
**Information technology — Programming
languages — Technical Report on C++
Library Extensions**

*Technologies de l'information — Langages de programmation —
Rapport technique sur l'extensions des librairies C++*



Reference number
ISO/IEC TR 19768:2007(E)

AVA Documents
Document Sharing Hub
© ISO/IEC 2007

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2007

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Foreword	x
List of Tables	xi
1 Scope	1
1.1 Relation to C++ Standard Library Introduction	1
1.2 Categories of extensions	1
1.3 Namespaces and headers	1
2 General Utilities	3
2.1 Reference wrappers	3
2.1.1 Additions to header <functional> synopsis	3
2.1.2 Class template reference_wrapper	3
2.1.2.1 reference_wrapper construct/copy/destroy	4
2.1.2.2 reference_wrapper assignment	5
2.1.2.3 reference_wrapper access	5
2.1.2.4 reference_wrapper invocation	5
2.1.2.5 reference_wrapper helper functions	5
2.2 Smart pointers	6
2.2.1 Additions to header <memory> synopsis	6
2.2.2 Class bad_weak_ptr	7
2.2.3 Class template shared_ptr	7
2.2.3.1 shared_ptr constructors	9
2.2.3.2 shared_ptr destructor	10
2.2.3.3 shared_ptr assignment	10
2.2.3.4 shared_ptr modifiers	11
2.2.3.5 shared_ptr observers	11
2.2.3.6 shared_ptr comparison	12
2.2.3.7 shared_ptr I/O	13
2.2.3.8 shared_ptr specialized algorithms	13
2.2.3.9 shared_ptr casts	13
2.2.3.10 get_deleter	14
2.2.4 Class template weak_ptr	14
2.2.4.1 weak_ptr constructors	15

2.2.4.2	weak_ptr destructor	15
2.2.4.3	weak_ptr assignment	16
2.2.4.4	weak_ptr modifiers	16
2.2.4.5	weak_ptr observers	16
2.2.4.6	weak_ptr comparison	17
2.2.4.7	weak_ptr specialized algorithms	17
2.2.5	Class template enable_shared_from_this	17
3	Function objects	21
3.1	Definitions	21
3.2	Additions to <functional> synopsis	21
3.3	Requirements	22
3.4	Function return types	23
3.5	Function template mem_fn	24
3.6	Function object binders	24
3.6.1	Class template is_bind_expression	24
3.6.2	Class template is_placeholder	25
3.6.3	Function template bind	25
3.6.4	Placeholders	26
3.7	Polymorphic function wrappers	26
3.7.1	Class bad_function_call	26
3.7.1.1	bad_function_call constructor	27
3.7.2	Class template function	27
3.7.2.1	function construct/copy/destroy	28
3.7.2.2	function modifiers	30
3.7.2.3	function capacity	30
3.7.2.4	function invocation	30
3.7.2.5	function target access	30
3.7.2.6	undefined operators	31
3.7.2.7	null pointer comparison operators	31
3.7.2.8	specialized algorithms	31
4	Metaprogramming and type traits	33
4.1	Requirements	33
4.2	Header <type_traits> synopsis	33
4.3	Helper classes	35
4.4	General Requirements	35
4.5	Unary Type Traits	36
4.5.1	Primary Type Categories	36
4.5.2	Composite type traits	37
4.5.3	Type properties	37
4.6	Relationships between types	39
4.7	Transformations between types	40
4.7.1	Const-volatile modifications	40
4.7.2	Reference modifications	41
4.7.3	Array modifications	41

4.7.4	Pointer modifications	42
4.8	Other transformations	42
4.9	Implementation requirements	42
5	Numerical facilities	45
5.1	Random number generation	45
5.1.1	Requirements	45
5.1.2	Header <random> synopsis	49
5.1.3	Class template variate_generator	51
5.1.4	Random number engine class templates	52
5.1.4.1	Class template linear_congruential	53
5.1.4.2	Class template mersenne_twister	54
5.1.4.3	Class template subtract_with_carry	55
5.1.4.4	Class template subtract_with_carry_01	57
5.1.4.5	Class template discard_block	58
5.1.4.6	Class template xor_combine	59
5.1.5	Engines with predefined parameters	61
5.1.6	Class random_device	62
5.1.7	Random distribution class templates	63
5.1.7.1	Class template uniform_int	63
5.1.7.2	Class bernoulli_distribution	64
5.1.7.3	Class template geometric_distribution	65
5.1.7.4	Class template poisson_distribution	65
5.1.7.5	Class template binomial_distribution	66
5.1.7.6	Class template uniform_real	67
5.1.7.7	Class template exponential_distribution	67
5.1.7.8	Class template normal_distribution	68
5.1.7.9	Class template gamma_distribution	69
5.2	Mathematical special functions	70
5.2.1	Additions to header <cmath> synopsis	70
5.2.1.1	associated Laguerre polynomials	73
5.2.1.2	associated Legendre functions	73
5.2.1.3	beta function	74
5.2.1.4	(complete) elliptic integral of the first kind	74
5.2.1.5	(complete) elliptic integral of the second kind	74
5.2.1.6	(complete) elliptic integral of the third kind	75
5.2.1.7	confluent hypergeometric functions	75
5.2.1.8	regular modified cylindrical Bessel functions	75
5.2.1.9	cylindrical Bessel functions (of the first kind)	75
5.2.1.10	irregular modified cylindrical Bessel functions	76
5.2.1.11	cylindrical Neumann functions	76
5.2.1.12	(incomplete) elliptic integral of the first kind	77
5.2.1.13	(incomplete) elliptic integral of the second kind	77
5.2.1.14	(incomplete) elliptic integral of the third kind	77
5.2.1.15	exponential integral	77
5.2.1.16	Hermite polynomials	78

CONTENTS

CONTENTS vi

5.2.1.17	hypergeometric functions	78
5.2.1.18	Laguerre polynomials	78
5.2.1.19	Legendre polynomials	79
5.2.1.20	Riemann zeta function	79
5.2.1.21	spherical Bessel functions (of the first kind)	79
5.2.1.22	spherical associated Legendre functions	80
5.2.1.23	spherical Neumann functions	80
5.2.2	Additions to header <math.h> synopsis	80
6	Containers	81
6.1	Tuple types	81
6.1.1	Header <tuple> synopsis	81
6.1.2	Additions to header <utility> synopsis	82
6.1.3	Class template tuple	83
6.1.3.1	Construction	83
6.1.3.2	Tuple creation functions	84
6.1.3.3	Tuple helper classes	85
6.1.3.4	Element access	86
6.1.3.5	Relational operators	86
6.1.4	Pairs	87
6.2	Fixed size array	88
6.2.1	Header <array> synopsis	88
6.2.2	Class template array	88
6.2.2.1	array constructors, copy, and assignment	90
6.2.2.2	array specialized algorithms	90
6.2.2.3	array size	90
6.2.2.4	Zero sized arrays	90
6.2.2.5	Tuple interface to class template array	90
6.3	Unordered associative containers	91
6.3.1	Unordered associative container requirements	91
6.3.1.1	Exception safety guarantees	96
6.3.2	Additions to header <functional> synopsis	97
6.3.3	Class template hash	97
6.3.4	Unordered associative container classes	98
6.3.4.1	Header <unordered_set> synopsis	98
6.3.4.2	Header <unordered_map> synopsis	98
6.3.4.3	Class template unordered_set	99
6.3.4.3.1	unordered_set constructors	101
6.3.4.3.2	unordered_set swap	102
6.3.4.4	Class template unordered_map	102
6.3.4.4.1	unordered_map constructors	104
6.3.4.4.2	unordered_map element access	105
6.3.4.4.3	unordered_map swap	105
6.3.4.5	Class template unordered_multiset	105
6.3.4.5.1	unordered_multiset constructors	107
6.3.4.5.2	unordered_multiset swap	108

6.3.4.6	Class template <code>unordered_multimap</code>	108
6.3.4.6.1	<code>unordered_multimap</code> constructors	110
6.3.4.6.2	<code>unordered_multimap</code> swap	111
7	Regular expressions	113
7.1	Definitions	113
7.2	Requirements	113
7.3	Regular expressions summary	115
7.4	Header <code><regex></code> synopsis	115
7.5	Namespace <code>tr1::regex_constants</code>	121
7.5.1	Bitmask Type <code>syntax_option_type</code>	121
7.5.2	Bitmask Type <code>regex_constants::match_flag_type</code>	122
7.5.3	Implementation defined <code>error_type</code>	124
7.6	Class <code>regex_error</code>	125
7.7	Class template <code>regex_traits</code>	125
7.8	Class template <code>basic_regex</code>	127
7.8.1	<code>basic_regex</code> constants	129
7.8.2	<code>basic_regex</code> constructors	129
7.8.3	<code>basic_regex</code> assign	131
7.8.4	<code>basic_regex</code> constant operations	131
7.8.5	<code>basic_regex</code> locale	132
7.8.6	<code>basic_regex</code> swap	132
7.8.7	<code>basic_regex</code> non-member functions	132
7.8.7.1	<code>basic_regex</code> non-member swap	132
7.9	Class template <code>sub_match</code>	132
7.9.1	<code>sub_match</code> members	133
7.9.2	<code>sub_match</code> non-member operators	133
7.10	Class template <code>match_results</code>	138
7.10.1	<code>match_results</code> constructors	139
7.10.2	<code>match_results</code> size	140
7.10.3	<code>match_results</code> element access	140
7.10.4	<code>match_results</code> formatting	141
7.10.5	<code>match_results</code> allocator	141
7.10.6	<code>match_results</code> swap	142
7.11	Regular expression algorithms	142
7.11.1	exceptions	142
7.11.2	<code>regex_match</code>	142
7.11.3	<code>regex_search</code>	144
7.11.4	<code>regex_replace</code>	145
7.12	Regular expression Iterators	146
7.12.1	Class template <code>regex_iterator</code>	146
7.12.1.1	<code>regex_iterator</code> constructors	147
7.12.1.2	<code>regex_iterator</code> comparisons	147
7.12.1.3	<code>regex_iterator</code> dereference	148
7.12.1.4	<code>regex_iterator</code> increment	148
7.12.2	Class template <code>regex_token_iterator</code>	149

