



C++ Templates

The Complete Guide



David Vandevor
Nicolai M. Josuttis

To Karina
—David

To those who help and love
—Nico

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Addison-Wesley was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers discounts on this book when ordered in quantity for special sales. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside of the United States, please contact:

International Sales
(317) 581-3793
international@pearsontechgroup.com

Visit Addison-Wesley on the Web: www.awprofessional.com

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.

ISBN 0-201-73484-2 (alk. paper)

1. Microsoft Visual C++. 2. C++ (Computer program language) 3. Standard template library. I. Josuttis, Nicolai M. II. Title.

QA76.73.C153 V37 2003
005.26'8—dc21

2002027933

Copyright © 2003 by Pearson Education, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher. Printed in the United States of America. Published simultaneously in Canada.

For information on obtaining permission for use of material from this work, please submit a written request to:

Pearson Education, Inc.
Rights and Contracts Department
75 Arlington Street, Suite 300
Boston, MA 02116
Fax: (617) 848-7047

ISBN 0-201-73484-2

Text printed in the United States on recycled paper at RR Donnelley Crawfordsville in Crawfordsville, Indiana.

12th Printing June 2010

Contents

Preface	xv
Acknowledgments	xvii
1 About This Book	1
1.1 What You Should Know Before Reading This Book	2
1.2 Overall Structure of the Book	2
1.3 How to Read This Book	3
1.4 Some Remarks About Programming Style	3
1.5 The Standard versus Reality	5
1.6 Example Code and Additional Informations	5
1.7 Feedback	5
 Part I: The Basics	 7
2 Function Templates	9
2.1 A First Look at Function Templates	9
2.1.1 Defining the Template	9
2.1.2 Using the Template	10
2.2 Argument Deduction	12
2.3 Template Parameters	13
2.4 Overloading Function Templates	15
2.5 Summary	19
 3 Class Templates	 21
3.1 Implementation of Class Template Stack	21
3.1.1 Declaration of Class Templates	22

3.1.2	Implementation of Member Functions	24
3.2	Use of Class Template <code>Stack</code>	25
3.3	Specializations of Class Templates	27
3.4	Partial Specialization	29
3.5	Default Template Arguments	30
3.6	Summary	33
4	Nontype Template Parameters	35
4.1	Nontype Class Template Parameters	35
4.2	Nontype Function Template Parameters	39
4.3	Restrictions for Nontype Template Parameters	40
4.4	Summary	41
5	Tricky Basics	43
5.1	Keyword <code>typename</code>	43
5.2	Using <code>this-></code>	45
5.3	Member Templates	45
5.4	Template Template Parameters	50
5.5	Zero Initialization	56
5.6	Using String Literals as Arguments for Function Templates	57
5.7	Summary	60
6	Using Templates in Practice	61
6.1	The Inclusion Model	61
6.1.1	Linker Errors	61
6.1.2	Templates in Header Files	63
6.2	Explicit Instantiation	65
6.2.1	Example of Explicit Instantiation	65
6.2.2	Combining the Inclusion Model and Explicit Instantiation	67
6.3	The Separation Model	68
6.3.1	The Keyword <code>export</code>	68
6.3.2	Limitations of the Separation Model	70
6.3.3	Preparing for the Separation Model	70
6.4	Templates and <code>inline</code>	72
6.5	Precompiled Headers	72
6.6	Debugging Templates	74

6.6.1	Decoding the Error Novel	75
6.6.2	Shallow Instantiation	77
6.6.3	Long Symbols	79
6.6.4	Tracers	79
6.6.5	Oracles	84
6.6.6	Archetypes	85
6.7	Afternotes	85
6.8	Summary	85
7	Basic Template Terminology	87
7.1	“Class Template” or “Template Class”?	87
7.2	Instantiation and Specialization	88
7.3	Declarations versus Definitions	89
7.4	The One-Definition Rule	90
7.5	Template Arguments versus Template Parameters	90

Part II: Templates in Depth

93

8	Fundamentals in Depth	95
8.1	Parameterized Declarations	95
8.1.1	Virtual Member Functions	98
8.1.2	Linkage of Templates	99
8.1.3	Primary Templates	100
8.2	Template Parameters	100
8.2.1	Type Parameters	101
8.2.2	Nontype Parameters	101
8.2.3	Template Template Parameters	102
8.2.4	Default Template Arguments	103
8.3	Template Arguments	104
8.3.1	Function Template Arguments	105
8.3.2	Type Arguments	108
8.3.3	Nontype Arguments	109
8.3.4	Template Template Arguments	111
8.3.5	Equivalence	113
8.4	Friends	113
8.4.1	Friend Functions	114

8.4.2	Friend Templates	117
8.5	Afternotes	117
9	Names in Templates	119
9.1	Name Taxonomy	119
9.2	Looking Up Names	121
9.2.1	Argument-Dependent Lookup	123
9.2.2	Friend Name Injection	125
9.2.3	Injected Class Names	126
9.3	Parsing Templates	127
9.3.1	Context Sensitivity in Nontemplates	127
9.3.2	Dependent Names of Types	130
9.3.3	Dependent Names of Templates	132
9.3.4	Dependent Names in Using-Declarations	133
9.3.5	ADL and Explicit Template Arguments	135
9.4	Derivation and Class Templates	135
9.4.1	Nondependent Base Classes	135
9.4.2	Dependent Base Classes	136
9.5	Afternotes	139
10	Instantiation	141
10.1	On-Demand Instantiation	141
10.2	Lazy Instantiation	143
10.3	The C++ Instantiation Model	146
10.3.1	Two-Phase Lookup	146
10.3.2	Points of Instantiation	146
10.3.3	The Inclusion and Separation Models	149
10.3.4	Looking Across Translation Units	150
10.3.5	Examples	151
10.4	Implementation Schemes	153
10.4.1	Greedy Instantiation	155
10.4.2	Queried Instantiation	156
10.4.3	Iterated Instantiation	157
10.5	Explicit Instantiation	159
10.6	Afternotes	163

