

Covers C++11, C++14, and C++17



C++ Templates

The Complete Guide

SECOND EDITION

David VANDEVOORDE
Nicolai M. JOSUTTIS
Douglas GREGOR



Contents

1. [Cover Page](#)
2. [Title Page](#)
3. [Copyright Page](#)
4. [Dedication](#)
5. [Contents](#)
6. [Preface](#)
7. [Acknowledgments for the Second Edition](#)
8. [Acknowledgments for the First Edition](#)
9. [About This Book](#)
 1. [What You Should Know Before Reading This Book](#)
 2. [Overall Structure of the Book](#)
 3. [How to Read This Book](#)
 4. [Some Remarks About Programming Style](#)
 5. [The C++11, C++14, and C++17 Standards](#)
 6. [Example Code and Additional Information](#)
 7. [Feedback](#)

10. [Part I: The Basics](#)

1. [1 Function Templates](#)
 1. [1.1 A First Look at Function Templates](#)
 1. [1.1.1 Defining the Template](#)
 2. [1.1.2 Using the Template](#)
 3. [1.1.3 Two-Phase Translation](#)

2. 1.2 Template Argument Deduction
3. 1.3 Multiple Template Parameters
 1. 1.3.1 Template Parameters for Return Types
 2. 1.3.2 Deducing the Return Type
 3. 1.3.3 Return Type as Common Type
4. 1.4 Default Template ...

C++ Templates

The Complete Guide

Second Edition

David Vandevoorde

Nicolai M. Josuttis

Douglas Gregor

 Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town • Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City • São Paulo • Sydney • Hong Kong • Seoul • Singapore •
Taipei • Tokyo

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 the publisher 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.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic

...

To Alessandra & Cassandra—David

To those who care for people and mankind—Nico

To Amy, Tessa & Molly—Doug

Contents

Preface

Acknowledgments for the Second Edition

Acknowledgments for the First Edition

About This Book

What You Should Know Before Reading This Book

Overall Structure of the Book

How to Read This Book

Some Remarks About Programming Style

The C++11, C++14, and C++17 Standards

Example Code and Additional Information

Feedback

Part I: The Basics

1 Function Templates

1.1 A First Look at Function Templates

1.1.1 Defining the Template

1.1.2 Using the Template

1.1.3 Two-Phase Translation

1.2 Template Argument Deduction

1.3 Multiple Template Parameters

 1.3.1 Template Parameters for Return Types

 1.3.2 Deducing the Return Type

 1.3.3 Return Type as Common Type

1.4 Default Template Arguments

1.5 Overloading Function Templates

1.6 But, ...

Preface

The notion of templates in C++ is over 30 years old. C++ templates were already documented in 1990 in “The Annotated C++ Reference Manual” (ARM; see [*EllisStroustrupARM*]), and they had been described before then 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. With the first edition of this book, we wanted to address this issue and decided to write *the* book about templates (with perhaps a slight lack of humility).

Much has changed in C++ since that first edition was published in late 2002. New iterations of the C++ standard have added new features, and continued ...

Acknowledgments for the Second Edition

Writing a book is hard. Maintaining a book is even harder. It took us more than five years—spread over the past decade—to come up with this second edition, and it couldn’t have been done without the support and patience of a lot of people.

First, we’d like to thank everyone in the C++ community and on the C++ standardization committee. In addition to all the work to add new language and library features, they spent many, many hours explaining and discussing their work with us, and they did so with patience and enthusiasm.

Part of this community also includes the programmers who gave feedback for errors and possible improvement for the first edition over the past 15 years. There are simply too many to list ...

Acknowledgments for the First Edition

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 ...

About This Book

The first edition of this book was published almost 15 years ago. We had set out to write the definitive guide to C++ templates, with the expectation that it would be useful to practicing C++ programmers. That project was successful: It's been tremendously gratifying to hear from readers who found our material helpful, to see our book time and again being recommended as a work of reference, and to be universally well reviewed.

That first edition has aged well, with most material remaining entirely relevant to the modern C++ programmer, but there is no denying that the evolution of the language—culminating in the “Modern C++” standards, C++11, C++14, and C++17—has raised the need for a revision of the material in the first edition. ...

Part IThe Basics

This part introduces the general concepts 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 features such as nontype template parameters, variadic templates, the keyword `typename`, and member templates. Also it discusses how to deal with move semantics, how to declare parameters, and how to use generic code for compile-time programming. It ends with some general hints about terminology and regarding the use and application of templates in practice both as application programmer and author of generic libraries.

WHY TEMPLATES?

C++ requires us to declare variables, ...

Chapter 1

Function Templates

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

1.1 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.

1.1.1 Defining the Template

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

[Click here to view code image](#)

basics/max1.hpp

```
template<typename T> T max ...
```

Chapter 2

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.

2.1 IMPLEMENTATION OF CLASS TEMPLATE STACK

As we did with function templates, we declare and define class `Stack<>` in a header file as follows:

[Click here to view code image](#)

basics/stack1.hpp

```
#include <vector>
#include <cassert>

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

```
void push(T const& elem);    ...
```

Chapter 3

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.

3.1 NONTYPE CLASS TEMPLATE PARAMETERS

In contrast to the sample implementations of a stack in previous ...

Chapter 4

Variadic Templates

Since C++11, templates can have parameters that accept a variable number of template arguments. This feature allows the use of templates in places where you have to pass an arbitrary number of arguments of arbitrary types. A typical application is to pass an arbitrary number of parameters of arbitrary type through a class or framework. Another application is to provide generic code to process any number of parameters of any type.

4.1 VARIADIC TEMPLATES

Template parameters can be defined to accept an unbounded number of template arguments. Templates with this ability are called *variadic templates*.

4.1.1 Variadic Templates by Example

For example, you can use the following code to call `print()` for a variable number of ...

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:

[Click here to view code image](#)

```
template<typename T> class MyClass { public:  
    ... void foo() {
```

Chapter 6

Move Semantics and `enable_if<>`

One of the most prominent features C++11 introduced was *move semantics*. You can use it to optimize copying and assignments by moving (“stealing”) internal resources from a source object to a destination object instead of copying those contents. This can be done provided the source no longer needs its internal value or state (because it is about to be discarded).

Move semantics has a significant influence on the design of templates, and special rules were introduced to support move semantics in generic code. This chapter introduces these features.

6.1 PERFECT FORWARDING

Suppose you want to write generic code that forwards the basic property of passed arguments:

- Modifiable objects should be forwarded so ...

Chapter 7

By Value or by Reference?

Since the beginning, C++ has provided call-by-value and call-by-reference, and it is not always easy to decide which one to choose: Usually calling by reference is cheaper for nontrivial objects but more complicated. C++11 added move semantics to the mix, which means that we now have different ways to pass by reference:¹

1. **X const&** (constant lvalue reference):

The parameter refers to the passed object, without the ability to modify it.

2. **X&** (nonconstant lvalue reference):

The parameter refers to the passed object, with the ability to modify it.

3. **X&&** (rvalue reference):

The parameter refers to the passed object, with move semantics, meaning that you can modify or “steal” the value.

Deciding how to declare ...

Chapter 8

Compile-Time Programming

C++ has always included some simple ways to compute values at compile time. Templates considerably increased the possibilities in this area, and further evolution of the language has only added to this toolbox.

In the simple case, you can decide whether or not to use certain or to choose between different template code. But the compiler even can compute the outcome of control flow at compile time, provided all necessary input is available.

In fact, C++ has multiple features to support compile-time programming:

- Since before C++98, templates have provided the ability to compute at compile time, including using loops and execution path selection. (However, some consider this an “abuse” of template features, e.g., ...

Chapter 9

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 14](#). To keep the discussion simple, we assume that our C++ compilation systems consist of fairly traditional compilers and linkers (C++ systems that don't ...

Chapter 10

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 terminology we use. This is necessary because, inside the C++ community (and even in an early version of the standard), there is sometimes a lack of precision regarding terminology.

10.1 “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.

There is some confusion about how a class that is a template is ...

Chapter 11

Generic Libraries

So far, our discussion of templates has focused on their specific features, capabilities, and constraints, with immediate tasks and applications in mind (the kind of things we run into as application programmers). However, templates are most effective when used to write generic libraries and frameworks, where our designs have to consider potential uses that are *a priori* broadly unconstrained. While just about all the content in this book can be applicable to such designs, here are some general issues you should consider when writing portable components that you intend to be usable for as-yet unimagined types.

The list of issues raised here is not complete in any sense, but it summarizes some of the features introduced ...

Part IITemplates 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 most 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 effects. 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 ...

Chapter 12

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.

12.1 PARAMETERIZED DECLARATIONS

C++ currently supports four fundamental kinds of templates: class templates, function templates, variable templates, and alias templates. Each of these template kinds can appear in namespace scope, but also in class scope. In class scope they become nested class templates, member function templates, static data member templates, and member alias templates. Such templates are declared much like ordinary classes, functions, variables, and type aliases (or their ...

Chapter 13

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 the entity being referred. From an implementer’s point of view, C++ is a hard language in this respect. Consider the C++ statement `x*y;`. If `x` 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 *context-sensitive language*: A construct cannot always be understood without knowing its wider context. How does this relate to templates? ...

Chapter 14

Instantiation

Template instantiation is the process that generates types, functions, and variables 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.

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 ...

Chapter 15

Template Argument Deduction

Explicitly specifying template arguments on every call to a function template (e.g., `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.

Although template argument deduction was first developed to ease the invocation of function templates, it has since been broadened to apply to ...

Chapter 16

Specialization and Overloading

So far we have studied how C++ templates allow a generic definition to be expanded into a family of related classes, functions, or variables. 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 generic-ness: template specialization and overloading ...

Chapter 17

Future Directions

C++ templates have been evolving almost continuously from their initial design in 1988, through the various standardization milestones in 1998, 2011, 2014, and 2017. It could be argued that templates were at least somewhat related to most major language additions after the original 1998 standard.

The first edition of this book listed a number of extensions that we might see after the first standard, and several of those became reality:

- The angle bracket hack: C++11 removed the need to insert a space between two closing angle brackets.
- Default function template arguments: C++11 allows function templates to have default template arguments.
- Typedef templates: C++11 introduced alias templates, which are similar. ...

Part IIITemplates and Design

Programs are generally constructed using design patterns that map relatively well on the mechanisms offered by a chosen programming language. Because templates introduce 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. Note that several of them are covered or used by the C++ standard library.

Templates differ 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.

Our presentation aims not only at listing various useful ...

Chapter 18

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 mechanisms 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 ...

Chapter 19

Implementing Traits

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 “templated” 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, and each additional template parameter complicates the contract between the component and its client.

Fortunately, it turns out that most of the extra parameters we would introduce ...

Chapter 20

Overloading on Type Properties

Function overloading allows the same function name to be used for multiple functions, so long as those functions are distinguished by their parameter types. For example:

```
void f (int); void f (char const*);
```

With function templates, one overloads on type patterns such as pointer-to-T or `Array<T>`:

[Click here to view code image](#)

```
template<typename T> void f(T*); template<typename T> void f(Array<T>);
```

Given the prevalence of type traits (discussed in [Chapter 19](#)), it is natural to want to overload function templates based on the properties of the template arguments. For example:

[Click here to view code image](#)

