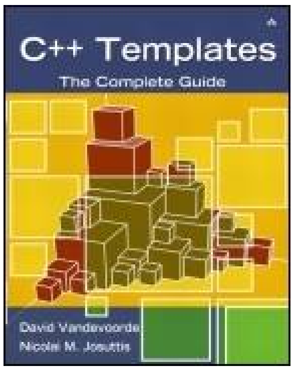


Table of Contents

C++ Templates: The Complete Guide

By David Vandevoorde, Nicolai M. Josuttis

Publisher : Addison Wesley
Pub Date : November 12, 2002
ISBN : 0-201-73484-2
Pages : 552

Templates are among the most powerful features of C++, but they are too often neglected, misunderstood, and misused. C++ Templates: The Complete Guide provides software architects and engineers with a clear understanding of why, when, and how to use templates to build and maintain cleaner, faster, and smarter software more efficiently.

C++ Templates begins with an insightful tutorial on basic concepts and language features. The remainder of the book serves as a comprehensive reference, focusing first on language details, then on a wide range of coding techniques, and finally on advanced applications for templates. Examples used throughout the book illustrate abstract concepts and demonstrate best practices.

Readers learn

- The exact behaviors of templates
- How to avoid the pitfalls associated with templates
- Idioms and techniques, from the basic to the previously undocumented
- How to reuse source code without threatening performance or safety
- How to increase the efficiency of C++ programs
-

How to produce more flexible and maintainable software

This practical guide shows programmers how to exploit the full power of the template features in C++.

NEXT

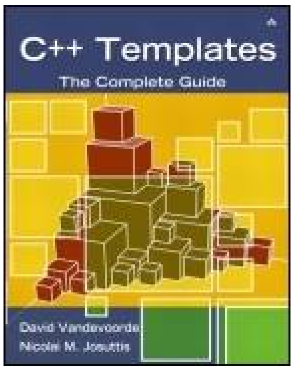


Table of Contents

C++ Templates: The Complete Guide

By David Vandevoorde, Nicolai M. Josuttis

Publisher : Addison Wesley
Pub Date : November 12, 2002
ISBN : 0-201-73484-2
Pages : 552

Copyright
Preface
Acknowledgments
    Nico's Acknowledgments
    David's Acknowledgments

Chapter 1. About This Book
    Section 1.1. What You Should Know Before Reading This Book
    Section 1.2. Overall Structure of the Book
    Section 1.3. How to Read This Book
    Section 1.4. Some Remarks About Programming Style
    Section 1.5. The Standard versus Reality
    Section 1.6. Example Code and Additional Informations
    Section 1.7. Feedback

Part I: The Basics
    Chapter 2. Function Templates
        Section 2.1. A First Look at Function Templates
        Section 2.2. Argument Deduction
        Section 2.3. Template Parameters
        Section 2.4. Overloading Function Templates
        Section 2.5. Summary

    Chapter 3. Class Templates
        Section 3.1. Implementation of Class Template Stack
        Section 3.2. Use of Class Template Stack
        Section 3.3. Specializations of Class Templates
        Section 3.4. Partial Specialization
        Section 3.5. Default Template Arguments
        Section 3.6. Summary

Chapter 4. Nontype Template Parameters
- Section 4.1. Nontype Class Template Parameters
- Section 4.2. Nontype Function Template Parameters
- Section 4.3. Restrictions for Nontype Template Parameters
- Section 4.4. Summary

Chapter 5. Tricky Basics
- Section 5.1. Keyword typename
- Section 5.2. Using this->
- Section 5.3. Member Templates
- Section 5.4. Template Template Parameters
- Section 5.5. Zero Initialization
- Section 5.6. Using String Literals as Arguments for Function Templates
- Section 5.7. Summary

Chapter 6. Using Templates in Practice
- Section 6.1. The Inclusion Model
- Section 6.2. Explicit Instantiation
- Section 6.3. The Separation Model
- Section 6.4. Templates and inline
- Section 6.5. Precompiled Headers
- Section 6.6. Debugging Templates
- Section 6.7. Afternotes
- Section 6.8. Summary

Chapter 7. Basic Template Terminology
- Section 7.1. "Class Template" or "Template Class"?
- Section 7.2. Instantiation and Specialization
- Section 7.3. Declarations versus Definitions
- Section 7.4. The One-Definition Rule
- Section 7.5. Template Arguments versus Template Parameters

Part II: Templates in Depth

Chapter 8. Fundamentals in Depth
- Section 8.1. Parameterized Declarations
- Section 8.2. Template Parameters
- Section 8.3. Template Arguments
- Section 8.4. Friends
- Section 8.5. Afternotes

Chapter 9. Names in Templates
- Section 9.1. Name Taxonomy
- Section 9.2. Looking Up Names
- Section 9.3. Parsing Templates
- Section 9.4. Derivation and Class Templates
- Section 9.5. Afternotes

Chapter 10. Instantiation
- Section 10.1. On-Demand Instantiation
- Section 10.2. Lazy Instantiation
- Section 10.3. The C++ Instantiation Model
- Section 10.4. Implementation Schemes
- Section 10.5. Explicit Instantiation
- Section 10.6. Afternotes

Chapter 11. Template Argument Deduction
Section 11.1. The Deduction Process
Section 11.2. Deduced Contexts
Section 11.3. Special Deduction Situations
Section 11.4. Allowable Argument Conversions
Section 11.5. Class Template Parameters
Section 11.6. Default Call Arguments
Section 11.7. The Barton-Nackman Trick
Section 11.8. Afternotes

Chapter 12. Specialization and Overloading
Section 12.1. When "Generic Code" Doesn't Quite Cut It
Section 12.2. Overloading Function Templates
Section 12.3. Explicit Specialization
Section 12.4. Partial Class Template Specialization
Section 12.5. Afternotes

Chapter 13. Future Directions
Section 13.1. The Angle Bracket Hack
Section 13.2. Relaxed typename Rules
Section 13.3. Default Function Template Arguments
Section 13.4. String Literal and Floating-Point Template Arguments
Section 13.5. Relaxed Matching of Template Template Parameters
Section 13.6. Typedef Templates
Section 13.7. Partial Specialization of Function Templates
Section 13.8. The typeof Operator
Section 13.9. Named Template Arguments
Section 13.10. Static Properties
Section 13.11. Custom Instantiation Diagnostics
Section 13.12. Overloaded Class Templates
Section 13.13. List Parameters
Section 13.14. Layout Control
Section 13.15. Initializer Deduction
Section 13.16. Function Expressions
Section 13.17. Afternotes

Part III: Templates and Design
Chapter 14. The Polymorphic Power of Templates
Section 14.1. Dynamic Polymorphism
Section 14.2. Static Polymorphism
Section 14.3. Dynamic versus Static Polymorphism
14.4 New Forms of Design Patterns
Section 14.5. Generic Programming
Section 14.6. Afternotes

Chapter 15. Traits and Policy Classes
Section 15.1. An Example: Accumulating a Sequence
Section 15.2. Type Functions
Section 15.3. Policy Traits
Section 15.4. Afternotes

Chapter 16. Templates and Inheritance
Section 16.1. Named Template Arguments

Section 16.2. The Empty Base Class Optimization (EBCO)
Section 16.3. The Curiously Recurring Template Pattern (CRTP)
Section 16.4. Parameterized Virtuality
Section 16.5. Afternotes

Chapter 17. Metaprograms
Section 17.1. A First Example of a Metaprogram
Section 17.2. Enumeration Values versus Static Constants
Section 17.3. A Second Example: Computing the Square Root
Section 17.4. Using Induction Variables
Section 17.5. Computational Completeness
Section 17.6. Recursive Instantiation versus Recursive Template Arguments
Section 17.7. Using Metaprograms to Unroll Loops
Section 17.8. Afternotes

Chapter 18. Expression Templates
Section 18.1. Temporaries and Split Loops
Section 18.2. Encoding Expressions in Template Arguments
Section 18.3. Performance and Limitations of Expression Templates
Section 18.4. Afternotes

Part IV: Advanced Applications
Chapter 19. Type Classification
Section 19.1. Determining Fundamental Types
Section 19.2. Determining Compound Types
Section 19.3. Identifying Function Types
Section 19.4. Enumeration Classification with Overload Resolution
Section 19.5. Determining Class Types
Section 19.6. Putting It All Together
Section 19.7. Afternotes

Chapter 20. Smart Pointers
Section 20.1. Holders and Trules
Section 20.2. Reference Counting
Section 20.3. Afternotes

Chapter 21. Tuples
Section 21.1. Duos
Section 21.2. Recursive Duos
Section 21.3. Tuple Construction
Section 21.4. Afternotes

Chapter 22. Function Objects and Callbacks
Section 22.1. Direct, Indirect, and Inline Calls
Section 22.2. Pointers and References to Functions
Section 22.3. Pointer-to-Member Functions
Section 22.4. Class Type Functors
Section 22.5. Specifying Functors
Section 22.6. Introspection
Section 22.7. Function Object Composition
Section 22.8. Value Binders
Functor Operations: A Complete Implementation
Section 22.10. Afternotes

Appendix A. The One-Definition Rule
  Section A.1. Translation Units
  Section A.2. Declarations and Definitions
  Section A.3. The One-Definition Rule in Detail

Appendix B. Overload Resolution
  Section B.1. When Does Overload Resolution Kick In?
  Section B.2. Simplified Overload Resolution
  Section B.3. Overloading Details

Bibliography
  Newsgroups
  Books and Web Sites

Glossary

Copyright

Library of Congress Cataloging-in-Publication Data

Vandevoorde, David.

C++ templates : the complete guide / David Vandevoorde, Nicolai M. Josuttis.

p. cm.

Includes bibliographical references and index.

Preface

The idea of templates in C++ is more than ten years old. C++ templates were already documented in 1990 in the "Annotated C++ Reference Manual" or so-called "ARM" (see [EllisStroustrupARM]) and they had been described before that in more specialized publications. However, well over a decade later we found a dearth of literature that concentrates on the fundamental concepts and advanced techniques of this fascinating, complex, and powerful C++ feature. We wanted to address this issue and decided to write the book about templates (with perhaps a slight lack of humility).

However, we approached the task with different backgrounds and with different intentions. David, an experienced compiler implementer and member of the C++ Standard Committee Core Language Working Group, was interested in an exact and detailed description of all the power (and problems) of templates. Nico, an "ordinary" application programmer and member of the C++ Standard Committee Library Working Group, was interested in understanding all the techniques of templates in a way that he could use and benefit from them. In addition, we both wanted to share this knowledge with you, the reader, and the whole community to help to avoid further misunderstanding, confusion, or apprehension.

As a consequence, you will see both conceptual introductions with day-to-day examples and detailed descriptions of the exact behavior of templates. Starting from the basic principles of templates and working up to the "art of template programming," you will discover (or rediscover) techniques such as static polymorphism, policy classes, metaprogramming, and expression templates. You will also gain a deeper understanding of the C++ standard library, in which almost all code involves templates.

We learned a lot and we had much fun while writing this book. We hope you will have the same experience while reading it. Enjoy!

Acknowledgments

This book presents ideas, concepts, solutions, and examples from many sources. We'd like to thank all the people and companies who helped and supported us during the past few years.

First, we'd like to thank all the reviewers and everyone else who gave us their opinion on early manuscripts. These people endow the book with a quality it would never have had without their input. The reviewers for this book were Kyle Blaney, Thomas Gschwind, Dennis Mancl, Patrick Mc Killen, and Jan Christiaan van Winkel. Special thanks to Dietmar Kühl, who meticulously reviewed and edited the whole book. His feedback was an incredible contribution to the quality of this book.

We'd also like to thank all the people and companies who gave us the opportunity to test our examples on different platforms with different compilers. Many thanks to the Edison Design Group for their great compiler and their support. It was a big help during the standardization process and the writing of this book. Many thanks also go to all the developers of the free GNU and egcs compilers (Jason Merrill was especially responsive), and to Microsoft for an evaluation version of Visual C++ (Jonathan Caves, Herb Sutter, and Jason Shirk were our contacts there).

Much of the existing "C++ wisdom" was collectively created by the online C++ community. Most of it comes from the moderated Usenet groups comp.lang.c++.moderated and comp.std.c++. We are therefore especially indebted to the active moderators of those groups, who keep the discussions useful and constructive. We also much appreciate all those who over the years have taken the time to describe and explain their ideas for us all to share.

The Addison-Wesley team did another great job. We are most indebted to Debbie Lafferty (our editor) for her gentle prodding, good advice, and relentless hard work in support of this book. Thanks also go to Tyrrell Albaugh, Bunny Ames, Melanie Buck, Jacquelyn Doucette, Chanda Leary-Coutu, Catherine Ohala, and Marty Rabinowitz. We're grateful as well to Marina Lang, who first sponsored this book within Addison-Wesley. Susan Winer contributed an early round of editing that helped shape our later work.

Nico's Acknowledgments

My first personal thanks go with a lot of kisses to my family: Ulli, Lucas, Anica, and Frederic supported this book with a lot of patience, consideration, and encouragement.

In addition, I want to thank David. His expertise turned out to be incredible, but his patience was even better (sometimes I ask really silly questions). It is a lot of fun to work with him.

David's Acknowledgments

My wife, Karina, has been instrumental in this book coming to a conclusion, and I am immensely grateful for the role that she plays in my life. Writing "in your spare time" quickly becomes erratic when many other activities vie for your schedule. Karina helped me to manage that schedule, taught me to say "no" in order to make the time needed to make regular progress in the writing process, and above all was amazingly supportive of this project. I thank God every day for her friendship and love.

I'm also tremendously grateful to have been able to work with Nico. Besides his directly visible contributions to the text, his experience and discipline moved us from my pitiful doodling to a well-organized production.

John "Mr. Template" Spicer and Steve "Mr. Overload" Adamczyk are wonderful friends and colleagues, but in my opinion they are (together) also the ultimate authority regarding the core C++ language. They clarified many of the trickier issues described in this book, and should you find an error in the description of a C++ language element, it is almost certainly attributable to my failing to consult with them.

Finally, I want to express my appreciation to those who were supportive of this project without necessarily contributing to it directly (the power of cheer cannot be understated). First, my parents: Their love for me and their encouragement made all the difference. And then there are the numerous friends inquiring: "How is the book going?" They, too, were a source of encouragement: Michael Beckmann, Brett and Julie Beene, Jarran Carr, Simon Chang, Ho and Sarah Cho, Christophe De Dinechin, Ewa Deelman, Neil Eberle, Sassan Hazeghi, Vikram Kumar, Jim and Lindsay Long, R.J. Morgan, Mike Puritano, Ragu Raghavendra, Jim and Phuong Sharp, Gregg Vaughn, and John Wiegley.

Chapter 1. About This Book

Although templates have been part of C++ for well over a decade (and available in various forms for almost as long), they still lead to misunderstanding, misuse, or controversy. At the same time, they are increasingly found to be powerful instruments for the development of cleaner, faster, and smarter software. Indeed, templates have become the cornerstone of several new C++ programming paradigms.

Yet we have found that most existing books and articles are at best superficial in their treatment of the theory and application of C++ templates. Even those few books that do an excellent job of surveying various template-based techniques fail to describe accurately how these techniques are supported by the language. As a result, beginning and advanced C++ programmers alike are finding themselves wrestling with templates, attempting to decide why their code is handled unexpectedly.

This observation was one of the main motivations for us to write this book. However, we both came up with the topic independently and had somewhat distinct approaches in mind:

- David's goal was to provide a complete reference to the details of the C++ template language mechanism and the major advanced programming techniques that templates enable. His focus was on precision and completeness.

- Nico's interest was to have a book that helps himself and others use templates in the day-to-day life of a programmer. This implies that the book should present the material in an intuitive manner, while dealing with the practical aspects of templates.

In a sense, you could see us as a scientist-engineer pair: We both deal with the same discipline, but our emphasis is somewhat different (with much overlap, of course).

Addison-Wesley brought us together and as a result you get what we think is a solid combination of a careful C++ template tutorial with a detailed reference. The tutorial aspect covers not only an introduction to the language elements, but also aims at developing a sense for design methods that lead to practical solutions. Similarly, the book is not only a reference for the details of C++ template syntax and semantics, but also a compendium of well-known and lesser known idioms and techniques.

1.1 What You Should Know Before Reading This Book

To get the most from this book you should already know C++: We describe the details of a particular language feature, not the fundamentals of the language itself. You should be familiar with the concepts of classes and inheritance, and you should be able to write C++ programs using components such as IOstreams and containers from the C++ standard library. In addition, we review more subtle issues as the need arises, even when such issues aren't directly related to templates. This ensures that the text is accessible to experts and intermediate programmers alike.

We deal mostly with the C++ language as standardized in 1998 (see [Standard98]), plus the clarifications provided by the C++ Standardization Committee in its first technical corrigendum (see [Standard02]). If you feel your understanding of the basics of C++ is rusty or out-of-date, we recommend [StroustrupC++PL], [JosuttisOOP], and [JosuttisStdLib] to refresh your knowledge. These books are excellent introductions to the modern language and its standard library. Additional publications are listed in Appendix B.3.5.

1.2 Overall Structure of the Book

Our goal is to provide the information necessary for starting to use templates and benefit from their power, as well as to provide information that will enable experienced programmers to push the limits of the state-of-the-art. To achieve this, we decided to organize our text in parts:

- Part I introduces the basic concepts underlying templates. It is written in a tutorial style.
- Part II presents the language details and is a handy reference to template-related constructs.
- Part III explains fundamental design techniques supported by C++ templates. They range from near-trivial ideas to sophisticated idioms that may not have been published elsewhere.
- Part IV builds on the previous two parts and adds a discussion of various popular applications for templates.

Each of these parts consists of several chapters. In addition, we provide a few appendixes that cover material not exclusively related to templates (for example, an overview of overload resolution in C++).

The chapters of Part I are meant to be read in sequence. For example, Chapter 3 builds on the material covered in Chapter 2. In the other parts, however, the connection between chapters is considerably looser. For example, it would be entirely natural to read the chapter about functors (Chapter 22) before the chapter about smart pointers (Chapter 20).

Last, we provide a rather complete index that encourages additional ways to read this book out of sequence.

1.3 How to Read This Book

If you are a C++ programmer who wants to learn or review the concepts of templates, carefully read Part I, The Basics. Even if you're quite familiar with templates already, it may help to skim through this part quickly to familiarize yourself with the style and terminology that we use. This part also covers some of the logistical aspects of organizing your source code when it contains templates.

Depending on your preferred learning method, you may decide to absorb the many details of templates in Part II, or instead you could read about practical coding techniques in Part III (and refer back to Part II for the more subtle language issues). The latter approach is probably particularly useful if you bought this book with concrete day-to-day challenges in mind. Part IV is somewhat similar to Part III, but the emphasis is on understanding how templates can contribute to specific applications rather than design techniques. It is therefore probably best to familiarize yourself with the topics of Part III before delving into Part IV.

The appendixes contain much useful information that is often referred to in the main text. We have also tried to make them interesting in their own right.

In our experience, the best way to learn something new is to look at examples. Therefore, you'll find a lot of examples throughout the book. Some are just a few lines of code illustrating an abstract concept, whereas others are complete programs that provide a concrete application of the material. The latter kind of examples will be introduced by a C++ comment describing the file containing the program code. You can find these files at the Web site of this book at http://www.josuttis.com/tmplbook/.

1.4 Some Remarks About Programming Style

C++ programmers use different programming styles, and so do we: The usual questions about where to put whitespace, delimiters (braces, parentheses), and so forth came up. We tried to be consistent in general, although we occasionally make concessions to the topic at hand. For example, in tutorial sections we may prefer generous use of whitespace and concrete names to help visualize code, whereas in more advanced discussions a more compact style could be more appropriate.

We do want to draw your attention to one slightly uncommon decision regarding the declaration of types, parameters, and variables. Clearly, several styles are possible:

```
void foo (const int &x);
void foo (const int& x);
void foo (int const &x);
void foo (int const& x);
```

Although it is a bit less common, we decided to use the order int const rather than const int for "constant integer." We have two reasons for this. First, it provides for an easier answer to the question, "What is constant?" It's always what is in front of the const qualifier. Indeed, although

```
const int N = 100;
```

is equivalent to

```
int const N = 100;
```

there is no equivalent form for

```
int* const bookmark;       // the pointer cannot change, but the
                           // value pointed to can change
```

that would place the const qualifier before the pointer operator *. In this example, it is the pointer itself that is constant, not the int to which it points.

Our second reason has to do with a syntactical substitution principle that is very common when dealing with templates. Consider the following two type definitions [1]:

[1] Note that in C++ a type definition defines a "type alias" rather than a new type. For example:
```
typedef int Length; // define Length as an alias for int
int i = 42;
Lengthl = 88;
i = l;              // OK
l = i;              // OK
```

```
typedef char* CHARS;
typedef CHARS const CPTR;  // constant pointer to chars
```

The meaning of the second declaration is preserved when we textually replace CHARS with what it stands for:

```
typedef char* const CPTR;  // constant pointer to chars
```

However, if we write const before the type it qualifies, this principle doesn't apply. Indeed, consider the alternative to our first two type definitions presented earlier:

1.5 The Standard versus Reality

The C++ standard has been available since late 1998. However, it was not until 2002 that a publically available compiler could make the claim to "conform fully to the standard." Thus, compilers still differ in their support of the language. Several will compile most of the code in this book, but a few fairly popular compilers may not be able to handle many of our examples. We often present alternative techniques that may help cobble together a full or partial solution for these substandard C++ implementations, but some techniques are currently beyond their reach. Still, we expect that this problem will largely be resolved as programmers everywhere demand standard support from their vendors.

Even so, the C++ programming language is likely to evolve as time passes. Already the experts of the C++ community (regardless of whether they participate in the C++ Standardization Committee) are discussing various ways to improve the language, and several candidate improvements affect templates. Chapter 13 presents some trends in this area.

1.6 Example Code and Additional Informations

You can access all example programs and find more information about this book from its Web site, which has the following URL:

http://www.josuttis.com/tmplbook

Also, you can find a lot of additional information about this topic at David Vandevoorde's Web site at http://www.vandevoorde.com/Templates and on the Web in general. See the Bibliography on page 499 for suggestions on where to start.

1.7 Feedback

We welcome your constructive input—both the negative and the positive. We worked very hard to bring you what we hope you'll find to be an excellent book. However, at some point we had to stop writing, reviewing, and tweaking so we could "release the product." You may therefore find errors, inconsistencies, and presentations that could be improved, or topics that are missing altogether. Your feedback gives us a chance to inform all readers through the book's Web site and to improve any subsequent editions.

The best way to reach us is by e-mail:

tmplbook@josuttis.com

Be sure to check the book's Web site for the currently known errata before submitting reports.

Many thanks.

Part I: The Basics

This part introduces the general concept and language features of C++ templates. It starts with a discussion of the general goals and concepts by showing examples of function templates and class templates. It continues with some additional fundamental template techniques such as nontype template parameters, the keyword typename, and member templates. It ends with some general hints regarding the use and application of templates in practice.

This introduction to templates is also partially used in Nicolai M. Josuttis's book Object-Oriented Programming in C++, published by John Wiley and Sons Ltd, ISBN 0-470-84399-3. This book teaches all language features of C++ and the C++ standard library and explains their practical usage in a step-by-step tutorial.

Why Templates?

C++ requires us to declare variables, functions, and most other kinds of entities using specific types. However, a lot of code looks the same for different types. Especially if you implement algorithms, such as quicksort, or if you implement the behavior of data structures, such as a linked list or a binary tree for different types, the code looks the same despite the type used.

If your programming language doesn't support a special language feature for this, you only have bad alternatives:

1. You can implement the same behavior again and again for each type that needs this behavior.

2. You can write general code for a common base type such as Object or void*.

3. You can use special preprocessors.

If you come from C, Java, or similar languages, you probably have done some or all of this before. However, each of these approaches has its drawbacks:

1. If you implement a behavior again and again, you reinvent the wheel. You make the same mistakes and you tend to avoid complicated but better algorithms because they lead to even more mistakes.

2. If you write general code for a common base class you lose the benefit of type checking. In addition, classes may be required to be derived from special base classes, which makes it more difficult to maintain your code.

3.

Chapter 2. Function Templates

This chapter introduces function templates. Function templates are functions that are parameterized so that they represent a family of functions.

2.1 A First Look at Function Templates

Function templates provide a functional behavior that can be called for different types. In other words, a function template represents a family of functions. The representation looks a lot like an ordinary function, except that some elements of the function are left undetermined: These elements are parameterized. To illustrate, let's look at a simple example.

2.1.1 Defining the Template

The following is a function template that returns the maximum of two values:

```
// basics/max.hpp

template <typename T>
inline T const& max (T const& a, T const& b)
{
    // if a < b then use b else use a
    return a<b?b:a;
}
```

This template definition specifies a family of functions that returns the maximum of two values, which are passed as function parameters a and b. The type of these parameters is left open as template parameter T. As seen in this example, template parameters must be announced with syntax of the following form:

```
template < comma-separated-list-of-parameters >
```

In our example, the list of parameters is typename T. Note how the less-than and the greater-than symbols are used as brackets; we refer to these as angle brackets. The keyword typename introduces a so-called type parameter. This is by far the most common kind of template parameter in C++ programs, but other parameters are possible, and we discuss them later (see Chapter 4).

Here, the type parameter is T. You can use any identifier as a parameter name, but using T is the convention. The type parameter represents an arbitrary type that is specified by the caller when the caller calls the function. You can use any type (fundamental type, class, and so on) as long as it provides the operations that the template uses. In this case, type T has to support operator < because a and b are compared using this operator.

For historical reasons, you can also use class instead of typename to define a type parameter. The keyword typename came relatively late in the evolution of the C++ language. Prior to that, the keyword class was the only way to introduce a type parameter, and this remains a valid way to do so. Hence, the template max() could be defined equivalently as follows:

```
template <class T>
inline T const& max (T const& a, T const& b)
{
    // if a < b then use b else use a
    return a<b?b:a;
}
```

Semantically there is no difference in this context. So, even if you use class here, any type may be used for template arguments. However, because this use of class can be misleading (not only class types can be substituted for T), you should prefer the use of typename in this context. Note also that unlike class type declarations, the keyword struct cannot be used in place of typename when declaring type parameters.

2.2 Argument Deduction

When we call a function template such as max() for some arguments, the template parameters are determined by the arguments we pass. If we pass two ints to the parameter types T const&, the C++ compiler must conclude that T must be int. Note that no automatic type conversion is allowed here. Each T must match exactly. For example:

```
template <typename T>
inline T const& max (T const& a, T const& b);

max(4,7)      // OK: T is int for both arguments
max(4,4.2)    // ERROR: first T is int, second T is double
```

There are three ways to handle such an error:

1. Cast the arguments so that they both match:
```
max(static_cast<double>(4),4.2)    // OK
```
2. Specify (or qualify) explicitly the type of T:
```
max<double>(4,4.2)                 // OK
```
3. Specify that the parameters may have different types.

For a detailed discussion of these topics, see the next section.

2.3 Template Parameters

Function templates have two kinds of parameters:

1. Template parameters, which are declared in angle brackets before the function template name:

```
template <typename T>            // T is template parameter
```

2. Call parameters, which are declared in parentheses after the function template name:

```
    max (T const& a, T const& b)   // a and b are call parameters
```

You may have as many template parameters as you like. However, in function templates (unlike class templates) no default template arguments can be specified. [3] For example, you could define the max() template for call parameters of two different types:

[3] This restriction is mainly the result of a historical glitch in the development of function templates. There are probably no technical hindrances to implementing such a feature in modern C++ compilers, and in the future it will probably be available (see Section 13.3 on page 207).

```
template <typename T1, typename T2>
inline T1 max (T1 const& a, T2 const& b)
{
    return a < b ? b : a;
}

max(4,4.2)   // OK, but type of first argument defines return type
```

This may appear to be a good method to enable passing two call parameters of different types to the max() template, but in this example it has drawbacks. The problem is that the return type must be declared. If you use one of the parameter types, the argument for the other parameter might get converted to this type, regardless of the caller's intention. C++ does not provide a means to specify choosing "the more powerful type" (however, you can provide this feature by some tricky template programming, see Section 15.2.4 on page 271). Thus, depending on the call argument order the maximum of 42 and 66.66 might be the double 66.66 or the int 66. Another drawback is that converting the type of the second parameter into the return type creates a new, local temporary object. As a consequence, you cannot return the result by reference. [4] In our example, therefore, the return type has to be T1 instead of T1 const&.

[4] You are not allowed to return values by reference if they are local to a function because you'd return something that doesn't exist when the program leaves the scope of this function.

Because the types of the call parameters are constructed from the template parameters, template and call parameters are usually related. We call this concept function template argument deduction. It allows you to call a function template as you would an ordinary function.

However, as mentioned earlier, you can instantiate a template explicitly for certain types:

```
template <typename T>
inline T const& max (T const& a, T const& b);

max<double>(4,4.2)    // instantiate T as double
```

2.4 Overloading Function Templates

Like ordinary functions, function templates can be overloaded. That is, you can have different function definitions with the same function name so that when that name is used in a function call, a C++ compiler must decide which one of the various candidates to call. The rules for this decision may become rather complicated, even without templates. In this section we discuss overloading when templates are involved. If you are not familiar with the basic rules of overloading without templates, please look at Appendix B, where we provide a reasonably detailed survey of the overload resolution rules.

The following short program illustrates overloading a function template:

```
// basics/max2.cpp

// maximum of two int values
inline int const& max (int const& a, int const& b)
{
    return a<b?b:a;
}

// maximum of two values of any type
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a<b?b:a;
}

// maximum of three values of any type
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return max (max(a,b), c);
}

int main()
{
    ::max(7, 42, 68);     // calls the template for three arguments
    ::max(7.0, 42.0);     // calls max<double> (by argument deduction)
    ::max('a', 'b');      // calls max<char> (by argument deduction)
    ::max(7, 42);         // calls the nontemplate for two ints
    ::max<>(7, 42);       // calls max<int> (by argument deduction)
    ::max<double>(7, 42); // calls max<double> (no argument deduction)
    ::max('a', 42.7);     // calls the nontemplate for two ints
}
```

As this example shows, a nontemplate function can coexist with a function template that has the same name and can be instantiated with the same type. All other factors being equal, the overload resolution process normally prefers this nontemplate over one generated from the template. The fourth call falls under this rule:

```
max(7, 42)      // both int values match the nontemplate function perfectly
```

If the template can generate a function with a better match, however, then the template is selected. This is demonstrated by the second and third call of max():

```
max(7.0, 42.0)  // calls the max<double> (by argument deduction)
max('a', 'b');  // calls the max<char> (by argument deduction)
```

It is also possible to specify explicitly an empty template argument list. This syntax indicates that only templates may resolve a call, but all the template parameters should be deduced from the call arguments:

PREVIOUS NEXT

2.5 Summary

- Template functions define a family of functions for different template arguments.
- When you pass template arguments, function templates are instantiated for these argument types.
- You can explicitly qualify the template parameters.
- You can overload function templates.
- When you overload function templates, limit your changes to specifying template parameters explicitly.
- Make sure you see all overloaded versions of function templates before you call them.

Chapter 3. Class Templates

Similar to functions, classes can also be parameterized with one or more types. Container classes, which are used to manage elements of a certain type, are a typical example of this feature. By using class templates, you can implement such container classes while the element type is still open. In this chapter we use a stack as an example of a class template.

3.1 Implementation of Class Template Stack

As we did with function templates, we declare and define class Stack<> in a header file as follows (we discuss the separation of declaration and definition in different files in Section 7.3 on page 89):

```
// basics/stack1.hpp

#include <vector>
#include <stdexcept>

template <typename T>
class Stack {
  private:
    std::vector<T> elems;     // elements

  public:
    void push(T const&);      // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const {      // return whether the stack is empty
        return elems.empty();
    }
};
template <typename T>
void Stack<T>::push (T const& elem)
{
    elems.push_back(elem);    // append copy of passed elem
}

template<typename T>
void Stack<T>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    elems.pop_back();         // remove last element
}

template <typename T>
T Stack<T>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): empty stack");
    }
    return elems.back();      // return copy of last element
}
```

As you can see, the class template is implemented by using a class template of the C++ standard library: vector<>. As a result, we don't have to implement memory management, copy constructor, and assignment operator, so we can concentrate on the interface of this class template.