7.12.2.1	regex_token_iterator constructors	150
7.12.2.2	regex_token_iterator comparisons	151
7.12.2.3	regex_token_iterator dereference	151
7.12.2.4	regex_token_iterator increment	151
7.13	Modified ECMAScript regular expression grammar	152
8	C compatibility	155
8.1	Additions to header <complex>	155
8.1.1	Synopsis	155
8.1.2	Function acos	155
8.1.3	Function asin	155
8.1.4	Function atan	156
8.1.5	Function acosh	156
8.1.6	Function asinh	156
8.1.7	Function atanh	156
8.1.8	Function fabs	156
8.1.9	Additional Overloads	156
8.2	Header <ccomplex>	157
8.3	Header <complex.h>	157
8.4	Additions to header <cctype>	157
8.4.1	Synopsis	157
8.4.2	Function isblank	157
8.5	Additions to header <ctype.h>	157
8.6	Header <cfenv>	157
8.6.1	Synopsis	157
8.6.2	Definitions	158
8.7	Header <fenv.h>	158
8.8	Additions to header <cfloat>	158
8.9	Additions to header <float.h>	158
8.10	Additions to header <ios>	158
8.10.1	Synopsis	158
8.10.2	Function hexfloat	158
8.11	Header < cinttypes>	159
8.11.1	Synopsis	159
8.11.2	Definitions	159
8.12	Header < inttypes.h>	159
8.13	Additions to header <climits>	160
8.14	Additions to header <limits.h>	160
8.15	Additions to header <locale>	160
8.16	Additions to header <cmath>	160
8.16.1	Synopsis	160
8.16.2	Definitions	164
8.16.3	Function template definitions	164
8.16.4	Additional overloads	165
8.17	Additions to header <math.h>	166
8.18	Additions to header <cstdarg>	166

ix CONTENTS

CONTENTS

8.19	Additions to header <stdarg.h>	166
8.20	The header <stdbool>	166
8.21	The header <stdbool.h>	166
8.22	The header <stdint>	166
8.22.1	Synopsis	166
8.22.2	Definitions	167
8.23	The header <stdint.h>	167
8.24	Additions to header <stdio>	167
8.24.1	Synopsis	167
8.24.2	Definitions	168
8.24.3	Additional format specifiers	168
8.24.4	Additions to header <stdio.h>	168
8.25	Additions to header <stdlib>	168
8.25.1	Synopsis	168
8.25.2	Definitions	169
8.25.3	Function abs	169
8.25.4	Function div	169
8.26	Additions to header <stdlib.h>	169
8.27	Header <tgmath>	169
8.28	Header <tgmath.h>	169
8.29	Additions to header <ctime>	169
8.30	Additions to header <wchar>	169
8.30.1	Synopsis	169
8.30.2	Definitions	170
8.30.3	Additional wide format specifiers	170
8.31	Additions to header <wchar.h>	170
8.32	Additions to header <wctype>	170
8.32.1	Synopsis	170
8.32.2	Function iswblank	170
8.33	Additions to header <wctype.h>	170
A	Implementation quantities	171
	Bibliography	173
	Index	175

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, the joint technical committee may propose the publication of a Technical Report of one of the following types:

- type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;
- type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;
- type 3, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 19768, which is a Technical Report of type 2, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

List of Tables

1	Utilities library summary	3
2	Function object library summary	21
3	Type traits library summary	33
4	Primary Type Category Predicates	36
5	Composite Type Category Predicates	37
6	Type Property Predicates	37
7	Type Property Queries	38
8	Type Relationship Predicates	39
9	Const-volatile modifications	40
10	Reference modifications	41
11	Array modifications	41
12	Pointer modifications	42
13	Other transformations	42
14	Numerical library summary	45
15	Uniform random number generator requirements	45
16	Pseudo-random number engine requirements (in addition to uniform random number generator, Copy-Constructible, and Assignable)	46
17	Random distribution requirements (in addition to CopyConstructible, and Assignable)	48
18	Additions to header <cmath> synopsis	72
19	Container library summary	81
20	Container requirements that are not required for unordered associative containers	91
21	Unordered associative container requirements (in addition to container)	92
22	regular expression traits class requirements	114
23	syntax_option_type effects	122
24	regex_constants::match_flag_type effects when obtaining a match against a character container sequence [first,last).	123
25	error_type values in the C locale	124
26	match_results assignment operator effects	140
27	Effects of regex_match algorithm	142
28	Effects of regex_search algorithm	144

1 Scope

[tr.intro]

- 1 This Technical Report describes extensions to the *C++ standard library* that is described in the International Standard for the C++ programming language [14].
- 2 This Technical Report is non-normative. Some of the library components in this Technical Report may be considered for standardization in a future version of C++, but they are not currently part of any C++ standard. Some of the components in this Technical Report may never be standardized, and others may be standardized in a substantially changed form.
- 3 The goal of this Technical Report is to build more widespread existing practice for an expanded C++ standard library. It gives advice on extensions to those vendors who wish to provide them.

1.1 Relation to C++ Standard Library Introduction

[tr.description]

- 1 Unless otherwise specified, the whole of the ISO C++ Standard Library introduction [lib.library] is included in this Technical Report by reference.

1.2 Categories of extensions [tr.intro.ext]

- 1 This Technical Report describes four general categories of library extensions:
 1. New requirement tables, such as the regular expression traits requirements in 7.2. These are not directly expressed as software; they specify the circumstances under which user-written components will interoperate with the components described in this Technical Report.
 2. New library components (types and functions) that are declared in entirely new headers, such as the class templates in the `<unordered_set>` header (6.3.4.1).
 3. New library components declared as additions to existing standard headers, such as the mathematical special functions added to the headers `<cmath>` and `<math.h>` in 5.2.1 and 5.2.2
 4. Additions to standard library components, such as the extensions to class `std::pair` in 6.1.4.
- 2 New headers are distinguished from extensions to existing headers by the title of the *synopsis* clause. In the first case, the title is of the form “Header `<foo>` synopsis”, and the synopsis includes all namespace scope declarations contained in the header. In the second case, the title is of the form “Additions to header `<foo>` synopsis” and the synopsis includes only the extensions, *i.e.* those namespace scope declarations that are not present in the C++ standard [14].

1.3 Namespaces and headers**[tr.intro.namespaces]**

- 1 Since the extensions described in this Technical Report are not part of the C++ standard library, they should not be declared directly within namespace `std`. Unless otherwise specified, all components described in this Technical Report are declared in namespace `std::tr1`. [*Note*: Some components are declared in subnamespaces of namespace `std::tr1`. —*end note*]
- 2 Unless otherwise specified, references to other entities described in this Technical Report are assumed to be qualified with `std::tr1::`, and references to entities described in the International Standard are assumed to be qualified with `std::`.
- 3 Even when an extension is specified as additions to standard headers (the third category in 1.2), vendors should not simply add declarations to standard headers in a way that would be visible to users by default. [*Note*: That would fail to be standard conforming, because the new names, even within a namespace, could conflict with user macros. —*end note*] Users should be required to take explicit action to have access to library extensions.
- 4 It is recommended either that additional declarations in standard headers be protected with a macro that is not defined by default, or else that all extended headers, including both new headers and parallel versions of standard headers with non-standard declarations, be placed in a separate directory that is not part of the default search path.

2 General Utilities

[tr.util]

- 1 This clause describes basic components used to implement other library facilities. They may also be used by C++ programs.
- 2 The following subclauses describe reference wrappers and smart pointers, as summarized in Table 1.