```
template<typename Number> void f(Number); // only for numbers
template<typename Container> void f(Container); ...
```

Chapter 21

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 13](#) that deriving from dependent base classes forces us to deal carefully with unqualified names. However, it turns out that some interesting techniques combine these two features, including the Curiously Recurring Template Pattern (CRTP) and mixins. In this chapter, we describe a few of these techniques.

21.1 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, ...

Chapter 22

Bridging Static and Dynamic Polymorphism

[Chapter 18](#) described the nature of static polymorphism (via templates) and dynamic polymorphism (via inheritance and virtual functions) in C++. Both kinds of polymorphism provide powerful abstractions for writing programs, yet each has tradeoffs: Static polymorphism provides the same performance as nonpolymorphic code, but the set of types that can be used at run time is fixed at compile time. On the other hand, dynamic polymorphism via inheritance allows a single version of the polymorphic function to work with types not known at the time it is compiled, but it is less flexible because types must inherit from the common base class.

This chapter describes how to bridge between static and dynamic ...

Chapter 23

Metaprogramming

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 (i.e., an additional or different bit of the 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 ...

Chapter 24

Typelists

Effective programming typically requires the use of various data structures, and metaprogramming is no different. For type metaprogramming, the central data structure is the *type-list*, which, as its name implies, is a list containing types.

Template metaprograms can operate on these lists of types, manipulating them to eventually produce a part of the executable program. In this chapter, we discuss techniques for working with typelists. Since most operations involving typelists make use of template metaprogramming, we recommend that you familiarize yourself with metaprogramming, as discussed in [Chapter 23](#).

24.1 ANATOMY OF A TYPELIST

A typelist is a type that represents a list of types and can be manipulated by a template metaprogram. ...

Chapter 25

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). This chapter explores *tuples*, which aggregate data in a manner similar to classes and structs. For example, a tuple containing an `int`, a `double`, and a `std::string` is similar to a struct with `int`, `double`, and `std::string` members, except that the elements of the tuple are referenced positionally (as 0, 1, 2) rather than through names. The positional interface and the ability to easily construct a tuple from a type-list make tuples ...

Chapter 26

Discriminated Unions

The tuples developed in the previous chapter aggregate values of some list of types into a single value, giving them roughly the same functionality as a simple struct. Given this analogy, it is natural to wonder what the corresponding type would be for a union: It would contain a single value, but that value would have a type selected from some set of possible types. For example, a database field might contain an integer, floating-point value, string, or binary blob, but it can only contain a value of one of those types at any given time.

In this chapter, we develop a class template `Variant` that dynamically stores a value of one of a given set of possible value types, similar to the C++17 standard library's `std::variant<...>`.

Chapter 27

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:

[Click here to view code image](#)

```
Array<double> x(1000), y(1000); ... x = 1.2*x ...
```

Chapter 28

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 chapter, we deal mostly with the constraints that lead to compilation errors when violated, and ...

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 or objects exactly once across all files, and define classes, inline functions, and inline variables 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 ...

Appendix B

Value Categories

Expressions are a cornerstone of the C++ language, providing the primary mechanism by which it can express computations. Every expression has a type, which describes the static type of the value that its computation produces. The expression `7` has type `int`, as does the expression `5 + 2`, and the expression `x` if `x` is a variable of type `int`. Each expression also has a *value category*, which describes something about how the value was formed and affects how the expression behaves.

B.1 TRADITIONAL LVALUES AND RVALUES

Historically, there were only two value categories: lvalues and rvalues. Lvalues are expressions that refer to actual values stored in memory or in a machine register, such as the expression `x` where `x` is the ...

Appendix C

Overload Resolution

Overload resolution is the process that selects the function to call for a given call expression. Consider the following simple example:

[Click here to view code image](#)

```
void display_num(int);      // #1 void
display_num(double); // #2 int main() {      dis-
play_num(399);      // #1 matches better than #2
                     display_num(3.99); // #2 matches 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 ...

Appendix D

Standard Type Utilities

The C++ standard library largely consists of templates, many of which rely on various techniques introduced and discussed in this book. For this reason, a couple of techniques were “standardized” in the sense that the standard library defines several templates to implement libraries with generic code. These type utilities (type traits and other helpers) are listed and explained here in this chapter.

Note that some type traits need compiler support, while others can just be implemented in the library using existing in-language features (we discuss some of them in [Chapter 19](#)).

D.1 USING TYPE TRAITS

When using type traits, in general you have to include the header file `<type_traits>`:

```
#include <type_traits>
```

Then ...

Appendix E

Concepts

For many years now, C++ language designers have explored how to constrain the parameters of templates. For example, in our prototypical `max()` template, we'd like to state up front that it shouldn't be called for types that aren't comparable using the less-than operator. Other templates may want to require that they be instantiated with types that are valid "iterator" types (for some formal definition of that term) or valid "arithmetic" type (which could be a broader notion than the set of built-in arithmetic types).

A *concept* is a named set of constraints on one or more template parameters. While C++11 was being developed, a very rich concept system was designed for it, but integrating the feature into the language specification ...

Code Snippets

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

```
requires LessThanComparable<T>
```

```
class Person
{
private:
    std::string name;
public:
    template<typename STR>
    requires std::is_convertible_v<STR, std::string>
    explicit Person(STR&& n)
        : name(std::forward<STR>(n)) {
            std::cout << "TMPL-CONSTR for '" << name << "'\n";
    }
    ...
};
```

```
template<typename Seq>
    requires Sequence<Seq> &&
        EqualityComparable<typename Seq::value_type>
typename Seq::iterator find(Seq const& seq,
                           typename Seq::value_type const& val)
{
    return std::find(seq.begin(), seq.end(), val);
}
```

```
template<typename T>
    requires Integral<T> || 
        FloatingPoint<T>
T power(T b, T p);
```

```
template<typename T, typename U>
    requires SomeConcept<T, U>
auto f(T x, U y) -> decltype(x+y)
```

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

Bibliography

This bibliography 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 do not claim that our list is close to being comprehensive. However, we do find that the resources 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.tplbook.com>

Glossary

This glossary is a compilation of the most important technical terms that are used in this book. See [*StroustrupGlossary*] for a comprehensive, 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 ...

Index

- > [249](#)
<
 parsing [225](#)
>
 in template argument list [50](#)
 parsing [225](#)
>>
 versus > > [28](#), [226](#)
[] [685](#)

A

ABC [759](#), see abstract base class
about the book [249](#)
Abrahams, David [515](#), [547](#), [573](#)
AbrahamsGurtovoyMeta [750](#)
abstract base class [369](#)
 as concept [377](#)
abstract class [759](#)
ACCU [750](#)
actual parameter [155](#)
Adamczyk, Steve [321](#), [352](#)
adapter

iterator [505](#)
add_const [729](#)
add_cv [729](#)
add_lvalue_reference [730](#)
add_pointer [730](#)
addressof [166](#), [737](#)
add_rvalue_reference [730](#)
add_volatile [729](#)
ADL [217](#), [218](#), [219](#), [759](#)
aggregate [692](#)
 template [43](#)
 trait [711](#)
Alexandrescu, Andrei [266](#), [397](#), [463](#), [547](#), [573](#), [601](#), [628](#)
AlexandrescuAdHocVisitor [750](#)
AlexandrescuDesign [750](#)
AlexandrescuDiscriminatedUnions [750](#)
algorithm specialization [465](#), [557](#)
alias declaration [38](#)
alias template [39](#), [312](#), ...

Code Snippets

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

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

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

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

```
using CPTR = char* const; // constant pointer to chars
```

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

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

Code Snippets

basics/max1.hpp

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

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

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

basics/max1.cpp

```
#include "max1.hpp"
#include <iostream>
#include <string>
```

```
double f1 = 3.4;
double f2 = -6.7;
std::cout << "max(f1,f2): " << ::max(f1,f2) << '\n';

std::string s1 = "mathematics";
std::string s2 = "math";
std::cout << "max(s1,s2): " << ::max(s1,s2) << '\n';
}
```

```
max(7,i): 42  
max(f1,f2): 3.4  
max(s1,s2): mathematics
```

```
double max (double, double);
std::string max (std::string, std::string);
```

Code Snippets

basics/stack1.hpp

```
#include <vector>
#include <cassert>

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

public:
    void push(T const& elem); // push element
    void pop();               // pop element
    T const& 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 ()
{
    assert(!elems.empty());
    elems.pop_back();          // remove last element
}

template<typename T>
T const& Stack<T>::top () const
{
    assert(!elems.empty());
    return elems.back();        // return copy of last element
}
```

```
template<typename T>
class Stack {
    ...
};
```

```
template<class T>
class Stack {
    ...
};
```

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

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

```
template<typename T>
class Stack {

    ...
    Stack (Stack const&);           // copy constructor
    Stack& operator= (Stack const&); // assignment operator
    ...

};
```

```
template<typename T>
class Stack {

...
    Stack (Stack<T> const&);           // copy constructor
    Stack<T>& operator= (Stack<T> const&); // assignment operator
...
};
```

Code Snippets

basics/stacknontype.hpp

```
#include <array>
#include <cassert>

template<typename T, std::size_t Maxsize>
class Stack {
private:
    std::array<T,Maxsize> elems; //elements
    std::size_t numElems;        //current number of elements
public:
    Stack();                    //constructor
    void push(T const& elem); //push element
    void pop();                //pop element
    T const& top() const;      //return top element
    bool empty() const {       //return whether the stack is empty
        return numElems == 0;
    }
    std::size_t size() const { //return current number of elements
        return numElems;
    }
};
```

```
template<typename T, std::size_t Maxsize>
Stack<T,Maxsize>::Stack ()
: numElems(0)           // start with no elements
{
    // nothing else to do
}

template<typename T, std::size_t Maxsize>
void Stack<T,Maxsize>::push (T const& elem)
{
    assert(numElems < Maxsize);
    elems[numElems] = elem;    // append element
    ++numElems;               // increment number of elements
}
```

```
template<typename T, std::size_t Maxsize>
void Stack<T,Maxsize>::pop ()
{
    assert(!elems.empty());
    --numElems;           //decrement number of elements
}

template<typename T, std::size_t Maxsize>
T const& Stack<T,Maxsize>::top () const
{
    assert(!elems.empty());
    return elems[numElems-1]; //return last element
}
```

```
template<typename T, std::size_t Maxsize>
class Stack {
    private:
        std::array<T,Maxsize> elems; //elements
        ...
};
```

```
template<typename T, std::size_t Maxsize>
void Stack<T,Maxsize>::push (T const& elem)
{
    assert(numElems < Maxsize);
    elems[numElems] = elem;      //append element
    ++numElems;                  //increment number of elements
}
```

basics/stacknontype.cpp

```
#include "stacknontype.hpp"
#include <iostream>
#include <string>

int main()
{
    Stack<int,20>      int20Stack;      // stack of up to 20 ints
    Stack<int,40>      int40Stack;      // stack of up to 40 ints
    Stack<std::string,40> stringStack;  // stack of up to 40 strings

    // manipulate stack of up to 20 ints
    int20Stack.push(7);
    std::cout << int20Stack.top() << '\n';
    int20Stack.pop();

    // manipulate stack of up to 40 strings
    stringStack.push("hello");
    std::cout << stringStack.top() << '\n';
    stringStack.pop();
}
```

```
template<typename T = int, std::size_t Maxsize = 100>
class Stack {
    ...
};
```

Code Snippets

basics/varprint1.hpp

```
#include <iostream>

void print ()
{
}

template<typename T, typename... Types>
void print (T firstArg, Types... args)
{
    std::cout << firstArg << '\n'; // print first argument
    print(args...);                // call print() for remaining arguments
}
```

```
void print (T firstArg, Types... args)
```

```
template<typename T, typename... Types>
```

```
std::string s("world");
print (7.5, "hello", s);
```

7.5
hello
world

```
print<double, char const*, std::string> (7.5, "hello", s);
```

```
print<char const*, std::string> ("hello", s);
```

Code Snippets

```
template<typename T>
class MyClass {
public:
    ...
    void foo() {
        typename T::SubType* ptr;
    }
};
```

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 << '\n';
}
```

```
class stlcontainer {
public:
    using iterator = ...;           // iterator for read/write access
    using const_iterator = ...;     // iterator for read access
    ...
};
```

```
typename T::const_iterator pos;
```

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

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

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

Code Snippets

basics/move1.cpp

```
#include <utility>
#include <iostream>

class X {
    ...
};

void g (X&) {
    std::cout << "g() for variable\n";
}
void g (X const&) {
    std::cout << "g() for constant\n";
}
void g (X&&) {
    std::cout << "g() for movable object\n";
}

// let f() forward argument val to g():
void f (X& val) {
    g(val);           // val is non-const lvalue => calls g(X&)
}
```

