 8
balanceFactor 0
node 7
parent 8
balanceFactor 0
node 8
parent 6
left 7
right 9
balanceFactor 0
node 9
parent 8
balanceFactor 0
node 10
left 6
right 12
balanceFactor 0
node 11
parent 12
balanceFactor 0
node 12
parent 10
left 11
right 13
balanceFactor 1
node 13
parent 12
right 14
balanceFactor 1
node 14
parent 13
balanceFactor 0
```



# Part 16: Time Complexity

## 16.1: Iterator Categories

*Source folders*

*distance  
iteratorTags*

*Chapter outline*

- Random access vs. bidirectional iterators
- Overloading functions by iterator category

The following table summarizes the categories of the iterators discussed / implemented previously:

Iterator type	iterator_category (alias of type)	Chapter
ListIter	std::bidirectional_iterator_tag	10.3
Pointer (T*)	std::random_access_iterator_tag	11.2
RingIter	std::random_access_iterator_tag	12.4
BinaryTreeIter	std::bidirectional_iterator_tag	14.6

An ordinary pointer (int\*, double\*, etc.) or RingIter is known as a “random access iterator” because it can be pointed to any element in the underlying sequence by simply applying an integer offset:

```
Array<int>::iterator i;
Ring<double>::const_iterator k;

// ...

i += 5;      // Point to the element 5 elements away (forward)
k -= 10;     // Point to the element 10 elements away (reverse)
```

More specifically, this is known as a “constant time” operation: the time required to offset a random access iterator by n elements is constant for all values of n. Offsetting an iterator by 1000 elements, for example, takes just as long as offsetting it by 1 element.

A ListIter or BinaryTreeIter is known as a “bidirectional iterator” because it can only be pointed to the next or previous element in the underlying sequence:

```
List<double>::const_iterator p;
AvlTree<string, int>::iterator q;

// ...
```

```
++p;           // Point to the next element
--q;           // Point to the previous element
```

The only way to point these iterators to an element n elements away is to increment or decrement them n times:

```
for (int n = 0; n != 5; ++n)      // Point to the element 5 elements away
    ++p;                         // (forward)

for (int n = 0; n != 10; ++n)     // Point to the element 10 elements away
    --q;                         // (reverse)
```

This is known as a “linear time” operation: the time required to offset a bidirectional iterator by n elements is linearly proportional to n. Offsetting an iterator by 1000 elements, for example, takes 1000 times longer than offsetting it by 1 element.

Iterator tags allow a function to be overloaded by iterator category, which allows the compiler to automatically select the most efficient version for a given iterator type. Consider, for example, the functions (distance.h, lines 12-20)

```
template <class Iter>
typename std::iterator_traits<Iter>::difference_type _distance(Iter begin,
    Iter end,
    std::bidirectional_iterator_tag iteratorTag);

template <class Iter>
typename std::iterator_traits<Iter>::difference_type _distance(Iter begin,
    Iter end,
    std::random_access_iterator_tag iteratorTag);
```

Both functions return the distance (number of elements) from begin to end, and their return type is the Iter's difference\_type.

The functions differ in the type of their third parameter, iteratorTag: the first version takes an argument of type `bidirectional_iterator_tag`, while the second takes an argument of type `random_access_iterator_tag`.

The `bidirectional_iterator_tag` and `random_access_iterator_tag` classes do not contain any member functions or data; their sole purpose is to provide different types for function overloading. The bidirectional version of `_distance` counts the number of elements by traversing the entire sequence one element at a time (lines 31-45); the random access version simply subtracts begin from end (lines 47-53).

The function (lines 8-10)

```
template <class Iter>
typename std::iterator_traits<Iter>::difference_type distance(Iter begin,
    Iter end);
```

returns the distance from begin to end by simply calling `_distance` (lines 22-29):

```
template <class Iter>
inline typename std::iterator_traits<Iter>::difference_type distance(
    Iter begin,
    Iter end)
{
    return _distance(begin,
        end,
        std::iterator_traits<Iter>::iterator_category());
}
```

## The expression

```
std::iterator_traits<Iter>::iterator_category()
```

constructs a temporary, unnamed iterator tag, which is passed to `_distance`. The compiler uses the type of this object (i.e. the type of the iterator tag) to select the appropriate overload of `_distance`.

Lines 12-19 (main.cpp) construct a `Ring<int>` `r` and a `List<int>` `t`, each containing the elements  $\{0, 1, 2, 3, 4\}$ . Lines 21-22 then use the `distance` function to print the number of elements in each container. The expression (line 21)