Table 1: Utilities library summary

Subclause	Header(s)
2.1 Reference wrapper	<functional>
2.2 Smart pointers	<memory>

2.1 Reference wrappers

[tr.util.refwrap]

2.1.1 Additions to header <functional> synopsis

[tr.util.refwrap.synopsis]

```

namespace std {
namespace tr1 {
    template <class T> class reference_wrapper;

    template <class T> reference_wrapper<T> ref(T&);
    template <class T> reference_wrapper<const T> cref(const T&);

    template <class T> reference_wrapper<T> ref(reference_wrapper<T>);
    template <class T> reference_wrapper<const T> cref(reference_wrapper<T>);
} // namespace tr1
} // namespace std

```

2.1.2 Class template reference_wrapper

[tr.util.refwrap.refwrap]

```

template <class T> class reference_wrapper
: public unary_function<T1, R>           // see below
: public binary_function<T1, T2, R>      // see below
{
public :
    // types
    typedef T type;
    typedef see below result_type; // Not always defined

```

```

// construct/copy/destroy
explicit reference_wrapper(T&);
reference_wrapper(const reference_wrapper<T>& x);

// assignment
reference_wrapper& operator=(const reference_wrapper<T>& x);

// access
operator T& () const;
T& get() const;

// invocation
template <class T1, class T2, ..., class TN>
typename result_of<T(T1, T2, ..., TN)>::type
operator() (T1&, T2&, ..., TN&) const;
};

```

- 1 `reference_wrapper<T>` is a CopyConstructible and Assignable wrapper around a reference to an object of type `T`.
- 2 `reference_wrapper` has a weak result type ([3.3]).
- 3 The template instantiation `reference_wrapper<T>` shall be derived from `std::unary_function<T1, R>` only if the type `T` is any of the following:
 - a function type or a pointer to function type taking one argument of type `T1` and returning `R`
 - a pointer to member function type with cv-qualifier `cv` and no arguments; the type `T1` is `cv T*` and `R` is the return type of the pointer to member function
 - a class type that is derived from `std::unary_function<T1, R>`
- 4 The template instantiation `reference_wrapper<T>` shall be derived from `std::binary_function<T1, T2, R>` only if the type `T` is any of the following:
 - a function type or a pointer to function type taking two arguments of types `T1` and `T2` and returning `R`
 - a pointer to member function with cv-qualifier `cv` and taking one argument of type `T2`; the type `T1` is `cv T*` and `R` is the return type of the pointer to member function
 - a class type that is derived from `std::binary_function<T1, T2, R>`

2.1.2.1 `reference_wrapper` construct/copy/destroy

[tr.util.refwrp.const]

```
explicit reference_wrapper(T& t);
```

- 1 *Effects:* Constructs a `reference_wrapper` object that stores a reference to `t`.
- 2 *Throws:* nothing.

```
reference_wrapper(const reference_wrapper<T>& x);
```

- 3 *Effects:* Constructs a `reference_wrapper` object that stores a reference to `x.get()`.

5 General Utilities

2.1 Reference wrappers

4 *Throws:* nothing.

2.1.2.2 reference_wrapper assignment

[tr.util.refwrp.assign]

```
reference_wrapper& operator=(const reference_wrapper<T>& x);
```

1 *Postconditions:* *this stores a reference to x.get().

2 *Throws:* Nothing.

2.1.2.3 reference_wrapper access

[tr.util.refwrp.access]

```
operator T& () const;
```

1 *Returns:* The stored reference.

2 *Throws:* nothing.

```
T& get() const;
```

3 *Returns:* The stored reference.

4 *Throws:* nothing.

2.1.2.4 reference_wrapper invocation

[tr.util.refwrp.invoke]

```
template <class T1, class T2, ..., class TN>
    typename result_of<T(T1, T2, ..., TN)>::type operator()(T1& a1, T2& a1, ..., TN& aN) const;
```

1 *Returns:* *INVOKE*(get(), a1, a2, ..., aN). ([3.3])

2 *Note:* operator() is described for exposition only. Implementations are not required to provide an actual reference_wrapper::operator(). Implementations are permitted to support reference_wrapper function invocation through multiple overloaded operators or through other means.

2.1.2.5 reference_wrapper helper functions

[tr.util.refwrp.helpers]

```
template <class T> reference_wrapper<T> ref(T& t);
```

1 *Returns:* reference_wrapper<T>(t)

2 *Throws:* nothing.

```
template <class T> reference_wrapper<T> ref(reference_wrapper<T>t);
```

3 *Returns:* ref(t.get())

4 *Throws:* nothing.

```
template <class T> reference_wrapper<const T> cref(const T& t);
```

5 *Returns:* reference_wrapper <const T>(t)

6 *Throws:* nothing.

```
template <class T> reference_wrapper<const T> cref(reference_wrapper<T> t);
```

7 *Returns:* cref(t.get());

8 *Throws:* nothing.

2.2 Smart pointers

[tr.util.smartptr]

2.2.1 Additions to header <memory> synopsis

[tr.util.smartptr.synopsis]

```
namespace std {
namespace tr1 {
    // [2.2.2] Class bad_weak_ptr
    class bad_weak_ptr;

    // [2.2.3] Class template shared_ptr
    template<class T> class shared_ptr;

    // [2.2.3.6] shared_ptr comparisons
    template<class T, class U> bool operator==(shared_ptr<T> const& a, shared_ptr<U> const& b);
    template<class T, class U> bool operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b);
    template<class T, class U> bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b);

    // [2.2.3.8] shared_ptr specialized algorithms
    template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

    // [2.2.3.9] shared_ptr casts
    template<class T, class U> shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);
    template<class T, class U> shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);
    template<class T, class U> shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r);

    // [2.2.3.7] shared_ptr I/O
    template<class E, class T, class Y>
        basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);

    // [2.2.3.10] shared_ptr get_deleter
    template<class D, class T> D* get_deleter(shared_ptr<T> const& p);

    // [2.2.4] Class template weak_ptr
    template<class T> class weak_ptr;

    // [2.2.4.6] weak_ptr comparison
    template<class T, class U> bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);

    // [2.2.4.7] weak_ptr specialized algorithms
    template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b);
```

7 General Utilities

2.2 Smart pointers

```
// [2.2.5] Class enable_shared_from_this
template<class T> class enable_shared_from_this;
} // namespace tr1
} // namespace std
```

2.2.2 Class bad_weak_ptr

[tr.util.smartptr.weakptr]

```
namespace std {
namespace tr1 {
    class bad_weak_ptr: public std::exception
    {
    public:
        bad_weak_ptr();
    };
} // namespace tr1
} // namespace std
```

- 1 An exception of type bad_weak_ptr is thrown by the shared_ptr constructor taking a weak_ptr.

```
bad_weak_ptr();
```

- 2 *Postconditions:* what() returns "tr1::bad_weak_ptr".
- 3 *Throws:* nothing.

2.2.3 Class template shared_ptr

[tr.util.smartptr.shared]

- 1 The shared_ptr class template stores a pointer, usually obtained via new. shared_ptr implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer.

```
namespace std {
namespace tr1 {
    template<class T> class shared_ptr {
    public:
        typedef T element_type;

        // [2.2.3.1] constructors
        shared_ptr();
        template<class Y> explicit shared_ptr(Y* p);
        template<class Y, class D> shared_ptr(Y* p, D d);
        shared_ptr(shared_ptr const& r);
        template<class Y> shared_ptr(shared_ptr<Y> const& r);
        template<class Y> explicit shared_ptr(weak_ptr<Y> const& r);
        template<class Y> explicit shared_ptr(auto_ptr<Y>& r);

        // [2.2.3.2] destructor
        ~shared_ptr();

        // [2.2.3.3] assignment
```

```

shared_ptr& operator=(shared_ptr const& r);
template<class Y> shared_ptr& operator=(shared_ptr<Y> const& r);
template<class Y> shared_ptr& operator=(auto_ptr<Y>& r);

// [2.2.3.4] modifiers
void swap(shared_ptr& r);
void reset();
template<class Y> void reset(Y* p);
template<class Y, class D> void reset(Y* p, D d);

// [2.2.3.5] observers
T* get() const;
T& operator*() const;
T* operator->() const;
long use_count() const;
bool unique() const;
operator unspecified-bool-type() const;
};

// [2.2.3.6] shared_ptr comparisons
template<class T, class U> bool operator==(shared_ptr<T> const& a, shared_ptr<U> const& b);
template<class T, class U> bool operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b);
template<class T, class U> bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b);

// [2.2.3.7] shared_ptr I/O
template<class E, class T, class Y>
    basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);

// [2.2.3.8] shared_ptr specialized algorithms
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

// [2.2.3.9] shared_ptr casts
template<class T, class U> shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);
template<class T, class U> shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);
template<class T, class U> shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r);

// [2.2.3.10] shared_ptr get_deleter
template<class D, class T> D* get_deleter(shared_ptr<T> const& p);
} // namespace tr1
} // namespace std

```

- 2 Specializations of `shared_ptr` shall be CopyConstructible, Assignable, and LessThanComparable, allowing their use in standard containers. Specializations of `shared_ptr` shall be convertible to `bool`, allowing their use in boolean expressions and declarations in conditions. The template parameter `T` of `shared_ptr` may be an incomplete type.

3 [Example:

```

if(shared_ptr<X> px = dynamic_pointer_cast<X>(py))
{
    // do something with px
}

```

9 General Utilities

2.2 Smart pointers

}

—end example.]

