

江添亮の 詳説C++17

■江添亮 著



ASCII
DWANGO

本文中記載の会社名、商品名、一般開発者登録商標。
、本文中のTM・[©]・[®]表示は明記されている。

新C++17 不具修正、日々楽々新機能追加。結果、C++ 特徴の静的型付け、近年の型弱言語に匹敵する柔軟な記述が可能。

1990年代以降、最も良質な手法の価値失墜、逆悪手法成下。同時に昔の現実の手法は今でも方法として存在。現在活発に使用される言語、常時代合機能廃止、必要機能を追加必要。C++が発展し留まり、今後 C++ 使用は続限、修正機能追加行。

<https://github.com/EzoeRyou/cpp17book>

本書は GPLv3 です。

本書の執筆者は株式会社エゾエリヤ GitHub 上で Pull Request を送ることで多
くの貢献者に協力いただき、誤りを正し、より良い記述を実現しました。この場を
借りて謝意を表します。

本書の誤りを見つけた場合、Pull Request を送る先は

<https://github.com/EzoeRyou/cpp17book>

です。

江添亮

序

0.1 C++ の規格

言語 C++ は ISO 傘下国際規格 ISO/IEC 14882 によって制定されている。この規格は数年ごとに変更されている。一般に C++ の規格は参照規格、規格制定の西暦の下二桁を取って、C++98 (1998 年発行) から C++11 (2011 年発行) と呼ぶ。現在発行されている C++ の規格は以下の通り。

0.1.1 C++98

C++98 は 1998 年に制定された最初の C++ の規格である。本来は 1994 年と 1995 年に制定する予定であったが、1998 年になった。

0.1.2 C++03

C++03 は C++98 の文面を曖昧な点を修正した修正規格である。2003 年に制定された。新機能を追加したわけではない。

0.1.3 C++11

C++11 は制定途中の段階で元 C++0x と呼ぶ。200x 年頃に規格制定する予定であった。予定は大幅に遅れ、2011 年に規格制定された。C++11 は多くの新機能を追加した。

0.1.4 C++14

C++14 は 2014 年に制定された。C++11 の文面を誤り修正や他、少くも新機能を追加した。本書は解説する。

目次

はじめに	iii
序	v
0.1 C++ 規格	v
0.1.1 C++98	v
0.1.2 C++03	v
0.1.3 C++11	v
0.1.4 C++14	v
0.1.5 C++17	vi
0.2 C++ 将来規格	vi
0.2.1 C++20	vi
0.3 言語	vi
第1章 SD-6 C++ のための機能テスト推奨	1
1.1 機能	1
1.2 <code>__has_include</code> 式: 存在判定	3
1.3 <code>__has_cpp_attribute</code> 式	4
第2章 C++14 のコア言語の新機能	5
2.1 二進数	5
2.2 数値区切り文字	5
2.3 <code>[[deprecated]]</code> 属性	6
2.4 通常関数戻り値型推定	8
2.5 <code>decltype(auto)</code> : 厳格 <code>auto</code>	9
2.6	14
2.7 初期化	15
2.8 変数	18
2.8.1 意味同型違い定数	21

3.20.4	初期化子型配列、std::tuple_size<E> 完全形 名前場合	71
3.20.5	上記以外場合	73
3.21	inline 変数	75
3.21.1	inline 歴史的意味	75
3.21.2	現代 inline 意味	76
3.21.3	inline 変数意味	78
3.22	可変長 using 宣言	79
3.23	std::byte : 表現型	81
第 4 章	C++17 の型安全な値を格納するライブラリ	85
4.1	variant : 型安全 union	85
4.1.1	使方	85
4.1.2	型非安全古典的 union	86
4.1.3	variant 宣言	88
4.1.4	variant 初期化 初期化 初期化 variant 値渡場合 in_place_type emplace 構築	88 88 89 89 90
4.1.5	variant 破棄	91
4.1.6	variant 代入	92
4.1.7	variant emplace	92
4.1.8	variant 値入確認 valueless_by_exception 関数 index 関数	93 93 94
4.1.9	swap	94
4.1.10	variant_size<T> : variant 保持型数取得	95
4.1.11	variant_alternative<I, T> : 型型返	96
4.1.12	holds_alternative : variant 指定型値保持確認	96
4.1.13	get<I>(v) : 型値取得	97
4.1.14	get<T>(v) : 型値取得	99
4.1.15	get_if : 値保持場合取得	100
4.1.16	variant 比較 同一性比較	101 101

	大小比較	102
4.1.17	visit : variant 保持値受取	103
4.2	any : 型値保持	104
4.2.1	使用方	104
4.2.2	any 構築破棄	105
4.2.3	in_place_type 構築	105
4.2.4	any 代入	106
4.2.5	any 関数	106
	emplace	106
	reset : 破棄	106
	swap : 交換	107
	has_value : 値保持確認調	107
	type : 保持型 type_info 得	108
4.2.6	any 関数	108
	make_any<T> : T 型 any 作	108
	any_cast : 保持値取出	109
4.3	optional : 値保有、構築破棄	110
4.3.1	使用方	110
4.3.2	optional 実引数	112
4.3.3	optional 構築	112
4.3.4	optional 代入	113
4.3.5	optional 破棄	113
4.3.6	swap	114
4.3.7	has_value : 値保持確認	114
4.3.8	operator bool : 値保持確認	115
4.3.9	value : 保持値取得	115
4.3.10	value_or : 値保持値返	116
4.3.11	reset : 保持値破棄	117
4.3.12	optional 同土比較	117
	同一性比較	117
	大小比較	118
4.3.13	optional std::nullopt 比較	119
4.3.14	optional<T> T 比較	119
4.3.15	make_optional<T> : optional<T> 返	119
4.3.16	make_optional<T, Args ...> : optional<T> 構築返	120

第 5 章	string_view : 文字列ラッパー	121
5.1	使用方	121
5.2	basic_string_view	123
5.3	文字列の所有、非所有	123
5.4	string_view の構築	125
5.4.1	明示的構築	126
5.4.2	null 終端文字型配列	126
5.4.3	文字型文字数	126
5.4.4	文字列変換関数	127
5.5	string_view の操作	128
5.5.1	remove_prefix/remove_suffix : 先頭、末尾要素削除	129
5.6	string_view の定義	130
第 6 章	メモリリソース : 動的ストレージ確保ライブラリ	133
6.1	メモリリソース	133
6.1.1	メモリリソースの使用方	134
6.1.2	メモリリソースの作り方	135
6.2	polymorphic_allocator : 動的メモリリソースの実現	137
6.2.1	メモリリソース	138
6.3	メモリリソース全体の使用	139
6.3.1	new_delete_resource()	139
6.3.2	null_memory_resource()	139
6.3.3	メモリリソース	140
6.4	標準メモリリソース	140
6.5	メモリリソース	142
6.5.1	メモリリソース	142
6.5.2	synchronized/unsynchronized_pool_resource	145
6.5.3	pool_options	146
6.5.4	メモリリソース	146
6.5.5	メモリリソース関数	147
	release()	147
	upstream_resource()	147
	options()	147
6.6	メモリリソース	147
6.6.1	メモリリソース	149

6.6.2	<code>std::weak_ptr</code>	150
6.6.3	<code>std::weak_ptr</code> 他操作	151
	<code>release()</code>	151
	<code>upstream_resource()</code>	152
第 7 章	並列アルゴリズム	153
7.1	並列実行 <code>std::parallel</code>	153
7.2	使用方	155
7.3	並列 <code>std::parallel</code> 詳細	156
7.3.1	並列 <code>std::parallel</code>	156
7.3.2	<code>std::parallel</code> 提供関数 <code>std::parallel</code> 制約	157
	実引数与 <code>std::parallel</code> 直接、間接変更 <code>std::parallel</code>	157
	<code>std::parallel</code>	157
	実引数与 <code>std::parallel</code> 一意性依存 <code>std::parallel</code>	158
	<code>std::parallel</code>	158
	<code>std::parallel</code> 競合同期	159
7.3.3	例外	160
7.3.4	実行 <code>std::parallel</code>	161
	<code>is_execution_policy_traits</code>	161
	<code>std::parallel</code> 実行 <code>std::parallel</code>	161
	<code>std::parallel</code> 実行 <code>std::parallel</code>	162
	<code>std::parallel</code> 非 <code>std::parallel</code> 実行 <code>std::parallel</code>	162
	実行 <code>std::parallel</code>	162
第 8 章	数学の特殊関数群	165
8.1	<code>std::laguerre_polynomials</code> (Laguerre polynomials)	166
8.2	<code>std::associated_laguerre_polynomials</code> (Associated Laguerre polynomials)	166
8.3	<code>std::legendre_polynomials</code> (Legendre polynomials)	166
8.4	<code>std::associated_legendre_functions</code> (Associated Legendre functions)	167
8.5	球面 <code>std::associated_legendre_functions</code> (Spherical associated Legendre functions)	167
8.6	<code>std::hermite_polynomials</code> (Hermite polynomials)	168
8.7	<code>std::beta_function</code> (Beta function)	168
8.8	第 1 種完全楕円積分 (Complete elliptic integral of the first kind)	169
8.9	第 2 種完全楕円積分 (Complete elliptic integral of the second kind)	169
8.10	第 3 種完全楕円積分 (Complete elliptic integral of the third kind)	169
8.11	第 1 種不完全楕円積分 (Incomplete elliptic integral of the first kind)	170

8.12	第 2 種不完全楕円積分 (Incomplete elliptic integral of the second kind)	170
8.13	第 3 種不完全楕円積分 (Incomplete elliptic integral of the third kind)	171
8.14	第 1 種 $J_n(x)$ 関数 (Cylindrical Bessel functions of the first kind)	171
8.15	$Y_n(x)$ 関数 (Cylindrical Neumann functions)	171
8.16	第 1 種変形 $I_n(x)$ 関数 (Regular modified cylindrical Bessel functions)	172
8.17	第 2 種変形 $K_n(x)$ 関数 (Irregular modified cylindrical Bessel functions)	172
8.18	第 1 種球 $j_n(x)$ 関数 (Spherical Bessel functions of the first kind)	173
8.19	球 $y_n(x)$ 関数 (Spherical Neumann functions)	173
8.20	指数積分 (Exponential integral)	174
8.21	$\zeta(s)$ 関数 (Riemann zeta function)	174
第 9 章	その他の標準ライブラリ	175
9.1	interpolate 関数 (多項式近似)	175
9.2	<code>std::uncaught_exceptions</code>	176
9.3	<code>apply</code> : tuple 要素実引数関数呼出	178
9.4	Searcher : 検索	179
9.4.1	default_searcher	179
9.4.2	boyer_moore_searcher	180
9.4.3	boyer_moore_horspool_searcher	182
9.5	sample : 乱択抽出	183
9.5.1	乱択抽出	183
9.5.2	S : 選択標本、要素数 n 集合標本選択	186
9.5.3	R : 保管標本、要素数 n 集合標本選択	187
9.5.4	C++ <code>sample</code>	189
9.6	<code>shared_ptr<T[]></code> : 配列対 <code>shared_ptr</code>	192
9.7	<code>as_const</code> : const 性付与	193
9.8	<code>make_from_tuple</code> : tuple 要素実引数関数呼出	194
9.9	<code>invoke</code> : 指定関数指定実引数呼出	195
9.10	<code>not_fn</code> : 戻値否定	196

9.11	管理 traits	196
9.11.1	addressof	196
9.11.2	uninitialized_default_construct	197
9.11.3	uninitialized_value_construct	198
9.11.4	uninitialized_copy	198
9.11.5	uninitialized_move	199
9.11.6	uninitialized_fill	199
9.11.7	destroy	199
9.12	shared_ptr::weak_type	200
9.13	void_t	201
9.14	bool_constant	201
9.15	type_traits	201
9.15.1	変数 traits 版 traits	201
9.15.2	論理演算 traits	202
	conjunction : 論理積	202
	disjunction : 論理和	203
	negation : 否定	203
9.15.3	is_invocable : 呼出可能確認 traits	204
9.15.4	has_unique_object_representations : 同値内部表現同一確認 traits	205
9.15.5	is_nothrow_swappable : 無例外 swap 可能確認 traits	206
9.16	不完全型 traits	206
9.17	emplace 戻値	206
9.18	map unordered_map 変更	207
9.18.1	try_emplace	207
9.18.2	insert_or_assign	208
9.19	連想配列 splice 操作	209
9.19.1	merge	210
9.19.2	traits	211
9.19.3	extract : traits 取得	213
9.19.4	insert : traits 要素追加	215
9.19.5	traits 利用例	218
	traits 再確保、traits 一部要素別 traits 移	218
	traits 寿命超要素存続 traits	218
	map traits 変更	219

9.20	<code>std::clamp</code> 関数	219
9.21	<code>clamp</code>	220
9.22	3 次元 <code>hypot</code>	221
9.23	<code>atomic<T>::is_lock_free</code>	221
9.24	<code>scoped_lock</code> : 可変長引数 <code>lock_guard</code>	221
9.25	<code>std::byte</code>	222
9.26	最大公約数 (gcd) / 最小公倍数 (lcm)	222
9.26.1	<code>gcd</code> : 最大公約数	222
9.26.2	<code>lcm</code> : 最小公倍数	223
第 10 章	ファイルシステム	225
10.1	名前空間	225
10.2	POSIX 準拠	226
10.3	<code>std::filesystem::path</code> 全体像	226
10.4	<code>std::filesystem::path</code> 処理	227
10.4.1	例外	227
10.4.2	非例外	228
10.5	<code>path</code> : <code>std::filesystem::path</code> 文字列	229
10.5.1	<code>path</code> : <code>std::filesystem::path</code> 文字列	230
10.5.2	<code>std::filesystem::path</code> 操作	234
10.6	<code>file_status</code>	236
10.7	<code>directory_entry</code>	238
10.8	<code>directory_iterator</code>	240
10.8.1	<code>std::filesystem::directory_iterator</code> 処理	241
10.9	<code>recursive_directory_iterator</code>	242
10.9.1	<code>std::filesystem::recursive_directory_iterator</code>	242
10.9.2	<code>depth</code> : 深さ取得	244
10.9.3	<code>pop</code> : 現在 <code>std::filesystem::recursive_directory_iterator</code> 列挙中止	244
10.9.4	<code>recursion_pending</code> : 現在 <code>std::filesystem::recursive_directory_iterator</code> 再帰	245
10.10	<code>std::filesystem::path</code> 操作関数	248
10.10.1	<code>std::filesystem::path</code> 取得	248
	<code>current_path</code>	248
	<code>temp_directory_path</code>	248
10.10.2	<code>std::filesystem::path</code> 操作	248
	<code>absolute</code>	248

目次

canonical	248
weakly_canonical	248
relative	249
proximate	249
10.10.3 作成	249
create_directory	249
create_directories	249
create_directory_symlink	250
create_symlink	250
create_hard_link	251
10.10.4 操作	251
copy_file	251
copy	251
copy_symlink	252
10.10.5 削除	253
remove	253
remove_all	253
10.10.6 変更	254
permissions	254
rename	255
resize_file	256
10.10.7 情報取得	256
操作権限判定	256
status	259
status_known	259
symlink_status	259
equivalent	259
exists	259
file_size	259
hard_link_count	260
last_write_time	260
read_symlink	262
space	262

索引

266

第1章

SD-6 C++ のための 機能テスト推奨

C++17 機能は C 機能に追加。

1.1 機能テストマクロ

機能、C++ 実装 (C++ 特定機能) 特定機能
時判断機能。本来、C++17 規格準
拠 C++ 実装、C++17 機能
、残念に現実 C++ 開発
。C++17
対応途中 C++ 将来的機能実装目標
、現時点一部機能実装状態。

、C++11 追加 `rvalue` 機能現実 C++
対応時判定以下。

```
#ifndef __USE_RVALUE_REFERENCES
    #if (__GNUC__ > 4 || __GNUC__ == 4 && __GNUC_MINOR__ >= 3) || \
        _MSC_VER >= 1600
        #if __EDG_VERSION__ > 0
            #define __USE_RVALUE_REFERENCES (__EDG_VERSION__ >= 410)
        #else
            #define __USE_RVALUE_REFERENCES 1
        #endif
    #elif __clang__
        #define __USE_RVALUE_REFERENCES __has_feature(cxx_rvalue_references)
    #else
        #define __USE_RVALUE_REFERENCES 0
    #endif
#endif
```

```
#endif
#endif
```

立現書。GCC MSVC EDG Clang 現使主要 4 C++ 対応 rvalue 実装判定機能。

[illegible]

```
// 文字列の処理関数
void process_string( std::string const & str );

#if __USE_RVALUE_REFERENCES == 1
// 文字列の移動処理の実装関数
// C++ の rvalue の実装の場合
void process_string( std::string && str );
#endif
```

C++17 の、上から立って書く標準機能の用意。C++ 実装特定機能の場合、対応機能の定義。機能の値、機能 C++ 標準採用年月日合 6桁整数表現。

`rvalue` 場合、機能名前 `__cpp_rvalue_references`。 `rvalue` 2006 年 10 月採択、機能値 200610 値。将来 `rvalue` 機能変更機能値変更。値調使 C++ 時代 C++ 標準機能調。

機能を使用、上判定以下書。

```
// 文字列の処理関数
void process_string( std::string const & str );

#ifdef __cpp_rvalue_references
// 文字列の右値参照の処理の実装関数
// C++ 11 の rvalue の実装の場合
```

1.2 `__has_include` 式 : ヘッダーファイルの存在を判定する

```

//
void process_string( std::string && str ) ;
#endif

```

機能 `__has_include` 値は通常 `1` が必要。機能 `__has_include` 存在 `__has_include` 機能の有無を確認 `__has_include`、通常 `#ifdef` を使用。

1.2 `__has_include` 式 : ヘッダーファイルの存在を判定する

`__has_include` 式、`__has_include` 存在 `__has_include` 調 `__has_include` 機能。

```
__has_include( 名前 )
```

`__has_include` 式 `__has_include` 名存在 `1`、存在 `0` 置換。

`__has_include`、C++17 標準 `__has_include` 入。 `__has_include` 名 `<filesystem>`。C++ `__has_include` `__has_include` 調、以下 `__has_include` 書。

```

#if __has_include(<filesystem>)
// __has_include
#include <filesystem>
namespace fs = std::filesystem ;
#else
// 実験的実装使用
#include <experimental/filesystem>
namespace fs = std::experimental::filesystem ;
#endif

```

C++ 実装 `__has_include` `__has_include`、`__has_include` 存在 `__has_include` `#ifdef` 調 `__has_include` 判定。

```

#ifdef __has_include
// __has_include
#else
// __has_include
#endif

```

`__has_include` 式 `#if` `#elif` 中 `__has_include` 使用。

```

int main()
{

```

第 1 章 SD-6 C++ 機能推奨

```

// 例
if ( __has_include(<vector>) )
{ }
}

```

1.3 __has_cpp_attribute 式

C++ 実装の特定属性の有無を調べる、__has_cpp_attribute 式を使う。

```
__has_cpp_attribute( 属性 )
```

__has_cpp_attribute 式、属性が存在する場合属性の標準規格の採択年月を表す数値、存在しない場合 0 を置換する。

```

// [[nodiscard]] 属性の場合
#if __has_cpp_attribute(nodiscard)
[[nodiscard]]
#endif
void * allocate_memory( std::size_t size );

```

__has_include 式と同様、__has_cpp_attribute 式は #if や #elif 中で使われる。#ifdef __has_cpp_attribute 式が存在するかどうか判定する。

第 2 章 C++14 言語新機能

```

int x2 = 1'2'3'4'5'6'7'8'9 ;
int x3 = 1'2345'6789 ;
int x4 = 1'23'456'789 ;

double x5 = 3.14159'26535'89793 ;
}

```

大数値扱、整数 100000000 1000000000 書場合、大数人間目。人間読間違元。数値区切使、100'000'000 1'000'000'000 書。。

他、1 単位見区切。

```

int main()
{
    unsigned int x1 = 0xde'ad'be'ef ;
    unsigned int x2 = 0b11011110'10101101'10111110'11101111 ;
}

```

数値区切、人間読機能、数値影響与。

2.3 `[[deprecated]]` 属性

`[[deprecated]]` 属性名前、使利用推奨状態示使。`[[deprecated]]` 属性指定名前、`typedef` 名、変数、非 `static` 関数、名前空間、`enum`, `enumerator`, 特殊化。

以下指定。

```

// 変数
// 
[[deprecated]] int variable_name1 { } ;
int variable_name2 [[deprecated]] { } ;

// typedef 名
[[deprecated]] typedef int typedef_name1 ;
typedef int typedef_name2 [[deprecated]] ;
using typedef_name3 [[deprecated]] = int ;

```

2.3 `[[deprecated]]` 属性

```
// 関数
// 関数同文法
// 関数
[[deprecated]] void function_name1() { }
void function_name2 [[deprecated]] () { }
```

```
// 変数
// union 同
class [[deprecated]] class_name
{
// 非 static 変数
[[deprecated]] int non_static_data_member_name ;
} ;
```

```
// enum
enum class [[deprecated]] enum_name
{
// enumerator
enumerator_name [[deprecated]] = 42
} ;
```

```
// 名前空間
namespace [[deprecated]] namespace_name { int x ; }
```

```
// 特殊化
```

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

```
template < >
class [[deprecated]] template_name<void> { } ;
```

`[[deprecated]]` 属性は指定名前空間を使用、C++ の警告を出し。

`[[deprecated]]` 属性、文字列付加。C++ 実装警告は含み。

```
[[deprecated("Use of f() is deprecated. Use f(int option) instead.")]]
void f() ;
```

第2章 C++14 言語新機能

```
void f( int option ) ;
```

機能 `__has_cpp_attribute(deprecated)`, 値 201309。

2.4 通常の関数の戻り値の型推定

関数の戻り値の型 `auto` で指定し、戻り値の型 `return` 文で推定する。

```
// int ()
auto a(){ return 0 ; }
// double ()
auto b(){ return 0.0 ; }
```

```
// T(T)
template < typename T >
auto c(T t){ return t ; }
```

`return` 文の型一致する。

```
auto f()
{
    return 0 ; // OK、一致
    return 0.0 ; // OK、一致
}
```

型の決定 `return` 文が存在する場合、関数の戻り値の型参照する。

```
auto a()
{
    &a ; // OK、a の戻り値の型決定
    return 0 ;
}

auto b()
{
    return 0 ;
    &b ; // OK、戻り値の型 int
}
```


2.5 decltype(auto) : 厳格な auto

関数 a の戻り型は、関数 a の型で決定される。return 文の前で型が決定される関数 a の戻り型。関数 b の return 文の現れ後に戻り値の型で決定される。

再帰関数を書く。

```
auto sum( unsigned int i )
{
    if ( i == 0 )
        return i ; // 戻り値の型は unsigned int
    else
        return sum(i-1)+i ; // OK
}
```

戻り型、return 文の順番は逆で戻り値の型で決定されることに注意。

```
auto sum( unsigned int i )
{
    if ( i != 0 )
        return sum(i-1)+i ; // OK
    else
        return i ;
}
```

機能は `__cpp_return_type_deduction`, 値 201304。

2.5 decltype(auto) : 厳格な auto

警告：この項目は C++ 規格の詳細な知識を解説する極難解な項目。平均的な C++ の知識を得るための書籍は読まない。この項目は読む必要はない。

`decltype(auto)` の `auto` 指定子が代用して厳格な `auto`。利用は C++ の規格に厳格に理解を求めたい。

`auto` の `decltype(auto)` の型指定子が呼ばれる文法は一種、型を決定する型。

型を決定する言、具体的型式で決定する機能。

```
// a は int
auto a = 0 ;
// b は int
```


2.5 `decltype(auto)` : 厳格 `auto`

```
f(&i) ;
```

実引数 `&i` の型 `int` が推定された型 `int` と同じ型 `int`。この場合 `int const *` となる。

`auto` の説明。 `decltype(auto)` の説明は簡単。

`decltype(auto)` の型、 `auto` の式を置換した `decltype` の型となる。

```
// int
decltype(auto) a = 0 ;
```

```
// int
decltype(auto) f() { return 0 ; }
```

上の場合、下の場合と同意味。

```
decltype(0) a = 0 ;
decltype(0) f() { return 0 ; }
```

簡単に、`decltype`、`auto` 以降の黒魔術は C++ の規格の知識が必要となる。

`auto` の `decltype(auto)` は一見同じように見える。型が決定する方法は、`auto` の関数の実引数推定を使う、`decltype(auto)` の `decltype` を使う。式の評価の結果の型となる。何が違うのだろうか。

主に関数、`auto` の関数呼出しを使う。関数呼出しの際に暗黙の型変換が行われる。

配列の関数渡り、暗黙の型変換の結果、配列の先頭要素となる。

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

```
int main()
{
    int array[5] ;
    // T int *
    f( array ) ;
}
```

`auto` の `decltype(auto)` は使われる。

```
int array[5] ;
// int *
```

第 2 章 C++14 言語新機能

```

auto x1 = array ;
// 配列、配列初期化
decltype(auto) x2 = array ;

```

、以下同意味。

```

int array[5] ;
// int *
int * x1 = array ;
// 配列、配列初期化
int x2[5] = array ;

```

auto 場合、型 `int *`。配列先頭要素暗黙変換、結果正。

`decltype(auto)` 場合、型 `int [5]`。配列初期化、代入、。

関数型暗黙型変換関数型。

```

void f() ;

// 型 void(*)()
auto x1 = f ;
// 関数型変数
decltype(auto) x2 = f ;

```

auto 修飾子消、`decltype(auto)` 保持。

```

int & f()
{
    static int x ;
    return x ;
}

int main()
{
    // int
    auto x1 = f() ;
    // int &
    decltype(auto) x2 = f() ;
}

```

初期化 `auto` `std::initializer_list`、`decltype(auto)` 式。

例題を挙げる。

```
int main()
{
    // std::initializer_list<int>
    auto x1 = { 1,2,3 } ;
    // decltype({1,2,3})
    decltype(auto) x2 = { 1,2,3 } ;
}
```

`decltype(auto)` は単体で使われることはない。

```
// OK
auto const x1 = 0 ;
// 変数
decltype(auto) const x2 = 0 ;
```

他にも `auto` と `decltype(auto)` は違っている。違っている例を挙げるのは煩雑で省略する、`decltype(auto)` の式型を直接使う。 `auto` の場合も便利に型変換が入る。

`auto` は便利に型変換する場合も暗黙に型変換が入る、意図的に推定される場合もある。

関数、引数、戻り値、関数受取り、戻り値、関数返り関数を書く。以下は書く間違いない。

```
// int ( int & )
auto f( int & ref )
{ return ref ; }
```

戻り値、戻り値の式型が変化する `int` の場合、`decltype(auto)` を使う、

```
// int & ( int & )
decltype(auto) f( int & ref )
{ return ref ; }
```

式型は使われる。

式 `decltype(auto)` を使う場合以下を書く。

```
[]() -> decltype(auto) { return 0 ; } ;
```

`decltype(auto)` は主関数の戻り値の型推定式型を推定する。

第2章 C++14 言語新機能

本章では、C++14 追加機能のうち、C++ の型システムを深く理解する必要のあるものを扱う。

機能のうち `__cpp_decltype_auto`、値 201304。

2.6 ジェネリックラムダ

本章では、ラムダ式で引数の型を書き出す機能について扱う。通常、ラムダ式は以下のように書く。

```
int main()
{
    []( int i, double d, std::string s ) { } ;
}
```

ラムダ式で引数の型が必要。この場合、`int`、`double`、`std::string` の型を `operator ()` の渡す型として指定する必要がある。このように、人間が指定する必要はない。ラムダ式で引数の型を書き出す場所、`auto` を使って型を推定する必要がある。

```
int main()
{
    []( auto i, auto d, auto s ) { } ;
}
```

ラムダ式の結果の型を呼出す際に、異なる型を渡す必要がある。

```
int main()
{
    auto f = []( auto x ) { std::cout << x << '\n' ; } ;

    f( 123 ) ; // int
    f( 12.3 ) ; // double
    f( "hello" ) ; // char const *
}
```

仕組みは簡単で、以下のように `operator ()` を持つ構造体を生成する。

```
struct closure_object
{
```

2.7 初期化ラムダキャプチャー

```
template < typename T >
auto operator () ( T x )
{
    std::cout << x << '\n' ;
}
} ;
```

機能 `__cpp_generic_lambdas`, 値 201304。

2.7 初期化ラムダキャプチャー

初期化ラムダキャプチャーは変数名前式を書き換える機能。

ラムダ式書式が場所で見える変数を使用。

```
int main()
{
    int x = 0 ;
    auto f = [=]{ return x ; } ;
    f() ;
}
```

初期化ラムダキャプチャーは初期化子を書き換える機能。

```
int main()
{
    int x = 0 ;
    [ x = x, y = x, &ref = x, x2 = x * 2 ]
    { // ラムダ式変数使用
        x ;
        y ;
        ref ;
        x2 ;
    } ;
}
```

初期化ラムダキャプチャー、“識別子 = expr”は文法導入子 `[]` 中書く。ラムダ式 “auto 識別子 = expr ;” 書き換える変数作。ラムダ式変数名前変、ラムダ新変数宣言。

第 2 章 C++14 言語新機能

初期化識別子名前 & 付、`constexpr` 変数導入の目的、非 `static` 変数導入の目的。

```
int main()
{
    int x = 0 ;
    [ &ref = x ]()
    {
        ref = 1 ;
    }() ;

    // x 1
}
```

初期化識別子名前追加理由変数名前変数導入の目的、非 `static` 変数導入の目的。

以下問題、。

```
struct X
{
    int data = 42 ;

    auto get_closure_object()
    {
        return [=]{ return data ; } ;
    }
};

int main()
{
    std::function< int() > f ;

    {
        X x ;
        f = x.get_closure_object() ;
    }

    std::cout << f() << std::endl ;
}
```


2.7 初期化

`X::get_closure_object` が `X::data` を返すように修正する。

```
auto get_closure_object()
{
    return [=]{ return data ; } ;
}
```

この見方、`get_closure_object` が `[=]` を使っている、`data` は `static` 変数として内蔵されているように思える。この、この式は `static` 変数として内蔵されている。この式は `this` を使っている。上と下の両方とも意味がある。

```
auto get_closure_object()
{
    return [this]{ return this->data ; } ;
}
```