11	Template Argument Deduction	167
11.1	The Deduction Process	167
11.2	Deduced Contexts	169
11.3	Special Deduction Situations	171
11.4	Allowable Argument Conversions	172
11.5	Class Template Parameters	173
11.6	Default Call Arguments	173
11.7	The Barton-Nackman Trick	174
11.8	Afternotes	177
12	Specialization and Overloading	179
12.1	When “Generic Code” Doesn’t Quite Cut It	179
12.1.1	Transparent Customization	180
12.1.2	Semantic Transparency	181
12.2	Overloading Function Templates	183
12.2.1	Signatures	184
12.2.2	Partial Ordering of Overloaded Function Templates	186
12.2.3	Formal Ordering Rules	188
12.2.4	Templates and Nontemplates	189
12.3	Explicit Specialization	190
12.3.1	Full Class Template Specialization	190
12.3.2	Full Function Template Specialization	194
12.3.3	Full Member Specialization	197
12.4	Partial Class Template Specialization	200
12.5	Afternotes	203
13	Future Directions	205
13.1	The Angle Bracket Hack	205
13.2	Relaxed typename Rules	206
13.3	Default Function Template Arguments	207
13.4	String Literal and Floating-Point Template Arguments	209
13.5	Relaxed Matching of Template Template Parameters	211
13.6	Typedef Templates	212
13.7	Partial Specialization of Function Templates	213
13.8	The typeof Operator	215

13.9	Named Template Arguments	216
13.10	Static Properties	218
13.11	Custom Instantiation Diagnostics	218
13.12	Overloaded Class Templates	221
13.13	List Parameters	222
13.14	Layout Control	224
13.15	Initializer Deduction	225
13.16	Function Expressions	226
13.17	Afternotes	228

Part III: Templates and Design 229

14 The Polymorphic Power of Templates 231

14.1	Dynamic Polymorphism	231
14.2	Static Polymorphism	234
14.3	Dynamic versus Static Polymorphism	238
14.4	New Forms of Design Patterns	239
14.5	Generic Programming	240
14.6	Afternotes	243

15 Traits and Policy Classes 245

15.1	An Example: Accumulating a Sequence	245
15.1.1	Fixed Traits	246
15.1.2	Value Traits	250
15.1.3	Parameterized Traits	254
15.1.4	Policies and Policy Classes	255
15.1.5	Traits and Policies: What's the Difference?	258
15.1.6	Member Templates versus Template Template Parameters	259
15.1.7	Combining Multiple Policies and/or Traits	261
15.1.8	Accumulation with General Iterators	262
15.2	Type Functions	263
15.2.1	Determining Element Types	264
15.2.2	Determining Class Types	266
15.2.3	References and Qualifiers	268
15.2.4	Promotion Traits	271

15.3 Policy Traits 275

15.3.1 Read-only Parameter Types 276

15.3.2 Copying, Swapping, and Moving 279

15.4 Afternotes 284

16 Templates and Inheritance 285

16.1 Named Template Arguments 285

16.2 The Empty Base Class Optimization (EBCO) 289

16.2.1 Layout Principles 290

16.2.2 Members as Base Classes 293

16.3 The Curiously Recurring Template Pattern (CRTP) 295

16.4 Parameterized Virtuality 298

16.5 Afternotes 299

17 Metaprograms 301

17.1 A First Example of a Metaprogram 301

17.2 Enumeration Values versus Static Constants 303

17.3 A Second Example: Computing the Square Root 305

17.4 Using Induction Variables 309

17.5 Computational Completeness 312

17.6 Recursive Instantiation versus Recursive Template Arguments 313

17.7 Using Metaprograms to Unroll Loops 314

17.8 Afternotes 318

18 Expression Templates 321

18.1 Temporaries and Split Loops 322

18.2 Encoding Expressions in Template Arguments 328

18.2.1 Operands of the Expression Templates 328

18.2.2 The Array Type 332

18.2.3 The Operators 334

18.2.4 Review 336

18.2.5 Expression Templates Assignments 338

18.3 Performance and Limitations of Expression Templates 340

18.4 Afternotes 341

Part IV: Advanced Applications 345

19	Type Classification	347
19.1	Determining Fundamental Types	347
19.2	Determining Compound Types	350
19.3	Identifying Function Types	352
19.4	Enumeration Classification with Overload Resolution	356
19.5	Determining Class Types	359
19.6	Putting It All Together	359
19.7	Afternotes	363
20	Smart Pointers	365
20.1	Holders and Trules	365
20.1.1	Protecting Against Exceptions	366
20.1.2	Holders	368
20.1.3	Holders as Members	370
20.1.4	Resource Acquisition Is Initialization	373
20.1.5	Holder Limitations	373
20.1.6	Copying Holders	375
20.1.7	Copying Holders Across Function Calls	375
20.1.8	Trules	376
20.2	Reference Counting	379
20.2.1	Where Is the Counter?	380
20.2.2	Concurrent Counter Access	381
20.2.3	Destruction and Deallocation	382
20.2.4	The <code>CountingPtr</code> Template	383
20.2.5	A Simple Noninvasive Counter	386
20.2.6	A Simple Invasive Counter Template	388
20.2.7	Constness	390
20.2.8	Implicit Conversions	390
20.2.9	Comparisons	393
20.3	Afternotes	394
21	Tuples	395
21.1	Duos	395
21.2	Recursive Duos	401
21.2.1	Number of Fields	401

21.2.2	Type of Fields	403
21.2.3	Value of Fields	404
21.3	Tuple Construction	410
21.4	Afternotes	415
22	Function Objects and Callbacks	417
22.1	Direct, Indirect, and Inline Calls	418
22.2	Pointers and References to Functions	421
22.3	Pointer-to-Member Functions	423
22.4	Class Type Functors	426
22.4.1	A First Example of Class Type Functors	426
22.4.2	Type of Class Type Functors	428
22.5	Specifying Functors	429
22.5.1	Functors as Template Type Arguments	429
22.5.2	Functors as Function Call Arguments	430
22.5.3	Combining Function Call Parameters and Template Type Parameters	431
22.5.4	Functors as Nontype Template Arguments	432
22.5.5	Function Pointer Encapsulation	433
22.6	Introspection	436
22.6.1	Analyzing a Functor Type	436
22.6.2	Accessing Parameter Types	437
22.6.3	Encapsulating Function Pointers	439
22.7	Function Object Composition	445
22.7.1	Simple Composition	446
22.7.2	Mixed Type Composition	450
22.7.3	Reducing the Number of Parameters	454
22.8	Value Binders	457
22.8.1	Selecting the Binding	458
22.8.2	Bound Signature	460
22.8.3	Argument Selection	462
22.8.4	Convenience Functions	468
22.9	Functor Operations: A Complete Implementation	471
22.10	Afternotes	474