```
dss::distance(r.begin(), r.end())
```

is equivalent to

```
dss::distance<Ring<int>::iterator>(r.begin(), r.end())
```

The compiler generates the function

```
inline Ring<int>::iterator::difference_type distance(  
    Ring<int>::iterator begin,  
    Ring<int>::iterator end)  
{  
    return _distance(begin,  
        end,  
        Ring<int>::iterator::iterator_category());  
}
```

RingIter's difference\_type and iterator\_category are aliases of the types int and random access iterator tag, so this is equivalent to

```
inline int distance(
    Ring<int>::iterator begin,
    Ring<int>::iterator end)
{
    return _distance(begin,
                     end,
                     std::random_access_iterator_tag()); // Unnamed object of type
} // random access iterator tag
```

The compiler then selects the random access version of `_distance`, generating the function

```
Ring<int>::iterator::difference_type _distance(Ring<int>::iterator begin,
    Ring<int>::iterator end,
    std::random_access_iterator_tag iteratorTag)
{
    return end - begin;
}
```

Similarly, the expression (line 22)

```
dss::distance(t.begin(), t.end())
```

is equivalent to

```
dss::distance<List<int>::iterator>(t.begin(), t.end())
```

The compiler generates the function

```
inline List<int>::iterator::difference_type distance(
    List<int>::iterator begin,
    List<int>::iterator end)
{
    return _distance(begin,
        end,
        List<int>::iterator::iterator_category());
}
```

`ListIter`'s `difference_type` and `iterator_category` are aliases of the types `int` and `bidirectional_iterator_tag`, so this is equivalent to

```
inline int distance(
    List<int>::iterator begin,
    List<int>::iterator end)
{
    return _distance(begin,
        end,
        std::bidirectional_iterator_tag()); // Unnamed object of type
                                            // bidirectional_iterator_tag
}
```

The compiler then selects the bidirectional version of `_distance`, generating the function

```
List<int>::iterator::difference_type _distance(List<int>::iterator begin,
List<int>::iterator end,
std::bidirectional_iterator_tag iteratorTag)
{
    List<int>::iterator::difference_type totalElements = 0;