```

void f (X const& val) {
    g(val);           //val is const lvalue => calls g(X const&)
}
void f (X&& val) {
    g(std::move(val)); //val is non-const lvalue => needs std::move() to call g(X&&)
}

int main()
{
    X v;           //create variable
    X const c;     //create constant

    f(v);          //f() for nonconstant object calls f(X&) => calls g(X&)
    f(c);          //f() for constant object calls f(X const&) => calls g(X const&)
    f(X());        //f() for temporary calls f(X&&) => calls g(X&&)
    f(std::move(v)); //f() for movable variable calls f(X&&) => calls g(X&&)
}

```

```
void f (X& val) {
    g(val);           // val is non-const lvalue => calls g(X&)
}
void f (X const& val) {
    g(val);           // val is const lvalue => calls g(X const&)
}
void f (X&& val) {
    g(std::move(val)); // val is non-const lvalue => needs std::move() to call g(X&&)
}
```

```
template<typename T>
void f (T val) {
    g(T);
}
```

```
template<typename T>
void f (T&& val) {
    g(std::forward<T>(val)); // perfect forward val to g()
}
```

basics/move2.cpp

```
#include <utility>
#include <iostream>

class X {
    ...
};

void g (X&) {
    std::cout << "g() for variable\n";
}
void g (X const&) {
    std::cout << "g() for constant\n";
}
void g (X&&) {
    std::cout << "g() for movable object\n";
}
```

```

// let f() perfect forward argument val to g():
template<typename T>
void f (T&& val) {
    g(std::forward<T>(val)); // call the right g() for any passed argument val
}

int main()
{
    X v;           // create variable
    X const c;    // create constant

    f(v);         // f() for variable calls f(X&) => calls g(X&)
    f(c);         // f() for constant calls f(X const&) => calls g(X const&)
    f(X());       // f() for temporary calls f(X&&) => calls g(X&&)
    f(std::move(v)); // f() for move-enabled variable calls f(X&&) => calls g(X&&)
}

```

Code Snippets

```
template<typename T>
void printV (T arg) {
    ...
}
```

```
void printV (int arg) {  
    ...  
}
```

```
void printV (std::string arg)
{
    ...
}
```

```
std::string returnString();
std::string s = "hi";
printV(s);           // copy constructor
printV(std::string("hi")); // copying usually optimized away (if not, move constructor)
printV(returnString()); // copying usually optimized away (if not, move constructor)
printV(std::move(s)); // move constructor
```

```
template<typename T>
void printV (T arg) {
    ...
}
```

```
std::string const c = "hi";
printV(c);           // c decays so that arg has type std::string

printV("hi");        // decays to pointer so that arg has type char const*

int arr[4];
printV(arr);         // decays to pointer so that arg has type char const*
```

```
void printV (char const* arg)
{
    ...
}
```

Code Snippets

basics/isprime.hpp

```
template<unsigned p, unsigned d> // p: number to check, d: current divisor
struct DoIsPrime {
    static constexpr bool value = (p%d != 0) && DoIsPrime<p,d-1>::value;
};

template<unsigned p> // end recursion if divisor is 2
struct DoIsPrime<p,2> {
    static constexpr bool value = (p%2 != 0);
};

template<unsigned p> // primary template
struct IsPrime {
    // start recursion with divisor from p/2:
    static constexpr bool value = DoIsPrime<p,p/2>::value;
};

// special cases (to avoid endless recursion with template instantiation):
template<>
struct IsPrime<0> { static constexpr bool value = false; };
template<>
struct IsPrime<1> { static constexpr bool value = false; };
template<>
struct IsPrime<2> { static constexpr bool value = true; };
template<>
struct IsPrime<3> { static constexpr bool value = true; };
```

```
9%4!=0 && DoIsPrime<9,3>::value
```

```
9%4!=0 && 9%3!=0 && DoIsPrime<9,2>::value
```

$9\%4!=0 \quad \&\& \quad 9\%3!=0 \quad \&\& \quad 9\%2!=0$

basics/isprime11.hpp

```
constexpr bool
doIsPrime (unsigned p, unsigned d) // p: number to check, d: current divisor
{
    return d!=2 ? (p%d!=0) && doIsPrime(p,d-1) // check this and smaller divisors
                : (p%2!=0); // end recursion if divisor is 2
}

constexpr bool isPrime (unsigned p)
{
    return p < 4 ? !(p<2) // handle special cases
                  : doIsPrime(p,p/2); // start recursion with divisor from p/2
}
```

basics/isprime14.hpp

```
constexpr bool isPrime (unsigned int p)
{
    for (unsigned int d=2; d<=p/2; ++d) {
        if (p % d == 0) {
            return false; //found divisor without remainder
        }
    }
    return p > 1;      //no divisor without remainder found
}
```

```
constexpr bool b1 = isPrime(9); // evaluated at compile time
```

Code Snippets

basics/myfirst.hpp

```
#ifndef MYFIRST_HPP
#define MYFIRST_HPP

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

#endif // MYFIRST_HPP
```

basics/myfirst.cpp

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

// implementation/definition of template
template<typename T>
void printTypeof (T const& x)
{
    std::cout << typeid(x).name() << '\n';
}
```

basics/myfirstmain.cpp

```
#include "myfirst.hpp"

// use of the template
int main()
{
    double ice = 3.0;
    printTypeof(ice); // call function template for type double
}
```

basics/myfirst2.hpp

```
#ifndef MYFIRST_HPP
#define MYFIRST_HPP

#include <iostream>
#include <typeinfo>

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

// implementation/definition of template
template<typename T>
void printTypeof (T const& x)
{
    std::cout << typeid(x).name() << '\n';
}

#endif // MYFIRST_HPP
```

basics/errornove1.cpp

```
#include <string>
#include <map>
#include <algorithm>

int main()
{
    std::map<std::string, double> coll;
    ...
    // find the first nonempty string in coll:
    auto pos = std::find_if (coll.begin(), coll.end(),
                           [] (std::string const& s) {
                               return s != "";
                           });
}
```

```

1 In file included from /cygdrive/p/gcc/gcc61/include/bits/stl_algobase.h:71:0,
2         from /cygdrive/p/gcc/gcc61/include/bits/char_traits.h:39,
3         from /cygdrive/p/gcc/gcc61/include/string:40,
4         from errornovel1.cpp:1:
5 /cygdrive/p/gcc/gcc61/include/bits/predefined_ops.h: In instantiation of 'bool __gnu_cxx
6 ::__ops::__Iter_pred<_Predicate>::operator()(_Iterator) [with _Iterator = std::__Rb_tree_i
7 terator<std::pair<const std::__cxx11::basic_string<char>, double> >; _Predicate = main()
8 ::<lambda(const string&)>]':
9 /cygdrive/p/gcc/gcc61/include/bits/stl_algo.h:104:42:   required from '_InputIterator
10 std::__find_if(_InputIterator, _InputIterator, _Predicate, std::input_iterator_tag)
11 [with _InputIterator = std::__Rb_tree_iterator<std::pair<const std::__cxx11::basic_string
12 <char>, double> >; _Predicate = __gnu_cxx::__ops::__Iter_pred<main()::<lambda(const
13 string&)> >]
14 /cygdrive/p/gcc/gcc61/include/bits/stl_algo.h:161:23:   required from '_Iterator std::__
15 find_if(_Iterator, _Iterator, _Predicate) [with _Iterator = std::__Rb_tree_iterator<std::
16 pair<const std::__cxx11::basic_string<char>, double> >; _Predicate = __gnu_cxx::__ops::__
17 Iter_pred<main()::<lambda(const string&)> >]
18 /cygdrive/p/gcc/gcc61/include/bits/stl_algo.h:3824:28:   required from '_IIter std::find
19 _if(_IIter, _IIter, _Predicate) [with _IIter = std::__Rb_tree_iterator<std::pair<const
20 std::__cxx11::basic_string<char>, double> >; _Predicate = main()::<lambda(const string&)
21 >]
22 errornovel1.cpp:13:29:   required from here
23 /cygdrive/p/gcc/gcc61/include/bits/predefined_ops.h:234:11: error: no match for call to
24   '(main()::<lambda(const string&)>) (std::pair<const std::__cxx11::basic_string<char>,
25   double>&)'
26   { return bool(_M_pred(*_it)); }
27   ^~~~~~
28 /cygdrive/p/gcc/gcc61/include/bits/predefined_ops.h:234:11: note: candidate: bool (*)(
29   const string& {aka bool (*)(const std::__cxx11::basic_string<char>&)} <conversion>
30 /cygdrive/p/gcc/gcc61/include/bits/predefined_ops.h:234:11: note:  candidate expects 2
31   arguments, 2 provided
32 errornovel1.cpp:11:52: note: candidate: main()::<lambda(const string&)>
33   [] (std::string const& s) {
34   ^
35 errornovel1.cpp:11:52: note:  no known conversion for argument 1 from 'std::pair<const
36   std::__cxx11::basic_string<char>, double>' to 'const string& {aka const std::__cxx11::
37   basic_string<char>&}'

```

```
auto pos = std::find_if (coll.begin(), coll.end(),
    [] (std::string const& s) {
        return s != "";
});
```

Code Snippets

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

template<>                                // explicit specialization
class MyClass<std::string, float> {
    ...
};
```

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

template<typename T> // partial specialization
class MyClass<bool,T> {
    ...
};
```

```
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
```

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

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

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

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

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

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

```
class C;           // C is an incomplete type
C const* cp;      // cp is a pointer to an incomplete type
extern C elems[10]; // elems has an incomplete type
extern int arr[]; // arr has an incomplete type
...
class C { };      // C now is a complete type (and therefore cp and elems
                  // no longer refer to an incomplete type)
int arr[10];      // arr now has a complete type
```

Code Snippets

basics/foreach.hpp

```
template<typename Iter, typename Callable>
void foreach (Iter current, Iter end, Callable op)
{
    while (current != end) { // as long as not reached the end
        op(*current);           // call passed operator for current element
        ++current;              // and move iterator to next element
    }
}
```

basics/foreach.cpp

```
#include <iostream>
#include <vector>
#include "foreach.hpp"

// a function to call:
void func(int i)
{
    std::cout << "func() called for: " << i << '\n';
}

// a function object type (for objects that can be used as functions):
class FuncObj {
public:
    void operator()(int i) const { //Note: const member function
        std::cout << "FuncObj::op() called for: " << i << '\n';
    }
};
```

```
int main()
{
    std::vector<int> primes = { 2, 3, 5, 7, 11, 13, 17, 19 };

    foreach(primes.begin(), primes.end(), //range
           func);                      //function as callable (decays to pointer)

    foreach(primes.begin(), primes.end(), //range
           &func);                     //function pointer as callable

    foreach(primes.begin(), primes.end(), //range
           FuncObj());                //function object as callable

    foreach(primes.begin(), primes.end(), //range
           [] (int i) {                  //lambda as callable
               std::cout << "lambda called for: " << i << '\n';
           });
}
```

```
op.operator()(*current); // call operator() with parameter *current for op
```

basics/foreachinvoke.hpp

```
#include <utility>
#include <functional>

template<typename Iter, typename Callable, typename... Args>
void foreach (Iter current, Iter end, Callable op, Args const&... args)
{
    while (current != end) {      // as long as not reached the end of the elements
        std::invoke(op,           // call passed callable with
                    args...,       // any additional args
                    *current);     // and the current element
        ++current;
    }
}
```

basics/foreachinvoke.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include "foreachinvoke.hpp"

// a class with a member function that shall be called
class MyClass {
public:
    void memfunc(int i) const {
        std::cout << "MyClass::memfunc() called for: " << i << '\n';
    }
};

int main()
{
    std::vector<int> primes = { 2, 3, 5, 7, 11, 13, 17, 19 };
```

```

// pass lambda as callable and an additional argument:
foreach(primes.begin(), primes.end(),           //elements for 2nd arg of lambda
       [](std::string const& prefix, int i) { //lambda to call
           std::cout << prefix << i << '\n';
       },
       "- value: ");                      //1st arg of lambda