3.1.1 Declaration of Class Templates

Declaring class templates is similar to declaring function templates: Before the declaration, a statement declares an identifier as a type parameter. Again, T is usually used as an identifier:

```
template <typename T>
class Stack {

};
```

3.2 Use of Class Template Stack

To use an object of a class template, you must specify the template arguments explicitly. The following example shows how to use the class template Stack<>:

```
// basics/stack1test.cpp

#include <iostream>
#include <string>
#include <cstdlib>
#include "stack1.hpp"

int main()
{
    try {
        Stack<int>         intStack;       // stack of ints
        Stack<std::string> stringStack;    // stack of strings

        // manipulate int stack
        intStack.push(7);
        std::cout << intStack.top() << std::endl;

        // manipulate string stack
        stringStack.push("hello");
        std::cout << stringStack.top() << std::endl;
        stringStack.pop();
        stringStack.pop();
    }
    catch (std::exception const& ex) {
        std::cerr << "Exception: " << ex.what() << std::endl;
        return EXIT_FAILURE;  // exit program with ERROR status
    }
}
```

By declaring type Stack<int>, int is used as type T inside the class template. Thus, intStack is created as an object that uses a vector of ints as elements and, for all member functions that are called, code for this type is instantiated. Similarly, by declaring and using Stack<std::string>, an object that uses a vector of strings as elements is created, and for all member functions that are called, code for this type is instantiated.

Note that code is instantiated only for member functions that are called. For class templates, member functions are instantiated only when they are used. This, of course, saves time and space. It has the additional benefit that you can instantiate a class even for those types that cannot perform all the operations of all the member functions, as long as these member functions are not called. As an example, consider a class in which some member functions use the operator < to sort elements. If you refrain from calling these member functions, you can instantiate the class template for types for which operator < is not defined.

In this example, the default constructor, push(), and top() are instantiated for both int and strings. However, pop() is instantiated only for strings. If a class template has static members, these are instantiated once for each type.

You can use a type of an instantiated class template as any other type, as long as the operations are supported:

```
void foo (Stack<int> const& s)     // parameter s is int stack
{
    Stack<int> istack[10];         // istack is array of 10 int stacks

}
```

3.3 Specializations of Class Templates

You can specialize a class template for certain template arguments. Similar to the overloading of function templates (see page 15), specializing class templates allows you to optimize implementations for certain types or to fix a misbehavior of certain types for an instantiation of the class template. However, if you specialize a class template, you must also specialize all member functions. Although it is possible to specialize a single member function, once you have done so, you can no longer specialize the whole class.

To specialize a class template, you have to declare the class with a leading template<> and a specification of the types for which the class template is specialized. The types are used as a template argument and must be specified directly following the name of the class:

```
template<>
class Stack<std::string> {

};
```

For these specializations, any definition of a member function must be defined as an "ordinary" member function, with each occurrence of T being replaced by the specialized type:

```
void Stack<std::string>::push (std::string const& elem)
{
    elems.push_back(elem);    // append copy of passed elem
}
```

Here is a complete example of a specialization of Stack<> for type std::string:

```
// basics/stack2.hpp

#include <deque>
#include <string>
#include <stdexcept>
#include "stack1.hpp"

template<>
class Stack<std::string> {
  private:
    std::deque<std::string> elems;  // elements

  public:
    void push(std::string const&);  // push element
    void pop();                     // pop element
    std::string top() const;        // return top element
    bool empty() const {            // return whether the stack is empty
        return elems.empty();
    }
};

void Stack<std::string>::push (std::string const& elem)
{
    elems.push_back(elem);    // append copy of passed elem
}

void Stack<std::string>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range
                ("Stack<std::string>::pop(): empty stack");
    }
    elems.pop_back();         // remove last element
```

3.4 Partial Specialization

Class templates can be partially specialized. You can specify special implementations for particular circumstances, but some template parameters must still be defined by the user. For example, for the following class template

```
template <typename T1, typename T2>
class MyClass {

};
```

the following partial specializations are possible:

```
// partial specialization: both template parameters have same type
template <typename T>
class MyClass<T,T> {

};

// partial specialization: second type is int
template <typename T>
class MyClass<T,int> {

};

// partial specialization: both template parameters are pointer types
template <typename T1, typename T2>
class MyClass<T1*,T2*> {

};
```

The following example shows which template is used by which declaration:

```
MyClass<int,float> mif;    // uses MyClass<T1,T2>
MyClass<float,float> mff;  // uses MyClass<T,T>
MyClass<float,int> mfi;    // uses MyClass<T,int>
MyClass<int*,float*> mp;   // uses MyClass<T1*,T2*>
```

If more than one partial specialization matches equally well, the declaration is ambiguous:

```
MyClass<int,int> m;        // ERROR: matches MyClass<T,T>
                           //        and MyClass<T,int>
MyClass<int*,int*> m;      // ERROR: matches MyClass<T,T>
                           //        and MyClass<T1*,T2*>
```

To resolve the second ambiguity, you can provide an additional partial specialization for pointers of the same type:

```
template <typename T>
class MyClass<T*,T*> {

};
```

For details, see Section 12.4 on page 200.

3.5 Default Template Arguments

For class templates you can also define default values for template parameters. These values are called default template arguments. They may even refer to previous template parameters. For example, in class Stack<> you can define the container that is used to manage the elements as a second template parameter, using std::vector<> as the default value:

```
// basics/stack3.hpp

#include <vector>
#include <stdexcept>

template <typename T, typename CONT = std::vector<T> >
class Stack {
  private:
    CONT elems;               // elements

  public:
    void push(T const&);      // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const {      // return whether the stack is empty
        return elems.empty();
    }
};

template <typename T, typename CONT>
void Stack<T,CONT>::push (T const& elem)
{
    elems.push_back(elem);    // append copy of passed elem
}

template <typename T, typename CONT>
void Stack<T,CONT>::pop ()
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    elems.pop_back();         // remove last element
}

template <typename T, typename CONT>
T Stack<T,CONT>::top () const
{
    if (elems.empty()) {
        throw std::out_of_range("Stack<>::top(): empty stack");
    }
    return elems.back();      // return copy of last element
}
```

Note that we now have two template parameters, so each definition of a member function must be defined with these two parameters:

```
template <typename T, typename CONT>
void Stack<T,CONT>::push (T const& elem)
{
    elems.push_back(elem);    // append copy of passed elem
}
```

You can use this stack the same way it was used before. Thus, if you pass a first and only argument as an element type, a vector is used to manage the elements of this type:

3.6 Summary

- A class template is a class that is implemented with one or more type parameters left open.
- To use a class template, you pass the open types as template arguments. The class template is then instantiated (and compiled) for these types.
- For class templates, only those member functions that are called are instantiated.
- You can specialize class templates for certain types.
- You can partially specialize class templates for certain types.
- You can define default values for class template parameters. These may refer to previous template parameters.

Chapter 4. Nontype Template Parameters

For function and class templates, template parameters don't have to be types. They can also be ordinary values. As with templates using type parameters, you define code for which a certain detail remains open until the code is used. However, the detail that is open is a value instead of a type. When using such a template, you have to specify this value explicitly. The resulting code then gets instantiated. This chapter illustrates this feature for a new version of the stack class template. In addition, we show an example of nontype function template parameters and discuss some restrictions to this technique.

4.1 Nontype Class Template Parameters

In contrast to the sample implementations of a stack in previous chapters, you can also implement a stack by using a fixed-size array for the elements. An advantage of this method is that the memory management overhead, whether performed by you or by a standard container, is avoided. However, determining the best size for such a stack can be challenging. The smaller the size you specify, the more likely it is that the stack will get full. The larger the size you specify, the more likely it is that memory will be reserved unnecessarily. A good solution is to let the user of the stack specify the size of the array as the maximum size needed for stack elements.

To do this, define the size as a template parameter:

```
// basics/stack4.hpp

#include <stdexcept>

template <typename T, int MAXSIZE>
class Stack {
  private:
    T elems[MAXSIZE];         // elements
    int numElems;             // current number of elements
  public:
    Stack();                  // constructor
    void push(T const&);      // push element
    void pop();               // pop element
    T top() const;            // return top element
    bool empty() const {      // return whether the stack is empty
        return numElems == 0;
    }
    bool full() const {       // return whether the stack is full
        return numElems == MAXSIZE;
    }
};

// constructor
template <typename T, int MAXSIZE>
Stack<T,MAXSIZE>::Stack ()
  : numElems(0)               // start with no elements
{
    // nothing else to do
}

template <typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::push (T const& elem)
{
    if (numElems == MAXSIZE) {
        throw std::out_of_range("Stack<>::push(): stack is full");
    }
    elems[numElems] = elem;  // append element
    ++numElems;               // increment number of elements
}

template<typename T, int MAXSIZE>
void Stack<T,MAXSIZE>::pop ()
{
    if (numElems <= 0) {
        throw std::out_of_range("Stack<>::pop(): empty stack");
    }
    --numElems;               // decrement number of elements
}

template <typename T, int MAXSIZE>
```

4.2 Nontype Function Template Parameters

You can also define nontype parameters for function templates. For example, the following function template defines a group of functions for which a certain value can be added:

```
// basics/addval.hpp

template <typename T, int VAL>
T addValue (T const& x)
{
    return x + VAL;
}
```

These kinds of functions are useful if functions or operations in general are used as parameters. For example, if you use the Standard Template Library (STL) you can pass an instantiation of this function template to add a value to each element of a collection:

```
std::transform (source.begin(), source.end(),  // start and end of source
                dest.begin(),                  // start of destination
                addValue<int,5>);              // operation
```

The last argument instantiates the function template addValue() to add 5 to an int value. The resulting function is called for each element in the source collection source, while it is translated into the destination collection dest.

Note that there is a problem with this example: addValue<int,5> is a function template, and function templates are considered to name a set of overloaded functions (even if the set has only one member). However, according to the current standard, sets of overloaded functions cannot be used for template parameter deduction. Thus, you have to cast to the exact type of the function template argument:

```
std::transform (source.begin(), source.end(),  // start and end of source
                dest.begin(),                  // start of destination
                (int(*)(int const&)) addValue<int,5>);  // operation
```

There is a proposal for the standard to fix this behavior so that the cast isn't necessary in this context (see [CoreIssue115] for details), but until then the cast may be necessary to be portable.

4.3 Restrictions for Nontype Template Parameters

Note that nontype template parameters carry some restrictions. In general, they may be constant integral values (including enumerations) or pointers to objects with external linkage.

Floating-point numbers and class-type objects are not allowed as nontype template parameters:

```
template <double VAT>        // ERROR: floating-point values are not
double process (double v)    //        allowed as template parameters
{
    return v * VAT;
}

template <std::string name>  // ERROR: class-type objects are not
class MyClass {              //        allowed as template parameters

};
```

Not being able to use floating-point literals (and simple constant floating-point expressions) as template arguments has historical reasons. Because there are no serious technical challenges, this may be supported in future versions of C++ (see Section 13.4 on page 210).

Because string literals are objects with internal linkage (two string literals with the same value but in different modules are different objects), you can't use them as template arguments either:

```
template <char const* name>
class MyClass {

};

MyClass<"hello"> x;   // ERROR: string literal "hello" not allowed
```

You cannot use a global pointer either:

```
template <char const* name>
class MyClass {

};

char const* s = "hello";

MyClass<s> x;         // ERROR: s is pointer to object with internal linkage
```

However, the following is possible:

```
template <char const* name>
class MyClass {

};

extern char const s[] = "hello";

MyClass<s> x;        // OK
```

The global character array s is initialized by "hello" so that s is an object with external linkage.

See Section 8.3.3 on page 109 for a detailed discussion and Section 13.4 on page 209 for a discussion of possible

4.4 Summary

- Templates can have template parameters that are values rather than types.
- You cannot use floating-point numbers, class-type objects, and objects with internal linkage (such as string literals) as arguments for nontype template parameters.

Chapter 5. Tricky Basics

This chapter covers some further basic aspects of templates that are relevant to the practical use of templates: an additional use of the typename keyword, defining member functions and nested classes as templates, template template parameters, zero initialization, and some details about using string literals as arguments for function templates. These aspects can be tricky at times, but every day-to-day programmer should have heard of them.

5.1 Keyword typename

The keyword typename was introduced during the standardization of C++ to clarify that an identifier inside a template is a type. Consider the following example:

```
template <typename T>
class MyClass {
   typename T::SubType * ptr;

};
```

Here, the second typename is used to clarify that SubType is a type defined within class T. Thus, ptr is a pointer to the type T::SubType.

Without typename, SubType would be considered a static member. Thus, it would be a concrete variable or object. As a result, the expression

```
T::SubType * ptr
```

would be a multiplication of the static SubType member of class T with ptr.

In general, typename has to be used whenever a name that depends on a template parameter is a type. This is discussed in detail in Section 9.3.2 on page 130.

A typical application of typename is the access to iterators of STL containers in template code:

```
// basics/printcoll.hpp

#include <iostream>

// print elements of an STL container
template <typename T>
void printcoll (T const& coll)
{
    typename T::const_iterator pos;  // iterator to iterate over coll
    typename T::const_iterator end(coll.end());  // end position

    for (pos=coll.begin(); pos!=end; ++pos) {
        std::cout << *pos << ' ';
    }
    std::cout << std::endl;
}
```

In this function template, the call parameter is an STL container of type T. To iterate over all elements of the container, the iterator type of the container is used, which is declared as type const_iterator inside each STL container class:

```
class stlcontainer {

  typedef     iterator;        // iterator for read/write access
  typedef     const_iterator;  // iterator for read access

};
```

Thus, to access type const_iterator of template type T, you have to qualify it with a leading typename:

5.2 Using this->

For class templates with base classes, using a name x by itself is not always equivalent to this->x, even though a member x is inherited. For example:

```
template <typename T>
class Base {
  public:
    void exit();
};

template <typename T>
class Derived : Base<T> {
  public:
    void foo() {
        exit();   // calls external exit() or error
    }
};
```

In this example, for resolving the symbol exit inside foo(), exit() defined in Base is never considered. Therefore, either you have an error, or another exit() (such as the standard exit()) is called.

We discuss this issue in Section 9.4.2 on page 136 in detail. For the moment, as a rule of thumb, we recommend that you always qualify any symbol that is declared in a base that is somehow dependent on a template parameter with this-> or Base<T>::. If you want to avoid all uncertainty, you may consider qualifying all member accesses (in templates).

5.3 Member Templates

Class members can also be templates. This is possible for both nested classes and member functions. The application and advantage of this ability can again be demonstrated with the Stack<> class template. Normally you can assign stacks to each other only when they have the same type, which implies that the elements have the same type. However, you can't assign a stack with elements of any other type, even if there is an implicit type conversion for the element types defined:

```
Stack<int> intStack1, intStack2;   // stacks for ints
Stack<float> floatStack;           // stack for floats

intStack1 = intStack2;   // OK: stacks have same type
floatStack = intStack1;  // ERROR: stacks have different types
```

The default assignment operator requires that both sides of the assignment operator have the same type, which is not the case if stacks have different element types.

By defining an assignment operator as a template, however, you can enable the assignment of stacks with elements for which an appropriate type conversion is defined. To do this you have to declare Stack<> as follows:

```
// basics/stack5decl.hpp

template <typename T>
class Stack {
  private:
    std::deque<T> elems;   // elements

  public:
    void push(T const&);   // push element
    void pop();            // pop element
    T top() const;         // return top element
    bool empty() const {   // return whether the stack is empty
        return elems.empty();
    }

    // assign stack of elements of type T2
    template <typename T2>
    Stack<T>& operator= (Stack<T2> const&);
};
```

The following two changes have been made:

1. We added a declaration of an assignment operator for stacks of elements of another type T2.

2. The stack now uses a deque as an internal container for the elements. Again, this is a consequence of the implementation of the new assignment operator.

The implementation of the new assignment operator looks like this:

```
// basics/stack5assign.hpp

template <typename T>
 template <typename T2>
```

5.4 Template Template Parameters

It can be useful to allow a template parameter itself to be a class template. Again, our stack class template can be used as an example.

To use a different internal container for stacks, the application programmer has to specify the element type twice. Thus, to specify the type of the internal container, you have to pass the type of the container and the type of its elements again:

```
Stack<int,std::vector<int> > vStack;  // integer stack that uses a vector
```

Using template template parameters allows you to declare the Stack class template by specifying the type of the container without respecifying the type of its elements:

```
stack<int,std::vector> vStack;        // integer stack that uses a vector
```

To do this you must specify the second template parameter as a template template parameter. In principle, this looks as follows [2]:

[2] There is a problem with this version that we explain in a minute. However, this problem affects only the default value std::deque. Thus, we can illustrate the general features of template template parameters with this example.

```
// basics/stack7decl.hpp

template <typename T,
          template <typename ELEM> class CONT = std::deque >
class Stack {
  private:
    CONT<T> elems;         // elements

  public:
    void push(T const&);   // push element
    void pop();            // pop element
    T top() const;         // return top element
    bool empty() const {   // return whether the stack is empty
        return elems.empty();
    }
};
```

The difference is that the second template parameter is declared as being a class template:

```
template <typename ELEM> class CONT
```

The default value has changed from std::deque<T> to std::deque. This parameter has to be a class template, which is instantiated for the type that is passed as the first template parameter:

```
CONT<T> elems;
```

This use of the first template parameter for the instantiation of the second template parameter is particular to this example. In general, you can instantiate a template template parameter with any type inside a class template.

As usual, instead of typename you could use the keyword class for template parameters. However, CONT is used to define a class and must be declared by using the keyword class. Thus, the following is fine:

```
template <typename T,
```

5.5 Zero Initialization

For fundamental types such as int, double, or pointer types, there is no default constructor that initializes them with a useful default value. Instead, any noninitialized local variable has an undefined value:

```
void foo()
{
    int x;     // x has undefined value
    int* ptr;  // ptr points to somewhere (instead of nowhere)
}
```

Now if you write templates and want to have variables of a template type initialized by a default value, you have the problem that a simple definition doesn't do this for built-in types:

```
template <typename T>
void foo()
{
    T x;      // x has undefined value if T is built-in type
}
```

For this reason, it is possible to call explicitly a default constructor for built-in types that initializes them with zero (or false for bool). That is, int() yields zero. As a consequence you can ensure proper default initialization even for built-in types by writing the following:

```
template <typename T>
void foo()
{
    T x = T();    // x is zero (or false)ifT is a built-in type
}
```

To make sure that a member of a class template, for which the type is parameterized, gets initialized, you have to define a default constructor that uses an initializer list to initialize the member:

```
template <typename T>
class MyClass {
  private:
    T x;
  public:
    MyClass() : x() {  // ensures that x is initialized even for built-in types
    }

};
```

5.6 Using String Literals as Arguments for Function Templates

Passing string literal arguments for reference parameters of function templates sometimes fails in a surprising way. Consider the following example:

```
// basics/max5.cpp

#include <string>

// note: reference parameters
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

int main()
{
    std::string s;

    ::max("apple","peach");   // OK: same type
    ::max("apple","tomato");  // ERROR: different types
    ::max("apple",s);         // ERROR: different types
}
```

The problem is that string literals have different array types depending on their lengths. That is, "apple" and "peach" have type char const[6] whereas "tomato" has type char const[7]. Only the first call is possible because the template expects both parameters to have the same type. However, if you declare nonreference parameters, you can substitute them with string literals of different size:

```
// basics/max6.cpp

#include <string>

// note: nonreference parameters
template <typename T>
inline T max (T a, T b)
{
    return a < b ? b : a;
}
int main()
{
    std::string s;

    ::max("apple","peach");   // OK: same type
    ::max("apple","tomato");  // OK: decays to same type
    ::max("apple",s);         // ERROR: different types
}
```

The explanation for this behavior is that during argument deduction array-to-pointer conversion (often called decay) occurs only if the parameter does not have a reference type. This is demonstrated by the following program:

```
// basics/refnonref.cpp

#include <typeinfo>
#include <iostream>

template <typename T>
void ref (T const& x)
{
    std::cout << "x in ref(T const&): "
              << typeid(x).name() << '\n';
```

5.7 Summary

- To access a type name that depends on a template parameter, you have to qualify the name with a leading typename.

- Nested classes and member functions can also be templates. One application is the ability to implement generic operations with internal type conversions. However, type checking still occurs.

- Template versions of assignment operators don't replace default assignment operators.

- You can also use class templates as template parameters, as so-called template template parameters.

- Template template arguments must match exactly. Default template arguments of template template arguments are ignored.

- By explicitly calling a default constructor, you can make sure that variables and members of templates are initialized by a default value even if they are instantiated with a built-in type.

- For string literals there is an array-to-pointer conversion during argument deduction if and only if the parameter is not a reference.

Chapter 6. Using Templates in Practice

Template code is a little different from ordinary code. In some ways templates lie somewhere between macros and ordinary (nontemplate) declarations. Although this may be an oversimplification, it has consequences not only for the way we write algorithms and data structures using templates, but also for the day-to-day logistics of expressing and analyzing programs involving templates.

In this chapter we address some of these practicalities without necessarily delving into the technical details that underlie them. Many of these details are explored in Chapter 10. To keep the discussion simple, we assume that our C++ compilation systems consist of fairly traditional compilers and linkers (C++ systems that don't fall in this category are quite rare).

6.1 The Inclusion Model

There are several ways to organize template source code. This section presents the most popular approach as of the time of this writing: the inclusion model.

6.1.1 Linker Errors

Most C and C++ programmers organize their nontemplate code largely as follows:

- Classes and other types are entirely placed in header files. Typically, this is a file with a .hpp (or .H, .h, .hh, .hxx) filename extension.
- For global variables and (noninline) functions, only a declaration is put in a header file, and the definition goes into a so-called dot-C file. Typically, this is a file with a .cpp (or .C, .c, .cc, or .hxx) filename extension.

This works well: It makes the needed type definition easily available throughout the program and avoids duplicate definition errors on variables and functions from the linker.

With these conventions in mind, a common error about which beginning template programmers complain is illustrated by the following (erroneous) little program. As usual for "ordinary code," we declare the template in a header file:

```
// basics/myfirst.hpp

#ifndef MYFIRST_HPP
#define MYFIRST_HPP

// declaration of template
template <typename T>
void print_typeof (T const&);

#endif // MYFIRST_HPP
```

print_typeof() is the declaration of a simple auxiliary function that prints some type information. The implementation of the function is placed in a dot-C file:

```
// basics/myfirst.cpp

#include <iostream>
#include <typeinfo>
#include "myfirst.hpp"

// implementation/definition of template
template <typename T>
void print_typeof (T const& x)
{
    std::cout << typeid(x).name() << std::endl;
}
```

The example uses the typeid operator to print a string that describes the type of the expression passed to it (see Section 5.6 on page 58).

PREVIOUS NEXT

6.2 Explicit Instantiation

The inclusion model ensures that all the needed templates are instantiated. This happens because the C++ compilation system automatically generates those instantiations as they are needed. The C++ standard also offers a construct to instantiate templates manually: the explicit instantiation directive.

6.2.1 Example of Explicit Instantiation

To illustrate manual instantiation, let's revisit our original example that leads to a linker error (see page 62). To avoid this error we add the following file to our program:

```
// basics/myfirstinst.cpp

#include "myfirst.cpp"

// explicitly instantiate print_typeof() for type double
template void print_typeof<double>(double const&);
```

The explicit instantiation directive consists of the keyword template followed by the fully substituted declaration of the entity we want to instantiate. In our example, we do this with an ordinary function, but it could be a member function or a static data member. For example:

```
// explicitly instantiate a constructor of MyClass<> for int
template MyClass<int>::MyClass();

// explicitly instantiate a function template max() for int
template int const& max (int const&, int const&);
```

You can also explicitly instantiate a class template, which is short for requesting the instantiation of all its instantiatable members. This excludes members that were previously specialized as well as those that were already instantiated:

```
// explicitly instantiate class Stack<> for int:
template class Stack<int>;

// explicitly instantiate some member functions of Stack<> for strings:
template Stack<std::string>::Stack();
template void Stack<std::string>::push(std::string const&);
template std::string Stack<std::string>::top();

// ERROR: can't explicitly instantiate a member function of a
//        class that was itself explicitly instantiated:
template Stack<int>::Stack();
```

There should be, at most, one explicit instantiation of each distinct entity in a program. In other words, you could explicitly instantiate both print_typeof<int> and print_typeof<double>, but each directive should appear only once in a program. Not following this rule usually results in linker errors that report duplicate definitions of the instantiated entities.

Manual instantiation has a clear disadvantage: We must carefully keep track of which entities to instantiate. For large projects this quickly becomes an excessive burden; hence we do not recommend it. We have worked on several projects that initially underestimated this burden, and we came to regret our decision as the code matured.

However, explicit instantiation also has a few advantages because the instantiation can be tuned to the needs of the program. Clearly, the overhead of large headers is avoided. The source code of template definition can be kept hidden, but then no additional instantiations can be created by a client program. Finally, for some applications it can

6.3 The Separation Model

Both approaches advocated in the previous sections work well and conform entirely to the C++ standard. However, this same standard also provides the alternative mechanism of exporting templates. This approach is sometimes called the C++ template separation model.

6.3.1 The Keyword export

In principle, it is quite simple to make use of the export facility: Define the template in just one file, and mark that definition and all its nondefining declarations with the keyword export. For the example in the previous section, this results in the following function template declaration:

```
// basics/myfirst3.hpp

#ifndef MYFIRST_HPP
#define MYFIRST_HPP

// declaration of template
export
template <typename T>
void print_typeof (T const&);

#endif // MYFIRST_HPP
```

Exported templates can be used without their definition being visible. In other words, the point where a template is being used and the point where it is defined can be in two different translation units. In our example, the file myfirst.hpp now contains only the declaration of the member functions of the class template, and this is sufficient to use those members. Comparing this with the original code that was triggering linker errors, we had to add only one export keyword in our code and things now work just fine.

Within a preprocessed file (that is, within a translation unit), it is sufficient to mark the first declaration of a template with export. Later redeclarations, including definitions, implicitly keep that attribute. This is why myfirst.cpp does not need to be modified in our example. The definitions in this file are implicitly exported because they were so declared in the #included header file. On the other hand, it is perfectly acceptable to provide redundant export keywords on template definitions, and doing so may improve the readability of the code.

The keyword export really applies to function templates, member functions of class templates, member function templates, and static data members of class templates. export can also be applied to a class template declaration. It implies that every one of its exportable members is exported, but class templates themselves are not actually exported (hence, their definitions still appear in header files). You can still have implicitly or explicitly defined inline member functions. However, these inline functions are not exported:

```
export template <typename T>
class MyClass {
  public:
    void memfun1();      // exported
    void memfun2() {     // not exported because implicitly inline

    }
    void memfun3();      // not exported because explicitly inline

};

template <typename T>
inline void MyClass<T>::memfun3 ()
```

6.4 Templates and inline

Declaring short functions to be inline is a common tool to improve the running time of programs. The inline specifier indicates to the implementation that inline substitution of the function body at the point of call is preferred over the usual function call mechanism. However, an implementation is not required to perform this inline substitution at the point of call.

Both function templates and inline functions can be defined in multiple translation units. This is usually achieved by placing the definition in a header file that is included by multiple dot-C files.

This may lead to the impression that function templates are inline by default. However, they're not. If you write function templates that should be handled as inline functions, you should use the inline specifier (unless the function is inline already because it is defined inside a class definition).

Therefore, many short template functions that are not part of a class definition should be declared with inline. [3]

[3] We may not always apply this rule of thumb because it may distract from the topic at hand.

6.5 Precompiled Headers

Even without templates, C++ header files can become very large and therefore take a long time to compile. Templates add to this tendency, and the outcry of waiting programmers has in many cases driven vendors to implement a scheme usually known as precompiled headers. This scheme operates outside the scope of the standard and relies on vendor-specific options. Although we leave the details on how to create and use precompiled header files to the documentation of the various C++ compilation systems that have this feature, it is useful to gain some understanding of how it works.

When a compiler translates a file, it does so starting from the beginning of the file and works through to the end. As it processes each token from the file (which may come from #included files), it adapts its internal state, including such things as adding entries to a table of symbols so they may be looked up later. While doing so, the compiler may also generate code in object files.

The precompiled header scheme relies on the fact that code can be organized in such a manner that many files start with the same lines of code. Let's assume for the sake of argument that every file to be compiled starts with the same N lines of code. We could compile these N lines and save the complete state of the compiler at that point in a so-called precompiled header. Then, for every file in our program, we could reload the saved state and start compilation at line N+1. At this point it is worthwhile to note that reloading the saved state is an operation that can be orders of magnitude faster than actually compiling the first N lines. However, saving the state in the first place is typically more expensive than just compiling the N lines. The increase in cost varies roughly from 20 to 200 percent.

The key to making effective use of precompiled headers is to ensure that—as much as possible— files start with a maximum number of common lines of code. In practice this means the files must start with the same #include directives, which (as mentioned earlier) consume a substantial portion of our build time. Hence, it can be very advantageous to pay attention to the order in which headers are included. For example, the following two files

```
#include <iostream>
#include <vector>
#include <list>
```

and

```
#include <list>
#include <vector>
```

inhibit the use of precompiled headers because there is no common initial state in the sources.

Some programmers decide that it is better to #include some extra unnecessary headers than to pass on an opportunity to accelerate the translation of a file using a precompiled header. This decision can considerably ease the management of the inclusion policy. For example, it is usually relatively straightforward to create a header file named std.hpp that includes all the standard headers [4]:

[4] In theory, the standard headers do not actually need to correspond to physical files. In practice, however, they do, and the files are very large.

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

6.6 Debugging Templates

Templates raise two classes of challenges when it comes to debugging them. One set of challenges is definitely a problem for writers of templates: How can we ensure that the templates we write will function for any template arguments that satisfy the conditions we document? The other class of problems is almost exactly the opposite: How can a user of a template find out which of the template parameter requirements it violated when the template does not behave as documented?

Before we discuss these issues in depth, it is useful to contemplate the kinds of constraints that may be imposed on template parameters. In this section we deal mostly with the constraints that lead to compilation errors when violated, and we call these constraints syntactic constraints. Syntactic constraints can include the need for a certain kind of constructor to exist, for a particular function call to be unambiguous, and so forth. The other kind of constraint we call semantic constraints. These constraints are much harder to verify mechanically. In the general case, it may not even be practical to do so. For example, we may require that there be a < operator defined on a template type parameter (which is a syntactic constraint), but usually we'll also require that the operator actually defines some sort of ordering on its domain (which is a semantic constraint).

The term concept is often used to denote a set of constraints that is repeatedly required in a template library. For example, the C++ standard library relies on such concepts as random access iterator and default constructible. Concepts can form hierarchies in the sense that one concept can be a refinement of another. The more refined concept includes all the constraints of the other concept but adds a few more. For example, the concept random access iterator refines the concept bidirectional iterator in the C++ standard library. With this terminology in place, we can say that debugging template code includes a significant amount of determining how concepts are violated in the template implementation and in their use.

6.6.1 Decoding the Error Novel

Ordinary compilation errors are normally quite succinct and to the point. For example, when a compiler says "class X has no member 'fun'," it usually isn't too hard to figure out what is wrong in our code (for example, we might have mistyped run as fun). Not so with templates. Consider the following relatively simple code excerpt using the C++ standard library. It contains a fairly small mistake: list<string> is used, but we are searching using a greater<int> function object, which should have been a greater<string> object:

```
std::list<std::string> coll;

// Find the first element greater than "A"
std::list<std::string>::iterator pos;
pos = std::find_if(coll.begin(),coll.end(),                 // range
                   std::bind2nd(std::greater<int>(),"A")); // criterion
```

This sort of mistake commonly happens when cutting and pasting some code and forgetting to adapt parts of it.

A version of the popular GNU C++ compiler reports the following error:

```
/local/include/stl/_algo.h: In function 'struct _STL::_List_iterator<_STL::basic
_string<char,_STL::char_traits<char>,_STL::allocator<char> >,_STL::_Nonconst_tra
its<_STL::basic_string<char,_STL::char_traits<char>,_STL::allocator<char> > > >
_STL::find_if<_STL::_List_iterator<_STL::basic_string<char,_STL::char_traits<cha
r>,_STL::allocator<char> >,_STL::_Nonconst_traits<_STL::basic_string<char,_STL::
char_traits<char>,_STL::allocator<char> > > >, _STL::binder2nd<_STL::greater<int
> > >(_STL::_List_iterator<_STL::basic_string<char,_STL::char_traits<char>,_STL:
:allocator<char> >,_STL::_Nonconst_traits<_STL::basic_string<char,_STL::char_tra
```

6.7 Afternotes

The organization of source code in header files and dot-C files is a practical consequence of various incarnations of the so-called one-definition rule or ODR. An extensive discussion of this rule is presented in Appendix A.

The inclusion versus separation model debate has been a controversial one. The inclusion model is a pragmatic answer dictated largely by existing practice in C++ compiler implementations. However, the first C++ implementation was different: The inclusion of template definitions was implicit, which created a certain illusion of separation (see Chapter 10 for details on this original model).