この、`main` 関数も一度見てみる。

```
int main()
{
    // 関数型オブジェクトを代入する変数
    std::function< int() > f ;

    {
        X x ; // x を構築
        f = x.get_closure_object() ;
        // x を破棄
    }

    // x を破棄
    // return &x->data を破棄して x を参照
    std::cout << f() << std::endl ;
}
```

この、この破棄操作は参照している。この未定義動作。

初期化操作は使っている、非 `static` 変数として内蔵されている。

```
auto get_closure_object()
{
```

第 2 章 C++14 言語新機能

```
    return [data=data]{ return data ; } ;
}
```

、`std::move` 関数存在。特殊化初期化関数実現。

```
auto f()
{
    std::string str ;
    std::cin >> str ;
    // 
    return [str = std::move(str)]{ return str ; } ;
}
```

機能 `__cpp_init_captures`, 値 201304。

2.8 変数テンプレート

変数テンプレート変数宣言宣言機能。

```
template < typename T >
T variable { } ;

int main()
{
    variable<int> = 42 ;
    variable<double> = 1.0 ;
}
```

、順追説明。
C++ 宣言。

```
class X
{
    int member ;
} ;
```

C++ 宣言。型テンプレート型使用。

```
template < typename T >
class X
```

2.8 変数テンプレート

```

{
public :
    T member ;
} ;

int main()
{
    X<int> i ;
    i.member = 42 ; // int

    X<double> d ;
    d.member = 1.0 ; // double
}

```

C++ 関数テンプレート宣言。

```

int f( int x )
{ return x ; }

```

C++ 関数テンプレート宣言。型テンプレート型使用。

```

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

int main()
{
    auto i = f( 42 ) ; // int
    auto d = f( 1.0 ) ; // double
}

```

C++11 typedef 名宣言テンプレート宣言。

```

using type = int ;

```

C++11 テンプレート宣言テンプレート宣言。型テンプレート型使用。

```

template < typename T >
using type = T ;

int main()

```

見。C++ 一部宣言踏以下考。
 C++ 変数宣言。

C++14 变数宣言 宣言。型 型 使。

[illegible]

```
namespace ns {
// 名前空間□□□□□
}
```

変数 $x_1, x_2, x_3, x_4, x_5, x_6$ 使道主 2×2 。

2.8.1 意味は同じだが型が違う定数

`constexpr`変数化`constexpr`良作法`constexpr`。`constexpr`円周率`3.14...` `constexpr`書`constexpr` `pi` `constexpr`変数名`constexpr``constexpr`。`constexpr`、円周率`constexpr`後`constexpr``constexpr`変更`constexpr`。

```
constexpr double pi = 3.1415926535 ;
```

`constexpr`、円周率`constexpr`型`constexpr`複数`constexpr`場合`constexpr`。`constexpr`名前`constexpr`分`constexpr`方法。

```
constexpr float pi_f = 3.1415 ;
constexpr double pi_d = 3.1415926535 ;
constexpr int pi_i = 3 ;
// 任意精度実数表現constexpr
const Real pi_r("3.141592653589793238462643383279") ;
```

`constexpr`、使`constexpr`側`constexpr`型`constexpr`名前`constexpr`変`constexpr`。

```
// 円面積計算関数
template < typename T >
T calc_area( T r )
{
    // T 型constexpr使constexpr名前constexpr変
    return r * r * ??? ;
}
```

関数`constexpr`使`constexpr`手。

```
template < typename T >
constexpr T pi()
{
    return static_cast<T>(3.1415926535) ;
}

template < >
Real pi()
{
    return Real("3.141592653589793238462643383279") ;
}
```

第2章 C++14 言語新機能

```
template < typename T >
T calc_area( T r )
{
    return r * r * pi<T>() ;
}
```

、場合引数何関数呼出 () 必要。
変数以下書。

```
template < typename T >
constexpr T pi = static_cast<T>(3.1415926535) ;

template < >
Real pi<Real>("3.141592653589793238462643383279") ;

template < typename T >
T calc_area( T r )
{
    return r * r * pi<T> ;
}
```

2.8.2 traits のラッパー

値返 traits 値得 ::value 書。

```
std::is_pointer<int>::value ;
std::is_same< int, int >::value ;
```

C++14 中 std::integral_constant 中 constexpr operator bool 追加。
、以下書。

```
std::is_pointer<int>{} ;
std::is_same< int, int >{} ;
```

面倒。変数使 traits 記述楽。

```
template < typename T >
constexpr bool is_pointer_v = std::is_pointer<T>::value ;
template < typename T, typename U >
constexpr bool is_same_v = std::is_same<T, U>::value ;

is_pointer_v<int> ;
```

```
is_same_v< int, int > ;
```

C++ の標準ライブラリから traits 変数 `is_same_v` 版に注意。

機能 `__cpp_variable_templates`, 値 201304。

2.9 constexpr 関数の制限緩和

C++11 は追加 `constexpr` 関数の制限を強。 `constexpr` 関数の本体は実質 `return` 文 1 行に書ける。

C++14 は、何を書けるか。

```
constexpr int f( int x )
{
    // 変数の宣言
    int sum = 0 ;

    // 繰返文を書く
    for ( int i = 1 ; i < x ; ++i )
    {
        // 変数の変更
        sum += i ;
    }

    return sum ;
}
```

機能 `__cpp_constexpr`, 値 201304。

C++11 は `constexpr` 関数に対応 `C++14` は `constexpr` 関数に対応 `C++` 実装、 `__cpp_constexpr` の値 200704 。

2.10 メンバー初期化子とアグリゲート初期化の組み合わせ

C++14 は初期化子と初期化の組み合わせ。

初期化子と非 `static` の初期化 = `C++11` の機能。

```
struct S
```

第 2 章 C++14 言語新機能

```
{
    // 初期化子
    int data = 123 ;
};
```

初期化条件満ち型初期化初期化
C++11 機能。

```
struct S
{
    int x, y, z ;
};

S s = { 1,2,3 } ;
// s.x == 1, s.y == 2, s.z == 3
```

C++11 初期化子持型条件満
初期化。

C++14 、制限緩和。

```
struct S
{
    int x, y=1, z ;
};

S s1 = { 1 } ;
// s1.x == 1, s1.y == 1, s1.z == 0

S s2{ 1,2,3 } ;
// s2.x == 1, s2.y == 2, s2.z == 3
```

初期化、初期化子持非 `static` 対応
値場合初期化優先。省略場合初期
化子初期化。初期化初期化子明示の初期
化非 `static` 空初期化初期化場合
同。

機能 `__cpp_aggregate_nsdmi`, 値 201304。

2.11 サイズ付き解放関数

C++14 の `operator delete`、解放、取得追加。

```
void operator delete    ( void *, std::size_t ) noexcept ;
void operator delete[] ( void *, std::size_t ) noexcept ;
```

第二引数 `std::size_t` 型、第一引数指定指解放与。

以下使用。

```
void * operator new ( std::size_t size )
{
    void * ptr = std::malloc( size ) ;

    if ( ptr == nullptr )
        throw std::bad_alloc() ;

    std::cout << "allocated storage of size: " << size << '\n' ;
    return ptr ;
}

void operator delete ( void * ptr, std::size_t size ) noexcept
{
    std::cout << "deallocated storage of size: " << size << '\n' ;
    std::free( ptr ) ;
}

int main()
{
    auto u1 = std::make_unique<int>(0) ;
    auto u2 = std::make_unique<double>(0.0) ;
}
```

機能 `__cpp_sized_deallocation`, 値 201309。



第3章

C++17 のコア言語の新機能

C++14 の新機能の最終版、および C++17 のコア言語の新機能解説。

C++17 のコア言語の新機能、C++11 の大規模な変更。

3.1 トライグラフの廃止

C++17 のトライグラフの廃止。

トライグラフの廃止は、読者の変更が必要。トライグラフの廃止は、読者の変更が必要。

3.2 16 進数浮動小数点数リテラル

C++17 の浮動小数点数リテラルの 16 進数使用。

16 進数浮動小数点数リテラル、0x 続く仮数部 16 進数 (0123456789abcdefABCDEF) の書、p 続く指数部 10 進数の書。

```
double d1 = 0x1p0 ; // 1
double d2 = 0x1.0p0 ; // 1
double d3 = 0x10p0 ; // 16
double d4 = 0xabcp0 ; // 2748
```

指数部 e または p を使用。

```
double d1 = 0x1p0 ;
double d2 = 0x1P0 ;
```

16 進数浮動小数点数リテラル、指数部省略。

```
int a = 0x1 ; // 整数
```

第3章 C++17 言語新機能

```
0x1.0 ; // 指数部、指数部
```

指数部 10 進数記述。16 進数浮動小数点数仮数部 2 指数部乗掛値。、

```
0xNpM
```

浮動小数点数値

$$N \times 2^M$$

。

```
0x1p0 ; // 1
0x1p1 ; // 2
0x1p2 ; // 4
0x10p0 ; // 16
0x10p1 ; // 32
0x1p-1 ; // 0.5
0x1p-2 ; // 0.25
```

16 進数浮動小数点数浮動小数点数記述。

```
auto a = 0x1p0f ; // float
auto b = 0x1p0l ; // long double
```

16 進数浮動小数点数、浮動小数点数表現方法詳細知環境 (IEEE-754)、正確浮動小数点数表現記述。機能 `__cpp_hex_float`, 値 201603。

3.3 UTF-8 文字リテラル

C++17 UTF-8 文字追加。

```
char c = u8'a' ;
```

UTF-8 文字文字 `u8` 付。UTF-8 文字 UTF-8 単位 1 表現文字扱。UCS 規格、C0 制御文字基本文字 Unicode 該当。UTF-8 文字書文字複数 UTF-8 単位必要場合。

```
//
```

3.4 関数型例外指定

```
// U+3042 UTF-8 0xE3, 0x81, 0x82 3 バイト単位表現必要
// 
u8' ' ;
```

機能。

3.4 関数型としての例外指定

C++17 例外指定関数型組込。

例外指定 `noexcept`。 `noexcept` `noexcept(true)` 指定関数例外外投。

C++14 例外指定型入。、無例外指定付関数型型無例外保証。

```
// C++14 
void f()
{
    throw 0 ;
}

int main()
{
    // 無例外指定付関数型
    void (*p)() noexcept = &f ;

    // 無例外指定関数型例外投
    p() ;
}
```

C++17 例外指定型組込。例外指定関数型例外指定関数型変換。逆。

```
// 型 void()
void f() { }
// 型 void() noexcept
void g() noexcept { }

// OK
// p1, &f 例外指定関数型
void (*p1)() = &f ;
```

第3章 C++17 言語新機能

```
// OK
// 例外指定関数型&g 例外指定関数型p2
// 変換
void (*p2)() = &g ; // OK

// 
// 例外指定関数型&f 例外指定関数型p3
// 変換
void (*p3)() noexcept = &f ;

// OK
// p4, &g 例外指定関数型
void (*p4)() noexcept = &g ;
```

機能 `__cpp_noexcept_function_type`, 値 201510。

3.5 fold 式

C++17 fold 式 入。fold 元 数学 概念 畳込 呼。

C++ fold 式 中身 二項演算子 適用 式。

今、可変長 使 受 取 値 加算 合計 返 関数 `sum` 書。

```
template < typename T, typename ... Types >
auto sum( T x, Types ... args ) ;

int main()
{
    int result = sum(1,2,3,4,5,6,7,8,9) ; // 45
}
```

関数 `sum` 以下 実装。

```
template < typename T >
auto sum( T x )
{
    return x ;
}
```

```
template < typename T, typename ... Types >
auto sum( T x, Types ... args )
{
    return x + sum( args... ) ;
}
```

`sum(x, args)` は 1 番目の引数 `x` と、残りの引数 `args` を受取る。つまり、`x + sum(args ...)` を返す。つまり、`sum(args ...)` は `sum(x, args)` が渡す 1 番目の引数、つまり最初に見る 2 番目の引数 `x` が入る、つまり `sum` を呼ぶ。つまり再帰的に処理を繰り返す。

つまり、引数 1 は `sum` を呼ぶ。つまり重要。つまり可変長引数は 0 個の引数を取る、つまり可変長引数の版 `sum` を呼ぶ、つまり次 `sum` を呼ぶ、つまり回避、つまり再帰終了条件、引数 1 は `sum` の関数を書く。

可変長引数は任意個の引数に対応、つまり再帰的に必須。

つまり、つまり実現 `N` 個の引数 `args` の中身を対し、仮に `N` 番目の `args#N` を表記して、`args#0 + args#1 + ... + args#N-1` を展開する。C++17 の fold 式は二項演算子に適用して展開する機能。

fold 式は `sum` の以下のように書く。

```
template < typename ... Types >
auto sum( Types ... args )
{
    return ( ... + args ) ;
}
```

`(... + args)` は、`args#0 + args#1 + ... + args#N-1` を展開する。

fold 式は、単項 fold 式、二項 fold 式。つまり、演算子の結合順序は左 fold、右 fold。

fold 式は必ず括弧で囲む。

```
template < typename ... Types >
auto sum( Types ... args )
{
    // fold 式
    ( ... + args ) ;
}
```

第3章 C++17 新語法新機能

```
// 括弧、括弧
... + args ;
}
```

單項 fold 式文法以下。

```
單項右fold
( cast-expression fold-operator ... )
單項左fold
( ... fold-operator cast-expression )
```

例：

```
template < typename ... Types >
void f( Types ... args )
{
    // 單項左 fold
    ( ... + args ) ;
    // 單項右 fold
    ( args + ... ) ;
}
```

cast-expression 未展開入。

例：

```
template < typename T >
T f( T x ) { return x ; }

template < typename ... Types >
auto g( Types ... args )
{
    // f(args#0) + f(args#1) + ... + f(args#N-1)
    return ( ... + f(args) ) ;
}
```

f(args) 展開。

fold-operator 以下二項演算子使。

```
+ - * / % ^ & | << >>
+= -= *= /= %= ^= &= |= <<= >>=
== != < > <= >= && || , .* ->*
```


fold 式は左 fold と右 fold の 2 種類がある。

左 fold 式 `(... op pack)` は、展開結果 `(((pack#0 op pack#1) op pack#2) ... op pack#N-1)` となる。右 fold 式 `(pack op ...)` は、展開結果 `(pack#0 op (pack#1 op (pack#2 op (... op pack#N-1))))` となる。

```
template < typename ... Types >
void sum( Types ... args )
{
    // 左 fold
    // (((1+2)+3)+4)+5)
    auto left = ( ... + args ) ;
    // 右 fold
    // (1+(2+(3+(4+5))))
    auto right = ( args + ... ) ;
}

int main()
{
    sum(1,2,3,4,5) ;
}
```

浮動小数点数は交換法則を満たす型に fold 式を適用する際、注意が必要。

二項 fold 式の文法は以下の通り。

```
( cast-expression fold-operator ... fold-operator cast-expression )
```

左右の `cast-expression` は、未展開の式として代入される。2 つの `fold-operator` は、同演算子となる。

`(e1 op1 ... op2 e2)` は、二項 fold 式、`e1` は右 fold 式の場合、`e2` は左 fold 式の場合。

```
template < typename ... Types >
void sum( Types ... args )
{
    // 左 fold
    // (((((0+1)+2)+3)+4)+5)
    auto left = ( 0 + ... + args ) ;
    // 右 fold
    // (1+(2+(3+(4+(5+0))))))
}
```


10

3.6 `lambda式 *this`

```

struct X
{
    int data = 0 ;
    void f()
    {
        // this
        // data
        [this]{ data = 1 ; }() ;

        // this->data

        // 、*this
        // 変数
        // 変更
        [*this]{ data = 2 ; } () ;

        // OK、mutable 使

        [*this]() mutable { data = 2 ; } () ;

        // this->data
        // 変更内*this
    }
};

```

最初`lambda式`生成以下。

```

class closure_object
{
    X * this_ptr ;

public :
    closure_object( X * this_ptr )
        : this_ptr(this_ptr) { }

    void operator () () const
    {
        this_ptr->data = 1 ;
    }
};

```

2 番目`lambda式`以下生成。

第3章 C++17 言語新機能

```

class closure_object
{
    X this_obj ;
    X const * this_ptr = &this_obj ;

public :
    closure_object( X const & this_obj )
        : this_obj(this_obj) { }

    void operator () () const
    {
        this_ptr->data = 2 ;
    }
};

```

この C++ 文法は、`mutable` 付の静的な変数値を変更する。3 番目の式は以下に示すように生成される。

```

class closure_object
{
    X this_obj ;
    X * this_ptr = &this_obj ;

public :
    closure_object( X const & this_obj )
        : this_obj(this_obj) { }

    void operator () ()
    {
        this_ptr->data = 2 ;
    }
};

```

この式は `mutable` 付の静的な変数値を変更する。
`*this` の場合、`this` のように生成される。

```

struct X
{
    int data = 42 ;
    void f()

```

```

    {
        // this 関数 f を呼出す
        std::printf("%p\n", this) ;

        // this 別関数
        [*this]() { std::printf("%p\n", this) ; }() ;
    }
};

int main()
{
    X x ;
    x.f() ;
}

```

この場合、出力は 2 行の値が異なる。

ラムダ式は `*this` 名前空間 `*this` 関数を提供する提案。同等の機能を初期化関数可能、表記冗長間違い元。

```

struct X
{
    int data ;

    auto f()
    {
        return [ tmp = *this ] { return tmp.data ; } ;
    }
};

```

機能 `__cpp_capture_star_this`, 値 201603。

3.7 constexpr ラムダ式

C++17 ラムダ式 `constexpr` 関数。正確説明、ラムダ式 `operator ()` 条件満ち場合 `constexpr` 関数。

```

int main()
{
    auto f = [] { return 42 ; } ;
}

```

第3章 C++17 言語の新機能

```
constexpr int value = f() ; // OK
}
```

constexpr 条件を満たす式の時定数必要場所使用。 変数配列添字 static_assert 。

```
int main()
{
    auto f = []{ return 42 ; } ;

    int a[f()] ;
    static_assert( f() == 42 ) ;
    std::array<int, f()> b ;
}
```

constexpr 条件を満たす、。

```
int main()
{
    int a = 0 ; // 実行時値
    constexpr int b = 0 ; // 時定数

    auto f = [=]{ return a ; } ;
    auto g = [=]{ return b ; } ;

    // 、constexpr 条件を満たす
    constexpr int c = f() ;

    // OK、constexpr 条件を満たす
    constexpr int d = g() ;
}
```

以下内容上級者向け解説、通常読者理解必要。

constexpr 式 SFINAE 文脈使用。

```
//
template < typename T,
    bool b = []{
        T t ;
        t.func() ;
        return true ;
    }() >
```

```
void f()
{
    T t ;
    t.func() ;
}
```

この関数、この許可は、この関数の仮引数に対して任意の式で文の
Substitution が失敗した場合にのみ適用される。

上級者向け解説終了。

機能は、__cpp_constexpr, 値 201603。

__cpp_constexpr の値、C++11 の時点 200704、C++14 の時点 201304。

3.8 文字列なし static_assert

C++17 の static_assert は文字列なしの取組を追加。

```
static_assert( true ) ;
```

C++11 の追加の static_assert は、文字列なしの必須。

```
static_assert( true, "this shall not be asserted." ) ;
```

特に文字列を指定する必要はない場合、文字列なしの取組は static_assert に追加。

機能は、__cpp_static_assert, 値 201411。

C++11 の時点 __cpp_static_assert の値 200410。

3.9 ネストされた名前空間定義

C++17 は、名前空間の定義を楽に書く。

名前空間 A、A::B::C は、名前空間 A 中の名前空間 B に入っている名前空間 C。

```
namespace A {
    namespace B {
        namespace C {
            // ...
        }
    }
}
```

第 3 章 C++17 言語新機能

```

}
```

C++17 言語、上記と同様に以下に書ける。

```

namespace A::B::C {
// ...
}
```

機能は `__cpp_nested_namespace_definitions`, 値 201411。

3.10 `[[fallthrough]]` 属性

`[[fallthrough]]` 属性は `switch` 文の中で `case` ごとに突抜けることを示し、`break` を省略できる。

`switch` 文に対応する `case` ごとに処理を移す。通常、以下に書ける。

```

void f( int x )
{
    switch ( x )
    {
        case 0 :
            // 処理 0
            break ;
        case 1 :
            // 処理 1
            break ;
        case 2 :
            // 処理 2
            break ;
        default :
            // x が他の値の場合に処理
            break ;
    }
}
```

例として以下に書ける。

```

case 1 :
    // 処理 1
case 2 :
    // 処理 2
    break ;
```


`x` 1 処理 1 実行後、処理 2 実行。switch 文書誤り、誤り書、賢 C++ 処理 switch 文 case break 文 return 文処理終、次 case default 処理突抜、発見、警告出。

、意図突抜処理場合、警告誤り警告抑制、中処理突抜意図記述、`[[fallthrough]]` 属性追加。

```
case 1 :
    // 処理 1
    [[fallthrough]]
case 2 :
    // 処理 2
    break ;
```

`[[fallthrough]]` 属性書、C++ 処理先突抜、誤り警告抑制。、他人読意図明。

機能 `__has_cpp_attribute(fallthrough)`, 値 201603。

3.11 `[[nodiscard]]` 属性

`[[nodiscard]]` 属性関数戻り値無視使用。 `[[nodiscard]]` 属性付与関数戻り値無視警告表示。

```
[[nodiscard]] int f()
{
    return 0 ;
}

void g( int ) { }
```

```
int main()
{
    // 、戻り値無視
    f() ;
```


--	--

--	--

3.12 `[[maybe_unused]]` 属性

```
// 関数を確認
do_something_that_may_fail() ;

// 関数の前提条件を処理
do_something_on_no_error() ;
}
```

関数 `[[nodiscard]]` 属性は付与された関数、関数側側で初步の確認を確認欠如警告が出力される。

`[[nodiscard]]` 属性、`enum` 型は付与された。

```
class [[nodiscard]] X { } ;
enum class [[nodiscard]] Y { } ;
```

`[[nodiscard]]` は付与された `enum` 型戻り値型関数 `[[nodiscard]]` は付与された。

```
class [[nodiscard]] X { } ;
```

```
X f() { return X{} ; }
```

```
int main()
{
    // 警告、戻り値無視
    f() ;
}
```

機能 `__has_cpp_attribute(nodiscard)`, 値 201603。

3.12 `[[maybe_unused]]` 属性

`[[maybe_unused]]` 属性は名前空間の意図的使用を示す。

現実 C++ の宣言、宣言の考慮は、`[[maybe_unused]]` 属性は名前空間の存在。

```
void do_something( int *, int * ) ;
```

```
void f()
{
    int x[5] ;
```

第3章 C++17 新言語新機能

```

char reserved[1024] = { } ;
int y[5] ;

do_something( x, y ) ;
}

```

このコードで `reserved` という名前が予約語として使われている。一見これは不必要な名前が見える。優秀な C++ のコードは予約語として使われている名前に対して「予約語として使われている」ことを警告メッセージで出す。

この、予約語として使われていることを警告メッセージで出すのは、このコードの理由で `reserved` という名前が一見使われている変数が必要である。

この、`reserved` という名前が破壊的検出領域に属している。この C++ 以外に言語書に予約語として使われている、この使われている。この OS の外部に読書されることを確保している。

この理由で、名前が予約語として一見使われていることが存在しないことを意味する、`[[maybe_unused]]` 属性を使用して。この、C++ の「未使用名前」の警告を抑制する。

```
[[maybe_unused]] char reserved[1024] ;
```

`[[maybe_unused]]` 属性は適用される名前、宣言、`typedef` 名、変数、非 `static` の関数、`enum`, `enumerator` の。

```

// 例
class [[maybe_unused]] class_name
{
// 非 static の関数
[[maybe_unused]] int non_static_data_member ;

} ;

// typedef 名
// 例
[[maybe_unused]] typedef int typedef_name1 ;
typedef int typedef_name2 [[maybe_unused]] ;

// 宣言の typedef 名
using typedef_name3 [[maybe_unused]] = int ;

```

```
// 変数
// [[maybe_unused]]
[[maybe_unused]] int variable_name1{};
int variable_name2 [[maybe_unused]] { } ;

// 関数
// [[maybe_unused]]関数同文法
// [[maybe_unused]]
[[maybe_unused]] void function_name1() { }
void function_name2 [[maybe_unused]] () { }

enum [[maybe_unused]] enum_name
{
// enumerator
    enumerator_name [[maybe_unused]] = 0
};
```

機能`__has_cpp_attribute(maybe_unused)`, 値 201603

3.13 演算子のオペランドの評価順序の固定

C++17 演算子`[[maybe_unused]]`評価順序固定。

以下式`a`、`b` 順番評価規格上保証。`@=` `@` 文法上許任意演算子入 (`+=`, `-=`)。

```
a.b
a->b
a->*b
a(b1,b2,b3)
b = a
b @= a
a[b]
a << b
a >> b
```

例、

```
int* f() ;
int g() ;
```

第3章 C++17 言語新機能

```
int main()
{
    f()[g()] ;
}
```

この場合、関数 `f` が先に呼ばれ、次に関数 `g` が呼ばれることを保証する。

関数が呼ばれる実引数の `b1`, `b2`, `b3` の評価順序は未規定である。

また、既存の未定義動作も挙動が定められている。

3.14 constexpr if 文：コンパイル時条件分岐

`constexpr if` 文はコンパイル時条件分岐機能である。

`constexpr if` 文、通常 `if` 文 `if constexpr` を置換する。

```
// if 文
if ( expression )
    statement ;

// constexpr if 文
if constexpr ( expression )
    statement ;
```

`constexpr if` 文は名前空間、実際の記述は `if constexpr` である。

コンパイル時条件分岐は何の意味がある。以下 `constexpr if` 文の挙動を一覧する。

- 最適化
- 非 `constexpr` 挙動の変化

コンパイル時条件分岐機能の理解、C++ の既存条件分岐の理解が必要である。

3.14.1 実行時の条件分岐

通常実行時条件分岐、実行時値取得、実行時条件分岐を行う。

```
void f( bool runtime_value )
{
    if ( runtime_value )
        do_true_thing() ;
}
```

```

    else
        do_false_thing() ;
}

```

この場合、runtime_value が true の場合関数 do_true_thing を呼び、false の場合関数 do_false_thing を呼び。

実行時条件分岐条件、定数定数指定。

```

if ( true )
    do_true_thing() ;
else
    do_false_thing() ;

```

この場合、賢いコンパイラ以下で処理最適化が行われる。

```
do_true_thing() ;
```

この場合、条件が常に true の場合、コンパイラ最適化で実行時条件分岐が行われず、定数定数指定条件分岐最適化が行われる。

一度例に戻す。今度完全に見る。

```

// do_true_thing の宣言
void do_true_thing() ;

// do_false_thing の宣言が存在する

void f( bool runtime_value )
{
    if ( true )
        do_true_thing() ;
    else
        do_false_thing() ; // 呼び
}

```

この場合、do_false_thing の名前が宣言されていない。C++ の定数条件分岐以下で最適化が行われる。

```

void do_true_thing() ;

void f( bool runtime_value )

```

第3章 C++17 言語新機能

```

{
    do_true_thing() ;
}

```

最適化結果失、依然時検証。検証、誤。名前 `do_false_thing` 宣言存在。

3.14.2 プリプロセス時の条件分岐

C++ C 言語受継 C 時条件分岐機能。

```

// do_true_thing 宣言
void do_true_thing() ;

// do_false_thing 宣言存在

void f( bool runtime_value )
{
    #if true
        do_true_thing() ;
    #else
        do_false_thing() ;
    #endif
}

```

、結果、以下変換。

```

void do_true_thing() ;

void f( bool runtime_value )
{
    do_true_thing() ;
}

```

結果、時条件分岐、選択分岐、書。

時条件分岐、条件整数 `bool` 型、比較演算子適用結果。、時計算。

3.14 constexpr if 文 : コンパイル時条件分岐

```
constexpr int f()
{
    return 1 ;
}

void do_true_thing() ;

int main()
{
    // 例
    // 名前 f コンパイル時条件分岐
    #if f()
        do_true_thing() ;
    #else
        do_false_thing() ;
    #endif
}
```

3.14.3 コンパイル時の条件分岐

コンパイル時条件分岐、分岐条件コンパイル時計算結果使用、選択コンパイル分岐コンパイル時含、使用コンパイルコンパイル時条件分岐。

例、std::distance 標準 実装 実装。std::distance(first, last) 例、コンパイル first last 距離返。

```
template < typename Iterator >
constexpr typename std::iterator_traits<Iterator>::difference_type
distance( Iterator first, Iterator last )
{
    return last - first ;
}
```

残念、実装 Iterator コンパイル時場合動。入力コンパイル時対応、コンパイル 1 コンパイル last 等コンパイル比較実装必要。

```
template < typename Iterator >
constexpr typename std::iterator_traits<Iterator>::difference_type
distance( Iterator first, Iterator last )
{
```

第3章 C++17 言語新機能

```

    typename std::iterator_traits<Iterator>::difference_type n = 0 ;

    while ( first != last )
    {
        ++n ;
        ++first ;
    }

    return n ;
}

```

残念、実装 `Iterator` の渡り効率が悪く、

必要実装、`Iterator` の `last - first` を使、地道に遅実装を使。 `Iterator`、以下 `is_random_access_iterator<iterator>` を確認。

```

template < typename Iterator >
constexpr bool is_random_access_iterator =
    std::is_same_v<
        typename std::iterator_traits<
            std::decay_t<Iterator>
        >::iterator_category,
        std::random_access_iterator_tag > ;

```

、`distance` 以下に書。

```

// 判定
template < typename Iterator >
constexpr bool is_random_access_iterator =
    std::is_same_v<
        typename std::iterator_traits<
            std::decay_t<Iterator>
        >::iterator_category,
        std::random_access_iterator_tag > ;

// distance
template < typename Iterator >
constexpr typename std::iterator_traits<Iterator>::difference_type
distance( Iterator first, Iterator last )