2.2.3.1 shared_ptr constructors

[tr.util.smartptr.shared.const]

shared_ptr();

1 *Effects:* Constructs an *empty* shared_ptr object.2 *Postconditions:* use_count() == 0 && get() == 0.3 *Throws:* nothing.

template<class Y> explicit shared_ptr(Y* p);

4 *Requires:* p shall be convertible to T*. Y shall be a complete type. The expression delete p shall be well-formed, shall have well defined behavior, and shall not throw exceptions.5 *Effects:* Constructs a shared_ptr object that *owns* the pointer p.6 *Postconditions:* use_count() == 1 && get() == p.7 *Throws:* bad_alloc, or an implementation-defined exception when a resource other than memory could not be obtained.8 *Exception safety:* If an exception is thrown, delete p is called.

template<class Y, class D> shared_ptr(Y* p, D d);

9 *Requires:* p shall be convertible to T*. D shall be CopyConstructible. The copy constructor and destructor of D shall not throw exceptions. The expression d(p) shall be well-formed, shall have well defined behavior, and shall not throw exceptions.10 *Effects:* Constructs a shared_ptr object that *owns* the pointer p and the deleter d.11 *Postconditions:* use_count() == 1 && get() == p.12 *Throws:* bad_alloc, or an implementation-defined exception when a resource other than memory could not be obtained.13 *Exception safety:* If an exception is thrown, d(p) is called.

shared_ptr(shared_ptr const& r);

template<class Y> shared_ptr(shared_ptr<Y> const& r);

14 *Requires:* For the second constructor Y* shall be convertible to T*.15 *Effects:* If r is *empty*, constructs an *empty* shared_ptr object; otherwise, constructs a shared_ptr object that *shares ownership* with r.16 *Postconditions:* get() == r.get() && use_count() == r.use_count().17 *Throws:* nothing.

template<class Y> explicit shared_ptr(weak_ptr<Y> const& r);

- 18 *Requires:* Y^* shall be convertible to T^* .
- 19 *Effects:* Constructs a `shared_ptr` object that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.
- 20 *Postconditions:* `use_count() == r.use_count()`.
- 21 *Throws:* `bad_weak_ptr` when `r.expired()`.
- 22 *Exception safety:* If an exception is thrown, the constructor has no effect.
- ```
template<class Y> shared_ptr(auto_ptr<Y>& r);
```
- 23 *Requires:* `r.release()` shall be convertible to  $T^*$ .  $Y$  shall be a complete type. The expression `delete r.release()` shall be well-formed, shall have well defined behavior, and shall not throw exceptions.
- 24 *Effects:* Constructs a `shared_ptr` object that stores and *owns* `r.release()`.
- 25 *Postconditions:* `use_count() == 1 && r.get() == 0`.
- 26 *Throws:* `bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.
- 27 *Exception safety:* If an exception is thrown, the constructor has no effect.

**2.2.3.2 shared\_ptr destructor**

[tr.util.smartptr.shared.dest]

~shared\_ptr();

- 1 *Effects:*
- If `*this` is *empty*, there are no side effects.
  - If `*this` *shares ownership* with another `shared_ptr` instance (`use_count() > 1`), decrements that instance's use count.
  - Otherwise, if `*this` *owns* a pointer `p` and a deleter `d`, `d(p)` is called.
  - Otherwise, `*this` *owns* a pointer `p`, and `delete p` is called.
- 2 *Throws:* nothing.

**2.2.3.3 shared\_ptr assignment**

[tr.util.smartptr.shared.assign]

```
shared_ptr& operator=(shared_ptr const& r);
template<class Y> shared_ptr& operator=(shared_ptr<Y> const& r);
template<class Y> shared_ptr& operator=(auto_ptr<Y>& r);
```

- 1 *Effects:* Equivalent to `shared_ptr(r).swap(*this)`.
- 2 *Returns:* `*this`.

## 11 General Utilities

## 2.2 Smart pointers

- 3 [Note: The use count updates caused by the temporary object construction and destruction are not observable side effects, so the implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary. In particular, in the example:

```
shared_ptr<int> p(new int);
shared_ptr<void> q(p);
p = p;
q = p;
```

both assignments may be no-ops. —end note]

## 2.2.3.4 shared\_ptr modifiers

[tr.util.smartptr.shared.mod]

```
void swap(shared_ptr& r);
```

- 1 *Effects:* Exchanges the contents of \*this and r.

- 2 *Throws:* nothing.

```
void reset();
```

- 3 *Effects:* Equivalent to shared\_ptr().swap(\*this).

```
template<class Y> void reset(Y* p);
```

- 4 *Effects:* Equivalent to shared\_ptr(p).swap(\*this).

```
template<class Y, class D> void reset(Y* p, D d);
```

- 5 *Effects:* Equivalent to shared\_ptr(p, d).swap(\*this).

## 2.2.3.5 shared\_ptr observers

[tr.util.smartptr.shared.obs]

```
T* get() const;
```

- 1 *Returns:* the stored pointer. Returns a null pointer if \*this is empty.

- 2 *Throws:* nothing.

```
T& operator*() const;
```

- 3 *Requires:* get() != 0.

- 4 *Returns:* \*get().

- 5 *Throws:* nothing.

- 6 *Notes:* When T is void, attempting to instantiate this member function renders the program ill-formed. [Note: Instantiating shared\_ptr<void> does not necessarily result in instantiating this member function. —end note]

```
T* operator->() const;
```

7 *Requires:* `get() != 0`.

8 *Returns:* `get()`.

9 *Throws:* nothing.

`long use_count() const;`

10 *Returns:* the number of `shared_ptr` objects, `*this` included, that *share ownership* with `*this`, or 0 when `*this` is *empty*.

11 *Throws:* nothing.

12 *[Note:* `use_count()` *is not necessarily efficient. Use only for debugging and testing purposes, not for production code. —end note]*

`bool unique() const;`

13 *Returns:* `use_count() == 1`.

14 *Throws:* nothing.

15 *[Note:* `unique()` *may be faster than* `use_count()` *. If you are using* `unique()` *to implement copy on write, do not rely on a specific value when* `get() == 0` *. —end note]*

`operator unspecified-bool-type() const;`

16 *Returns:* an unspecified value that, when used in boolean contexts, is equivalent to `get() != 0`.

17 *Throws:* nothing.

18 *[Note:* This conversion operator allows `shared_ptr` objects to be used in boolean contexts. *[Example:* `if (p && p->valid())` *—end example.]* One possible choice for the return type is a pointer to member function, which avoids many of the implicit conversion pitfalls of a `bool` or `void*` return type. *—end note]*

### 2.2.3.6 `shared_ptr` comparison

[tr.util.smartptr.shared.cmp]

`template<class T, class U> bool operator==(shared_ptr<T> const& a, shared_ptr<U> const& b);`

1 *Returns:* `a.get() == b.get()`.

2 *Throws:* nothing.

`template<class T, class U> bool operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b);`

3 *Returns:* `a.get() != b.get()`.

4 *Throws:* nothing.

`template<class T, class U> bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b);`

5 *Returns:* an unspecified value such that

— `operator<` is a strict weak ordering as described in [lib.alg.sorting];

## 13 General Utilities

## 2.2 Smart pointers

— under the equivalence relation defined by `operator<`, `!(a < b) && !(b < a)`, two `shared_ptr` instances are equivalent if and only if they *share ownership* or are both empty.

6 *Throws:* nothing.

7 [Note: Defining a comparison operator allows `shared_ptr` objects to be used as keys in associative containers. —end note]

2.2.3.7 `shared_ptr` I/O

[tr.util.smartptr.shared.io]

```
template<class E, class T, class Y>
 basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);
```

1 *Effects:* `os << p.get();`.

2 *Returns:* `os`.

2.2.3.8 `shared_ptr` specialized algorithms

[tr.util.smartptr.shared.spec]

```
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);
```

1 *Effects:* Equivalent to `a.swap(b)`.

2 *Throws:* nothing.

2.2.3.9 `shared_ptr` casts

[tr.util.smartptr.shared.cast]

```
template<class T, class U> shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);
```

1 *Requires:* The expression `static_cast<T*>(r.get())` shall be well-formed.

2 *Returns:* If `r` is *empty*, an *empty* `shared_ptr<T>`; otherwise, a `shared_ptr<T>` object that stores `static_cast<T*>(r.get())` and *shares ownership* with `r`.

3 *Throws:* nothing.

4 [Note: The seemingly equivalent expression `shared_ptr<T>(static_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. —end note]

```
template<class T, class U> shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);
```

5 *Requires:* The expression `dynamic_cast<T*>(r.get())` shall be well-formed and shall have well defined behavior.

6 *Returns:*

— When `dynamic_cast<T*>(r.get())` returns a nonzero value, a `shared_ptr<T>` object that stores a copy of it and *shares ownership* with `r`;

— Otherwise, an *empty* `shared_ptr<T>` object.

- 7 *Throws:* nothing.
- 8 [Note: The seemingly equivalent expression `shared_ptr<T>(dynamic_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. —end note]
- ```
template<class T, class U> shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r);
```
- 9 *Requires:* The expression `const_cast<T*>(r.get())` shall be well-formed.
- 10 *Returns:* If `r` is empty, an empty `shared_ptr<T>`; otherwise, a `shared_ptr<T>` object that stores `const_cast<T*>(r.get())` and shares ownership with `r`.
- 11 *Throws:* nothing.
- 12 [Note: The seemingly equivalent expression `shared_ptr<T>(const_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. —end note]

2.2.3.10 get_deleter**[tr.util.smartptr.getdeleter]**

```
template<class D, class T> D* get_deleter(shared_ptr<T> const& p);
```

- 1 *Returns:* If `p` owns a deleter `d` of type cv-unqualified `D`, returns `&d`; otherwise returns 0.
- 2 *Throws:* nothing.