[StroustrupDnE] contains a good presentation of Stroustrup's vision for template code organization and the associated implementation challenges. It clearly wasn't the inclusion model. Yet, at some point in the standardization process, it seemed as if the inclusion model was the only viable approach after all. After some intense debates, however, those envisioning a more decoupled model garnered sufficient support for what eventually became the separation model. Unlike the inclusion model, this was a theoretical model not based on any existing implementation. It took more than five years to see its first implementation published (May 2002).

It is sometimes tempting to imagine ways of extending the concept of precompiled headers so that more than one header could be loaded for a single compilation. This would in principle allow for a finer grained approach to precompilation. The obstacle here is mainly the preprocessor: Macros in one header file can entirely change the meaning of subsequent header files. However, once a file has been precompiled, macro processing is completed, and it is hardly practical to attempt to patch a precompiled header for the preprocessor effects induced by other headers.

A fairly systematic attempt to improve C++ compiler diagnostics by adding dummy code in high-level templates can be found in Jeremy Siek's Concept Check Library (see [BCCL]). It is part of the Boost library (see [Boost]).

6.8 Summary

- Templates challenge the classic compiler-plus-linker model. Therefore there are different approaches to organize template code: the inclusion model, explicit instantiation, and the separation model.
- Usually, you should use the inclusion model (that is, put all template code in header files).
- By separating template code into different header files for declarations and definitions, you can more easily switch between the inclusion model and explicit instantiation.
- The C++ standard defines a separate compilation model for templates (using the keyword export). It is not yet widely available, however.
- Debugging code with templates can be challenging.
- Template instances may have very long names.
- To take advantage of precompiled headers, be sure to keep the same order for #include directives.

Chapter 7. Basic Template Terminology

So far we have introduced the basic concept of templates in C++. Before we go into details, let's look at the terms of the concepts we use. This is necessary because, inside the C++ community (and even in the standard), there is a lack of precision regarding concepts and terminology.

7.1 "Class Template" or "Template Class"?

In C++, structs, classes, and unions are collectively called class types. Without additional qualification, the word "class" in plain text type is meant to include class types introduced with either the keyword class or the keyword struct. [1] Note specifically that "class type" includes unions, but "class" does not.

[1] In C++, the only difference between class and struct is that the default access for class is private whereas the default access for struct is public. However, we prefer to use class for types that use new C++ features, and we use struct for ordinary C data structure that can be used as "plain old data" (POD).

There is some confusion about how a class that is a template is called:

- The term class template states that the class is a template. That is, it is a parameterized description of a family of classes.

- The term template class on the other hand has been used

  - as a synonym for class template.

  - to refer to classes generated from templates.

  - to refer to classes with a name that is a template-id.

The difference between the second and third meaning is somewhat subtle and unimportant for the remainder of the text.

Because of this imprecision, we avoid the term template class in this book.

Similarly, we use function template and member function template, but avoid template function and template member function.

7.2 Instantiation and Specialization

The process of creating a regular class, function, or member function from a template by substituting actual values for its arguments is called template instantiation. This resulting entity (class, function, or member function) is generically called a specialization.

However, in C++ the instantiation process is not the only way to produce a specialization. Alternative mechanisms allow the programmer to specify explicitly a declaration that is tied to a special substitution of template parameters. As we introduced in Section 3.3 on page 27, such a specialization is introduced by template<>:

```
template <typename T1, typename T2>    // primary class template
class MyClass {

};

template<>                             // explicit specialization
class MyClass<std::string,float> {

};
```

Strictly speaking, this is called a so-called explicit specialization (as opposed to an instantiated or generated specialization).

As introduced in Section 3.4 on page 29, specializations that still have template parameters are called partial specializations:

```
template <typename T>                 // partial specialization
class MyClass<T,T> {

};

template <typename T>                 // partial specialization
class MyClass<bool,T> {

};
```

When talking about (explicit or partial) specializations, the general template is also called the primary template.

7.3 Declarations versus Definitions

So far, the words declaration and definition have been used only a few times in this book. However, these words carry with them a rather precise meaning in standard C++, and that is the meaning that we use.

A declaration is a C++ construct that introduces or reintroduces a name into a C++ scope. This introduction always includes a partial classification of that name, but the details are not required to make a valid declaration. For example:

```
class C;        // a declaration of C as a class
void f(int p);  // a declaration of f() as a function and p as a named parameter
extern int v;   // a declaration of v as a variable
```

Note that even though they have a "name," macro definitions and goto labels are not considered declarations in C++.

Declarations become definitions when the details of their structure are made known or, in the case of variables, when storage space must be allocated. For class type and function definitions, this means a brace-enclosed body must be provided. For variables, initializations and a missing extern lead to definitions. Here are examples that complement the preceding nondefinition declarations:

```
class C {};         // definition (and declaration) of class C

void f(int p) {     // definition (and declaration) of function f()
    std::cout << p << std::endl;
}

extern int v = 1;  // an initializer makes this a definition for v

int w;              // global variable declarations not preceded by
                    // extern are also definitions
```

By extension, the declaration of a class template or function template is called a definition if it has a body. Hence,

```
template <typename T>
void func (T);
```

is a declaration that is not a definition, whereas

```
template <typename T>
class S {};
```

is in fact a definition.

7.4 The One-Definition Rule

The C++ language definition places some constraints on the redeclaration of various entities. The totality of these constraints is known as the one-definition rule or ODR. The details of this rule are quite complex and span a large variety of situations. Later chapters illustrate the various resulting facets in each applicable context, and you can find a complete description of the ODR in Appendix A. For now, it suffices to remember the following ODR basics:

- Noninline functions and member functions, as well as global variables and static data members should be defined only once across the whole program.
- Class types (including structs and unions) and inline functions should be defined at most once per translation unit, and all these definitions should be identical.

A translation unit is what results from preprocessing a source file; that is, it includes the contents named by #include directives.

In the remainder of this book, linkable entity means one of the following: a noninline function or member function, a global variable or a static data member, including any such things generated from a template.

7.5 Template Arguments versus Template Parameters

Compare the following class template

```
template <typename T, int N>
class ArrayInClass {
  public:
    T array[N];
};
```

with a similar plain class:

```
class DoubleArrayInClass {
  public:
    double array[10];
};
```

The latter becomes essentially equivalent to the former if we replace the parameters T and N by double and 10 respectively. In C++, the name of this replacement is denoted as

```
ArrayInClass<double,10>
```

Note how the name of the template is followed by so-called template arguments in angle brackets.

Regardless of whether these arguments are themselves dependent on template parameters, the combination of the template name, followed by the arguments in angle brackets, is called a template-id.

This name can be used much like a corresponding nontemplate entity would be used. For example:

```
int main()
{
    ArrayInClass<double,10> ad;
    ad.array[0] = 1.0;
}
```

It is essential to distinguish between template parameters and template arguments. In short, you can say that you "pass arguments to become parameters." [2] Or more precicely:

[2] In the academic world, arguments are sometimes called actual parameters whereas parameters are called formal parameters.

- Template parameters are those names that are listed after the keyword template in the template declaration or definition (T and N in our example).

- Template arguments are the items that are substituted for template parameters (double and 10 in our example). Unlike template parameters, template arguments can be more than just "names."

The substitution of template parameters by template arguments is explicit when indicated with a template-id, but there

Part II: Templates in Depth

The first part of this book provided a tutorial for most of the language concepts underlying C++ templates. That presentation is sufficient to answer the majority of questions that may arise in everyday C++ programming. The second part of this book provides a reference that answers even the more unusual questions that arise when pushing the envelope of the language to achieve some advanced software effect. If desired, you can skip this part on a first read and return to specific topics as prompted by references in later chapters or after looking up a concept in the index.

Our goal is to be clear and complete, but also to keep the discussion concise. To this end, our examples are short and often somewhat artificial. This also ensures that we don't stray from the topic at hand to unrelated issues.

In addition, we look at possible future changes and extensions for the templates language feature in C++. Topics include:

- Fundamental template declaration issues
- The meaning of names in templates
- The C++ template instantiation mechanisms
- The template argument deduction rules
- Specialization and overloading
- Future possibilities

Chapter 8. Fundamentals in Depth

In this chapter we review some of the fundamentals introduced in the first part of this book in depth: the declaration of templates, the restrictions on template parameters, the constraints on template arguments, and so forth.

8.1 Parameterized Declarations

C++ currently supports two fundamental kinds of templates: class templates and function templates (see Section 13.6 on page 212 for a possible future change in this area). This classification includes member templates. Such templates are declared much like ordinary classes and functions, except for being introduced by a parameterization clause of the form

```
template<  parameters here  >
```

or perhaps

```
export template<  parameters here  >
```

(see Section 6.3 on page 68 and Section 10.3.3 on page 149 for a detailed explanation of the keyword export).

We'll come back to the actual template parameter declarations in a later section. An example illustrates the two kinds of templates, both as class members and as ordinary namespace scope declarations:

```
template <typename T>
class List {                      // a namespace scope class template
  public:
    template <typename T2>        // a member function template
    List (List<T2> const&);       // (constructor)

};
template <typename T>
 template <typename T2>
List<T>::List (List<T2> const& b) // an out-of-class member function
{                                 // template definition

}

template <typename T>
int length (List<T> const&);      // a namespace scope function template

class Collection {
    template <typename T>         // an in-class member class template
    class Node {                  // definition

    };

    template <typename T>         // another member class template,
    class Handle;                 // without its definition

    template <typename T>         // an in-class (and therefore implicitly
    T* alloc() {                  // inline) member function template
                                  // definition
    }

};

template <typename T>             // an out-of-class member class
class Collection::Node {          // template definition

};
```

Note how member templates defined outside their enclosing class can have multiple template< > parameterization clauses: one for the template itself and one for every enclosing class template. The clauses are listed starting from the outermost class template.

8.2 Template Parameters

There are three kinds of template parameters:

1. Type parameters (these are by far the most common)
2. Nontype parameters
3. Template template parameters

Template parameters are declared in the introductory parameterization clause of a template declaration. Such declarations do not necessarily need to be named:

```
template <typename, int>
class X;
```

A parameter name is, of course, required if the parameter is referred to later in the template. Note also that a template parameter name can be referred to in a subsequent parameter declaration (but not before):

```
template <typename T,              // the first parameter is used in the
          T* Root,                 // declaration of the second one and
          template<T*> class Buf> // the third one
class Structure;
```

8.2.1 Type Parameters

Type parameters are introduced with either the keyword typename or the keyword class: The two are entirely equivalent. [2] The keyword must be followed by a simple identifier and that identifier must be followed by a comma to denote the start of the next parameter declaration, a closing angle bracket (>) to denote the end of the parameterization clause, or an equal sign (=) to denote the beginning of a default template argument.

[2] The keyword class does not imply that the substituting argument should be a class type. It could be almost any accessible type. However, class types that are defined in a function (local classes) cannot be used as template arguments (independent of whether the parameter was declared with typename or class).

Within a template declaration, a type parameter acts much like a typedef name. For example, it is not possible to use an elaborated name of the form class T when T is a template parameter, even if T were to be substituted by a class type:

```
template <typename Allocator>
class List {
    class Allocator* allocator;  // ERROR
    friend class Allocator;      // ERROR

};
```

It is possible that a mechanism to enable such a friend declaration will be added in the future.

8.3 Template Arguments

Template arguments are the "values" that are substituted for template parameters when instantiating a template. These values can be determined using several different mechanisms:

- Explicit template arguments: A template name can be followed by explicit template argument values enclosed in angle brackets. The resulting name is called a template-id.

- Injected class name: Within the scope of a class template X with template parameters P1, P2, , the name of that template (X) can be equivalent to the template-id X<P1, P2, >. See Section 9.2.3 on page 126 for details.

- Default template arguments: Explicit template arguments can be omitted from class template instances if default template arguments are available. However, even if all template parameters have a default value, the (possibly empty) angle brackets must be provided.

- Argument deduction: Function template arguments that are not explicitly specified may be deduced from the types of the function call arguments in a call. This is described in detail in Chapter 11. Deduction is also done in a few other situations. If all the template arguments can be deduced, no angle brackets need to be specified after the name of the function template.

8.3.1 Function Template Arguments

Template arguments for a function template can be specified explicitly or deduced from the way the template is used. For example:

```
// details/max.cpp

template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a<b?b:a;
}

int main()
{
    max<double>(1.0, -3.0);  // explicitly specify template argument
    max(1.0, -3.0);          // template argument is implicitly deduced
                             // to be double
    max<int>(1.0, 3.0);      // the explicit <int> inhibits the deduction;
                             // hence the result has type int
}
```

Some template arguments can never be deduced (see Chapter 11). The corresponding parameters are best placed at the beginning of the list of template parameters so they can be specified explicitly while allowing the other arguments to be deduced. For example:

8.4 Friends

The basic idea of friend declarations is a simple one: Identify classes or functions that have a privileged connection with the class in which the friend declaration appears. Matters are somewhat complicated, however, by two facts:

1. A friend declaration may be the only declaration of an entity.

2. A friend function declaration can be a definition.

Friend class declarations cannot be definitions and therefore are rarely problematic. In the context of templates, the only new facet of friend class declarations is the ability to name a particular instance of a class template as a friend:

```
template <typename T>
class Node;

template <typename T>
class Tree {
    friend class Node<T>;

};
```

Note that the class template must be visible at the point where one of its instances is made a friend of a class or class template. With an ordinary class, there is no such requirement:

```
template <typename T>
class Tree {
    friend class Factory;         // OK, even if first declaration of Factory
    friend class class Node<T>;   // ERROR if Node isn't visible
};
```

Section 9.2.2 on page 125 has more to say about this.

8.4.1 Friend Functions

An instance of a function template can be made a friend by making sure the name of the friend function is followed by angle brackets. The angle brackets can contain the template arguments, but if the arguments can be deduced, the angle brackets can be left empty:

```
template <typename T1, typename T2>
void combine(T1, T2);

class Mixer {
    friend void combine<>(int&, int&);
                    // OK: T1 = int&, T2 = int&
    friend void combine<int, int>(int, int);
                    // OK: T1 = int, T2 = int
    friend void combine<char>(char, int);
                    // OK: T1 = char T2 = int
    friend void combine<char>(char&, int);
                    // ERROR: doesn't match combine() template
    friend void combine<>(long, long) {   }
                    // ERROR: definition not allowed!
```

8.5 Afternotes

The general concept and syntax of C++ templates have remained relatively stable since their inception in the late 1980s. Class templates and function templates were part of the initial template facility. So were type parameters and nontype parameters.

However, there were also some significant additions to the original design, mostly driven by the needs of the C++ standard library. Member templates may well be the most fundamental of those additions. Curiously, only member function templates were formally voted into the C++ standard. Member class templates became part of the standard by an editorial oversight.

Friend templates, default template arguments, and template template parameters are also relatively recent additions to the language. The ability to declare template template parameters is sometimes called higher-order genericity. They were originally introduced to support a certain allocator model in the C++ standard library, but that allocator model was later replaced by one that does not rely on template template parameters. Later, template template parameters came close to being removed from the language because their specification had remained incomplete until very late in the standardization process. Eventually a majority of committee members voted to keep them and their specifications were completed.

Chapter 9. Names in Templates

Names are a fundamental concept in most programming languages. They are the means by which a programmer can refer to previously constructed entities. When a C++ compiler encounters a name, it must "look it up" to identify to which entity is being referred. From an implementer's point of view, C++ is a hard language in this respect. Consider the C++ statement x*y; .Ifx and y are the names of variables, this statement is a multiplication, but if x is the name of a type, then the statement declares y as a pointer to an entity of type x.

This small example demonstrates that C++ (like C) is a so-called context-sensitive language: A construct cannot always be understood without knowing its wider context. How does this relate to templates? Well, templates are constructs that must deal with multiple wider contexts: (1) the context in which the template appears, (2) the context in which the template is instantiated, and (3) the contexts associated with the template arguments for which the template is instantiated. Hence it should not be totally surprising that "names" must be dealt with quite carefully in C++.

9.1 Name Taxonomy

C++ classifies names in a variety of ways—a large variety of ways in fact. To help cope with this abundance of terminology, we provide tables Table 9.1 and Table 9.2, which describe these classifications. Fortunately, you can gain good insight into most C++ template issues by familiarizing yourself with two major naming concepts:

1. A name is a qualified name if the scope to which it belongs is explicitly denoted using a scoperesolution operator (::) or a member access operator (. or ->). For example, this->count is a qualified name, but count is not (even though the plain count might actually refer to a class member).

2. A name is a dependent name if it depends in some way on a template parameter. For example, std::vector<T>::iterator is a dependent name if T is a template parameter, but it is a nondependent name if T is a known typedef (for example, of int).

Table 9.1. Name Taxonomy (part one)

Classification	**Explanation and Notes**
Identifier	A name that consists solely of an uninterrupted sequences of letters, underscores (_) and digits. It cannot start with a digit, and some identifiers are reserved for the implementation: You should not introduce them in your programs (as a rule of thumb, avoid leading underscores and double underscores). The concept of "letter" should be taken broadly and includes special universal character names (UCNs) that encode glyphs from nonalphabetical languages.
Operator-function-id	The keyword operator followed by the symbol for an operator— for example, operator new and operator [ ]. Many operators have alternative representations. For example, operator & can equivalently be written as operator bitand even when it denotes the unary address of operator.
Conversion-function-id	Used to denote user-defined implicit conversion operator—for example operator int&, which could also be obfuscated as operator int bitand.
Template-id	The name of a template followed by template arguments enclosed in angle brackets; for example, List<T, int, 0>. (Strictly speaking, the C++ standard allows only simple identifiers for the template name of a template-id. However, this is probably an oversight and an

9.2 Looking Up Names

There are many small details to looking up names in C++, but we will focus only on a few major concepts. The details are necessary to ensure only that (1) normal cases are treated intuitively, and (2) pathological cases are covered in some way by the standard.

Qualified names are looked up in the scope implied by the qualifying construct. If that scope is a class, then base classes may also be looked up. However, enclosing scopes are not considered when looking up qualified names. The following illustrates this basic principle:

```
int x;

class B {
  public:
    int i;
};

class D : public B {
};
void f(D* pd)
{
    pd->i = 3;  // finds B::i
    D::x = 2;   // ERROR: does not find ::x in the enclosing scope
}
```

In contrast, unqualified names are typically looked up in successively more enclosing scopes (although in member function definitions the scope of the class and its base classes is searched before any other enclosing scopes). This is called ordinary lookup. Here is a basic example showing the main idea underlying ordinary lookup:

```
extern int count;               // (1)

int lookup_example(int count)   // (2)
{
    if (count < 0) {
        int count = 1;          // (3)
        lookup_example(count);  // unqualified count refers to (3)
    }
    return count + ::count;     // the first (unqualified) count refers to (2);
}                               // the second (qualified) count refers to (1)
```

A more recent twist to the lookup of unqualified names is that—in addition to ordinary lookup—they may sometimes undergo so-called argument-dependent lookup (ADL). [1] Before proceeding with the details of ADL, let's motivate the mechanism with our perennial max() template:

[1] This is also called Koenig lookup (or extended Koenig lookup) after Andrew Koenig, who first proposed a variation of this mechanism.

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}
```

Suppose now that we need to apply this template to a type defined in another namespace:

```
namespace BigMath {
    class BigNumber {
```

9.3 Parsing Templates

Two fundamental activities of compilers for most programming languages are tokenization—also called scanning or lexing—and parsing. The tokenization process reads the source code as a sequence of characters and generates a sequence of tokens from it. For example, on seeing the sequence of characters int* p=0;, the "tokenizer" will generate token descriptions for a keyword int,a symbol/operator *, an identifier p, a symbol/operator =, an integer literal 0, and a symbol/operator ;.

A parser will then find known patterns in the token sequence by recursively reducing tokens or previously found patterns into higher level constructs. For example, the token 0 is a valid expression, the combination * followed by an identifier p is a valid declarator, and that declarator followed by "=" followed by the expression "0" is also a valid declarator. Finally, the keyword int is a known type name, and, when followed by the declarator *p=0, you get the initializating declaration of p.

9.3.1 Context Sensitivity in Nontemplates

As you may know or expect, tokenizing is easier than parsing. Fortunately, parsing is a subject for which a solid theory has been developed, and many useful languages are not hard to parse using this theory. However, the theory works best for so-called context-free language, and we have already noted that C++ is context sensitive. To handle this, a C++ compiler will couple a symbol table to the tokenizer and parser: When a declaration is parsed, it is entered in the symbol table. When the tokenizer finds an identifier, it looks it up and annotates the resulting token if it finds a type.

For example, if the C++ compiler sees

```
x*
```

the tokenizer looks up x. If it finds a type, the parser sees

```
identifier, type, x
symbol, *
```

and concludes that a declaration has started. However, if x is not found to be a type, then the parser receives from the tokenizer

```
identifier, nontype, x
symbol, *
```

and the construct can be parsed validly only as a multiplication. The details of these principles are dependent on the particular implementation strategy, but the gist should be there.

Another example of context sensitivity is illustrated in the following expression:

```
X<1>(0)
```

If X is the name of a class template, then the previous expression casts the integer 0 to the type X<1> generated from that template. If X is not a template, then the previous expression is equivalent to

```
(X<1)>0
```

In other words, X is compared with 1, and the result of that comparison—true or false, implicitly converted to 1 or 0

9.4 Derivation and Class Templates

Class templates can inherit or be inherited from. For many purposes, there is nothing significantly different between the template and nontemplate scenarios. However, there is one important subtlety when deriving a class template from a base class referred to by a dependent name. Let's first look at the somewhat simpler case of nondependent base classes.

9.4.1 Nondependent Base Classes

In a class template, a nondependent base class is one with a complete type that can be determined without knowing the template arguments. In other words, the name of this base is denoted using a nondependent name. For example:

```
template<typename X>
class Base {
  public:
    int basefield;
    typedef int T;
};

class D1: public Base<Base<void> > {  // not a template case really
  public:
    void f() { basefield = 3; }       // usual access to inherited member
};

template<typename T>
class D2 : public Base<double> {      // nondependent base
  public:
    void f() { basefield = 7; }       // usual access to inherited member
    T strange;        // T is Base<double>::T, not the template parameter!
};
```

Nondependent bases in templates behave very much like bases in ordinary nontemplate classes, but there is a slightly unfortunate surprise: When an unqualified name is looked up in the templated derivation, the nondependent bases are considered before the list of template parameters. This means that in the previous example, the member strange of the class template D2 always has the type T corresponding to Base<double>::T (in other words, int). For example, the following function is not valid C++ (assuming the previous declarations):

```
void g (D2<int*>& d2, int* p)
{
    d2.strange = p; // ERROR: type mismatch!
}
```

This is counterintuitive and requires the writer of the derived template to be aware of names in the nondependent bases from which it derives—even when that derivation is indirect or the names are private. It would probably have been preferable to place template parameters in the scope of the entity they "templatize."

9.4.2 Dependent Base Classes

In the previous example, the base class is fully determined. It does not depend on a template parameter. This implies that a C++ compiler can look up nondependent names in those base classes as soon as the template definition is seen. An alternative—not allowed by the C++ standard—would consist in delaying the lookup of such names until the template is instantiated. The disadvantage of this alternative approach is that it also delays any error messages resulting from missing symbols until instantiation. Hence, the C++ standard specifies that a nondependent name appearing in a template is looked up as soon as it is encountered. Keeping this in mind, consider the following example:

9.5 Afternotes

The first compiler really to parse template definitions was developed by a company called Taligent in the mid-1990s. Before that—and even after that—most compilers treated templates as a sequence of tokens to be played back through the parser at instantiation time. Hence no parsing was done, except for a minimal amount sufficient to find the end of a template definition. Bill Gibbons was Taligent's representative to the C++ committee and was the principal advocate for making templates unambiguously parsable. The Taligent effort was not released until the compiler was acquired and completed by Hewlett-Packard (HP), to become the aC++ compiler. Among its competitive advantages, the aC++ compiler was quickly recognized for its high quality diagnostics. The fact that template diagnostics were not always delayed until instantiation time undoubtedly contributed to this perception.

Relatively early during the development of templates, Tom Pennello—a widely recognized parsing expert working for Metaware—noted some of the problems associated with angle brackets. Stroustrup also comments on that topic in [StroustrupDnE] and argues that humans prefer to read angle brackets rather than parentheses. However, other possibilities exist, and Pennello specifically proposed braces (for example, List{::X}) at a C++ standards meeting in 1991 (held in Dallas). [8] At that time the extent of the problem was more limited because templates nested inside other templates—so-called member templates—were not valid and thus the discussion of Section 9.3.3 on page 132 was largely irrelevant. As a result, the committee declined the proposal to replace the angle brackets.

[8] Braces are not entirely without problems either. Specifically, the syntax to specialize class templates would require nontrivial adaptation.

The name lookup rule for nondependent names and dependent base classes that is described in Section 9.4.2 on page 136 was introduced in the C++ standard in 1993. It was described to the "general public" in Bjarne Stroustrup's [StroustrupDnE] in early 1994. Yet the first generally available implementation of this rule did not appear until early 1997 when HP incorporated it into their aC++ compiler, and by then large amounts of code derived class templates from dependent bases. Indeed, when the HP engineers started testing their implementation, they found that most of the programs that used templates in nontrivial ways no longer compiled. [9] In particular, all implementations of the STL [10] broke the rule in many hundreds—and sometimes thousands—of places. To ease the transition process for their customers, HP softened the diagnostic associated with code that assumed that nondependent names could be found in dependent base classes as follows. When a nondependent name used in the scope of a class template is not found using the standard rules, aC++ peeks inside the dependent bases. If the name is still not found, a hard error is issued and compilation fails. However, if the name is found in a dependent base, a warning is issued, and the name is marked to be treated as if it were dependent, so that lookup will be reattempted at instantiation time.

[9] Fortunately, they found out before they released the new functionality.

[10] Ironically, the first of these implementations had been developed by HP as well.

The lookup rule that causes a name in nondependent bases to hide an identically named template parameter (Section 9.4.1 on page 135) is an oversight, and it is not impossible that this will be changed in a revision of the standard. In any case, it is probably wise to avoid code with template parameter names that are also used in nondependent base classes.

Andrew Koenig first proposed ADL for operator functions only (which is why ADL is sometimes called Koenig lookup). The motivation was primarily esthetic: explicitly qualifying operator names with their enclosing namespace looks awkward at best (for example, instead of a+b we may need to write N::operator+(a, b)) and having to write

Chapter 10. Instantiation

Template instantiation is the process that generates types and functions from generic template definitions. [1] The concept of instantiation of C++ templates is fundamental but also somewhat intricate. One of the underlying reasons for this intricacy is that the definitions of entities generated by a template are no longer limited to a single location in the source code. The location of the template, the location where the template is used, and the locations where the template arguments are defined all play a role in the meaning of the entity.

[1] The term instantiation is sometimes also used to refer to the creation of objects from types. In this book, however, it always refers to template instantiation.

In this chapter we explain how we can organize our source code to enable proper template use. In addition, we survey the various methods that are used by the most popular C++ compilers to handle template instantiation. Although all these methods should be semantically equivalent, it is useful to understand basic principles of your compiler's instantiation strategy. Each mechanism comes with its set of little quirks when building real-life software and, conversely, each influenced the final specifications of standard C++.

10.1 On-Demand Instantiation

When a C++ compiler encounters the use of a template specialization, it will create that specialization by substituting the required arguments for the template parameters. [2] This is done automatically and requires no direction from the client code (or from the template definition for that matter). This on-demand instantiation feature sets C++ templates apart from similar facilities in other compiled languages. It is sometimes also called implicit or automatic instantiation.

[2] The term specialization is used in the general sense of an entity that is a specific instance of a template (see Chapter 7). It does not refer to the explicit specialization mechanism described in Chapter 12.

On-demand instantiation implies that the compiler usually needs access to the full definition (in other words, not just the declaration) of the template and some of its members at the point of use. Consider the following tiny source code file:

```
template<typename T> class C;  // (1) declaration only

C<int>* p = 0;                 // (2) fine: definition of C<int> not needed

template<typename T>
class C {
  public:
    void f();                 // (3) member declaration
};                            // (4) class template definition completed

void g (C<int>& c)            // (5) use class template declaration only
{
   c.f();                     // (6) use class template definition;
}                             //     will need definition of C::f()
```

At point (1) in the source code, only the declaration of the template is available, not the definition (such a declaration is sometimes called a forward declaration). As is the case with ordinary classes, you do not need the definition of a class template to be in scope to declare pointers or references to this type (as was done at point (2)). For example, the type of the parameter of function g does not require the full definition of the template C. However, as soon as a component needs to know the size of a template specialization or if it accesses a member of such a specialization, the entire class template definition is required to be in scope. This explains why at point (6) in the source code, the class template definition must seen; otherwise, the compiler cannot verify that the member exists and is accessible (not private or protected).

Here is another expression that needs the instantiation of the previous class template because the size of C<void> is needed:

```
C<void>* p = new C<void>;
```

In this case, instantiation is needed so that the compiler can determine the size of C<void>.You might observe that for this particular template, the type of the argument X substituted for T will not influence the size of the template because in any case, C<X> is an empty class. However, a compiler is not required to detect this. Furthermore, instantiation is also needed in this example to determine whether C<void> has an accessible default constructor and to ensure C<void> does not declare private operators new or delete.

The need to access a member of a class template is not always very explicitly visible in the source code. For example, C++ overload resolution requires visibility into class types for parameters of candidate functions:

```
template<typename T>
```

10.2 Lazy Instantiation

The examples so far illustrate requirements that are not fundamentally different from the requirements when using nontemplate classes. Many uses require a class type to be complete. For the template case, the compiler will generate this complete definition from the class template definition.

A pertinent question now arises How much of the template is instantiated? A vague answer is the following: Only as much as is really needed. In other words, a compiler should be "lazy" when instantiating templates. Let's look at exactly what this laziness entails.

When a class template is implicitly instantiated, each declaration of its members is instantiated as well, but the corresponding definitions are not. There are a few exceptions to this. First, if the class template contains an anonymous union, the members of that union's definition are also instantiated. [3] The other exception occurs with virtual member functions. Their definitions may or may not be instantiated as a result of instantiating a class template. Many implementations will, in fact, instantiate the definition because the internal structure that enables the virtual call mechanism requires the virtual functions actually to exist as linkable entities.

[3] Anonymous unions are always special in this way: Their members can be considered to be members of the enclosing class. An anonymous union is primarily a construct that says that some class members share the same storage.

Default function call arguments are considered separately when instantiating templates. Specifically, they are not instantiated unless there is a call to that function (or member function) that actually makes use of the default argument. If, on the other hand, that function is called with explicit arguments that override the default, then the default arguments are not instantiated.

Let's put together an example that illustrates all these issues:

```
// details/lazy.cpp

template <typename T>
class Safe {
};

template <int N>
class Danger {
  public:
    typedef char Block[N];  // would fail for N<=0
};

template <typename T, int N>
class Tricky {
  public:
    virtual ~Tricky() {
    }
    void no_body_here(Safe<T> = 3);
    void inclass() {
        Danger<N> no_boom_yet;
    }
    // void error() { Danger<0> boom; }
    // void unsafe(T (*p)[N]);
    T operator->();
    // virtual Safe<T> suspect();
```

10.3 The C++ Instantiation Model

Template instantiation is the process of obtaining a regular class or function from a corresponding template entity by appropriately substituting the template parameters. This may sound fairly straightforward, but in practice many details need to be formally established.

10.3.1 Two-Phase Lookup

In Chapter 9 we saw that dependent names cannot be resolved when parsing templates. Instead, they are looked up again at the point of instantiation. Nondependent names, however, are looked up early so that many errors can be diagnosed when the template is first seen. This leads to the concept of two-phase lookup [5]: The first phase is the parsing of a template, and the second phase is its instantiation.

[5] Beside two-phase lookup, terms such as two-stage lookup or two-phase name lookup are also used.

During the first phase, nondependent names are looked up while the template is being parsed using both the ordinary lookup rules and, if applicable, the rules for argument-dependent lookup (ADL). Unqualified dependent names (which are dependent because they look like the name of a function in a function call with dependent arguments) are also looked up that way, but the result of the lookup is not considered complete until an additional lookup is performed when the template is instantiated.

During the second phase, which occurs when templates are instantiated at a point called the point of instantiation (POI), dependent qualified names are looked up (with the template parameters replaced with the template arguments for that specific instantiation), and an additional ADL is performed for the unqualified dependent names.

10.3.2 Points of Instantiation

We have already illustrated that there are points in the source of template clients where a C++ compiler must have access to the declaration or the definition of a template entity. A point of instantiation (POI) is created when a code construct refers to a template specialization in such a way that the definition of the corresponding template needs to be instantiated to create that specialization. The POI is a point in the source where the substituted template could be inserted. For example:

```
class MyInt {
  public:
   MyInt(int i);
};