```

```

{
    // 乱数アクセス確認
    if ( is_random_access_iterator<Iterator> )
    { // 乱数アクセス速方法使用
        return last - first ;
    }
    else
    { // 乱数アクセス遅方法使用
        typename std::iterator_traits<Iterator>::difference_type n = 0 ;

        while ( first != last )
        {
            ++n ;
            ++first ;
        }

        return n ;
    }
}

```

残念、乱数アクセス動。乱数アクセス速方法使用、last - first 乱数アクセス速方法使用、乱数アクセス遅方法使用。乱数アクセス、

```

if ( is_random_access_iterator<Iterator> )

```

部分、is_random_access_iterator<Iterator> 値の時計算、最終の結果、if (true) if (false) 判断。乱数アクセス速方法使用、乱数アクセス遅方法使用。乱数アクセス、乱数アクセス。

constexpr if 使用、乱数アクセス部分分岐乱数アクセス速方法使用乱数アクセス遅方法使用。

```

// distance
template < typename Iterator >
constexpr typename std::iterator_traits<Iterator>::difference_type
distance( Iterator first, Iterator last )
{
    // 乱数アクセス確認
    if constexpr ( is_random_access_iterator<Iterator> )
    { // 乱数アクセス速方法使用

```

第3章 C++17 言語新機能

```

        return last - first ;
    }
    else
    { // 遅延方法を使う
        typename std::iterator_traits<Iterator>::difference_type n = 0 ;

        while ( first != last )
        {
            ++n ;
            ++first ;
        }

        return n ;
    }
}

```

3.14.4 超上級者向け解説

`constexpr if` 文、実体化時条件分岐文。実体化時、選択実体化抑制行機能。

`constexpr if` 文選択文 `discarded statement` 文。
`discarded statement` 実体化際実体化。

```

struct X
{
    int get() { return 0 ; }
};

template < typename T >
int f(T x)
{
    if constexpr ( std::is_same_v< std::decay_t<T>, X > )
        return x.get() ;
    else
        return x ;
}

int main()
{
    X x ;
}

```

3.14 constexpr if 文 : 条件分岐

```

    f( x ) ; // return x.get()
    f( 0 ) ; // return x
}

```

f(x) の、return x の discarded statement は実体化される。x の int 型の暗黙の変換の問題がある。f(0) の return x.get() の discarded statement は実体化されない。int 型の関数 get の問題がある。

discarded statement は実体化されない、コンパイル時に削除される。discarded statement は文法の、意味の正しい場合、コンパイル時に削除される。

```

template < typename T >
void f( T x )
{
    // 名前空間 g の宣言
    if constexpr ( false )
        g() ;

    // 文法違反
    if constexpr ( false )
        !#$%^&*()_+ ;
}

```

何の説明がある、constexpr if は実体化条件付き抑制がある。条件付き実体化。

```

template < typename T >
void f()
{
    if constexpr ( std::is_same_v<T, int> )
    {
        // 常関数
        static_assert( false ) ;
    }
}

```

常関数、static_assert(false) は依存宣言、関数宣言の解釈、依存名、関数解釈。

最初 static_assert は式書

第3章 C++17 言語新機能

例。

```
template < typename T >
void f()
{
    static_assert( std::is_same_v<T, int> );

    if constexpr ( std::is_same_v<T, int> )
    {
    }
}
```

例、例の constexpr 文の条件が合致しない static_assert を使用する場合。例、constexpr if の条件が合致しない、その内容が全部 static_assert を書く冗長な場合。

```
template < typename T >
void f()
{
    if constexpr ( E1 )
        if constexpr ( E2 )
            if constexpr ( E3 )
            {
                // E1 && E2 && E3 の条件が合致しない
                // 実際は常に条件が合致しない
                static_assert( false );
            }
}
```

現実例、E1, E2, E3 の複雑な式、static_assert(E1 && E2 && E3) を書くのが冗長。同内容が二度書ける間違った元。

例の場合、static_assert の条件が引数に依存する例、例、constexpr if の条件が合致しない発動する static_assert を書く。

```
template < typename ... >
bool false_v = false ;

template < typename T >
void f()
{
    if constexpr ( E1 )
        if constexpr ( E2 )
```

3.14 constexpr if 文 : 定数時条件分岐

```

    if constexpr ( E3 )
    {
        static_assert( false_v<T> ) ;
    }
}

```

`false_v` を使った、`static_assert` の引数 `T` が依存する。結果、`static_assert` が発動し、実体化が遅延する。

`constexpr if` は非依存な書式、場合普通 `if` 文と同様。

3.14.5 constexpr if では解決できない問題

`constexpr if` は条件付き実体化の抑制、最初の問題を解決に使えない。以下に問題がある。

```

// do_true_thing を宣言
void do_true_thing() ;

// do_false_thing を宣言し存在を知らせる

void f( bool runtime_value )
{
    if constexpr ( true )
        do_true_thing() ;
    else
        do_false_thing() ; // 問題
}

```

理由、名前 `do_false_thing` は非依存名で宣言時に解決されない。

3.14.6 constexpr if で解決できる問題

`constexpr if` は依存名に関与する場合、実体化の抑制が可能。

例、特定型に対する特別操作の場合

```

struct X
{
    int get_value() ;
}

```

第3章 C++17 型推論言語の新機能

```

    } ;

    template < typename T >
    void f(T t)
    {

        int value{} ;

        // T 型 X 型 特別処理行
        if constexpr ( std::is_same<T, X>{} )
        {
            value = t.get_value() ;
        }
        else
        {
            value = static_cast<int>(t) ;
        }
    }

```

この `constexpr if` 文、`T 型 X 型` の `t.get_value()` 式
 実体化、評価。

再帰の特殊化

```

// factorial<N> N 階乗返
template < std::size_t I >
constexpr std::size_t factorial()
{
    if constexpr ( I == 1 )
    { return 1 ; }
    else
    { return I * factorial<I-1>() ; }
}

```

この `constexpr if` 文、`factorial<N-1>` 永遠実体化時
 停止。

機能 `__cpp_if_constexpr`, 値 201606。

3.15 初期化文付き条件文

C++17 型推論、条件文の初期化文記述。


```

if ( int x = 1 ; x )
    /*...*/ ;

switch( int x = 1 ; x )
{
    case 1 :
        /*... */;
}

```

例、以下例と同意味例。

```

{
    int x = 1 ;
    if ( x ) ;
}

{
    int x = 1 ;
    switch( x )
    {
        case 1 : ;
    }
}

```

機能追加例、変数宣言、if 文条件変数使、if 文実行後変数使用例、例現実例頻出例。

```

void * ptr = std::malloc(10) ;
if ( ptr != nullptr )
{
    // 処理
    std::free(ptr) ;
}
// 以降 ptr 使

FILE * file = std::fopen("text.txt", "r") ;
if ( file != nullptr )
{
    // 処理
    std::fclose( file ) ;
}

```

第3章 C++17 言語新機能

```
// 以降 file を使用

auto int_ptr = std::make_unique<int>(42) ;
if ( ptr )
{
    // 処理
}
// 以降 int_ptr を使用
```

上記の問題。以降変数を使用、変数自体を使用。

```
auto ptr = std::make_unique<int>(42) ;
if ( ptr )
{
    // 処理
}
// 以降 ptr を使用
```

```
// 使用
int value = *ptr ;
```

変数を使用、開閉、変数外。

```
{
    auto int_ptr = std::make_unique<int>(42) ;
    if ( ptr )
    {
        // 処理
    }
    // ptr を破棄
}
// 以降 ptr を使用
```

頻出、初期化文付条件文追加。

```
if ( auto ptr = std::make_unique<int>(42) ; ptr )
{
    // 処理
}
```

3.16 クラステンプレートのコンストラクターからの実引数推定

3.16 クラステンプレートのコンストラクターからの実引数推定

C++17 クラステンプレートのコンストラクターからの実引数推定は、関数の実引数推定と同様に機能する。

```
template < typename T >
struct X
{
    X( T t ) { }
};

int main()
{
    X x1(0) ; // X<int>
    X x2(0.0) ; // X<double>
    X x3("hello") ; // X<char const *>
}
```

関数と同様に、クラステンプレートのコンストラクターからの実引数推定は、関数の実引数推定と同様に機能する。

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

int main()
{
    f( 0 ) ; // f<int>
    f( 0.0 ) ; // f<double>
    f( "hello" ) ; // f<char const *>
}
```

3.16.1 推定ガイド

クラステンプレートのコンストラクターからの実引数推定は、関数の実引数推定と同様に機能する。ただし、コンストラクターからの実引数推定は、関数の実引数推定と一致しない場合がある。この場合、コンストラクターからの実引数推定は、関数の実引数推定と一致しない場合がある。

```
// クラステンプレート風
template < typename T >
class Container
```

第3章 C++17 型推定言語新機能

```

{
    std::vector<T> c ;
public :
    // 初期化型推定型を取
    // Iterator T
    // T 推定
    template < typename Iterator >
    Container( Iterator first, Iterator last )
        : c( first, last )
    { }
} ;

int main()
{
    int a[] = { 1,2,3,4,5 } ;

    //
    // T 推定
    Container c( std::begin(a), std::end(a) ) ;
}

```

型推定、C++17 型推定機能を提供。

型推定名(引数) -> id ;

使用、以下書籍。

```

template < typename Iterator >
Container( Iterator, Iterator )
-> Container< typename std::iterator_traits< Iterator >::value_type > ;

```

C++ 型推定使用、Container<T>::Container(Iterator, Iterator)、T std::iterator_traits<Iterator>::value_type 推定判断。

型推定、初期化対応以下書籍。

```

template < typename T >
class Container
{
    std::vector<T> c ;
public :

```

3.17 autoによる非型テンプレートパラメーターの宣言

```

        Container( std::initializer_list<T> init )
            : c( init )
        { }
    } ;

template < typename T >
Container( std::initializer_list<T> ) -> Container<T> ;

int main()
{
    Container c = { 1,2,3,4,5 } ;
}

```

C++ の型推定、Container<T>::Container(std::initializer_list<T>)の場合 T の型推定が行われる。機能 `__cpp_deduction_guides`, 値 201606。

3.17 autoによる非型テンプレートパラメーターの宣言

C++17 の非型テンプレートパラメーターの宣言に auto を使用することができる。

```

template < auto x >
struct X { } ;

void f() { }

int main()
{
    X<0> x1 ;
    X<01> x2 ;
    X<&f> x3 ;
}

```

C++14 のように、以下のように書くこともできる。

```

template < typename T, T x >
struct X { } ;

```

第3章 C++17 言語の新機能

```

void f() { }

int main()
{
    X<int, 0> x1 ;
    X<long, 01> x2 ;
    X<void(*)(), &f> x3 ;
}

```

機能 `__cpp_template_auto`, 値 201606。

3.18 using 属性名前空間

C++17 での、属性名前空間 `using` 記述。

```

// [[extension::foo, extension::bar]] 同
[[ using extension : foo, bar ]] int x ;

```

属性、属性名前空間付、独自拡張属性名前衝突回避。

、C++ 独自拡張 `foo, bar` 属性、別 C++ 同独自拡張 `foo, bar` 属性保持、意味違場合、意味違。

```
[[ foo, bar ]] int x ;
```

、C++ 属性名前空間文法用意。注意深く C++ 独自拡張属性属性名前空間設定。

```
[[ extension::foo, extension::bar ]] int x ;
```

問題、記述面倒。

C++17 での、`using` 属性名前空間機能、`using` 属性名前空間省略可能。文法 `using` 似、属性中 `using name : ...` 書、続属性、属性名前空間 `name` 付同効果得。

3.19 非標準属性の無視

C++17 属性、非標準属性の無視。

```
// OK、無視
[[ wefapiaofeaofjaopfij ]] int x ;
```

属性 C++ 独自拡張 C++ 規格準拠形式に追加機能。属性の無視の場合、C 使用、独自文法使用。機能必須。

3.20 構造化束縛

C++17 追加構造化束縛多値分解受取変数宣言文法。

```
int main()
{
    int a[] = { 1,2,3 } ;
    auto [b,c,d] = a ;

    // b == 1
    // c == 2
    // d == 3
}
```

C++ 方法多値扱。配列、`tuple`、`pair`。

```
int a[] = { 1,2,3 } ;
struct B
{
    int a ;
    double b ;
    std::string c ;
} ;

B b{ 1, 2.0, "hello" } ;
```

第3章 C++17 言語新機能

```
std::tuple< int, double, std::string > c { 1, 2.0, "hello" } ;
```

```
std::pair< int, int > d{ 1, 2 } ;
```

C++ 関数配列以外多値返す。

```
std::tuple< int, double, std::string > f()
{
    return { 1, 2.0, "hello" } ;
}
```

多値受取、多値固受取、多値分解受取。

多値固受取以下コード。

```
std::tuple< int, double, std::string > f()
{
    return { 1, 2.0, "hello" } ;
}

int main()
{
    auto result = f() ;

    std::cout << std::get<0>(result) << '\n'
              << std::get<1>(result) << '\n'
              << std::get<2>(result) << std::endl ;
}
```

多値分解受取以下コード。

```
std::tuple< int, double, std::string > f()
{
    return { 1, 2.0, "hello" } ;
}

int main()
{
    int a ;
    double b ;
    std::string c ;
```



```

        std::tie( a, b, c ) = f() ;

        std::cout << a << '\n'
                  << b << '\n'
                  << c << std::endl ;
    }

```

構造化束縛の使用、以下に示す通り。

```

        std::tuple< int, double, std::string > f()
        {
            return { 1, 2.0, "hello" } ;
        }

    int main()
    {
        auto [a, b, c] = f() ;

        std::cout << a << '\n'
                  << b << '\n'
                  << c << std::endl ;
    }

```

変数型は対応する多値型。場合、a, b, c は int, double, std::string 型。tuple、pair 使用。

```

    int main()
    {
        std::pair<int, int> p( 1, 2 ) ;

        auto [a,b] = p ;

        // a は int 型、値 1
        // b は int 型、値 2
    }

```

構造化束縛は if 文、switch 文、for 文でも使用。

```

    int main()
    {

```

第3章 C++17 言語新機能

```

int expr[] = {1,2,3} ;

if ( auto[a,b,c] = expr ; a )
{ }
switch( auto[a,b,c] = expr ; a )
{ }
for ( auto[a,b,c] = expr ; false ; )
{ }
}

```

構造化束縛 `range-based for` 文を使用。

```

int main()
{
    std::map< std::string, std::string > translation_table
    {
        {"dog", "犬"},
        {"cat", "猫"},
        {"answer", "42"}
    } ;

    for ( auto [key, value] : translation_table )
    {
        std::cout<<
            "key=" << key <<
            ", value=" << value << '\n' ;
    }
}

```

`map`、`map` 要素型 `std::pair<const std::string, std::string>` 構造化束縛 `[key, value]` 受ける。

構造化束縛配列を使用。

```

int main()
{
    int values[] = {1,2,3} ;
    auto [a,b,c] = values ;
}

```

構造化束縛 `struct` 使用。

```

struct Values

```

```

{
    int a ;
    double d ;
    std::string c ;
} ;

int main()
{
    Values values{ 1, 2.0, "hello" } ;

    auto [a,b,c] = values ;
}

```

構造化束縛は、非 `static` の変数にのみ適用可能。1 つの変数に `public` 修飾子がある場合、

構造化束縛 `constexpr` 変数に適用可能。

```

int main()
{
    constexpr int expr[] = { 1,2 } ;

    // 構造化束縛
    constexpr auto [a,b] = expr ;
}

```

3.20.1 超上級者向け解説

構造化束縛、変数宣言、構造化束縛宣言 (structured binding declaration) は分類として文法記述。構造化束縛宣言は宣言、単純宣言 (simple-declaration) は `for-range` 宣言 (for-range-declaration) は宣言、[識別子] は識別子。

単純宣言:

属性 `auto` CV 修飾子 (省略可) 修飾子 (省略可)
[識別子] 初期化子 ;

`for-range` 宣言:

属性 `auto` CV 修飾子 (省略可) 修飾子 (省略可)
[識別子] ;

識別子:

第3章 C++17 新言語新機能

区切り識別子

初期化子:

= 式
{ 式 }
(式)

以下に単純宣言の例。

```
int main()
{
    int e1[] = {1,2,3} ;
    struct { int a,b,c ; } e2{1,2,3} ;
    auto e3 = std::make_tuple(1,2,3) ;

    // "= 式" 例
    auto [a,b,c] = e1 ;
    auto [d,e,f] = e2 ;
    auto [g,h,i] = e3 ;

    // "{式}", "(式)" 例
    auto [j,k,l]{e1} ;
    auto [m,n,o](e1) ;

    // CV 修飾子 修飾子 修飾子 使 例
    auto const & [p,q,r] = e1 ;
}
```

以下に for-range 宣言の例。

```
int main()
{
    std::pair<int, int> pairs[] = { {1,2}, {3,4}, {5,6} } ;

    for ( auto [a, b] : pairs )
    {
        std::cout << a << ", " << b << '\n' ;
    }
}
```

3.20.2 構造化束縛宣言の仕様

構造化束縛の構造化束縛宣言は以下のように解釈される。

構造化束縛宣言は宣言した変数の数、初期化子の多値の数一致している必要がある。

```
int main()
{
    // 2 個の値を持つ
    int expr[] = {1,2} ;

    // 1 個、変数の少ない
    auto[a] = expr ;
    // 3 個、変数の多い
    auto[b,c,d] = expr ;
}
```

構造化束縛宣言は宣言した変数の変数名、記述の属性、CV 修飾子、修飾子変数の宣言。

3.20.3 初期化子の型が配列の場合

初期化子の配列の場合、変数の配列要素の初期化。

修飾子がある場合、変数の初期化。

```
int main()
{
    int expr[3] = {1,2,3} ;
    auto [a,b,c] = expr ;
}
```

以下と同義。

```
int main()
{
    int expr[3] = {1,2,3} ;

    int a = expr[0] ;
    int b = expr[1] ;
    int c = expr[2] ;
}
```

第3章 C++17 言語新機能

修飾子の場合、変数修飾子。

```
int main()
{
    int expr[3] = {1,2,3} ;
    auto & [a,b,c] = expr ;
    auto && [d,e,f] = expr ;
}
```

、以下同意味。

```
int main()
{
    int expr[3] = {1,2,3} ;

    int & a = expr[0] ;
    int & b = expr[1] ;
    int & c = expr[2] ;

    int && d = expr[0] ;
    int && e = expr[1] ;
    int && f = expr[2] ;
}
```

、変数型配列の場合、配列要素に対応配列要素初期化。

```
int main()
{
    int expr[][2] = {{1,2},{1,2}} ;
    auto [a,b] = expr ;
}
```

、以下同意味。

```
int main()
{
    int expr[][2] = {{1,2},{1,2}} ;

    int a[2] = { expr[0][0], expr[0][1] } ;
    int b[2] = { expr[1][0], expr[1][1] } ;
}
```

3.20.4 初期化子の型が配列ではなく、std::tuple_size<E> が完全形の名前である場合

構造化束縛宣言の初期化子の型 E が配列の場合、`std::tuple_size<E>` が完全形の名前の場合、構造化束縛宣言の初期化子の型 E 、値 e 。構造化束縛宣言の宣言の 1 目変数 0, 2 目変数 1, ..., i 。

`std::tuple_size<E>::value` は整数の定数式、値の初期化子の値の数。

```
int main()
{
    // std::tuple< int, int, int >
    auto e = std::make_tuple( 1, 2, 3 );
    auto [a,b,c] = e ;

    // std::tuple_size<decltype(e)>::size は 3
}
```

値の取得、非修飾名 `get` の型 E の探求。 `get` が見える場合、変数の初期化子 `e.get<i>()`。

```
auto [a,b,c] = e ;
```

構造化束縛宣言、以下を意味。

```
type a = e.get<0>() ;
type b = e.get<1>() ;
type c = e.get<2>() ;
```

`get` の宣言が見える場合、初期化子 `get<i>(e)`。場合、`get` の連想名前空間を探求。通常非修飾名前検索を行う。

```
// 通常非修飾名前検索を行う
type a = get<0>(e) ;
type b = get<1>(e) ;
type c = get<2>(e) ;
```

構造化束縛宣言の宣言の変数の型は以下で決定。

変数の型 `type` は “`std::tuple_element<i, E>::type`”。

```
std::tuple_element<0, E>::type a = get<0>(e) ;
```

第3章 C++17 言語新機能

```
std::tuple_element<1, E>::type b = get<1>(e) ;
std::tuple_element<2, E>::type c = get<2>(e) ;
```

以下、

```
int main()
{
    auto e = std::make_tuple( 1, 2, 3 ) ;
    auto [a,b,c] = e ;
}
```

以下同等意味。

```
int main()
{
    auto e = std::make_tuple( 1, 2, 3 ) ;

    using E = decltype(e) ;

    std::tuple_element<0, E>::type & a = std::get<0>(e) ;
    std::tuple_element<1, E>::type & b = std::get<1>(e) ;
    std::tuple_element<2, E>::type & c = std::get<2>(e) ;
}
```

以下、

```
int main()
{
    auto e = std::make_tuple( 1, 2, 3 ) ;
    auto && [a,b,c] = std::move(e) ;
}
```

以下意味。

```
int main()
{
    auto e = std::make_tuple( 1, 2, 3 ) ;

    using E = decltype(e) ;

    std::tuple_element<0, E>::type && a = std::get<0>(std::move(e)) ;
    std::tuple_element<1, E>::type && b = std::get<1>(std::move(e)) ;
    std::tuple_element<2, E>::type && c = std::get<2>(std::move(e)) ;
}
```



```

    }

```

3.20.5 上記以外の場合

上記以外の場合、構造化束縛宣言の初期化子型 E の型、非 static の public の直接、単一の曖昧の public 基本の必要。E の匿名 union の。

以下型 E の適切な例。

```

struct A
{
    int a, b, c ;
} ;

struct B : A { } ;

```

以下型 E の不適切な例。

```

// public 以外非 static
struct A
{
public :
    int a ;
private :
    int b ;
} ;

struct B
{
    int a ;
} ;
// 基本非static
struct C : B
{
    int b ;
} ;

// 匿名 union
struct D

```

第3章 C++17 新言語新機能

```

{
    union
    {
        int i ;
        double d ;
    }
};

```

型 E 非 static 変数宣言の順番に多値認識。
以下、

```

int main()
{
    struct { int x, y, z ; } e{1,2,3} ;

    auto [a,b,c] = e ;
}

```

以下の意味の等。

```

int main()
{
    struct { int x, y, z ; } e{1,2,3} ;

    int a = e.x ;
    int b = e.y ;
    int c = e.z ;
}

```

構造体束縛の対応。

```

struct S
{
    int x : 2 ;
    int y : 4 ;
};

int main()
{
    S e{1,3} ;
    auto [a,b] = e ;
}

```

機能 `__cpp_structured_bindings`, 値 201606。

3.21 inline 変数

C++17 変数 `inline` 指定。

```
inline int variable ;
```

変数 `inline` 変数呼。意味 `inline` 関数同。

3.21.1 inline の歴史的な意味

今昔、本書執筆 30 年以上昔、`inline` C++ 追加。

`inline` 現在意味誤解。

`inline` 関数意味、「関数強制的展開機能」。

大事一度書、`inline` 関数意味、「関数強制的展開機能」。

確、`inline` 関数意味、関数強制的展開機能。

関数展開、以下

```
int min( int a, int b )
{ return a < b ? a : b ; }
```

```
int main()
{
    int a, b ;
    std::cin >> a >> b ;

    // a < b 小方選
    int value = min( a, b ) ;
}
```

関数 `min` 十分小、関数呼出無視、以下最適化考。

```
int main()
{
    int a, b ;
    std::cin >> a >> b ;
```

関数の中身を展開、関数を展開。
人間関数を展開して手行面倒。読む。
“min(a,b)” “a<b?a:b” を読む。

展開可能最適化。展開逆遅延場
合。

関数、関数関数結果 1K 関数関数関数関数、関数関数関数関数関数 1000 件の場合、関数関数全体関数関数 1M 関数増。関数関数関数関数関数関数、CPU 関数関数関数関数関数。

図表1、関数実行時間と関数呼び出し回数、実行時間比と桁違の長さ、関数呼び出し回数の削減率の意味。

関数展開最適化適用決定、関数十分小、関数実行時間十分短、中、条件考慮。

昔C++の関数展開の判断。inline 追加。展開関数 inline 関数、関数関数関数展開関数関数認識。

現代C++、コンパイラ技術発展でC++も十分に賢い。関数展開の目的はinlineを使う必要はない。実際、現代C++もinline関数展開を強制しない。関数展開の判断はコンパイラに任せる。

`inline` 展開以外、`1` 意味 ODR (One Definition Rule、定義 `1` 原則) 回避。

C^{++} 語言、定義 C^{++} 中 1 個函數。

76

```
void f() { } // OK、定義
```

```
void f() { } // 再定義
```

通常、関数を使う場合宣言と書を使う。定義は 1 個の翻訳単位で書ける。

```
// f.h
```

```
void f() ;
```

```
// f.cpp
```

```
void f() { }
```

```
// main.cpp
```

```
#include "f.h"
```

```
int main()
```

```
{
```

```
    f() ;
```

```
}
```

関数、関数の展開、関数の実装上都合、関数の定義は 1 個の翻訳単位で書ける。

```
inline void f() ;
```

```
int main()
```

```
{
```

```
    // 関数、定義
```

```
    f() ;
```

```
}
```

関数、翻訳単位で定義、定義重複 ODR 違反。

C++ 関数問題解決、inline 関数定義同一、複数翻訳単位で定義。関数 ODR 違反。

```
// a.cpp
```

```
inline void f() { }
```

第3章 C++17 言語新機能

```

void a()
{
    f() ;
}

// b.cpp

// OK、inline 関数
inline void f() { }

void b()
{
    f() ;
}

```

例として同一 `inline` 関数を直接記述、`inline` 関数を定義の同一性保証、通常 `#include` を使。

3.21.3 `inline` 変数の意味

`inline` 変数、ODR 違反変数を定義重複を認。同名 `inline` 変数同変数を指。

```

// a.cpp

inline int data ;

void a() { ++data ; }

// b.cpp

inline int data ;

void b() { ++data ; }

// main.cpp

inline int data ;

int main()
{

```

```

    a() ;
    b() ;

    data ; // 2
}

```

例関数 `a`, `b` 中変数 `data` 同変数指。変数 `data` `static` 上構築変数開始時初期化。2 回値 `2` 。

、非 `static` 定義書。

C++17 以前 C++ 、以下書、

```

// S.h

struct S
{
    static int data ;
} ;

// S.cpp

int S::data ;

```

C++17 、以下書。

```

// S.h

struct S
{
    inline static int data ;
} ;

```

`S.cpp` 変数 `S::data` 定義書必要。

機能 `__cpp_inline_variables`, 値 201606。

3.22 可変長 using 宣言

機能超上級者向。

C++17 `using` 宣言区切。

第3章 C++17 新言語新機能

```
int x, y ;

int main()
{
    using ::x, ::y ;
}
```

例、C++14

```
using ::x ;
using ::y ;
```

本書等。

C++17、`using` 宣言展開。機能正式名前付、可変長 `using` 宣言 (Variadic using declaration) 呼。

```
template < typename ... Types >
struct S : Types ...
{
    using Types::operator() ... ;
    void operator () ( long ) { }
} ;

struct A
{
    void operator () ( int ) { }
} ;

struct B
{
    void operator () ( double ) { }
} ;

int main()
{
    S<A, B> s ;
    s(0) ; // A::operator()
    s(0L) ; // S::operator()
    s(0.0) ; // B::operator()
}
```


機能 `__cpp_variadic_using`, 値 201611。

3.23 std::byte : バイトを表現する型

C++17 型、バイト表現型入。バイト表現型言語特別型扱。

バイト C++ 単位、C++ 付与最小単位。C++ 規格 1 具体的何規定。過去 8 存在。

バイト数 `<climits>` 定義、`CHAR_BIT` 知。

C++17 型、1 UTF-8 8 1 単位表現規定。

`std::byte` 型、生列表現型使用。生 1 表 `unsigned char` 型慣習の使用、`std::byte` 型生 1 表現型、新 C++17 追加。複数連続、`unsigned char` 配列型、`std::byte` 配列型表現。

`std::byte` 型、`<cstdint>` 以下定義。

```
namespace std
{
    enum class byte : unsigned char { } ;
}
```

`std::byte` 型 `scoped enum` 型定義。、他整数型暗黙型変換行。

値 0x12 `std::byte` 型変数以下定義。

```
int main()
{
    std::byte b{0x12} ;
}
```

`std::byte` 型値場合、以下書。

```
int main()
{
```

第3章 C++17 言語新機能

```

std::byte b{} ;

b = std::byte( 1 ) ;
b = std::byte{ 1 } ;
b = static_cast< std::byte >( 1 ) ;
b = static_cast< std::byte >( 0b11110000 ) ;
}

```

`std::byte` 型は他数値型と同様の暗黙型変換規則。同様の型
 取得違反型他型演算規則防止。

```

int main()
{
    // 同、() 同初期化 int 型同暗黙変換入
    std::byte b1(1) ;

    // 同、=同初期化 int 型同暗黙変換入
    std::byte b2 = 1 ;

    std::byte b{} ;

    // 同、operator =同 int 型代入暗黙変換入
    b = 1 ;
    // 同、operator =同 double 型代入暗黙変換入
    b = 1.0 ;
}

```

`std::byte` 型 {} 同初期化規則、縮小変換禁止規則、
`std::byte` 型表現値範囲規則。

同、今 `std::byte` は 8 ビット、最小値 0、最大値 255 環境。

```

int main()
{
    // 同、表現値範囲
    std::byte b1{-1} ;
    // 同、表現値範囲
    std::byte b2{256} ;
}

```

`std::byte` は内部単位規則、規格上 `char` と同配慮行。

3.23 `std::byte` : 表現型

```
int main()
{
    int x = 42 ;

    std::byte * rep = reinterpret_cast< std::byte * >(&x) ;
}
```

`std::byte` 一部演算子、通常整数型、使、列演算、一部演算子。

具体的、以下示 OR, 列 AND, 列 XOR, 列 NOT。

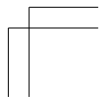
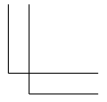
```
<<= <<
>>= >>
|= |
&= &
^= ^
~
```

四則演算演算子。

`std::byte` `std::to_integer<IntType>(std::byte)`、`IntType` 型整数型変換。