2.2.4 Class template weak_ptr**[tr.util.smartptr.weak]**

- 1 The `weak_ptr` class template stores a weak reference to an object that is already managed by a `shared_ptr`. To access the object, a `weak_ptr` can be converted to a `shared_ptr` using the member function `lock`.

```
namespace std {
namespace tr1 {
    template<class T> class weak_ptr
    {
    public:
        typedef T element_type;

        // constructors
        weak_ptr();
        template<class Y> weak_ptr(shared_ptr<Y> const& r);
        weak_ptr(weak_ptr const& r);
        template<class Y> weak_ptr(weak_ptr<Y> const& r);

        // destructor
        ~weak_ptr();

        // assignment
        weak_ptr& operator=(weak_ptr const& r);
        template<class Y> weak_ptr& operator=(weak_ptr<Y> const& r);
        template<class Y> weak_ptr& operator=(shared_ptr<Y> const& r);
```

15 General Utilities

2.2 Smart pointers

```

    // modifiers
    void swap(weak_ptr& r);
    void reset();

    // observers
    long use_count() const;
    bool expired() const;
    shared_ptr<T> lock() const;
};

// comparison
template<class T, class U> bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);

// specialized algorithms
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b);
} // namespace tr1
} // namespace std

```

- 2 Specializations of `weak_ptr` shall be `CopyConstructible`, `Assignable`, and `LessThanComparable`, allowing their use in standard containers. The template parameter `T` of `weak_ptr` may be an incomplete type.

2.2.4.1 `weak_ptr` constructors

[tr.util.smartptr.weak.const]

```
weak_ptr();
```

- 1 *Effects:* Constructs an *empty* `weak_ptr` object.

- 2 *Postconditions:* `use_count() == 0`.

- 3 *Throws:* nothing.

```

template<class Y> weak_ptr(shared_ptr<Y> const& r);
weak_ptr(weak_ptr const& r);
template<class Y> weak_ptr(weak_ptr<Y> const& r);

```

- 4 *Requires:* For the second and third constructors, `Y*` shall be convertible to `T*`.

- 5 *Effects:* If `r` is *empty*, constructs an *empty* `weak_ptr` object; otherwise, constructs a `weak_ptr` object that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.

- 6 *Postconditions:* `use_count() == r.use_count()`.

- 7 *Throws:* nothing.

2.2.4.2 `weak_ptr` destructor

[tr.util.smartptr.weak.dest]

```
~weak_ptr();
```

- 1 *Effects:* Destroys this `weak_ptr` object but has no effect on the object its stored pointer points to.

- 2 *Throws:* nothing.

2.2.4.3 weak_ptr assignment

[tr.util.smartptr.weak.assign]

```
weak_ptr& operator=(weak_ptr const& r);
template<class Y> weak_ptr& operator=(weak_ptr<Y> const& r);
template<class Y> weak_ptr& operator=(shared_ptr<Y> const& r);
```

1 *Effects:* Equivalent to `weak_ptr(r).swap(*this)`.

2 *Throws:* nothing.

3 *Notes:* The implementation may meet the effects (and the implied guarantees) via different means, without creating a temporary.

2.2.4.4 weak_ptr modifiers

[tr.util.smartptr.weak.mod]

```
void swap(weak_ptr& r);
```

1 *Effects:* Exchanges the contents of `*this` and `r`.

2 *Throws:* nothing.

```
void reset();
```

3 *Effects:* Equivalent to `weak_ptr().swap(*this)`.

2.2.4.5 weak_ptr observers

[tr.util.smartptr.weak.obs]

```
long use_count() const;
```

1 *Returns:* 0 if `*this` is *empty*; otherwise, the number of `shared_ptr` instances that *share ownership* with `*this`.

2 *Throws:* nothing.

3 *[Note:* `use_count()` is not necessarily efficient. Use only for debugging and testing purposes, not for production code. *—end note]*

```
bool expired() const;
```

4 *Returns:* `use_count() == 0`.

5 *Throws:* nothing.

6 *[Note:* `expired()` may be faster than `use_count()`. *—end note]*

```
shared_ptr<T> lock() const;
```

7 *Returns:* `expired() ? shared_ptr<T>() : shared_ptr<T>(*this)`.

8 *Throws:* nothing.

17 General Utilities

2.2 Smart pointers

2.2.4.6 weak_ptr comparison

[tr.util.smartptr.weak.cmp]

```
template<class T, class U> bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);
```

1 *Returns:* an unspecified value such that

- operator< is a strict weak ordering as described in [lib.alg.sorting];
- under the equivalence relation defined by operator<, !(a < b) && !(b < a), two weak_ptr instances are equivalent if and only if they *share ownership* or are both empty.

2 *Throws:* nothing.

3 [Note: Allows weak_ptr objects to be used as keys in associative containers. —end note]

2.2.4.7 weak_ptr specialized algorithms

[tr.util.smartptr.weak.spec]

```
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b)
```

1 *Effects:* Equivalent to a.swap(b).

2 *Throws:* nothing.

2.2.5 Class template enable_shared_from_this

[tr.util.smartptr.enab]

1 A class T can inherit from enable_shared_from_this<T> to inherit the shared_from_this member functions that obtain a *shared_ptr* instance pointing to *this.

2 [Example:

```
struct X: public enable_shared_from_this<X>
{
};

int main()
{
    shared_ptr<X> p(new X);
    shared_ptr<X> q = p->shared_from_this();
    assert(p == q);
    assert(!(p < q) && !(q < p)); //p and q share ownership
}
```

—end example.]

```
namespace std {
namespace tr1 {
    template<class T> class enable_shared_from_this {
    protected:
        enable_shared_from_this();
        enable_shared_from_this(enable_shared_from_this const&);
        enable_shared_from_this& operator=(enable_shared_from_this const&);
        ~enable_shared_from_this();
    };
}
```

```

public:
    shared_ptr<T> shared_from_this();
    shared_ptr<T const> shared_from_this() const;
};
} // namespace tr1
} // namespace std

```

- 3 The template parameter T of `enable_shared_from_this` may be an incomplete type.

```

enable_shared_from_this();
enable_shared_from_this(enable_shared_from_this<T> const&);

```

- 4 *Effects:* Constructs an `enable_shared_from_this<T>` object.

- 5 *Throws:* nothing.

```

enable_shared_from_this<T>& operator=(enable_shared_from_this<T> const&);

```

- 6 *Returns:* `*this`.

- 7 *Throws:* nothing.

```

~enable_shared_from_this();

```

- 8 *Effects:* Destroys `*this`.

- 9 *Throws:* nothing.

```

shared_ptr<T> shared_from_this();
shared_ptr<T const> shared_from_this() const;

```

- 10 *Requires:* `enable_shared_from_this<T>` shall be an accessible base class of T. `*this` shall be a subobject of an object t of type T. There shall be at least one `shared_ptr` instance p that *owns* &t.

- 11 *Returns:* A `shared_ptr<T>` object r that *shares ownership* with p.

- 12 *Postconditions:* `r.get() == this`.

- 13 [Note: a possible implementation is shown below:

```

template<class T> class enable_shared_from_this
{
private:
    weak_ptr<T> __weak_this;

protected:
    enable_shared_from_this() {}
    enable_shared_from_this(enable_shared_from_this const &) {}
    enable_shared_from_this& operator=(enable_shared_from_this const &) { return *this; }
    ~enable_shared_from_this() {}

public:
    shared_ptr<T> shared_from_this() { return shared_ptr<T>(__weak_this); }
    shared_ptr<T const> shared_from_this() const { return shared_ptr<T const>(__weak_this); }
}

```

};

- 14 The `shared_ptr` constructors that create unique pointers can detect the presence of an `enable_shared_from_this` base and assign the newly created `shared_ptr` to its `__weak_this` member. —*end note*]

(blank page)

3 Function objects

[tr.func]

- 1 This clause defines components for creating and manipulating function objects, and for higher-order programming.
- 2 The following subclauses describe class template `result_of`, function template `mem_fn`, function object binders, and the polymorphic function wrapper function, as summarized in Table 2.

Table 2: Function object library summary

Subclause	Header(s)
3.4 <code>result_of</code>	<functional>
3.5 <code>mem_fn</code>	<functional>
3.6 Function object binders	<functional>
3.7 Function object wrappers	<functional>

3.1 Definitions

[tr.func.def]

- 1 The following definitions apply to this clause:
- 2 A *call signature* is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.
- 3 A *call wrapper* is an object of a call wrapper type.
- 4 A *call wrapper type* is a type that holds a callable object and supports a call operation that forwards to that object.
- 5 A *callable object* is an object of a callable type.
- 6 A *callable type* is a pointer to function, a pointer to member function, a pointer to member data, or a class type whose objects can appear immediately to the left of a function call operator.
- 7 A *target object* is the callable object held by a call wrapper object.