Appendixes	475
A The One-Definition Rule	475
A.1 Translation Units	475
A.2 Declarations and Definitions	476
A.3 The One-Definition Rule in Detail	477
A.3.1 One-per-Program Constraints	477
A.3.2 One-per-Translation Unit Constraints	479
A.3.3 Cross-Translation Unit Equivalence Constraints	481
B Overload Resolution	487
B.1 When Does Overload Resolution Kick In?	488
B.2 Simplified Overload Resolution	488
B.2.1 The Implied Argument for Member Functions	490
B.2.2 Refining the Perfect Match	492
B.3 Overloading Details	493
B.3.1 Prefer Nontemplates	493
B.3.2 Conversion Sequences	494
B.3.3 Pointer Conversions	494
B.3.4 Functors and Surrogate Functions	496
B.3.5 Other Overloading Contexts	497
Bibliography	499
Newsgroups	499
Books and Web Sites	500
Glossary	507
Index	517

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!

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

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>,
                      public Discriminator<Setter2,2>,
                      public Discriminator<Setter3,3>,
                      public Discriminator<Setter4,4> {
};
```

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).

Note the use of an intermediate `Discriminator` template. It is needed to allow the various `Setter` types to be identical. (You cannot have multiple direct base classes of the same type. Indirect base classes, on the other hand, can have types that are identical to those of other bases.)

As announced earlier, we're collecting the defaults in a base class:

```
// name default policies as P1, P2, P3, P4
class DefaultPolicies {
public:
    typedef DefaultPolicy1 P1;
    typedef DefaultPolicy2 P2;
    typedef DefaultPolicy3 P3;
    typedef DefaultPolicy4 P4;
};
```

However, we must be careful to avoid ambiguities if we end up inheriting multiple times from this base class. Therefore, we ensure that the base class is inherited virtually:

```
// class to define a use of the default policy values
// avoids ambiguities if we derive from DefaultPolicies more than once
class DefaultPolicyArgs : virtual public DefaultPolicies {
};
```

Finally, we also need some templates to override the default policy values:

```
template <typename Policy>
class Policy1_is : virtual public DefaultPolicies {
public:
    typedef Policy P1; // overriding typedef
};

template <typename Policy>
class Policy2_is : virtual public DefaultPolicies {
public:
    typedef Policy P2; // overriding typedef
};

template <typename Policy>
class Policy3_is : virtual public DefaultPolicies {
public:
    typedef Policy P3; // overriding typedef
};
```



```
template <typename Policy>
class Policy4_is : virtual public DefaultPolicies {
public:
    typedef Policy P4; //overriding typedef
};
```

With all this in place, our desired objective is achieved. Now let's look at what we have by example. Let's instantiate a `BreadSlicer<>` as follows:

```
BreadSlicer<Policy3_is<CustomPolicy> > bc;
```

For this `BreadSlicer<>` the type `Policies` is defined as

```
PolicySelector<Policy3_is<CustomPolicy>,
              DefaultPolicyArgs,
              DefaultPolicyArgs,
              DefaultPolicyArgs>
```

With the help of the `Discriminator<>` class templates this results in a hierarchy, in which all template arguments are base classes (see Figure 16.1). The important point is that these base classes

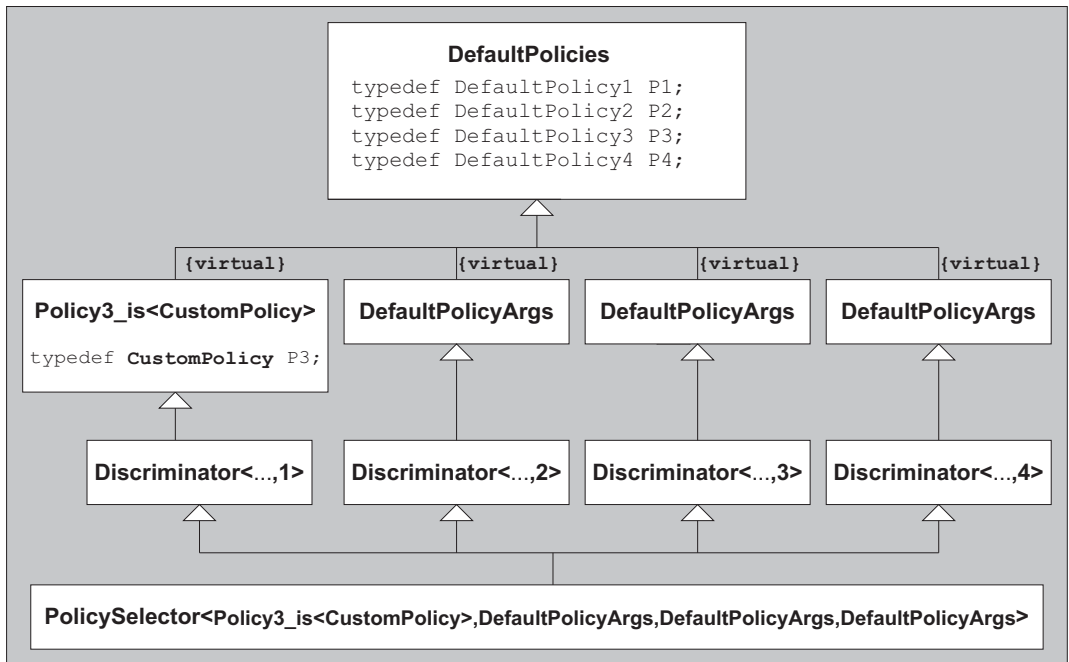


Figure 16.1. Resulting type hierarchy of `BreadSlicer<>::Policies`

all have the same virtual base class `DefaultPolicies`, which defines the default types for `P1`, `P2`, `P3`, and `P4`. However, `P3` is redefined in one of the derived classes—namely, in `Policy3_is<>`. According to the so-called *domination rule* this definition hides the definition of the base class. Thus, this is *not* an ambiguity.²

Inside the template `BreadSlicer` you can refer to the four policies by using qualified names such as `Policies::P3`. For example:

```
template <... >
class BreadSlicer {
    ...
public:
    void print () {
        Policies::P3::doPrint();
    }
    ...
};
```

In `inherit/namedtmpl.cpp` you can find the entire example.

We developed the technique for four template type parameters, but it obviously scales to any reasonable number of such parameters. Note that we never actually instantiate objects of the helper class that contain virtual bases. Hence, the fact that they are virtual bases is not a performance or memory consumption issue.

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 {
};
```

² You can find the domination rule in Section 10.2/6 in the C++ Standard (see [Standard98]) and a discussion about it in Section 10.1.1 of [EllisStroustrupARM].

```
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()
{
    std::cout << "sizeof(Empty):      " << sizeof(Empty)
               << '\n';
    std::cout << "sizeof(EmptyToo):   " << sizeof(EmptyToo)
               << '\n';
    std::cout << "sizeof(EmptyThree): " << sizeof(EmptyThree)
               << '\n';
}

```

If your compiler implements the empty base optimization, it will print the same size for every class, but none of these classes has size zero (see Figure 16.2). This means that within class `EmptyToo`, the class `Empty` is not given any space. Note also that an empty class with optimized empty bases (and no other bases) is also empty. This explains why class `EmptyThree` can also have the same size as class `Empty`. If your compiler does not implement the empty base optimization, it will print different sizes (see Figure 16.3).



Figure 16.2. Layout of `EmptyThree` by a compiler that implements the EBCO

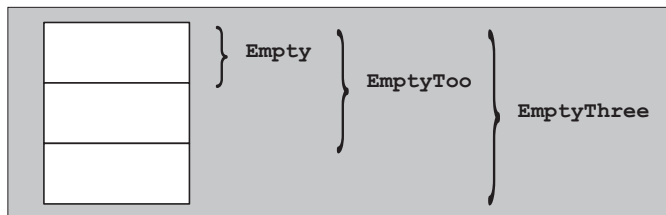


Figure 16.3. Layout of `EmptyThree` by a compiler that does not implement the EBCO

Consider an example that runs into a constraint of empty base optimization:

```

// inherit/ebco2.cpp
#include <iostream>

class Empty {
    typedef int Int; // typedef members don't make a class nonempty
};

```

```

class EmptyToo : public Empty {
};

class NonEmpty : public Empty, public EmptyToo {
};

int main()
{
    std::cout << "sizeof(Empty):    " << sizeof(Empty) << '\n';
    std::cout << "sizeof(EmptyToo): " << sizeof(EmptyToo) << '\n';
    std::cout << "sizeof(NonEmpty): " << sizeof(NonEmpty) << '\n';
}

```

It may come as a surprise that class `NonEmpty` is not an empty class. After all, it does not have any members and neither do its base classes. However, the base classes `Empty` and `EmptyToo` of `NonEmpty` cannot be allocated to the same address because this would cause the base class `Empty` of `EmptyToo` to end up at the same address as the base class `Empty` of class `NonEmpty`. In other words, two subobjects of the same type would end up at the same offset, and this is not permitted by the object layout rules of C++. It may be conceivable to decide that one of the `Empty` base subobjects is placed at offset “0 bytes” and the other at offset “1 byte,” but the complete `NonEmpty` object still cannot have a size of one byte because in an array of two `NonEmpty` objects, an `Empty` subobject of the first element cannot end up at the same address as an `Empty` subobject of the second element (see Figure 16.4).

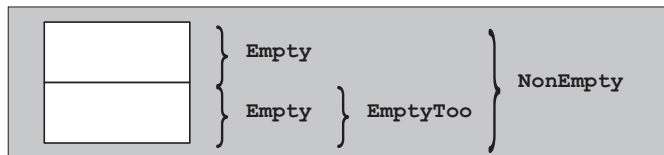


Figure 16.4. Layout of `NonEmpty` by a compiler that implements the EBCO

The rationale for the constraint on empty base optimization stems from the fact that it is desirable to be able to compare whether two pointers point to the same object. Because pointers are nearly always internally represented as just addresses, we must ensure that two different addresses (that is, pointer values) correspond to two different objects.

The constraint may not seem very significant. However, in practice, it is often encountered because many classes tend to inherit from a small set of empty classes that define some common type-defs. When two subobjects of such classes are used in the same complete object, the optimization is inhibited.

16.2.2 Members as Base Classes

The empty base class optimization has no equivalent for data members because (among other things) it would create some problems with the representation of pointers to members. As a result, it is sometimes desirable to implement as a (private) base class what would at first sight be thought of as a member variable. However, this is not without its challenges.

The problem is most interesting in the context of templates because template parameters are often substituted with empty class types, but in general one cannot rely on this rule. If nothing is known about a template type parameter, empty base optimization cannot easily be exploited. Indeed, consider the following trivial example:

```
template <typename T1, typename T2>
class MyClass {
    private:
        T1 a;
        T2 b;
        ...
};
```

It is entirely possible that one or both template parameters are substituted by an empty class type. If this is the case, then the representation of `MyClass<T1,T2>` may be suboptimal and may waste a word of memory for every instance of a `MyClass<T1,T2>`.

This can be avoided by making the template arguments base classes instead:

```
template <typename T1, typename T2>
class MyClass : private T1, private T2 {
};
```

However, this straightforward alternative has its own set of problems. It doesn't work when `T1` or `T2` is substituted with a nonclass type or with a union type. It also doesn't work when the two parameters are substituted with the same type (although this can be addressed fairly easily by adding another layer of inheritance; see page 287 or page 449). However, even if we satisfactorily addressed these problems, a very serious problem persists: Adding a base class can fundamentally modify the interface of the given class. For our `MyClass` class, this may not seem very significant because there are very few interface elements to affect, but as we see later in this chapter, inheriting from a template parameter can affect whether a member function is virtual. Clearly, this approach to exploiting EBCO is fraught with all kinds of trouble.

A more practical tool can be devised for the common case when a template parameter is known to be substituted by class types only and when another member of the class template is available. The main idea is to “merge” the potentially empty type parameter with the other member using EBCO. For example, instead of writing

```

template <typename CustomClass>
class Optimizable {
private:
    CustomClass info;      // might be empty
    void*        storage;
    ...
};

```

a template implementer would use the following:

```

template <typename CustomClass>
class Optimizable {
private:
    BaseMemberPair<CustomClass, void*> info_and_storage;
    ...
};

```

Even without seeing the implementation of the template `BaseMemberPair`, it is clear that its use makes the implementation of `Optimizable` more verbose. However, various template library implementers have reported that the performance gains (for the clients of their libraries) do justify the added complexity.