// call obj.memfunc() for/with each elements in primes passed as argument
MyClass obj;
foreach(primes.begin(), primes.end(), //elements used as args
       &MyClass::memfunc,           //member function to call
       obj);                      //object to call memfunc() for
}

```

Code Snippets

details/definitions1.hpp

```
template<typename T> // a namespace scope class template
class Data {
public:
    static constexpr bool copyable = true;
    ...
};

template<typename T> // a namespace scope function template
void log (T x) {
    ...
}

template<typename T> // a namespace scope variable template (since C++14)
T zero = 0;

template<typename T> // a namespace scope variable template (since C++14)
bool dataCopyable = Data<T>::copyable;

template<typename T> // a namespace scope alias template
using DataList = Data<T*>;
```

details/definitions2.hpp

```
class Collection {
public:
    template<typename T> // an in-class member class template definition
    class Node {
        ...
    };

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

    template<typename T> // a member variable template (since C++14)
    static T zero = 0;

    template<typename T> // a member alias template
    using NodePtr = Node<T>*;
};
```

details/definitions3.hpp

```
template<typename T> // a namespace scope class template
class List {
public:
    List() = default; // because a template constructor is defined

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

    template<typename U> // a member function template
    List (List<U> const&); // (constructor)

    template<typename U> // a member variable template (since C++14)
    static U zero;
};
```

```
template<typename T> // out-of-class member class template definition
template<typename U>
class List<T>::Handle {
    ...
};

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

template<typename T> // out-of-class static data member template definition
template<typename U>
U List<T>::zero = 0;
```

```
template<typename T>
union AllocChunk {
    T object;
    unsigned char bytes[sizeof(T)];
};
```

```
template<typename T>
void report_top (Stack<T> const&, int number = 10);

template<typename T>
void fill (Array<T>&, T const& = T{}); //T{} is zero for built-in types
```

```
template<typename T>
void fill (Array<T>&, T const& = T()); //T() is zero for built-in types
```

Code Snippets

```
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
}
```

```
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
}
```

```
namespace BigMath {
    class BigNumber {
        ...
    };
    bool operator < (BigNumber const&, BigNumber const&);
    ...
}

using BigMath::BigNumber;

void g (BigNumber const& a, BigNumber const& b)
{
    ...
    BigNumber x = ::max(a,b);
    ...
}
```

details/adl.cpp

```
#include <iostream>

namespace X {
    template<typename T> void f(T);
}

namespace N {
    using namespace X;
    enum E { e1 };
    void f(E) {
        std::cout << "N::f(N::E) called\n";
    }
}

void f(int)
{
    std::cout << "::f(int) called\n";
}

int main()
{
    ::f(N::e1); // qualified function name: no ADL
    f(N::e1); // ordinary lookup finds ::f() and ADL finds N::f(),
} // the latter is preferred
```

```
template<typename T>
class C {
    ...
    friend void f();
    friend void f(C<T> const&);
    ...
};

void g (C<int>* p)
{
    f();      // is f() visible here?
    f(*p);   // is f(C<int> const&) visible here?
}
```

details/inject.cpp

```
#include <iostream>

int C;
class C {
    private:
        int i[2];
    public:
        static int f() {
            return sizeof(C);
        }
};

int f()
{
    return sizeof(C);
}

int main()
{
    std::cout << "C::f() = " << C::f() << ','
           << " ::f() = " << ::f() << '\n';
}
```

```
template<template<typename> class TT> class X {  
};  
  
template<typename T> class C {  
    C* a;           // OK: same as “C<T>* a;”  
    C<void>& b; // OK  
    X<C> c;       // OK: C without a template argument list denotes the template C  
    X<::C> d;     // OK: ::C is not the injected class name and therefore always  
                  //      denotes the template  
};
```

Code Snippets

```

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()
// in this translation unit

template<typename T>
void C<T>::f() // required definition due to #6
{
}

```

```
template<typename T>
class C {
public:
    C(int);           // a constructor that can be called with a single parameter
};                   // may be used for implicit conversions

void candidate(C<double>); // #1
void candidate(int) { }     // #2

int main()
{
    candidate(42); // both previous function declarations can be called
}
```

```
template<typename T> T f (T p) { return 2*p; }
decltype(f(2)) x = 2;
```

```
template<typename T> class Q {  
    using Type = typename T::Type;  
};  
  
Q<int>* p = 0;           // OK: the body of Q<int> is not substituted
```

```
template<typename T> T v = T::default_value();
decltype(v<int>) s; //OK: initializer of v<int> not instantiated
```

details/lazy1.hpp

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

template<int N>
class Danger {
    int arr[N];           // OK here, although would fail for N<=0
};

template<typename T, int N>
class Tricky {
public:
    void noBodyHere(Safe<T> = 3); // OK until usage of default value results in an error
    void inclass() {
        Danger<N> noBoomYet;      // OK until inclass() is used with N<=0
    }
    struct Nested {
        Danger<N> pfew;          // OK until Nested is used with N<=0
    };
    union {                   // due anonymous union:
        Danger<N> anonymous;   // OK until Tricky is instantiated with N<=0
        int align;
    };
    void unsafe(T (*p)[N]);    // OK until Tricky is instantiated with N<=0
    void error() {
        Danger<-1> boom;       // always ERROR (which not all compilers detect)
    }
};
```

details/lazy2.cpp

```
template<typename T>
class VirtualClass {
public:
    virtual ~VirtualClass() {}
    virtual T vmem(); // Likely ERROR if instantiated without definition
};

int main()
{
    VirtualClass<int> inst;
```

Code Snippets

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

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

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

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

```
template<typename T> void f(T);    //parameterized type P is T
template<typename T> void g(T&);   //parameterized type P is also T

double arr[20];
int const seven = 7;

f(arr);      //nonreference parameter: T is double*
g(arr);      //reference parameter:    T is double[20]
f(seven);    //nonreference parameter: T is int
g(seven);    //reference parameter:    T is int const
f(7);        //nonreference parameter: T is int
g(7);        //reference parameter:    T is int => ERROR: can't pass 7 to int&
```

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

```

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
}

```

details/fppm.cpp

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

template<int N>
void fppm(void (X<N>::*p)(typename X<N>::I));

int main()
{
    fppm(&X<33>::f); //fine: N deduced to be 33
}
```

```
template<typename T>
void f(X<Y<T>, Y<T>>);
void g()
{
    f(X<Y<int>, Y<int>>()); // OK
    f(X<Y<int>, Y<char>>()); // ERROR: deduction fails
}
```

Code Snippets

```

template<typename T>
class Array {
private:
    T* data;
    ...
public:
    Array(Array<T> const&);
    Array<T>& operator= (Array<T> const&);

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

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

```

```
template<typename T>
void genericAlgorithm(T* x, T* y)
{
    ...
    exchange(x, y); //How do we select the right algorithm?
    ...
}
```

```
template<typename T>
void exchange (Array<T>* a, Array<T>* b)
{
    T* p = &(*a)[0];
    T* q = &(*b)[0];
    for (std::size_t k = a->size(); k-- != 0; ) {
        exchange(p++, q++);
    }
}
```

details/funcoverload1.cpp

```
#include <iostream>
#include "funcoverload1.hpp"

int main()
{
    std::cout << f<int*>((int*)nullptr); // calls f<T>(T)
    std::cout << f<int>((int*)nullptr);   // calls f<T>(T*)
}
```

```
template<typename T1, typename T2>
void f1(T1, T2);

template<typename T1, typename T2>
void f1(T2, T1);

template<typename T>
long f2(T);

template<typename T>
char f2(T);
```

```
f1<T1 = char, T2 = char>(T1, T2)
```

```
f1<T1 = char, T2 = char>(T2, T1)
```

Code Snippets

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

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

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

int main()
{
    // instantiates Diagnoser<'S','u','r','p','r','i','s','e','!'>
    Diagnoser<"Surprise!">().print();
}
```

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

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

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

```
template<double Ratio>
class Converter {
public:
    static double convert (double val) {
        return val*Ratio;
    }
};

using InchToMeter = Converter<0.0254>;
```

```
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
```

```
template<typename T>
void add (T& x, int i); // a primary template

template<typename T1, typename T2>
void add (T1 a, T2 b); // another (overloaded) primary template

template<typename T>
void add<T*> (T*&, int); // Which primary template does this specialize?
```

```
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 {
    ...
};

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

Code Snippets

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;
    ...
    virtual ~GeoObj() = default;
};

// concrete geometric object class Circle
// - derived from GeoObj
```

poly/dynopoly.cpp

```
#include "dynahier.hpp"
#include <vector>

// draw any GeoObj
void myDraw (GeoObj const& obj)
{
    obj.draw();           // call draw() according to type of object
}

// compute distance of center of gravity between two GeoObjs
Coord distance (GeoObj const& x1, GeoObj const& x2)
{
    Coord c = x1.center_of_gravity() - x2.center_of_gravity();
    return c.abs();      // return coordinates as absolute values
}

// draw heterogeneous collection of GeoObjs
void drawElems (std::vector<GeoObj*> const& elems)
{
    for (std::size_type i=0; i<elems.size(); ++i) {
        elems[i]->draw(); // call draw() according to type of element
    }
}
```

```
void myDraw (GeoObj const& obj)      //GeoObj is abstract base class
{
    obj.draw();
}
```

```
template<typename GeoObj>
void myDraw (GeoObj const& obj)      // GeoObj is template parameter
{
    obj.draw();
}
```

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:
    void draw() const;
    Coord center_of_gravity() const;
    ...
};
...
```

poly/staticpoly.cpp

```
#include "statichier.hpp"
#include <vector>

// draw any GeoObj
template<typename GeoObj>
void myDraw (GeoObj const& obj)
{
    obj.draw();      // call draw() according to type of object
}

// compute distance of center of gravity between two GeoObjs
template<typename GeoObj1, typename GeoObj2>
Coord distance (GeoObj1 const& x1, GeoObj2 const& x2)
{
    Coord c = x1.center_of_gravity() - x2.center_of_gravity();
    return c.abs(); // return coordinates as absolute values
}

// draw homogeneous collection of GeoObjs
template<typename GeoObj>
void drawElems (std::vector<GeoObj> const& elems)
{
    for (unsigned i=0; i<elems.size(); ++i) {
        elems[i].draw();      // call draw() according to type of element
    }
}
```

```
distance(l,c); //distance<Line,Circle>(GeoObj1&,GeoObj2&)
```

Code Snippets

traits/accum1.hpp

```
#ifndef ACCUM_HPP
#define ACCUM_HPP

template<typename T>
T accum (T const* beg, T const* end)
{
    T total{}; //assume this actually creates a zero value
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}