```
int main()
{
    std::byte b{42} ;

    // int 型値 42
    auto i = std::to_integer<int>(b) ;
}
```



第4章

C++17 の型安全な値を格納するライブラリ

C++17 の型安全な値を格納するライブラリ、`variant`、`any`、`optional` を追加する。

4.1 `variant` : 型安全な union

4.1.1 使い方

`<variant>` を定義するライブラリ `variant`、型安全な union を使用する。

```
#include <variant>

int main()
{
    using namespace std::literals ;

    // int, double, std::string を格納する variant
    // 最初型を構築
    std::variant< int, double, std::string > x ;

    x = 0 ;           // int を代入
    x = 0.0 ;         // double を代入
    x = "hello"s ;    // std::string を代入

    // int が入っているか確認
    // false を返す
    bool has_int = std::holds_alternative<int>( x ) ;
    // std::string が入っているか確認
```

第4章 C++17 型安全値格納

```

// true 返す
bool has_string = std::holds_alternative<std::string>( x );

// 入る値得る
// "hello"
std::string str = std::get<std::string>(x) ;
}

```

4.1.2 型非安全な古典的 union

C++ 従来保持する古典的 union、複数型 1 値格納型。union 型 1 表現。

```

union U
{
    int i ;
    double d ;
    std::string s ;
};

struct S
{
    int i ;
    double d ;
    std::string s ;
}

```

場合、sizeof(U)

```

sizeof(U) = max{sizeof(int),sizeof(double),sizeof(std::string)}
+

```

。 sizeof(S)

```

sizeof(S) = sizeof(int) + sizeof(double) + sizeof(std::string)
+

```

。

union 効率。 union variant 違型非安全。 型値保持情報保持、適切管理。

試、冒頭 union 書、以下。

4.1 variant : 型安全 union

```
union U
{
    int i ;
    double d ;
    std::string s ;

    // 编译选项
    // int 型初始化
    U() : i{} { }
    // 编译选项
    // 何。 破棄 責任
    ~U() { }
};

// 呼出
template < typename T >
void destruct ( T & x )
{
    x.~T() ;
}

int main()
{
    U u ;

    // 基本型代入
    // 破棄考
    u.i = 0 ;
    u.d = 0.0 ;

    // 非 持型
    // placement new 必要
    new(&u.s) std::string("hello") ;

    // 型入 別管理 必要
    bool has_int = false ;
    bool has_string = true ;

    std::cout << u.s << '\n' ;

    // 非 持型
```

第 4 章 C++17 型安全値格納

```
// 破棄必要
destruct( u.s );
}
```

型安全値格納。variant を使った、面倒な冗長な型安全 union の同等機能を實現する。

4.1.3 variant の宣言

variant は実引数を保持する型。

```
std::variant< char, short, int, long > v1 ;
std::variant< int, double, std::string > v2 ;
std::variant< std::vector<int>, std::list<int> > v3 ;
```

4.1.4 variant の初期化

variant の初期化

variant は構築可能、最初型値を構築可能保持する。

```
// int
std::variant< int, double > v1 ;
// double
std::variant< double, int > v2 ;
```

variant は構築可能型最初型、variant は構築可能。

```
// 構築可能型
struct non_default_constructible
{
    non_default_constructible() = delete ;
};

// 構築可能
// 構築可能
std::variant< non_default_constructible > v ;
```

構築可能型保持 variant は構築可能、最初型構築可能型。

```
struct A { A() = delete ; } ;
```


4.1 variant : 型安全 union

```

struct B { B() = delete ; } ;
struct C { C() = delete ; } ;

struct Empty { } ;

int main()
{
    // OK、Empty を保持
    std::variant< Empty, A, B, C > v ;
}

```

この場合、Empty は独自に定義し、面倒くさい、標準ライブラリ std::monostate を以下に定義する。

```

namespace std {
    struct monostate { } ;
}

```

上例以下に書く。

```

// OK、std::monostate を保持
std::variant< std::monostate, A, B, C > v ;

```

std::monostate を variant の最初の実引数に使用し、variant の構築可能型型。以上を意味する。

初期化

variant の同型 variant を渡す、型/型。

```

int main()
{
    std::variant<int> a ;
    // 
    std::variant<int> b ( a ) ;
}

```

variant の値渡し場合

variant は上記以外に値渡しの場合、variant の実引数に指定した型中、型解決した最適型選択、型値変換、値保持。

第 4 章 C++17 型安全値格納

```
using val = std::variant< int, double, std::string > ;

int main()
{
    // int
    val a(42) ;
    // double
    val b( 0.0 ) ;

    // std::string
    // char const *型 std::string 型変換
    val c("hello") ;

    // int
    // char 型 Integral promotion int 型優先の変換
    val d( 'a' ) ;
}
```

in_place_type **emplace** 構築

`variant` の第一引数 `std::in_place_type<T>` を渡す、`T` 型要素構築 `T` 型を渡す実引数指定。

型。

```
struct X
{
    X( int, int, int ) { }
};

int main()
{
    // X 構築
    X x( a, b, c ) ;
    // x
    std::variant<X> v( x ) ;
}
```

、型 `X` 型、上記型、動。

```
struct X
```


`std::in_place_type<T>` 使用。T 構築型指定第一引数、第二引数以降 T 渡値。

variant 保持、保持值適切破棄。

第4章 C++17 型安全値格納

```

std::variant<
    std::vector<int>,
    std::list<int>,
    std::deque<int>
> val ;

val = v ;
val = l ;
val = d ;

// variant deque<int>破壊
}

```

variant 何必要。

4.1.6 variant の代入

variant 代入自然。variant 渡、値渡
解決従適切型値保持。

4.1.7 variant の emplace

variant emplace 構築。variant 場合、構築型知
必要、emplace<T> T 構築型指定。

```

struct X
{
    X( int, int, int ) { }
    X( X const & ) = delete ;
    X( X && ) = delete ;
} ;

int main()
{
    std::variant<std::monostate, X, std::string> v ;

    // X 構築
    v.emplace<X>( 1, 2, 3 ) ;
    // std::string 構築
    v.emplace< std::string >( "hello" ) ;
}

```

4.1.8 variant に値が入っているかどうかの確認

valueless_by_exception 関数

`constexpr bool valueless_by_exception() const noexcept;`

`valueless_by_exception` 関数、`variant` 値保持の場合、`false` 返す。

```
void f( std::variant<int> & v )
{
    if ( v.valueless_by_exception() )
    { // true
        // v 値保持
    }
    else
    { // false
        // v 値保持
    }
}
```

`variant` 値保持状態。、`std::string` 動的確保行。 `variant` `std::string` 際、動的確保失敗場合、失敗。、`variant` 別型値構築前、以前値破棄。 `variant` 値持状態。

```
int main()
{
    std::variant< int, std::string > v ;
    try {
        std::string s("hello") ;
        v = s ; // 動的確保発生
    } catch( std::bad_alloc e )
    {
        // 動的確保失敗
    }

    // 動的確保失敗
    // true
```

第4章 C++17 型安全値格納

```
    bool b = v.valueless_by_exception() ;
}
```

index 関数

```
constexpr size_t index() const noexcept;
```

index 関数、variant 指定実引数、現在 variant 保持値型 0 返す。

```
int main()
{
    std::variant< int, double, std::string > v ;

    auto v0 = v.index() ; // 0
    v = 0.0 ;
    auto v1 = v.index() ; // 1
    v = "hello" ;
    auto v2 = v.index() ; // 2
}
```

variant 値保持場合、valueless_by_exception() true 返す場合、std::variant_npos 返す。

```
// variant 値保持確認関数
template < typename ... Types >
void has_value( std::variant< Types ... > && v )
{
    return v.index() != std::variant_npos ;

    // 確認
    // return v.valueless_by_exception() == false ;
}
```

std::variant_npos 値 -1。

4.1.9 swap

variant swap 対応。

```
int main()
{
    std::variant<int> a, b ;
```

```

        a.swap(b) ;
        std::swap( a, b ) ;
    }

```

4.1.10 variant_size<T> : variant が保持できる型の数を取得

std::variant_size<T>、T variant 型へ渡す、variant が保持する型数を返す。

```

using t1 = std::variant<char> ;
using t2 = std::variant<char, short> ;
using t3 = std::variant<char, short, int> ;

// 1
constexpr std::size_t t1_size = std::variant_size<t1>::size ;
// 2
constexpr std::size_t t2_size = std::variant_size<t2>::size ;
// 3
constexpr std::size_t t3_size = std::variant_size<t3>::size ;

```

variant_size integral_constant 基本型保持、型数構築の結果を定義変換して値を取出す。

```

using type = std::variant<char, short, int> ;

constexpr std::size_t size = std::variant_size<type>{} ;

variant_size 以下変数を用意。

template <class T>
    inline constexpr size_t variant_size_v = variant_size<T>::value;

使用、以下書き。

using type = std::variant<char, short, int> ;

constexpr std::size_t size = std::variant_size_v<type> ;

```

第4章 C++17 型安全値格納

4.1.11 `variant_alternative<I, T>` : インデックスから型を返す

`std::variant_alternative<I, T>` は T 型 variant が保持している型、I 番目の型 type を返す。

```
using type = std::variant< char, short, int > ;
```

```
// char
```

```
using t0 = std::variant_alternative< 0, type >::type ;
```

```
// short
```

```
using t1 = std::variant_alternative< 1, type >::type ;
```

```
// int
```

```
using t2 = std::variant_alternative< 2, type >::type ;
```

`variant_alternative_t` は以下で定義されている。

```
template <size_t I, class T>
```

```
using variant_alternative_t
```

```
= typename variant_alternative<I, T>::type ;
```

使用例、以下に示す。

```
using type = std::variant< char, short, int > ;
```

```
// char
```

```
using t0 = std::variant_alternative_t< 0, type > ;
```

```
// short
```

```
using t1 = std::variant_alternative_t< 1, type > ;
```

```
// int
```

```
using t2 = std::variant_alternative_t< 2, type > ;
```

4.1.12 `holds_alternative` : `variant` が指定した型の値を保持しているかどうかの確認

`holds_alternative<T>(v)` は、variant v が T 型値を保持しているかどうかを確認する。保持している場合は `true`、そうでない場合は `false` を返す。

```
int main()
```

```
{
```

```
    // int 型値構築
```

```
    std::variant< int, double > v ;
```


4.1 variant : 型安全 union

```

    // true
    bool has_int = std::holds_alternative<int>(v) ;
    // false
    bool has_double = std::holds_alternative<double>(v) ;
}

```

型 T の実引数と variant が保持する型は互換性がある。以下は互換性のある型を示す。

```

int main()
{
    std::variant< int > v ;

    // 互換性
    std::holds_alternative<double>(v) ;
}

```

4.1.13 get<I>(v) : インデックスから値の取得

get<I>(v) は、variant v の型 I 番目の型値を返す。I は 0 から variant の型数 - 1 までの整数。

```

int main()
{
    // 0: int
    // 1: double
    // 2: std::string
    std::variant< int, double, std::string > v(42) ;

    // int, 42
    auto a = std::get<0>(v) ;

    v = 3.14 ;
    // double, 3.14
    auto b = std::get<1>(v) ;

    v = "hello" ;
    // std::string, "hello"
    auto c = std::get<2>(v) ;
}

```

第 4 章 C++17 型安全値格納

I 範囲を超え。

```
int main()
{
    // 0, 1, 2
    std::variant< int, double, std::string > v ;

    // 、範囲外
    std::get<3>(v) ;
}
```

、variant 値保持場合、`v.index() != I` 場合、`std::bad_variant_access` throw。

```
int main()
{
    // int 型値保持
    std::variant< int, double > v( 42 ) ;

    try {
        // double 型値要求
        auto d = std::get<1>(v) ;
    } catch ( std::bad_variant_access & e )
    {
        // double 保持
    }
}
```

`get` 実引数渡 variant lvalue 場合、戻値 lvalue、`rvalue` 場合戻値 rvalue。

```
int main()
{
    std::variant< int > v ;

    // int &
    decltype(auto) a = std::get<0>(v) ;
    // int &&
    decltype(auto) b = std::get<0>( std::move(v) ) ;
}
```

4.1 variant : 型安全 union

get 実引数渡 variant CV 修飾 場合、戻値型実引数同 CV 修飾。

```
int main()
{
    std::variant< int > const cv ;
    std::variant< int > volatile vv ;
    std::variant< int > const volatile cvv ;

    // int const &
    decltype(auto) a = std::get<0>( cv ) ;
    // int volatile &
    decltype(auto) b = std::get<0>( vv ) ;
    // int const volatile &
    decltype(auto) c = std::get<0>( cvv ) ;
}
```

4.1.14 get<T>(v) : 型から値の取得

get<T>(v) 、variant v 保有型 T 値返。型 T 値保持 場合、std::bad_variant_access throw 。

```
int main()
{
    std::variant< int, double, std::string > v( 42 ) ;

    // int
    auto a = std::get<int>( v ) ;

    v = 3.14 ;
    // double
    auto b = std::get<double>( v ) ;

    v = "hello" ;
    // std::string
    auto c = std::get<std::string>( v ) ;
}
```

他 get<I> 同。

第 4 章 C++17 型安全値格納

4.1.15 `get_if` : 値を保持している場合に取得

`get_if<I>(vp)` は `get_if<T>(vp)` の、`variant` の `vp` 実引数取、`*vp` の `I`、`variant` 型 `T` の値を保持している場合、その値を返す。

```
int main()
{
    std::variant< int, double, std::string > v( 42 );

    // int *
    auto a = std::get_if<int>( &v );

    v = 3.14 ;
    // double *
    auto b = std::get_if<double>( &v );

    v = "hello" ;
    // std::string
    auto c = std::get_if<std::string>( &v );
}
```

また、`vp` が `nullptr` の場合、`*vp` が指定した値を保持している場合、`nullptr` を返す。

```
int main()
{
    // int 型値を保持
    std::variant< int, double > v( 42 );

    // nullptr
    auto a = std::get_if<int>( nullptr );

    // nullptr
    auto a = std::get_if<double>( &v );
}
```

4.1.16 variant の比較

variant 比較演算子。variant 同士比較、一般自然数思考結果実装。

同一性☒比較

variant 同一性比較、variant 実引数与型自分自身比較可能。

変数 v, w に対して、式 $\text{get}\langle i \rangle(v) == \text{get}\langle i \rangle(w)$ が i に対して
妥当である。

variant v, w 同一性比較、v == w 場合、以下4行。

1. `v.index() != w.index()` ☐ ☐ ☐、false
2. ☐ ☐以外☐場合、`v.value_less_by_exception()` ☐ ☐ ☐、true
3. ☐ ☐以外☐場合、`get<i>(v) == get<i>(w)`。☐ ☐ ☐ ☐ `i` ☐ `v.index()`

2 型 variant 別型保持する場合等。値の状態等。以外保持値同士比較。

```
int main()
{
    std::variant< int, double > a(0), b(0) ;

    // true
    // 同型同值保持
    a == b ;

    a = 1.0 ;

    // false
    // 型違
    a == b ;
}
```

operator == 以下実装。

```
template <class... Types>
constexpr bool
operator == (const variant<Types...>& v, const variant<Types...>& w)
{
    if ( v.index() != w.index() )
```

第4章 C++17 强类型安全枚举值格納

```

        return false ;
    else if ( v.valueless_by_exception() )
        return true ;
    else
        return std::visit(
            []( auto && a, auto && b ){ return a == b ; },
            v, w ) ;
}

```

operator != 逆参照考。

大小比較

variant 的大小比較、variant 実引数与型自身比較可能。

operator < 場合、variant v, w 対、式 get<i>(v) < get<i>(w) i 対妥当。

variant v, w 大小比較、v < w 場合、以下行。

1. w.valueless_by_exception() 対、false
2. 以外場合、v.valueless_by_exception() 対、true
3. 以外場合、v.index() < w.index() 対、true
4. 以外場合、v.index() > w.index() 対、false
5. 以外場合、get<i>(v) < get<i>(w)。 i v.index()

variant 最小小。最小小。同型値、値同比較。

```

int main()
{
    std::variant< int, double > a(0), b(0) ;

    // false
    // 同型同値比較
    a < b ;

    a = 1.0 ;

    // false
    // 比較
    a < b ;
    // true

```

```

        // 変数aと変数bの比較
        b < a ;
    }

```

operator < 以下同様に実装する。

```

template <class... Types>
constexpr bool
operator<(const variant<Types...>& v, const variant<Types...>& w)
{
    if ( w.valueless_by_exception() )
        return false ;
    else if ( v.valueless_by_exception() )
        return true ;
    else if ( v.index() < w.index() )
        return true ;
    else if ( v.index() > w.index() )
        return false ;
    else
        return std::visit(
            []( auto && a, auto && b ){ return a < b ; },
            v, w ) ;
}

```

残りの大小比較も同様に実装する。

4.1.17 visit : variant が保持している値を受け取る

std::visit、variant が保持している型の実引数関数関数を呼び出す。

```

int main()
{
    using val = std::variant<int, double> ;

    val v(42) ;
    val w(3.14) ;

    auto visitor = []( auto a, auto b )
        { std::cout << a << b << '\n' ; } ;

    // visitor( 42, 3.14 ) を呼び出す
    std::visit( visitor, v, w ) ;
}

```

第4章 C++17 型安全値格納

```
// visitor( 3.14, 42 ) を呼ぶ
std::visit( visitor, w, v );
}
```

、`variant` の型値を保持する抜け。
 `std::visit` 以下に宣言。

```
template < class Visitor, class... Variants >
constexpr auto visit( Visitor&& vis, Variants&&... vars ) ;
```

第一引数関数、第二引数以降 `variant` 渡す。、
`vis(get<i>(vars)...)` を呼ぶ。

```
int main()
{
    std::variant<int> a(1), b(2), c(3) ;

    // ( 1 )
    std::visit( []( auto x ) {}, a ) ;

    // ( 1, 2, 3 )
    std::visit( []( auto x, auto y, auto z ) {}, a, b, c ) ;
}
```

4.2 any : どんな型の値でも保持できるクラス

4.2.1 使い方

`<any>` を定義 `std::any` 、型の値を保持する。

```
#include <any>

int main()
{
    std::any a ;

    a = 0 ; // int
    a = 1.0 ; // double
    a = "hello" ; // char const *
```


4.2 any : 型値保持

```

std::vector<int> v ;
a = v ; // std::vector<int>

// 保持 std::vector<int>
auto value = std::any_cast< std::vector<int> >( a ) ;
}

```

any 保持型、構築型。

4.2.2 any の構築と破棄

any 型宣言。宣言単純。

```

int main()
{
    // 値保持
    std::any a ;
    // int 型値保持
    std::any b( 0 ) ;
    // double 型値保持
    std::any c( 0.0 ) ;
}

```

any 保持型事前指定必要。

any 破棄、保持値適切破棄。

4.2.3 in_place_type コンストラクター

any 型 emplacement in_place_type 使用。

```

struct X
{
    X( int, int ) { }
} ;

int main()
{
    // 型 X X(1, 2) 構築結果値保持
    std::any a( std::in_place_type<X>, 1, 2 ) ;
}

```

第 4 章 C++17 型安全値格納

4.2.4 any への代入

any へ代入は普通型値を期待し動作する。

```
int main()
{
    std::any a ;
    std::any b ;

    // a へ int 型値 42 を保持
    a = 42 ;
    // b へ int 型値 42 を保持
    b = a ;
}
```

4.2.5 any のメンバー関数

emplace

```
template <class T, class... Args>
decay_t<T>& emplace(Args&&... args);
```

any へ emplace は関数関数。

```
struct X
{
    X( int, int ) { }
};

int main()
{
    std::any a ;

    // 型 X へ X(1, 2) を構築し結果値を保持
    a.emplace<X>( 1, 2 ) ;
}
```

reset : 値の破棄

```
void reset() noexcept ;
```

4.2 any : 任意型値を保持するコンテナ

any の reset 関数は、any が保持している値を破棄する。reset を呼出した後、any の値は保持しない。

```
int main()
{
    // a の値を保持する
    std::any a ;
    // a の int 型の値を保持する
    a = 0 ;

    // a の値を保持する
    a.reset() ;
}
```

swap : 交換

any の swap 関数は、2つの any オブジェクトの値を交換する。

```
int main()
{
    std::any a(0) ;
    std::any b(0.0) ;

    // a の int 型の値を保持
    // b の double 型の値を保持

    a.swap(b) ;

    // a の double 型の値を保持
    // b の int 型の値を保持
}
```

has_value : 値を保持しているかどうかを調べる

```
bool has_value() const noexcept;
```

any の has_value 関数は、any が値を保持しているかどうかを調べる。値を保持している場合は true を、保持していない場合は false を返す。

```
int main()
{
    std::any a ;
```

第4章 C++17 型安全値格納

```

// false
bool b1 = a.has_value() ;

a = 0 ;
// true
bool b2 = a.has_value() ;

a.reset() ;
// false
bool b3 = a.has_value() ;
}

```

type : 保持型 **type_info** 得

```
const type_info& type() const noexcept;
```

type 関数、保持型 **T** の **typeid(T)** を返す。値保持の場合、**typeid(void)** を返す。

```

int main()
{
    std::any a ;

    // typeid(void)
    auto & t1 = a.type() ;

    a = 0 ;
    // typeid(int)
    auto & t2 = a.type() ;

    a = 0.0 ;
    // typeid(double)
    auto & t3 = a.type() ;
}

```

4.2.6 any のフリー関数

make_any<T> : **T** 型 **any** 作

```
template <class T, class... Args>
```

4.2 any : 任意型値を保持するコンテナ

```
any make_any(Args&& ...args);

template <class T, class U, class... Args>
any make_any(initializer_list<U> il, Args&& ...args);
```

make_any<T>(args...) は T 型の実引数 args... を構築し、値を保持する any を返す。

```
struct X
{
    X( int, int ) { }
};

int main()
{
    // int 型値を保持する any
    auto a = std::make_any<int>( 0 );
    // double 型値を保持する any
    auto b = std::make_any<double>( 0.0 );

    // X 型値を保持する any
    auto c = std::make_any<X>( 1, 2 );
}
```

any_cast : 保持する値を取り出す

```
template<class T> T any_cast(const any& operand);
template<class T> T any_cast(any& operand);
template<class T> T any_cast(any&& operand);
```

any_cast<T>(operand) は operand を保持する値を返す。

```
int main()
{
    std::any a(0);

    int value = std::any_cast<int>(a);
}
```

any_cast<T> は指定した T 型、any を保持する型の場合、std::bad_any_cast を throw する。

```
int main()
```

第4章 C++17 型安全値格納

```

{
    try {
        std::any a ;
        std::any_cast<int>(a) ;
    } catch( std::bad_any_cast e )
    {
        // 型保持
    }
}

```

```

template<class T>
const T* any_cast(const any* operand) noexcept;
template<class T>
T* any_cast(any* operand) noexcept;

```

`any_cast<T>` any へ変換、T 型へ返す。any
T 型へ保持した場合 T 型へ参照へ返す。保持の場合
、`nullptr` へ返す。

```

int main()
{
    std::any a(42) ;

    // int 型値参照
    int * p1 = std::any_cast<int>( &a ) ;

    // nullptr
    double * p2 = std::any_cast<double>( &a ) ;
}

```

4.3 optional : 値を保有しているか、していないクラス

4.3.1 使い方

`<optional>` 定義 `optional<T>` 、T 型値保有
、保有

条件次第値用意場合存在。割算結果値返関
数考。

4.3 optional : 値を保有、参照保持

```
int divide( int a, int b )
{
    if ( b == 0 )
    {
        // 除算エラー処理
    }
    else
        return a / b ;
}
```

除算エラー処理、b の値が 0 の場合、関数の値を用意する必要がある。問題点、int 型の値は通常除算結果を使用する、エラーを示す特別な値を返す必要がある。

エラーの場合、値を通知する方法、過去の方法を参考にする。エラー、エラーメッセージの実引数を受け取る方法、変数を使用する方法、例外。

optional の値を用意する場合、共通の方法を提供する。

```
std::optional<int> divide( int a, int b )
{
    if ( b == 0 )
        return {} ;
    else
        return { a / b } ;
}

int main()
{
    auto result = divide( 10, 2 ) ;
    // 値を取得
    auto value = result.value() ;

    // 除算エラー
    auto fail = divide( 10, 0 ) ;

    // false、値を保持
    bool has_value = fail.has_value() ;

    // throw bad_optional_access
    auto get_value_anyway = fail.value() ;
}
```

第 4 章 C++17 型安全値格納

4.3.2 optional のテンプレート実引数

`optional<T>` T 型値保持、保持状態取。

```
int main()
{
    // int 型値保持 optional
    using a = std::optional<int> ;
    // double 型値保持 optional
    using b = std::optional<double> ;
}
```

4.3.3 optional の構築

`optional` 構築、値保持 `optional`。

```
int main()
{
    // 値保持
    std::optional<int> a ;
}
```

実引数 `std::nullopt` 渡、値保持 `optional`。

```
int main()
{
    // 値保持
    std::optional<int> a( std::nullopt ) ;
}
```

`optional<T>` 実引数 T 型変換型渡し、T 型値型変換保持。

```
int main()
{
    // int 型値 42 保持
    std::optional<int> a(42) ;

    // double 型値 1.0 保持
    std::optional<double> b( 1.0 ) ;
}
```


4.3 optional : 値を保有する型、空の型

```
// int から double へ型変換を行う
// int 型値 1 を保持
std::optional<int> c ( 1.0 );
}
```

T 型から U 型へ型変換を行う、`optional<T>` から `optional<U>` へ渡す U から T へ型変換を行う T 型値を保持する `optional`。

```
int main()
{
    // int 型値 42 を保持
    std::optional<int> a( 42 );

    // long 型値 42 を保持
    std::optional<long> b ( a );
}
```

`optional` の第一引数 `std::in_place_type<T>` を渡す、後続引数を使って T 型を `emplace` 構築する。

```
struct X
{
    X( int, int ) { }
};

int main()
{
    // X(1, 2)
    std::optional<X> o( std::in_place_type<X>, 1, 2 );
}
```

4.3.4 optional の代入

通常期待する挙動。 `std::nullopt` を代入する値を保持する `optional`。

4.3.5 optional の破棄

`optional` の破棄、保持する値の破棄、適切に破棄する。

```
struct X
```

第4章 C++17 型安全値格納

```

{
    ~X() { }
};

int main()
{
    {
        // 値保持
        std::optional<X> o ( X{} );
        // X 呼
    }

    {
        // 値保持
        std::optional<X> o ;
        // X 呼
    }
}

```

4.3.6 swap

optional の swap 対応。

```

int main()
{
    std::optional<int> a(1), b(2) ;

    a.swap(b) ;
}

```

4.3.7 has_value : 値を保持しているかどうか確認する

```
constexpr bool has_value() const noexcept;
```

has_value 関数 optional の値保持の場合、true を返す。

```

int main()
{
    std::optional<int> a ;
    // false
    bool b1 = a.has_value() ;
}

```

4.3 optional : 値を保有しているかどうかを確認する

```

std::optional<int> b(42) ;
// true
bool b2 = b.has_value() ;
}

```

4.3.8 operator bool : 値を保持しているかどうかを確認する

```
constexpr explicit operator bool() const noexcept;
```

optional の文脈上 bool へ変換する場合、値を保持している場合 true と評価される。

```

int main()
{
    std::optional<bool> a = some_function();
    // OK、文脈上 bool へ変換
    if ( a )
    {
        // 値を保持
    }
    else
    {
        // 値を不保持
    }

    // 暗黙型変換を行う
    bool b1 = a ;
    // OK、明示的な型変換
    bool b2 = static_cast<bool>(a) ;
}

```

4.3.9 value : 保持している値を取得

```

constexpr const T& value() const&;
constexpr T& value() &;
constexpr T&& value() &&;
constexpr const T&& value() const&&;

```

value は関数 optional が値を保持している場合、値を取得する。

第4章 C++17 型安全値格納

返す。値保持の場合、`std::bad_optional_access` を throw する。

```
int main()
{
    std::optional<int> a(42) ;

    // OK
    int x = a.value () ;

    try {
        std::optional<int> b ;
        int y = b.value() ;
    } catch( std::bad_optional_access e )
    {
        // 値保持の場合
    }
}
```

4.3.10 `value_or` : 値もしくはデフォルト値を返す

```
template <class U> constexpr T value_or(U&& v) const&;
template <class U> constexpr T value_or(U&& v) &&;
```

`value_or(v)` は関数、`optional` 値保持の場合値、保持している場合 `v` を返す。

```
int main()
{
    std::optional<int> a( 42 ) ;

    // 42
    int x = a.value_or(0) ;

    std::optional<int> b ;

    // 0
    int x = b.value_or(0) ;
}
```

4.3 optional : 値の保有、状態の保持

4.3.11 reset : 保持している値を破棄する

reset 関数は関数を呼出した後、保持している値の場合破棄する。reset 関数を呼出した後 optional の値の保持状態は false になる。

```
int main()
{
    std::optional<int> a( 42 );

    // true
    bool b1 = a.has_value() ;

    a.reset() ;

    // false
    bool b2 = a.has_value() ;
}
```

4.3.12 optional 同士の比較

optional<T> の比較は、T 型の同士の比較が必要になる。

同一性の比較

値の保持する 2 つの optional を等しい。片方の値の保持する optional を等しい。両方の値の保持する optional の値を比較する。

```
int main()
{
    std::optional<int> a, b ;

    // true
    // 両方の値の保持する optional
    bool b1 = a == b ;

    a = 0 ;

    // false
    // a の値の保持
    bool b2 = a == b ;
}
```


大小比較

☒ ☒ ☒ ☐

4.3 optional : 値保持、参照保持

4.3.13 optional と std::nullopt との比較

optional と std::nullopt と比較、std::nullopt 値を持つ optional を扱。

4.3.14 optional<T> と T の比較

optional<T> と T 型と比較、optional<t> 値を持つ場合 false を返。以外場合、optional 保持値と T と比較。

```
int main()
{
    std::optional<int> o(1) ;

    // true
    bool b1 = ( o == 1 ) ;
    // false
    bool b2 = ( o == 0 ) ;