MyInt operator - (MyInt const&);

bool operator > (MyInt const&, MyInt const&);
typedef MyInt Int;

template<typename T>
void f(T i)
{
    if (i>0) {
        g(-i);
    }
}
// (1)
void g(Int)
{
```

10.4 Implementation Schemes

In this section we review some ways in which popular C++ implementations support the inclusion model. All these implementations rely on two classic components: a compiler and a linker. The compiler translates source code to object files, which contain machine code with symbolic annotations (cross-referencing other object files and libraries). The linker creates executable programs or libraries by combining the object files and resolving the symbolic cross-references they contain. In what follows, we assume such a model even though it is entirely possible (but not popular) to implement C++ in other ways. For example, you could imagine a C++ interpreter.

When a class template specialization is used in multiple translation units, a compiler will repeat the instantiation process in every translation unit. This poses very few problems because class definitions do not directly create low-level code. They are used only internally by a C++ implementation to verify and interpret various other expressions and declarations. In this regard, the multiple instantiations of a class definition are not materially different from the multiple inclusions of a class definition—typically through header file inclusion—in various translation units.

However, if you instantiate a (noninline) function template, the situation may be different. If you were to provide multiple definitions of an ordinary noninline function, you would violate the ODR. Assume, for example, that you compile and link a program consisting of the following two files:

```
// File a.cpp:
int main()
{
}

// File b.cpp:
int main()
{
}
```

C++ compilers will compile each module separately without any problems because indeed they are valid C++ translation units. However, your linker will most likely protest if you try to link the two together. Duplicate definitions are not allowed.

In contrast, consider the template case:

```
// File t.hpp:
// common header (inclusion model)
template<typename T>
class S {
  public:
    void f();
};

template<typename T>
void S::f()    // member definition
{
}

void helper(S<int>*);

// File a.cpp:
#include "t.hpp"

void helper(S<int>* s)
{
    s->f();    // (1) first point of instantiation of S::f
```

10.5 Explicit Instantiation

It is possible to create explicitly a point of instantiation for a template specialization. The construct that achieves this is called an explicit instantiation directive. Syntactically, it consists of the keyword template followed by a declaration of the specialization to be instantiated. For example:

```
template<typename T>
void f(T) throw(T)
{
}

// four valid explicit instantiations:
template void f<int>(int) throw(int);
template void f<>(float) throw(float);
template void f(long) throw(long);
template void f(char);
```

Note that every instantiation directive is valid. Template arguments can be deduced (see Chapter 11), and exception specifications can be omitted. If they are not omitted, they must match the one of the template.

Members of class templates can also be explicitly instantiated in this way:

```
template<typename T>
class S {
  public:
    void f() {
    }
};

template void S<int>::f();

template class S<void>;
```

Furthermore, all the members of a class template specialization can be explicitly instantiated by explicitly instantiating the class template specialization.

Many early C++ compilation systems did not have automatic instantiation capabilities when they first implemented support for templates. Instead, some systems required that the function template specializations used by a program be manually instantiated in a single location. This manual instantiation usually involved implementation-specific #pragma directives.

The C++ standard therefore codified this practice by specifying a clean syntax for it. The standard also specifies that there can be at most one explicit instantiation of a certain template specialization in a program. Furthermore, if a template specialization is explicitly instantiated, it should not be explicitly specialized, and vice versa.

In the original context of manual instantiations, these limitations may seem harmless, but in current practice they cause some grief.

First, consider a library implementer who releases a first version of a function template:

```
// File toast.hpp:
template<typename T>
void toast(T const& x)
```

10.6 Afternotes

This chapter deals with two related but different issues: the C++ template compilation models and various C++ template instantiation mechanisms.

The compilation model determines the meaning of a template at various stages of the translation of a program. In particular, it determines what the various constructs in a template mean when it is instantiated. Name lookup is an essential ingredient of the compilation model of course. When we talk about the inclusion model and the separation model, we talk about compilation models. These models are part of the language definition.

The instantiation mechanisms are the external mechanisms that allow C++ implementations to create instantiations correctly. These mechanisms may be constrained by requirements of the linker and other software building tools.

However, the original (Cfront) implementation of templates transcended these two concepts. It created new translation units for the instantiation of templates using a particular convention for the organization of source files. The resulting translation unit was then compiled using what is essentially the inclusion model (although the C++ name lookup rules were substantially different back then). So although Cfront did not implement "separate compilation" of templates, it managed to create an illusion of separate compilation by creating implicit inclusions. Various later implementations provided a somewhat similar implicit inclusion mechanism by default (Sun Microsystems) or as an option (HP, EDG) to provide some amount of compatibility with existing code developed for Cfront.

An example illustrates the details of the Cfront implementation scheme:

```
// File template.hpp:
template<class T>  // Cfront doesn't know typename
void f(T);

// File template.cpp:
template<class T>  // Cfront doesn't know typename
void f(T)
{
}

// File app.hpp:
class App {

};

// File main.cpp:
#include "app.hpp"
#include "template.hpp"

int main()
{
    App a;
    f(a);
}
```

At link time, Cfront's iterated instantiation scheme then creates a new translation unit including files it expects to contain the implementation of the templates it found in header files. Cfront's convention for this is to replace the .h (or similar) suffix of header files by .c (or one of a few other suffixes like .C or .cpp). In this case, the generated translation unit becomes

```
// File main.cpp:
```

Chapter 11. Template Argument Deduction

Explicitly specifying template arguments on every call to a function template (for example, concat<std::string, int>(s, 3)) can quickly lead to unwieldy code. Fortunately, a C++ compiler can often automatically determine the intended template arguments using a powerful process called template argument deduction.

In this chapter we explain the details of the template argument deduction process. As is often the case in C++, there are many rules that usually produce an intuitive result. A solid understanding of this chapter allows us to avoid the more surprising situations.

11.1 The Deduction Process

The deduction process compares the types of an argument of a function call with the corresponding parameterized type of a function template and attempts to conclude the correct substitution for one or more of the deduced parameters. Each argument-parameter pair is analyzed independently, and if the conclusions differ in the end, the deduction process fails. Consider the following example:

```
template<typename T>
T const& max (T const& a, T const& b)
{
    return a<b?b:a;
}

int g = max(1, 1.0);
```

Here the first call argument is of type int so the parameter T of our original max() template is tentatively deduced to be int. The second call argument is a double, however, and so T should be double for this argument: This conflicts with the previous conclusion. Note that we say that "the deduction process fails," not that "the program is invalid." After all, it is possible that the deduction process would succeed for another template named max (function templates can be overloaded much like ordinary functions; see Section 2.4 on page 15 and Chapter 12).

If all the deduced template parameters are consistently determined, the deduction process can still fail if substituting the arguments in the rest of the function declaration results in an invalid construct. For example:

```
template<typename T>
typename T::ElementT at (T const& a, int i)
{
    return a[i];
}

void f (int* p)
{
    int x = at(p, 7);
}
```

Here T is concluded to be int* (there is only one parameter type where T appears, so there are obviously no analysis conflicts). However, substituting int* for T in the return type T::ElementT is clearly invalid C++, and the deduction process fails. [1] The error message is likely to say that no match was found for the call to at(). In contrast, if all the template arguments are mentioned explicitly, then there is no chance that the deduction process will succeed for another template, and the error message is more likely to say that the template arguments for at() are invalid. You can investigate this by comparing the diagnostic for the previous example with

[1] In this case, deduction failure leads to an error. However, this falls under the SFINAE principle (see Section 8.3.1 on page 106): If there were another function for which deduction succeeds, the code could be valid.

```
void f (int* p)
{
    int x = at<int*>(p, 7);
}
```

on your favorite C++ implementation.

We still need to explore how argument-parameter matching proceeds. We describe it in terms of matching a type A (derived from the argument type) to a parameterized type P (derived from the parameter declaration). If the parameter is declared with a reference declarator, P is taken to be the type referenced, and A is the type of the

11.2 Deduced Contexts

Parameterized types that are considerably more complex than T can be matched to a given argument

type. Here are a few examples that are still fairly basic:

```
template<typename T>
void f1(T*);

template<typename E, int N>
void f2(E(&)[N]);

template<typename T1, typename T2, typename T3>
void f3(T1 (T2::*)(T3*));

class S {
  public:
    void f(double*);
};

void g (int*** ppp)
{
    bool b[42];
    f1(ppp);    // deduces T to be int**
    f2(b);      // deduces E to be bool and N to be 42
    f3(&S::f);  // deduces T1 = void, T2=S, and T3 = double
}
```

Complex type declarations are built from more elementary constructs (pointer, reference, array, and function declarators; pointer-to-member declarators; template-ids; and so forth), and the matching process proceeds from the top-level construct and recurses through the composing elements. It is fair to say that most type declaration constructs can be matched in this way, and these are called deduced contexts. However, a few constructs are not deduced contexts:

- Qualified type names. A type name like Q<T>::X will never be used to deduce a template parameter T, for example.

- Nontype expressions that are not just a nontype parameter. A type name like S<I+1> will never be used to deduce I, for example. Neither will T be deduced by matching against a parameter of type int(&)[sizeof(S<T>)].

These limitations should come as no surprise because the deduction would, in general, not be unique (or even finite), although qualified type names are sometimes easily overlooked. A nondeduced context does not automatically imply that the program is in error or even that the parameter being analyzed cannot participate in type deduction. To illustrate this, consider the following, more intricate example:

```
// details/fppm.cpp

template <int N>
class X {
  public:
    typedef int I;
    void f(int) {
```

11.3 Special Deduction Situations

There are two situations in which the pair (A, P) used for deduction is not obtained from the arguments to a function call and the parameters of a function template. The first situation occurs when the address of a function template is taken. In this case, P is the parameterized type of the function template declarator, and A is the function type underlying the pointer that is initialized or assigned to. For example:

```
template<typename T>
void f(T, T);

void (*pf)(char, char) = &f;
```

In this example, P is void(T, T) and A is void(char, char). Deduction succeeds with T substituted with char, and pf is initialized to the address of the specialization f<char>.

The other special situation occurs with conversion operator templates. For example:

```
class S {
  public:
    template<typename T, int N> operator T[N]&();
};
```

In this case, the pair (P, A) is obtained as if it involved an argument of the type to which we are attempting to convert and a parameter type that is the return type of the conversion operator. The following code illustrates one variation:

```
void f(int (&)[20]);

void g(S s)
{
    f(s);
}
```

Here we are attempting to convert S to int (&)[20]. Type A is therefore int[20] and type P is T[N]. The deduction succeeds with T substituted with int and N with 20.

11.4 Allowable Argument Conversions

Normally, template deduction attempts to find a substitution of the function template parameters that make the parameterized type P identical to type A. However, when this is not possible, the following differences are tolerable:

- If the original parameter was declared with a reference declarator, the substituted P type may be more const/volatile-qualified than the A type.
- If the A type is a pointer or pointer-to-member type, it may be convertible to the substituted P type by a qualification conversion (in other words, a conversion that adds const and/or volatile qualifiers).
- Unless deduction occurs for a conversion operator template, the substituted P type may be a base class type of the A type, or a pointer to a base class type of the class type for which A is a pointer type. For example:

```
template<typename T>
class B<T> {
};

template<typename T>
class D : B<T> {
};

template<typename T> void f(B<T>*);

void g(D<long> dl)
{
    f(&dl);  // deduction succeeds with T substituted with long
}
```

The relaxed matching requirements are considered only if an exact match was not possible. Even so, deduction succeeds only if exactly one substitution was found to fit the A type to the substituted P type with these added conversions.

11.5 Class Template Parameters

Template argument deduction applies exclusively to function and member function templates. In particular, the arguments for a class template are not deduced from the arguments to a call of one of its constructors. For example:

```
template<typename T>
class S {
  public:
    S(T b) : a(b) {
    }
  private:
    T a;
};

S x(12); // ERROR: the class template parameter T is not deduced
         //        from the constructor call argument 12
```

11.6 Default Call Arguments

Default function call arguments can be specified in function templates just as they are in ordinary functions:

```
template<typename T>
void init (T* loc, T const& val = T())
{
    *loc = val;
}
```

In fact, as this example shows, the default function call argument can depend on a template parameter. Such a dependent default argument is instantiated only if no explicit argument is provided—a principle that makes the following example valid:

```
class S {
  public:
    S(int, int);
};

S s(0, 0);

int main()
{
   init(&s, S(7, 42));  // T() is invalid for T=S, but the default
                        // call argument T() needs no instantiation
                        // because an explicit argument is given
}
```

Even when a default call argument is not dependent, it cannot be used to deduce template arguments. This means that the following is invalid C++:

```
template<typename T>
void f (T x = 42)
{
}

int main()
{
    f<int>(); // OK: T = int
    f();      // ERROR: cannot deduce T from default call argument
}
```

11.7 The Barton-Nackman Trick

In 1994, John J. Barton and Lee R. Nackman presented a template technique that they called restricted template expansion. The technique was motivated in part by the fact that—at the time— function templates could not be overloaded [3] and namespaces were not available in most compilers.

[3] It may be worthwhile to read Section 12.2 on page 183 to understand how function template overloading works in modern C++.

To illustrate this, suppose we have a class template Array for which we want to define the equality operator ==. One possibility is to declare the operator as a member of the class template, but this is not good practice because the first argument (binding to the this pointer) is subject to conversion rules that are different from the second argument. Because operator == is meant to be symmetrical with respect to its arguments, it is preferable to declare it as a namespace scope function. An outline of a natural approach to its implementation may look like the following:

```
template<typename T>
class Array {
  public:

};

template<typename T>
bool operator == (Array<T> const& a, Array<T> const& b)
{

}
```

However, if function templates cannot be overloaded, this presents a problem: No other operator == template can be declared in that scope, and yet it is likely that such a template would be needed for other class templates. Barton and Nackman resolved this problem by defining the operator in the class as a normal friend function:

```
template<typename T>
class Array {
  public:

    friend bool operator == (Array<T> const& a,
                             Array<T> const& b) {
        return ArraysAreEqual(a, b);
    }
};
```

Suppose this version of Array is instantiated for type float. The friend operator function is then declared as a result of that instantiation, but note that this function itself is not an instantiation of a function template. It is a normal nontemplate function that gets injected in the global scope as a side effect of the instantiation process. Because it is a nontemplate function, it could be overloaded with other declarations of operator == even before overloading of function templates was added to the language. Barton and Nackman referred to this as restricted template expansion because it avoided the use of a template operator==(T, T) that applied to all types T (in other words, unrestricted expansion).

Because operator == (Array<T> const&, Array<T> const&) is defined inside a class definition, it is implicitly considered to be an inline function, and we therefore decided to delegate the implementation to a function template ArraysAreEqual, which doesn't need to be inline and is unlikely to conflict with another template of the same name.

The Barton-Nackman trick is no longer needed for its original purpose, but it is interesting to study it because it

11.8 Afternotes

Template argument deduction for function templates was part of the original C++ design. In fact, the alternative provided by explicit template arguments did not become part of C++ until many years later.

Friend name injection was considered harmful by many C++ language experts because it made the validity of programs more sensitive to the ordering of instantiations. Bill Gibbons (who at the time was working on the Taligent compiler) was among the most vocal supporters of addressing the problem, because eliminating instantiation order dependencies enabled new and interesting C++ development environments (on which Taligent was rumored to be working). However, the Barton-Nackman trick required a form of friend name injection, and it is this particular technique that caused it to remain in the language in its current (weakened) form.

Interestingly, many people have heard of the "Barton-Nackman trick," but few correctly associate it with the technique described earlier. As a result, you may find many other techniques involving friends and templates being referred to incorrectly as the "Barton-Nackman trick" (for example, see Section 16.5 on page 299).

Chapter 12. Specialization and Overloading

So far we have studied how C++ templates allow a generic definition to be expanded into a family of related classes or functions. Although this is a powerful mechanism, there are many situations in which the generic form of an operation is far from optimal for a specific substitution of template parameters.

C++ is somewhat unique among other popular programming languages with support for generic programming because it has a rich set of features that enable the transparent replacement of a generic definition by a more specialized facility. In this chapter we study the two C++ language mechanisms that allow pragmatic deviations from pure genericness: template specialization and overloading of function templates.

12.1 When "Generic Code" Doesn't Quite Cut It

Consider the following example:

```
template<typename T>
class Array {
  private:
    T* data;

  public:
    Array(Array<T> const&);
    Array<T>& operator = (Array<T> const&);

    void exchange_with (Array<T>* b) {
        T* tmp = data;
        data = b->data;
        b->data = tmp;
    }
    T& operator[] (size_t k) {
        return data[k];
    }

};

template<typename T> inline
void exchange (T* a, T* b)
{
     T tmp(*a);
     *a = *b;
     *b = tmp;
}
```

For simple types, the generic implementation of exchange() works well. However, for types with expensive copy operations, the generic implementation may be much more expensive—both in terms of machine cycles and in terms of memory usage—than an implementation that is tailored to the particular, given structure. In our example, the generic implementation requires one call to the copy constructor of Array<T> and two calls to its copy-assignment operator. For large data structures these copies can often involve copying relatively large amounts of memory. However, the functionality of exchange() could presumably often be replaced just by swapping the internal data pointers, as is done in the member function exchange_with().

12.1.1 Transparent Customization

In our previous example, the member function exchange_with() provides an efficient alternative to the generic exchange() function, but the need to use a different function is inconvenient in several ways:

1. Users of the Array class have to remember an extra interface and must be careful to use it when possible.

2. Generic algorithms can generally not discriminate between various possibilities. For example:

```
template<typename T>
void generic_algorithm(T* x, T* y)
{

   exchange(x, y);  // How do we select the right algorithm?

```

12.2 Overloading Function Templates

In the previous section we saw that two function templates with the same name can coexist, even though they may be instantiated so that both have identical parameter types. Here is another simple example of this:

```
// details/funcoverload.hpp

template<typename T>
int f(T)
{
    return 1;
}

template<typename T>
int f(T*)
{
    return 2;
}
```

When T is substituted by int* in the first template, a function is obtained that has exactly the same parameter (and return) types as the one obtained by substituting int for T in the second template. Not only can these templates coexist, their respective instantiations can coexist even if they have identical parameter and return types.

The following demonstrates how two such generated functions can be called using explicit template argument syntax (assuming the previous template declarations):

```
// details/funcoverload.cpp

#include <iostream>
#include "funcoverload.hpp"

int main()
{
    std::cout << f<int*>((int*)0) << std::endl;
    std::cout << f<int>((int*)0)  << std::endl;
}
```

This program has the following output:

```
1
2
```

To clarify this, let's analyze the call f<int*>((int*)0) in detail. [1] The syntax f<int*> indicates that we want to substitute the first template parameter of the template f with int* without relying on template argument deduction. In this case there is more than one template f, and therefore an overload set is created containing two functions generated from templates: f<int*>(int*) (generated from the first template) and f<int*>(int**) (generated from the second template). The argument to the call (int*)0 has type int*. This matches only the function generated from the first template, and hence that is the function that ends up being called.

[1] Note that the expression 0 is an integer and not a null pointer constant. It becomes a null pointer constant after a special implicit conversion, but this conversion is not considered during template argument deduction.

A similar analysis can be written for the second call.

12.2.1 Signatures

12.3 Explicit Specialization

The ability to overload function templates, combined with the partial ordering rules to select the "best" matching function template, allows us to add more specialized templates to a generic implementation to tune code transparently for greater efficiency. However, class templates cannot be overloaded. Instead, another mechanism was chosen to enable transparent customization of class templates: explicit specialization. The standard term explicit specialization refers to a language feature that we call full specialization instead. It provides an implementation for a template with template parameters that are fully substituted: No template parameters remain. Class templates and function templates can be fully specialized. So can members of class templates that may be defined outside the body of a class definition (i.e., member functions, nested classes, and static data members).

In a later section, we will describe partial specialization. This is similar to full specialization, but instead of fully substituting the template parameters, some parameterization is left in the alternative implementation of a template. Full specializations and partial specializations are both equally "explicit" in our source code, which is why we avoid the term explicit specialization in our discussion. Neither full nor partial specialization introduces a totally new template or template instance. Instead, these constructs provide alternative definitions for instances that are already implicitly declared in the generic (or unspecialized) template. This is a relatively important conceptual observation, and it is a key difference with overloaded templates.

12.3.1 Full Class Template Specialization

A full specialization is introduced with a sequence of three tokens: template, <, and >. [3] In addition, the class name declarator is followed by the template arguments for which the specialization is declared. The following example illustrates this:

[3] The same prefix is also needed to declare full function template specializations. Earlier designs of the C++ language did not include this prefix, but the addition of member templates required additional syntax to disambiguate complex specialization cases.

```
template<typename T>
class S {
  public:
    void info() {
        std::cout << "generic (S<T>::info())\n";
    }
};
template<>
class S<void> {
  public:
    void msg() {
        std::cout << "fully specialized (S<void>::msg())\n";
    }
};
```

Note how the implementation of the full specialization does not need to be related in any way to the generic definition: This allows us to have member functions of different names (info versus msg). The connection is solely determined by the name of the class template.

The list of specified template arguments must correspond to the list of template parameters. For example, it is not valid to specify a nontype value for a template type parameter. However, template arguments for parameters with default template arguments are optional:

```
template<typename T>
```

12.4 Partial Class Template Specialization

Full template specialization is often useful, but sometimes it is natural to want to specialize a class template for a family of template arguments rather than just one specific set of template arguments. For example, let's assume we have a class template implementing a linked list:

```
template<typename T>
class List {          // (1)
  public:

   void append(T const&);
   inline size_t length() const;

};
```

A large project making use of this template may instantiate its members for many types. For member functions that are not expanded inline (say, List<T>::append()), this may cause noticeable growth in the object code. However, we may know that from a low-level point of view, the code for List<int*>::append() and List<void*>::append() is the same. In other words, we'd like to specify that all Lists of pointers share an implementation. Although this cannot be expressed in C++, we can achieve something quite close by specifying that all Lists of pointers should be instantiated from a different template definition:

```
template<typename T>
class List<T*> { // (2)
  private:
    List<void*> impl;

  public:

    void append(T* p) {
        impl.append(p);
    }
    size_t length() const {
        return impl.length();
    }

};
```

In this context, the original template at point (1) is called the primary template, and the latter definition is called a partial specialization (because the template arguments for which this template definition must be used have been only partially specified). The syntax that characterizes a partial specialization is the combination of a template parameter list declaration (template<...>) and a set of explicitly specified template arguments on the name of the class template (<T*> in our example).

Our code contains a problem because List<void*> recursively contains a member of that same List<void*> type. To break the cycle, we can precede the previous partial specialization with a full specialization:

```
template<>
class List<void*> {   // (3)

    void append (void* p);
    inline size_t length() const;

};
```

This works because matching full specializations are preferred over partial specializations. As a result, all member functions of Lists of pointers are forwarded (through easily inlineable functions) to the implementation of List<void*>. This is an effective way to combat so-called code bloat (of which C++ templates are often accused).

12.5 Afternotes

Full template specialization was part of the C++ template mechanism from the start. Function template overloading and class template partial specialization, on other hand, came much later. The HP aC++ compiler was the first to implement function template overloading, and EDG's C++ front end was the first to implement class template partial specialization. The partial ordering principles described in this chapter were originally invented by Steve Adamczyk and John Spicer (who are both of EDG).

The ability of template specializations to terminate an otherwise infinitely recursive template definition (such as the List<T*> example presented in Section 12.4 on page 200) was known for a long time. However, Erwin Unruh was perhaps the first to note that this could lead to the interesting notion of template metaprogramming: Using the template instantiation mechanism to perform nontrivial computations at compile time. We devote Chapter 17 to this topic.

You may legitimately wonder why only class templates can be partially specialized. The reasons are mostly historical. It is probably possible to define the same mechanism for function templates (see Chapter 13). In some ways the effect of overloading function templates is similar, but there are also some subtle differences. These differences are mostly related to the fact that only the primary template needs to be looked up when a use is encountered. The specializations are considered only afterward, to determine which implementation should be used. In contrast, all overloaded function templates must be brought into an overload set by looking them up, and they may come from different namespaces or classes. This increases the likelihood of unintentionally overloading a template name somewhat.

Conversely, it is also imaginable to allow a form of overloading of class templates. Here is an example:

```
// invalid overloading of class templates
template<typename T1, typename T2> class Pair;
template<int N1, int N2> class Pair;
```

However, there doesn't seem to be a pressing need for such a mechanism.

Chapter 13. Future Directions

C++ templates evolved considerably from their initial design in 1988 until the standardization of C++ in 1998 (the technical work was completed in November 1997). After that, the language definition was stable for several years, but during that time various new needs have arisen in the area of C++ templates. Some of these needs are simply a consequence of a desire for more consistency or orthogonality in the language. For example, why wouldn't default template arguments be allowed on function templates when they are allowed on class templates? Other extensions are prompted by increasingly sophisticated template programming idioms that often stretch the abilities of existing compilers.

In what follows we describe some extensions that have come up more than once among C++ language and compiler designers. Often such extensions were prompted by the designers of various advanced C++ libraries (including the C++ standard library). There is no guarantee that any of these will ever be part of standard C++. On the other hand, some of these are already provided as extensions by certain C++ implementations.

13.1 The Angle Bracket Hack

Among the most common surprises for beginning template programmers is the necessity to add some blank space between consecutive closing angle brackets. For example:

```
#include <list>
#include <vector>

typedef std::vector<std::list<int> > LineTable;   // OK

typedef std::vector<std::list<int>>  OtherTable;  // SYNTAX ERROR
```

The second typedef declaration is an error because the two closing angle brackets with no intervening blank space constitute a "right shift" (>>) operator, which makes no sense at that location in the source.

Yet detecting such an error and silently treating the >> operator as two closing angle brackets (a feature sometimes referred to as the angle bracket hack) is relatively simple compared with many of the other capabilities of C++ source code parsers. Indeed, many compilers are already able to recognize such situations and will accept the code with a warning.

Hence, it is likely that a future version of C++ will require the declaration of OtherTable (in the previous example) to be valid. Nevertheless, we should note that there are some subtle corners to the angle bracket hack. Indeed, there are situations when the >> operator is a valid token within a template argument list. The following example illustrates this:

```
template<int N> class Buf;

template<typename T> void strange() {}
template<int N> void strange() {}

int main()
{
    strange<Buf<16>>2> >();  // the >> token is not an error
}
```

A somewhat related issue deals with the accidental use of the digraph <:, which is equivalent to the bracket [(see Section 9.3.1 on page 129). Consider the following code extract:

```
template<typename T> class List;
class Marker;

List<::Marker>* markers; // ERROR
```

The last line of this example is treated as List[:Marker>* markers;, which makes no sense at all. However, a compiler could conceivably take into account that a template such as List can never validly be followed by a left bracket and disable the recognition of the corresponding digraph in that context.

13.2 Relaxed typename Rules

Some programmers and language designers find the rules for the use of typename (see Section 5.1 on page 43 and Section 9.3.2 on page 130) too strict. For example, in the following code, the occurrence of typename in typename Array<T>::ElementT is mandatory, but the one in typename Array<int>::ElementT is prohibited (an error):

```
template <typename T>
class Array {
  public:
    typedef T ElementT;

};

template <typename T>
void clear (typename Array<T>::ElementT& p);      // OK

template<>
void clear (typename Array<int>::ElementT& p);    // ERROR
```

Examples such as this can be surprising, and because it is not difficult for a C++ compiler implementation simply to ignore the extra keyword, the language designers are considering allowing the typename keyword in front of any qualified typename that is not already elaborated with one of the keywords struct, class, union, or enum. Such a decision would probably also clarify when the .template, ->template, and ::template constructs (see Section 9.3.3 on page 132) are permissible.

Ignoring extraneous uses of typename and template is relatively straightforward from an implementer's point of view. Interestingly, there are also situations when the language currently requires these keywords but when an implementation could do without them. For example, in the previous function template clear(), a compiler can know that the name Array<T>::ElementT cannot be anything but a type name (no expressions are allowed at that point), and therefore the use of typename could be made optional in that situation. The C++ standardization committee is therefore also examining changes that would reduce the number of situations when typename and template are required.

13.3 Default Function Template Arguments

When templates were originally added to the C++ language, explicit function template arguments were not a valid construct. Function template arguments always had to be deducible from the call expression. As a result, there seemed to be no compelling reason to allow default function template arguments because the default would always be overridden by the deduced value.

Since then, however, it is possible to specify explicitly function template arguments that cannot be deduced. Hence, it would be entirely natural to specify default values for those nondeducible template arguments. Consider the following example:

```
template <typename T1, typename T2 = int>
T2 count (T1 const& x);
class MyInt {

};

void test (Container const& c)
{
    int i = count(c);
    MyInt = count<MyInt>(c);
    assert(MyInt == i);
}
```

In this example, we have respected the constraint that if a template parameter has a default argument value, then each parameter after that must have a default template argument too. This constraint is needed for class templates; otherwise, there would be no way to specify trailing arguments in the general case. The following erroneous code illustrates this:

```
template <typename T1 = int, typename T2>
class Bad;

Bad<int>* b;  // Is the given int a substitution for T1 or for T2?
```

For function templates, however, the trailing arguments may be deduced. Hence, there is no technical difficulty in rewriting our example as follows:

```
template <typename T1 = int, typename T2>
T1 count (T2 const& x);

void test (Container const& c)
{
    int i = count(c);
    MyInt = count<MyInt>(c);
    assert(MyInt == i);
}
```

At the time of this writing the C++ standardization committee is considering extending function templates in this direction.

In hindsight, programmers have also noted uses that do not involve explicit template arguments. For example:

```
template <typename T = double>
void f(T const& = T());
int main()
{
    f(1);           // OK: deduce T = int
```

13.4 String Literal and Floating-Point Template Arguments

Among the restrictions on nontype template arguments, perhaps the most surprising to beginning and advanced template writers alike is the inability to provide a string literal as a template argument.

The following example seems intuitive enough:

```
template <char const* msg>
class Diagnoser {
  public:
    void print();
};

int main()
{
    Diagnoser<"Surprise!">().print();
}
```

However, there are some potential problems. In standard C++, two instances of Diagnoser are the same type if and only if they have the same arguments. In this case the argument is a pointer value—in other words, an address. However, two identical string literals appearing in different source locations are not required to have the same address. We could thus find ourselves in the awkward situation that Diagnoser<"X"> and Diagnoser<"X"> are in fact two different and incompatible types! (Note that the type of "X" is char const[2], but it decays to char const* when passed as a template argument.)

Because of these (and related) considerations, the C++ standard prohibits string literals as arguments to templates. However, some implementations do offer the facility as an extension. They enable this by using the actual string literal contents in the internal representation of the template instance. Although this is clearly feasible, some C++ language commentators feel that a nontype template parameter that can be substituted by a string literal value should be declared differently from one that can be substituted by an address. At the time of this writing, however, no such declaration syntax has received overwhelming support.

We should also note an additional technical wrinkle in this issue. Consider the following template declarations, and let's assume that the language has been extended to accept string literals as template arguments in this case:

```
template <char const* str>
class Bracket {
  public:
    static char const* address() const;
    static char const* bytes() const;
};

template <char const* str>
char const* Bracket<T>::address() const
{
    return str;
}

template <char const* str>
char const* Bracket<T>::bytes() const
{
    return str;
}
```

In the previous code, the two member functions are identical except for their names—a situation that is not that uncommon. Imagine that an implementation would instantiate Bracket<"X"> using a process much like macro

13.5 Relaxed Matching of Template Template Parameters

A template used to substitute a template template parameter must match that parameter's list of template parameters exactly. This can sometimes have surprising consequences, as shown in the following example:

```
#include <list>
    // declares:
    //  namespace std {
    //      template <typename T,
    //                typename Allocator = allocator<T> >
    //      class list;
    // }

template<typename T1,
         typename T2,
         template<typename> class Container>
                   // Container expects templates with only one parameter
class Relation {
  public:

  private:
    Container<T1> dom1;
    Container<T2> dom2;
};

int main()
{
    Relation<int, double, std::list> rel;
        // ERROR: std::list has more than one template parameter

}
```

This program is invalid because our template template parameter Container expects a template taking one parameter, whereas std::list has an allocator parameter in addition to its parameter that determines the element type.

However, because std::list has a default template argument for its allocator parameter, it would be possible to specify that Container matches std::list and that each instantiation of Container uses the default template argument of std::list (see Section 8.3.4 on page 112).

An argument in favor of the status quo (no match) is that the same rule applies to matching function types. However, in this case the default arguments cannot always be determined because the value of a function pointer usually isn't fixed until run time. In contrast, there are no "template pointers," and all the required information can be available at compile time.

Some C++ compilers already offer the relaxed matching rule as an extension. This issue is also related to the issue of typedef templates (discussed in the next section). Indeed, consider replacing the definition of main() in our previous example with:

```
template <typename T>
typedef list<T> MyList;

int main()
{
    Relation<int, double, MyList> rel;
}
```

The typedef template introduces a new template that now exactly matches Container with respect to its parameter

13.6 Typedef Templates

Class templates are often combined in relatively sophisticated ways to obtain other parameterized types. When such parameterized types appear repeatedly in source code, it is natural to want a shortcut for them, just as typedefs provide a shortcut for unparameterized types.

Therefore, C++ language designers are considering a construct that may look as follows:

```
template <typename T>
typedef vector<list<T> > Table;
```

After this declaration, Table would be a new template that can be instantiated to become a concrete type definition. Such a template is called a typedef template (as opposed to a class template or a function template). For example:

```
Table<int> t;        // t has type vector<list<int> >
```

Currently, the lack of typedef templates is worked around by using member typedefs of class templates. For our example we might use:

```
template <typename T>
class Table {
  public:
    typedef vector<list<T> > Type;
};

Table<int>::Type t;  // t has type vector<list<int> >
```

Because typedef templates are to be full-fledged templates, they could be specialized much like class templates:

```
// primary typedef template:
template<typename T> typedef T Opaque;

// partial specialization:
template<typename T> typedef void* Opaque<T*>;

// full specialization:
template<> typedef bool Opaque<void>;
```

Typedef templates are not entirely straightforward. For example, it is not clear how they would participate in the deduction process:

```
void candidate(long);

template<typename T> typedef T DT;

template<typename T> void candidate(DT<T>);

int main()
{
    candidate(42);  // which candidate() should be called?
}
```

It is not clear that deduction should succeed in this case. Certainly, deduction is not possible with arbitrary typedef patterns.

13.7 Partial Specialization of Function Templates

In Chapter 12 we discussed how class templates can be partially specialized, whereas function templates are simply overloaded. The two mechanisms are somewhat different.

Partial specialization doesn't introduce a completely new template: It is an extension of an existing template (the primary template). When a class template is looked up, only primary templates are considered at first. If, after the selection of a primary template, it turns out that there is a partial specialization of that template with a template argument pattern that matches that of the instantiation, its definition (in other words, its body) is instantiated instead of the definition of the primary template. (Full template specializations work exactly the same way.)

In contrast, overloaded function templates are separate templates that are completely independent of one another. When selecting which template to instantiate, all the overloaded templates are considered together, and overload resolution attempts to choose one as the best fit. At first this might seem like an adequate alternative, but in practice there are a number of limitations:

- It is possible to specialize member templates of a class without changing the definition of that class. However, adding an overloaded member does require a change in the definition of a class. In many cases this is not an option because we may not own the rights to do so. Furthermore, the C++ standard does not currently allow us to add new templates to the std namespace, but it does allow us to specialize templates from that namespace.

- To overload function templates, their function parameters must differ in some material way. Consider a function template R convert(T const&) where R and T are template parameters. We may very well want to specialize this template for R = void, but this cannot be done using overloading.

- Code that is valid for a nonoverloaded function may no longer be valid when the function is overloaded. Specifically, given two function templates f(T) and g(T) (where T is a template parameter), the expression g(&f<int>) is valid only if f is not overloaded (otherwise, there is no way to decide which f is meant).

- Friend declarations refer to a specific function template or an instantiation of a specific function template. An overloaded version of a function template would not automatically have the privileges granted to the original template.

Together, this list forms a compelling argument in support of a partial specialization construct for function templates.

A natural syntax for partially specializing function templates is the generalization of the class template notation:

```
template <typename T>
T const& max (T const&, T const&);        // primary template

template <typename T>
T* const& max <T*>(T* const&, T* const&); // partial specialization
```

13.8 The typeof Operator

When writing templates, it is often useful to be able to express the type of a template-dependent expression. Perhaps the poster child of this situation is the declaration of an arithmetic operator for a numeric array template in which the element types of the operands are mixed. The following example should make this clear:

```
template <typename T1, typename T2>
Array<???> operator+ (Array<T1> const& x, Array<T2> const& y);
```

Presumably, this operator is to produce an array of elements that are the result of adding corresponding elements in the arrays x and y. The type of a resulting element is thus the type of x[0]+y[0]. Unfortunately, C++ does not offer a reliable way to express this type in terms of T1 and T2.

Some compilers provide the typeof operator as an extension that addresses this issue. It is reminiscent of the sizeof operator in that it can take an expression and produce a compile-time entity from it, but in this case the compile-time entity can act as the name of a type. In our previous example this allows us to write:

```
template <typename T1, typename T2>
Array<typeof(T1()+T2())> operator+ (Array<T1> const& x,
                                    Array<T2> const& y);
```

This is nice, but not ideal. Indeed, it assumes that the given types can be default-initialized. We can work around this assumption by introducing a helper template as follows:

```
template <typename T>
T makeT();  // no definition needed

template <typename T1, typename T2>
Array<typeof(makeT<T1>()+makeT<T2>())>
  operator+ (Array<T1> const& x,
             Array<T2> const& y);
```

We really would prefer to use x and y in the typeof argument, but we cannot do so because they have not been declared at the point of the typeof construct. A radical solution to this problem is to introduce an alternative function declaration syntax that places the return type after the parameter types:

```
// operator function template:
template <typename T1, typename T2>
operator+ (Array<T1> const& x, Array<T2> const& y)
  -> Array<typeof(x+y)>;

// regular function template:
template <typename T1, typename T2>
function exp(Array<T1> const& x, Array<T2> const& y)
  -> Array<typeof(exp(x, y))>
```

As the example illustrates, a new keyword (here, function) is necessary to enable the new syntax for nonoperator functions (for operator functions, the operator keyword is sufficient to guide the parsing process).

Note that typeof must be a compile-time operator. In particular, typeof will not take into account covariant return types, as the following example shows:

```
class Base {
  public:
    virtual Base clone();
};
```

13.9 Named Template Arguments

Section 16.1 on page 285 describes a technique that allows us to provide a nondefault template argument for a specific parameter without having to specify other template arguments for which a default value is available. Although it is an interesting technique, it is also clear that it results in a fair amount of work for a relatively simple effect. Hence, providing a language mechanism to name template arguments is a natural thought.

We should note at this point, that a similar extension (sometimes called keyword arguments) was proposed earlier in the C++ standardization process by Roland Hartinger (see Section 6.5.1 of [StroustrupDnE]). Although technically sound, the proposal was ultimately not accepted into the language for various reasons. At this point there is no reason to believe named template arguments will ever make it into the language.

However, for the sake of completeness, we mention one syntactic idea that has floated among certain designers:

```
template<typename T,
         Move: typename M = defaultMove<T>,
         Copy: typename C = defaultCopy<T>,
         Swap: typename S = defaultSwap<T>,
         Init: typename I = defaultInit<T>,
         Kill: typename K = defaultKill<T> >
class Mutator {

};

void test(MatrixList ml)
{
   mySort (ml, Mutator <Matrix, Swap: matrixSwap>);
}
```

Note how the argument name (preceding a colon) is distinct from the parameter name. This allows us to keep the practice of using short names for the parameters used in the implementation while having a self-documenting name for the argument names. Because this can be overly verbose for some programming styles, one can also imagine the ability to omit the argument name if it is identical to the parameter name:

```
template<typename T,
         : typename Move = defaultMove<T>,
         : typename Copy = defaultCopy<T>,
         : typename Swap = defaultSwap<T>,
         : typename Init = defaultInit<T>,
         : typename Kill = defaultKill<T> >
class Mutator {

};
```

13.10 Static Properties

In Chapter 15 and Chapter 19 we discuss various ways to categorize types "at compile time." Such traits are useful in selecting specializations of templates based on the static properties of the type. (See, for example, our CSMtraits class in Section 15.3.2 on page 279, which attempts to select optimal or near-optimal policies to copy, swap, or move elements of the argument type.)

Some language designers have observed that if such "specialization selections" are commonplace, they shouldn't require elaborate user-defined code if all that is sought is a property that the implementation knows internally anyway. The language could instead provide a number of built-in type traits. The following could be a valid complete C++ program with such an extension:

```
#include <iostream>

int main()
{
    std::cout << std::type<int>::is_bit_copyable << '\n';
    std::cout << std::type<int>::is_union << '\n';
}
```

Although a separate syntax could be developed for such a construct, fitting it in a user-definable syntax may allow for a more smooth transition from the current language to a language that would include such facilities. However, some of the static properties that a C++ compiler can easily provide may not be obtainable using traditional traits techniques (for example, determining whether a type is a union), which is an argument in favor of making this a language element. Another argument is that it can significantly reduce the amount of memory and machine cycles required by a compiler to translate programs that rely on such properties.

13.11 Custom Instantiation Diagnostics

Many templates put some implicit requirements on their parameters. When the arguments of an instantiation of such a template do not fulfill the requirements, either a generic error is issued or the generated instantiation does not function correctly. In early C++ compilers, the generic errors produced during template instantiations were often exceedingly opaque (see page 75 for an example). In more recent compilers, the error messages are sufficiently clear for an experienced programmer to track down a problem quickly, but there is still a desire to improve the situation. Consider the following artificial example (meant to illustrate what happens in real template libraries):

```
template <typename T>
void clear (T const& p)
{
    *p = 0;  // assumes T is a pointerlike type
}

template <typename T>
void core (T const& p)
{
    clear(p);
}

template <typename T>
void middle (typename T::Index p)
{
    core(p);
}

template <typename T>
void shell (T const& env)
{
    typename T::Index i;
    middle<T>(i);
}

class Client {
  public:
    typedef int Index;

};

Client main_client;

int main()
{
    shell(main_client);
}
```

This example illustrates the typical layering of software development: High-level function templates like shell() rely on components like middle(), which themselves make use of basic facilities like core(). When we instantiate shell(), all the layers below it also need to be instantiated. In this example, a problem is revealed in the deepest layer: core() is instantiated with type int (from the use of Client::Index in middle()) and attempts to dereference a value of that type, which is an error. A good generic diagnostic will include a trace of all the layers that led to the problems, but this amount of information may be unwieldy.

An alternative that has often been proposed is to insert a device in the highest level template to inhibit deeper instantiation if known requirements from lower levels are not satisfied. Various attempts have been made to implement such devices in terms of existing C++ constructs (for example, see [BCCL]), but they are not always effective. Hence, it is not surprising that language extensions have been proposed to address the issue. Such an extension could clearly build on top of the static properties facilities discussed earlier. For example, we can envision

13.12 Overloaded Class Templates

It is entirely possible to imagine that class templates could be overloaded on their template parameters. For example, one can imagine the following:

```
template <typename T1>
class Tuple {
   // singleton

};

template <typename T1, typename T2>
class Tuple {
   // pair

};

template <typename T1, typename T2, typename T3>
class Tuple {
   // three-element tuple

};
```

In the next section we discuss an application of such overloading.

The overloading isn't necessarily restricted to the number of template parameters (such overloading could be emulated using partial specialization as is done for FunctionPtr in Chapter 22). The kind of parameters can be varied too:

```
template <typename T1, typename T2>
class Pair {
   // pair of fields

};

template <int I1, int I2>
class Pair {
   // pair of constant integer values

};
```

Although this idea has been discussed informally by some language designers, it has not yet been formally presented to the C++ standardization committee.

13.13 List Parameters

A need that shows up sometimes is the ability to pass a list of types as a single template argument. Usually, this list is meant for one of two purposes: declaring a function with a parameterized number of parameters or defining a type structure with a parameterized list of members.

For example, we may want to define a template that computes the maximum of an arbitrary list of values. A potential declaration syntax uses the ellipsis token to denote that the last template parameter is meant to match an arbitrary number of arguments:

```
#include <iostream>

template <typename T, ... list>
T const& max (T const&, T const&, list const&);

int main()
{
    std::cout << max(1, 2, 3, 4) << std::endl;
}
```

Various possibilities can be thought of to implement such a template. Here is one that doesn't require new keywords but adds a rule to function template overloading to prefer a function template without a list parameter:

```
template <typename T> inline
T const& max (T const& a, T const& b)
{
    // our usual binary maximum:
    return a<b?b:a;
}

template <typename T, ... list> inline
T const& max (T const& a, T const& b, list const& x)
{
   return max (a, max(b,x));
}
```

Let's go through the steps that would make this work for the call max(1, 2, 3, 4). Because there are four arguments, the binary max() function doesn't match, but the second one does match with T = int and list = int, int. This causes us to call the binary function template max() with the first argument equal to 1 and the second argument equal to the evaluation of max(2, 3, 4). Again, the binary operation doesn't match, and we call the list parameter version with T = int and list = int. This time the subexpression max(b,x) expands to max(3,4), and the recursion ends by selecting the binary template.

This works fairly well thanks to the ability of overloading function templates. There is more to it than our discussion, of course. For example, we'd have to specify precisely what list const& means in this context.

Sometimes, it may be desirable to refer to particular elements or subsets of the list. For example, we could use the subscript brackets for this purpose. The following example shows how we could construct a metaprogram to count the elements in a list using this technique:

```
template <typename T>
class ListProps {
  public:
    enum { length = 1 };
};
```

13.14 Layout Control

A fairly common template programming challenge is to declare an array of bytes that will be sufficiently large (but not excessively so) to hold an object of an as yet unknown type T—in other words, a template parameter. One application of this is the so-called discriminated unions (also called variant types or tagged unions):

```
template <... list>
class D_Union {
  public:
    enum { n_bytes; };
    char bytes[n_bytes];  // will eventually hold one of various types
                          // described by the template arguments

};
```

The constant n_bytes cannot always be set to sizeof(T) because T may have more strict alignment requirements than the bytes buffer. Various heuristics exist to take this alignment into account, but they are often complicated or make somewhat arbitrary assumptions.

For such an application, what is really desired is the ability to express the alignment requirement of a type as a constant expression and, conversely, the ability to impose an alignment on a type, a field, or a variable. Many C and C++ compilers already support an __alignof__ operator, which returns the alignment of a given type or expression. This is almost identical to the sizeof operator except that the alignment is returned instead of the size of the given type. Many compilers also provide #pragma directives or similar devices to set the alignment of an entity. A possible approach may be to introduce an alignof keyword that can be used both in expressions (to obtain the alignment) and in declarations (to set the alignment).

```
template <typename T>
class Alignment {
  public:
    enum { max = alignof(T) };
};

template <... list>
class Alignment {
  public:
    enum { max = alignof(list[0]) > Alignment<list[1 ...]>::max
                  ? alignof(list[0])
                  : Alignment<list[1 ...]>::max; }
};

// a set of Size templates could similarly be designed
// to determine the largest size among a given list of types

template <... list>
class Variant {
  public:
    char buffer[Size<list>::max] alignof(Alignment<list>::max);

};
```

13.15 Initializer Deduction

It is often said that "programmers are lazy," and sometimes this refers to our desire to keep programmatic notation compact. Consider, in that respect, the following declaration:

```
std::map<std::string, std::list<int> >* dict
= new std::map<std::string, std::list<int> >;
```

This is verbose, and in practice we would (and most likely should) introduce a typedef synonym for the type. However, there is something redundant in this declaration: We specify the type of dict, but it is also implicit in the type of its initializer. Wouldn't it be considerably more elegant to be able to write an equivalent declaration with only one type specification? For example:

```
dcl dict = new std::map<std::string, std::list<int> >;
```

In this last declaration, the type of a variable is deduced from the type of the initializer. A keyword (dcl in the example, but var, let, and even auto have been proposed as alternatives) is needed to make the declaration distinguishable from an ordinary assignment.

So far, this isn't a template-only issue. In fact, it appears such a construct was accepted by a very early version of the Cfront compiler (in 1982, before templates came on the scene). However, it is the verbosity of many template-based types that increases the demand for this feature.

One could also imagine partial deduction in which only the arguments of a template must be deduced:

```
std::list<> index = create_index();
```

Another variant of this is to deduce the template arguments from the constructor arguments. For example:

```
template <typename T>
class Complex {
  public:
    Complex(T const& re, T const& im);

};

Complex<> z(1.0, 3.0);  // deduces T = double
```

Precise specifications for this kind of deduction are made more complicated by the possibility of overloaded constructors, including constructor templates. Suppose, for example, that our Complex template contains a constructor template in addition to a normal copy constructor:

```
template <typename T>
class Complex {
  public:
    Complex(Complex<T> const&);

    template <typename T2> Complex(Complex<T2> const&);

};

Complex<double> j(0.0, 1.0);
Complex<> z = j;  // Which constructor was intended?
```

In the latter initialization, it is probable that the regular copy constructor was intended; hence z should have the same type as j. However, making it an implicit rule to ignore constructor templates may be overly bold.

13.16 Function Expressions

Chapter 22 illustrates that it is often convenient to pass small functions (or functors) as parameters to other functions. We also mention in Chapter 17 that expression template techniques can be used to build small functors concisely without the overhead of explicit declarations (see Section 18.3 on page 340).

For example, we may want to call a particular member function on each element of a standard vector to initialize it:

```
class BigValue {
  public:
    void init();

};

class Init {
  public:
    void operator() (BigValue& v) const {
        v.init();
    }
};
void compute (std::vector<BigValue>& vec)
{
    std::for_each (vec.begin(), vec.end(),
                   Init());

}
```

The need to define a separate class Init for this purpose is unwieldy. Instead, we can imagine that we may write (unnamed) function bodies as part of an expression:

```
class BigValue {
  public:
    void init();

};

void compute (std::vector<BigValue>& vec)
{
    std::for_each (vec.begin(), vec.end(),
                   $(BigValue&) { $1.init(); });

}
```

The idea here is that we can introduce a function expression with a special token $ followed by parameter types in parentheses and a brace-enclosed body. Within such a construct, we can refer to the parameters with the special notation $n, where n is a constant indicating the number of the parameter.

This form is closely related to so-called lambda expressions (or lambda functions) and closures in other programming languages. However, other solutions are possible. For example, a solution might use anonymous inner classes, as seen in Java:

```
class BigValue {
  public:
    void init();

};

void compute (std::vector<BigValue>& vec)
```

13.17 Afternotes

It seems perhaps premature to talk about extending the language when C++ compilers are only barely becoming mostly compliant to the 1998 standard (C++98). However, it is in part because this compliance is being achieved that we (the C++ programmers community) are gaining insight into the true limitations of C++ (and templates in particular).

To meet the new needs of C++ programmers, the C++ standards committee (often referred to as ISO WG21/ANSI J16, or just WG21/J16) started examining a road to a new standard: C++0x. After a preliminary presentation at its April 2001 meeting in Copenhagen, WG21/J16 started examining concrete library extension proposals.

Indeed, the intention is to attempt as much as possible to confine extensions to the C++ standard library. However, it is well understood that some of these extensions may require work in the core language. We expect that many of these required modifications will relate to C++ templates, just as the introduction of STL in the C++ standard library stimulated template technology in the 1990s.

Finally, C++0x is also expected to address some "embarrassments" in C++98. It is hoped that doing so will improve the accessibility of C++. Some of the extensions in that direction were discussed in this chapter.

Part III: Templates and Design

Programs are generally constructed using designs that map relatively well on the mechanisms offered by a chosen programming language. Because templates are a whole new language mechanism, it is not surprising to find that they call for new design elements. We explore these elements in this part of the book.

Templates are different from more traditional language constructs in that they allow us to parameterize the types and constants of our code. When combined with (1) partial specialization and (2) recursive instantiation, this leads to a surprising amount of expressive power. In the following chapters, this is illustrated by a large number of design techniques:

- Generic programming
- Traits
- Policy classes
- Metaprogramming
- Expression templates

Our presentation aims not only at listing the various known design elements, but also at conveying the principles that inspire such designs so that new techniques may be created.

Chapter 14. The Polymorphic Power of Templates

Polymorphism is the ability to associate different specific behaviors with a single generic notation. [1] Polymorphism is also a cornerstone of the object-oriented programming paradigm, which in C++ is supported mainly through class inheritance and virtual functions. Because these mechanism are (at least in part) handled at run time, we talk about dynamic polymorphism. This is usually what is thought of when talking about plain polymorphism in C++. However, templates also allow us to associate different specific behaviors with a single generic notation, but this association is generally handled at compile time, which we refer to as static polymorphism. In this chapter we review the two forms of polymorphism and discuss which form is appropriate in which situations.

[1] Polymorphism literally refers to the condition of having many forms or shapes (from the Greek polumorphos).

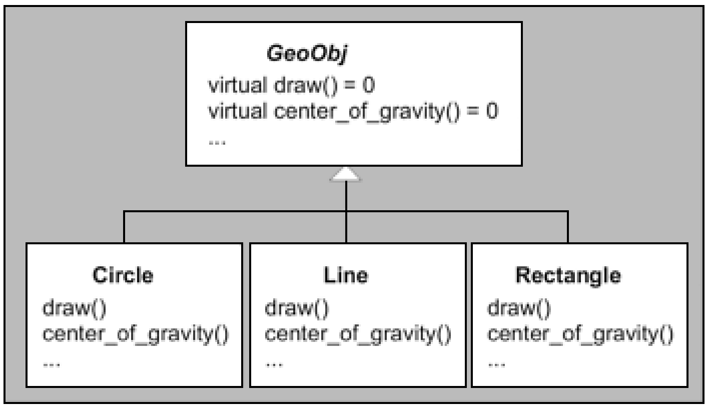
14.1 Dynamic Polymorphism

Historically, C++ started with supporting polymorphism only through the use of inheritance combined with virtual functions. [2] The art of polymorphic design in this context consists of identifying a common set of capabilities among related object types and declaring them as virtual function interfaces in a common base class.

[2] Strictly speaking, macros can also be thought of as an early form of static polymorphism. However, they are left out of consideration because they are mostly orthogonal to the other language mechanisms.

The poster child for this design approach is an application that manages geometric shapes and allows them to be rendered in some way (for example, on a screen). In such an application we might identify a so-called abstract base class (ABC) GeoObj, which declares the common operations and properties applicable to geometric objects. Each concrete class for specific geometric objects then derives from GeoObj (see Figure 14.1):

Figure 14.1. Polymorphism implemented via inheritance

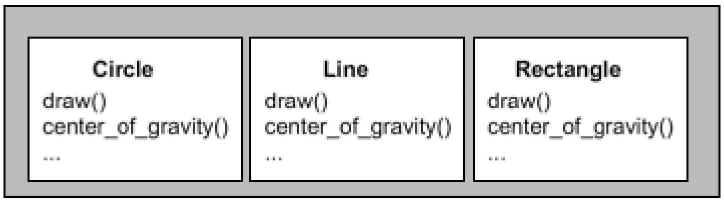


```
// poly/dynahier.hpp

#include "coord.hpp"

// common abstract base class GeoObj for geometric objects
class GeoObj {
  public:
    // draw geometric object:
    virtual void draw() const = 0;
    // return center of gravity of geometric object:
    virtual Coord center_of_gravity() const = 0;

};

// concrete geometric object class Circle
// - derived from GeoObj
class Circle : public GeoObj {
  public:
    virtual void draw() const;
    virtual Coord center_of_gravity() const;

};

// concrete geometric object class Line
// - derived from GeoObj
class Line : public GeoObj {
  public:
    virtual void draw() const;
```

14.2 Static Polymorphism

Templates can also be used to implement polymorphism. However, they don't rely on the factoring of common behavior in base classes. Instead, the commonality is implicit in that the different "shapes" of an application must support operations using common syntax (that is, the relevant functions must have the same names). Concrete classes are defined independently from each other (see Figure 14.2). The polymorphic power is then enabled when templates are instantiated with the concrete classes.

Figure 14.2. Polymorphism implemented via templates



For example, the function myDraw() in the previous section

```
void myDraw (GeoObj const& obj)   // GeoObj is abstract base class
{
    obj.draw();
}
```

could conceivably be rewritten as follows:

```
template <typename GeoObj>
void myDraw (GeoObj const& obj)   // GeoObj is template parameter
{
    obj.draw();
}
```

Comparing the two implementations of myDraw(), we may conclude that the main difference is the specification of GeoObj as a template parameter instead of a common base class. There are, however, more fundamental differences under the hood. For example, using dynamic polymorphism we had only one myDraw() function at run time, whereas with the template we have distinct functions, such as myDraw<Line>() and myDraw<Circle>().

We may attempt to recode the complete example of the previous section using static polymorphism. First, instead of a hierarchy of geometric classes, we have several individual geometric classes:

```
// poly/statichier.hpp

#include "coord.hpp"

// concrete geometric object class Circle
// - not derived from any class
class Circle {
  public:
    void draw() const;
    Coord center_of_gravity() const;
    …
};

// concrete geometric object class Line
// - not derived from any class
class Line {
  public:
```

14.3 Dynamic versus Static Polymorphism

Let's categorize and compare both forms of polymorphisms.

Terminology

Dynamic and static polymorphism provide support for different C++ programming idioms [3]:

[3] For a detailed discussion of polymorphism terminology, see also Sections 6.5 to 6.7 of [CzarneckiEiseneckerGenProg].

- Polymorphism implemented via inheritance is bounded and dynamic:

  - Bounded means that the interfaces of the types participating in the polymorphic behavior are predetermined by the design of the common base class (other terms for this concept are invasive or intrusive).

  - Dynamic means that the binding of the interfaces is done at run time (dynamically).

- Polymorphism implemented via templates is unbounded and static:

- Unbounded means that the interfaces of the types participating in the polymorphic behavior are not predetermined (other terms for this concept are noninvasive or nonintrusive).

- Static means that the binding of the interfaces is done at compile time (statically).

So, strictly speaking, in C++ parlance, dynamic polymorphism and static polymorphism are shortcuts for bounded dynamic polymorphism and unbounded static polymorphism. In other languages other combinations exist (for example, Smalltalk provides unbounded dynamic polymorphism). However, in the context of C++, the more concise terms dynamic polymorphism and static polymorphism do not cause confusion.

Strengths and Weaknesses

Dynamic polymorphism in C++ exhibits the following strengths:

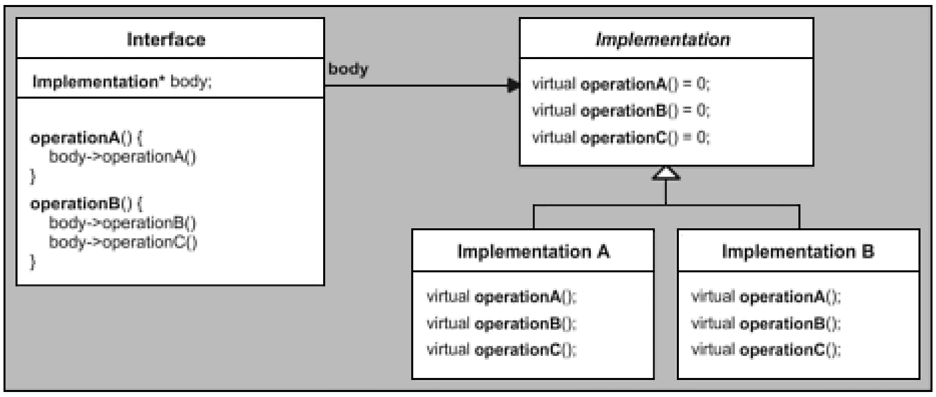
- Heterogeneous collections are handled elegantly.

- The executable code size is potentially smaller (because only one polymorphic function is needed, whereas distinct template instances must be generated to handle different types).

14.4 New Forms of Design Patterns

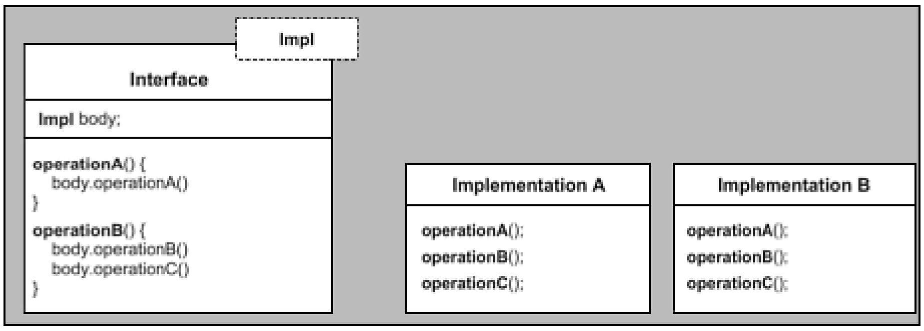
The new form of static polymorphism leads to new ways of implementing design patterns. Take, for example, the bridge pattern, which plays a major role in C++ programs. One goal of using the bridge pattern is to switch between different implementations of an interface. According to [DesignPatternsGoV] this is usually done by using a pointer to refer to the actual implementation and delegating all calls to this class (see Figure 14.3).

Figure 14.3. Bridge pattern implemented using inheritance



However, if the type of the implementation is known at compile time, you could use the approach via templates instead (see Figure 14.4). This leads to more type safety, avoids pointers, and should be faster.

Figure 14.4. Bridge pattern implemented using templates



14.5 Generic Programming

Static polymorphism leads to the concept of generic programming. However, there is no one universally agreed-on definition of generic programming (just as there is no one agreed-on definition of object-oriented programming). According to [CzarneckiEiseneckerGenProg], definitions go from programming with generic parameters to finding the most abstract representation of efficient algorithms. The book summarizes:

Generic programming is a subdiscipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization . Generic programming focuses on representing families of domain concepts. (pages 169 and 170)

In the context of C++, generic programming is sometimes defined as programming with templates (whereas object-oriented programming is thought of as programming with virtual functions). In this sense, just about any use of C++ templates could be thought of as an instance of generic programming. However, practitioners often think of generic programming as having an additional essential ingredient: Templates have to be designed in a framework for the purpose of enabling a multitude of useful combinations.

By far the most significant contribution in this area is the STL (the Standard Template Library, which later was adapted and incorporated into the C++ standard library). The STL is a framework that provides a number of useful operations, called algorithms, for a number of linear data structures for collections of objects, called containers. Both algorithms and containers are templates. However, the key is that the algorithms are not member functions of the containers. Instead, the algorithms are written in a generic way so that they can be used by any container (and linear collection of elements). To do this, the designers of STL identified an abstract concept of iterators that can be provided for any kind of linear collection. Essentially, the collection-specific aspects of container operations have been factored out into the iterators' functionality.

As a consequence, implementing an operation such as computing the maximum value in a sequence can be done without knowing the details of how values are stored in that sequence:

```
template <class Iterator>
Iterator max_element (Iterator beg,   // refers to start of collection
                      Iterator end)   // refers to end of collection
{
    // use only certain Iterator's operations to traverse all elements
    // of the collection to find the element with the maximum value
    // and return its position as Iterator

}
```

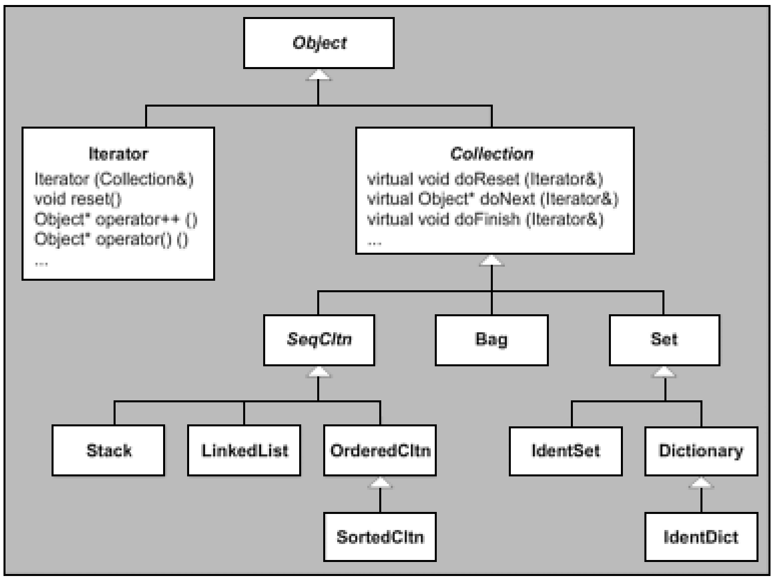
Instead of providing all useful operations such as max_element() by every linear container, the container has to provide only an iterator type to traverse the sequence of values it contains and member functions to create such iterators:

```
namespace std {
    template <class T,   >
    class vector {
      public:
        typedef   const_iterator;     // implementation-specific iterator
                                      // type for constant vectors
        const_iterator begin() const; // iterator for start of collection
        const_iterator end() const;   // iterator for end of collection

    };
```

14.6 Afternotes

Container types were a primary motivation for the introduction of templates into the C++ programming language. Prior to templates, polymorphic hierarchies were a popular approach to containers. A popular example was the National Institutes of Health Class Library (NIHCL), which to a large extent translated the container class hierarchy of Smalltalk (see Figure 14.5).

Figure 14.5. Class hierarchy of the NIHCL



Much like the C++ standard library, the NIHCL supported a rich variety of containers as well as iterators. However, the implementation followed the Smalltalk style of dynamic polymorphism: Iterators used the abstract base class Collection to operate on different types of collections:

```
Bag c1;
Set c2;

Iterator i1(s);
Iterator i2(b);
```

Unfortunately, the price of this approach was high both in terms of running time and memory usage. Running time was typically orders of magnitude worse than equivalent code using the C++ standard library because most operations ended up requiring a virtual call (whereas in the C++ standard library many operations are inlined, and no virtual functions are involved in iterator and container interfaces). Furthermore, because (unlike Smalltalk) the interfaces were bounded, built-in types had to be wrapped in larger polymorphic classes (such wrappers were provided by the NIHCL), which in turn could lead to dramatic increases in storage requirements.

Some sought solace in macros, but even in today's age of templates many projects still make suboptimal choices in their approach to polymorphism. Clearly there are many situations when dynamic polymorphism is the "right choice." Heterogeneous iterations are an example. However, in the same vein, many programming tasks are naturally and efficiently solved using templates, and homogeneous containers are an example of this.

Static polymorphism lends itself well to code very fundamental computing structures. In contrast, the need to choose a common base type implies that a dynamic polymorphic library will normally have to make domain-specific choices. It's no surprise then that the STL part of the C++ standard library never included polymorphic containers, but it contains a rich set of containers and iterators that use static polymorphism (as demonstrated in Section 14.5 on page

Chapter 15. Traits and Policy Classes

Templates enable us to parameterize classes and functions for various types. It could be tempting to introduce as many template parameters as possible to enable the customization of every aspect of a type or algorithm. In this way, our "templatized" components could be instantiated to meet the exact needs of client code. However, from a practical point of view it is rarely desirable to introduce dozens of template parameters for maximal parameterization. Having to specify all the corresponding arguments in the client code is overly tedious.

Fortunately, it turns out that most of the extra parameters we would introduce have reasonable default values. In some cases the extra parameters are entirely determined by a few main parameters, and we'll see that such extra parameters can be omitted altogether. Other parameters can be given default values that depend on the main parameters and will meet the needs of most situations, but the default values must occasionally be overridden (for special applications). Yet other parameters are unrelated to the main parameters: In a sense they are themselves main parameters, except for the fact that there exist default values that almost always fit the bill.

Policy classes and traits (or traits templates) are C++ programming devices that greatly facilitate the management of the sort of extra parameters that come up in the design of industrial-strength templates. In this chapter we show a number of situations in which they prove useful and demonstrate various techniques that will enable you to write robust and powerful devices of your own.

15.1 An Example: Accumulating a Sequence

Computing the sum of a sequence of values is a fairly common computational task. However, this seemingly simple problem provides us with an excellent example to introduce various levels at which policy classes and traits can help.

15.1.1 Fixed Traits

Let's first assume that the values of the sum we want to compute are stored in an array, and we are given a pointer to the first element to be accumulated and a pointer one past the last element to be accumulated. Because this book is about templates, we wish to write a template that will work for many types. The following may seem straightforward by now [1]:

[1] Most examples in this section use ordinary pointers for the sake of simplicity. Clearly, an industrial-strength interface may prefer to use iterator parameters following the conventions of the C++ standard library (see [JosuttisStdLib]). We revisit this aspect of our example later.