#endif //ACCUM_HPP
```

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, num+5) / 5
        << '\n';

    // create array of character values
    char name[] = "templates";
    int length = sizeof(name)-1;

    // (try to) print average character value
    std::cout << "the average value of the characters in \""
        << name << "\" is "
        << accum(name, name+length) / length
        << '\n';
}
```

```
int num[] = { 1, 2, 3, 4, 5 };  
...  
accum(num0, num+5)
```

the average value of the integer values is 3

the average value of the characters in "templates" is -5

traits/accumtraits2.hpp

```
template<typename T>
struct AccumulationTraits;

template<>
struct AccumulationTraits<char> {
    using AccT = int;
};

template<>
struct AccumulationTraits<short> {
    using AccT = int;
};

template<>
struct AccumulationTraits<int> {
    using AccT = long;
};

template<>
struct AccumulationTraits<unsigned int> {
    using AccT = unsigned long;
};

template<>
struct AccumulationTraits<float> {
    using AccT = double;
};
```

traits/accum2.hpp

```
#ifndef ACCUM_HPP
#define ACCUM_HPP

#include "accumtraits2.hpp"

template<typename T>
auto accum (T const* beg, T const* end)
{
    // return type is traits of the element type
    using AccT = typename AccumulationTraits<T>::AccT;

    AccT total{}; //assume this actually creates a zero value
    while (beg != end) {
        total += *beg;
        ++beg;
    }
    return total;
}

#endif //ACCUM_HPP
```

the average value of the integer values is 3

the average value of the characters in "templates" is 108

Code Snippets

```
template<typename T> void f(T*);  
template<typename T> void f(Array<T>);
```

```
template<typename Number> void f(Number);      // only for numbers
template<typename Container> void f(Container); // only for containers
```

```
template<typename T>
void swap(T& x, T& y)
{
    T tmp(x);
    x = y;
    y = tmp;
}
```

```
template<typename T>
void swap(Array<T>& x, Array<T>& y)
{
    swap(x.ptr, y.ptr);
    swap(x.len, y.len);
}
```

```
template<typename InputIterator, typename Distance>
void advanceIter(InputIterator& x, Distance n)
{
    while (n > 0) { //linear time
        ++x;
        --n;
    }
}
```

```
template<typename RandomAccessIterator, typename Distance>
void advanceIter(RandomAccessIterator& x, Distance n) {
    x += n;           // constant time
}
```

```
template<typename Iterator, typename Distance>
void advanceIterImpl(Iterator& x, Distance n, std::input_iterator_tag)
{
    while (n > 0) { //linear time
        ++x;
        --n;
    }
}

template<typename Iterator, typename Distance>
void advanceIterImpl(Iterator& x, Distance n,
                     std::random_access_iterator_tag) {
    x += n;           //constant time
}
```

Code Snippets

inherit/empty.cpp

```
#include <iostream>

class EmptyClass {
};

int main()
{
    std::cout << "sizeof(EmptyClass): " << sizeof(EmptyClass) << '\n';
}
```

```
ZeroSizedT z[10];  
...  
&z[i] - &z[j]      // compute distance between pointers/addresses
```

inherit/ebco1.cpp

```
#include <iostream>

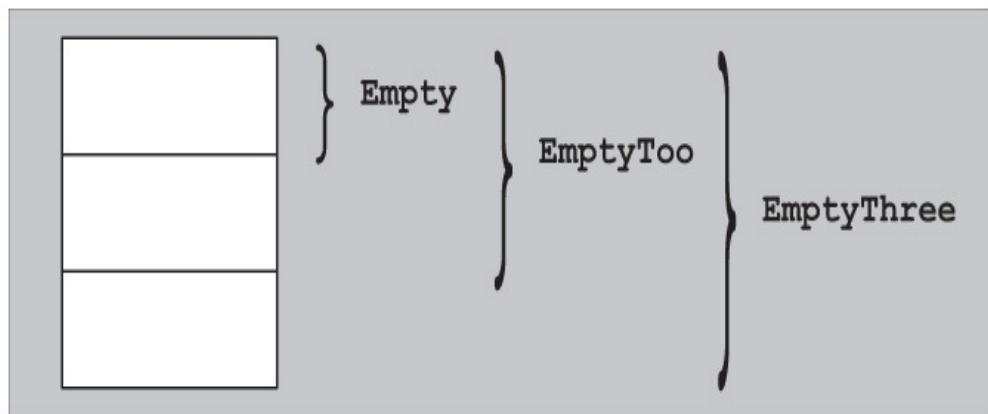
class Empty {
    using Int = int; // type alias members don't make a class nonempty
};

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';
}
```





inherit/ebc02.cpp

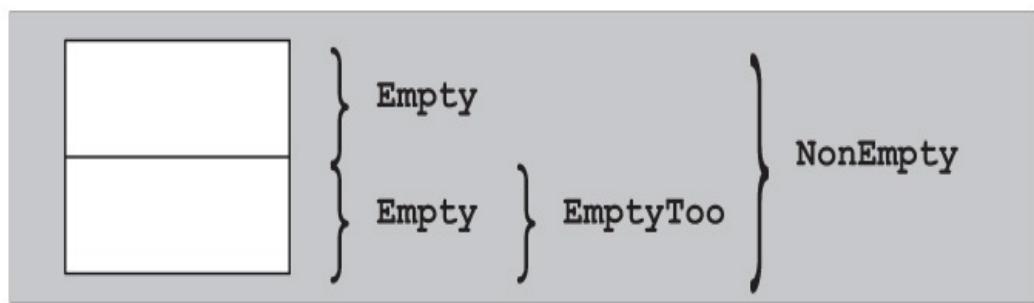
```
#include <iostream>

class Empty {
    using Int = int; // type alias 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';
}
```



Code Snippets

bridge/forupto1.cpp

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

template<typename F>
void forUpTo(int n, F f)
{
    for (int i = 0; i != n; ++i)
    {
        f(i); //call passed function f for i
    }
}

void printInt(int i)
{
    std::cout << i << ' ';
}

int main()
{
    std::vector<int> values;
```

```
// insert values from 0 to 4:  
forUpTo(5,  
    [&values](int i) {  
        values.push_back(i);  
    });  
  
// print elements:  
forUpTo(5,  
    printInt);      // prints 0 1 2 3 4  
std::cout << '\n';  
}
```

bridge/forupto2.hpp

```
void forUpTo(int n, void (*f)(int))
{
    for (int i = 0; i != n; ++i)
    {
        f(i); // call passed function f for i
    }
}
```

```
forUpTo(5,  
    printInt);           // OK: prints 0 1 2 3 4  
  
forUpTo(5,  
    [&values](int i) {  // ERROR: lambda not convertible to a function pointer  
        values.push_back(i);  
    });
```

bridge/forupto3.hpp

```
#include <functional>

void forUpTo(int n, std::function<void(int)> f)
{
    for (int i = 0; i != n; ++i)
    {
        f(i); //call passed function f for i
    }
}
```

bridge/forupto4.cpp

```
#include "functionptr.hpp"
#include <vector>
#include <iostream>

void forUpTo(int n, FunctionPtr<void(int)> f)
{
    for (int i = 0; i != n; ++i)
    {
        f(i); // call passed function f for i
    }
}

void printInt(int i)
{
    std::cout << i << ' ';
}

int main()
{
    std::vector<int> values;

    // insert values from 0 to 4:
    forUpTo(5,
            [&values](int i) {
                values.push_back(i);
            });

    // print elements:
    forUpTo(5,
            printInt); // prints 0 1 2 3 4
    std::cout << '\n';
}
```

bridge/functionptr.hpp

```
// primary template:  
template<typename Signature>  
class FunctionPtr;  
  
// partial specialization:  
template<typename R, typename... Args>  
class FunctionPtr<R(Args...)>  
{  
private:  
    FunctorBridge<R, Args...>* bridge;  
public:  
    // constructors:  
    FunctionPtr() : bridge(nullptr) {  
    }  
    FunctionPtr(FunctionPtr const& other);      // see functionptr-cpinv.hpp  
    FunctionPtr(FunctionPtr& other)  
        : FunctionPtr(static_cast<FunctionPtr const&>(other)) {  
    }  
    FunctionPtr(FunctionPtr&& other) : bridge(other.bridge) {  
        other.bridge = nullptr;  
    }
```

Code Snippets

meta/sqrtconstexpr.hpp

```
template<typename T>
constexpr T sqrt(T x)
{
    // handle cases where x and its square root are equal as a special case to simplify
    // the iteration criterion for larger x:
    if (x <= 1) {
        return x;
    }

    // repeatedly determine in which half of a [lo, hi] interval the square root of x is located,
    // until the interval is reduced to just one value:
    T lo = 0, hi = x;
    for (;;) {
        auto mid = (hi+lo)/2, midSquared = mid*mid;
        if (lo+1 >= hi || midSquared == x) {
            // mid must be the square root:
            return mid;
        }
        // continue with the higher/lower half-interval:
        if (midSquared < x) {
            lo = mid;
        }
        else {
            hi = mid;
        }
    }
}
```

```
static_assert(sqrt(25) == 5, "");      // OK (evaluated at compile time)
static_assert(sqrt(40) == 6, "");      // OK (evaluated at compile time)

std::array<int, sqrt(40)+1> arr;    // declares array of 7 elements (compile time)
```

```
long long l = 53478;  
std::cout << sqrt(l) << '\n';           //prints 231 (evaluated at run time)
```

meta/removeallextents.hpp

```
// primary template: in general we yield the given type:  
template<typename T>  
struct RemoveAllExtentsT {  
    using Type = T;  
};  
  
// partial specializations for array types (with and without bounds):  
template<typename T, std::size_t SZ>  
struct RemoveAllExtentsT<T[SZ]> {  
    using Type = typename RemoveAllExtentsT<T>::Type;  
};  
template<typename T>  
struct RemoveAllExtentsT<T[]> {  
    using Type = typename RemoveAllExtentsT<T>::Type;  
};  
  
template<typename T>  
using RemoveAllExtents = typename RemoveAllExtentsT<T>::Type;
```

```
RemoveAllExtents<int []>           // yields int
RemoveAllExtents<int [5] [10]>        // yields int
RemoveAllExtents<int [] [10]>         // yields int
RemoveAllExtents<int (*)[5]>          // yields int (*) [5]
```

```
namespace std {
    template<typename T, size_t N> struct array;
}
```

```
template<typename T, std::size_t N>
auto dotProduct(std::array<T, N> const& x, std::array<T, N> const& y)
{
    T result{};
    for (std::size_t k = 0; k < N; ++k) {
        result += x[k] * y[k];
    }
    return result;
}
```

Code Snippets

typelist/typelist.hpp

```
template<typename... Elements>
class Typelist
{  
};
```

```
using SignedIntegralTypes =  
    Typelist<signed char, short, int, long, long long>;
```

typelist/typelistfront.hpp

```
template<typename List>
class FrontT;

template<typename Head, typename... Tail>
class FrontT<Typelist<Head, Tail...>>
{
public:
    using Type = Head;
};

template<typename List>
using Front = typename FrontT<List>::Type;
```

typelist/typelistpopfront.hpp

```
template<typename List>
class PopFrontT;

template<typename Head, typename... Tail>
class PopFrontT<Typelist<Head, Tail...>> {
public:
    using Type = Typelist<Tail...>;
};

template<typename List>
using PopFront = typename PopFrontT<List>::Type;
```

TypeList<short, int, long, long long>

typelist/typelistpushfront.hpp

```
template<typename List, typename NewElement>
class PushFrontT;

template<typename... Elements, typename NewElement>
class PushFrontT<Typelist<Elements...>, NewElement> {
public:
    using Type = Typelist<NewElement, Elements...>;
};

template<typename List, typename NewElement>
using PushFront = typename PushFrontT<List, NewElement>::Type;
```

```
PushFront<SignedIntegralTypes, bool>
```

Code Snippets

```
template<typename... Types>
class Tuple {
    ... // implementation discussed below
};

Tuple<int, double, std::string> t(17, 3.14, "Hello, World!");
```

tuples/tuple0.hpp

```
template<typename... Types>
class Tuple;

// recursive case:
template<typename Head, typename... Tail>
class Tuple<Head, Tail...>
{
private:
    Head head;
    Tuple<Tail...> tail;
public:
// constructors:
    Tuple() {
    }
```

```
Tuple(Head const& head, Tuple<Tail...> const& tail)
    : head(head), tail(tail) {
}
...