    while (begin != end)
    {
        ++totalElements;
        ++begin;
    }

    return totalElements;
}
```

Lines 21-22 generate the output

```
r contains 5 elements
t contains 5 elements
```

The namespace qualifier (dss::) in lines 21-22 is necessary because the C++ Standard Library also contains a distance function (declared in namespace std).

The Standard Library header <iterator> contains additional types of iterator tags. Note, however, that taking full advantage of iterator tags requires an understanding of inheritance, which is beyond the scope of this book. A list of additional resources is available at

<http://www.cppdatastructures.com/>



## 16.2: Big O Notation

### *Chapter outline*

- Constant, linear, and logarithmic time
- Big O notation

The “time complexity” of an operation is the relationship between the time required to perform that operation and the size of a data structure (i.e. the number of contained elements). Time complexity, which is represented using “Big O notation,” can be constant, linear, or logarithmic.

`push_back` on a `List`, for example, takes constant time: whether the `List` contains 1, 100, or 1000 elements, the operation always takes the same amount of time. In Big O notation, constant time is represented as  $O(1)$ .

Relocating a `Vector` is an example of a linear time operation: the time required to perform the operation is linearly proportional to the size of the container. A 10-element `Vector` copies 10 elements, a 100-element `Vector` copies 100 elements, etc. In Big O notation, linear time is represented as  $O(n)$ , where  $n$  denotes the size of the container.

Searching an `AVL tree` takes logarithmic time: the time required to perform the operation is logarithmically proportional to the size of the container. In an `AVL tree`, the maximum number of elements that must be examined is equal to the height of the tree (number of levels), which is logarithmically proportional to the number of elements. A 7-element tree, for example, has at most 4 levels ( $\log_2(7) + 1$ ), while a 255-element tree has at most 9 levels ( $\log_2(255) + 1$ ). In Big O notation, logarithmic time is represented as  $O(\log(n))$ .

The following tables list the average and worst-case time complexity (where applicable) of the various operations. `push_front` on a `Ring`, for example, takes constant time on average, while the worst-case scenario takes linear time.

Container	<code>push_front</code>	<code>pop_front</code>	<code>push_back</code>	<code>pop_back</code>
<code>Vector</code>	----	----	$O(1) / O(n)$	$O(1)$
<code>List</code>	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<code>Ring</code>	$O(1) / O(n)$	$O(1)$	$O(1) / O(n)$	$O(1)$
<code>MultiRing</code>	$O(1) / O(n)$	$O(1)$	$O(1) / O(n)$	$O(1)$

Container	<code>find</code>	Random element access
<code>Vector</code>	----	$O(1)$
<code>List</code>	----	----
<code>Ring</code>	----	$O(1)$
<code>MultiRing</code>	----	$O(1)$
<code>BinaryTree</code>	$O(\log(n)) / O(n)$	----
<code>AvlTree</code>	$O(\log(n))$	----

Container	insert	erase
Vector	$O(n)$	$O(n)$
List	$O(1)$	$O(1)$
Ring	$O(n)$	$O(n)$
MultiRing	$O(n)$	$O(n)$
BinaryTree	$O(\log(n)) / O(n)$	$O(1)^*$
AvlTree	$O(\log(n))$	$O(\log(n))^{**}$

Iterator	Dereference	Increment/decrement	Addition/subtraction
Vector	$O(1)$	$O(1)$	$O(1)$
List	$O(1)$	$O(1)$	----
Ring	$O(1)$	$O(1)$	$O(1)$
MultiRing	$O(1)$	$O(1)$	$O(1)$
BinaryTree	$O(1)$	$O(1)$	----
AvlTree	$O(1)$	$O(1)$	----

\*Recall that BinaryTree's erase method takes an iterator to an existing element, which has already been located via find. The erase method itself can therefore detach the desired node by simply updating the appropriate links (which is a constant-time operation).

\*\*As with BinaryTree, detaching the desired node is a constant-time operation. The rebalancing algorithm (which must climb the tree), however, is a logarithmic-time operation.

# Index

## Operators

### arithmetic

- addition (+), 5
- addition assignment (+=), 17
- assignment (=), 1-2
- compound assignment, 17
- decrement (--), 15
- division (/), 6
- division assignment (/=), 18
- increment (++), 12
- modulo (%), 6
- modulo assignment (%=), 19
- multiplication (\*), 6
- multiplication assignment (\*=), 18
- subtraction (-), 5