The implementation of `BaseMemberPair` can be fairly compact:

```

// inherit/basememberpair.hpp

#ifndef BASE_MEMBER_PAIR_HPP
#define BASE_MEMBER_PAIR_HPP

template <typename Base, typename Member>
class BaseMemberPair : private Base {
private:
    Member member;
public:
    // constructor
    BaseMemberPair (Base const & b, Member const & m)
        : Base(b), member(m) {
    }

    // access base class data via first()
    Base const& first() const {
        return (Base const&)*this;
    }
};

```

```

    Base& first() {
        return (Base&)*this;
    }

    // access member data via second()
    Member const& second() const {
        return this->member;
    }
    Member& second() {
        return this->member;
    }
};

#endif // BASE_MEMBER_PAIR_HPP

```

An implementation needs to use the member functions `first()` and `second()` to access the encapsulated (and possibly storage-optimized) data members.

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() {
        ++count;
    }
};

```

```

    // copy constructor
    ObjectCounter (ObjectCounter<CountedType> const&) {
        ++count;
    }

    // destructor
    ~ObjectCounter() {
        --count;
    }

public:
    // return number of existing objects:
    static size_t live() {
        return count;
    }
};

// initialize counter with zero
template <typename CountedType>
size_t ObjectCounter<CountedType>::count = 0;

```

If we want to count the number of live (that is, not yet destroyed) objects for a certain class type, it suffices to derive the class from the `ObjectCounter` template. For example, we can define and use a counted string class along the following lines:

```

// inherit/testcounter.cpp

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

template <typename CharT>
class MyString : public ObjectCounter<MyString<CharT> > {
    ...
};