Head& getHead() { return head; }
Head const& getHead() const { return head; }
Tuple<Tail...>& getTail() { return tail; }
Tuple<Tail...> const& getTail() const { return tail; }
};

// basis case:
template<>
class Tuple<> {
    // no storage required
};
```

tuples/tupleget.hpp

```
// recursive case:  
template<unsigned N>  
struct TupleGet {  
    template<typename Head, typename... Tail>  
    static auto apply(Tuple<Head, Tail...> const& t) {  
        return TupleGet<N-1>::apply(t.getTail());  
    }  
};  
  
// basis case:  
template<>  
struct TupleGet<0> {  
    template<typename Head, typename... Tail>  
    static Head const& apply(Tuple<Head, Tail...> const& t) {  
        return t.getHead();  
    }  
};  
  
template<unsigned N, typename... Types>  
auto get(Tuple<Types...> const& t) {  
    return TupleGet<N>::apply(t);  
}
```

```
Tuple() {  
}  
  
Tuple(Head const& head, Tuple<Tail...> const& tail)  
    : head(head), tail(tail) {  
}
```

```
Tuple(Head const& head, Tail const&... tail)
    : head(head), tail(tail...) {
}
```

```
Tuple<int, double, std::string> t(17, 3.14, "Hello, World!");
```

Code Snippets

```
Variant<int, double, string> field;
```

variant/variant.cpp

```
#include "variant.hpp"
#include <iostream>
#include <string>

int main()
{
    Variant<int, double, std::string> field(17);
    if (field.is<int>()) {
        std::cout << "Field stores the integer "
              << field.get<int>() << '\n';
    }
    field = 42;      // assign value of same type
    field = "hello"; // assign value of different type
    std::cout << "Field now stores the string ?>"
              << field.get<std::string>() << "'\n";
}
```

Field stores the integer 17
Field now stores the string "hello"

variant/variantstorageastuple.hpp

```
template<typename... Types>
class Variant {
public:
    Tuple<Types...> storage;
    unsigned char discriminator;
};
```

variant/variantstorageasunion.hpp

```
template<typename... Types>
union VariantStorage;

template<typename Head, typename... Tail>
union VariantStorage<Head, Tail...> {
    Head head;
    VariantStorage<Tail...> tail;
};

template<>
union VariantStorage<> {
};
```

variant/variantstorage.hpp

```
#include <new> //for std::launder()

template<typename... Types>
class VariantStorage {
    using LargestT = LargestType<Typelist<Types...>>;
    alignas(Types...) unsigned char buffer[sizeof(LargestT)];
    unsigned char discriminator = 0;
public:
    unsigned char getDiscriminator() const { return discriminator; }
    void setDiscriminator(unsigned char d) { discriminator = d; }
    void* getRawBuffer() { return buffer; }
    const void* getRawBuffer() const { return buffer; }

    template<typename T>
    T* getBufferAs() { return std::launder(reinterpret_cast<T*>(buffer)); }
    template<typename T>
    T const* getBufferAs() const {
        return std::launder(reinterpret_cast<T const*>(buffer));
    }
};
```

variant/variantchoice.hpp

```
#include "findindexof.hpp"

template<typename T, typename... Types>
class VariantChoice {
    using Derived = Variant<Types...>;
    Derived& getDerived() { return *static_cast<Derived*>(this); }
    Derived const& getDerived() const {
        return *static_cast<Derived const*>(this);
    }
protected:
    // compute the discriminator to be used for this type
    constexpr static unsigned Discriminator =
        FindIndexOfT<Typelist<Types...>, T>::value + 1;
public:
    VariantChoice() { }
    VariantChoice(T const& value);           // see variantchoiceinit.hpp
    VariantChoice(T&& value);             // see variantchoiceinit.hpp
    bool destroy();                         // see variantchoicedestroy.hpp
    Derived& operator=(T const& value);    // see variantchoiceassign.hpp
    Derived& operator=(T&& value);        // see variantchoiceassign.hpp
};
```

Code Snippets

```
Array<double> x(1000), y(1000);  
...  
x = 1.2*x + x*y;
```

exprtmp/sarray1.hpp

```
#include <cstddef>
#include <cassert>

template<typename T>
class SArray {
public:
    // create array with initial size
    explicit SArray (std::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
std::size_t size() const {
    return storage_size;
}

// index operator for constants and variables
T const& operator[] (std::size_t idx) const {
    return storage[idx];
}
T& operator[] (std::size_t idx) {
    return storage[idx];
}

protected:
// init values with default constructor
void init() {
    for (std::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 (std::size_t idx = 0; idx<size(); ++idx) {
        storage[idx] = orig.storage[idx];
    }
}

private:
    T*           storage;      // storage of the elements
    std::size_t   storage_size; // number of elements
};
```

exprtmp/sarrayops1.hpp

```
// addition of two SArrays
template<typename T>
SArray<T> operator+ (SArray<T> const& a, SArray<T> const& b)
{
    assert(a.size()==b.size());
    SArray<T> result(a.size());
    for (std::size_t k = 0; k<a.size(); ++k) {
        result[k] = a[k]+b[k];
    }
    return result;
}

// multiplication of two SArrays
template<typename T>
SArray<T> operator* (SArray<T> const& a, SArray<T> const& b)
{
    assert(a.size()==b.size());
    SArray<T> result(a.size());
    for (std::size_t k = 0; k<a.size(); ++k) {
        result[k] = a[k]*b[k];
```

```
    }
    return result;
}

// multiplication of scalar and SArray
template<typename T>
SArray<T> operator* (T const& s, SArray<T> const& a)
{
    SArray<T> result(a.size());
    for (std::size_t k = 0; k < a.size(); ++k) {
        result[k] = s*a[k];
    }
    return result;
}

// multiplication of SArray and scalar
// addition of scalar and SArray
// addition of SArray and scalar

...
```

exprtmp/sarray1.cpp

```
#include "sarray1.hpp"
#include "sarrayops1.hpp"

int main()
{
    SArray<double> x(1000), y(1000);
    ...
    x = 1.2*x + x*y;
}
```

Code Snippets

```

template<typename T>
void clear (T& p)
{
    *p = 0; //assumes T is a pointer-like type
}

template<typename T>
void core (T& 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:
        using Index = int;
};

int main()
{
    Client mainClient;
    shell(mainClient);
}
```

```
template<typename T>
void ignore(T const&)
{
}

template<typename T>
void shell (T const& env)
{
    class ShallowChecks
    {
        void deref(typename T::Index ptr) {
            ignore(*ptr);
        }
    };
    typename T::Index i;
    middle(i);
}
```

```
static_assert(sizeof(void*) * CHAR_BIT == 64, "Not a 64-bit platform");
```

debugging/hasderef.hpp

```
#include <utility>           //for declval()
#include <type_traits>        //for true_type and false_type

template<typename T>
class HasDereference {
private:
    template<typename U> struct Identity;
    template<typename U> static std::true_type
        test(Identity<decltype(*std::declval<U>())>*);
    template<typename U> static std::false_type
        test(...);
public:
    static constexpr bool value = decltype(test<T>(nullptr))::value;
};
```

```
template<typename T>
void shell (T const& env)
{
    static_assert(HasDereference<T>::value, "T is not dereferenceable");

    typename T::Index i;
    middle(i);
}
```

```
template<typename T>
class C {
    static_assert(HasDereference<T>::value, "T is not dereferenceable");
    static_assert(std::is_default_constructible<T>::value,
                 "T is not default constructible");
    ...
};
```

Code Snippets

```
//== header.hpp:  
#ifdef DO_DEBUG  
    #define debug(x) std::cout << x << '\n'  
#else  
    #define debug(x)  
#endif  
  
void debugInit();  
//== myprog.cpp:  
#include "header.hpp"  
  
int main()  
{  
    debugInit();  
    debug("main()");  
}
```

```
//== translation unit 1:  
int counter;  
  
//== translation unit 2:  
int counter; // ERROR: defined twice (ODR violation)
```

```
//== translation unit 1:  
static int counter = 2; //unrelated to other translation units  
  
namespace {  
    void unique() //unrelated to other translation units  
    {  
    }  
}  
  
//== translation unit 1:  
static int counter = 0; //unrelated to other translation units  
  
namespace {  
    void unique() //unrelated to other translation units  
    {  
        ++counter;  
    }  
}  
  
int main()  
{  
    unique();  
}
```

```
#include <typeinfo>

class Decider {
#if defined(DYNAMIC)
    virtual ~Decider() {
    }
#endif
};

extern Decider d;

int main()
{
    char const* name = typeid(d).name();
    return (int)sizeof(d);
}
```

```
inline void f() {}
inline void f() {} //ERROR: duplicate definition
```

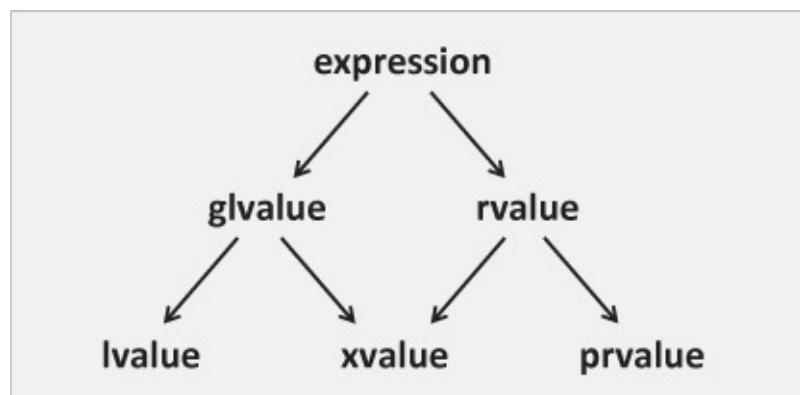
```
//== translation unit 1:  
class X {  
public:  
    X(int, int);  
    X(int, int, int);  
};  
  
X::X(int, int = 0)  
{  
}  
class D {  
    X x = 0;  
};  
  
D d1; //X(int, int) called by D()
```

```
//—— translation unit 2:  
class X {  
    public:  
        X(int, int);  
        X(int, int, int);  
};  
  
X::X(int, int = 0, int = 0)  
{  
}  
  
class D : public X {  
    X x = 0;  
};  
  
D d2; //X(int, int, int) called by D()
```

Code Snippets

```
int const x; // x is a nonmodifiable lvalue
x = 7;       // ERROR: modifiable lvalue required on the left
```

```
std::vector<int> v;  
v.front()           //yields an lvalue because the return type is an lvalue reference
```



```
int x = 3; // x here is a variable, not an lvalue. 3 is a prvalue initializing
           // the variable x.
int y = x; // x here is an lvalue. The evaluation of that lvalue expression does not
           // produce the value 3, but a designation of an object containing the value 3.
           // That lvalue is then converted to a prvalue, which is what initializes y.
```

```
class N {
public:
    N();
    N(N const&) = delete; //this class is neither copyable ...
    N(N&&) = delete;     //... nor movable
};

N make_N() {
    return N{};           //Always creates a conceptual temporary prior to C++17.
}                         //In C++17, no temporary is created at this point.

auto n = make_N();        //ERROR prior to C++17 because the prvalue needs a
                          //conceptual copy. OK since C++17, because n is
                          //initialized directly from the prvalue.
```

```
class X {  
};  
  
X v;  
X const c;  
  
void f(X const&); // accepts an expression of any value category  
void f(X&&); // accepts prvalues and xvalues only but is a better match  
//for those than the previous declaration  
  
f(v); // passes a modifiable lvalue to the first f()  
f(c); // passes a nonmodifiable lvalue to the first f()  
f(X()); // passes a prvalue (since C++17 materialized as xvalue) to the 2nd f()  
f(std::move(v)); // passes an xvalue to the second f()
```

```
if constexpr (std::is_lvalue_reference<decltype((e))>::value) {
    std::cout << "expression is lvalue\n";
}
else if constexpr (std::is_rvalue_reference<decltype((e))>::value) {
    std::cout << "expression is xvalue\n";
}
else {
    std::cout << "expression is prvalue\n";
}
```

Code Snippets

```
void display_num(int);      // #1
void display_num(double);   // #2

int main()
{
    display_num(399);       // #1 matches better than #2
    display_num(3.99);      // #2 matches better than #1
}
```

```
void combine(int, double);
void combine(long, int);

int main()
{
    combine(1, 2); //ambiguous!
}
```

```
int f1(int);           // #1
int f1(double);        // #2
f1(4);                // calls #1: perfect match (#2 requires a standard conversion)

int f2(int);           // #3
int f2(char);          // #4
f2(true);              // calls #3: match with promotion
                       //           (#4 requires stronger standard conversion)

class X {
    public:
        X(int);
};

int f3(X);             // #5
int f3(...);            // #6
f3(7);                // calls #5: match with user-defined conversion
                       //           (#6 requires a match with ellipsis)
```

```
template<typename T>
class MyString {
public:
    MyString(T const*); // converting constructor
    ...
};

template<typename T>
MyString<T> truncate(MyString<T> const&, int);
int main()
{
    MyString<char> str1, str2;
    str1 = truncate<char>("Hello World", 5); // OK
    str2 = truncate("Hello World", 5);        // ERROR
}
```

```
template<typename T> void strange(T&&, T&&);
template<typename T> void bizarre(T&&, double&&);

int main()
{
    strange(1.2, 3.4); // OK: with T deduced to double
    double val = 1.2;
    strange(val, val); // OK: with T deduced to double&
    strange(val, 3.4); // ERROR: conflicting deductions
    bizarre(val, val); // ERROR: lvalue val doesn't match double&&
}
```

```

#include <cstddef>

class BadString {
public:
    BadString(char const* );
    ...