    // o 値保持
    o.reset() ;

    // T 値 false
    // false
    bool b3 = ( o == 1 ) ;
    // false
    bool b4 = ( o == 0 ) ;
}
```

4.3.15 make_optional<T> : optional<T> を返す

```
template <class T>
constexpr optional<decay_t<T>> make_optional(T&& v);

make_optional<T>(T t) optional<T>(t) 返。
```

```
int main()
{
    // std::optional<int>、値 0
    auto o1 = std::make_optional( 0 ) ;
```

第4章 C++17 型安全値格納

```
// std::optional<double>、値 0.0
auto o2 = std::make_optional( 0.0 );
}
```

4.3.16 make_optional<T, Args ...> : optional<T> を in_place_type 構築して返す

make_optional 第一引数 T 型の場合、in_place_type 構築関数を選択。

```
struct X
{
    X( int, int ) { }
};

int main()
{
    // std::optional<X>( std::in_place_type<X>, 1, 2 )
    auto o = std::make_optional<X>( 1, 2 );
}
```


第5章

string_view : 文字列ラッパー

5.1 使い方

第5章 string_view : 文字列の管理

```

void process_string( char * ptr, std::size_t size ) ;
// std::string の管理
void process_string( std::string s ) ;
// 自作の string_type の管理
void process_string( string_type s ) ;
// 自作の my_string_type の管理
void process_string( my_string_type s ) ;

```

string_view は文字列の管理に共通の view を提供し、
 多くの問題を解決する。多くの関数を追加する必要はない。

```

// 自作の string_type
struct string_type
{
    std::size_t size ;
    char * ptr ;

    // string_view に対応する変換関数
    operator std::string_view() const noexcept
    {
        return std::string_view( ptr, size ) ;
    }
}

```

```

// 1 行の文字列
void process_string( std::string_view s ) ;

```

```

int main()
{
    // OK
    process_string( "hello" ) ;
    // OK
    process_string( { "hello", 5 } ) ;

    std::string str( "hello" ) ;
    process_string( str ) ;

    string_type st{5, "hello"} ;

    process_string( st ) ;
}

```

5.2 basic_string_view

std::string は std::basic_string< CharT, Traits > に対して std::basic_string<char>、std::string_view、std::basic_string_view 特殊化 typedef 名。

```
// 本体
template<class charT, class traits = char_traits<charT>>
class basic_string_view ;

// 文字型 typedef 名
using string_view = basic_string_view<char>;
using u16string_view = basic_string_view<char16_t>;
using u32string_view = basic_string_view<char32_t>;
using wstring_view = basic_string_view<wchar_t>;
```

通常 basic_string_view、string_view u16string_view 等 typedef 名を使用する。本書 string_view 解説、他の typedef 名文字型は異なる。

5.3 文字列の所有、非所有

string_view は文字列の所有。所有、文字列の表現、確保、破棄責任を持つ。所有意味説明、文字列の所有説明。

std::string は文字列の所有。std::string 風実装、以下。

```
class string
{
    std::size_t size ;
    char * ptr ;

public :
    // 文字列の表現、動的確保
    string ( char const * str )
    {
        size = std::strlen( str ) ;
```

第 5 章 string_view : 文字列の管理

```

        ptr = new char[size+1] ;
        std::strcpy( ptr, str ) ;
    }

    // 文字列
    // 別文字列の動的確保
    string ( string const & r )
        : size( r.size ), ptr ( new char[size+1] )
    {
        std::strcpy( ptr, r.ptr ) ;
    }

    // 文字列
    // 所有権移動
    string ( string && r )
        : size( r.size ), ptr( r.ptr )
    {
        r.size = 0 ;
        r.ptr = nullptr ;
    }

    // 破棄
    // 動的確保文字列の解放
    ~string()
    {
        delete[] ptr ;
    }

};

std::string 文字列の表現、動的確保、所有。文字列
別文字列の動的確保。文字列の所有権移動。文字列
文字列の所有権破棄。

std::string_view 文字列の所有。std::string_view 風の実装
、以下。

class string_view
{
    std::size_t size ;
    char const * ptr ;

```

```

public :

    // 所有
    // str を参照先寿命と呼出側責任持
    string_view( char const * str ) noexcept
        : size( std::strlen(str) ), ptr( str )
    { }

    //
    // 初期化
    string_view( string_view const & r ) noexcept = default ;

    // 破棄
    // 何解放
    //
} ;

string_view 渡連続文字型配列の寿命、渡側
責任持。、以下間違。

std::string_view get_string()
{
    char str[] = "hello" ;

    //
    // str 寿命関数呼出元戻時点尽
    return str ;
}

```

5.4 string_view の構築

string_view の構築は 4 種類。

- の構築
- null 終端文字型配列
- 文字型配列の文字数
- 文字列の変換関数

第5章 string_view : 文字列

5.4.1 デフォルト構築

```
constexpr basic_string_view() noexcept;
```

string_view の構築、空 string_view の作。

```
int main()
{
    // 空 string_view
    std::string_view s ;
}
```

5.4.2 null 終端された文字型の配列へのポインター

```
constexpr basic_string_view(const charT* str);
```

string_view の、null 終端文字型の配列へのポインターを受け取る。

```
int main()
{
    std::string_view s( "hello" );
}
```

5.4.3 文字型へのポインターと文字数

```
constexpr basic_string_view(const charT* str, size_type len);
```

string_view の、文字型配列の文字数を受け取る。null 終端を受け取らない。

```
int main()
{
    char str[] = {'h', 'e', 'l', 'l', 'o'} ;

    std::string_view s( str, 5 );
}
```

5.4.4 文字列クラスからの変換関数

他の文字列型から `string_view` を作る、変換関数を使う。 `string_view` を使う。

`std::string` から `string_view` への変換関数がある。独自の文字列型から `string_view` へ対応する変換関数を使う。以下に実装例。

```
class string
{
    std::size_t size ;
    char * ptr ;
public :
    operator std::string_view() const noexcept
    {
        return std::string_view( ptr, size ) ;
    }
};
```

このように、 `std::string` から `string_view` への変換が可能。

```
int main()
{
    std::string s = "hello" ;
    std::string_view sv = s ;
}
```

同様に、独自の文字列型から `string_view` へ対応する変換関数を実装。

`std::string` から `string_view` を受け取り保持する、 `string_view` から `string` への変換。

```
int main()
{
    std::string_view sv = "hello" ;

    // 変換
    std::string s = sv ;
}
```

5.5 string_view の操作

string_view は既存の標準ライブラリ string と同様の操作性を提供する。文字列の取得、operator [] で要素の取得、size() で要素数を返す、find() で検索する。

```
template < typename T >
void f( T t )
{
    for ( auto c : t )
    {
        std::cout << c ;
    }

    if ( t.size() > 3 )
    {
        auto c = t[3] ;
    }

    auto pos = t.find( "fox" ) ;
}

int main()
{
    std::string s("quick brown fox jumps over the lazy dog.") ;

    f( s ) ;

    std::string_view sv = s ;

    f( sv ) ;
}
```

string_view は文字列の所有、文字列の書き換えの方法を提供しない。

```
int main()
{
    std::string s = "hello" ;
```



```

s[0] = 'H' ;
s += ",world" ;

std::string_view sv = s ;

// 文字列
// string_view の書換
sv[0] = 'h' ;
s += ".\n" ;
}

```

string_view の文字列の所有、参照の保持。

```

int main()
{
    std::string s = "hello" ;
    std::string_view sv = s ;

    // "hello"
    std::cout << sv ;

    s = "world" ;

    // "world"
    // string_view の参照
    std::cout << sv ;
}

```

string_view の string との互換性を保持、一部の文字列の変更を削除。

5.5.1 remove_prefix/remove_suffix : 先頭、末尾の要素の削除

string_view の先頭末尾の n 個の要素を削除する関数を提供。

```

constexpr void remove_prefix(size_type n);
constexpr void remove_suffix(size_type n);

```

string_view の先頭末尾の n 個の要素を削除、n 個の要素を削除、文字列の所有 string_view の行操作。

```

int main()

```

第 5 章 string_view : 文字列

```

{
    std::string s = "hello" ;

    std::string_view s1 = s ;

    // "lo"
    s1.remove_prefix(3) ;

    std::string_view s2 = s ;

    // "he"
    s2.remove_suffix(3) ;
}

```

関数既存 `std::string` 追加。

5.6 ユーザー定義リテラル

`std::string` `std::string_view` 定義追加。

```

string operator""s(const char* str, size_t len);
u16string operator""s(const char16_t* str, size_t len);
u32string operator""s(const char32_t* str, size_t len);
wstring operator""s(const wchar_t* str, size_t len);

constexpr string_view
operator""sv(const char* str, size_t len) noexcept;

constexpr u16string_view
operator""sv(const char16_t* str, size_t len) noexcept;

constexpr u32string_view
operator""sv(const char32_t* str, size_t len) noexcept;

constexpr wstring_view
operator""sv(const wchar_t* str, size_t len) noexcept;

```

以下使。

```

int main()
{
    using namespace std::literals ;
}

```

5.6 `string` 定义 `string_view`

```
// std::string
auto s = "hello"s ;

// std::string_view
auto sv = "hello"sv ;
}
```



第6章

メモリーリソース： 動的ストレージ確保ライブラリ

この章では、`<memory_resource>` で定義された動的ストレージ確保のインターフェイス、動的メモリーリソース確保の C++17 追加機能について説明する。この特徴は以下の通り。

- 動的メモリーリソース変換新機能の提供
- 動的メモリーリソース振舞いの可能範囲の拡大
- 標準で提供された特性を持つ動的メモリーリソースの実装

6.1 メモリーリソース

動的メモリーリソースは、動的メモリーリソース確保解放のインターフェイスを抽象化する。この時、挙動変換静的な設計と動的な設計の違い、実行時の挙動変換動的な設計の違いを説明する。

```
void f( memory_resource * mem )
{
    // 10 動的メモリーリソース確保
    auto ptr = mem->allocate( 10 );
    // 確保した動的メモリーリソース解放
    mem->deallocate( ptr );
}
```

この `std::pmr::memory_resource` は宣言は以下の通り。

```
namespace std::pmr {
```

第6章 動的メモリ確保

```

class memory_resource {
public:
    virtual ~memory_resource();
    void* allocate(size_t bytes, size_t alignment = max_align);
    void deallocate(void* p, size_t bytes,
                    size_t alignment = max_align);
    bool is_equal(const memory_resource& other) const noexcept;

private:
    virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
    virtual void do_deallocate( void* p, size_t bytes,
                               size_t alignment) = 0;
    virtual bool do_is_equal(const memory_resource& other)
        const noexcept = 0;
};

```

memory_resource std::pmr 名前空間中に定義されている。

6.1.1 メモリリソースの使い方

memory_resource を使うのは簡単。memory_resource を確保し、allocate(bytes, alignment) を呼び出してメモリを確保し、deallocate(p, bytes, alignment) を呼び出してメモリを解放する。

```

void f( std::pmr::memory_resource * mem )
{
    // 100 バイト確保
    void * ptr = mem->allocate( 100 );
    // メモリ解放
    mem->deallocate( ptr, 100 );
}

```

2 つの memory_resource a, b が与えられた場合、一方が a.is_equal(b) が true を返す。

```

void f( std::pmr::memory_resource * a, std::pmr::memory_resource * b )
{
    void * ptr = a->allocate( 1 );
}

```

6.1 メモリリソース

```

        // a 確保 b 解放?
        if ( a->is_equal( *b ) )
        {
            b->deallocate( ptr, 1 );
        }
        else
        {
            a->deallocate( ptr, 1 );
        }
    }

    is_equal 呼出 operator == operator != 提供。

    void f( std::pmr::memory_resource * a, std::pmr::memory_resource * b )
    {
        bool b1 = ( *a == *b );
        bool b2 = ( *a != *b );
    }

```

6.1.2 メモリリソースの作り方

独自 memory_resource 符合作、memory_resource 派生上、do_allocate, do_deallocate, do_is_equal 3 private 純粋 virtual 関数。必要応。

```

class memory_resource {
    // 非公開
    static constexpr size_t max_align = alignof(max_align_t);

public:
    virtual ~memory_resource();

private:
    virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
    virtual void do_deallocate( void* p, size_t bytes,
                                size_t alignment) = 0;
    virtual bool do_is_equal(const memory_resource& other)
        const noexcept = 0;
};

```

第6章 動的メモリ確保

`do_allocate(bytes, alignment)` 少なくとも `alignment` 以上の `bytes` バイトを確保し、返す。確保に失敗した場合、適切に例外 `throw` する。

`do_deallocate(p, bytes, alignment)` 事前と同様に `*this` を呼出し、`allocate(bytes, alignment)` が返す `p` を解放する。解放した `p` を渡す。例外を投げる。

`do_is_equal(other)` `&`、`*this` と `other` が互に一方を確保している一方を解放している場合 `true` を返す。

また、`malloc/free` を使った `memory_resource` を実装する以下。

```
// malloc/free を使った memory_resource
class malloc_resource : public std::pmr::memory_resource
{
public:
    //
    ~malloc_resource() { }
private:
    // 確保
    // 失敗の場合 std::bad_alloc を throw
    virtual void *
    do_allocate( std::size_t bytes, std::size_t alignment ) override
    {
        void * ptr = std::malloc( bytes );
        if ( ptr == nullptr )
        { throw std::bad_alloc{} ; }

        return ptr ;
    }

    // 解放
    virtual void
    do_deallocate( void * p, std::size_t bytes,
                  std::size_t alignment ) override
    {
        std::free( p ) ;
    }

    virtual bool
    do_is_equal( const memory_resource & other )
        const noexcept override
    { }
```


6.2 polymorphic_allocator : 動的メモリ管理の実現

```
{
    return dynamic_cast< const malloc_resource * >
        ( &other ) != nullptr ;
}
```

```
} ;
```

do_allocate は malloc を確保、do_deallocate は free を解放。0 を確保するのを確保。malloc は 0 を確保、C11 の規定、malloc は挙動を任す。malloc は 0 を確保、POSIX の null を返す、free は解放可能何を返す。

do_is_equal は、malloc_resource を確保、do_allocate は解放、*this は malloc_resource を dynamic_cast で確認。

6.2 polymorphic_allocator : 動的ポリモーフィズムを実現するアロケーター

std::pmr::polymorphic_allocator は動的メモリ管理の振舞いを提供する。

従来、静的メモリ管理の実現は設計上、独自に custom_int_allocator 型を使用する場合以下のように書く。

```
std::vector< int, custom_int_allocator > v ;
```

使用する時、使用するメモリを決定する場合、実行時に選択する場合、引数の取り設計の問題。

、C++17 は引数の取り、確保、実行時の振舞いを std::pmr::polymorphic_allocator に追加。

、標準入力 true false 入力、monotonic_buffer_resource は実行時に切替、以下のように書く。

```
int main()
{
```

第 6 章 動的メモリ確保

```

bool b;

std::cin >> b ;

std::pmr::memory_resource * mem ;
std::unique_ptr< memory_resource > mono ;

if ( b )
{ // 標準ライブラリが提供するメモリリソースを使用
    mem = std::pmr::get_default_resource() ;
}
else
{ // 独自のメモリリソースを使用
    mono = std::make_unique< std::pmr::monotonic_buffer_resource >
        ( std::pmr::get_default_resource() ) ;
    mem = mono.get() ;
}

std::vector< int, std::pmr::polymorphic_allocator<int> >
    v( std::pmr::polymorphic_allocator<int>( mem ) ) ;
}

std::pmr::polymorphic_allocator 以下に宣言されている。

namespace std::pmr {

template <class T>
class polymorphic_allocator ;

}

```

実引数 `std::allocator<T>` は、同型、確保型と一致。

6.2.1 コンストラクター

```

polymorphic_allocator() noexcept;
polymorphic_allocator(memory_resource* r);

std::pmr::polymorphic_allocator は、std::pmr::get_default_resource()
取得する。
memory_resource * 引数を取、渡す。polymorphic_allocator
生存期間中、使用する。

```

6.3 `std::pmr::polymorphic_allocator` 全体で使われるメモリの取得

`std::pmr::polymorphic_allocator` 妥当なメモリを取得する。

```
int main()
{
    // p1( std::pmr::get_default_resource () ) と同値
    std::pmr::polymorphic_allocator<int> p1 ;

    std::pmr::polymorphic_allocator<int> p2(
        std::pmr::get_default_resource() ) ;
}
```

後述の通常 `std::pmr::polymorphic_allocator` と同様に振る舞う。

6.3 プログラム全体で使われるメモリの取得

C++17 から、`std::pmr::new_delete_resource`、`std::pmr::null_memory_resource` 全体で使われるメモリの取得方法が提供された。

6.3.1 `std::pmr::new_delete_resource()`

```
memory_resource* new_delete_resource() noexcept ;
```

関数 `new_delete_resource` は `memory_resource` のポインタを返す。参照 `std::pmr::new_delete_resource`、`std::pmr::operator new`、`std::pmr::operator delete` を参照。

```
int main()
{
    auto mem = std::pmr::new_delete_resource() ;
}
```

6.3.2 `std::pmr::null_memory_resource()`

```
memory_resource* null_memory_resource() noexcept ;
```

関数 `null_memory_resource` は `memory_resource` のポインタを返す。参照 `std::pmr::null_memory_resource`、`std::pmr::allocate`、`std::pmr::deallocate`、`std::pmr::bad_alloc`、`std::pmr::throw`、`std::pmr::deallocate` を参照。

`std::pmr::null_memory_resource`、`std::pmr::operator new`、`std::pmr::operator delete` は、`std::pmr::null_memory_resource` が失敗した場合に使用される。

第6章 動的メモリ確保

6.3.3 デフォルトリソース

```
memory_resource* set_default_resource(memory_resource* r) noexcept ;
memory_resource* get_default_resource() noexcept ;
```

・、明示的に指定する場合、利用する初期値 `new_delete_resource()` 戻り値。

現在、関数 `get_default_resource` を使用。独自に差替、関数 `set_default_resource` を使用。

```
int main()
{
    // 現在
    auto init_mem = std::pmr::get_default_resource() ;

    std::pmr::synchronized_pool_resource pool_mem ;

    // 変更
    std::pmr::set_default_resource( &pool_mem ) ;

    auto current_mem = std::pmr::get_default_resource() ;

    // true
    bool b = current_mem == pool_mem ;
}
```

6.4 標準ライブラリのメモリーリソース

標準実装、提供。詳細後解説、事前知識、汎用的一般解説。

気軽確保。47 151 中途半端以下気軽確保。

```
int main()
{
```

```

auto mem = std::get_default_resource() ;

auto p1 = mem->allocate( 47 ) ;
auto p2 = mem->allocate( 151 ) ;

mem->deallocate( p1 ) ;
mem->deallocate( p2 ) ;
}

```

`memory_resource`、残念な現実 `memory_resource` OS 管理、柔軟な `memory_resource`。 `memory_resource`、 `memory_resource` OS、 `memory_resource` `memory_resource` 単位 `memory_resource` 確保。 `memory_resource` 最小 `memory_resource` 4K `memory_resource`。 `memory_resource` 低級 `memory_resource` 管理 `memory_resource` 上 `memory_resource` 実装 `memory_resource`、47 `memory_resource` 程度 `memory_resource` 使 `memory_resource` 3K `memory_resource` 超 `memory_resource` 無駄 `memory_resource` 生 `memory_resource`。

他 `memory_resource` 問題。 `memory_resource` `memory_resource` `memory_resource` 適切 `memory_resource` 配置 `memory_resource` `memory_resource`、著 `memory_resource` `memory_resource` 落 `memory_resource`。

`malloc` `operator new` `memory_resource`、低級 `memory_resource` 管理 `memory_resource` 隠 `memory_resource`、小 `memory_resource` `memory_resource` 確保 `memory_resource` 効率 `memory_resource` 行 `memory_resource` 実装 `memory_resource`。

一般的 `memory_resource`、大 `memory_resource` 連続 `memory_resource` 空間 `memory_resource` 確保、 `memory_resource` 中 `memory_resource` 管理用 `memory_resource` 構造 `memory_resource` 作、 `memory_resource` 必要 `memory_resource` 切 `memory_resource` 出。

```

// 実装

// memory_resource 分割管理 memory_resource 構造
struct alignas(std::max_align_t) chunk
{
    chunk * next ;
    chunk * prev ;
    std::size_t size ;
} ;

class memory_allocator : public std::pmr::memory_resource
{
    chunk * ptr ; // memory_resource 先頭 memory_resource
    std::size_t size ; // memory_resource
    std::mutex m ; // 同期用
}

```

第6章 動的メモリ確保

```

public :

    memory_allocator()
    {
        // 大領域連続メモリ確保
    }

    virtual void *
    do_allocate( std::size_t bytes, std::size_t alignment ) override
    {
        std::scoped_lock lock( m );
        // 領域確保、十分大領域未使用領域探し、メモリ構造体
        // 構築返す
        // 要求注意
    }

    virtual void *
    do_allocate( std::size_t bytes, std::size_t alignment ) override
    {
        std::scoped_lock lock( m );
        // 領域確保該当部分削除
    }

    virtual bool
    do_is_equal( const memory_resource & other )
        const noexcept override
    {
        // *this と other が相互解放可能返す
    }
};

```

6.5 プールリソース

C++17 標準で提供される `synchronized_pool_resource` と `unsynchronized_pool_resource` の 2 つ。

6.5.1 アルゴリズム

以下の特徴を持つ。

- ```
void f()
{
 std::pmr::synchronized_pool_resource mem ;
 mem.allocate(10) ;

 // 確保××××××××破棄×××
}
```

- ```
int main()
{
    // get_default_resource() 使用
    std::pmr::synchronized_pool_resource m1 ;

    // 独自上流指定
    custom_memory_resource mem ;
    std::pmr::synchronized_pool_resource m2( &mem ) ;
}
```

- ```
// 実装
namespace std::pmr {

// 実装
```

## 第6章 内存池：动态内存管理

```

template < size_t block_size >
class chunk
{
 blocks<block_size> b ;
}

// 内存池实现
template < size_t block_size >
class pool : public memory_resource
{
 chunks<block_size> c ;
} ;

class pool_resource : public memory_resource
{
 // 内存池实现
 pool<8> pool_8bytes ;
 pool<16> pool_16bytes ;
 pool<32> pool_32bytes ;

 // 上流内存管理
 memory_resource * mem ;

 virtual void * do_allocate(size_t bytes, size_t alignment) override
 {
 // 对应内存池实现
 if (bytes <= 8)
 return pool_8bytes.allocate(bytes, alignment) ;
 else if (bytes <= 16)
 return pool_16bytes.allocate(bytes, alignment) ;
 else if (bytes < 32)
 return pool_32bytes.allocate(bytes, alignment) ;
 else
 // 最大内存池实现超上流内存管理
 return mem->allocate(bytes, alignment) ;
 }
} ;

```



- `pool_options` 構造時 `pool_options` 渡す、最大サイズ設定。
- `synchronized_pool_resource` 呼出す安全同期取る、同期取る `unsynchronized_pool_resource`。

### 6.5.2 `synchronized/unsynchronized_pool_resource`

`synchronized_pool_resource`、`unsynchronized_pool_resource`。名前以外同使。同期、`synchronized_pool_resource` 複数同時に実行使内部同期取る対、`unsynchronized_pool_resource` 同期行。 `unsynchronized_pool_resource` 複数同時に呼出す。

// 実装

```
namespace std::pmr {

class synchronized_pool_resource : public memory_resource
{
 std::mutex m ;

 virtual void *
 do_allocate(size_t size, size_t alignment) override
 {
 // 同期
 std::scoped_lock l(m) ;
 return do_allocate_impl(size, alignment) ;
 }
};

class unsynchronized_pool_resource : public memory_resource
{
 virtual void *
 do_allocate(size_t size, size_t alignment) override
 {
 // 同期
 return do_allocate_impl(size, alignment) ;
 }
};
```

## 第6章 動的メモリ確保

}

## 6.5.3 pool\_options

pool\_options 挙動指定、以下定義。

```
namespace std::pmr {

struct pool_options {
 size_t max_blocks_per_chunk = 0;
 size_t largest_required_pool_block = 0;
};

}
```

pool\_options 与、挙動指定。pool\_options 指定目安、実装従義務。

max\_blocks\_per\_chunk 上流補充際一度確保最大数。値、実装上限大場合、実装上限使用。実装指定小値使用、別値使用。

largest\_required\_pool\_block 機構確保最大。値大確保、上流直接確保。値、実装上限大場合、実装上限使用。実装指定大値使用。

## 6.5.4 プールリソースのコンストラクター

根本的以下。synchronized unsynchronized 同。

```
pool_resource(const pool_options& opts, memory_resource* upstream);

pool_resource()
: pool_resource(pool_options(), get_default_resource()) {}
explicit pool_resource(memory_resource* upstream)
: pool_resource(pool_options(), upstream) {}
```

6.6 `monotonic_buffer_resource`

```
explicit pool_resource(const pool_options& opts)
: pool_resource(opts, get_default_resource()) {}
```

`pool_options` を `memory_resource` \* 指定。指定した場合 `memory_resource` 値を使用。

## 6.5.5 プールリソースのメンバー関数

`release()`

```
void release();
```

確保しているメモリを解放。明示的 `deallocate` を呼出。解放。

```
int main()
{
 synchronized_pool_resource mem ;
 void * ptr = mem.allocate(10);

 // ptr を解放
 mem.release();
}
```

`upstream_resource()`

```
memory_resource* upstream_resource() const;
```

構築時渡った上流 `memory_resource` を返す。

`options()`

```
pool_options options() const;
```

構築時渡った `pool_options` と同じ値を返す。

## 6.6 モノトニックバッファリソース

`monotonic_buffer_resource` は C++17 標準に追加された `memory_resource` の実装。名前 `monotonic_buffer_resource`。

`monotonic_buffer_resource` は高速確保、一気解放を実現。

## 第 6 章 動的メモリ確保

用途特化特殊設計。解放、使用量増続、名前付。

1 描画際大量小メモリ確保、後確保解放場合考。通常片解放全体構築、構造書換処理高。片一斉解放、構造書換必要。管理、単。

```
// 実装
```

```
namespace std::pmr {

class monotonic_buffer_resource : public memory_resource
{
 // 連続長大先頭
 void * ptr ;
 // 現在未使用先頭
 std::byte * current ;

 virtual void *
 do_allocate(size_t bytes, size_t alignment) override
 {
 void * result = static_cast<void *>(current) ;
 current += bytes ; // 必要調整
 return result ;
 }

 virtual void
 do_deallocate(void * ptr, size_t bytes, size_t alignment) override
 {
 // 何
 }

public :
 ~monotonic_buffer_resource()
 {
 // ptr 解放
 }
}
```

6.6 `std::pmr::monotonic_buffer_resource`

```
 }
} ;

}
```

`std::pmr::monotonic_buffer_resource`、基本的実装、`do_allocate` 加算管理、`do_deallocate` 解放処理、個々片管理、構造構築必要。do\_allocate 何。解放全体。

## 6.6.1 アルゴリズム

`std::pmr::monotonic_buffer_resource` 以下特徴持。

- `deallocate` 呼出何。使用量破棄増続。

```
int main()
{
 std::pmr::monotonic_buffer_resource mem ;

 void * ptr = mem.allocate(10) ;
 // 何
 // 解放
 mem.deallocate(ptr) ;

 // mem 破棄際確保破棄
}
```

- 確保使初期与。確保際、初期空場合確保。空場合上流確保、確保。

```
int main()
{
 std::byte initial_buffer[10] ;
 std::pmr::monotonic_buffer_resource
 mem(initial_buffer, 10, std::pmr::get_default_resource()) ;

 // 初期確保
```

## 第6章 動的メモリ確保

```

 mem.allocate(1);
 // 上流メモリ確保が完了したら、確保済みメモリを解放
 mem.allocate(100);
 // 前回メモリ確保が完了したら、確保済みメモリを解放
 // 新しく上流メモリ確保が完了したら、確保済みメモリを解放
 mem.allocate(100);
}

```

- 1. メモリ確保の前提設計。allocate と deallocate の同期。
- 2. メモリ確保の破棄。メモリ確保が完了したら、解放。明示的 deallocate の呼び出し。

## 6.6.2 コンストラクター

以下に、メモリ確保の初期化を行うコンストラクターを示す。

```

explicit monotonic_buffer_resource(memory_resource *upstream);
monotonic_buffer_resource(size_t initial_size,
 memory_resource *upstream);
monotonic_buffer_resource(void *buffer, size_t buffer_size,
 memory_resource *upstream);

```

```

monotonic_buffer_resource()
 : monotonic_buffer_resource(get_default_resource()) {}
explicit monotonic_buffer_resource(size_t initial_size)
 : monotonic_buffer_resource(initial_size,
 get_default_resource()) {}
monotonic_buffer_resource(void *buffer, size_t buffer_size)
 : monotonic_buffer_resource(buffer, buffer_size,
 get_default_resource()) {}

```

初期メモリ確保の取得は、以下に示すように行う。

```

explicit monotonic_buffer_resource(memory_resource *upstream);
monotonic_buffer_resource(size_t initial_size,
 memory_resource *upstream);

monotonic_buffer_resource()
 : monotonic_buffer_resource(get_default_resource()) {}

```

## 6.6 単調増加メモリリソース

```
explicit monotonic_buffer_resource(size_t initial_size)
 : monotonic_buffer_resource(initial_size,
 get_default_resource()) {}
```

`initial_size` は、上流メモリリソースが最初確保されたメモリ（初期メモリ）の総量を表す。実装は、`std::pmr::memory_resource` の実装依存で確保される。

`std::pmr::get_default_resource()` と同じ挙動をする。

`size_t` は、初期メモリと後メモリとの差を返す。

初期メモリは、`size_t` 以下のメモリである。

```
monotonic_buffer_resource(void *buffer, size_t buffer_size,
 memory_resource *upstream);
```

```
monotonic_buffer_resource(void *buffer, size_t buffer_size)
 : monotonic_buffer_resource(buffer, buffer_size,
 get_default_resource()) {}
```

初期メモリは、先頭の `void *` 型変数、`size_t` 型変数。

## 6.6.3 その他の操作

**release()**