- subtraction assignment (-=), 18

### array subscript, primitive ([]), 76-77

### comment

- single-line (//), 3
- multiple-line /\* \*/), 3

### comparison, *See* relational

### function call (), 145

### logical

- and (&&), 27
- not (!), 28
- or (||), 28

### member access operator, class .(), 119, 121

### memory allocation

- delete, 186
- new, 185
- sizeof, 185

### overloading, *See* operator overloading

### pointer / iterator

- addition (+), 85

### pointer / iterator (*continued*)

- addition assignment (+=), 86
- address-of (&), 68
- array subscript ([]), 86
- comparison (==, !=, <, <=, >, >=), 87
- decrement (--), 86, 94-97
- increment (++), 86, 94-97
- indirection (\*)
- dereference, 69, 86
- pointer to, 67
- member access (->), 121
- subtraction (-), 87
- subtraction assignment (-=), 86

### reference (&), 99

### relational (comparison)

- equal to (==), 21-25
- greater than (>), 21-25
- greater than or equal to (>=), 21-25
- less than (<), 21-25
- less than or equal to (<=), 21-25
- not equal to (!=), 21-25

### scope resolution (::), 56, 157

### stream

- insertion (<<), 1
- extraction (>>), 9

### type conversion (typecast)

- const\_cast, 245
- static\_cast, 185, 194

# A

### alias, 99, 156

### Allocator, 183-188

### architecture, 67

### Array (class), 155-163, 277

### array (primitive), 75-77

### automatic object, 176, 183

### AVL tree, definition, 447-448

### AvlTree (class)

#### back, 449

#### \_balanceOnErasure, 485-488

#### \_balanceOnInsertion, 464

#### \_balanceOnLeftPruning, 479-484

AvlTree (class) (*continued*)  
  `_balanceOnRightPruning`, 479-484  
  `begin`, 449  
  `clear`, 449  
  `copy constructor`, 449  
  `_copyNode`, 449  
  `_createNode`, 449  
  `default constructor`, 449  
  `_destroyAllNodes`, 449  
  `_destroyNode`, 449  
  `destructor`, 449  
  `empty`, 449  
  `end`, 449  
  `erase`, 484-485  
  `find`, 449  
    `_fixLeftLeft`, 452-453, 477  
    `_fixLeftRight`, 455-457, 477-478  
    `_fixLeftZero`, 478-479  
    `_fixRightLeft`, 457-460, 475-476  
    `_fixRightRight`, 454-455, 475  
    `_fixRightZero`, 476-477  
  `front`, 449  
  `head`, 449  
  `insert`, 464-465  
  member types, 449  
  `operator=`, 449  
    `_overwriteElement`, 449  
  `rbegin`, 449  
  `rend`, 449  
  `root`, 449  
  `size`, 449  
  `tail`, 449  
AvlTreeNode, 447-448

## B

bidirectional iterator, 495-496  
Big O notation, 501  
binary tree  
  2-child node, detaching, 403-408  
  height, 338, 427-428  
  in-order traversal  
    iterative, 363-366  
    recursive, 355-361  
  search, 338-339  
  subtree, 337-338  
binary tree utility functions (bt library)  
  `attachNewNode`, 383-385

binary tree utility functions (bt library) (*continued*)  
  `detachLeafNode`, 395-399  
  `detachOneChildNode`, 399-403  
  `findInsertionPoint`, 379-383  
  `findNode`, 371-374  
  `hasLeftChild`, 342-343  
  `hasPredecessor`, 344  
  `hasRightChild`, 342-343  
  `hasSuccessor`, 344  
  `isLeaf`, 342-343  
  `isLeftChild`, 341-342  
  `isRightChild`, 341-342  
  `isRoot`, 340-341  
  `key`, 346-347  
  `leftmostNode`, 344  
  `rotateLeft`, 436-441  
  `rotateRight`, 431-436  
  `rightmostNode`, 344-345

BinaryTree (class)  
  `back`, 378  
  `begin`, 378  
  `clear`, 417  
  `copy constructor`, 421-423  
  `_copyNode`, 417-421  
  `_createNode`, 378  
  `default constructor`, 378  
  `_destroyAllNodes`, 378  
  `_destroyNode`, 378  
  `destructor`, 378  
  `empty`, 378  
  `end`, 378  
  `erase`, 408-413  
  `find`, 378  
  `front`, 378  
  `head`, 378  
  `insert`, 386-392  
  member types, 378  
  `operator=`, 423-424  
    `_overwriteElement`, 406-408  
  `rbegin`, 378  
  `rend`, 378  
  `root`, 378  
  `rotate`, 441  
  `size`, 378  
  `tail`, 378

BinaryTreeIter, 367  
BinaryTreeNode, 337-340  
bool, 2  
boolean expression  
  compound, 27

**boolean expression (*continued*)**

individual, 21

**boolean variable**, 41-42

**break**, 38-40

**bubble sort**

array subscript implementation, 77-83

pointer arithmetic implementation, 90-94

**byte**, 67-68, 85-86

# C

**carriage return**, 1

**char**, 2-3

**checkBtIntegrity**, 413

**cin**, 9, 130-131

**class**, 117, 140

**class template**, 145

**<cmath>**, 60

**comment**, 3

**compile time**, 64, 173

**conditional statement**

if, 21-22