// character access through subscripting:
    char& operator[] (std::size_t); // #1
    char const& operator[] (std::size_t) const;

// implicit conversion to null-terminated byte string:
    operator char* (); // #2
    operator char const* ();
    ...
};

int main()
{
    BadString str("correkt");
    str[5] = 'c'; //possibly an overload resolution ambiguity!
}

```

```
struct S {  
    void f1();      // implicit *this parameter is an lvalue reference (see below)  
    void f2() &&; // implicit *this parameter is an rvalue reference  
    void f3() &;   // implicit *this parameter is an lvalue reference  
};
```

Code Snippets

```
typename std::trait<...>::type
std::trait_t<...> // since C++14
```

```
std::trait<...>::value
std::trait<...>()
std::trait_v<...>
// implicit conversion to its type
// since C++17
```

utils/traits1.cpp

```
#include <type_traits>
#include <iostream>
int main()
{
    int i = 42;
    std::add_const<int>::type c = i;          // c is int const
    std::add_const_t<int> c14 = i;           // since C++14
    static_assert(std::is_const<decltype(c)>::value, "c should be const");

    std::cout << std::boolalpha;
    std::cout << std::is_same<decltype(c), int const>::value // true
        << '\n';
    std::cout << std::is_same_v<decltype(c), int const>       // since C++17
        << '\n';
    if (std::is_same<decltype(c), int const>{}) { // implicit conversion to bool
        std::cout << "same\n";
    }
}
```

```
namespace std {
    template<typename T, T val>
    struct integral_constant {
        static constexpr T value = val; // value of the trait
        using value_type = T; // type of the value
        using type = integral_constant<T, val>;
        constexpr operator value_type() const noexcept {
            return value;
        }
        constexpr value_type operator() () const noexcept { // since C++14
            return value;
        }
    };
}
```

```
namespace std {
    template<bool B>
    using bool_constant = integral_constant<bool, B>; //since C++17
    using true_type = bool_constant<true>;
    using false_type = bool_constant<false>;
}
```

utils/traits2.cpp

```
#include <type_traits>
#include <iostream>

int main()
{
    using namespace std;
    cout << boolalpha;

    using MyType = int;
    cout << is_const<MyType>::value << '\n'; //prints false

    using VT = is_const<MyType>::value_type; //bool
    using T = is_const<MyType>::type;          //integral_constant<bool, false>
    cout << is_same<VT,bool>::value << '\n'; //prints true
    cout << is_same<T, integral_constant<bool, false>>::value
        << '\n';                           //prints true
    cout << is_same<T, bool_constant<false>>::value
        << '\n';                         //prints true (not valid
                                         //prior to C++17)
```

```
auto ic = is_const<MyType>();           // object of trait type
cout << is_same<decltype(ic), is_const<int>>::value << '\n'; // true
cout << ic() << '\n';                  // function call (prints false)