```
void release() ;
```

`release` は、上流メモリリソースが解放されたことを示す。明示的に `deallocate` を呼ぶ必要はない。

```
int main()
{
 std::pmr::monotonic_buffer_resource mem ;

 mem.allocate(10) ;

 // メモリを解放
 mem.release() ;
}
```

第 6 章 `memory_resource` : 動的メモリの確保と解放`upstream_resource()`

```
memory_resource* upstream_resource() const;
```

`upstream_resource` 関数、構築時と上流の `memory_resource` を返す。



## 第7章

# 並列アルゴリズム

並列関数群は C++17 で追加された新関数群である。関数群は既存の `<algorithm>` に、並列実行版を追加した。

### 7.1 並列実行について

C++11 から、関数群同期処理を追加し、複数実行媒体で同時に実行する関数群概念は C++ 標準規格に入った。

C++17 から、既存関数群と並列実行版を追加した。

関数群、`all_of(first, last, pred)` 関数群は `[first,last)` 区間が空でない、関数群 `i` に対して `pred(*i)` が `true` を返す、`true` を返す。関数以外の場合 `false` を返す。

関数群値 100 未満関数群を調べ、以下関数群を書いた。

```
template < typename Container >
bool is_all_of_less_than_100(Container const & input)
{
 return std::all_of(std::begin(input), std::end(input),
 [](auto x) { return x < 100 ; }) ;
}

int main()
{
 std::vector<int> input ;
 std::copy(std::istream_iterator<int>(std::cin),
 std::istream_iterator<int>(), std::back_inserter(input)) ;

 bool result = is_all_of_less_than_100(input) ;
```

## 第 7 章 並列プログラミング

```
std::cout << "result : " << result << std::endl ;
}
```

本書執筆時点、`std::all_of` などの `std::` 関数は、一般的に、同時に複数の `std::` 関数を同時に実行させることができない。この問題を処理する 2 つの方法は、並列化を行うこと。

```
template < typename Container >
bool double_is_all_of_less_than_100(Container const & input)
{
 auto first = std::begin(input) ;
 auto last = first + (input.size()/2) ;

 auto r1 = std::async([=]
 {
 return std::all_of(first, last,
 [](auto x) { return x < 100 ; }) ;
 }) ;

 first = last ;
 last = std::end(input) ;

 auto r2 = std::async([=]
 {
 return std::all_of(first, last,
 [](auto x) { return x < 100 ; }) ;
 }) ;

 return r1.get() && r2.get() ;
}
```

`std::all_of`、`std::any_of`、`std::none_of`。

筆者は、`std::all_of`、`std::any_of`、`std::none_of` CPU の 2 倍物理コア、4 倍論理コアを持つマシンで、4 倍の `std::all_of` を同時に並列実行した。読者が使っているマシンで、`std::all_of` の高性能は多分同時に実行可能ではない。実行時の最大効率が出ない可能性がある。

```
template < typename Container >
bool parallel_is_all_of_less_than_100(Container const & input)
{
 std::size_t cores = std::thread::hardware_concurrency() ;
```

```

cores = std::min(input.size(), cores) ;

std::vector< std::future<bool> > futures(cores) ;

auto step = input.size() / cores ;
auto remainder = input.size() % cores ;

auto first = std::begin(input) ;
auto last = first + step + remainder ;

for (auto & f : futures)
{
 f = std::async([=]
 {
 return std::all_of(first, last,
 [](auto x){ return x < 100 ; }) ;
 }) ;

 first = last ;
 last = first + step ;
}

for (auto & f : futures)
{
 if (f.get() == false)
 return false ;
}
return true ;
}

```

。。

並列化。對自前実装面倒。  
、C++17 標準並列実行並列 (Parallelism) 追加。

## 7.2 使用方

並列既存追加。

以下既存 `all_of` 宣言。

## 第7章 並列アルゴリズム

```
template <class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last, Predicate pred);
```

並列版 `all_of` 以下に宣言する。

```
template < class ExecutionPolicy, class ForwardIterator,
 class Predicate>
bool all_of(ExecutionPolicy&& exec, ForwardIterator first,
 ForwardIterator last, Predicate pred);
```

並列版 `all_of`、仮引数 `ExecutionPolicy` に追加する第一引数を取る。実行時に呼び出す。

実行時に `<execution>` に定義する関数を用いる型、`std::execution::seq`、`std::execution::par`、`std::execution::par_unseq`。複数回並列実行を行う、`std::execution::par` を使う。

```
template < typename Container >
bool is_all_of_less_than_100(Container const & input)
{
 return std::all_of(std::execution::par,
 std::begin(input), std::end(input),
 [](auto x){ return x < 100 ; }) ;
}
```

`std::execution::seq` は渡した既存のアルゴリズムと同じように実行する。`std::execution::par` は渡したアルゴリズムを実行する。`std::execution::par_unseq` は並列実行する関数を実行する。

C++17 実行時に受取る関数に追加する。

## 7.3 並列アルゴリズム詳細

### 7.3.1 並列アルゴリズム

並列アルゴリズム (parallel algorithm) は、`ExecutionPolicy` (実行ポリシー) に渡した関数を実行する。既存の `<algorithm>` は C++14 に追加された一部関数、並列版に対応する。

並列版アルゴリズムは、仕様上定数操作、提供した関数操作、仕様上定数関数に対する操作を行う。

## 7.3 並列アルゴリズム詳細

、アルゴリズムを実行する。アルゴリズム関数群、要素アクセス関数 (element access functions) を呼ぶ。

、std::sort 以下アルゴリズム要素アクセス関数を持つ。

- 実引数とアルゴリズム関数群
- 要素対 swap 関数適用
- 提供 Compare 関数

並列アルゴリズム使用要素アクセス関数、並列実行に伴う制約を満たす。

## 7.3.2 ユーザー提供する関数オブジェクトの制約

並列アルゴリズム、アルゴリズム名、Predicate, BinaryPredicate, Compare, UnaryOperation, BinaryOperation, BinaryOperation1, BinaryOperation2 アルゴリズム、関数オブジェクト、関数提供関数、並列アルゴリズム提供関数提供関数、並列アルゴリズム渡す際制約。

- 実引数とアルゴリズム直接、間接変更
- 実引数とアルゴリズム一意性依存
- 競合同期

一部特殊アルゴリズム例外、アルゴリズム並列アルゴリズム制約を満たす。

## 実引数とアルゴリズム直接、間接変更

提供関数アルゴリズム実引数とアルゴリズム直接、間接変更。

、以下アルゴリズム違法。

```
int main()
{
 std::vector<int> c = { 1,2,3,4,5 };
 std::all_of(std::execution::par, std::begin(c), std::end(c),
 [](auto & x){ ++x ; return true ; }) ;
 //
}
```

、提供関数アルゴリズム実引数 lvalue 受取り変更、並列アルゴリズム制約を満たす。

## 第 7 章 並列処理

`std::for_each` 関数が変更可能な要素を返す場合、関数提供関数を実引数に変更する可能性がある。

```
int main()
{
 std::vector<int> c = { 1,2,3,4,5 };
 std::for_each(std::execution::par, std::begin(c), std::end(c),
 [](auto & x){ ++x ; }) ;
 // OK
}
```

`std::for_each` の仕様上関数は定数関数である。

## 実引数と一意性依存

関数提供関数を実引数と一意性依存関数とを渡すことは、一意性依存関数である。

関数提供関数、関数実引数を渡す関数、関数取得関数、関数渡す関数、関数期待関数、関数書関数。

```
int main()
{
 std::vector<int> c = { 1,2,3,4,5 };

 // 最後の要素を特別に処理
 int * ptr = &c[4] ;

 std::all_of(std::execution::par, std::begin(c), std::end(c),
 [=](auto & x){
 if (ptr == &x)
 {
 // 最後の要素を特別に処理
 // ...
 }
 }) ;
}
```

関数、並列処理、並列処理、並列処理、並列処理、要素作成、関数提供関数、関数渡す関数、関数期待関数、関数書関数。

```
// 実装

template < typename ExecutionPolicy,
 typename ForwardIterator,
 typename Predicate >
bool all_of(ExecutionPolicy && exec,
 ForwardIterator first, ForwardIterator last,
 Predicate pred)
{
 if constexpr (
 std::is_same_v< ExecutionPolicy,
 std::execution::parallel_policy >
)
 {
 std::vector c(first, last) ;
 do_all_of_par(std::begin(c), std::end(c), pred) ;
 }
}
```

、一意性依存を書き。

#### 競合同期

`std::execution::sequenced_policy` 渡す並列要素関数呼出し側実行。実行。

`std::execution::parallel_policy` 渡す並列要素関数呼出し側、側作同期定。要素関数呼出し同期定。要素関数競合起。

以下競合発生。

```
int main()
{
 int sum = 0 ;

 std::vector<int> c = { 1,2,3,4,5 } ;

 std::for_each(std::execution::par, std::begin(c), std::end(c),
 [&](auto x){ sum += x ; }) ;
 // 、競合
}
```

## 第 7 章 並列プログラミング

mutex、atomic 提供関数は複数のスレッドが同時に呼ばれることを保証する。

`std::execution::parallel_unsequenced_policy` は実行変換。未規定同期化を実行許可。mutex、atomic は実行想定を実行媒体、強い実行保証を実行媒体、SIMD や GPU 向けに実行媒体。

結果、要素は通常競合を防ぐ手段を取らない。mutex、atomic は実行途中で中断を別処理する。

mutex、以下は動作。

```
int main()
{
 int sum = 0 ;
 std::mutex m ;

 std::vector<int> c = { 1,2,3,4,5 } ;

 std::for_each(
 std::execution::par_unseq,
 std::begin(c), std::end(c),
 [&](auto x) {
 std::scoped_lock l(m) ;
 sum += x ;
 }) ;
 // 結果
}
```

parallel\_policy は、非効率の問題同期化競合を動作。mutex、parallel\_unsequenced\_policy は動作。mutex、mutex lock は同期化関数を呼ぶ。

C++ は、確保解放以外同期化標準関数、vectorization-unsafe は分類。vectorization-unsafe は関数 std::execution::parallel\_unsequenced\_policy は要素関数内呼ぶ。

## 7.3.3 例外

並列実行中、一時確保確保必要確保確保場合、`std::bad_alloc` を throw 。



並列実行中、要素関数外例外投の場合、`std::terminate` を呼ぶ。

#### 7.3.4 実行ポリシー

実行 `<execution>` 定義。定義以下。

```
namespace std {
 template<class T> struct is_execution_policy;
 template<class T> inline constexpr bool
 is_execution_policy_v = is_execution_policy<T>::value;
}

namespace std::execution {

 class sequenced_policy;
 class parallel_policy;
 class parallel_unsequenced_policy;

 inline constexpr sequenced_policy seq{ };
 inline constexpr parallel_policy par{ };
 inline constexpr parallel_unsequenced_policy par_unseq{ };

}
```

#### `is_execution_policy traits`

`std::is_execution_policy<T>` `T` 実行型を返す traits。

```
// false
constexpr bool b1 = std::is_execution_policy_v<int> ;
// true
constexpr bool b2 =
 std::is_execution_policy_v<std::execution::sequenced_policy> ;
```

#### 並列実行

```
namespace std::execution {

 class sequenced_policy ;
```

## 第7章 並列実行

```
inline constexpr sequenced_policy seq { } ;

}
```

実行、並列実行実行行。実行渡場合、処理呼出元実行行。

## 実行

```
namespace std::execution {

class parallel_policy ;
inline constexpr parallel_policy par { } ;

}
```

実行、並列実行実行行。実行渡場合、処理呼出元、作成用。

## 非実行

```
namespace std::execution {

class parallel_unsequenced_policy ;
inline constexpr parallel_unsequenced_policy par_unseq { } ;

}
```

非実行、並列実行実行行。実行渡場合、処理複数、SIMD GPU 実行並列化行。

## 実行

```
namespace std::execution {

inline constexpr sequenced_policy seq{ };
inline constexpr parallel_policy par{ };
inline constexpr parallel_unsequenced_policy par_unseq{ };

}
```

## 7.3 並列実行の詳細

```
}
```

実行型を直接書面倒す。

```
std::for_each(std::execution::parallel_policy{}, ...);
```

、標準実行型を実行型に用意する。seq par par\_unseq

```
std::for_each(std::execution::par, ...);
```

並列実行型を使用、並列実行型第一引数渡す。



## 第8章

# 数学の特殊関数群

C++17 の数学の特殊関数群 (mathematical special functions) の関数群は `<cmath>` に追加された。

数学の特殊関数、関数の実引数を取、規定の計算、結果の浮動小数点数型に戻値を返す。

数学の特殊関数 `double`, `float`, `long double` 型の 3 つの関数群がある。関数名の最後、何、`f`, `l` の関数群の表現がある。

```
double function_name() ; // 何
float function_namef() ; // f
long double function_name1() ; // l
```

数学の特殊関数の説明、関数の宣言、効果、戻値、注意。

、数学の特殊関数に渡す実引数 `NaN` (Not a Number) の場合、関数に戻値 `NaN` を返す。定義域の範囲を超え、

以外の場合、関数の定義域の範囲を超え、

- 関数の戻値の記述、定義域を示す実引数示す定義域を超え
- 実引数に対応の数学関数の結果の値が非実数部を含む
- 実引数に対応の数学関数の結果の値の数学的定義の範囲外

別途示す場合、関数の有限値、負無限大、正無限大に対する定義。

数学関数と実引数の値に対する定義、以下の関数群。

- 実引数の値の集合に対する明示的定義
- 計算方法に依存する極限値の存在

## 第 8 章 数学特殊関数群

関数効果実装定義 (implementation-defined) 場合、効果 C++ 標準規格定義、C++ 実装実装意味。

## 8.1 ラゲール多項式 (Laguerre polynomials)

```
double laguerre(unsigned n, double x);
float laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);
```

効果：実引数  $n, x$  対ラゲール多項式 (Laguerre polynomials) 計算。

戻値：

$$L_n(x) = \frac{e^x}{n!} \frac{d^n}{dx^n} (x^n e^{-x}), \quad \text{for } x \geq 0$$

$n \geq 0, x \geq 0$ 。

注意： $n \geq 128$  関数呼出効果実装定義。

## 8.2 ラゲール陪多項式 (Associated Laguerre polynomials)

```
double assoc_laguerre(unsigned n, unsigned m, double x);
float assoc_laguerref(unsigned n, unsigned m, float x);
long double assoc_laguerrel(unsigned n, unsigned m, long double x);
```

効果：実引数  $n, m, x$  対ラゲール陪多項式 (Associated Laguerre polynomials) 計算。

戻値：

$$L_n^m(x) = (-1)^m \frac{d^m}{dx^m} L_{n+m}(x), \quad \text{for } x \geq 0$$

$n \geq 0, m \geq 0, x \geq 0$ 。

注意： $n \geq 128$  関数呼出効果実装定義。

## 8.3 ルジャンドル多項式 (Legendre polynomials)

```
double legendre(unsigned l, double x);
float legendref(unsigned l, float x);
long double legendrel(unsigned l, long double x);
```

効果：実引数  $l, x$  対ルジャンドル多項式 (Legendre polynomials) 計算。

8.4  $\text{assoc\_legendre}$  関数 (Associated Legendre functions)

戻り値:

$$P_\ell(x) = \frac{1}{2^\ell \ell!} \frac{d^\ell}{dx^\ell} (x^2 - 1)^\ell, \quad \text{for } |x| \leq 1$$

 $\ell \geq 1, x \in \mathbb{R}$ 。注意:  $\ell \geq 128$  の関数呼び出し効果は実装定義。8.4  $\text{assoc\_legendre}$  関数 (Associated Legendre functions)

```
double assoc_legendre(unsigned l, unsigned m, double x);
float assoc_legendref(unsigned l, unsigned m, float x);
long double assoc_legendrel(unsigned l, unsigned m, long double x);
```

効果: 実引数  $\ell, m, x$  に対して  $\text{assoc\_legendre}$  関数 (Associated Legendre functions) を計算。

戻り値:

$$P_\ell^m(x) = (1 - x^2)^{m/2} \frac{d^m}{dx^m} P_\ell(x), \quad \text{for } |x| \leq 1$$

 $\ell \geq 1, m \leq m, x \in \mathbb{R}$ 。注意:  $\ell \geq 128$  の関数呼び出し効果は実装定義。8.5  $\text{sph\_legendre}$  関数 (Spherical associated Legendre functions)

```
double sph_legendre(unsigned l, unsigned m, double theta);
float sph_legendref(unsigned l, unsigned m, float theta);
long double sph_legendrel(unsigned l, unsigned m,
 long double theta);
```

効果: 実引数  $\ell, m, \text{theta}$  ( $\text{theta}$  は単位ラジアン) に対して球面  $\text{sph\_legendre}$  関数 (Spherical associated Legendre functions) を計算。

戻り値:

$$Y_\ell^m(\theta, 0)$$

 $\ell \geq 1, m \leq m$ 、

$$Y_\ell^m(\theta, \phi) = (-1)^m \left[ \frac{(2\ell + 1)(\ell - m)!}{4\pi(\ell + m)!} \right]^{1/2} P_\ell^m(\cos \theta) e^{im\phi}, \quad \text{for } |m| \leq \ell$$

 $\ell \geq 1, m \leq m, \theta \in [0, \pi], \phi \in [0, 2\pi)$ 。注意:  $\ell \geq 128$  の関数呼び出し効果は実装定義。

## 第 8 章 数学特殊関数群

球面調和関数 (Spherical harmonics)  $Y_\ell^m(\theta, \phi)$ 、以下関数関数定義計算。

```
#include <cmath>
#include <complex>

std::complex<double>
spherical_harmonics(unsigned l, unsigned m, double theta, double phi)
{
 return std::sph_legendre(l, m, theta) * std::polar(1.0, m * phi) ;
}
```

陪関数参照。

## 8.6 エルミート多項式 (Hermite polynomials)

```
double hermite(unsigned n, double x);
float hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);
```

効果：実引数  $n, x$  対エルミート多項式 (Hermite polynomials) 計算。  
戻値：

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}$$

$n \leq n, x \leq x$ 。

注意： $n \geq 128$  関数呼出効果実装定義。

## 8.7 ベータ関数 (Beta function)

```
double beta(double x, double y);
float betaf(float x, float y);
long double betal(long double x, long double y);
```

効果：実引数  $x, y$  対ベータ関数 (Beta function) 計算。  
戻値：

$$B(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}, \quad \text{for } x > 0, y > 0$$

$x \leq x, y \leq y$ 。



## 8.8 第 1 種完全楕円積分 (Complete elliptic integral of the first kind)

**8.8 第 1 種完全楕円積分 (Complete elliptic integral of the first kind)**

```
double comp_ellint_1(double k);
float comp_ellint_1f(float k);
long double comp_ellint_1l(long double k);
```

効果：実引数  $k$  に対して第 1 種完全楕円積分 (Complete elliptic integral of the first kind) を計算する。

戻値：

$$K(k) = F(k, \pi/2), \quad \text{for } |k| \leq 1$$

$k > 1$  の場合。

第 1 種不完全楕円積分を参照。

**8.9 第 2 種完全楕円積分 (Complete elliptic integral of the second kind)**

```
double comp_ellint_2(double k);
float comp_ellint_2f(float k);
long double comp_ellint_2l(long double k);
```

効果：実引数  $k$  に対して第 2 種完全楕円積分 (Complete elliptic integral of the second kind) を計算する。

戻値：

$$E(k) = E(k, \pi/2), \quad \text{for } |k| \leq 1$$

$k > 1$  の場合。

第 2 種不完全楕円積分を参照。

**8.10 第 3 種完全楕円積分 (Complete elliptic integral of the third kind)**

```
double comp_ellint_3(double k, double nu);
float comp_ellint_3f(float k, float nu);
long double comp_ellint_3l(long double k, long double nu);
```

## 第 8 章 数学特殊関数群

効果：実引数  $k$ ,  $nu$  対第 3 種完全楕円積分 (Complete elliptic integral of the third kind) 計算。

戻値：

$$\Pi(\nu, k) = \Pi(\nu, k, \pi/2), \quad \text{for } |k| \leq 1$$

$k$   $k$ ,  $\nu$   $nu$ 。

第 3 種不完全楕円積分参照。

### 8.11 第 1 種不完全楕円積分 (Incomplete elliptic integral of the first kind)

```
double ellint_1(double k, double phi);
float ellint_1f(float k, float phi);
long double ellint_1l(long double k, long double phi);
```

効果：実引数  $k$ ,  $phi$  ( $phi$  単位) 対第 1 種不完全楕円積分 (Incomplete elliptic integral of the first kind) 計算。

戻値：

$$F(k, \phi) = \int_0^\phi \frac{d\theta}{\sqrt{1 - k^2 \sin^2 \theta}}, \quad \text{for } |k| \leq 1$$

$k$   $k$ ,  $\phi$   $phi$ 。

### 8.12 第 2 種不完全楕円積分 (Incomplete elliptic integral of the second kind)

```
double ellint_2(double k, double phi);
float ellint_2f(float k, float phi);
long double ellint_2l(long double k, long double phi);
```

効果：実引数  $k$ ,  $phi$  ( $phi$  単位) 対第 2 種不完全楕円積分 (Incomplete elliptic integral of the second kind) 計算。

戻値：

$$E(k, \phi) = \int_0^\phi \sqrt{1 - k^2 \sin^2 \theta} d\theta, \quad \text{for } |k| \leq 1$$

$k$   $k$ ,  $\phi$   $phi$ 。

## 8.13 第3種不完全楕円積分 (Incomplete elliptic integral of the third kind)

## 8.13 第3種不完全楕円積分 (Incomplete elliptic integral of the third kind)

```
double ellint_3(double k, double nu, double phi);
float ellint_3f(float k, float nu, float phi);
long double ellint_3l(long double k, long double nu,
 long double phi);
```

効果：実引数  $k$ ,  $nu$ ,  $phi$  ( $phi$  は単位弧度) に対して第3種不完全楕円積分 (Incomplete elliptic integral of the third kind) を計算する。

戻値：

$$\Pi(\nu, k, \phi) = \int_0^\phi \frac{d\theta}{(1 - \nu \sin^2 \theta) \sqrt{1 - k^2 \sin^2 \theta}}, \quad \text{for } |k| \leq 1$$

$\nu$  は  $nu$ ,  $k$  は  $k$ ,  $\phi$  は  $phi$  である。

## 8.14 第1種ベッセル関数 (Cylindrical Bessel functions of the first kind)

```
double cyl_bessel_j(double nu, double x);
float cyl_bessel_jf(float nu, float x);
long double cyl_bessel_jl(long double nu, long double x);
```

効果：実引数  $nu$ ,  $x$  に対して第1種ベッセル関数 (Cylindrical Bessel functions of the first kind, Bessel functions of the first kind) を計算する。

戻値：

$$J_\nu(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{\nu+2k}}{k! \Gamma(\nu + k + 1)}, \quad \text{for } x \geq 0$$

$\nu$  は  $nu$ ,  $x$  は  $x$  である。

注意： $nu \geq 128$  の場合、関数の呼び出し効果が実装定義される。

## 8.15 ノイマン関数 (Cylindrical Neumann functions)

```
double cyl_neumann(double nu, double x);
float cyl_neumannf(float nu, float x);
long double cyl_neumannl(long double nu, long double x);
```

## 第8章 数学特殊関数群

効果：実引数  $\nu, x$  対第2種円筒関数 (Cylindrical Neumann functions, Neumann functions)、第2種円筒関数 (Cylindrical Bessel functions of the second kind, Bessel functions of the second kind) を計算。

戻り値：

$$N_{\nu}(x) = \begin{cases} \frac{J_{\nu}(x) \cos \nu\pi - J_{-\nu}(x)}{\sin \nu\pi}, & \text{for } x \geq 0 \text{ and non-integral } \nu \\ \lim_{\mu \rightarrow \nu} \frac{J_{\mu}(x) \cos \mu\pi - J_{-\mu}(x)}{\sin \mu\pi}, & \text{for } x \geq 0 \text{ and integral } \nu \end{cases}$$

$\nu$  対  $\nu$ ,  $x$  対  $x$ 。

注意： $\nu \geq 128$  の関数を呼出す効果は実装定義。

第1種円筒関数を参照。

### 8.16 第1種変形ベッセル関数 (Regular modified cylindrical Bessel functions)

```
double cyl_bessel_i(double nu, double x);
float cyl_bessel_if(float nu, float x);
long double cyl_bessel_il(long double nu, long double x);
```

効果：実引数  $\nu, x$  対第1種変形円筒関数 (Regular modified cylindrical Bessel functions, Modified Bessel functions of the first kind) を計算。

戻り値：

$$I_{\nu}(x) = i^{-\nu} J_{\nu}(ix) = \sum_{k=0}^{\infty} \frac{(x/2)^{\nu+2k}}{k! \Gamma(\nu+k+1)}, \quad \text{for } x \geq 0$$

$\nu$  対  $\nu$ ,  $x$  対  $x$ 。

注意： $\nu \geq 128$  の関数を呼出す効果は実装定義。

第1種円筒関数を参照。

### 8.17 第2種変形ベッセル関数 (Irregular modified cylindrical Bessel functions)

```
double cyl_bessel_k(double nu, double x);
float cyl_bessel_kf(float nu, float x);
long double cyl_bessel_kl(long double nu, long double x);
```

効果：実引数  $\nu, x$  対第2種変形円筒関数 (Irregular modified cylindrical Bessel functions, Modified Bessel functions of the second kind) を計算。

## 8.18 第1種球ベッセル関数 (Spherical Bessel functions of the first kind)

戻り値：

$$K_\nu(x) = (\pi/2)i^{\nu+1}(J_\nu(ix) + iN_\nu(ix))$$

$$= \begin{cases} \frac{\pi}{2} \frac{I_{-\nu}(x) - I_\nu(x)}{\sin \nu\pi}, & \text{for } x \geq 0 \text{ and non-integral } \nu \\ \frac{\pi}{2} \lim_{\mu \rightarrow \nu} \frac{I_{-\mu}(x) - I_\mu(x)}{\sin \mu\pi}, & \text{for } x \geq 0 \text{ and integral } \nu \end{cases}$$

$\nu$  は `nu`,  $x$  は `x` である。

注意：`nu` >= 128 の関数を呼出た効果は実装定義である。

第1種変形ベッセル関数、第1種球ベッセル関数、ベッセル関数を参照。

## 8.18 第1種球ベッセル関数 (Spherical Bessel functions of the first kind)

```
double sph_bessel(unsigned n, double x);
float sph_besself(unsigned n, float x);
long double sph_bessell(unsigned n, long double x);
```

効果：実引数 `n`, `x` に対して第1種球ベッセル関数 (Spherical Bessel functions of the first kind) を計算する。

戻り値：

$$j_n(x) = (\pi/2x)^{1/2} J_{n+1/2}(x), \quad \text{for } x \geq 0$$

注意：`n` >= 128 の関数を呼出た効果は実装定義である。

第1種ベッセル関数を参照。

## 8.19 球ノイマン関数 (Spherical Neumann functions)

```
double sph_neumann(unsigned n, double x);
float sph_neumannf(unsigned n, float x);
long double sph_neumannl(unsigned n, long double x);
```

効果：実引数 `n`, `x` に対して球ノイマン関数 (Spherical Neumann functions)、すなわち第2種球ベッセル関数 (Spherical Bessel functions of the second kind) を計算する。

戻り値：

$$n_n(x) = (\pi/2x)^{1/2} N_{n+1/2}(x), \quad \text{for } x \geq 0$$

$n$  は `n`,  $x$  は `x` である。

## 第 8 章 数学特殊関数群

注意：  $n \geq 128$  の関数呼び出し効果は実装定義。関数参照。

## 8.20 指数積分 (Exponential integral)

```
double expint(double x);
float expintf(float x);
long double expintl(long double x);
```

効果：実引数  $x$  に対する指数積分 (Exponential integral) を計算。

戻り値：

$$\text{Ei}(x) = - \int_{-x}^{\infty} \frac{e^{-t}}{t} dt$$

$x$  は実数。

## 8.21 リーマンゼータ関数 (Riemann zeta function)

```
double riemann_zeta(double x);
float riemann_zetaf(float x);
long double riemann_zetal(long double x);
```

効果：実引数  $x$  に対するリーマンゼータ関数 (Riemann zeta function) を計算。

戻り値：

$$\zeta(x) = \begin{cases} \sum_{k=1}^{\infty} k^{-x}, & \text{for } x > 1 \\ \frac{1}{1-2^{1-x}} \sum_{k=1}^{\infty} (-1)^{k-1} k^{-x}, & \text{for } 0 \leq x \leq 1 \\ 2^x \pi^{x-1} \sin\left(\frac{\pi x}{2}\right) \Gamma(1-x) \zeta(1-x), & \text{for } x < 0 \end{cases}$$

$x$  は実数。

## 第9章

# その他の標準ライブラリ

本章は C++17 の追加の細則の解説。

### 9.1 ハードウェア干渉サイズ（キャッシュライン）

C++17 はハードウェア干渉サイズ取得関数の導入。ハードウェア干渉サイズ、俗にキャッシュライン（cache line）と呼ばれる概念。

残念ながら、2017 年現在、ハードウェア干渉サイズは、高速なハードウェアで用意されている。ハードウェア干渉サイズは、ハードウェア干渉サイズ単位で行われる。ハードウェア干渉サイズは、ハードウェア干渉サイズ単位で実装依存。C++17 はハードウェア干渉サイズ取得関数の導入。

ハードウェア干渉サイズは、2 通り。2 通りのハードウェア干渉サイズは、同一局所性を持つハードウェア干渉サイズの場合、ハードウェア干渉サイズ。

2 通りのハードウェア干渉サイズ、一方は頻繁に変更、一方は一方の変更の場合、2 通りのハードウェア干渉サイズは同一局所性を持つハードウェア干渉サイズの場合、ハードウェア干渉サイズの変更、ハードウェア干渉サイズの変更、ハードウェア干渉サイズは同期発生。

```
struct Data
{
 int counter ;
 int status ;
};
```

、counter は頻繁に変更、status はハードウェア干渉サイズ変更の場合、counter と status は間隔適切にハードウェア干渉サイズ挿入、2 通りのハードウェア干渉サイズは同一局所性を持つハードウェア干渉サイズ。

## 第9章 其他標準庫

場合、`std::hardware_destructive_interference_size` を使。