```
// traits/accum1.hpp

#ifndef ACCUM_HPP
#define ACCUM_HPP

template <typename T>
inline
T accum (T const* beg, T const* end)
{
    T total = T();  // assume T() actually creates a zero value
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}

#endif // ACCUM_HPP
```

The only slightly subtle decision here is how to create a zero value of the correct type to start our summation. We use the expression T() here, which normally should work for built-in numeric types like int and float (see Section 5.5 on page 56).

To motivate our first traits template, consider the following code that makes use of our accum():

```
// traits/accum1.cpp

#include "accum1.hpp"
#include <iostream>

int main()
{

// create array of 5 integer values
int num[]={1,2,3,4,5};

// print average value
std::cout << "the average value of the integer values is "
          << accum(&num[0], &num[5]) / 5
          << '\n';
```

15.2 Type Functions

The initial traits example demonstrates that you can define behavior that depends on types. This is different from what you usually implement in programs. In C and C++, functions more exactly can be called value functions: They take some values as parameters and return another value as a result. Now, what we have with templates are type functions: a function that takes some type arguments and produces a type or constant as a result.

A very useful built-in type function is sizeof, which returns a constant describing the size (in bytes) of the given type argument. Class templates can also serve as type functions. The parameters of the type function are the template parameters, and the result is extracted as a member type or member constant. For example, the sizeof operator could be given the following interface:

```
// traits/sizeof.cpp

#include <stddef.h>
#include <iostream>

template <typename T>
class TypeSize {
  public:
    static size_t const value = sizeof(T);
};

int main()
{
    std::cout << "TypeSize<int>::value = "
              << TypeSize<int>::value << std::endl;
}
```

In what follows we develop a few more general-purpose type functions that can be used as traits classes in this way.

15.2.1 Determining Element Types

For another example, assume that we have a number of container templates such as vector<T>, list<T>, and stack<T>. We want a type function that, given such a container type, produces the element type. This can be achieved using partial specialization:

```
// traits/elementtype.cpp

#include <vector>
#include <list>
#include <stack>
#include <iostream>
#include <typeinfo>

template <typename T>
class ElementT;                     // primary template

template <typename T>
class ElementT<std::vector<T> > {  // partial specialization
  public:
    typedef T Type;
};

template <typename T>
class ElementT<std::list<T> > {    // partial specialization
  public:
    typedef T Type;
```

15.3 Policy Traits

So far, our examples of traits templates have been used to determine properties of template parameters: what sort of type they represent, to which type they should promote in mixed-type operations, and so forth. Such traits are called property traits.

In contrast, some traits define how some types should be treated. We call them policy traits. This is reminiscent of the previously discussed concept of policy classes (and we already pointed out that the distinction between traits and policies is not entirely clear), but policy traits tend to be more unique properties associated with a template parameter (whereas policy classes are usually independent of other template parameters).

Although property traits can often be implemented as type functions, policy traits usually encapsulate the policy in member functions. As a first illustration, let's look at a type function that defines a policy for passing read-only parameters.

15.3.1 Read-only Parameter Types

In C and C++, function call arguments are passed "by value" by default. This means that the values of the arguments computed by the caller are copied to locations controlled by the callee. Most programmers know that this can be costly for large structures and that for such structures it is appropriate to pass the arguments "by reference-to-const" (or "by pointer-to-const" in C). For smaller structures, the picture is not always clear, and the best mechanism from a performance point of view depends on the exact architecture for which the code is being written. This is not so critical in most cases, but sometimes even the small structures must be handled with care.

With templates, of course, things get a little more delicate: We don't know a priori how large the type substituted for the template parameter will be. Furthermore, the decision doesn't depend just on size: A small structure may come with an expensive copy constructor that would still justify passing read-only parameters "by reference-to-const."

As hinted at earlier, this problem is conveniently handled using a policy traits template that is a type function: The function maps an intended argument type T onto the optimal parameter type T or T const&. As a first approximation, the primary template can use "by value" passing for types no larger than two pointers and "by reference-to-const" for everything else:

```
template<typename T>
class RParam {
  public:
    typedef typename IfThenElse<sizeof(T)<=2*sizeof(void*),
                                T,
                                T const&>::ResultT Type;
};
```

On the other hand, container types for which sizeof returns a small value may involve expensive copy constructors. So we may need many specializations and partial specializations, such as the following:

```
template<typename T>
class RParam<Array<T> > {
  public:
    typedef Array<T> const& Type;
};
```

Because such types are common in C++, it may be safer to mark nonclass types "by value" in the primary template

15.4 Afternotes

Nathan Myers was the first to formalize the idea of traits parameters. He originally presented them to the C++ standardization committee as a vehicle to define how character types should be treated in standard library components (for example, input and output streams). At that time he called them baggage templates and noted that they contained traits. However, some C++ committee members did not like the term baggage, and the name traits was promoted instead. The latter term has been widely used since then.

Client code usually does not deal with traits at all: The default traits classes satisfy the most common needs, and because they are default template arguments, they need not appear in the client source at all. This argues in favor of long descriptive names for the default traits templates. When client code does adapt the behavior of a template by providing a custom traits argument, it is good practice to typedef the resulting specializations to a name that is appropriate for the custom behavior. In this case the traits class can be given a long descriptive name without sacrificing too much source estate.

Our discussion has presented traits templates as being class templates exclusively. Strictly speaking, this does not need to be the case. If only a single policy trait needs to be provided, it could be passed as an ordinary function template. For example:

```
template <typename T, void (*Policy)(T const&, T const&)>
class X;
```

However, the original goal of traits was to reduce the baggage of secondary template arguments, which is not achieved if only a single trait is encapsulated in a template parameter. This justifies Myers's preference for the term baggage as a collection of traits. We revisit the problem of providing an ordering criterion in Chapter 22.

The standard library defines a class template std::char_traits, which is used as a policy traits parameter. To adapt algorithms easily to the kind of STL iterators for which they are used, a very simple std::iterator_traits property traits template is provided (and used in standard library interfaces). The template std::numeric_limits can also be useful as a property traits template, but it is not visibly used in the standard library proper. The class templates std::unary_function and std::binary_function fall in the same category and are very simple type functions: They only typedef their arguments to member names that make sense for functors (also known as function objects, see Chapter 22). Lastly, memory allocation for the standard container types is handled using a policy traits class. The template std::allocator is provided as the standard item for this purpose.

Policy classes have apparently been developed by many programmers and a few authors. Andrei Alexandrescu made the term policy classes popular, and his book Modern C++ Design covers them in more detail than our brief section (see [AlexandrescuDesign]).

Chapter 16. Templates and Inheritance

A priori, there might be no reason to think that templates and inheritance interact in interesting ways. If anything, we know from Chapter 9 that deriving from dependent base classes forces us to deal carefully with unqualified names. However, it turns out that some interesting techniques make use of so-called parameterized inheritance. In this chapter we describe a few of these techniques.

16.1 Named Template Arguments

Various template techniques sometimes cause a class template to end up with many different template type parameters. However, many of these parameters often have reasonable default values. A natural way to define such a class template may look as follows:

```
template<typename Policy1 = DefaultPolicy1,
         typename Policy2 = DefaultPolicy2,
         typename Policy3 = DefaultPolicy3,
         typename Policy4 = DefaultPolicy4>
class BreadSlicer {

};
```

Presumably, such a template can often be used with the default template argument values using the syntax BreadSlicer<>. However, if a nondefault argument must be specified, all preceding arguments must be specified too (even though they may have the default value).

Clearly, it would be attractive to be able to use a construct akin to BreadSlicer<Policy3 = Custom> rather than BreadSlicer<DefaultPolicy1, DefaultPolicy2, Custom> as is the case right now. In what follows we develop a technique to enable almost exactly that. [1]

[1] Note that a similar language extension for function call arguments was proposed (and rejected) earlier in the C++ standardization process (see Section 13.9 on page 216 for details).

Our technique consists of placing the default type values in a base class and overriding some of them through derivation. Instead of directly specifying the type arguments, we provide them through helper classes. For example, we could write BreadSlicer<Policy3_is<Custom> >. Because each template argument can describe any of the policies, the defaults cannot be different. In other words, at a high level every template parameter is equivalent:

```
template <typename PolicySetter1 = DefaultPolicyArgs,
          typename PolicySetter2 = DefaultPolicyArgs,
          typename PolicySetter3 = DefaultPolicyArgs,
          typename PolicySetter4 = DefaultPolicyArgs>
class BreadSlicer {
    typedef PolicySelector<PolicySetter1, PolicySetter2,
                           PolicySetter3, PolicySetter4>
            Policies;
    // use Policies::P1, Policies::P2,   to refer to the various policies

};
```

The remaining challenge is to write the PolicySelector template. It has to merge the different template arguments into a single type that overrides default typedef members with whichever non-defaults were specified. This merging can be achieved using inheritance:

```
// PolicySelector<A,B,C,D> creates A,B,C,D as base classes
// Discriminator<> allows having even the same base class more than once

template<typename Base, int D>
class Discriminator : public Base {
};

template <typename Setter1, typename Setter2,
          typename Setter3, typename Setter4>
class PolicySelector : public Discriminator<Setter1,1>,
```

16.2 The Empty Base Class Optimization (EBCO)

C++ classes are often "empty," which means that their internal representation does not require any bits of memory at run time. This is the case typically for classes that contain only type members, nonvirtual function members, and static data members. Nonstatic data members, virtual functions, and virtual base classes, on the other hand, do require some memory at run time.

Even empty classes, however, have nonzero size. Try the following program if you'd like to verify this:

```
// inherit/empty.cpp

#include <iostream>

class EmptyClass {
};
int main()
{
    std::cout << "sizeof(EmptyClass): " << sizeof(EmptyClass)
              << '\n';
}
```

For many platforms, this program will print 1 as size of EmptyClass. A few systems impose more strict alignment requirements on class types and may print another small integer (typically, 4).

16.2.1 Layout Principles

The designers of C++ had various reasons to avoid zero-size classes. For example, an array of zero-size classes would presumably have size zero too, but then the usual properties of pointer arithmetic would not apply anymore. For example, let's assume ZeroSizedT is a zero-size type:

```
ZeroSizedT z[10];

&z[i] - &z[j]      // compute distance between pointers/addresses
```

Normally, the difference in the previous example is obtained by dividing the number of bytes between the two addresses by the size of the type to which it is pointing, but when that size is zero this is clearly not satisfactory.

However, even though there are no zero-size types in C++, the C++ standard does specify that when an empty class is used as a base class, no space needs to be allocated for it provided that it does not cause it to be allocated to the same address as another object or subobject of the same type. Let's look at some examples to clarify what this so-called empty base class optimization (or EBCO) means in practice. Consider the following program:

```
// inherit/ebco1.cpp

#include <iostream>

class Empty {
    typedef int Int;  // typedef members don't make a class nonempty
};

class EmptyToo : public Empty {
};

class EmptyThree : public EmptyToo {
};
int main()
```

16.3 The Curiously Recurring Template Pattern (CRTP)

This oddly named pattern refers to a general class of techniques that consists of passing a derived class as a template argument to one of its own base classes. In its simplest form, C++ code for such a pattern looks as follows:

```
template <typename Derived>
class CuriousBase {

};

class Curious : public CuriousBase<Curious> {

};
```

Our first outline of CRTP shows a nondependent base class: The class Curious is not a template and is therefore immune to some of the name visibility issues of dependent base classes. However, this is not an intrinsic characteristic of CRTP. Indeed, we could just as well have used the following alternative outline:

```
template <typename Derived>
class CuriousBase {

};

template <typename T>
class CuriousTemplate : public CuriousBase<CuriousTemplate<T> > {

};
```

From this outline, however, it is not a far stretch to propose yet another alternative formulation, this time involving a template template parameter:

```
template <template<typename> class Derived>
class MoreCuriousBase {

};

template <typename T>
class MoreCurious : public MoreCuriousBase<MoreCurious> {

};
```

A simple application of CRTP consists of keeping track of how many objects of a certain class type were created. This is easily achieved by incrementing an integral static data member in every constructor and decrementing it in the destructor. However, having to provide such code in every class is tedious. Instead, we can write the following template:

```
// inherit/objectcounter.hpp

#include <stddef.h>

template <typename CountedType>
class ObjectCounter {
  private:
    static size_t count;    // number of existing objects

  protected:
    // default constructor
    ObjectCounter() {
        ++ObjectCounter<CountedType>::count;
    }
```

16.4 Parameterized Virtuality

C++ allows us to parameterize directly three kinds of entities through templates: types, constants ("nontypes"), and templates. However, indirectly, it also allows us to parameterize other attributes such as the virtuality of a member function. A simple example shows this rather surprising technique:

```
// inherit/virtual.cpp

#include <iostream>

class NotVirtual {
};

class Virtual {
  public:
    virtual void foo() {
    }
};

template <typename VBase>
class Base : private VBase {
  public:
    // the virtuality of foo() depends on its declaration
    // (if any) in the base class VBase
    void foo() {
        std::cout << "Base::foo()" << '\n';
    }
};

template <typename V>
class Derived : public Base<V> {
  public:
    void foo() {
       std::cout << "Derived::foo()" << '\n';
    }
};

int main()
{
    Base<NotVirtual>* p1 = new Derived<NotVirtual>;
    p1->foo();  // calls Base::foo()

    Base<Virtual>* p2 = new Derived<Virtual>;
    p2->foo();  // calls Derived::foo()
}
```

This technique can provide a tool to design a class template that is usable both to instantiate concrete classes and to extend using inheritance. However, it is rarely sufficient just to sprinkle virtuality on some member functions to obtain a class that makes a good base class for more specialized functionality. This sort of development method requires more fundamental design decisions. It is therefore usually more practical to design two different tools (class or class template hierarchies) rather than trying to integrate them all into one template hierarchy.

16.5 Afternotes

Named template arguments are used to simplify certain class templates in the Boost library. Boost uses metaprogramming to create a type with properties similar to our PolicySelector (but without using virtual inheritance). The simpler alternative presented here was developed by one of us (Vandevoorde).

CRTPs have been in use since at least 1991. However, James Coplien was first to describe them formally as a class of so-called patterns (see [CoplienCRTP]). Since then, many applications of CRTP have been published. The phrase parameterized inheritance is sometimes wrongly equated with CRTP. As we have shown, CRTP does not require the derivation to be parameterized at all, and many forms of parameterized inheritance do not conform to CRTP. CRTP is also sometimes confused with the Barton-Nackman trick (see Section 11.7 on page 174) because Barton and Nackman frequently used CRTP in combination with friend name injection (and the latter is an important component of the Barton-Nackman trick). Our ObjectCounter example is almost identical to a technique developed by Scott Meyers in [MeyersCounting].

Bill Gibbons was the main sponsor behind the introduction of EBCO into the C++ programming language. Nathan Myers made it popular and proposed a template similar to our BaseMemberPair to take better advantage of it. The Boost library contains a considerably more sophisticated template, called compressed_pair, that resolves some of the problems we reported for the MyClass template in this chapter. boost::compressed_pair can also be used instead of our BaseMemberPair.

Chapter 17. Metaprograms

Metaprogramming consists of "programming a program." In other words, we lay out code that the programming system executes to generate new code that implements the functionality we really want. Usually the term metaprogramming implies a reflexive attribute: The metaprogramming component is part of the program for which it generates a bit of code/program.

Why would metaprogramming be desirable? As with most other programming techniques, the goal is to achieve more functionality with less effort, where effort can be measured as code size, maintenance cost, and so forth. What characterizes metaprogramming is that some user-defined computation happens at translation time. The underlying motivation is often performance (things computed at translation time can frequently be optimized away) or interface simplicity (a metaprogram is generally shorter than what it expands to) or both.

Metaprogramming often relies on the concepts of traits and type functions as developed in Chapter 15. We therefore recommend getting familiar with that chapter prior to delving into this one.

17.1 A First Example of a Metaprogram

In 1994 during a meeting of the C++ standardization committee, Erwin Unruh discovered that templates can be used to compute something at compile time. He wrote a program that produced prime numbers. The intriguing part of this exercise, however, was that the production of the prime numbers was performed by the compiler during the compilation process and not at run time. Specifically, the compiler produced a sequence of error messages with all prime numbers from two up to a certain configurable value. Although this program wasn't strictly portable (error messages aren't standardized), the program did show that the template instantiation mechanism is a primitive recursive language that can perform nontrivial computations at compile time. This sort of compile-time computation that occurs through template instantiation is commonly called template metaprogramming.

As an introduction to the details of metaprogramming we start with a simple exercise (we will show Erwin's prime number program later on page 318). The following program shows how to compute at compile time the power of three for a given value:

```
// meta/pow3.hpp

#ifndef POW3_HPP
#define POW3_HPP

// primary template to compute 3 to the Nth
template<int N>
class Pow3 {
  public:
    enum { result=3*Pow3<N-1>::result };
};

// full specialization to end the recursion
template<>
class Pow3<0> {
  public:
    enum { result = 1 };
};

#endif // POW3_HPP
```

The driving force behind template metaprogramming is recursive template instantiation. [1] In our program to compute 3N , recursive template instantiation is driven by the following two rules:

[1] We saw an example of a recursive template in Section 12.4 on page 200. It could be considered a simple case of metaprogramming.

1.

   3N = 3 * 3N -1

2.

   30 = 1

The first template implements the general recursive rule:

```
template<int N>
class Pow3 {
  public:
```

17.2 Enumeration Values versus Static Constants

In old C++ compilers, enumeration values were the only available possibility to have "true constants" (so-called constant-expressions) inside class declarations. However, this has changed during the standardization of C++, which introduced the concept of in-class static constant initializers. A brief example illustrates the construct:

```
struct TrueConstants {
    enum { Three = 3 };
    static int const Four = 4;
};
```

In this example, Four is a "true constant"—just as is Three.

With this, our Pow3 metaprogram may also look as follows:

```
// meta/pow3b.hpp

#ifndef POW3_HPP
#define POW3_HPP

// primary template to compute 3 to the Nth
template<int N>
class Pow3 {
  public:
    static int const result = 3 * Pow3<N-1>::result;
};

// full specialization to end the recursion
template<>
class Pow3<0> {
  public:
    static int const result = 1;
};

#endif // POW3_HPP
```

The only difference is the use of static constant members instead of enumeration values. However, there is a drawback with this version: Static constant members are lvalues. So, if you have a declaration such as

```
void foo(int const&);
```

and you pass it the result of a metaprogram

```
foo(Pow3<7>::result);
```

a compiler must pass the address of Pow3<7>::result, which forces the compiler to instantiate and allocate the definition for the static member. As a result, the computation is no longer limited to a pure "compile-time" effect.

Enumeration values aren't lvalues (that is, they don't have an address). So, when you pass them "by reference," no static memory is used. It's almost exactly as if you passed the computed value as a literal. These considerations motivate us to use enumeration values in all metaprograms throughout this book.

17.3 A Second Example: Computing the Square Root

Lets look at a slightly more complicated example: a metaprogram that computes the square root of a given value N . The metaprogram looks as follows (explanation of the technique follows):

```
// meta/sqrt1.hpp

#ifndef SQRT_HPP
#define SQRT_HPP

// primary template to compute sqrt(N)
template <int N, int LO=1, int HI=N>
class Sqrt {
  public:
    // compute the midpoint, rounded up
    enum { mid = (LO+HI+1)/2 };

    // search a not too large value in a halved interval
    enum { result = (N<mid*mid) ? Sqrt<N,LO,mid-1>::result
                                : Sqrt<N,mid,HI>::result };
};

// partial specialization for the case when LO equals HI
template<int N, int M>
class Sqrt<N,M,M> {
  public:
    enum { result=M};
};

#endif // SQRT_HPP
```

The first template is the general recursive computation that is invoked with the template parameter N (the value for which to compute the square root) and two other optional parameters. The latter represent the minimum and maximum values the result can have. If the template is called with only one argument, we know that the square root is at least one and at most the value itself.

Our recursion then proceeds using a binary search technique (often called method of bisection in this context). Inside the template, we compute whether result is in the first or the second half of the range between LO and HI. This case differentiation is done using the conditional operator ? :. If mid2 is greater than N, we continue the search in the first half. If mid2 is less than or equal to N, we use the same template for the second half again.

The specialization that ends the recursive process is invoked when LO and HI have the same value M, which is our final result.

Again, let's look at the details of a simple program that uses this metaprogram:

```
// meta/sqrt1.cpp

#include <iostream>
#include "sqrt1.hpp"

int main()
{
    std::cout << "Sqrt<16>::result = " << Sqrt<16>::result
              << '\n';
    std::cout << "Sqrt<25>::result = " << Sqrt<25>::result
              << '\n';
```

17.4 Using Induction Variables

You may argue that the way the metaprogram is written in the previous example looks rather complicated. And you may wonder whether you have learned something you can use whenever you have a problem to solve by a metaprogram. So, let's look for a more "naive" and maybe "more iterative" implementation of a metaprogram that computes the square root.

A "naive iterative algorithm" can be formulated as follows: To compute the square root of a given value N, we write a loop in which a variable I iterates from one to N until its square is equal to or greater than N. This value I is our square root of N. If we formulate this problem in ordinary C++, it looks as follows:

```
int I;
for (I=1; I*I<N; ++I) {
    ;
}
// I now contains the square root of N
```

However, as a metaprogram we have to formulate this loop in a recursive way, and we need an end criterion to end the recursion. As a result, an implementation of this loop as a metaprogram looks as follows:

```
// meta/sqrt3.hpp

#ifndef SQRT_HPP
#define SQRT_HPP

// primary template to compute sqrt(N) via iteration
template <int N, int I=1>
class Sqrt {
  public:
    enum { result = (I*I<N) ? Sqrt<N,I+1>::result
                            : I };
};

// partial specialization to end the iteration
template<int N>
class Sqrt<N,N> {
  public:
    enum { result = N };
};

#endif // SQRT_HPP
```

We loop by "iterating" I over Sqrt<N,I>. As long as I*I<N yields true, we use the result of the next iteration Sqrt<N,I+1>::result as result. Otherwise I is our result.

For example, if we evaluate Sqrt<16> this gets expanded to Sqrt<16,1>. Thus, we start an iteration with one as a value of the so-called induction variable I. Now, as long as I2 (that is I*I) is less than N, we use the next iteration value by computing Sqrt<N,I+1>::result. When I2 is equal to or greater than N we know that I is the result.

You may wonder why we need a template specialization to end the recursion because the first template always, sooner or later, finds I as the result, which seems to end the recursion. Again, this is the effect of the instantiation of both branches of operator ?:, which was discussed in the previous section. Thus, the compiler computes the result of Sqrt<4> by instantiating as follows:

-

17.5 Computational Completeness

The Pow3<> and Sqrt<> examples show that a template metaprogram can contain:

- State variables: the template parameters
- Loop constructs: through recursion
- Path selection: by using conditional expressions or specializations
- Integer arithmetic

If there are no limits to the amount of recursive instantiations and the amount of state variables that are allowed, it can be shown that this is sufficient to compute anything that is computable. However, it may not be convenient to do so using templates. Furthermore, template instantiation typically requires substantial compiler resources, and extensive recursive instantiation quickly slows down a compiler or even exhausts the resources available. The C++ standard recommends but does not mandate that 17 levels of recursive instantiations be allowed as a minimum. Intensive template metaprogramming easily exhausts such a limit.

Hence, in practice, template metaprograms should be used sparingly. The are a few situations, however, when they are irreplaceable as a tool to implement convenient templates. In particular, they can sometimes be hidden in the innards of more conventional templates to squeeze more performance out of critical algorithm implementations.

17.6 Recursive Instantiation versus Recursive Template Arguments

Consider the following recursive template:

```
template<typename T, typename U>
struct Doublify {};

template<int N>
struct Trouble {
    typedef Doublify<typename Trouble<N-1>::LongType,
                     typename Trouble<N-1>::LongType> LongType;
};

template<>
struct Trouble<0> {
    typedef double LongType;
};

Trouble<10>::LongType ouch;
```

The use of Trouble<10>::LongType not only triggers the recursive instantiation of Trouble<9>, Trouble<8>, , Trouble<0>, but it also instantiates Doublify over increasingly complex types. Indeed, Table 17.1 illustrates how quickly it grows.

As can be seen from Table 17.1, the complexity of the type description of the expression Trouble<N>::LongType grows exponentially with N. In general, such a situation stresses a C++ compiler even more than recursive instantiations that do not involve recursive template arguments. One of the problems here is that a compiler keeps a representation of the mangled name for the type. This mangled name encodes the exact template specialization in some way, and early C++ implementations used an encoding that is roughly proportional to the length of the template-id. These compilers then used well over 10,000 characters for Trouble<10>::LongType.

Newer C++ implementations take into account the fact that nested template-ids are fairly common in modern C++ programs and use clever compression techniques to reduce considerably the growth

Table 17.1. Growth of Trouble<N>::LongType

Typedef Name	**Underlying Type**
`Trouble<0>::LongType`	`double`
`Trouble<1>::LongType`	`Doublify<double,double>`

`Trouble<2>::LongType`	`Doublify<Doublify<double,double>,`
	`         Doublify<double,double> >`

`Trouble<3>::LongType`	`Doublify<Doublify<Doublify<double,double>,`
	`                  Doublify<double,double>`
	`>,`
	`         <Doublify<double,double>,`
	`                   Doublify<double,double>`
	`> >`

in name encoding (for example, a few hundred characters for Trouble<10>::LongType). Still, all other things being equal, it is probably preferable to organize recursive instantiation in such a way that template arguments need not also be nested recursively.

17.7 Using Metaprograms to Unroll Loops

One of the first practical applications of metaprogramming was the unrolling of loops for numeric computations, which is shown here as a complete example.

Numeric applications often have to process n-dimensional arrays or mathematical vectors. One typical operation is the computation of the so-called dot product. The dot product of two mathematical vectors a and b is the sum of all products of corresponding elements in both vectors. For example, if each vectors has three elements, the result is

```
a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
```

A mathematical library typically provides a function to compute such a dot product. Consider the following straightforward implementation:

```
// meta/loop1.hpp

#ifndef LOOP1_HPP
#define LOOP1_HPP

template <typename T>
inline T dot_product (int dim, T* a, T* b)
{
    T result = 0;
    for (int i=0; i<dim; ++i) {
        result += a[i]*b[i];
    }
    return result;
}

#endif // LOOP1_HPP
```

When we call this function as follows

```
// meta/loop1.cpp

#include <iostream>
#include "loop1.hpp"

int main()
{
    int a[3] = { 1, 2, 3 };
    int b[3] = { 5, 6, 7 };

    std::cout << "dot_product(3,a,b) = " << dot_product(3,a,b)
              << '\n';
    std::cout << "dot_product(3,a,a) = " << dot_product(3,a,a)
              << '\n';
}
```

we get the following result:

```
dot_product(3,a,b) = 38
dot_product(3,a,a) = 14
```

This is correct, but it takes too long for serious high-performance applications. Even declaring the function inline is often not sufficient to attain optimal performance.

The problem is that compilers usually optimize loops for many iterations, which is counterproductive in this case

17.8 Afternotes

As mentioned earlier, the earliest documented example of a metaprogram was by Erwin Unruh, then representing Siemens on the C++ standardization committee. He noted the computational completeness of the template instantiation process and demonstrated his point by developing the first metaprogram. He used the Metaware compiler and coaxed it into issuing error messages that would contain successive prime numbers. Here is the code that was circulated at a C++ committee meeting in 1994 (modified so that it now compiles on standard conforming compilers) [3]:

[3] Thanks to Erwin Unruh for providing the code for this book. You can find the original example at [UnruhPrimeOrig].