static constexpr auto mytypeIsConst = is_const<MyType>{};
if constexpr(mytypeIsConst) {            // compile-time check since C++17 => false
    ...
}                                     // discarded statement
static_assert(!std::is_const<MyType>{}, "MyType should not be const");
}
```

Table of Contents

Title Page	4
Copyright Page	5
Dedication	6
Contents	7
Preface	9
Acknowledgments for the Second Edition	10
Acknowledgments for the First Edition	11
About This Book	12
What You Should Know Before Reading This Book	12
Overall Structure of the Book	12
How to Read This Book	12
Some Remarks About Programming Style	12
The C++11, C++14, and C++17 Standards	12
Example Code and Additional Information	12
Feedback	12
Part I: The Basics	13
1 Function Templates	14
1.1 A First Look at Function Templates	14
1.1.1 Defining the Template	14
1.1.2 Using the Template	14
1.1.3 Two-Phase Translation	14
1.2 Template Argument Deduction	14
1.3 Multiple Template Parameters	14
1.3.1 Template Parameters for Return Types	14
1.3.2 Deducing the Return Type	14
1.3.3 Return Type as Common Type	14
1.4 Default Template Arguments	14
1.5 Overloading Function Templates	14
1.6 But, Shouldn't We ...?	14
1.6.1 Pass by Value or by Reference?	14
1.6.2 Why Not inline?	14

1.6.3 Why Not <code>constexpr</code> ?	14
1.7 Summary	14
2 Class Templates	15
2.1 Implementation of Class Template Stack	15
2.1.1 Declaration of Class Templates	15
2.1.2 Implementation of Member Functions	15
2.2 Use of Class Template Stack	15
2.3 Partial Usage of Class Templates	15
2.3.1 Concepts	15
2.4 Friends	15
2.5 Specializations of Class Templates	15
2.6 Partial Specialization	15
2.7 Default Class Template Arguments	15
2.8 Type Aliases	15
2.9 Class Template Argument Deduction	15
2.10 Templatized Aggregates	15
2.11 Summary	15
3 Nontype Template Parameters	17
3.1 Nontype Class Template Parameters	17
3.2 Nontype Function Template Parameters	17
3.3 Restrictions for Nontype Template Parameters	17
3.4 Template Parameter Type <code>auto</code>	17
3.5 Summary	17
4 Variadic Templates	18
4.1 Variadic Templates	18
4.1.1 Variadic Templates by Example	18
4.1.2 Overloading Variadic and Nonvariadic Templates	18
4.1.3 Operator <code>sizeof...</code>	18
4.2 Fold Expressions	18
4.3 Application of Variadic Templates	18
4.4 Variadic Class Templates and Variadic Expressions	18
4.4.1 Variadic Expressions	18
4.4.2 Variadic Indices	18
4.4.3 Variadic Class Templates	18
4.4.4 Variadic Deduction Guides	18
4.4.5 Variadic Base Classes and using	18

4.5 Summary	18
5 Tricky Basics	19
5.1 Keyword typename	19
5.2 Zero Initialization	19
5.3 Using this->	19
5.4 Templates for Raw Arrays and String Literals	19
5.5 Member Templates	19
5.5.1 The .template Construct	19
5.5.2 Generic Lambdas and Member Templates	19
5.6 Variable Templates	19
5.7 Template Template Parameters	19
5.8 Summary	19
6 Move Semantics and enable_if<>	20
6.1 Perfect Forwarding	20
6.2 Special Member Function Templates	20
6.3 Disable Templates with enable_if<>	20
6.4 Using enable_if<>	20
6.5 Using Concepts to Simplify enable_if<> Expressions	20
6.6 Summary	20
7 By Value or by Reference?	21
7.1 Passing by Value	21
7.2 Passing by Reference	21
7.2.1 Passing by Constant Reference	21
7.2.2 Passing by Nonconstant Reference	21
7.2.3 Passing by Forwarding Reference	21
7.3 Using std::ref() and std:: cref()	21
7.4 Dealing with String Literals and Raw Arrays	21
7.4.1 Special Implementations for String Literals and Raw Arrays	21
7.5 Dealing with Return Values	21
7.6 Recommended Template Parameter Declarations	21
7.7 Summary	21
8 Compile-Time Programming	22
8.1 Template Metaprogramming	22
8.2 Computing with constexpr	22
8.3 Execution Path Selection with Partial Specialization	22

8.4 SFINAE (Substitution Failure Is Not An Error)	22
8.4.1 Expression SFINAE with decltype	22
8.5 Compile-Time if	22
8.6 Summary	22
9 Using Templates in Practice	23
9.1 The Inclusion Model	23
9.1.1 Linker Errors	23
9.1.2 Templates in Header Files	23
9.2 Templates and inline	23
9.3 Precompiled Headers	23
9.4 Decoding the Error Novel	23
9.5 Afternotes	23
9.6 Summary	23
10 Basic Template Terminology	24
10.1 “Class Template” or “Template Class”?	24
10.2 Substitution, Instantiation, and Specialization	24
10.3 Declarations versus Definitions	24
10.3.1 Complete versus Incomplete Types	24
10.4 The One-Definition Rule	24
10.5 Template Arguments versus Template Parameters	24
10.6 Summary	24
11 Generic Libraries	25
11.1 Callables	25
11.1.1 Supporting Function Objects	25
11.1.2 Dealing with Member Functions and Additional Arguments	25
11.1.3 Wrapping Function Calls	25
11.2 Other Utilities to Implement Generic Libraries	25
11.2.1 Type Traits	25
11.2.2 std::addressof()	25
11.2.3 std::declval()	25
11.3 Perfect Forwarding Temporaries	25
11.4 References as Template Parameters	25
11.5 Defer Evaluations	25
11.6 Things to Consider When Writing Generic Libraries	25
11.7 Summary	25

Part II: Templates in Depth	26
12 Fundamentals in Depth	27
12.1 Parameterized Declarations	27
12.1.1 Virtual Member Functions	27
12.1.2 Linkage of Templates	27
12.1.3 Primary Templates	27
12.2 Template Parameters	27
12.2.1 Type Parameters	27
12.2.2 Nontype Parameters	27
12.2.3 Template Template Parameters	27
12.2.4 Template Parameter Packs	27
12.2.5 Default Template Arguments	27
12.3 Template Arguments	27
12.3.1 Function Template Arguments	27
12.3.2 Type Arguments	27
12.3.3 Nontype Arguments	27
12.3.4 Template Template Arguments	27
12.3.5 Equivalence	27
12.4 Variadic Templates	27
12.4.1 Pack Expansions	27
12.4.2 Where Can Pack Expansions Occur?	27
12.4.3 Function Parameter Packs	27
12.4.4 Multiple and Nested Pack Expansions	27
12.4.5 Zero-Length Pack Expansions	27
12.4.6 Fold Expressions	27
12.5 Friends	27
12.5.1 Friend Classes of Class Templates	27
12.5.2 Friend Functions of Class Templates	27
12.5.3 Friend Templates	27
12.6 Afternotes	27
13 Names in Templates	28
13.1 Name Taxonomy	28
13.2 Looking Up Names	28
13.2.1 Argument-Dependent Lookup	28
13.2.2 Argument-Dependent Lookup of Friend Declarations	28

13.2.3 Injected Class Names	28
13.2.4 Current Instantiations	28
13.3 Parsing Templates	28
13.3.1 Context Sensitivity in Nontemplates	28
13.3.2 Dependent Names of Types	28
13.3.3 Dependent Names of Templates	28
13.3.4 Dependent Names in Using Declarations	28
13.3.5 ADL and Explicit Template Arguments	28
13.3.6 Dependent Expressions	28
13.3.7 Compiler Errors	28
13.4 Inheritance and Class Templates	28
13.4.1 Nondependent Base Classes	28
13.4.2 Dependent Base Classes	28
13.5 Afternotes	28
14 Instantiation	29
14.1 On-Demand Instantiation	29
14.2 Lazy Instantiation	29
14.2.1 Partial and Full Instantiation	29
14.2.2 Instantiated Components	29
14.3 The C++ Instantiation Model	29
14.3.1 Two-Phase Lookup	29
14.3.2 Points of Instantiation	29
14.3.3 The Inclusion Model	29
14.4 Implementation Schemes	29
14.4.1 Greedy Instantiation	29
14.4.2 Queried Instantiation	29
14.4.3 Iterated Instantiation	29
14.5 Explicit Instantiation	29
14.5.1 Manual Instantiation	29
14.5.2 Explicit Instantiation Declarations	29
14.6 Compile-Time if Statements	29
14.7 In the Standard Library	29
14.8 Afternotes	29
15 Template Argument Deduction	30
15.1 The Deduction Process	30
15.2 Deduced Contexts	30

15.3 Special Deduction Situations	30
15.4 Initializer Lists	30
15.5 Parameter Packs	30
15.5.1 Literal Operator Templates	30
15.6 Rvalue References	30
15.6.1 Reference Collapsing Rules	30
15.6.2 Forwarding References	30
15.6.3 Perfect Forwarding	30
15.6.4 Deduction Surprises	30
15.7 SFINAE (Substitution Failure Is Not An Error)	30
15.7.1 Immediate Context	30
15.8 Limitations of Deduction	30
15.8.1 Allowable Argument Conversions	30
15.8.2 Class Template Arguments	30
15.8.3 Default Call Arguments	30
15.8.4 Exception Specifications	30
15.9 Explicit Function Template Arguments	30
15.10 Deduction from Initializers and Expressions	30
15.10.1 The <code>auto</code> Type Specifier	30
15.10.2 Expressing the Type of an Expression with <code>decltype</code>	30
15.10.3 <code>decltype(auto)</code>	30
15.10.4 Special Situations for <code>auto</code> Deduction	30
15.10.5 Structured Bindings	30
15.10.6 Generic Lambdas	30
15.11 Alias Templates	30
15.12 Class Template Argument Deduction	30
15.12.1 Deduction Guides	30
15.12.2 Implicit Deduction Guides	30
15.12.3 Other Subtleties	30
15.13 Afternotes	30
16 Specialization and Overloading	31
16.1 When “Generic Code” Doesn’t Quite Cut It	31
16.1.1 Transparent Customization	31
16.1.2 Semantic Transparency	31
16.2 Overloading Function Templates	31
16.2.1 Signatures	31

16.2.2 Partial Ordering of Overloaded Function Templates	31
16.2.3 Formal Ordering Rules	31
16.2.4 Templates and Nontemplates	31
16.2.5 Variadic Function Templates	31
16.3 Explicit Specialization	31
16.3.1 Full Class Template Specialization	31
16.3.2 Full Function Template Specialization	31
16.3.3 Full Variable Template Specialization	31
16.3.4 Full Member Specialization	31
16.4 Partial Class Template Specialization	31
16.5 Partial Variable Template Specialization	31
16.6 Afternotes	31
17 Future Directions	32
17.1 Relaxed typename Rules	32
17.2 Generalized Nontype Template Parameters	32
17.3 Partial Specialization of Function Templates	32
17.4 Named Template Arguments	32
17.5 Overloaded Class Templates	32
17.6 Deduction for Nonfinal Pack Expansions	32
17.7 Regularization of void	32
17.8 Type Checking for Templates	32
17.9 Reflective Metaprogramming	32
17.10 Pack Facilities	32
17.11 Modules	32
Part III: Templates and Design	33
18 The Polymorphic Power of Templates	34
18.1 Dynamic Polymorphism	34
18.2 Static Polymorphism	34
18.3 Dynamic versus Static Polymorphism	34
18.4 Using Concepts	34
18.5 New Forms of Design Patterns	34
18.6 Generic Programming	34
18.7 Afternotes	34
19 Implementing Traits	35
19.1 An Example: Accumulating a Sequence	35

19.1.1 Fixed Traits	35
19.1.2 Value Traits	35
19.1.3 Parameterized Traits	35
19.2 Traits versus Policies and Policy Classes	35
19.2.1 Traits and Policies: What's the Difference?	35
19.2.2 Member Templates versus Template Template Parameters	35
19.2.3 Combining Multiple Policies and/or Traits	35
19.2.4 Accumulation with General Iterators	35
19.3 Type Functions	35
19.3.1 Element Types	35
19.3.2 Transformation Traits	35
19.3.3 Predicate Traits	35
19.3.4 Result Type Traits	35
19.4 SFINAE-Based Traits	35
19.4.1 SFINAE Out Function Overloads	35
19.4.2 SFINAE Out Partial Specializations	35
19.4.3 Using Generic Lambdas for SFINAE	35
19.4.4 SFINAE-Friendly Traits	35
19.5 IsConvertibleT	35
19.6 Detecting Members	35
19.6.1 Detecting Member Types	35
19.6.2 Detecting Arbitrary Member Types	35
19.6.3 Detecting Nontype Members	35
19.6.4 Using Generic Lambdas to Detect Members	35
19.7 Other Traits Techniques	35
19.7.1 If-Then-Else	35
19.7.2 Detecting Nonthrowing Operations	35
19.7.3 Traits Convenience	35
19.8 Type Classification	35
19.8.1 Determining Fundamental Types	35
19.8.2 Determining Compound Types	35
19.8.3 Identifying Function Types	35
19.8.4 Determining Class Types	35
19.8.5 Determining Enumeration Types	35
19.9 Policy Traits	35

19.9.1 Read-Only Parameter Types	35
19.10 In the Standard Library	35
19.11 Afternotes	35
20 Overloading on Type Properties	36
20.1 Algorithm Specialization	36
20.2 Tag Dispatching	36
20.3 Enabling/Disabling Function Templates	36
20.3.1 Providing Multiple Specializations	36
20.3.2 Where Does the EnableIf Go?	36
20.3.3 Compile-Time if	36
20.3.4 Concepts	36
20.4 Class Specialization	36
20.4.1 Enabling/Disabling Class Templates	36
20.4.2 Tag Dispatching for Class Templates	36
20.5 Instantiation-Safe Templates	36
20.6 In the Standard Library	36
20.7 Afternotes	36
21 Templates and Inheritance	37
21.1 The Empty Base Class Optimization (EBCO)	37
21.1.1 Layout Principles	37
21.1.2 Members as Base Classes	37
21.2 The Curiously Recurring Template Pattern (CRTP)	37
21.2.1 The Barton-Nackman Trick	37
21.2.2 Operator Implementations	37
21.2.3 Facades	37
21.3 Mixins	37
21.3.1 Curious Mixins	37
21.3.2 Parameterized Virtuality	37
21.4 Named Template Arguments	37
21.5 Afternotes	37
22 Bridging Static and Dynamic Polymorphism	38
22.1 Function Objects, Pointers, and std::function<>	38
22.2 Generalized Function Pointers	38
22.3 Bridge Interface	38
22.4 Type Erasure	38
22.5 Optional Bridging	38

22.6 Performance Considerations	38
22.7 Afternotes	38
23 Metaprogramming	39
23.1 The State of Modern C++ Metaprogramming	39
23.1.1 Value Metaprogramming	39
23.1.2 Type Metaprogramming	39
23.1.3 Hybrid Metaprogramming	39
23.1.4 Hybrid Metaprogramming for Unit Types	39
23.2 The Dimensions of Reflective Metaprogramming	39
23.3 The Cost of Recursive Instantiation	39
23.3.1 Tracking All Instantiations	39
23.4 Computational Completeness	39
23.5 Recursive Instantiation versus Recursive Template Arguments	39
23.6 Enumeration Values versus Static Constants	39
23.7 Afternotes	39
24 Typelists	40
24.1 Anatomy of a Typelist	40
24.2 Typelist Algorithms	40
24.2.1 Indexing	40
24.2.2 Finding the Best Match	40
24.2.3 Appending to a Typelist	40
24.2.4 Reversing a Typelist	40
24.2.5 Transforming a Typelist	40
24.2.6 Accumulating Typelists	40
24.2.7 Insertion Sort	40
24.3 Nontype Typelists	40
24.3.1 Deducible Nontype Parameters	40
24.4 Optimizing Algorithms with Pack Expansions	40
24.5 Cons-style Typelists	40
24.6 Afternotes	40
25 Tuples	41
25.1 Basic Tuple Design	41
25.1.1 Storage	41
25.1.2 Construction	41
25.2 Basic Tuple Operations	41

25.2.1 Comparison	41
25.2.2 Output	41
25.3 Tuple Algorithms	41
25.3.1 Tuples as Typelists	41
25.3.2 Adding to and Removing from a Tuple	41
25.3.3 Reversing a Tuple	41
25.3.4 Index Lists	41
25.3.5 Reversal with Index Lists	41
25.3.6 Shuffle and Select	41
25.4 Expanding Tuples	41
25.5 Optimizing Tuple	41
25.5.1 Tuples and the EBCO	41
25.5.2 Constant-time get()	41
25.6 Tuple Subscript	41
25.7 Afternotes	41
26 Discriminated Unions	42
26.1 Storage	42
26.2 Design	42
26.3 Value Query and Extraction	42
26.4 Element Initialization, Assignment and Destruction	42
26.4.1 Initialization	42
26.4.2 Destruction	42
26.4.3 Assignment	42
26.5 Visitors	42
26.5.1 Visit Result Type	42
26.5.2 Common Result Type	42
26.6 Variant Initialization and Assignment	42
26.7 Afternotes	42
27 Expression Templates	43
27.1 Temporaries and Split Loops	43
27.2 Encoding Expressions in Template Arguments	43
27.2.1 Operands of the Expression Templates	43
27.2.2 The Array Type	43
27.2.3 The Operators	43
27.2.4 Review	43
27.2.5 Expression Templates Assignments	43

27.3 Performance and Limitations of Expression Templates	43
27.4 Afternotes	43
28 Debugging Templates	44
28.1 Shallow Instantiation	44
28.2 Static Assertions	44
28.3 Archetypes	44
28.4 Tracers	44
28.5 Oracles	44
28.6 Afternotes	44
Appendixes	45
A The One-Definition Rule	45
A.1 Translation Units	45
A.2 Declarations and Definitions	45
A.3 The One-Definition Rule in Detail	45
A.3.1 One-per-Program Constraints	45
A.3.2 One-per-Translation Unit Constraints	45
A.3.3 Cross-Translation Unit Equivalence Constraints	45
B Value Categories	46
B.1 Traditional Lvalues and Rvalues	46
B.1.1 Lvalue-to-Rvalue Conversions	46
B.2 Value Categories Since C++11	46
B.2.1 Temporary Materialization	46
B.3 Checking Value Categories with decltype	46
B.4 Reference Types	46
C Overload Resolution	47
C.1 When Does Overload Resolution Kick In?	47
C.2 Simplified Overload Resolution	47
C.2.1 The Implied Argument for Member Functions	47
C.2.2 Refining the Perfect Match	47
C.3 Overloading Details	47
C.3.1 Prefer Nontemplates or More Specialized Templates	47
C.3.2 Conversion Sequences	47
C.3.3 Pointer Conversions	47
C.3.4 Initializer Lists	47
C.3.5 Functors and Surrogate Functions	47

C.3.6 Other Overloading Contexts	47
D Standard Type Utilities	48
D.1 Using Type Traits	48
D.1.1 std::integral_constant and std::bool_constant	48
D.1.2 Things You Should Know When Using Traits	48
D.2 Primary and Composite Type Categories	48
D.2.1 Testing for the Primary Type Category	48
D.2.2 Test for Composite Type Categories	48
D.3 Type Properties and Operations	48
D.3.1 Other Type Properties	48
D.3.2 Test for Specific Operations	48
D.3.3 Relationships Between Types	48
D.4 Type Construction	48
D.5 Other Traits	48
D.6 Combining Type Traits	48
D.7 Other Utilities	48
E Concepts	49
E.1 Using Concepts	49
E.2 Defining Concepts	49
E.3 Overloading on Constraints	49
E.3.1 Constraint Subsumption	49
E.3.2 Constraints and Tag Dispatching	49
E.4 Concept Tips	49
E.4.1 Testing Concepts	49
E.4.2 Concept Granularity	49
E.4.3 Binary Compatibility	49
Bibliography	58
Forums	58
Books and Web Sites	58
Glossary	59
Index	60