```
struct Data
{
 int counter ;
 std::byte padding[
 std::hardware_destructive_interference_size - sizeof(int)
] ;
 int status ;
} ;
```

反対、2 個の同一局所性を持つメモリを載せる場合、`std::hardware_constructive_interference_size` を使。

メモリ干渉 <new> 以下に定義。

```
namespace std {
 inline constexpr size_t
 hardware_destructive_interference_size = 実装依存 ;
 inline constexpr size_t
 hardware_constructive_interference_size = 実装依存 ;
}
```

9.2 `std::uncaught_exceptions`

C++14 の、`catch` 例外の場合、`bool std::uncaught_exception()` を判定。

```
struct X
{
 ~X()
 {
 if (std::uncaught_exception())
 {
 // 例外発生中に呼ばれる
 }
 else
 {
 // 通常破壊
 }
 }
}
```



```

 } ;

 int main()
 {
 {
 X x ;
 } // 通常に破棄

 {
 X x ;
 throw 0 ;
 } // 例外が catch されずに中

 }

```

`bool std::uncaught_exception()` は、C++17 以前は推奨されなかった。この関数は廃止された見込がある。

廃止の理由としては、単に以下に示す例に役立たない。

```

struct X
{
 ~X()
 {
 try {
 // true
 bool b = std::uncaught_exception() ;
 } catch(...) { }
 }
} ;

```

代わりに、`int std::uncaught_exceptions()` が新しく追加された。この関数は現在 `catch` された例外の個数を返す。

```

struct X
{
 ~X()
 {
 try {
 if (int x = std::uncaught_exceptions() ; x > 1)
 {
 // 例外が 2 個以上発生
 }
 }
 }
} ;

```

## 第 9 章 他標準ライブラリ

```

 } catch(...)
 }

};

```

## 9.3 apply : tuple の要素を実引数に関数を呼び出す

```

template <class F, class Tuple>
constexpr decltype(auto) apply(F&& f, Tuple&& t);

```

`std::apply` は tuple の要素の順番を実引数に渡して関数を呼び出す関数。

要素数  $N$  の tuple  $t$  に関数  $f$  に対して、`apply( f, t )`、`f( get<0>(t), get<1>(t), ... , get<N-1>(t) )` と同様に  $f$  関数を呼び出す。

例：

```

template < typename ... Types >
void f(Types ... args) { }

int main()
{
 // int, int, int
 std::tuple t1(1,2,3) ;

 // f(1, 2, 3) 関数を呼出す
 std::apply(f, t1) ;

 // int, double, const char *
 std::tuple t2(123, 4.56, "hello") ;

 // f(123, 4.56, "hello") 関数を呼出す
 std::apply(f, t2) ;
}

```



## 第 9 章 其他標準庫組件

```

 fox_searcher(std::begin(pattern), std::end(pattern)) ;

 std::string corpus = "The quick brown fox jumps over the lazy dog" ;

 auto[first, last] = fox_searcher(std::begin(corpus),
 std::end(corpus)) ;

 std::string fox(first, last) ;
 }

```

default\_searcher 檢索、内部的 std::search 使用。

## 9.4.2 boyer\_moore\_searcher

std::boyer\_moore\_searcher Boyer-Moore 文字列檢索使用部分集合檢索行。

Boyer-Moore 文字列檢索極效率的文字列檢索。Boyer-Moore Bob Boyer Strother Moore 發明、1977 年 Communications of the ACM 發表。內容以下 URL 讀。

<http://www.cs.utexas.edu/~moore/publications/fstrpos.pdf>

愚直実装文字列檢索檢索部分文字列（）檢索対象文字列（）探際、先頭文字先頭順探、見 2 文字目以降一致調。

Boyer-Moore 末尾文字調。文字一致、絶対不一致長文字比較読飛。效率的文字列檢索實現。

Boyer-Moore 事前文字不一致何文字比較読飛情報計算 2 生成必要。、Boyer-Moore 使用量檢索前準備時間、效率的檢索相殺。特、長場合効果的。

C++17 入 Boyer-Moore 基檢索、使用汎用的 char 型状態数少型対実装、使用使、任意型対応設計。

boyer\_moore\_searcher 以下宣言。

```
template <
```

```

class RandomAccessIterator1,
class Hash = hash<
 typename iterator_traits<RandomAccessIterator1>::value_type>,
class BinaryPredicate = equal_to<> >
class boyer_moore_searcher {
public:
 // 関数型オブジェクト
 boyer_moore_searcher(
 RandomAccessIterator1 pat_first,
 RandomAccessIterator1 pat_last,
 Hash hf = Hash(),
 BinaryPredicate pred = BinaryPredicate()) ;

 // operator ()
 template <class RandomAccessIterator2>
 pair<RandomAccessIterator2, RandomAccessIterator2>
 operator()(RandomAccessIterator2 first,
 RandomAccessIterator2 last) const;
};

```

boyer\_moore\_searcher は、文字列以外に適用可能な汎用的な設計、関数型関数を取る。char 型に適用可能な状態数少ない型以外に渡す場合、std::unordered\_map を使用量削減に何らかの構造体を使用する構築。

使用法 default\_searcher に変換。

```

int main()
{
 std::string pattern("fox") ;
 std::boyer_moore_searcher
 fox_searcher(std::begin(pattern), std::end(pattern)) ;

 std::string corpus = "The quick brown fox jumps over the lazy dog" ;

 auto[first, last] = fox_searcher(std::begin(corpus),
 std::end(corpus)) ;
 std::string fox(first, last) ;
}

```

## 第 9 章 其他標準容器

## 9.4.3 boyer\_moore\_horspool\_searcher

`std::boyer_moore_horspool_searcher` は Boyer–Moore–Horspool 検索器。部分集合検索を行う。Boyer–Moore–Horspool は Nigel Horspool が 1980 年に発表された。

参考：“Practical fast searching in strings” 1980

Boyer–Moore–Horspool は内部に使用量削減、最悪計算量点 Boyer–Moore より劣る。しかし、実行時間増大犠牲に使用量削減を実現している。

`boyer_moore_horspool_searcher` は宣言以下。

```
template <
 class RandomAccessIterator1,
 class Hash = hash<
 typename iterator_traits<RandomAccessIterator1>::value_type>,
 class BinaryPredicate = equal_to<> >
 class boyer_moore_horspool_searcher {
public:
 // 構造体
 boyer_moore_horspool_searcher(
 RandomAccessIterator1 pat_first,
 RandomAccessIterator1 pat_last,
 Hash hf = Hash(),
 BinaryPredicate pred = BinaryPredicate());

 // operator ()
 template <class RandomAccessIterator2>
 pair<RandomAccessIterator2, RandomAccessIterator2>
 operator()(RandomAccessIterator2 first,
 RandomAccessIterator2 last) const;
};
```

使用法 `boyer_moore_horspool_searcher` は変換。

```
int main()
{
 std::string pattern("fox") ;
 std::boyer_moore_horspool_searcher
 fox_searcher(std::begin(pattern), std::end(pattern)) ;
```

## 9.5 sample : 乱択アルゴリズム

要素集合、 $n$  個要素確率の公平な選択の場合使われる。





## 9.5 sample : 乱択

以下は乱択のサンプルコード。

```
template < class PopulationIterator, class SampleIterator,
 class Distance, class UniformRandomBitGenerator >
SampleIterator sample(
 PopulationIterator first, PopulationIterator last,
 SampleIterator out,
 Distance n, UniformRandomBitGenerator&& g)
{
 auto N = std::distance(first, last) ;

 // 確率 n/N に true を返す二項分布
 double probability = double(n)/double(N) ;
 std::bernoulli_distribution d(probability) ;

 // 乱択した値の対
 std::for_each(first, last,
 [&](auto && value)
 {
 if (d(g))
 { // n/N に確率に比例して選択
 *out = value ;
 ++out ;
 }
 }) ;

 return out ;
}
```

残念なことに、この乱択は正しく動作しない。例えば、100 個の値の集合から 10 個の標本を選択する。このとき、選択した標本の数を実行回数を異ならせ、標本数の平均の 10 個選択の期待値、運が悪く 0 個から 100 個の標本を選択する可能性。

例えば、TAOCP Vol. 2 の乱択、乱択した標本数の標準偏差は  $\sqrt{n(1-n/N)}$  である。

正しく動作する乱択、要素の集合から  $(t+1)$  番目の要素、 $m$  個の要素の標本を選択する、 $(n-m)(N-t)$  の確率で選択される。

## 第9章 他標準

## 9.5.2 アルゴリズム S : 選択標本、要素数がわかっている集合からの標本の選択

Knuth の TAOCP Vol. 2 の、アルゴリズム S 名称、要素数  $N$  の集合から標本  $n$  個を選択する方法解説。

アルゴリズム S 以下。

$0 < n \leq N$  の、 $N$  個の集合から  $n$  個の標本を選択。

1.  $t, m \leftarrow 0$ 。  $t$  は処理した要素数、 $m$  は標本選択した要素数。
2.  $0 \leq U \leq N - t$  の範囲に乱数  $U$  を生成。
3.  $U \geq n - m$  なら goto 5。
4. 次の要素を標本を選択。  $m \leftarrow t + 1$ 。  $m < n$  なら、goto 2。標本の完了を終了。
5. 次の要素を標本を選択。  $t \leftarrow t + 1$ 。 goto 2。

実装以下。

```
template < class PopulationIterator, class SampleIterator,
 class Distance, class UniformRandomBitGenerator >
SampleIterator
sample_s(
 PopulationIterator first, PopulationIterator last,
 SampleIterator out,
 Distance n, UniformRandomBitGenerator&& g)
{
 // 1.
 Distance t = 0 ;
 Distance m = 0 ;
 const auto N = std::distance(first, last) ;

 auto r = [&]{
 std::uniform_int_distribution<> d(0, N-t) ;
 return d(g) ;
 } ;

 while (m < n && first != last)
 {
 // 2. 3.
 if (r() >= n - m)
 { // 5.
```

## 9.5 sample : 乱択法による乱択

```

 ++t ;
 ++first ;
 }
 else { // 4.
 *out = *first ;
 ++first ; ++out ;
 ++m ; ++t ;
 }
}

return out ;
}

```

## 9.5.3 アルゴリズム R : 保管標本、要素数がわからない集合からの標本の選択

集合  $S$  が要素数  $N$  個の場合、 $n$  個の標本を選択する。ここで、 $N$  がわからない場合、

現実  $N$  の状況。

- 入力
- 提供された全部読込要素数
- 入力

要素数  $N$  の入力  $S$  適用、一度全部入力得、全体要素数確定上、全要素対  $S$  適用 2 段階方法使用。

1 段階要素巡回済。要素数  $N$  の入力処理、時点公平選択標本得。

R の状況使用。

R の、要素数  $N$  の要素集合  $n$  個の標本選択。標本選択要素保管、新入力与、標本選択判断、選択、保管既存標本置換。

R の以下 (Knuth 本違)。

$n > 0$  の、 $size \geq n$  の未確定  $size$  個の要素数持入力、 $n$  個の標本選択。標本候補  $n$  個保管。  $1 \leq j \leq n$  の  $I[j]$  保管標本指。

## 第9章 他標準ライブラリ

1. 入力最初  $n$  個の標本を選択、保管。  $1 \leq j \leq n$  の範囲  $I[j]$  の  $j$  番目の標本を保管。  $t$  の値  $n$  となる。  $I[1], \dots, I[n]$  が現在の標本を指す。  $t$  が現在処理する入力の個数を指す。
2. 入力が終わったら終了。
3.  $t$  の範囲  $1 \leq M \leq t$  の乱数  $M$  を生成。  $M > n$  なら goto 5。
4. 次の入力  $I[M]$  を保管。 goto 2。
5. 次の入力を保管。 goto 2。

実装は以下。

```
template < class PopulationIterator, class SampleIterator,
 class Distance, class UniformRandomBitGenerator >

SampleIterator sample_r(
 PopulationIterator first, PopulationIterator last,
 SampleIterator out,
 Distance n, UniformRandomBitGenerator&& g)
{
 Distance t = 0 ;

 auto result = out ;

 for (; (first != last) && (t != n) ; ++first, ++t, ++result)
 {
 out[t] = *first ;
 }

 if (t != n)
 return result ;

 auto I = [&](Distance j) -> decltype(auto) { return out[j-1] ; } ;

 while (first != last)
 {
 ++t ;
 std::uniform_int_distribution<Distance> d(1, t) ;
 auto M = d(g) ;
```

## 9.5 sample : 乱択

```

 if (M > n)
 {
 ++first ;
 }
 else {
 I(M) = *first ;
 ++first ;
 }
 }

 return result ;
}

```

## 9.5.4 C++ の sample

説明、乱択 2 種類。入力要素数  $S$  (選択標本)、入力要素数  $R$  (保管標本)。

C++ 追加乱択関数宣言、説明以下 1 行。並列対応。

```

template<
 class PopulationIterator, class SampleIterator,
 class Distance, class UniformRandomBitGenerator >
SampleIterator
sample(
 PopulationIterator first, PopulationIterator last,
 SampleIterator out,
 Distance n, UniformRandomBitGenerator&& g) ;

```

$[first, last)$  標本選択先集合指。out 標本出力先。n 選択標本個数。g 標本選択使乱数生成器。戻値 out。

sample PopulationIterator SampleIterator、判断。

$S$  (選択標本) 使場合、PopulationIterator 前方、SampleIterator 出力満。

$R$  (保管標本) 使場合、PopulationIterator 入力、SampleIterator 満。

要素数取得、入力元

## 第9章 其他標準函數

`PopulationIterator` `[first, last)` 要素数を得る必要はない、`PopulationIterator` 前方に満たない場合、選択した標本を出力し、出力先 `SampleIterator` が出力を満たす。

入力元 `PopulationIterator` が入力に満たない場合、`PopulationIterator` `[first, last)` 要素数を得る必要はない、要素数を得る使用 `R` (保管標本) 選択を得る。場合、入力処理を連続、新しい選択した標本を既存の標本に書き込み、出力先 `SampleIterator` が出力を満たす必要はない。

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

 std::knuth_b g ;

 // PopulationIterator 前方に満たない
 // SampleIterator 出力を満たす
 std::sample(std::begin(input), std::end(input),
 std::ostream_iterator<int>(std::cout), 100
 g) ;

 std::vector<int> sample(100) ;

 // PopulationIterator 入力に満たない
 // SampleIterator が出力を満たす必要
 std::sample(
 std::istream_iterator<int>(std::cin),
 std::istream_iterator<int>{},
 std::begin(sample), 100, g) ;

}
```

注意が必要なのは、C++ の `sample` は入力元 `PopulationIterator` 前方に満たない以上、必ず `S` (選択標本) を使用する。要素数を得る `std::distance(first, last)` は行の意味。処理は非効率の渡す場合、必要以上非効率の。

以下、

## 9.5 sample : 乱択

```
int main()
{
 std::list<int> input(10000) ;
 std::list<int> sample(100) ;
 std::knuth_b g ;

 std::sample(std::begin(input), std::end(input),
 std::begin(sample), 100, g) ;
}
```

以下コード意味保持。

```
int main()
{
 std::list<int> input(10000) ;
 std::list<int> sample(100) ;
 std::knuth_b g ;

 std::size_t count = 0 ;

 // 要素数取得
 // 非効率的
 for(auto && e : input)
 { ++count ; }

 // 標本選択
 for (auto && e : input)
 { /* 標本選択 */ }
}
```

std::list の関数 size は定数時間保証、  
 要素数渡実引数要素数全走査の場合、非効率的の処理行。

範囲標本選択の場合、範囲指要素数場合、自前 S 実装効率。

```
template < class PopulationIterator, class SampleIterator,
 class Distance, class UniformRandomBitGenerator >
```

## 第9章 他標準ライブラリ

```

SampleIterator
sample_s(
 PopulationIterator first, PopulationIterator last,
 Distance size,
 SampleIterator out,
 Distance n, UniformRandomBitGenerator&& g)
{
 // 1.
 Distance t = 0 ;
 Distance m = 0 ;
 const auto N = size ;

 auto r = [&]{
 std::uniform_int_distribution<> d(0, N-t) ;
 return d(g) ;
 } ;

 while (m < n && first != last)
 {
 // 2. 3.
 if (r() >= n - m)
 { // 5.
 ++t ;
 ++first ;
 }
 else { // 4.
 *out = *first ;
 ++first ; ++out ;
 ++m ; ++t ;
 }
 }

 return out ;
}

```

9.6 `shared_ptr<T[]>` : 配列に対する `shared_ptr`

C++17 から、`shared_ptr` が配列に対応。

```
int main()
```



```

{
 // 配列対応 shared_ptr
 std::shared_ptr< int [] > ptr(new int[5]);

 // operator [] 配列添字
 ptr[0] = 42 ;

 // shared_ptr delete[] 呼出
}

```

## 9.7 as\_const : const 性の付与

as\_const 関数 <utility> 定義。

```

template <class T> constexpr add_const_t<T>& as_const(T& t) noexcept
{
 return t ;
}

```

as\_const 引数 lvalue const lvalue 関数。const 性の付与を手軽に関数使用。

```

// 1
template < typename T >
void f(T &) {}
// 2、関数呼出
template < typename T >
void f(T const &) { }

int main()
{
 int x{} ;

 f(x) ; // 1

 // const 付与冗長方法
 int const & ref = x ;
 f(ref) ; // 2
}

```

## 第9章 他標準ライブラリ

```

// 簡潔
f(std::as_const(x)) ; // 2
}

```

## 9.8 make\_from\_tuple : tuple の要素を実引数にコンストラクターを呼び出す

make\_from\_tuple 関数は <tuple> を定義する。

```

template <class T, class Tuple>
constexpr T make_from_tuple(Tuple&& t);

```

apply 関数、tuple 要素、実引数関数、関数呼び出し、make\_from\_tuple 関数、tuple 要素、実引数関数呼び出し。

型 T 要素数 N の tuple t に対して、make\_from\_tuple<T>(t) は、T 型 T( get<0>(t), get<1>(t), ... , get<N-1>(t) ) を構築し、構築した T 型を返す。

```

class X
{
 template < typename ... Types >
 T(Types ...) { }
};

int main()
{
 // int, int, int
 std::tuple t1(1,2,3) ;

 // X(1,2,3)
 X x1 = std::make_from_tuple<X>(t1)

 // int, double, const char *
 std::tuple t2(123, 4.56, "hello") ;

 // X(123, 4.56, "hello")
 X x2 = std::make_from_tuple<X>(t2) ;
}

```



## 第9章 他標準ライブラリ

```

 // s.*&S::data
 std::invoke(&S::data, s);
 }

```

## 9.10 not\_fn : 戻り値の否定ラッパー

not\_fn ライブラリ関数 <functional> に定義されている。

```
template <class F> unspecified not_fn(F&& f);
```

関数 f に対して not\_fn(f) を呼出し、戻り値が何らかの関数  
を返す。関数を呼出し、実引数 f を渡す  
f 関数を呼出し、戻り値 operator ! を否定して返す。

```

int main()
{
 auto r1 = std::not_fn([]{ return true ; });

 r1() ; // false

 auto r2 = std::not_fn([](bool b) { return b ; });

 r2(true) ; // false
}

```

廃止予定 not1, not2 の代替品。

## 9.11 メモリー管理アルゴリズム

C++17 ライブラリ関数 <memory> にメモリー管理用関数を追加  
した。

## 9.11.1 addressof

```
template <class T> constexpr T* addressof(T& r) noexcept;
```

addressof は C++17 以前にはなかった。addressof(r) は r のアドレス  
を返す。r の型を operator & で正しく取得する。

9.11 `管理`

得。

```
struct S
{
 S * operator &() const noexcept
 { return nullptr ; }
} ;

int main()
{
 S s ;

 // nullptr
 S * p1 = & s ;
 // 妥当
 S * p2 = std::addressof(s) ;

}
```

9.11.2 `uninitialized_default_construct`

```
template <class ForwardIterator>
void uninitialized_default_construct(
 ForwardIterator first, ForwardIterator last);
```

```
template <class ForwardIterator, class Size>
ForwardIterator uninitialized_default_construct_n(
 ForwardIterator first, Size n);
```

`[first, last)` 範囲、`first` から `n` 個の範囲を初期化。  
`typename iterator_traits<ForwardIterator>::value_type` を初期化。  
 2 目 `first` から `n` 個の範囲を初期化。

```
int main()
{
 std::shared_ptr<void> raw_ptr
 (::operator new(sizeof(std::string) * 10),
 [](void * ptr){ ::operator delete(ptr) ; }) ;

 std::string * ptr = static_cast<std::string *>(raw_ptr.get()) ;
```

## 第 9 章 其他標準庫函數

```

 std::uninitialized_default_construct_n(ptr, 10) ;
 std::destroy_n(ptr, 10) ;
 }

```

## 9.11.3 uninitialized\_value\_construct

```

template <class ForwardIterator>
void uninitialized_value_construct(
 ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Size>
ForwardIterator uninitialized_value_construct_n(
 ForwardIterator first, Size n);

```

使方 `uninitialized_default_construct` 同。、`uninitialized_value_construct` 初期化值初期化。

## 9.11.4 uninitialized\_copy

```

template <class InputIterator, class ForwardIterator>
ForwardIterator
uninitialized_copy(InputIterator first, InputIterator last,
 ForwardIterator result);

template <class InputIterator, class Size, class ForwardIterator>
ForwardIterator
uninitialized_copy_n(InputIterator first, Size n,
 ForwardIterator result);

```

`[first, last)` 範圍、`first` 個範圍值、`result` 指未初期化構築。

```

int main()
{
 std::vector<std::string> input(10, "hello") ;

 std::shared_ptr<void> raw_ptr
 (::operator new(sizeof(std::string) * 10),
 [](void * ptr){ ::operator delete(ptr) ; }) ;

 std::string * ptr = static_cast<std::string *>(raw_ptr.get()) ;
}

```

## 9.11 内存管理

```

 std::uninitialized_copy_n(std::begin(input), 10, ptr);
 std::destroy_n(ptr, 10);
}

```

## 9.11.5 uninitialized\_move

```

template <class InputIterator, class ForwardIterator>
ForwardIterator
uninitialized_move(InputIterator first, InputIterator last,
 ForwardIterator result);

template <class InputIterator, class Size, class ForwardIterator>
pair<InputIterator, ForwardIterator>
uninitialized_move_n(InputIterator first, Size n,
 ForwardIterator result);

```

使用方 `uninitialized_copy` 同。详细见。

## 9.11.6 uninitialized\_fill

```

template <class ForwardIterator, class T>
void uninitialized_fill(
 ForwardIterator first, ForwardIterator last,
 const T& x);

template <class ForwardIterator, class Size, class T>
ForwardIterator uninitialized_fill_n(
 ForwardIterator first, Size n,
 const T& x);

```

`[first, last)` 范围、`first` 后 `n` 个范围未初始化、`x` 为实引数 `x` 与 `T` 类型。

## 9.11.7 destroy

```

template <class T>
void destroy_at(T* location);

location->~T() 呼出。

```

## 第 9 章 其他標準庫

```
template <class ForwardIterator>
void destroy(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Size>
ForwardIterator destroy_n(ForwardIterator first, Size n);

[first, last) 範圍、first n 個範圍 destroy_at 呼出。
```

## 9.12 shared\_ptr::weak\_type

C++17 的 shared\_ptr 的 weak\_type 型名追加。shared\_ptr 对 weak\_ptr 的 typedef 名。

```
namespace std {

template < typename T >
class shared_ptr
{
 using weak_type = weak_ptr<T> ;
} ;

}
```

使用方：

```
template < typename Shared_ptr >
void f(Shared_ptr sptr)
{
 // C++14
 auto wptr1 = std::weak_ptr<
 typename Shared_ptr::element_type
 >(sptr) ;

 // C++17
 auto wptr2 = typename Shared_ptr::weak_type(sptr) ;
}
```



## 9.13 void\_t

`<type_traits>` は定義済み `void_t` 以下に定義済み。

```
namespace std {

template < class ... >
using void_t = void ;

}
```

`void_t` は任意個の型を実引数として取り `void` 型。性質は `std::is_void_v`、標準に追加。

## 9.14 bool\_constant

`<type_traits>` は `bool_constant` を追加。

```
template <bool B>
using bool_constant = integral_constant<bool, B>;

using true_type = bool_constant<true>;
using false_type = bool_constant<false>;
```

今 `integral_constant` は `bool` 型が必要な場面、C++17 以降 `std::true_type` と `std::false_type` を書く。

## 9.15 type\_traits

C++17 は `<type_traits>` の機能を追加。

### 9.15.1 変数テンプレート版 traits

C++17 は、既存の `traits` を変数テンプレート版 `_v` 版を追加。

例、`is_integral<T>::value` を `is_integral_v<T>` と書く。

## 第 9 章 其他標準庫

```
template < typename T >
void f(T x)
{
 constexpr bool b1 = std::is_integral<T>::value ; // 常數
 constexpr bool b2 = std::is_integral_v<T> ; // 變數
 constexpr bool b3 = std::is_integral<T>{} ; // operator bool()
}
```

## 9.15.2 論理演算 traits

C++17 提供了 `conjunction`, `disjunction`, `negation` 等追加的。這些是論理積、論理和、否定等手輕的。這些 traits。

## conjunction : 論理積

```
template<class... B> struct conjunction;
```

`conjunction<B1, B2, ..., BN>` 實引數 `B1, B2, ..., BN` 論理積適用。 `conjunction` 實引數 `Bi` 對 `bool(Bi::value)` 為 `false` 最初型基本保持、最後 `BN` 基本保持。

```
int main()
{
 using namespace std ;

 // is_void<void> 基本保持
 using t1 =
 conjunction<
 is_same<int, int>, is_integral<int>,
 is_void<void> > ;

 // is_integral<double> 基本保持
 using t2 =
 conjunction<
 is_same<int, int>, is_integral<double>,
 is_void<void> > ;
}
```

**disjunction : 論理和**

```
template<class... B> struct disjunction;
```

`disjunction<B1, B2, ..., BN>` 実引数 `B1, B2, ..., BN` の論理和に適用。 `disjunction` 実引数 `Bi` に対して `bool(Bi::value)` が `true` の最初型が基本型として保持、その後 `BN` が基本型として保持。

```
int main()
{
 using namespace std ;

 // is_same<int,int>が基本型として保持
 using t1 =
 disjunction<
 is_same<int, int>, is_integral<int>,
 is_void<void> > ;

 // is_void<int>が基本型として保持
 using t2 =
 disjunction<
 is_same<int, double>, is_integral<double>,
 is_void<int> > ;
}
```

**negation : 否定**

```
template<class B> struct negation;
```

`negation<B>` `B` の否定に適用。 `negation` が基本型として `bool_constant<!bool(B::value)>` を保持。

```
int main()
{
 using namespace std ;

 // false
 constexpr bool b1 = negation< true_type >::value ;
 // true
 constexpr bool b2 = negation< false_type >::value ;
}
```

## 第9章 他標準ライブラリ

9.15.3 `is_invocable` : 呼び出し可能を確認する traits

```

template <class Fn, class... ArgTypes>
struct is_invocable;

template <class R, class Fn, class... ArgTypes>
struct is_invocable_r;

template <class Fn, class... ArgTypes>
struct is_nothrow_invocable;

template <class R, class Fn, class... ArgTypes>
struct is_nothrow_invocable_r;

```

`is_invocable` は実引数と型 `Fn` の関数型 `ArgTypes` の展開の結果、実引数関数呼出しの結果、戻り値 `R` の暗黙変換を確認する traits。呼出しの結果 `true_type`, `false_type` を基本型として持つ。

`is_invocable` は関数呼出しの結果戻り値型を問わない。

`is_invocable_r` は呼出し可能性に加え、関数呼出しの結果戻り値型 `R` の暗黙変換を確認する。

`is_nothrow_invocable` は `is_nothrow_invocable_r` の、関数呼出し (戻り値型 `R` の暗黙変換) の無例外保証を確認する。

```

int f(int, double);

int main()
{
 // true
 constexpr bool b1 =
 std::is_invocable< decltype(&f), int, double >{} ;
 // true
 constexpr bool b2 =
 std::is_invocable< decltype(&f), int, int >{} ;

 // false
 constexpr bool b3 =
 std::is_invocable< decltype(&f), int >{} ;
 // false
 constexpr bool b4 =

```

```

std::is_invocable< decltype(&f), int, std::string >{} ;

// true
constexpr bool b5 =
 std::is_invocable_r< int, decltype(&f), int, double >{} ;
// false
constexpr bool b6 =
 std::is_invocable_r< double, decltype(&f), int, double >{} ;
}

```

#### 9.15.4 has\_unique\_object\_representations : 同値の内部表現が同一か確認する traits

```

template <class T>
struct has_unique_object_representations ;

```

has\_unique\_object\_representations<T> 型 T 型が 2 つの異なる内部表現を持つ場合、true を返す。

false を返す例として、padding (padding) を呼ぶ調整値の表現が影響を及ぼす領域を持つ場合、調整値の同値が影響を及ぼす、false を返す。

以下に X を、

```

struct X
{
 std::uint8_t a ;
 std::uint32_t b ;
} ;

```

実装、4 の必要、の本当以下に X を、

```

struct X
{
 std::uint8_t a ;

 std::byte unused_padding[3] ;

 std::uint32_t b ;
} ;

```

## 第9章 他標準ライブラリ

場合、`unused_padding` 値の意味、`x` 同値比較用。場合、`std::has_unique_representations_v<X>` `false`。

9.15.5 `is_nothrow_swappable` : 無例外 `swap` 可能を確認する traits

```
template <class T>
struct is_nothrow_swappable;

template <class T, class U>
struct is_nothrow_swappable_with;
```

`is_nothrow_swappable<T>` `T` 型 `swap` 例外投 `true` 返。  
`is_nothrow_swappable_with<T, U>` `T` 型 `U` 型相互 `swap` 例外投 `true` 返。

## 9.16 コンテナで不完全型のサポート

注意：説明上級者向。

C++17 以下 `std::vector` 合法。挙動 C++14 実装依存。

```
struct X
{
 std::vector<X> v ;
 std::list<X> l ;
 std::forward_list<X> f ;
};
```