```
// meta/unruh.cpp

// prime number computation by Erwin Unruh

template <int p, int i>
class is_prime {
  public:
    enum { prim = (p==2) || (p%i) && is_prime<(i>2?p:0),i-1>::prim
         };
};

template<>
class is_prime<0,0> {
  public:
    enum {prim=1};
};

template<>
class is_prime<0,1> {
  public:
    enum {prim=1};
};

template <int i>
class D {
  public:
    D(void*);
};

template <int i>
class Prime_print {      // primary template for loop to print prime numbers
  public:
    Prime_print<i-1> a;
    enum { prim = is_prime<i,i-1>::prim
         };
    void f() {
        D<i> d = prim ? 1 : 0;
        a.f();
    }
};

template<>
class Prime_print<1> {  // full specialization to end the loop
  public:
    enum {prim=0};
    void f() {
        D<1> d = prim ? 1 : 0;
    };
};
```

Chapter 18. Expression Templates

In this chapter we explore a template programming technique called expression templates. It was originally invented in support of numeric array classes, and that is also the context in which we introduce it here.

A numeric array class supports numeric operations on whole array objects. For example, it is possible to add two arrays, and the result contains elements that are the sums of the corresponding values in the argument arrays. Similarly, a whole array can be multiplied by a scalar, meaning that each element of the array is scaled. Naturally, it is desirable to keep the operator notation that is so familiar for built-in scalar types:

```
Array<double> x(1000), y(1000);

x = 1.2*x + x*y;
```

For the serious number cruncher it is crucial that such expressions be evaluated as efficiently as can be expected from the platform on which the code is run. Achieving this with the compact operator notation of this example is no trivial task, but expression templates will come to our rescue.

Expression templates are reminiscent of template metaprogramming. In part this is due to the fact that expression templates rely on sometimes deeply nested template instantiations, which are not unlike the recursive instantiations encountered in template metaprograms. The fact that both techniques were originally developed to support high-performance (see our example using templates to unroll loops on page 314) array operations probably also contributes to a sense that they are related. Certainly the techniques are complementary. For example, metaprogramming is convenient for small, fixed-size array whereas expression templates are very effective for operations on medium-to-large arrays sized at run time.

18.1 Temporaries and Split Loops

To motivate expression templates, let's start with a straightforward (or maybe "naive") approach to implement templates that enable numeric array operations. A basic array template might look as follows (SArray stands for simple array):

```
// exprtmpl/sarray1.hpp

#include <stddef.h>
#include <cassert>

template<typename T>
class SArray {
  public:
    // create array with initial size
    explicit SArray (size_t s)
     : storage(new T[s]), storage_size(s) {
        init();
    }

    // copy constructor
    SArray (SArray<T> const& orig)
     : storage(new T[orig.size()]), storage_size(orig.size()) {
        copy(orig);
    }

    // destructor: free memory
    ~SArray() {
        delete[] storage;
    }

    // assignment operator
    SArray<T>& operator= (SArray<T> const& orig) {
        if (&orig!=this) {
            copy(orig);
        }
        return *this;
    }

    // return size
    size_t size() const {
        return storage_size;
    }

    // index operator for constants and variables
    T operator[] (size_t idx) const {
        return storage[idx];
    }
    T& operator[] (size_t idx) {
        return storage[idx];
    }

  protected:
    // init values with default constructor
    void init() {
        for (size_t idx = 0; idx<size(); ++idx) {
            storage[idx] = T();
        }
    }
    // copy values of another array
    void copy (SArray<T> const& orig) {
        assert(size()==orig.size());
        for (size_t idx = 0; idx<size(); ++idx) {
            storage[idx] = orig.storage[idx];
```

18.2 Encoding Expressions in Template Arguments

The key to resolving our problem is not to attempt to evaluate part of an expression until the whole expression has been seen (in our example, until the assignment operator is invoked). Thus, before the evaluation we must record which operations are being applied to which objects. The operations are determined at compile time and can therefore be encoded in template arguments.

For our example expression

```
1.2*x + x*y;
```

this means that the result of 1.2*x is not a new array but an object that represents each value of x multiplied by 1.2. Similarly, x*y must yield each element of x multiplied by each corresponding element of y. Finally, when we need the values of the resulting array, we do the computation that we stored for later evaluation.

Let's look at a concrete implementation. With this implementation we transform the written expression

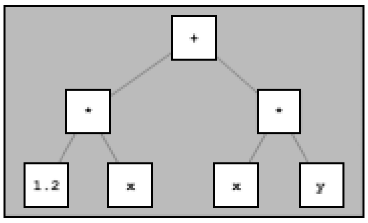
```
1.2*x + x*y;
```

into an object with the following type:

```
A_Add< A_Mult<A_Scalar<double>,Array<double> >,
       A_Mult<Array<double>,Array<double> > >
```

We combine a new fundamental Array class template with class templates A_Scalar, A_Add, and A_Mult. You may recognize a prefix representation for the syntax tree corresponding to this expression (see Figure 18.1). This nested template-id represents the operations involved and the types of the objects to which the operations should be applied. A_Scalar is presented later but is essentially just a placeholder for a scalar in an array expression.

Figure 18.1. Tree representation of expression 1.2*x+x*y



18.2.1 Operands of the Expression Templates

To complete the representation of the expression, we must store references to the arguments in each of the A_Add and A_Mult objects and record the value of the scalar in the A_Scalar object (or a reference thereto). Here are possible definitions for the corresponding operands:

```
// exprtmpl/exprops1.hpp

#include <stddef.h>

#include <cassert>

// include helper class traits template to select wether to refer to an
// ''expression template node'' either ''by value'' or ''by reference.''
```

18.3 Performance and Limitations of Expression Templates

To justify the complexity of the expression template idea, we have already invoked greatly enhanced performance on arraywise operations. As you trace what happens with the expression templates, you'll find that many small inline functions call each other and that many small expression template objects are allocated on the call stack. The optimizer must perform complete inlining and elimination of the small objects to produce code that performs as well as manually coded loops. The latter feat is still rare among C++ compilers at the time of this writing.

The expression templates technique does not resolve all the problematic situations involving numeric operations on arrays. For example, it does not work for matrix-vector multiplications of the form

```
x = A*x;
```

where x is a column vector of size n and A is an n-by-n matrix. The problem here is that a temporary must be used because each element of the result can depend on each element of the original x. Unfortunately, the expression template loop updates the first element of x right away and then uses that newly computed element to compute the second element, which is wrong. The slightly different expression

```
x = A*y;
```

on the other hand, does not need a temporary if x and y aren't aliases for each other, which implies that a solution would have to know the relationship of the operands at run time. This in turn suggests creating a run-time structure that represents the expression tree instead of encoding the tree in the type of the expression template. This approach was pioneered by the NewMat library of Robert Davies (see [NewMat]). It was known long before expression templates were developed.

Expression templates aren't limited to numeric computations either. An intriguing application, for example, is Jaakko Järvi and Gary Powell's Lambda Library (see [LambdaLib]). This library uses standard library function objects as expression objects. For example, it allows us to write the following:

```
void lambda_demo (std::vector<long*> & ones) {
    std::sort(ones.begin(), ones.end(), *_1 > *_2);
}
```

This short code excerpt sorts an array in increasing order of the value of what its elements refer to. Without the Lambda library, we'd have to define a simple (but cumbersome) special-purpose functor type. Instead, we can now use simple inline syntax to express the operations we want to apply. In our example, _1 and _2 are placeholders provided by the Lambda library. They correspond to elementary expression objects that are also functors. They can then be used to construct more complex expressions using the techniques developed in this chapter.

18.4 Afternotes

Expression templates were developed independently by Todd Veldhuizen and David Vandevoorde (Todd coined the term) at a time when member templates were not yet part of the C++ programming language (and it seemed at the time that they would never be added to C++). This caused some problems in implementing the assignment operator: It could not be parameterized for the expression template. One technique to work around this consisted of introducing in the expression templates a conversion operator to a Copier class parameterized with the expression template but inheriting from a base class that was parameterized only in the element type. This base class then provided a (virtual) copy_to interface to which the assignment operator could refer. Here is a sketch of the mechanism (with the template names used in this chapter):

```
template<typename T>
class CopierInterface {
  public:
    virtual void copy_to(Array<T, SArray<T> >&) const;
};

template<typename T, typename X>
class Copier : public CopierBase<T> {
  public:
    Copier(X const &x): expr(x) {}
    virtual void copy_to(Array<T, SArray<T> >&) const {
        // implementation of assignment loop

    }
  private:
    X const &expr;
};

template<typename T, typename Rep = SArray<T> >
class Array {
  public:
    // delegated assignment operator
    Array<T, Rep>& operator=(CopierBase<T> const &b) {
        b.copy_to(rep);
    };

};
template<typename T, typename A1, typename A2>
class A_mult {
  public:
    operator Copier<T, A_Mult<T, A1, A2> >();

};
```

This adds another level of complexity and some additional run-time cost to expression templates, but even so the resulting performance benefits were impressive at the time.

The C++ standard library contains a class template valarray that was meant to be used for applications that would justify the techniques used for the Array template developed in this chapter. A precursor of valarray had been designed with the intention that compilers aiming at the market for scientific computation would recognize the array type and use highly optimized internal code for their operations. Such compilers would have "understood" the types in some sense. However, this never happened (in part because the market in question is relatively small and in part because the problem grew in complexity as valarray became a template). Some time after the expression template technique was discovered, one of us (Vandevoorde) submitted to the C++ committee a proposal that turned valarray essentially into the Array template we developed (with many bells and whistles inspired by the existing valarray functionality). The proposal was the first time that the concept of the Rep parameter was documented. Prior to this, the arrays with actual storage and the expression template pseudo-arrays were different templates. When client code introduced a function foo() accepting an array—for example

Part IV: Advanced Applications

Templates can be used to develop elaborate libraries of elements that connect in seamless ways. Nontemplate libraries can often do such things too. However, when it comes to small, fairly simple utilities that make everyday programming easier, traditional procedural or object-oriented libraries are not always viable because the overhead needed to invoke the simple functionality is disproportionate to the facility offered. The C preprocessor allows some of these "simple needs" to be addressed, but often it is not quite adequate for the tasks at hand.

In this part we explore some small stand-alone utilities for which templates are an ideal means of implementation:

- A framework for type classification
- Smart Pointers
- Tuples
- Functors

Our goal is to demonstrate the techniques discussed earlier. We combine them and modify them to create genuinely useful software components. However, our main topic is still C++ Templates and not (for example) the development of a complete C++ library. We hope the code we present is a useful tutorial and source of inspiration for C++ library writers, but we don't claim that it is the best choice for off-the-shelf components.

Chapter 19. Type Classification

It is sometimes useful to be able to know whether a template parameter is a built-in type, a pointer type, or a class type, and so forth. In the following sections we develop a general-purpose type template that allows us to determine various properties of a given type. As a result we will be able to write code like the following:

```
if (TypeT<T>::IsPtrT) {

}
else if (TypeT<T>::IsClassT) {

}
```

Furthermore, expressions such as TypeT<T>::IsPtrT will be Boolean constants that are valid nontype template arguments. In turn, this allows the construction of more sophisticated and more powerful templates that specialize their behavior on the properties of their type arguments.

19.1 Determining Fundamental Types

To start, let's develop a template to determine whether a type is a fundamental type. By default, we assume a type is not fundamental, and we specialize the template for the fundamental cases:

```
// types/type1.hpp

// primary template: in general T is no fundamental type
template <typename T>
class IsFundaT {
  public:
    enum{ Yes = 0, No = 1};
};

// macro to specialize for fundamental types
#define MK_FUNDA_TYPE(T)                           \
    template<> class IsFundaT<T> {                 \
      public:                                      \
        enum { Yes = 1, No = 0 };                  \
    };

MK_FUNDA_TYPE(void)

MK_FUNDA_TYPE(bool)
MK_FUNDA_TYPE(char)
MK_FUNDA_TYPE(signed char)
MK_FUNDA_TYPE(unsigned char)
MK_FUNDA_TYPE(wchar_t)

MK_FUNDA_TYPE(signed short)
MK_FUNDA_TYPE(unsigned short)
MK_FUNDA_TYPE(signed int)
MK_FUNDA_TYPE(unsigned int)
MK_FUNDA_TYPE(signed long)
MK_FUNDA_TYPE(unsigned long)
#if LONGLONG_EXISTS
  MK_FUNDA_TYPE(signed long long)
  MK_FUNDA_TYPE(unsigned long long)
#endif  // LONGLONG_EXISTS

MK_FUNDA_TYPE(float)
MK_FUNDA_TYPE(double)
MK_FUNDA_TYPE(long double)

#undef MK_FUNDA_TYPE
```

The primary template defines the general case. That is, in general, IsFundaT<T >::Yes will yield 0 (or false):

```
template <typename T>
class IsFundaT {
  public:
    enum{ Yes = 0, No = 1 };
};
```

For each fundamental type a specialization is defined so that IsFundaT<T >::Yes will yield 1 (or true). This is done by defining a macro that expands the necessary code. For example,

```
MK_FUNDA_TYPE(bool)
```

expands to the following:

```
template<> class IsFundaT<bool> {
```

19.2 Determining Compound Types

Compound types are types constructed from other types. Simple compound types include plain types, pointer types, reference types, and even array types. They are constructed from a single base type. Class types and function types are also compound types, but their composition can involve multiple types (for parameters or members). Simple compound types can be classified using partial specialization. We start with a generic definition of a traits class describing compound types other than class types and enumeration types (the latter are treated separately):

```
// types/type2.hpp

template<typename T>
class CompoundT {           // primary template
  public:
    enum { IsPtrT = 0, IsRefT = 0, IsArrayT = 0,
           IsFuncT = 0, IsPtrMemT = 0 };
    typedef T BaseT;
    typedef T BottomT;
    typedef CompoundT<void> ClassT;
};
```

The member type BaseT is a synonym for the immediate type on which the template parameter type T builds. BottomT, on the other hand, refers to the ultimate nonpointer, nonreference, and nonarray type on which T is built. For example, if T is int**, then BaseT would be int*, and BottomT would be int. For pointer-to-member types, BaseT is the type of the member, and ClassT is the class to which the member belongs. For example, if T is a pointer-to-member function of type int(X::*)(), then BaseT is the function type int(), and ClassT is X. If T is not a pointer-to-member type, the ClassT is CompoundT<void> (an arbitrary choice; you might prefer a nonclass).

Partial specializations for pointers and references are fairly straightforward:

```
// types/type3.hpp

template<typename T>
class CompoundT<T&> {       // partial specialization for references
  public:
    enum { IsPtrT = 0, IsRefT = 1, IsArrayT = 0,
           IsFuncT = 0, IsPtrMemT = 0 };
    typedef T BaseT;
    typedef typename CompoundT<T>::BottomT BottomT;
    typedef CompoundT<void> ClassT;
};

template<typename T>
class CompoundT<T*> {       // partial specialization for pointers
  public:
    enum { IsPtrT = 1, IsRefT = 0, IsArrayT = 0,
           IsFuncT = 0, IsPtrMemT = 0 };
    typedef T BaseT;
    typedef typename CompoundT<T>::BottomT BottomT;
    typedef CompoundT<void> ClassT;
};
```

Arrays and pointers to members can be treated using the same technique, but it may come as a surprise that the partial specializations involve more template parameters than the primary template:

```
// types/type4.hpp

#include <stddef.h>

template<typename T, size_t N>
```

19.3 Identifying Function Types

The problem with function types is that because of the arbitrary number of parameters, there isn't a finite syntactic construct using template parameters that describes them all. One approach to resolve this problem is to provide partial specializations for functions with a template argument list that is shorter than a chosen limit. The first few such partial specializations can be defined as follows:

```
// types/type5.hpp

template<typename R>
class CompoundT<R()> {
  public:
    enum { IsPtrT = 0, IsRefT = 0, IsArrayT = 0,
           IsFuncT = 1, IsPtrMemT = 0 };
    typedef R BaseT();
    typedef R BottomT();
    typedef CompoundT<void> ClassT;
};

template<typename R, typename P1>
class CompoundT<R(P1)> {
  public:
    enum { IsPtrT = 0, IsRefT = 0, IsArrayT = 0,
           IsFuncT = 1, IsPtrMemT = 0 };
    typedef R BaseT(P1);
    typedef R BottomT(P1);
    typedef CompoundT<void> ClassT;
};

template<typename R, typename P1>
class CompoundT<R(P1, ...)> {
  public:
    enum { IsPtrT = 0, IsRefT = 0, IsArrayT = 0,
           IsFuncT = 1, IsPtrMemT = 0 };
    typedef R BaseT(P1);
    typedef R BottomT(P1);
    typedef CompoundT<void> ClassT;
};
```

This approach has the advantage that we can create typedef members for each parameter type.

A more general technique uses the SFINAE (substitution-failure-is-not-an-error) principle of Section 8.3.1 on page 106: An overloaded function template can be followed by explicit template arguments that are invalid for some of the templates. This can be combined with the approach used for the classification of enumeration types using overload resolution. The key to exploit SFINAE is to find a type construct that is invalid for function types but not for other types, or vice versa. Because we are already able to recognize various type categories, we can also exclude them from consideration. Therefore, one construct that is useful is the array type. Its elements cannot be void, references, or functions. This inspires the following code:

```
template<typename T>
class IsFunctionT {
  private:
    typedef char One;
    typedef struct { char a[2]; } Two;
    template<typename U> static One test(...);
    template<typename U> static Two test(U (*)[1]);
  public:
    enum { Yes = sizeof(IsFunctionT<T>::test<T>(0)) == 1 };
    enum { No = !Yes };
```

19.4 Enumeration Classification with Overload Resolution

Overload resolution is the process that selects among various functions with a same name based on the types of their arguments. As shown shortly, we can determine the outcome of a case of overload resolution without actually evaluating a function call. This is useful to test whether a particular implicit conversion exists. The implicit conversion that interests us particularly is the conversion from an enumeration type to an integral type: It allows us to identify enumeration types.

Explanations follow the complete implementation of this technique:

```
// types/type7.hpp

struct SizeOverOne { char c[2]; };

template<typename T,
         bool convert_possible = !CompoundT<T>::IsFuncT &&
                                 !CompoundT<T>::IsArrayT>
class ConsumeUDC {
  public:
    operator T() const;
};

// conversion to function types is not possible
template <typename T>
class ConsumeUDC<T, false> {
};

// conversion to void type is not possible
template <bool convert_possible>
class ConsumeUDC<void, convert_possible> {
};

char enum_check(bool);
char enum_check(char);
char enum_check(signed char);
char enum_check(unsigned char);
char enum_check(wchar_t);

char enum_check(signed short);
char enum_check(unsigned short);
char enum_check(signed int);
char enum_check(unsigned int);
char enum_check(signed long);
char enum_check(unsigned long);
#if LONGLONG_EXISTS
  char enum_check(signed long long);
  char enum_check(unsigned long long);
#endif  // LONGLONG_EXISTS

// avoid accidental conversions from float to int
char enum_check(float);
char enum_check(double);
char enum_check(long double);

SizeOverOne enum_check(...);    // catch all
template<typename T>
class IsEnumT {
  public:
    enum { Yes = IsFundaT<T>::No &&
                 !CompoundT<T>::IsRefT &&
                 !CompoundT<T>::IsPtrT &&
                 !CompoundT<T>::IsPtrMemT &&
```

19.5 Determining Class Types

With all the classification templates described in the previous section, only class types (classes, structs, and unions) remain to be recognized. One approach is to use the SFINAE principle as demonstrated in Section 15.2.2 on page 266.

Another approach is to proceed by elimination: If a type is not a fundamental type, not an enumeration type, and not a compound type, it must be a class type. The following straightforward template implements this idea:

```
// types/type8.hpp

template<typename T>
class IsClassT {
  public:
    enum { Yes = IsFundaT<T>::No &&
                 IsEnumT<T>::No &&
                 !CompoundT<T>::IsPtrT &&
                 !CompoundT<T>::IsRefT &&
                 !CompoundT<T>::IsArrayT &&
                 !CompoundT<T>::IsPtrMemT &&
                 !CompoundT<T>::IsFuncT };
    enum { No = !Yes };
};
```

19.6 Putting It All Together

Now that we are able to classify any type according to its kind, it is convenient to group all the classifying templates in a single general-purpose template. The following relatively small header file does just that:

```
// types/typet.hpp

#ifndef TYPET_HPP
#define TYPET_HPP

// define IsFundaT<>
#include "type1.hpp"

// define primary template CompoundT<> (first version)
//#include "type2.hpp"

// define primary template CompoundT<> (second version)
#include "type6.hpp"

// define CompoundT<> specializations
#include "type3.hpp"
#include "type4.hpp"
#include "type5.hpp"

// define IsEnumT<>
#include "type7.hpp"

// define IsClassT<>
#include "type8.hpp"

// define template that handles all in one style
template <typename T>
class TypeT {
  public:
    enum { IsFundaT  = IsFundaT<T>::Yes,
           IsPtrT    = CompoundT<T>::IsPtrT,
           IsRefT    = CompoundT<T>::IsRefT,
           IsArrayT  = CompoundT<T>::IsArrayT,
           IsFuncT   = CompoundT<T>::IsFuncT,
           IsPtrMemT = CompoundT<T>::IsPtrMemT,
           IsEnumT   = IsEnumT<T>::Yes,
           IsClassT  = IsClassT<T>::Yes };
};

#endif // TYPET_HPP
```

The following program shows an application of all these classification templates:

```
// types/types.cpp

#include "typet.hpp"
#include <iostream>

class MyClass {
};

void myfunc()
{
}

enum E { e1 };

// check by passing type as template argument
```

19.7 Afternotes

The ability for a program to inspect its own high-level properties (such as its type structures) is sometimes called reflection. Our framework therefore implements a form of compile-time reflection, which turns out to be a powerful ally to metaprogramming (see Chapter 17).

The idea of storing properties of types as members of template specializations dates back to at least the mid-1990s. Among the earlier serious applications of type classification templates was the __type_traits utility in the STL implementation distributed by SGI (then known as Silicon Graphics). The SGI template was meant to represent some properties of its template argument (for example, whether it was a POD type or whether its destructor was trivial). This information was then used to optimize certain STL algorithms for the given type. An interesting feature of the SGI solution was that some SGI compilers recognized the __type_traits specializations and provided information about the arguments that could not be derived using standard techniques. (The generic implementation of the __type_traits template was safe to use, albeit suboptimal.)

The use of the SFINAE principle for type classification purposes had been noted when the SFINAE principle was clarified during the standardization effort. However, it was never formally documented, and as a result much effort was later spent trying to recreate some of the techniques described in this chapter. One of the notable early contributions was by Andrei Alexandrescu who made popular the use of the sizeof operator to determine the outcome of overload resolution.

Finally, we should note that a rather complete type classification template has been incorporated in the Boost library (see [BoostTypeTraits]). In turn, this implementation is the basis of an effort to add such a facility to the standard library. See also Section 13.10 on page 218 for a related language extension.

Chapter 20. Smart Pointers

Memory is a resource that is normally explicitly managed in C++ programs. This management involves the acquisition and disposal of blocks of raw memory.

One of the more delicate issues in managing dynamically allocated memory is the decision of when to deallocate it. Among the various tools to simplify this aspect of programming are so-called smart pointer templates. In C++, smart pointers are classes that behave somewhat like ordinary pointers (in that they provide the dereferencing operators -> and *) but in addition encapsulate some memory or resource management policy.

In this chapter we develop smart pointer templates that encapsulate two different ownership models—exclusive and shared:

- Exclusive ownership can be enforced with little overhead, compared with handling raw pointers. Smart pointers that enforce such a policy are useful to deal with exceptions thrown while manipulating dynamically allocated objects.

- Shared ownership can sometimes lead to excessively complicated object lifetime situations. In such cases, it may be advisable to move the burden of the lifetime decisions from the programmer to the program.

The term smart pointer implies that objects are being pointed to. Alternatives for function pointers are subject to different issues, some of which are discussed in Chapter 22.

20.1 Holders and Trules

This section introduces two smart pointer types: a holder type to hold an object exclusively and a so-called trule to enable the transfer of ownership from one holder to another.

20.1.1 Protecting Against Exceptions

Exceptions were introduced in C++ to improve the reliability of C++ programs. They allow regular and exceptional execution paths to be more cleanly separated. Yet shortly after exceptions were introduced, various C++ programming authors and columnists started observing that a naive use of exceptions leads to trouble, and particularly to memory leaks. The following example shows but one of the many troublesome situations that could arise:

```
void do_something()
{
    Something* ptr = new Something;

    // perform some computation with *ptr
    ptr->perform();


    delete ptr;
}
```

This function creates an object with new, performs some operations with this object, and destroys the object at the end of the function with delete. Unfortunately, if something goes wrong after the creation but before the deletion of the object and an exception gets thrown, the object is not deallocated and the program leaks memory. Other problems may arise because the destructor is not called (for example, buffers may not be written out to disk, network connections may not be released, on-screen windows may not be closed, and so forth). This particular case can be handled fairly easily using an explicit exception handler:

```
void do_something()
{
    Something* ptr = 0;
    try {
        ptr = new Something;

        // perform some computation with *ptr
        ptr->perform();

    }
    catch (...) {
        delete ptr;
        throw;  // rethrow the exception that was caught
    }
    return result;
}
```

This is manageable, but already we find that the exceptional path is starting to dominate the regular path, and the deletion of the object has to be done in two different places: once in the regular path and once in the exceptional path. This avenue quickly grows worse. Consider what happens if we need to create two objects in a single function:

```
void do_two_things()
{
    Something* first = new Something;
    first->perform();

    Something* second = new Something;
    second->perform();
```

20.2 Reference Counting

The Holder template (and its Trule helper) works well to hold allocated structures temporarily so that they will be deallocated if an exception causes the local stack frame to be unwound. However, memory leaks can also occur in other contexts, and in particular when many objects are interconnected in complex structures.

A general rule about the management of dynamically allocated objects is easily stated: If nothing in an application points to a dynamically allocated object, that object should be destroyed and its storage should be made available for reuse. It is therefore not surprising that programmers everywhere have been looking for ways to automate such a policy. The challenge is to determine that nothing is pointing to an object.

One idea that has been implemented many times over is so-called reference counting: For each object that is pointed to, keep a count of the number of pointers to it, and when that count drops to zero, delete the object. For this to be feasible in C++, we need to adhere to some convention. Specifically, because it is not practical to track how ordinary pointers to an object are created, copied, and destroyed, it is common to require that the only "pointers" to a reference-counted object are a specific kind of smart pointer. In this section we discuss the implementation of such a reference-counting smart pointer. This pointer is a template whose main parameter is the type of the object to which it points:

```
template <typename T   >
class CountingPtr {
  public:
    // a constructor that starts a new count for the object
    // pointed to by T:
    explicit CountingPtr (T*);

    // copying increases the count:
    CountingPtr (CountingPtr<T  > const&);

    // destruction decreases the count:
    inline ~CountingPtr();

    // assignment decreases the count for the object previously
    // pointed to and increases it for the new object pointed to
    // (but beware of self-assignment):
    CountingPtr<T  >& operator= (CountingPtr<T  > const&);

    // the operators that make this a smart pointer:
    inline T& operator* ();
    inline T* operator-> ();

};
```

The parameter T is the only parameter that is truly needed to build a functional counting pointer template. Indeed, a good case can be made in favor of keeping a basic template like this as simple and reliable as possible. Nonetheless, we choose to use CountingPtr to demonstrate policy parameters (a concept described in detail in Chapter 15).

The comments in the code explain the general approach to reference counting: Every construction, destruction, and assignment of a CountingPtr may potentially change the reference counts (when one of the counts drops to zero, the object pointed to is deleted).

20.2.1 Where Is the Counter?

Because our idea is to count the number of pointers to an object, it would be entirely logical to place the counter in

20.3 Afternotes

Smart pointer templates are probably the second-most obvious application of templates after container templates; however, the details are far from obvious, as this chapter illustrates. Indeed, many authors cover the topic in some detail. Good material supplementing our discussion can be found in [MeyersMoreEffective], which offers a more basic discussion, and in [AlexandrescuDesign], which describes a complete, policy-based design of a family of smart pointers.

The C++ standard library contains a smart pointer template auto_ptr. It is intended for the same use as our Holder/Trule pair of templates, but avoids the use of a second template by exploiting a controversial piece of the C++ overloading rules in the context of variable initialization. [5]

[5] An explanation of the mechanisms involved is well beyond the scope of this text (and not really related to templates). The controversy arises because one of the mechanisms on which auto_ptr relies is considered by some to be a defect in the C++ standard. See [JosuttisAutoPtr] for additional discussion on this topic.

Other smart pointers were proposed for inclusion in the C++ standard library, but the C++ standardization committee decided not to support them.

The Boost project offers a library containing a variety of smart pointer classes to meet a variety of needs (see [BoostSmartPtr]).

Chapter 21. Tuples

Throughout this book we often use homogeneous containers and array-like types to illustrate the power of templates. Such homogeneous structures extend the concept of a C/C++ array and are pervasive in most applications. C++ (and C) also has a nonhomogeneous containment facility: the class (or struct). Tuples are class templates that similarly allow us to aggregate objects of differing types. We start with the duo—an entity analogous to the standard std::pair template—but we also show how it can be nested to assemble an arbitrary number of members, thereby forming trios, quartets, and so forth. [1]

[1] The number is not entirely arbitrary because there exists an implementation-dependent limit on the depth of template nesting.

21.1 Duos

A duo is the assembly of two objects into a single type. This is similar to the std::pair class template in the standard library, but because we will add slightly different functionality to this very basic utility, we opted for a name other than pair to avoid confusion with the standard item. At its very simplest, we can define Duo as follows:

```
template <typename T1, typename T2>
struct Duo {
    T1 v1;   // value of first field
    T2 v2;   // value of second field
};
```

This can, for example, be useful as a return type for a function that may return an invalid result:

```
Duo<bool,X> result = foo();
if (result.v1) {
     // result is valid; value is in result.v2

}
```

Many other applications are possible.

The benefit of Duo as defined here is not insignificant, but it is rather small. After all, it would not be that much work to define a structure with two fields, and doing so allows us to choose meaningful names for these fields. However, we can extend the basic facility in a few ways to add to the convenience. First, we can add constructors:

```
template <typename T1, typename T2>
class Duo {
  public:
    T1 v1;   // value of first field
    T2 v2;   // value of second field

    // constructors
    Duo() : v1(), v2() {
    }
    Duo (T1 const& a, T2 const& b)
     : v1(a), v2(b) {
    }
};
```

Note that we used an initializer list for the default constructor so that the members get zero initialized for built-in types (see Section 5.5 on page 56).

To avoid the need for explicit type parameters, we can further add a function so that the field types can be deduced:

```
template <typename T1, typename T2>
inline
Duo<T1,T2> make_duo (T1 const& a, T2 const& b)
{
    return Duo<T1,T2>(a,b);
}
```

Now the creation and initialization of a Duo becomes more convenient. Instead of

```
Duo<bool,int> result;
result.v1 = true;
result.v2 = 42;
return result;
```

21.2 Recursive Duos

Consider the following object definition:

```
Duo<int, Duo<char, Duo<bool, double> > > q4;
```

The type of q4 is a so-called recursive duo. It is a type instantiated from the Duo template, and the second type argument is itself a Duo as well. We could also use recursion of the first parameter, but in the remainder of this discussion, recursive duo refers only to Duos with a second template argument that is instantiated from the Duo template.

21.2.1 Number of Fields

It's relatively straightforward to count that q4 collects four values of types int, char, bool, and double respectively. To facilitate the formal counting of the number of fields, we can further partially specialize the Duo template:

```
// tuples/duo2.hpp

template <typename A, typename B, typename C>
class Duo<A, Duo<B,C> > {
  public:
    typedef A        T1;            // type of first field
    typedef Duo<B,C> T2;            // type of second field
    enum { N = Duo<B,C>::N + 1 };  // number of fields

  private:
    T1 value1;                      // value of first field
    T2 value2;                      // value of second field

  public:
    // the other public members are unchanged

};
```

For completeness, let's provide a partial specialization of Duo so that it can degenerate into a nonhomogeneous container holding just one field:

```
// tuples/duo6.hpp

// partial specialization for Duo<> with only one field
template <typename A>
struct Duo<A,void> {
  public:
    typedef A T1;     // type of first field
    typedef void T2;  // type of second field
    enum { N = 1 };   // number of fields

  private:
    T1 value1;        // value of first field

  public:
    // constructors
    Duo() : value1() {
    }
    Duo (T1 const & a)
     : value1(a) {
    }

    // field access
    T1& v1() {
```

21.3 Tuple Construction

The nested structure of recursive duos is convenient to apply template metaprogramming techniques to them. However, for a human programmer it is more pleasing to have a flat interface to this structure. To obtain this, we can define a recursive Tuple template with many parameters and have it be a derivation from a recursive duo type of appropriate size. We show the code here for tuples up to five fields, but it is not significantly harder to provide for a dozen fields or so. You can find the code in tuples/tuple1.hpp.

To allow for tuples of varying sizes, we have unused type parameters that default to a null type, NullT, which we define as a placeholder for that purpose. We use NullT rather than void because we will create parameters of that type (void cannot be a parameter type):

```
// type that represents unused type parameters
class NullT {
};
```

Tuple is defined as a template that derives from a Duo having one more type parameter with NullT defined:

```
// Tuple<> in general derives from Tuple<> with one more NullT
template<typename P1,
         typename P2 = NullT,
         typename P3 = NullT,
         typename P4 = NullT,
         typename P5 = NullT>
class Tuple
 : public Duo<P1, typename Tuple<P2,P3,P4,P5,NullT>::BaseT> {
  public:
    typedef Duo<P1, typename Tuple<P2,P3,P4,P5,NullT>::BaseT>
            BaseT;

    // constructors:
    Tuple() {}
    Tuple(TypeOp<P1>::RefConstT a1,
          TypeOp<P2>::RefConstT a2,
          TypeOp<P3>::RefConstT a3 = NullT(),
          TypeOp<P4>::RefConstT a4 = NullT(),
          TypeOp<P5>::RefConstT a5 = NullT())
     : BaseT(a1, Tuple<P2,P3,P4,P5,NullT>(a2,a3,a4,a5)) {
    }
};
```

Note the shifting pattern when passing the parameters to the recursive step. Because we derive from a base type that defines member types T1 and T2, we used template parameter names of the form Pn instead of the usual Tn. [2]

[2] A very curious lookup rule in C++ prefers names inherited from nondependent base classes over template parameter names. This should not be a problem in this case because the base class is dependent, but some compilers still get this wrong at the time of this writing.

We need a partial specialization to end this recursion with the derivation from a nonrecursive duo:

```
// specialization to end deriving recursion
template <typename P1, typename P2>
class Tuple<P1,P2,NullT,NullT,NullT> : public Duo<P1,P2> {
  public:
    typedef Duo<P1,P2> BaseT;
    Tuple() {}
    Tuple(TypeOp<P1>::RefConstT a1,
          TypeOp<P2>::RefConstT a2,
```

21.4 Afternotes

Tuple construction is one of those template applications that appears to have been independently attempted by many programmers. The details of these attempts vary widely, but many are based on the idea of a recursive pair structure (such as our recursive duos). One interesting alternative was developed by Andrei Alexandrescu in [AlexandrescuDesign]. He cleanly separates the list of types from the list of fields in the tuple. This leads to the concept of a type list that has various applications of its own (one of which is the construction of a tuple with the encapsulated types).

Section 13.13 on page 222 discusses the concept of template list parameters, which are a language extension that makes the implementation of tuples almost trivial.

Chapter 22. Function Objects and Callbacks

A function object (also called a functor) is any object that can be called using the function call syntax. In the C programming language, three kinds of entities can lead to syntax that looks like a function call: functions, function-like macros, and pointers to functions. Because functions and macros are not objects, this implies that only pointers to functions are available as functors in C. In C++, additional possibilities are added: The function call operator can be overloaded for class types, a concept of references to functions exists, and member functions and pointer-to-member functions have a call syntax of their own. Not all of these concepts are equally useful, but the combination of the concept of a functor with the compile-time parameterization offered by templates leads to powerful programming techniques.

Besides developing functor types, this chapter also delves into some usage idioms for functors. Nearly all uses end up being a form of callback: The client of a library wants that library to call back some function of the client code. The classic example is a sorting routine that needs a function to compare two elements in the set being sorted. The comparison routine is passed as a functor in this case. Traditionally, the term callback has been reserved for functors that are passed as function call arguments (as opposed to, for example, template arguments), and we maintain this tradition.

The terms function object and functor are unfortunately a little fuzzy in the sense that different members of the C++ programming community may give slightly different meanings to these terms. A common variation of the definition we have given is to include only objects of class types in the functor or function object concept; function pointers are then excluded. In addition, it is not uncommon to read or hear discussions referring to the class type of a function object as a "function object." In other words, the phrase "class of function objects so and so " is shortened to "function objects so and so ." Although we sometimes handle this terminology somewhat sloppily in our own daily work, we have made it a point to stick to our initial definitions in this chapter.

Before digging into the use of templates to implement useful functors, we discuss some properties of function calls that motivate some of the advantages of template-based functors.

22.1 Direct, Indirect, and Inline Calls

Typically, when a C or C++ compiler encounters the definition of a noninline function, it generates and stores machine code for that function in an object file. It also creates a name associated with the machine code; in C, this name is typically the function name itself, but in C++ the name is usually extended with an encoding of the parameter types to allow for unique names even when a function is overloaded (the resulting name is usually called a mangled name, although the term decorated name is also used). Similarly, when the compiler encounters a call site like

```
f();
```

it generates machine code for a call to a function of that type. For most machine languages, the call instruction itself necessitates the starting address of the routine. This address can be part of the instruction (in which case the instruction is called a direct call), or it may reside somewhere in memory or in a machine register (indirect call). Almost all modern computer architectures provide both types of routine calling instructions, but (for reasons that are beyond the scope of this book) direct calls are executed more efficiently than indirect calls. In fact, as computer architectures get more sophisticated, it appears that the performance gap between direct calls and indirect calls increases. Hence, compilers generally attempt to generate a direct call instruction when possible.

In general, a compiler does not know at which address a function is located (the function could, for example, be in another translation unit). However, if the compiler knows the name of the function, it generates a direct call instruction with a dummy address. In addition, it generates an entry in the generated object file directing the linker to update that instruction to point to the address of a function with the given name. Because the linker sees the object files created from all the translation units, it knows the call sites as well as the definition sites and hence is able to patch up all the direct call sites. [1]

[1] The linker performs a similar role for accesses to namespace scope variables, for example.

Unfortunately, when the name of the function is not available, an indirect call must be used. This is usually the case for calls through pointers to functions:

```
void foo (void (*pf)())
{
    pf();  // indirect call through pointer to function pf
}
```

In this example it is, in general, not possible for a compiler to know to which function the parameter pf points (after all, it is most likely different for a different invocation of foo()). Hence, the technique of having the linker match names does not work. The call destination is not known until the code is actually executed.

Although a modern computer can often execute a direct call instruction about as quickly as other common instructions (for example, an instruction to add two integers), function calls can still be a serious performance impediment. The following example shows this:

```
int f1(int const & r)
{
    return ++(int&)r;   // not reasonable, but legal
}

int f2(int const & r)
{
    return r;
}
```

22.2 Pointers and References to Functions

Consider the following fairly trivial definition of a function foo():

```
extern "C++" void foo() throw()
{
}
```

The type of this function ought to be "function with C++ linkage that takes no arguments, returns no value, and does not throw any exceptions." For historical reasons, the formal definition of the C++ language does not actually make the exception specification part of a function type. [3] However, that may change in the future. It is a good idea to make sure that when you create code in which function types must match, the exception specifications also match. Name linkage (usually for "C" and "C++") is properly a part of the type system, but some C++ implementations are a little lax in enforcing it. Specifically, they allow a pointer to a function with C linkage to be assigned to a pointer to a function with C++ linkage and vice versa. This is a consequence of the fact that, on most platforms, calling conventions for C and C++ functions are identical as far as the common subset of parameter and return types is concerned.

[3] The historical origin of this is not clear, and the C++ standard is somewhat inconsistent in this area.

In most contexts, the expression foo undergoes an implicit conversion to a pointer to the function foo(). Note that foo itself does not denote the pointer, just as the expression ia after the declaration

```
int ia[10];
```

does not denote a pointer to the array (or to the first element of the array). The implicit conversion from a function (or array) to a pointer is often called decay. To illustrate this, we can write the following complete C++ program:

```
// functors/funcptr.cpp

#include <iostream>
#include <typeinfo>

void foo()
{
    std::cout << "foo() called" << std::endl;
}

typedef void FooT();  // FooT is a function type,
                      // the same type as that of function foo()
int main()
{
    foo();           // direct call

    // print types of foo and FooT
    std::cout << "Types of foo:  " << typeid(foo).name()
              << '\n';
    std::cout << "Types of FooT: " << typeid(FooT).name()
              << '\n';

    FooT* pf = foo;  // implicit conversion (decay)
    pf();            // indirect call through pointer
    (*pf)();         // equivalent to pf()

    // print type of pf
    std::cout << "Types of pf:   " << typeid(pf).name()
              << '\n';

    FooT& rf = foo;  // no implicit conversion
```

22.3 Pointer-to-Member Functions

To understand why a distinction is made between pointers to ordinary functions and pointers to member functions, it is useful to study the typical C++ implementation of a call to a member function. Such a call could take the form p->mf() or a close variation of this syntax. Here, p is a pointer to an object or to a subobject. It is passed in some form as a hidden parameter to mf(), where it is known as the this pointer.

The member function mf() may have been defined for the subobject pointed to by p, or it may be inherited by the subobject. For example:

```
class B1 {
  private:
    int b1;
  public:
    void mf1();
};

void B1::mf1()
{
    std::cout << "b1="<<b1<<std::endl;
}
```

As a member function, mf1() expects to be called for an object of type B1. Thus, this refers to to an object of type B1.

Let's add some more code to this:

```
class B2 {
  private:
    int b2;
  public:
    void mf2();
};

void B1::mf2()
{
    std::cout << "b2="<<b2<<std::endl;
}
```

The member mf2() similarly expects the hidden parameter this to point to a B2 subobject.

Now let's derive a class from both B1 and B2:

```
class D: public B1, public B2 {
  private:
    int d;
};
```

With this declaration, an object of type D can behave as an object of type B1 or an object of type B2. For this to work, a D object contains both a B1 subobject and a B2 subobject. On nearly all 32-bit implementations we know of today, a D object will be organized as shown in Figure 22.1. That is, if the size of the int members is 4 bytes, member b1 has the address of this, member b2 has the address of this plus 4 bytes, and member d has the address of this plus 8 bytes. Note how the B1 subobject shares its origin with the origin of the D subobject, but the B2 subobject does not.

22.4 Class Type Functors

Although pointers to functions are functors directly available in the language, there are many situations in which it is advantageous to use a class type object with an overloaded function call operator. Doing so can lead to added flexibility, added performance, or both.

22.4.1 A First Example of Class Type Functors

Here is a very simple example of a class type functor:

```
// functors/functor1.cpp

#include <iostream>

// class for function objects that return constant value
class ConstantIntFunctor {
  private:
    int value;    // value to return on ''function call''
  public:
    // constructor: initialize value to return
    ConstantIntFunctor (int c) : value(c) {
    }

    // ''function call''
    int operator() () const {
        return value;
    }
};

// client function that uses the function object
void client (ConstantIntFunctor const& cif)
{
    std::cout << "calling back functor yields " << cif() << '\n';
}

int main()
{
    ConstantIntFunctor seven(7);
    ConstantIntFunctor fortytwo(42);
    client(seven);
    client(fortytwo);
}
```

ConstantIntFunctor is a class type from which functors can be generated. That is, if you create an object with

```
ConstantIntFunctor seven(7);  // create function object
```

the expression

```
seven();                      // call operator () for function object
```

is a call of operator () for the object seven rather than a call of function seven(). We achieve the same effect (indirectly) when passing the function objects seven and fortytwo through parameter cif to client().

This example illustrates what is in practice perhaps the most important advantage of class type functors over pointers to functions: the ability to associate some state (data) with the function. This is a fundamental improvement in capabilities for callback mechanisms. We can have multiple "instances" of a function with behavior that is (in a sense)

PREVIOUS NEXT

22.5 Specifying Functors

Our previous example of the standard set class shows only one way to handle the selection of functors. A number of different approaches are discussed in this section.

22.5.1 Functors as Template Type Arguments

One way to pass a functor is to make its type a template argument. A type by itself is not a functor, however, so the client function or class must create a functor object with the given type. This, of course, is possible only for class type functors, and it rules out function pointer types. A function pointer type does not by itself specify any behavior. Along the same lines of thought, this is not an appropriate mechanism to pass a class type functor that encapsulates some state information (because no particular state is encapsulated by the type alone; a specific object of that type is needed).

Here is an outline of a function template that takes a functor class type as a sorting criterion:

```
template <typename FO>
void my_sort (  )
{
    FO cmp;           // create function object

    if (cmp(x,y)) {  // use function object to compare two values

    }

}

// call function with functor
my_sort<std::less<  > > (  );
```

With this approach, the selection of the comparison code has become a compile-time affair. And because the comparison can be "inlined," a good optimizing compiler should be able to produce code that is essentially equivalent to replacing the functor calls by direct applications of the resulting operations. To be entirely perfect, an optimizer must also be able to elide the storage used by the cmp functor object. In practice, however, only a few compilers are capable of such features.

22.5.2 Functors as Function Call Arguments

Another way to pass functors is to pass them as function call arguments. This allows the caller to construct the function object (possibly using a nontrivial constructor) at run time.

The efficiency argument is essentially similar to that of having just a functor type parameter, except that we must now copy a functor object as it is passed into the routine. This cost is usually low and can in fact be reduced to zero if the functor object has no data members (which is often the case). Indeed, consider this variation of our my_sort example:

```
template <typename F>
void my_sort (  , F cmp)
{

    if (cmp(x,y)) {  // use function object to compare two values

    }
```

22.6 Introspection

In the context of programming, the term introspection refers to the ability of a program to inspect itself. For example, in Chapter 15 we designed templates that can inspect a type and determine what kind of type it is. For functors, it is often useful to be able to tell, for example, how many arguments the functor accepts, the return type of the functor, or the nth parameter type of the functor type.

Introspection is not easily achieved for an arbitrary functor. For example, how would we write a type function that evaluates to the type of the second parameter in a functor like the following?

```
class SuperFunc {
  public:
    void operator() (int, char**);
};
```

Some C++ compilers provide a special type function known as typeof. It evaluates to the type of its argument expression (but doesn't actually evaluate the expression, much like the sizeof operator). With such an operator, the previous problem can be solved to a large extent, albeit not easily. The typeof concept is discussed in Section 13.8 on page 215.

Alternatively, we can develop a functor framework that requires participating functors to provide some extra information to enable some level of introspection. This is the approach we use in the remainder of this chapter.

22.6.1 Analyzing a Functor Type

In our framework, we handle only class type functors [7] and require them to provide the following information:

[7] To reduce the strength of this constraint, we also develop a tool to encapsulate function pointers in the framework.

- The number of parameters of the functor (as a member enumerator constant NumParams)
- The type of each parameter (through member typedefs Param1T, Param2T, Param3T,...)
- The return type of the functor (through a member typedef ReturnT)

For example, we could rewrite our PersonSortCriterion as follows to fit this framework:

```
class PersonSortCriterion {
  public:
    enum { NumParams = 2 };
    typedef bool ReturnT;
    typedef Person const& Param1T;
    typedef Person const& Param2T;
    bool operator() (Person const& p1, Person const& p2) const {
        // returns whether p1 is ''less than'' p2
```

22.7 Function Object Composition

Let's assume we have the following two simple mathematical functors in our framework:

```
// functors/math1.hpp

#include <cmath>
#include <cstdlib>

class Abs {
  public:
    // ''function call'':
    double operator() (double v) const {
        return std::abs(v);
    }
};

class Sine {
  public:
    // ''function call'':
    double operator() (double a) const {
        return std::sin(a);
    }
};
```

However, the functor we really want is the one that computes the absolute value of the sine of a given angle. Writing the new functor is not hard:

```
class AbsSine {
  public:
    double operator() (double a) {
        return std::abs(std::sin(a));
    }
};
```

Nevertheless, it is inconvenient to write new declarations for every new combination of functors. Instead, we may prefer to write a functor utility that composes two other functors. In this section we develop some templates that enable us to do this. Along the way, we introduce various concepts that prove useful in the remainder of this chapter.

22.7.1 Simple Composition

Let's start with a first cut at an implementation of a composition tool:

```
// functors/compose1.hpp

template <typename FO1, typename FO2>
class Composer {
  private:
    FO1 fo1;  // first/inner function object to call
    FO2 fo2;  // second/outer function object to call
  public:
    // constructor: initialize function objects
    Composer (FO1 f1, FO2 f2)
     : fo1(f1), fo2(f2) {
    }

    // ''function call'': nested call of function objects
    double operator() (double v) {
        return fo2(fo1(v));
    }
```

22.8 Value Binders

Often, a functor with multiple parameters remains useful when one of the parameters is bound to a specific value. For example, a simple Min functor template such as

```
// functors/min.hpp

template <typename T>
class Min {
  public:
    typedef T ReturnT;
    typedef T Param1T;
    typedef T Param2T;
    enum { NumParams = 2 };
    ReturnT operator() (Param1T a, Param2T b) {
        return a<b ? b : a;
    }
};
```

can be used to build a new Clamp functor that behaves like Min with one of its parameters bound to a certain constant. The constant could be specified as a template argument or as a run-time argument. For example, we can write the new functor as follows:

```
// functors/clamp.hpp

template <typename T, T max_result>
class Clamp : private Min<T> {
  public:
    typedef T ReturnT;
    typedef T Param1T;
    enum { NumParams = 1 };
    ReturnT operator() (Param1T a) {
        return Min<T>::operator() (a, max_result);
    }
};
```

As with composition, it is very convenient to have some template that automates the task of binding a functor parameter available, even though it doesn't take very much code to do so manually.

22.8.1 Selecting the Binding

A binder binds a particular parameter of a particular functor to a particular value. Each of these aspects can be selected at run time (using function call arguments) or at compile time (using template arguments).

For example, the following template selects everything statically (that is, at compile time):

```
template<typename F, int P, int V>
class BindIntStatically;
    // F is the functor type
    // P is the parameter to bind
    // V is the value to be bound
```

Each of the three binding aspects (functor, bound parameter, and bound value) can instead be selected dynamically with various degrees of convenience.

Perhaps the least convenient is to make the selection of which parameter to bind dynamic. Presumably this would

Functor Operations: A Complete Implementation

To illustrate the overall effect achieved by our sophisticated treatment of functor composition and value binding, we provide here a complete implementation of these operations for functors with up to three parameters. (It is straightforward to extend this to a dozen parameters or so, but we prefer to keep the printed code relatively concise.)

Let's first look at some sample client code:

```
// functors/functorops.cpp

#include <iostream>
#include <string>
#include <typeinfo>
#include "functorops.hpp"

bool compare (std::string debugstr, double v1, float v2)
{
    if (debugstr != "") {
        std::cout << debugstr << ": " << v1
                                      << (v1<v2? '<' : '>')
                                      << v2 << '\n';
    }
    return v1<v2;
}

void print_name_value (std::string name, double value)
{
    std::cout << name << ": " << value << '\n';
}

double sub (double a, double b)
{
    return a-b;
}

double twice (double a)
{
    return 2*a;
}
int main()
{
    using std::cout;

    // demonstrate composition:
    cout << "Composition result: "
         << compose(func_ptr(sub), func_ptr(twice))(3.0, 7.0)
         << '\n';

    // demonstrate binding:
    cout << "Binding result: "
         << bindfp<1>(compare, "main()->compare()")(1.02, 1.03)
         << '\n';
    cout << "Binding output: ";
    bindfp<1>(print_name_value,
              "the ultimate answer to life")(42);

    // combine composition and binding:
    cout << "Mixing composition and binding (bind<1>): "
         << bind<1>(compose(func_ptr(sub),func_ptr(twice)),
                    7.0)(3.0)
         << '\n';
    cout << "Mixing composition and binding (bind<2>): "
```

PREVIOUS NEXT

22.10 Afternotes

The STL part of the C++ standard library uses the concept of functors. For example, all algorithms use functors to customize their exact behavior. Many of these functors are so-called predicates. Predicates are functions or function objects that return a Boolean value (a value that is convertible to bool). The predicates, in general, should be pure functors; otherwise, unexpected results may occur (see Section 8.1.4 of [JosuttisStdLib]).

The C++ standard library also provides several standard functors and adapters for composition. In fact, for every common unary and binary operator a function object is provided. See Sections 8.2 and 8.3 of [JosuttisStdLib] for details. However, note that the C++ standard library does not provide enough adapters to support every functional behavior as a combination of function objects. For example, it is not possible to combine the results of two unary operations to formulate a criterion such as "this and that." The Boost repository of C++ libraries provides supplementary adapters that fill this gap (see [BoostCompose]).

Appendix A. The One-Definition Rule

Affectionately known as the ODR, the one-definition rule is a cornerstone for the well-formed structuring of C++ programs. The most common consequences of the ODR are simple enough to remember and apply: Define noninline functions exactly once across all files, and define classes and inline functions at most once per translation unit, making sure that all definitions for the same entity are identical.

However, the devil is in the details, and when combined with template instantiation, these details can be daunting. This appendix is meant to provide a comprehensive overview of the ODR for the interested reader. We also indicate when specific related issues are expounded on in the main text.

A.1 Translation Units

In practice we write C++ programs by filling files with "code." However, the boundary set by a file is not terribly important in the context of the ODR. Instead, what matters are so-called translation units. Essentially, a translation unit is the result of applying the preprocessor to a file you feed to your compiler. The preprocessor drops sections of code not selected by conditional compilation directives (#if, #ifdef, and friends), drops comments, inserts #included files (recursively), and expands macros.

Hence, as far as the ODR is concerned, having the following two files

```
// File header.hpp:
#ifdef DO_DEBUG
 #define debug(x) std::cout << x << '\n'
#else
 #define debug(x)
#endif

void debug_init();
// File myprog.cpp:
#include "header.hpp"

int main()
{
    debug_init();
    debug("main()");
}
```

is equivalent to the following single file:

```
// File myprog.cpp:
void debug_init();

int main()
{
    debug_init();
}
```

Connections across translation unit boundaries are established by having corresponding declarations with external linkage in two translation units (for example, two declarations of the global function debug_init()) or by argument-dependent lookup during the instantation of exported templates.

Note that the concept of a translation unit is a little more abstract than just "a preprocessed file." For example, if we were to feed a preprocessed file twice to a compiler to form a single program, it would bring into the program two distinct translation units (there is no point in doing so, however).

A.2 Declarations and Definitions

The terms declaration and definition are often used interchangeably in common "programmer talk." In the context of the ODR, however, the exact meaning of these words is important. [1]

[1] We also think it's a good habit to handle the terms carefully when exchanging ideas about C or C++. We do so throughout this book.

A declaration is a C++ construct that introduces or reintroduces a name in your program. A declaration can also be a definition, depending on which entity it introduces and how it introduces it:

- **Namespaces and namespace aliases:** The declarations of namespaces and their aliases are always also definitions, although the term definition is unusual in this context because the list of members of a namespace can be "extended" at a later time (unlike classes and enumeration types for example).

- **Classes, class templates, functions, function templates, member functions, and member function templates:** The declaration is a definition if and only if the declaration includes a brace-enclosed body associated with the name. This rule includes unions, operators, member operators, static member functions, constructors and destructors, and explicit specializations of template versions of such things (that is, any class-like and function-like entity).

- **Enumerations:** The declaration is a definition if and only if it includes the brace-enclosed list of enumerators.

- **Local variables and nonstatic data members:** These entities can always be treated as definitions, although the distinction rarely matters.

- **Global variables:** If the declaration is not directly preceded by a keyword extern or if it has an initializer, the declaration of a global variable is also a definition of that variable. Otherwise, it is not a definition.

- **Static data members:** The declaration is a definition if and only if it appears outside the class or class template of which it is a member.

- **Typedefs, using-declarations, and using-directives:** These are never definitions, although typedefs can be combined with class or union definitions.

- **Explicit instantiation directives:** We can consider them to be definitions.

PREVIOUS NEXT

A.3 The One-Definition Rule in Detail

As we implied in the introduction to this appendix, there are many details to the actual rule. We organize the rule's constraints by their scope.

A.3.1 One-per-Program Constraints

There can be at most one definition of the following items per program:

- Noninline functions and noninline member functions
- Variables with external linkage (essentially, variables declared in a namespace scope or in the global scope, and with the static specifier)
- Static data members
- Noninline function templates, noninline member function templates, and noninline members of class templates when they are declared with export
- Static data members of class templates when they are declared with export:

For example, a C++ program consisting of the following two translation units is invalid [2]:

[2] Interestingly, it is valid C because C has a concept of tentative definition, which is a variable definition without an initializer and can appear more than once in a program.

```
// Translation unit 1:
int counter;

// Translation unit 2:
int counter;                // ERROR: defined twice! (ODR violation)
```

This rule does not apply to entities with internal linkage (essentially, entities declared in an unnamed namespace scope or in the global scope using the static specifier) because even when two such entities have the same name, they are considered distinct. In the same vein, entities declared in unnamed namespaces are considered distinct if they appear in distinct translation units. For example, the following two translation units can be combined into a valid C++ program:

```
// Translation unit 1:
static counter = 2;     // unrelated to other translation units

namespace {
```

Appendix B. Overload Resolution

Overload resolution is the process that selects the function to call for a given call expression. Consider the following simple example:

```
void display_num(int);     // (1)
void display_num(double);  // (2)

int main()
{
    display_num(399);      // matches (1) better than (2)
    display_num(3.99);     // matches (2) better than (1)
}
```

In this example, the function name display_num() is said to be overloaded. When this name is used in a call, a C++ compiler must therefore distinguish between the various candidates using additional information; mostly, this information is the types of the call arguments. In our example it makes intuitive sense to call the int version when the function is called with an integer argument and the double version when a floating-point argument is provided. The formal process that attempts to model this intuitive choice is the overload resolution process.

The general ideas behind the rules that guide overload resolution are simple enough, but the details have become quite complex during the C++ standardization process. This complexity was driven mostly by the desire to support various real-world examples that intuitively (to a human) seem to have an "obviously best match," but when trying to formalize this intuition, various subtleties arose.

In this appendix we provide a reasonably detailed survey of the overload resolution rules. However, the complexity of this process is such that we do not claim to cover every part of the topic.

B.1 When Does Overload Resolution Kick In?

Overload resolution is just one part of the complete processing of a function call. In fact, it is not part of every function call. First, calls through function pointers and calls through pointers to member functions are not subject to overload resolution because the function to call is entirely determined (at run time) by the pointers. Second, function-like macros cannot be overloaded and are therefore not subject to overload resolution.

At a very high level, a call to a named function can be processed in the following way:

- The name is looked up to form an initial overload set.
- If necessary, this set is tweaked in various ways (for example, template deduction occurs).
- Any candidate that doesn't match the call at all (even after considering implicit conversions and default arguments) is eliminated from the overload set. This results in a set of so-called viable function candidates.
- Overload resolution is performed to find a best candidate. If there is one, it is selected; otherwise, the call is ambiguous.
- The selected candidate is checked. For example, if it is an inaccessible private member, a diagnostic is issued.

Each of these steps has its own subtleties, but overload resolution is arguably the most complex. Fortunately, a few simple principles clarify the majority of situations. We examine these principles next.

B.2 Simplified Overload Resolution

Overload resolution ranks the viable candidate functions by comparing how each argument of the call matches the corresponding parameter of the candidates. For one candidate to be considered better than another, the better candidate cannot have any of its parameters be a worse match than the corresponding parameter in the other candidate. The following example illustrates this:

```
void combine(int, double);
void combine(long, int);

int main()
{
    combine (1, 2);  // ambiguous!
}
```

In this example, the call to combine() is ambiguous because the first candidate matches the first argument (the literal 1 of type int) best, whereas the second candidate matches the second argument best. We could argue that int is in some sense closer to long than to double (which supports choosing the second candidate), but C++ does not attempt to define a measure of closeness that involves multiple call arguments.

Given this first principle, we are left with specifying how well a given argument matches the corresponding parameter of a viable candidate. As a first approximation we can rank the possible matches as follows (from best to worst):

- Perfect match. The parameter has the type of the expression, or it has a type that is a reference to the type of the expression (possibly with added const and/or volatile qualifiers).

- Match with minor adjustments. This includes, for example, the decay of an array variable to a pointer to its first element, or the addition of const to match an argument of type int** to a parameter of type int const* const*.

- Match with promotion. Promotion is a kind of implicit conversion that includes the conversion of small integral types (such as bool, char, short, and sometimes enumerations) to int, unsigned int, long or unsigned long, and the conversion of float to double.

- Match with standard conversions only. This includes any sort of standard conversion (such as int to float) but excludes the implicit call to a conversion operator or a converting constructor.

- Match with user-defined conversions. This allows any kind of implicit conversion.

- Match with ellipsis. An ellipsis parameter can match almost any type (but non-POD class types result in undefined behavior).

B.3 Overloading Details

The previous section covers most of the overloading situations encountered in everyday C++ programming. There are, unfortunately, many more rules and exceptions to these rules—more than is reasonable to present in a book that is not really about function overloading in C++. Nonetheless, we discuss some of them here in part because they apply somewhat more often than other rules and in part to provide a sense for how deep the details go.

B.3.1 Prefer Nontemplates

When all other aspects of overload resolution are equal, a nontemplate function is preferred over an instance of a template (it doesn't matter whether that instance is generated from the generic template definition or whether it is provided as an explicit specialization). For example:

```
template<typename T> int f(T);    // (1)
void f(int);                      // (2)

int main()
{
    return f(7);  // ERROR: selects (2), which doesn't return a value
}
```

This example also clearly illustrates that overload resolution normally does not involve the return type of the selected function.

If the choice is between two templates, then the most specialized of the templates is preferred (provided one is actually more specialized than the other). See Section 12.2.2 on page 186 for a thorough explanation of this concept.

B.3.2 Conversion Sequences

An implicit conversion can, in general, be a sequence of elementary conversions. Consider the following code example:

```
class Base {
  public:
    operator short() const;
};

class Derived : public Base {
};

void count(int);

void process(Derived const& object)
{
    count(object);    // matches with user-defined conversion
}
```

The call count(object) works because object can implicitly be converted to int. However, this conversion requires several steps:

1. A conversion of object from Derived const to Base const

Bibliography

This appendix lists the resources that were mentioned, adopted, or cited in this book. These days many of the advancements in programming happen in electronic forums. It is therefore not surprising to find, in addition to the more traditional books and articles, quite a few Web sites. We certainly do not claim that our list is close to being comprehensive. However, we do find that they are relevant contributions to the topic of C++ templates.

Web sites are typically considerably more volatile than books and articles. The Internet links listed here may not be valid in the future. Therefore, we provide the actual list of links for this book at the following site (and we expect this site to be stable):

http://www.josuttis.com/tmplbook

Before listing the books, articles, and Web sites, we introduce the more interactive kind of resources that are provided by so-called newsgroups.

Newsgroups

Usenet is a large and diverse collection of electronic forums often called newsgroups. Some of these newsgroups are moderated, which means that every submission is examined in some way for its appropriateness.

A few Usenet groups are dedicated to the discussion of the C++ language. In fact, many of the most advanced techniques presented in this book were first published in some of these groups. In some cases, techniques were developed through collaborative discussion in these groups.

The following Usenet newsgroups discuss C++, the standard, and the C++ standard library:

- Tutorial level C++ (unmoderated)
`alt.comp.lang.learn.c-c++`
- General aspects of C++ (unmoderated)
`comp.lang.c++`
- General aspects of C++ (moderated)
`comp.lang.c++.moderated`
- Aspects of the C++ standard (moderated)
`comp.std.c++`

If you don't have access to a Usenet newsgroups server, you can use the Google Usenet archive:

http://groups.google.com

Books and Web Sites

[AlexandrescuDesign]
Andrei Alexandrescu
Modern C++ Design
Generic Programming and Design Patterns Applied
Addison-Wesley, Reading, MA, 2001

[AusternSTL]
Matthew H. Austern
Generic Programming and the STL
Using and Extending the C++ Standard Template Library Addison-Wesley, Reading, MA, 1999

[BCCL]
Jeremy Siek
The Boost Concept Check Library
http://www.boost.org/libs/concept_check/concept_check.htm

[Blitz++]
Todd Veldhuizen
Blitz++: Object-Oriented Scientific Computing
http://www.oonumerics.org/blitz

[Boost]
The Boost Repository for Free, Peer-Reviewed C++ Libraries
http://www.boost.org

[BoostCompose]
Boost Compose Library
http://www.boost.org/libs/compose

[BoostSmartPtr]
Smart Pointer Library
http://www.boost.org/libs/smart_ptr

[BoostTypeTraits]
Type Traits Library
http://www.boost.org/libs/type_traits

[CargillExceptionSafety]
Tom Cargill
Exception Handling: A False Sense of Security
Available at: http://www.awprofessional.com/meyerscddemo/demo/magazine/index.htm
C++ Report, November-December 1994

[CoplienCRTP]

PREVIOUS NEXT

Glossary

This glossary is a compilation of the most important technical terms that are topic in this book. See [StroustrupGlossary] for a very complete, general glossary of terms used by C++ programmers.
abstract class

A class for which the creation of concrete objects (instances) is impossible. Abstract classes can be used to collect common properties of different classes in a single type or to define a polymorphic interface. Because abstract classes are used as base classes, the acronym ABC is sometimes used for abstract base class.

ADL

An acronym for argument-dependent lookup. ADL is a process that looks for a name of a function (or operator) in namespaces and classes that are in some way associated with the arguments of the function call in which that function (or operator name) appears. For historical reasons, it is sometimes called extended Koenig lookup or just Koenig lookup (the latter is also used for ADL applied to operators only).

angle bracket hack

A nonstandard feature that allows a compiler to accept two consecutive > characters as two closing angle brackets (even though they normally require intervening whitespace). For example, the expression vector<list<int>> is not valid C++ but is treated identically to vector<list<int> > by the angle bracket hack.

angle brackets

The characters < and > when they are used to delimit a list of template arguments or template parameters.

ANSI

An acronym for American National Standard Institute. A private, nonprofit organization that coordinates efforts to produce standard specifications of all kinds. A subcommittee called J16 is a driving force behind the standardization of C++. It cooperates closely with the international standards organization (ISO).

argument

A value (in a broad sense) that substitutes a parameter of a programmatic entity. For example, in a function call abs(-3) the argument is -3. In some programming communities arguments are called actual parameters (whereas parameters are called formal parameters).

argument-dependent lookup See **[**ADL**] class**

The description of a category of objects. The class defines a set of characteristics for any object. These include its data (attributes, data members) as well as its operations (methods, member functions). In C++, classes are structures with members that can also be functions and are subject to access limitations. They are declared using the keywords class or struct.

PREVIOUS