int main()
{
    MyString<char> s1, s2;
    MyString<wchar_t> ws;
}

```

```

    std::cout << "number of MyString<char>:    "
               << MyString<char>::live() << std::endl;
    std::cout << "number of MyString<wchar_t>:  "
               << ws.live() << std::endl;
}

```

In general, CRTP is useful to factor out implementations of interfaces that can only be member functions (for example, constructor, destructors, and subscript operators).

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`.

Index

Note: Bold page numbers indicate major topics; italic page numbers indicate examples.

-> 145
[] 491
>> 206

A

ABC 507
about the book 1
abstract class **G507**
access declaration 133
actual parameter 91
Adamczyk, Steve 203
ADL 122, **123**, **G507**
Alexandrescu, Andrei 258, 364
AlexandrescuDesign 500
aliasing 420
alignof **224**
__alignof__ **224**
allocator 52, 284
angle brackets 10, **G507**
 hack **205**, **G507**
 parsing 128, 205
anonymous union 143
ANSI **G507**
archetype **85**
ArgSelect<> 463
argument 90, **104**, **G508**
 conversions 172
 deduction **167**
 see argument deduction
 derived class 295
 for function templates **105**

 for template template parameters 51, **111**, 211
 match 488
 named **285**
 nontype arguments **109**
 type arguments **108**
 versus parameter 90
argument deduction 12
 for function templates 13
argument-dependent lookup 122, **123**, **G508**
array
 as template parameter 59
 conversion to pointer 58, 168
 qualification 351
Array<> 328
assignment
 with type conversion 45
associated class 123
associated namespace 123
AusternSTL 500
automatic instantiation 141
auto_ptr 394
avoiding
 deduction **174**

B

back()
 for vectors 21, 24
baggage 284
barrier 420
Barton, John J. 174
Barton-Nackman trick **174**, 177
 versus CRTP 299

base class
 conversion to 494
 dependent 45, 136, 137
 duplicate 287, 449
 empty 289
 nondependent 135
 parameterized by derived class 295

BaseMem<> 449

BCCL 500

bibliography **499**

binary_function 284

bind() 468

binder

 for function objects 457

Binder<> 460, **465**

BinderParams<> 461

bindfp() 470

bitset 44

Blitz++ 318, 500

books **499**

bool

 conversion to 392, 494

Boost 500

BoostCompose 500

BoostSmartPtr 500

BoostTypeTraits 501

Borland 155, 342

bounded polymorphism 238

BoundVal<> 459

bridge pattern 239

by-reference versus by-value 331

by-value versus by-reference 331

C

call **418**

 parameter 13

call argument

 default **173**

callback **417**

CargillExceptionSafety 501

Cfront 163

char*

 as template argument **40**, 209

char_traits 284

class 10, 101, **G508**

 associated 123

 as template argument **40**

 definition 480

 dependent base 136

 inner 227

 local 101

 name injection 126

 nondependent base 135

 policy class **255**

 qualification 266, 359

 template

 see class template

 versus struct 87

class template **21**, 87, **G508**

 as member **45**, 95

 declaration 22, **95**

 default argument 30, **103**

 full specialization **190**

 overloading 221

 parameters **173**

 partial specialization **200**

class type 87, **G508**

 qualification 266, 359

class type functor 426

client code 245

C linkage 99, 435

code bloat 201

code layout

 control 224

 principles 290

collection class **G508**

 see container

compiling 74, 153

 models **141**

 symbol length 79

complete type 143

complex 12

compose() 447

Composer<> 446, **455**

composition

 of function objects 445

CompoundT<> 350, 354

concept 75, 78

constant

 true constant 515

constant-expression 91, 107, **303**, **G508**

const member function **G508**

constness

 and smart pointers 390

- contact with the authors **5**
- container **G508**
 - as template template parameter **112**
 - element type **264**
- context
 - deduced **169**
- context-sensitive **119**
- conversion
 - array to pointer **58, 168**
 - base-to-derived **494**
 - derived-to-base **494**
 - for pointers **494**
 - for smart pointers **390**
 - of arguments **172**
 - sequence **494**
 - standard **489**
 - to bool **392, 494**
 - to ellipsis **107, 489**
 - to pointer-to-member **392**
 - to void* **494**
 - type promotion **271**
 - user-defined **109, 489**
 - with templates **45**
- conversion-function-id **120**
- conversion operator **G508**
- Coplien, James **299**
- CoplienCRTP **501**
- CoreIssue115 **501**
- covariant return type **216**
- CRTP **295, G508**
 - versus Barton-Nackman trick **299**
- CSM **279**
- <cstddef> **4**
- curiously recurring template pattern **295, G509**
- CzarneckiEiseneckerGenProg **501**

D

- debugging **74**
- decay **58, 168, 421, G509**
 - of template parameters **102**
- declaration **89, 95, 476, G509**
 - access **133**
 - forward **142**
 - of class template **22**
 - versus definition **89, 476**
- decorated name **418**
- deduced context **169**

- deduced parameter **14**
- deduction **12, 167, G509**
 - avoiding **174**
 - of initializer **225**
- default
 - argument
 - see default argument
 - call arguments **173**
 - for template template argument **111**
 - for template template parameter **111**
 - nontype parameter **38**
- default argument
 - for call parameters **97**
 - for class templates **30, 103**
 - for function templates **13, 207**
- definition **9, 89, 476, G509**
 - of class types **480**
 - versus declaration **89, 476**
- demangler **59**
- dependent base class **136, G509**
- dependent name **119, 121, G509**
 - in using-declarations **133**
 - of templates **132**
 - of types **130**
- deprecated **133**
- deque **52**
- derivation **135, 285**
- derived class
 - as base class argument **295**
- design **229**
- design pattern **239**
- DesignPatternsGoV **501**
- determining types **278, 347, 361**
- diagnostics **218**
- digraph **127, 129, G509**
 - future **206**
- direct call **418**
- directive
 - for explicit instantiation **159**
- discriminated union **224**
- domination rule **289**
- dot-C file **61, G509**
- double
 - as template argument **40, 210**
 - as traits value **252**
 - zero initialization **56**
- duo **395**

- recursive 401
- Duo<> 397
- duplicate base class 287, 449
- dynamic polymorphism **231**, 238

E

- EBCDIC 247
- EBCO **289**, 449, **G510**
- EDG 501
- ellipsis 107, 358, 489
- EllisStroustrupARM 501
- e-mail to the authors **5**
- empty()
 - for vectors 21
- empty base class optimization **289**, 449, **G510**
- empty class object 431
- enumeration
 - qualification 356
 - versus static constant 303
- error handling 74
- error message **75**
- example code of the book **5**
- exception
 - protection 366
 - safety 24
- expansion
 - restricted 174
- explicit instantiation **65**, **159**
 - directive **G510**
- explicit instantiation directive 159
- explicit specialization 88, 130, 137, **190**, **G510**
- explicit template argument 135
- export **68**, **149**, 484
 - and inline 69
- expression
 - constant 107
 - for functions 226
- expression template **321**, **G510**
 - limitations 340
 - performance 340
- extended Koenig lookup 140

F

- feedback to the authors **5**
- file organization 61
- fixed traits **246**

- float
 - as template argument **40**, 210
 - as traits value 252
 - zero initialization **56**
- formal parameter 91
- forward declaration 142
- ForwardParamT 440
- friend 101, **113**
 - function **114**
 - function versus function template 177
 - name injection 125, **G510**
 - template **117**
- full specialization 301, **G510**