if...else, 22

if...else if, 23-24

nested, 24, 35

**const**

correctness, 105-106

double, 60

int, 76

iterator, 159

overloading, 120-122, 159-160, 245

pointer, 105, 121-122

reference, 105, 121-122

unsigned int, 156

**const\_cast**, 245

**constant time**, 495, 501

**constructor**

copy, 131-133, 353

default, 118, 150, 235

explicit, 263, 266-267

template, 264, 267, 334-335

**ConstIter**

bidirectional member functions, 241-245

random access operators, 304-305

**control**, 27-28

**cout**, 1, 129-130

# D

**data member**, 117, 317

**default template argument**, 377

**#define**, 66

**delete**, 186

**DemoBinaryTree**

back, 350

begin, 367-368

copy constructor, 353

**\_createAllNodes**, 350-351

**\_createNode**, 350-351

default constructor, 351, 353, 368

**\_destroyAllNodes**, 353, 369

**\_destroyNode**, 351

destructor, 353

empty, 350

end, 367-368

find, 374-375

front, 350

head, 350

member types, 349-350, 367

operator=, 353

rbegin, 367-368

rend, 367-368

root, 350

**\_setParentAndChildLinks**, 352-353

size, 350

tail, 350

**deque**, *See double-ended queue*

**destructor**, 176-177, 183, 185

**distance**, 496-499

**double**, 2

**double-ended queue**

linked list, 213

multi-block, 315

single-block, 279

**dss**, 71

**dynamic array**, *See MultiRing, Ring, Vector*

**dynamically-allocated object**, 183-188

# E

**else**, *See conditional statement*

**#endif**, 66

**endl**, 1

## explicit

- constructor, 263, 266-267
- type conversion
  - double to int, 185, 194
  - non-const to const pointer, 245

**F**

free store, 183

friend

- class, 234, 237
- function, 130, 178

function

- basic syntax, 49-51
- const member, 120-122, 159-160, 245
- const overloading, 159-160, 245
- friend, 130, 178
- inline, 65-66, 119, 139
- non-const member, 117-120
- object, 145-154
- overloading, 63-64, 159-160, 496
- return type, 49, 71, 159
- template, 137-140

**G**

global

- namespace, 55
- variable, 59-60, 62

Greater, 146-147

**H**

header file, 3, 65-66

heap, 183

hexadecimal, 67

**I**if, *See* conditional statement

#ifndef, 66

## implicit conversion

- disallowing, 263, 266-267
- double to int, 194
- iterator to const iterator, 244
- pointer to const pointer, 244
- reference to const reference, 244, 292
- T to Traceable<T>, 177

#include, 3-4

include guard, 66, 139

initializer list, constructor, 118

inline function, 65-66, 119, 139

inOrderPredecessor, 363-365

inOrderSuccessor, 365-366

insertionSort

- generic, 141-142, 148
- int\*, 107-115
- Student\*, 134

int, 2

&lt;iostream&gt;, 4, 9

isReflexivelyEqual, 153-154

&lt;iterator&gt;, 499

iterator

- bidirectional vs. random access, 495
- definition, 140
- tags, 496

iterator\_traits, 242-243, 269-272

**J****K**

key-mapped pair, 333-335

**L**

l-value, 11

Less, 145-146

lifetime,

- of object, 59-62

tracing, 173-181

&lt;limits&gt;, 184

linear time, 496, 501

linker, 65

List

- back, 215

begin, 237-238, 245

List (*continued*)

- clear, 227
- copy constructor, 247
- `_createNode`, 215-216
- default constructor, 215
- `_destroyNode`, 222
- `_destroyAllNodes`, 222-224
- destructor, 225
- empty, 215
- end, 237-238, 245
- erase, 256-258
- front, 215
- head, 214
- insert, 253-256
- member types, 215, 237, 265
- `operator=`, 247-250
- `pop_back`, 227-229
- `pop_front`, 229-232
- `push_back`, 216-219
- `push_front`, 219-222
- `rbegin`, 266
- `rend`, 265
- size, 215
- tail, 214

## ListIter

- constructors, 235
- member types, 234-235
- operators, 235-237
- `_pointToNextNode`, 235
- `_pointToPreviousNode`, 235-236

## ListNode, 213-214

logarithmic time, 501

## loop

- `break`, 38-40
- `for`, 31-33
- infinite, 40, 62
- nested, 35-38, 78
- true (condition), 38
- `while`, 33-35

**M**

machine code, 65  
`makePair`, 334-335  
 mapped value, 333  
 member  
   data, 117, 317

member (*continued*)

- function
  - const, 120-122, 159-160, 245
  - non-const, 117-120
- type, 158-159

## memory

- address, 67
- leak, 183

## method, 1

`MultiRing`

- back, 317-318
- `begin`, 327
- capacity, 281
- clear, 328-329
- copy constructor, 331
- `_createPage`, 318
- default constructor, 317
- `_destroyAllPages`, 319-320
- `_destroyPage`, 319
- destructor, 320
- `_element`, 323-325
- empty, 317
- end, 327
- erase, 331
- front, 317-318