3.2 Additions to <functional> synopsis

[tr.func.syn]

```

namespace std {
namespace tr1 {
    // [3.4] class template result_of
    template <class FunctionCallType> class result_of;

    // [3.5] function template mem_fn
    template<class R, class T> unspecified mem_fn(R T::* pm);

```

```

// [3.6] Function object binders
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;

template<class F, class T1, ..., class TN>          unspecified bind(F f, T1 t1, ..., TN tN);
template<class R, class F, class T1, ..., class Tn> unspecified bind(F f, T1 t1, ..., TN tN);

namespace placeholders {
    // M is the implementation-defined number of placeholders
    extern unspecified _1;
    extern unspecified _2;
    .
    .
    .
    extern unspecified _M;
}

// [3.7] polymorphic function wrappers
class bad_function_call;

template<class Function> class function;

template<class Function>
    void swap(function<Function>&, function<Function>&);

template<class Function1, class Function2>
    void operator==(const function<Function1>&, const function<Function2>&);
template<class Function1, class Function2>
    void operator!=(const function<Function1>&, const function<Function2>&);

template <class Function>
    bool operator==(const function<Function>&, unspecified-null-pointer-type);
template <class Function>
    bool operator==(unspecified-null-pointer-type, const function<Function>&);
template <class Function>
    bool operator!=(const function<Function>&, unspecified-null-pointer-type);
template <class Function>
    bool operator!=(unspecified-null-pointer-type, const function<Function>&);
} // namespace tr1
} // namespace std

```

3.3 Requirements

[tr.func.require]

1 Define *INVOKE*(f, t1, t2, ..., tN) as follows:

- (t1.*f)(t2, ..., tN) when f is a pointer to a member function of a class T and t1 is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T;
- ((*t1).*f)(t2, ..., tN) when f is a pointer to a member function of a class T and t1 is not one of the types

23 Function objects

3.4 Function return types

described in the previous item;

- $t1.*f$ when f is a pointer to member data of a class T and $t1$ is an object of type T or a reference to an object of type T or a reference to an object of a type derived from T ;
- $(*t1).*f$ when f is a pointer to member data of a class T and $t1$ is not one of the types described in the previous item;
- $f(t1, t2, \dots, tN)$ in all other cases.

2 Define *INVOKE*($f, t1, t2, \dots, tN, R$) as *INVOKE*($f, t1, t2, \dots, tN$) implicitly converted to R .

3 If a call wrapper ([3.1]) has a *weak result type* the type of its member type *result_type* is based on the type T of the wrapper's target object ([3.1]):

- if T is a function, reference to function, or pointer to function type, *result_type* shall be a synonym for the return type of T ;
- if T is a pointer to member function, *result_type* shall be a synonym for the return type of T ;
- if T is a class type with a member type *result_type*, then *result_type* shall be a synonym for $T::result_type$;
- otherwise *result_type* shall not be defined.

4 Every call wrapper [3.1] shall be CopyConstructible. A *simple call wrapper* is a call wrapper that is Assignable and whose copy constructor and assignment operator do not throw exceptions. A *forwarding call wrapper* is a call wrapper that can be called with an argument list $t1, t2, \dots, tN$ where each t_i is an lvalue. The effect of calling a forwarding call wrapper with one or more arguments that are rvalues is implementation defined. [Note: in a typical implementation forwarding call wrappers have overloaded function call operators of the form

```
template<class T1, class T2, ..., class TN>
R operator()(T1& t1, T2& t2, ..., TN& tN) cv-qual;
```

—end note]

3.4 Function return types

[tr.func.ret]

```
namespace std {
namespace tr1 {
    template <class FunctionCallTypes> //F(T1, T2, ..., TN)
    class result_of {
    public :
        // types
        typedef see below type;
    };
} // namespace tr1
} // namespace std
```

1 Given an rvalue f of type F and values $t1, t2, \dots, tN$ of types $T1, T2, \dots, TN$, respectively, the type member is the result type of the expression $f(t1, t2, \dots, tN)$. The values t_i are lvalues when the corresponding type T_i is a reference type, and rvalues otherwise.

3.5 Function template `mem_fn`**Function objects 24**

- 2 The implementation may determine the type member via any means that produces the exact type of the expression `f(t1, t2, ..., tN)` for the given types. [Note: The intent is that implementations are permitted to use special compiler hooks —end note]
- 3 If `F` is not a function object defined by the standard library, and if either the implementation cannot determine the type of the expression `f(t1, t2, ..., tN)` or the expression is ill-formed, the implementation shall use the following process to determine the type member:
 1. If `F` is a function pointer or function reference type, type shall be the return type of the function type.
 2. If `F` is a member function pointer type, type shall be the return type of the member function type.
 3. If `F` is a possibly *cv*-qualified class type with a member type `result_type`, type shall be `typename F::result_type`.
 4. If `F` is a possibly *cv*-qualified class type with no member named `result_type` or if `typename F::result_type` is not a type:
 - (a) If `N=0` (no arguments), type shall be `void`.
 - (b) If `N>0`, type shall be `typename F::template result<F(T1, T2, ..., TN)>::type`.
 5. Otherwise, the program is ill-formed.

3.5 Function template `mem_fn`**[tr.func.memfn]**

```
template<class R, class T> unspecified mem_fn(R T::* pm);
```

- 1 **Returns:** A simple call wrapper ([3.1]) `f` such that the expression `f(t, a2, ..., aN)` is equivalent to `INVOKE(pm, t, a2, ..., aN)` ([3.3]). `f` shall have a nested type `result_type` that is a synonym for the return type of `pm` when `pm` is a pointer to member function.
- 2 The simple call wrapper shall be derived from `std::unary_function<cv T*, Ret>` when `pm` is a pointer to member function with *cv*-qualifier *cv* and taking no arguments, where *Ret* is `pm`'s return type.
- 3 The simple call wrapper shall be derived from `std::binary_function<cv T*, T1, Ret>` when `pm` is a pointer to member function with *cv*-qualifier *cv* and taking one argument of type `T1`, where *Ret* is `pm`'s return type.
- 4 **Throws:** Nothing.
- 5 **Notes:** Implementations may implement `mem_fn` as a set of overloaded function templates.

3.6 Function object binders**[tr.func.bind]**

- 1 3.6 describes a uniform mechanism for binding arguments of function objects.

3.6.1 Class template `is_bind_expression`**[tr.func.bind.isbind]**

```
namespace std {
namespace tr1 {
template<class T> struct is_bind_expression {
    static const bool value = see below;
};
```

25 Function objects

3.6 Function object binders

```
};
} // namespace tr1
} // namespace std
```

- 1 `is_bind_expression` can be used to detect function objects generated by `bind`. `bind` uses `is_bind_expression` to detect subexpressions. Users may specialize this template to indicate that a type should be treated as a subexpression in a `bind` call.

```
static const bool value;
```

- 2 `true` if `T` is a type returned from `bind`, `false` otherwise.

3.6.2 Class template `is_placeholder`

[tr.func.bind.isplace]

```
namespace std {
namespace tr1 {
    template<class T> struct is_placeholder {
        static const int value = see below;
    };
} // namespace tr1
} // namespace std
```

- 1 `is_placeholder` can be used to detect the standard placeholders `_1`, `_2`, and so on. `bind` uses `is_placeholder` to detect placeholders. Users may specialize this template to indicate a placeholder type.

```
static const int value;
```

- 2 `value` is `J` if `T` is the type of `std::tr1::placeholders::_J`, 0 otherwise.

3.6.3 Function template `bind`

[tr.func.bind.bind]

```
template<class F, class T1, class T2, ..., class TN>
    unspecified bind(F f, T1 t1, T2 t2, ..., TN tN);
```

- 1 *Requires:* `F` and `Ti` shall be CopyConstructible. *INVOKE* (`f`, `w1`, `w2`, ..., `wN`) ([3.3]) shall be a valid expression for some values `w1`, `w2`, ..., `wN`.
- 2 *Returns:* A forwarding call wrapper `g` with a weak result type ([3.3]). The effect of `g(u1, u2, ..., uM)` shall be *INVOKE*(`f`, `v1`, `v2`, ..., `vN`, `result_of<F cv (V1, V2, ..., VN)>::type`), where `cv` represents the `cv`-qualifiers of `g` and the values and types of the bound arguments `v1`, `v2`, ..., `vN` are determined as specified below.

```
template<class R, class F, class T1, class T2, ..., class TN>
    unspecified bind(F f, T1 t1, T2 t2, ..., TN tN);
```

- 3 *Requires:* `F` and `Ti` shall be CopyConstructible. *INVOKE* (`f`, `w1`, `w2`, ..., `wN`) shall be a valid expression for some values `w1`, `w2`, ..., `wN`.

3.7 Polymorphic function wrappers**Function objects 26**

- 4 *Returns:* A forwarding call wrapper *g* with a nested type *result_type* defined as a synonym for *R*. The effect of *g*(*u1*, *u2*, ..., *uM*) shall be *INVOKE*(*f*, *v1*, *v2*, ..., *vN*, *R*), where the values and types of the bound arguments *v1*, *v2*, ..., *vN* are determined as specified below.
- 5 The values of the *bound arguments* *v1*, *v2*, ..., *vN* and their corresponding types *V1*, *V2*, ..., *VN* depend on the type of the corresponding argument *ti* of type *Ti* in the call to *bind* and the *cv*-qualifiers *cv* of the call wrapper *g* as follows:
- if *ti* is of type *reference_wrapper*<*T*> the argument is *ti.get()* and its type *Vi* is *T*&;
 - if the value of *std::tr1::is_bind_expression*<*Ti*>::value is true the argument is *ti(u1, u2, ..., uM)* and its type *Vi* is *result_of*<*Ti cv (U1*&, *U2*&, ..., *UM*&)>::type;
 - if the value *j* of *std::tr1::is_placeholder*<*Ti*>::value is not zero the argument is *uj* and its type *Vi* is *Uj*&;
 - otherwise the value is *ti* and its type *Vi* is *Ti cv* &.

3.6.4 Placeholders**[tr.func.bind.place]**

```

namespace std {
namespace tr1 {
    namespace placeholders {
        // M is the implementation-defined number of placeholders
        extern unspecified _1;
        extern unspecified _2;
        .
        .
        .
        extern unspecified _M;
    }
} // namespace tr1
} // namespace std

```

- 1 All placeholder types shall be *DefaultConstructible* and *CopyConstructible*, and their default constructors and copy constructors shall not throw exceptions. It is implementation defined whether placeholder types are *Assignable*. *Assignable* placeholders' copy assignment operators shall not throw exceptions.