 - of class templates **190**
 - of function templates **194**
 - of member templates **197**
- func_ptr() 443
- function
 - call
 - see function call
 - dispatch table 156
 - expression 226
 - for types **263**
 - object
 - see function object and functor
 - pointer 421
 - qualification 352
 - signature 184
 - spilled inline 156
 - surrogate **496**
 - template
 - see function template
- function call **418**
 - operator 417
 - syntax 417
- function object **417**, **G510**
 - and overloading 496
 - binders 457
 - class type 426
 - composition 445
 - value binders 457
- FunctionPtr<> 441
- FunctionPtrT<> 439
- function template **9**, 87, **G510**
 - argument deduction 13
 - arguments **105**
 - as member **45**, 95

- declaration **95**
- default call argument **97**
- default template argument **13, 207**
- friend **114**
- full specialization **194**
- inline **72**
- nontype parameters **39**
- overloading **15, 183**
- partial specialization **213**
- versus friend function **177**
- functor **417, G510**
 - and overloading **496**
 - binders **457**
 - class type **426**
 - composition **445**
 - value binders **457**
- FunctorParam<> **438**
- fundamental type
 - promotions **274**
 - qualification **347**
- future **205**

G

- generated specialization **88**
- generic programming **240**
- Gibbons, Bill **139, 177, 299**
- glossary **507**
- greater **429**
- greedy instantiation **155**
- guard
 - for header files **479**

H

- Hartinger, Roland **217**
- header file **61, G510**
 - <stddef> **4**
 - guard **479**
 - order **73**
 - precompiled **72**
 - <std> **74**
 - <stddef.h> **4**
- heterogeneous collection **238**
- Hewlett-Packard **139**
- higher-order genericity **118**
- holder **368**
- home page of the book **5**

I

- identifier **120**
- IfThenElse<> **272, 273, 308, 438**
- implicit conversion
 - for smart pointers **390**
- implicit instantiation **141**
- #include
 - order **73**
- include file **G511**
 - see header file
- inclusion model **63, 149**
- index of the book **517**
- indirect call **418, G511**
- induction variables
 - in metaprograms **309**
- inheritance **135, 285**
 - domination rule **289**
 - duplicate base class **287, 449**
- initialization
 - of fundamental types **56**
- initializer **G511**
- initializer deduction **225**
- initializer list **56, G511**
- injected
 - class name **126, G511**
 - friend name **125**
- inline **72**
 - and export **69**
- inlining **420**
- inner class **227**
- Inprise **155**
- instance **11, G511**
- instantiated specialization **88**
- instantiation **11, 88, 141, G511**
 - automatic **141**
 - diagnostics **218**
 - explicit **65, 159**
 - explicit directive **159**
 - greedy **155**
 - implicit **141**
 - iterated **157**
 - lazy **143**
 - levels **313**
 - manual **160**
 - mechanisms **141**
 - model **146**

- on-demand 141
- point **146**
- queried **156**
- recursive **302**, 313
- shallow **77**
- int
 - zero initialization **56**
- Internet resources **499**
- introspection 436
- intrusive 238
- invasive 238
 - counter 388
- isArrayT<> 351
- IsClassT<> 266, 276, 359
- IsEnumT<> 356
- IsFuncT<> 352
- IsFundat<> 347
- ISO **G511**
- IsPtrMemT<> 351
- IsPtrT<> 350
- IsRefT<> 350
- iterated instantiation **157**
- iterator 241, **G511**
- iterator traits
 - for pointers 262
- iterator_traits 262, 284

J

- Java 227
- JosuttisAutoPtr 502
- JosuttisOOP 502
- JosuttisStdLib 502

K

- Koenig, Andrew 122, 140
- Koenig lookup 122, 140
- KoenigMooAcc 502

L

- LambdaLib 502
- layout
 - control 224
 - principles 290
- lazy instantiation **143**

- length
 - of symbols 79
- less **429**
- levels of instantiation 313
- lexing **127**
- limit
 - levels of instantiation 313
- linkable entity 90, 155, **G512**
- linkage **99**
 - to C 435
- linking 153
- LippmanObjMod 502
- list 111, 241
- list parameters **222**
- literature **499**
- local class 101
- lookup
 - argument-dependent 122
 - for names 121
 - Koenig lookup 122, 140
 - ordinary 122, 146
 - qualified 120
 - two-phase **146**
 - unqualified 120
- loop
 - split 327
 - unrolling 314
- lvalue 423, **G512**

M

- macro 417
- make_pair() 59
- mangled name 59, 418
- manual instantiation 160
- map 428
- match 488
 - best 487
 - perfect 489
- max_element() 241
- maximum
 - levels of instantiation 313
 - munch 129
- member
 - as base class 293
 - class template **45**, 95
 - function
 - see member function

- function template **45, 95**
- initialization **56**
- template
 - see member template
- member class template **G512**
- member function **97**
 - and virtual **98**
 - as template **45, 95**
 - implementation **24**
 - pointer to **423**
 - template **87, G512**
- member template **45, 95, G512**
 - full specialization **197**
 - versus template template parameter **259**
- metaprogramming **203, 301**
 - induction variables **309**
 - reflection **363**
- Metaware **139, 318**
- MeyersCounting **502**
- MeyersEffective **502**
- MeyersMoreEffective **503**
- motivation of templates **7**
- MTL **318, 503**
- MusserWangDynaVeri **503**
- Myers, Nathan **258, 284, 299**
- MyersTraits **503**

N

- Nackman, Lee R. **174**
- name **90, 119, 121**
 - class name injection **126**
 - dependent **119, 121**
 - dependent of templates **132**
 - dependent of types **130**
 - friend name injection **125**
 - linkage **421**
 - lookup **119, 121**
 - nondependent **121**
 - qualified **119, 120**
 - two-phase lookup **146**
 - typedef **101**
 - unqualified **120**
- name()
 - of `std::type_info` **58, 62, 421**
- named template argument **216, 285**
- namespace
 - associated **123**

- template **133**
 - unnamed **478, 484**
- nested class **97**
 - as template **45, 95**
- NewMat **503**
- newsgroups **499**
- NewShorterOED **503**
- NIHCL **243**
- nondeduced parameter **14**
- nondependent base class **135**
- nondependent name **121, G512**
- nonintrusive **238**
- noninvasive **238**
 - counter **386**
- nonreference
 - versus reference **58, 168, 331, 493**
- nontemplate
 - overloading **189**
- nontype argument
 - for templates **109**
 - functor **432**
- nontype parameter **35, 101**
 - restrictions **40**
- null pointer **184**
- numeric_limits **284**

O

- ODR **90, 475, G512**
- on-demand instantiation **141**
- one-definition rule **90, 475, G512**
- operator
 - `->` **145**
 - `[]` **491**
 - `>>` **206**
 - `typeof` **215**
- operator-function-id **120**
- optimization
 - for empty base class **289**
- oracle **84**
- ordering
 - partial **186**
 - rules **188**
- order of header files **73**
- ordinary lookup **122, 146**
- overloading **15, 179, 487**
 - class templates **221**
 - nonreference versus reference **493**

- of function templates **183**
 - partial ordering 186
 - reference versus nonreference 493
 - templates and nontemplates 189
- overflow resolution 356, **487**, **G512**
- P**
- pair 395, 397
- parameter 90, **100**, **G513**
 - actual 91
 - ellipsis 107, 358, 489
 - for base class 45, 137
 - for call **13**
 - formal 91
 - list of types **222**
 - nontype 35, **101**
 - of class templates **173**
 - reference versus nonreference 58, 168, 331
 - template template parameter **50**, **102**
 - type **101**
 - versus argument 90
 - void 410
- parameterization clause 95
- parameterized class **G513**
- parameterized function 481, **G513**
- parameterized traits 254
- paramter
 - match 488
- parsing **127**
 - maximum munch 129