- insert, 331
- member types, 316-317, 327
- `operator[]`, 325
- `operator=`, 331
- page, 315-316
- `pop_back`, 327-328
- `pop_front`, 327-328
- `push_back`, 318-319
- `push_front`, 323
- `rbegin`, 327
- `rend`, 327
- size, 317

**N**

`\n` (newline character), 3, 10  
 namespace
 

- declaration, 53
- global, 55
- qualification, 56

 new, 185  
`nullptr`, 68, 70, 87  
`numeric_limits`, 184

# O

## object

- automatic, 176, 183
- dynamically-allocated, 183-188
- file, 65
- lifetime of, 59-62, 173-181
- unnamed, 150, 174, 185, 497

## one-definition rule (ODR), 65, 139

## operator overloading

- addition, 304
- addition assignment, 304
- array subscript, 161-162
- assignment, 132-133, 353
- comparison (relational), 125-128
- decrement, 236-237
- dereference, 235
- function call, 145-146
- increment, 236-237
- member access, 235
- relational (comparison), 125-128
- stream, 129-131, 178-179
- subtraction, 304
- subtraction assignment, 304

# P

## Pair, 333-335

- pass-by-reference, 71-73, 132
- pass-by-value, 73, 131
- placement new, 185
- point of declaration, 59
- pointer
  - arithmetic, 85-97
  - basic syntax, 67-70
  - const, 105
  - implicit conversion, 244
  - referent, 69
  - reseating, 70
  - "this", 128-129, 133, 245
  - to pointer, 395, 399
  - void\*, 185
  - vs. reference, 100-101, 104
- pow, 60, 64
- predicate, 148, 150-151, 338, 377
- preprocessor, 4

PrintBtNode, 359-360  
 printContainer, 202-203  
 printContainerReverse, 266  
 printSequence, 139  
 private, 117  
 ptrdiff\_t, 270  
 public, 117, 145

# Q

## quickSort, 163-172

# R

r-value, 11  
 random access iterator, 495-496  
 recursive algorithm, 164  
 reference
 

- const, 105
- implicit conversion, 244, 292
- referent, 99
- syntax, 99-103
- vs. pointer, 100-101, 104

## referent

- of pointer, 68
- of reference, 99

## reflexive comparison, 153

## remainder, calculating, 6, 19

## ReverseIter

- base, 263
- constructors, 263-264, 267
- operators
  - bidirectional, 264
  - random access, 272-277

## Ring

- back, 281
- begin, 305
- capacity, 281
- clear, 298
- copy constructor, 307
- default constructor, 281
- \_destroyAllElements, 287
- destructor, 287
- \_element, 291-292
- empty, 281
- end, 305

**Ring** (*continued*)

- erase, 310-313
- front, 281
- head, 279-280
- insert, 307-310
- member types, 280, 305
- operator[], 292
- operator=, 307
- pop\_back, 295-296
- pop\_front, 296-298
- push\_back, 284-287
- push\_front, 289-291
- rbegin, 305
- \_reallocate, 281-284
- rend, 305
- reserve, 284
- size, 281
- tail, 279-280

**RingIter**

- constructors, 303
- member types, 303
- operators, 304

**runtime**, 67

## S

**scope**

- and destructor, 176, 183
- types of, 59-62

set builder notation, 76

sizeof, 185

sorting algorithm

*See* bubble sort; insertionSort; quickSort

**stack**, 183**static\_cast**, 185, 194**static variable**, 317**std**, 55**<string>**, 4**string**, 2-4**struct**, 145**Student**

- constructors, 118, 131
- member functions, 119-121
- operators, 125-133
- predicates, 150-151

**subroutine**, 1**swap**

- generic, 137-138

**swap** (*continued*)

- int\*, 71
- int&, 107
- Student&, 134

## T

**template**

*See also* class template, function template

- argument deduction, 138, 334-335
- default argument, 377
- instantiation, 138
- specialization, 269-270
- "this" pointer, 128-129, 133, 245
- time complexity, 495-496, 501-502
- Traceable, 173-181
- translation unit, 65
- traverseInOrder, 355-361
- typedef, 156-159
- typename, 140, 159

## U

**unnamed object**, 150, 174, 185, 497**unsigned int**, 156**using**

- declaration, 57
- directive, 56

## V

**variable**

- declaration of, 1-2, 4
- global, 59-60, 62
- scope of, 59-62
- static, 317

**Vector**

- back, 190
- begin, 190
- capacity, 189
- clear, 195
- copy constructor, 200
- default constructor, 189
- \_destroyAllElements, 195

Vector (*continued*)

    destructor, 196  
    empty, 190  
    end, 190  
    erase, 209-211  
    front, 190  
    insert, 207-209  
    member types, 189, 277  
    operator[], 190  
    operator=, 200-202  
    pop\_back, 199-200  
    push\_back, 190-191, 194  
    rbegin, 277  
    \_reallocate, 192-194  
    rend, 277  
    reserve, 199  
    size, 189

## void

    pointer, 185  
    return type, 71

W  
X  
Y  
Z