3.7 Polymorphic function wrappers**[tr.func.wrap]**

- 1 3.7 describes a polymorphic wrapper class that encapsulates arbitrary function objects.

3.7.1 Class *bad_function_call***[tr.func.wrap.badcall]**

- 1 An exception of type *bad_function_call* is thrown by *function::operator()* ([3.7.2.4]) when the function wrapper object has no target.

```

namespace std {
namespace tr1 {
    class bad_function_call : public std::exception
    {
    public:

```

27 Function objects**3.7 Polymorphic function wrappers**

```

    // [3.7.1.1] constructor
    bad_function_call();
};
} // namespace tr1
} // namespace std

```

3.7.1.1 bad_function_call constructor**[tr.func.wrap.badcall.const]**

```
bad_function_call();
```

1 *Effects:* constructs a bad_function_call object.

3.7.2 Class template function**[tr.func.wrap.func]**

```

namespace std {
namespace tr1 {
    // Function type R (T1, T2, ..., TN), 0 ≤ N ≤ Nmax
    template<class Function>
    class function
    : public unary_function<T1, R>           // iff N == 1
    : public binary_function<T1, T2, R>      // iff N == 2
    {
    public:
        typedef R result_type;

        // [3.7.2.1] construct/copy/destroy
        explicit function();
        function(unspecified-null-pointer-type);
        function(const function&);
        template<class F> function(F);

        function& operator=(const function&);
        function& operator=(unspecified-null-pointer-type);
        template<class F> function& operator=(F);
        template<class F> function& operator=(reference_wrapper<F>);

        ~function();

        // [3.7.2.2] function modifiers
        void swap(function&);

        // [3.7.2.3] function capacity
        operator unspecified-bool-type() const;

        // [3.7.2.4] function invocation
        R operator()(T1, T2, ..., TN) const;

        // [3.7.2.5] function target access
        const std::type_info& target_type() const;
    };
}
}

```

```

template <typename T>      T* target();
template <typename T> const T* target() const;

private:
    // [3.7.2.6] undefined operators
    template<class Function2> bool operator==(const function<Function2>&);
    template<class Function2> bool operator!=(const function<Function2>&);
};

// [3.7.2.7] Null pointer comparisons
template <class Function>
    bool operator==(const function<Function>&, unspecified-null-pointer-type);

template <class Function>
    bool operator==(unspecified-null-pointer-type, const function<Function>&);

template <class Function>
    bool operator!=(const function<Function>&, unspecified-null-pointer-type);

template <class Function>
    bool operator!=(unspecified-null-pointer-type, const function<Function>&);

// [3.7.2.8] specialized algorithms
template<class Function> void swap(function<Function>&, function<Function>&);
} // namespace tr1
} // namespace std

```

- 1 The function class template provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary callable objects ([3.1]), given a call signature ([3.1]), allowing functions to be first-class objects.
- 2 A function object *f* of type *F* is Callable for argument types *T*₁, *T*₂, ..., *T*_{*N*} and a return type *R*, if, given *N* values *t*₁, *t*₂, ..., *t*_{*N*} of types *T*₁, *T*₂, ..., *T*_{*N*}, respectively, *INVOKE*(*f*, *t*₁, *t*₂, ..., *t*_{*N*}) is well-formed ([3.3]) and, if *R* is not void, convertible to *R*.
- 3 The function class template is a call wrapper ([3.1]) whose call signature ([3.1]) is *R*(*T*₁, *T*₂, ..., *T*_{*N*}).

3.7.2.1 function construct/copy/destroy

[tr.func.wrap.func.con]

```

explicit function();

1      Postconditions: !*this.
2      Throws: nothing.

function(unspecified-null-pointer-type);

3      Postconditions: !*this.
4      Throws: nothing.

function(const function& f);

```

29 Function objects**3.7 Polymorphic function wrappers**

5 *Postconditions:* `!*this` if `!f`; otherwise, `*this` targets a copy of `f.target()`.

6 *Throws:* shall not throw exceptions if `f`'s target is a function pointer or a function object passed via `reference_wrapper`. Otherwise, may throw `bad_alloc` or any exception thrown by the copy constructor of the stored function object.

```
template<class F> function(F f);
```

7 *Requires:* `f` shall be callable for argument types `T1`, `T2`, ..., `TN` and return type `R`.

8 *Postconditions:* `!*this` if any of the following hold:

- `f` is a `NULL` function pointer.
- `f` is a `NULL` member function pointer.
- `F` is an instance of the `function` class template, and `!f`

9 Otherwise, `*this` targets a copy of `f` if `f` is not a pointer to member function, and targets a copy of `mem_fn(f)` if `f` is a pointer to member function.

10 *Throws:* shall not throw exceptions when `f` is a function pointer or a `reference_wrapper<T>` for some `T`. Otherwise, may throw `bad_alloc` or any exception thrown by `F`'s copy constructor.

```
function& operator=(const function& f);
```

11 *Effects:* `function(f).swap(*this)`;

12 *Returns:* `*this`

```
function& operator=(unspecified-null-pointer-type);
```

13 *Effects:* If `*this != NULL`, destroys the target of `this`.

14 *Postconditions:* `!(*this)`.

15 *Returns:* `*this`

```
template<class F> function& operator=(F f);
```

16 *Effects:* `function(f).swap(*this)`;

17 *Returns:* `*this`

```
template<class F> function& operator=(reference_wrapper<F> f);
```

18 *Effects:* `function(f).swap(*this)`;

19 *Returns:* `*this`

20 *Throws:* nothing.

```
~function();
```

21 *Effects:* If `*this != NULL`, destroys the target of `this`.

3.7 Polymorphic function wrappers

Function objects 30

3.7.2.2 function modifiers**[tr.func.wrap.func.mod]**

```
void swap(function& other);
```

- 1 *Effects:* interchanges the targets of **this* and *other*.
- 2 *Throws:* nothing.

3.7.2.3 function capacity**[tr.func.wrap.func.cap]**

```
operator unspecified-bool-type() const
```

- 1 *Returns:* if **this* has a target, returns a value that will evaluate *true* in a boolean context; otherwise, returns a value that will evaluate *false* in a boolean context. The value type returned shall not be convertible to *int*.
- 2 *Throws:* nothing.
- 3 *[Note:* This conversion can be used in contexts where a *bool* is expected (e.g., an *if* condition); however, implicit conversions (e.g., to *int*) that can occur with *bool* are not allowed, eliminating some sources of user error. One possible implementation choice for this type is pointer-to-member. —end note]

3.7.2.4 function invocation**[tr.func.wrap.func.inv]**

```
R operator()(T1 t1, T2 t2, ..., TN tN) const
```

- 1 *Effects:* *INVOKE*(*f*, *t1*, *t2*, ..., *tN*, *R*) ([3.3]), where *f* is the target object ([3.1]) of **this*.
- 2 *Returns:* nothing, if *R* is *void*, otherwise the return value of *INVOKE*(*f*, *t1*, *t2*, ..., *tN*, *R*).
- 3 *Throws:* *bad_function_call* if *!*this*; otherwise, any exception thrown by the wrapped function object.

3.7.2.5 function target access**[tr.func.wrap.func.targ]**

```
const std::type_info& target_type() const;
```

- 1 *Returns:* If **this* has a target of type *T*, *typeid(T)*; otherwise, *typeid(void)*.
- 2 *Throws:* nothing.

```
template<typename T> T* target();
template<typename T> const T* target() const;
```

- 3 *Requires:* *T* is a function object type that is *Callable* ([3.7.2]) for parameter types *T1*, *T2*, ..., *TN* and return type *R*.
- 4 *Returns:* If *type() == typeid(T)*, a pointer to the stored function target; otherwise a null pointer.
- 5 *Throws:* nothing.

31 Function objects

3.7 Polymorphic function wrappers

3.7.2.6 undefined operators

[tr.func.wrap.func.undef]

```
template<class Function2> bool operator==(const function<Function2>&);
template<class Function2> bool operator!=(const function<Function2>&);
```

- 1 These member functions shall be left undefined.
- 2 [Note: the boolean-like conversion opens a loophole whereby two function instances can be compared via == or !=. These undefined void operators close the loophole and ensure a compile-time error. —end note]

3.7.2.7 null pointer comparison operators

[tr.func.wrap.func.nullptr]

```
template <class Function>
    bool operator==(const function<Function>& f, unspecified-null-pointer-type);
```

```
template <class Function>
    bool operator==(unspecified-null-pointer-type, const function<Function>& f);
```

- 1 Returns: !f.
- 2 Throws: nothing.

```
template <class Function>
    bool operator!=(const function<Function>& f, unspecified-null-pointer-type);
```

```
template <class Function>
    bool operator!=(unspecified-null-pointer-type, const function<Function>& f);
```

- 3 Returns: (bool) f.
- 4 Throws: nothing.

3.7.2.8 specialized algorithms

[tr.func.wrap.func.alg]

```
template<class Function>
    void swap(function<Function>& f1, function<Function>& f2);
```

- 1 Effects: f1.swap(f2);

(blank page)

4 Metaprogramming and type traits [tr.meta]

- 1 This clause describes components used by C++ programs, particularly in templates, to support the widest possible range of types, optimise template code usage, detect type related user errors, and perform type inference and transformation at compile time.
- 2 The following subclauses describe type traits requirements, unary type traits, traits that describe relationships between types, and traits that perform transformations on types, as summarized in Table 3.