 - of angle brackets 128, 205
- partial ordering
 - of overloading 186
- partial specialization 29, 88, 305, 331, **G513**
 - additional parameters 351
 - for function templates 213
 - of class templates **200**
- pattern 239
 - CRTP **295**
- Pennello, Tom 139
- perfect match 489, 492
- POD 87, **G513**
- POI **146**, 480, **G513**
- pointer
 - conversions 494
 - conversion to `void*` 494
 - iterator traits 262

- qualification 350
 - smart **365**
 - to function 421
 - zero initialization **56**
- pointer-to-member
 - conversion to 392
 - function 423
 - qualification 351
- point of instantiation **146**, 480, **G513**
- policy class **255**, **G513**
 - versus traits 258
- policy traits **275**
- polymorphic object 479
- polymorphism **231**, **G514**
 - bounded 238
 - dynamic **231**, 238
 - static **234**, 238
 - unbounded 238
- POOMA 318, 503
- `pop_back()`
 - for vectors 21, 24
- practice **61**
- precompiled header **72**, **G514**
- prelinker 157
- preprocessor
 - guard 479
- primary template 88, **100**, 201, 301, 305, **G514**
- prime numbers 318
- promotion 489
 - of types **271**
 - traits **271**
- property
 - static 218
 - traits 275
- `ptrdiff_t` **4**
 - versus `size_t` 491
- `push_back()`
 - for vectors 21

Q

- qualification
 - of array types 351
 - of class types 266, 359
 - of enumeration types 356
 - of function types 352
 - of fundamental types 347
 - of pointer-to-member types 351

- of pointer types 350
 - of reference types 350
- qualified-id 120
- qualified lookup 120
- qualified name 119, 120, **G514**
- qualifier 268
- queried instantiation **156**

R

RAII 373

- read-only parameter types 276
- recursive duo 401
- recursive instantiation **302, 313**
- reference 268
 - qualification 350
 - to function 421
 - to reference **269**
 - versus nonreference 58, 168, 331, 493
- reference counting **379, G514**
- reflection 363
- resource acquisition is initialization **373**
- restricted template expansion 174
- run-time analysis oracles 84
- rvalue 102, **G514**

S

- scanning **127**
- semantic transparency 181
- separation model **68, 149**
- sequence
 - of conversions 494
- set 428
- SFINAE **106, 266, 353**
- shallow instantiation **77**
- signature 184
- SignSelectT<> 463
- sizeof 263
- size_t **4**
 - versus ptrdiff_t 491
- Smalltalk 238, 243
- smart pointer **365**
 - constness 390
 - implicit conversions 390
- source file **G514**
- spaces 516
- specialization **27, 88, 141, 179, G514**

- explicit 88, 130, 137, **190**
 - full 301
 - generated 88
 - instantiated 88
 - partial 88, **200, 305, 331**
 - partial for function templates 213
- Spicer, John 203
- spilled inlined function 156
- split loop 327
- square root 305
- stack 264
- Stack<> **21**
- Standard02 504
- Standard98 503
- Standard Template Library 139, 240
 - see STL
- StaticBoundVal<> 459
- static constant
 - versus enumeration 303
- static member 26, 97
- static polymorphism **234, 238**
- static properties 218
- std::allocator 52, 284
- <std> 74
- std.hpp 73
- std::auto_ptr 394
- std::binary_function 284
- std::bitset 44
- std::char_traits 284
- std::complex 12
- <stddef.h> **4**
- std::deque 52
- std::greater 429
- std::iterator_traits 262, 284
- std::less **429**
- std::list 111, 241
- std::make_pair() 59
- std::map 428
- std::max_element() 241
- std::numeric_limits 284
- std::pair 395, 397
- std::ptrdiff_t **4**
- std::set 428
- std::size_t **4**
- std::stack 264
- std::type_info **58, 62, 421**
- std::unary_function 284

- `std::valarray` 342
- `std::vector` 21, 241
- STL 139, 240, 342
- string
 - `[]` 491
 - and reference template parameters 169
 - as template argument 40, 209
 - literal as template parameters 169
- StroustrupC++PL 504
- StroustrupDnE 504
- StroustrupGlossary 504
- struct
 - definition 480
 - qualification 266, 359
 - versus class 87
- Sun Microsystems 156
- surrogate 496
- SutterExceptional 504
- SutterMoreExceptional 504
- symbol
 - length 79

T

- tagged union 224
- Taligent 139
- taxonomy of names 119
- template G514
 - argument 90
 - see template argument
 - export 149
 - for namespaces 133
 - friend 117
 - id
 - see template-id
 - inline 72
 - instantiation 88, 141
 - see instantiation
 - member template 45, 95
 - metaprogramming 203, 301
 - motivation 7
 - name 90, 119
 - nontype arguments 109
 - of template 27
 - parameter 9, 13, 90, 100
 - partial specialization 29
 - primary 100, 201
 - specialization 27, 88
 - type arguments 108
 - union 96
- `.template` 44, 132
 - future 207
- `->template` 45, 132
 - future 207
- `::template` 132
 - future 207
- template argument 90, 104, G514
 - array 351
 - `char*` 40, 209
 - class 40
 - conversions 172
 - deduction 167, G515
 - derived class 295
 - double 40, 210
 - explicit 135
 - float 40, 210
 - named 216, 285
 - string 40, 169, 209
- template class 87
 - see class template
- template function 87
 - see function template
- template-id 87, 90, 104, 120, 132, G515
- template member function 87
 - see member function template
- template parameter G515
 - array 351
 - decay 102
 - string literal 169
- template template argument 51, 111, 211
- template template parameter 50, 102, 260
 - argument matching 51, 111, 211
 - container 112
 - future 211
 - versus member template 259
- temporaries 322
- terminology 87, 507
- `this->` 45, 137
- tokenization 127
 - maximum munch 129
- `to_string()`
 - for bitsets 44
- tracer 79
- traits 245, 331, G515
 - CSM 279

- fixed **246**
- for promotion **271**
- for value and reference 331
- parameterized 254
- policy traits **275**
- `std::iterator_traits` 262
- template **G515**
- value traits 250
- versus policy class 258
- transfer capsule 376
- translation unit 90, 150, **475**, **G515**
- transparency 180
- true constant **G515**
- trule **376**
- tuple **395**, **G515**
- `Tuple<>` 410
- two-phase lookup 137, **146**, **G515**
- two-stage lookup **146**
- type
 - arguments **108**
 - complete 143
 - conversion 45
 - definition **4**, **27**
 - see `typedef`
 - dependent name **130**
 - of container element 264
 - promotion **271**
 - qualification 278, **347**, 361
 - read-only parameter 276
- `typedef` **4**, 26, **27**
 - name 101
 - template **212**
- type function **263**, 403
- `typeid` **58**, 62, 421
- `type_info` **58**, 62, 421
- `typename` 10, **43**, 101, 130
 - future 206
- `typeof` **215**
- type parameter 10, **101**
- type safety 239
- `TypeT<>` 278, 359
- `__type_traits` 363

U

- UCN 120
- UDC 358
- `unary_function` 284

- unbounded polymorphism 238
- union
 - definition 480
 - discriminated 224
 - qualification 266, 359
 - tagged 224
 - template **96**
- universal character name 120
- unnamed namespace 478, 484
- unqualified-id 120
- unqualified lookup 120
- unqualified name 120
- unrolling loops 314
- Unruh, Erwin 301, **318**
- `UnruhPrimeOrig` 504
- `UsedFunctorParam<>` 438
- `USE_EXPORT` 71
- user-defined conversion 109, **G515**
- using 10, 133
- using-declaration 139
 - dependent name **133**

V

- `valarray` 342
- value
 - as parameter 35
 - binders 457
 - functions 263
- value traits 250
- `VandevoordeSolutions` 504
- variant type 224
- `vector` 21, 241
- Veldhuizen, Todd 320, 341
- `VeldhuizenMeta95` 505
- `VeldhuizenPapers` 505
- virtual
 - function dispatch table 156
 - member function **98**
 - parameterized **298**
- `void`
 - as parameter type 410
- `void*`
 - conversion to 494

W

- Web site of the book **5**

whitespace **G516**

Z

zero initialization **56**