Table 3: Type traits library summary

Subclause	Header(s)
4.1 Requirements	
4.5 Unary type traits	<code><type_traits></code>
4.6 Relationships between types	<code><type_traits></code>
4.7 Transformations between types	<code><type_traits></code>

4.1 Requirements

[tr.meta.rqmts]

- 1 A *UnaryTypeTrait* is a template that describes a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the property being described. It shall be *DefaultConstructible* and derived, directly or indirectly, from an instance of the template `integral_constant` (4.3), with the arguments to the template `integral_constant` determined by the requirements for the particular property being described.
- 2 A *BinaryTypeTrait* is a template that describes a relationship between two types. It shall be a class template that takes two template type arguments and, optionally, additional arguments that help define the relationship being described. It shall be *DefaultConstructible* and derived, directly or indirectly, from an instance of the template `integral_constant` (4.3), with the arguments to the template `integral_constant` determined by the requirements for the particular relationship being described.
- 3 A *TransformationTypeTrait* is a template that modifies a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the modification. It shall define a nested type named `type`, which shall be a synonym for the modified type.

4.2 Header `<type_traits>` synopsis

[tr.meta.type.synop]

```
namespace std {
namespace tr1 {
    // [4.3] helper class:
```

```

template <class T, T v> struct integral_constant;
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;

// [4.5.1] primary type categories:
template <class T> struct is_void;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_reference;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;

// [4.5.2] composite type categories:
template <class T> struct is_arithmetic;
template <class T> struct is_fundamental;
template <class T> struct is_object;
template <class T> struct is_scalar;
template <class T> struct is_compound;
template <class T> struct is_member_pointer;

// [4.5.3] type properties:
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_pod;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct has_trivial_constructor;
template <class T> struct has_trivial_copy;
template <class T> struct has_trivial_assign;
template <class T> struct has_trivial_destructor;
template <class T> struct has_nothrow_constructor;
template <class T> struct has_nothrow_copy;
template <class T> struct has_nothrow_assign;
template <class T> struct has_virtual_destructor;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
template <class T> struct alignment_of;
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;

// [4.6] type relations:
template <class T, class U> struct is_same;
template <class Base, class Derived> struct is_base_of;

```

35 Metaprogramming and type traits

4.3 Helper classes

```

template <class From, class To> struct is_convertible;

// [4.7.1] const-volatile modifications:
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;

// [4.7.2] reference modifications:
template <class T> struct remove_reference;
template <class T> struct add_reference;

// [4.7.3] array modifications:
template <class T> struct remove_extent;
template <class T> struct remove_all_extents;

// [4.7.4] pointer modifications:
template <class T> struct remove_pointer;
template <class T> struct add_pointer;

// [4.8] other transformations:
template <std::size_t Len, std::size_t Align> struct aligned_storage;
} // namespace tr1
} // namespace std

```

4.3 Helper classes

[tr.meta.help]

```

template <class T, T v>
struct integral_constant
{
    static const T          value = v;
    typedef T              value_type;
    typedef integral_constant<T,v> type;
};
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;

```

- 1 The class template `integral_constant` and its associated typedefs `true_type` and `false_type` are used as base classes to define the interface for various type traits.

4.4 General Requirements

[tr.meta.requirements]

- 1 Tables 4, 5, 6, and 8 define type predicates. Each type predicate `pred<T>` shall be a *UnaryTypeTrait* (4.1), derived directly or indirectly from `true_type` if the corresponding condition is true, otherwise from `false_type`. Each type predicate `pred<T, U>` shall be a *BinaryTypeTrait* (4.1), derived directly or indirectly from `true_type` if the corresponding condition is true, otherwise from `false_type`.
- 2 Table 7 defines various type queries. Each type query shall be a *UnaryTypeTrait* (4.1), derived directly or indirectly from `integral_constant<std::size_t, value>`, where `value` is the value of the property being queried.

- 3 Tables 9, 10, 11, and 12 define type transformations. Each transformation shall be a *TransformationTrait* (4.1).
- 4 Table 13 defines a template that can be instantiated to define a type with a specific alignment and size.

4.5 Unary Type Traits**[tr.meta.unary]**

- 1 This sub-clause contains templates that may be used to query the properties of a type at compile time.
- 2 For all of the class templates X declared in this clause, instantiating that template with a template-argument that is a class template specialization may result in the implicit instantiation of the template argument if and only if the semantics of X require that the argument must be a complete type.

4.5.1 Primary Type Categories**[tr.meta.unary.cat]**

- 1 The primary type categories correspond to the descriptions given in section [basic.types] of the C++ standard.
- 2 For any given type T, the result of applying one of these templates to T and to *cv-qualified* T shall yield the same result.
- 3 The behavior of a program that adds specializations for any of the class templates defined in this clause is undefined.
- 4 [Note: For any given type T, exactly one of the primary type categories has a value member that evaluates to true. —end note]

Table 4: Primary Type Category Predicates

Template	Condition	Comments
template <class T> struct is_void;	T is void	
template <class T> struct is_integral;	T is an integral type ([basic.fundamental])	
template <class T> struct is_floating_point;	T is a floating point type ([basic.fundamental])	
template <class T> struct is_array;	T is an array type ([basic.compound])	Class template array ([6.2]) is <i>not</i> an array type.
template <class T> struct is_pointer;	T is a pointer type ([basic.compound])	Includes function pointers, but not pointers to members.
template <class T> struct is_reference;	T is a reference type ([basic.fundamental])	Includes references to functions.
template <class T> struct is_member_object_pointer;	T is a pointer to data member	
template <class T> struct is_member_function_pointer;	T is a pointer to member function	
template <class T> struct is_enum;	T is an enumeration type ([basic.compound])	
template <class T> struct is_union;	T is a union type ([basic.compound])	
template <class T> struct is_class;	T is a class type but not a union type ([basic.compound])	

37 Metaprogramming and type traits

4.5 Unary Type Traits

template <class T> struct is_function;	T is a function type ([basic.compound])
---	--

4.5.2 Composite type traits

[tr.meta.unary.comp]

- 1 These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section [basic.types].
- 2 For any given type T, the result of applying one of these templates to T, and to *cv-qualified* T shall yield the same result.
- 3 The behavior of a program that adds specializations for any of the class templates defined in this clause is undefined.

Table 5: Composite Type Category Predicates

Template	Condition	Comments
template <class T> struct is_arithmetic;	T is an arithmetic type ([basic.fundamental])	
template <class T> struct is_fundamental;	T is a fundamental type ([basic.fundamental])	
template <class T> struct is_object;	T is an object type ([basic.types])	
template <class T> struct is_scalar;	T is a scalar type ([basic.types])	
template <class T> struct is_compound;	T is a compound type ([basic.compound])	
template <class T> struct is_member_pointer;	T is a pointer to a member or member function	

4.5.3 Type properties

[tr.meta.unary.prop]

- 1 These templates provide access to some of the more important properties of types.
- 2 It is unspecified whether the library defines any full or partial specialisations of any of these templates. A program may specialise any of these templates on a user-defined type, provided the semantics of the specialisation match those given for the template in its description.

Table 6: Type Property Predicates

Template	Condition	Preconditions
template <class T> struct is_const;	T is const-qualified ([basic.qualifier])	
template <class T> struct is_volatile;	T is volatile-qualified ([basic.qualifier])	
template <class T> struct is_pod;	T is a POD type ([basic.types])	T shall be a complete type.

template <class T> struct is_empty;	T is an empty class ([class])	T shall be a complete type.
template <class T> struct is_polymorphic;	T is a polymorphic class ([class.virtual])	T shall be a complete type.
template <class T> struct is_abstract;	T is an abstract class ([class.abstract])	T shall be a complete type.
template <class T> struct has_trivial_constructor;	The default constructor for T is trivial ([class.ctor])	T shall be a complete type.
template <class T> struct has_trivial_copy;	The copy constructor for T is trivial ([class.copy])	T shall be a complete type.
template <class T> struct has_trivial_assign;	The assignment operator for T is trivial ([class.copy])	T shall be a complete type.
template <class T> struct has_trivial_destructor;	The destructor for T is trivial ([class.dtor])	T shall be a complete type.
template <class T> struct has_nothrow_constructor;	The default constructor for T has an empty exception specification or can otherwise be deduced never to throw an exception	T shall be a complete type.
template <class T> struct has_nothrow_copy;	The copy constructor for T has an empty exception specification or can otherwise be deduced never to throw an exception	T shall be a complete type.
template <class T> struct has_nothrow_assign;	The assignment operator for T has an empty exception specification or can otherwise be deduced never to throw an exception	T shall be a complete type.
template <class T> struct has_virtual_destructor;	T has a virtual destructor ([class.dtor])	T shall be a complete type.
template <class T> struct is_signed;	T is a signed integral type ([basic.fundamental])	
template <class T> struct is_unsigned;	T is an unsigned integral type ([basic.fundamental])	

Table 7: Type Property Queries

Template	value
template <class T> struct alignment_of;	An integer value representing the number of bytes of the alignment of objects of type T; an object of type T may be allocated at an address that is a multiple of its alignment ([basic.types]). <i>Precondition:</i> T shall be a complete type.