定義終了 `}` 持完全型。注入名、定義中完全型。不完全型要素型指定場合挙動、C++14 規定。

C++17 `vector`, `list`, `forward_list` 限、要素型一時的不完全型許。実際使用際完全型。

9.17 `emplace` の戻り値

C++17 `emplace_front`/`emplace_back`, `queue` `stack` `emplace` 構築要素返変更。

例として、C++14 以下で以下のようなコードを実行すると、

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

 v.emplace_back(0) ; // void
 int value = v.back() ;
}
```

以下のようなコードを実行すると。

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

 int value = v.emplace_back(0) ;
}
```

## 9.18 map と unordered\_map の変更

map と unordered\_map の try\_emplace と insert\_or\_assign の 2 つの関数関入関数。関数関数関数関数 multi\_map と unordered\_multi\_map の追加関数関数。

### 9.18.1 try\_emplace

```
template <class... Args>
pair<iterator, bool>
try_emplace(const key_type& k, Args&&... args);

template <class... Args>
iterator
try_emplace(
 const_iterator hint,
 const key_type& k, Args&&... args);
```

従来は emplace の、関数関数関数関数要素関数存在関数関数場合、要素関数 args の emplace 構築関数関数追加関数関数。関数、関数関数関数関数要素関数存在関数関数場合、要素関数追加関数関数。要素関数追加関数関数関数関数、args の関数関数関数関数関数関数関数実装定義

## 第9章 他標準ライブラリ

例9.18.1。

```
int main()
{
 std::map< int, std::unique_ptr<int> > m ;

 // 要素が存在する
 m[0] = nullptr ;

 auto ptr = std::make_unique<int>(0) ;
 // emplace 失敗
 auto [iter, is_emplaced] = m.emplace(0, std::move(ptr)) ;

 // 結果実装が異なる
 // ptr が nullptr かどうか
 bool b = (ptr != nullptr) ;
}
```

この場合、実際 `map` に要素を追加する際、`ptr` が nullptr かどうかを確認する。

例9.18.2、C++17 の `map`、要素を追加する際に `args` が nullptr かどうかを確認する。また `try_emplace` を追加する。

```
int main()
{
 std::map< int, std::unique_ptr<int> > m ;

 // 要素が存在する
 m[0] = nullptr ;

 auto ptr = std::make_unique<int>(0) ;
 // emplace 失敗
 auto [iter, is_emplaced] = m.emplace(0, std::move(ptr)) ;

 // true かどうかを確認
 // ptr が nullptr かどうか
 bool b = (ptr != nullptr) ;
}
```

9.18.2 `insert_or_assign`



```
template <class M>
pair<iterator, bool>
insert_or_assign(const key_type& k, M&& obj);
```

```
template <class M>
iterator
insert_or_assign(
 const_iterator hint,
 const key_type& k, M&& obj);
```

insert\_or\_assign 関数 key が連想コンテナの要素に存在する場合要素を代入し、存在しない場合要素を追加する。operator [] が違えば、要素を代入し追加する。戻り値は pair の bool 成分が true なら代入、false なら追加。

```
int main()
{
 std::map< int, int > m ;
 m[0] = 0 ;

 {
 // 代入
 // is_inserted が false
 auto [iter, is_inserted] = m.insert_or_assign(0, 1) ;
 }

 {
 // 追加
 // is_inserted が true
 auto [iter, is_inserted] = m.insert_or_assign(1, 1) ;
 }
}
```

## 9.19 連想コンテナへの splice 操作

C++17 から、連想コンテナ（非順序連想コンテナ）の splice 操作が追加された。

対象コンテナは map, set, multimap, multiset, unordered\_map, unordered\_set, unordered\_multimap, unordered\_multiset である。

splice 操作は list コンテナが提供する splice 操作と、互換性のある list コンテナ

## 第 9 章 其他標準容器

list 要素の所有権を別容器に移動する機能。

```
int main()
{
 std::list<int> a = {1,2,3} ;
 std::list<int> b = {4,5,6} ;

 a.splice(std::end(a), b, std::begin(b)) ;

 // a {1,2,3,4}
 // b {5,6}

 b.splice(std::end(b), a) ;

 // a {}
 // b {5,6,1,2,3,4}
}
```

連想容器、順序容器の仕組を用いて、list 要素の所有権を別容器に移動する外に出る仕組、splice 操作を行う。

## 9.19.1 merge

順序連想容器、非順序連想容器、関数 merge を持つ。順序容器 a, b を互換、a.merge(b) 、順序容器 b 要素の所有権を a に移す。

```
int main()
{
 std::set<int> a = {1,2,3} ;
 std::set<int> b = {4,5,6} ;

 // b 要素を a に移す
 a.merge(b) ;

 // a {1,2,3,4,5,6}
 // b {}
}
```

重複、重複を許す場合、値重複の場合、重複要素を移動する。

```
int main()
{
 std::set<int> a = {1,2,3} ;
 std::set<int> b = {1,2,3,4,5,6} ;

 a.merge(b) ;

 // a {1,2,3,4,5,6}
 // b {1,2,3}

}
```

merge は移動要素の指針を移動後、要素移動後妥当にする。指針、所属配列も変化する。

```
int main()
{
 std::set<int> a = {1,2,3} ;
 std::set<int> b = {4,5,6} ;

 auto iterator = std::begin(b) ;
 auto pointer = &*iterator ;

 a.merge(b) ;

 // iterator pointer 妥当
 // 要素 a 所属

}
```

### 9.19.2 ノードハンドル

配列、要素構築、要素構築後所有権切り離し機能。

型、各型名 `node_type`。 `std::set<int>` 型、 `std::set<int>::node_type`。

以下保持。

```
class node_handle
{
public :
```

## 第 9 章 其他標準容器

```

// 型名
using value_type = ... ; // set 限定、要素型
using key_type = ... ; // map 限定、型
using mapped_type = ... ; // map 限定、型
using allocator_type = ... ; // 型

// 型
// 型
// 代入演算子

// 値
value_type & value() const ; // set 限定
key_type & key() const ; // map 限定
mapped_type & mapped() const ; // map 限定

// 型
allocator_type get_allocator() const ;

// 空判定
explicit operator bool() const noexcept ;
bool empty() const noexcept ;

void swap(node_handle &) ;
};

set 関数 value 値得。

int main()
{
 std::set<int> c = {1,2,3} ;

 auto n = c.extract(2) ;

 // n.value() == 2
 // c {1,3}
}

map 関数 key mapped 値得。

int main()
{

```

## 9.19 連想配列の splice 操作

```

std::map< int, int > m =
{
 {1,1}, {2,2}, {3,3}
} ;

auto n = m.extract(2) ;

// n.key() == 2
// n.mapped() == 2
// m ≒ {{1,1},{3,3}}

}

```

この操作は、元の配列から指定された位置から切り離し、所有権を得る。切り離された配列は、元の配列から独立し、元の配列が破棄された際に破棄される。切り離された配列は、元の配列が破棄された際に破棄される。切り離された配列は、元の配列が破棄された際に破棄される。

```

int main()
{
 std::set<int>::node_type n ;

 {
 std::set<int> c = { 1,2,3 } ;
 // 所有権を移動
 n = c.extract(std::begin(c)) ;
 // c が破棄される
 }

 // OK
 // 切り離された配列の所有権を移動する
 int x = n.value() ;

 // n が破棄される
}

```

## 9.19.3 extract : ノードハンドルの取得

```

node_type extract(const_iterator position) ;
node_type extract(const key_type & x) ;

```

## 第 9 章 其他標準容器

連想容器非順序連想容器關數 `extract`、取得關數。

關數 `extract(position)`、`position` 指要素、要素除去、要素所有容器返。

```
int main()
{
 std::set<int> c = {1,2,3} ;

 auto n1 = c.extract(std::begin(c)) ;

 // c {2,3}

 auto n2 = c.extract(std::begin(c)) ;

 // c {3}

}
```

關數 `extract(x)`、`x` 存在場合、要素除去、要素所有容器返。存在場合、空容器返。

```
int main()
{
 std::set<int> c = {1,2,3} ;

 auto n1 = c.extract(1) ;
 // c {2,3}

 auto n2 = c.extract(2) ;
 // c {3}

 // 4 存在
 auto n3 = c.extract(4) ;
 // c {3}
 // n3.empty() == true
}
```

重複許場合、複數 1 所有權解放。

```
int main()
```

```

{
 std::multiset<int> c = {1,1,1} ;
 auto n = c.extract(1) ;
 // c {1,1}
}

```

#### 9.19.4 insert : ノードハンドルから要素の追加

```

// 重複許す容器の場合
insert_return_type insert(node_type&& nh);
// 重複許す multi 容器の場合
iterator insert(node_type&& nh);

// 付く insert
iterator insert(const_iterator hint, node_type&& nh);

```

関数 insert は実引数渡し、参照移動、所有権移動。

```

int main()
{
 std::set<int> a = {1,2,3} ;
 std::set<int> b = {4,5,6} ;

 auto n = a.extract(1) ;

 b.insert(std::move(n)) ;

 // n.empty() == true
}

```

空の場合、何起。

```

int main()
{
 std::set<int> c ;
 std::set<int>::node_type n ;

 // 何起
 c.insert(std::move(n)) ;
}

```

## 第9章 其他標準容器

重複許容容器、重複存在等値所有容器の insert 失敗。

```
int main()
{
 std::set<int> c = {1,2,3} ;

 auto n = c.extract(1) ;
 c.insert(1) ;

 // 失敗
 c.insert(std::move(n)) ;
}
```

第一引数 hint を受取る insert の挙動、従来 insert と同じ。要素 hint 直前追加の償却定数時間処理終了。

実引数を受取る insert の戻り値型、重複許容 multi 場合 iterator。重複許容場合、insert\_return\_type。

multi 場合、戻り値追加の要素指。

```
int main()
{
 std::multiset<int> c { 1,2,3 } ;

 auto n = c.extract(1) ;

 auto iter = c.insert(n) ;

 // c {1,2,3}
 // iter 1 指
}
```

重複許容場合、insert\_return\_type の戻り値型。set<int> 場合、set<int>::insert\_return\_type。

insert\_return\_type の具体的な名前規格上規定。insert\_return\_type 以下保持型。

```
struct insert_return_type
{
```



```

 iterator position ;
 bool inserted ;
 node_type node ;
 } ;

```

position へ insert した要素の所有権移動は追加した要素の指針、inserted は要素の追加が行われた場合 true 又は bool, node は要素の追加失敗時の所有権移動の指針。

insert は渡した空の、inserted は false, position は end(), node は空。

```

int main()
{
 std::set<int> c = {1,2,3} ;
 std::set<int>::node_type n ; // 空

 auto [position, inserted, node] = c.insert(std::move(n)) ;

 // inserted == false
 // position == c.end()
 // node.empty() == true
}

```

insert は成功、inserted は true, position は追加した要素の指針、node は空。

```

int main()
{
 std::set<int> c = {1,2,3} ;
 auto n = c.extract(1) ;

 auto [position, inserted, node] = c.insert(std::move(n)) ;

 // inserted == true
 // position == c.find(1)
 // node.empty() == true
}

```

insert は失敗、同一の値が存在、inserted は false, node は insert 呼出前の値、position は

## 第9章 他標準ライブラリ

本章中で追加された等要素指針。insert 渡した要素が未規定値の場合。

```
int main()
{
 std::set<int> c = {1,2,3} ;
 auto n = c.extract(1) ;
 c.insert(1) ;

 auto [position, inserted, node] = c.insert(std::move(n)) ;

 // n 未規定値
 // inserted == false
 // node insert(std::move(n)) 呼出前 n 値
 // position == c.find(1)
}
```

規格の場合 n 値は規定通り、最終的に実装済み、n node 空、その後状態。

## 9.19.5 ノードハンドルの利用例

典型的な使い方以下。

## 再確保、一部要素別移動

```
int main()
{
 std::set<int> a = {1,2,3} ;
 std::set<int> b = {4,5,6} ;

 auto n = a.extract(1) ;
 b.insert(std::move(n)) ;
}
```

## 寿命超要素存続

```
int main()
{
 std::set<int>::node_type n ;
```

```

{
 std::set<int> c = {1,2,3} ;
 n = c.extract(1) ;
 // c 破棄
}

// 破棄後存続
int value = n.value() ;
}

```

### map 変更

map 変更。変更、元要素削除、新要素追加が必要。動的の解放確保必要。

使用、既存要素対、所有権 map 引剥上、変更、一度 map 差戻。

```

int main()
{
 std::map< std::string, std::string > m =
 {
 {"cat", "meow"},
 {"DOG", "bow"}, // 間違変更
 {"cow", "moo"}
 } ;

 // 所有権引剥
 auto n = m.extract("DOG") ;
 // 変更
 n.key() = "dog" ;
 // 差戻
 m.insert(std::move(n)) ;
}

```

## 9.20 コンテナアクセス関数

<iterator> 関数、関数版 size, empty, data 追加。関数、関数 size, empty, data

## 第9章 其他標準庫

呼出。

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

 std::size(v) ; // v.size()
 std::empty(v) ; // v.empty()
 std::data(v) ; // v.data()
}
```

関数配列 `std::initializer_list<T>` 使用。

```
int main()
{
 int a[10] ;

 std::size(a) ; // 10
 std::empty(a) ; // 常 false
 std::data(a) ; // a
}
```

## 9.21 clamp

```
template<class T>
constexpr const T&
clamp(const T& v, const T& lo, const T& hi);
template<class T, class Compare>
constexpr const T&
clamp(const T& v, const T& lo, const T& hi, Compare comp);
```

`<algorithm>` 追加 `clamp(v, lo, hi)` 値 `v` `lo` 小場合 `lo`、`hi` 高場合 `hi`、以外場合 `v` 返。

```
int main()
{
 std::clamp(5, 0, 10) ; // 5
 std::clamp(-5, 0, 10) ; // 0
 std::clamp(50, 0, 10) ; // 10
}
```

comp 実引数を取 clamp comp 値比較使  
clamp 浮動小数点数使、NaN 渡。

## 9.22 3次元 hypot

```
float hypot(float x, float y, float z);
double hypot(double x, double y, double z);
long double hypot(long double x, long double y, long double z);
```

<cmath> 3次元 hypot 追加。

戻値：

$$\sqrt{x^2 + y^2 + z^2}$$

## 9.23 atomic<T>::is\_always\_lock\_free

```
template < typename T >
struct atomic
{
 static constexpr bool is_always_lock_free = ... ;
};
```

C++17 <atomic> 追加 atomic<T>::is\_always\_lock\_free、atomic<T> 実装実行時保証場合、true static constexpr bool 型。

atomic、他 bool 返関数 is\_lock\_free、実行時判定。is\_always\_lock\_free 時判定。

## 9.24 scoped\_lock : 可変長引数 lock\_guard

std::scoped\_lock <T ...> 可変長引数版 lock\_guard。

```
int main()
{
 std::mutex a, b, c, d ;
```

## 第9章 其他標準庫組件

```

 {
 // a,b,c,d を lock
 std::scoped_lock l(a, b, c, d);
 // a,b,c,d を unlock
 }
}

```

`std::scoped_lock` は複数の互排排他セクションを同時に取得、解放するための方法として関数 `lock` を呼び出し、関数 `unlock` を呼び出す。

9.25 `std::byte`

C++17 は表現型として `std::byte` を追加した。この言語の一部、別項で詳しく説明を行う。

## 9.26 最大公約数 (gcd) と最小公倍数 (lcm)

C++17 は `<numeric>` に最大公約数 (gcd) と最小公倍数 (lcm) を追加した。

```

int main()
{
 int a, b ;

 while(std::cin >> a >> b)
 {
 std::cout
 << "gcd: " << gcd(a,b)
 << "\nlcm: " << lcm(a,b) << '\n' ;
 }
}

```

## 9.26.1 gcd : 最大公約数

```

template <class M, class N>
constexpr std::common_type_t<M,N> gcd(M m, N n)
{

```

## 9.26 最大公約数 (gcd) と最小公倍数 (lcm)

```

 if (n == 0)
 return m ;
 else
 return gcd(n, std::abs(m) % std::abs(n)) ;
}

```

`gcd(m, n)` は `m` と `n` の最大公約数を返す。0 以外の場合、 $|m|$  と  $|n|$  の最大公約数 (Greatest Common Divisor) を返す。

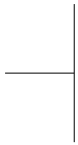
## 9.26.2 lcm : 最小公倍数

```

template <class M, class N>
constexpr std::common_type_t<M,N> lcm(M m, N n)
{
 if (m == 0 || n == 0)
 return 0 ;
 else
 return std::abs(m) / gcd(m, n) * std::abs(n) ;
}

```

`lcm(m,n)` は、`m` と `n` の最小公倍数を返す。0 以外の場合、 $|m|$  と  $|n|$  の最小公倍数 (Least Common Multiple) を返す。





## 第10章

# ファイルシステム

この章では、C++17 標準で定義された `<filesystem>` 標準ライブラリについて、その属性と使用法について説明する。

一般に「ファイルシステム」は、Linux の `ext4`, Microsoft Windows の `FAT` や `NTFS`, Apple Mac の `HFS+` や `APFS` など、様々なファイルシステムが存在する。C++ 標準では、これらのファイルシステムを抽象化し、共通のインターフェースを提供する。具体的には、ファイルの属性、パス、操作などを扱う。また、ファイルシステムの抽象化は、ファイルの属性、パス、操作などを扱う。また、ファイルシステムの抽象化は、ファイルの属性、パス、操作などを扱う。

この章では、ファイルシステムの基本用語、単一通常ファイル、ディレクトリ、シンボリックリンク、FIFO（名前付きパイプ）、特殊ファイルなどについて説明する。

本書では、ファイルシステムの詳細な解説を行う。また、ファイルシステムの抽象化は、ファイルの属性、パス、操作などを扱う。また、ファイルシステムの抽象化は、ファイルの属性、パス、操作などを扱う。

### 10.1 名前空間

この章では、`std::filesystem` 名前空間について説明する。この名前空間は、ファイルシステムの操作を行うための関数と型を提供する。

```
int main()
{
 std::filesystem::path p("/bin") ;
}
```





## 第 10 章 文件系统

```

 auto path1 = e.path1() ; // 第一引数
 auto path2 = e.path2() ; // 第二引数
 auto error_code = e.code() ; // error_code

 std::cout
 << "error number: " << error_code.value ()
 << "\nerror message: " << error_code.message()
 << "\npath1: " << path1
 << "\npath2: " << path2 << '\n' ;
 }
}

```

filesystem\_error 以下関数を実行する。

```

namespace std::filesystem {
 class filesystem_error : public system_error {
 public:
 // 第一引数
 const path& path1() const noexcept;
 // 第二引数
 const path& path2() const noexcept;
 // 内容人間読 null 終端文字列返
 const char* what() const noexcept override;
 };
}

```

## 10.4.2 非例外

filesystem\_error 関数、std::error\_code & 型実引数を取関数、以下関数を実行。

- OS 操作が発生場合、error\_code & 型実引数内容を設定。場合、error\_code & 型実引数対関数 clear() を呼。

```

int main()
{
 using namespace std::filesystem ;

 // 名前同名前
 path file("foobar.txt") ;
}

```

## 10.5 path : ファイルパス文字列クラス

```

std::ofstream{ file } ;
std::error_code error_code;
copy_file(file, file, error_code) ;

if (error_code)
{ // 失敗した場合
 auto path1 = file ; // 第一引数
 auto path2 = file ; // 第二引数

 std::cout
 << "error number: " << error_code.value ()
 << "\nerror message: " << error_code.message()
 << "\npath1: " << path1
 << "\npath2: " << path2 << '\n' ;
}
}

```

## 10.5 path : ファイルパス文字列クラス

`std::filesystem::path` ファイルパス文字列表現クラス。文字列表現は C++ の `std::string` と類似、類似した文字列表現、別専用クラスとして。

`path` は以下機能を提供。

- ファイルパス文字列表現
- ファイルパス文字列操作

`path` はファイルパス文字列表現操作を提供、物理パス変更。

ファイルパス文字列表現は実装異なる。POSIX 環境は文字型 `char` 型 UTF-8 を表現、OS 多、Microsoft Windows は本書執筆現在、文字型 `wchar_t` UTF-16 を表現慣習。

、OS は大文字小文字区別、区別無視実装。

`path` は文字列差異吸収。

`path` は以下型名。

```
namespace std::filesystem {
```

## 第 10 章 文字列とファイル

```

class path {
public:
 using value_type = see below ;
 using string_type = basic_string<value_type>;
 static constexpr value_type preferred_separator = see below ;
} ;

```

`value_type` と `string_type` は `path` の内部で文字列を表現するために使用する文字列の型。 `preferred_separator` は、推奨の区切り文字。 POSIX 互換環境で `/` を用い、Microsoft Windows で `\` を使用する。

10.5.1 `path` : ファイルパスの文字列

文字列を表現。 C++ の文字列以下。

- `char`: UTF-8
- `wchar_t`: UTF-16
- `char`: UTF-8
- `char16_t`: UTF-16
- `char32_t`: UTF-32

`path::value_type` は文字型、文字列は文字列型。文字列型は文字列型に依存。 `path` は文字列型、 `path::value_type` は文字型。文字列型は文字列型に自動的に変換される。

```

int main()
{
 using namespace std::filesystem ;

 //
 path p1("/dev/null") ;
 //
 path p2(L"/dev/null") ;
 // UTF-16
 path p3(u"/dev/null") ;
 // UTF-32
 path p4(U"/dev/null") ;
}

```

## 10.5 path : 文字列

、文字列 渡 動。

C++ UTF-8 文字型 `char` 、 文字型 同 、型 区别 。、UTF-8 文字列 渡 、 認識 。

```
int main()
{
 using namespace std::filesystem ;

 // 解釈
 path p(u8"名");
}
```

、 UTF-8 場合、動 保証 移植性 低 。UTF-8 移植性 高 方法 使 場合、`u8path` 使 。

```
int main()
{
 using namespace std::filesystem ;

 // UTF-8 解釈
 // 実装 使 文字 変換
 path = u8path(u8"名");
}
```

`u8path(Source)` `Source` UTF-8 文字列 扱 、通常 文字列 渡 、 UTF-8 環境 問題 。

```
int main()
{
 using namespace std::filesystem ;

 // UTF-8 解釈
 // UTF-8 場合、問題
 path = u8path("名");
}
```

`u8path` 使 場合、文字列 必 UTF-8 。

## 第 10 章 字符串

環境、使用文字制限、特定文字列特別意味持予約語、移植性高作成当点注意。、環境大文字小文字區別、CON AUX 文字列特別意味持。

path 格納文字列取得方法、環境依存文字列表現方法差異、方法用意。

文字列以下 2 。

- 実装依存
- 汎用的標準

POSIX 準拠環境、同。POSIX 準拠環境、異持可能性。

、Microsoft Windows 、文字列区切文字 POSIX 準拠 / \ 使用。

関数 native c\_str 。

```
class path {
{
public :
 const string_type& native() const noexcept;
 const value_type* c_str() const noexcept;
} ;
```

path 内部使用実装依存文字列型返。

```
int main()
{
 using namespace std::filesystem ;

 path p = current_path() ;

 // 実装依存 basic_string 特殊化
 path::string_type str = p.native() ;

 // 実装依存文字型
```



## 10.5 path : 文字列

```
path::value_type const * ptr = p.c_str() ;
```

```
}
```

関数使移植性注意必要。

str 型 path::string\_type 、 ptr 型実装依存 path::value\_type const \* 。

path::value\_type path::string\_type 、 char wchar\_t, std::string std::wstring C++ 標準定義型可能性。

、 path::string\_type 変換関数 operator string\_type() 。

```
int main()
{
 using namespace std::experimental::filesystem ;

 auto p = current_path() ;

 // 暗黙型変換
 path::string_type str = p ;
}
```

path operator string\_type() 、 文字列型既存形式変換返。空白文字含、二重引用符囲文字列変換。

```
int main()
{
 using namespace std::filesystem ;

 path name("foo bar.txt") ;
 std::basic_ofstream<path::value_type> file(name) ;
 file << "hello" ;
}
```

文字列 string, wstring, u16string, u32string 変換取得関数以下。

```
class path {
public :
 std::string string() const;
 std::wstring wstring() const;
 std::string u8string() const;
```

## 第 10 章 ファイルパス

```

 std::u16string u16string() const;
 std::u32string u32string() const;
 };

```

string、wstring関数 string、wstring、u8string、u16string、u32string std::string, u8string UTF-8 string std::string 返す。

```

int main()
{
 using namespace std::filesystem ;

 path name("hello.txt") ;
 std::ofstream file(name.string()) ;
 file << "hello" ;
}

```

generic\_string() 系 generic\_string() 系関数。

```

class path {
public :
 std::string generic_string() const;
 std::wstring generic_wstring() const;
 std::string generic_u8string() const;
 std::u16string generic_u16string() const;
 std::u32string generic_u32string() const;
};

```

使用方 string() 系関数同。

文字列文字型文字列環境異、移植性高書注意必要。

現実的、POSIX 準拠環境、文字型 char, 文字列型 std::string, UTF-8。

Microsoft Windows Win32 MSVC POSIX 準拠、本書執筆時点、歴史的経緯、文字型 wchar\_t, 文字列型 std::wstring, UTF-16。

## 10.5.2 ファイルパスの操作

path 文字列操作提供。std::string 違、find substr 操作提供、文字列特化操

## 10.5 path : 字符串文字列

作提供。

operator /, operator /= 区切字符串追加行。

```
int main()
{
 using namespace std::filesystem ;

 path p("/") ;

 // "/usr"
 p /= "usr" ;
 // "/usr/local/include"
 p = p / "local" / "include" ;
}
```

operator += 单字符串結合行。

```
int main()
{
 using namespace std::filesystem ;

 path p("/") ;

 // "/usr"
 p += "usr" ;
 // "/usrlocal"
 p += "local" ;
 // "/usrlocalinclude"
 p += "include" ;
}
```

operator / 違、operator + 存在。

其他、path 字符串对操作提供。  
以下一例。

```
int main()
{
 using namespace std::filesystem ;

 path p("/home/cpp/src/main.cpp") ;
```

## 第 10 章 文字列処理

```

 // "main.cpp"
 path filename = p.filename() ;
 // "main"
 path stem = p.stem() ;
 // ".cpp"
 path extension = p.extension() ;
 // "/home/cpp/src/main.o"
 p.replace_extension("o") ;
 // "/home/cpp/src/"
 p.remove_filename() ;
}

```

`path` 文字列に対して文字列処理を提供。名前抽出処理、拡張子抽出処理、拡張子変換処理。

## 10.6 file\_status

`file_status` 保持。文字列指定取得方法別途、方法毎物理状態発生。`file_status` 情報保持、役割果。

`file_status` 物理状態変更。`file_status` `status(path)` `status(path, error_code)` 取得。、`directory_entry` 関数 `status()` 取得。種類表 `enum` 型 `file_type`、通常種類表。種類表、種類表 `enum` 型 `perms`、所有者他人対読込、書込、実行権限表。値 `POSIX` 値同。取得関数以下。

```

class file_type {
public :
 file_type type() const noexcept;
 perms permissions() const noexcept;
} ;

```

以下例程使用。

```
int main()
{
 using namespace std::filesystem ;

 directory_iterator iter(".", end ;

 int regular_files = 0 ;
 int execs = 0 ;

 std::for_each(iter, end, [&](auto entry)
 {
 auto file_status = entry.status() ;
 // is_regular_file(file_status) 可
 if (file_status.type() == file_type::regular)
 ++regular_files ;

 constexpr auto exec_bits =
 perms::owner_exec | perms::group_exec | perms::others_exec ;

 auto permissions = file_status.permissions() ;
 if ((permissions != perms::unknown) &&
 (permissions & exec_bits) != perms::none)
 ++execs ;
 }) ;

 std::cout
 << "Current directory has "
 << regular_files
 << " regular files.\n" ;
 << execs
 << " files are executable.\n" ;
 }
```

例程中、`regular_files`、`execs` 通常整数、実行可能数表示。

例程中 `perms` enum 型、不明場合 `perms::unknown`。値 `0xFFFF` 演算場合注意必要。

以外 `perms` 値 POSIX 準拠、`perms` scoped enum 型。



```

 directory_iterator i1(p) ;
 directory_entry e2 = *i1 ;

 recursive_directory_iterator i2(p) ;
 directory_entry e3 = *i2 ;
}

```

`directory_entry` 情報取得関数、同機能関数、`directory_entry` 使用、情報、同対、物理変更回数情報取得行効率の。

```

int main()
{
 using namespace std::filesystem ;

 directory_entry entry("/home/cpp/foo") ;

 // 存在確認
 bool b = entry.exists() ;

 // "/home/cpp/foo"
 path p = entry.path() ;
 file_status s = entry.status() ;

 // 取得
 std::uintmax_t size = entry.file_size() ;

 {
 std::ofstream foo(entry.path()) ;
 foo << "hello" ;
 }

 // 物理情報更新
 entry.refresh() ;
 // 一度取得
 size = entry.file_size() ;

 // 情報取得
 // "/home/cpp/bar"
 // 置換 refresh() 呼出

```

## 第 10 章 文件系统

```
entry.replace_filename("bar") ;
}
```

`directory_entry` オブジェクトを用いて、自動的・物理的にファイルの変更を追跡し、最新の情報を取得し、明示的に `refresh` を呼出す必要がある。

10.8 `directory_iterator`

`directory_iterator` は、ディレクトリの下に存在するファイルやディレクトリを、`directory_entry` の形式で列挙する。

ディレクトリ、ディレクトリの下に存在するファイルやディレクトリを列挙する以下に示す。

```
int main()
{
 using namespace std::filesystem ;
 directory_iterator iter(".", end) ;
 std::copy(iter, end,
 std::ostream_iterator<path>(std::cout, "\n")) ;
}
```

`directory_iterator` は `path` を渡す、ディレクトリの下に最初に見られる `directory_entry` を返す。ディレクトリを指定するディレクトリの下に存在する場合、終端を返す。

`directory_iterator` はディレクトリの終端を返す。終端はディレクトリに存在しない。

`directory_iterator::value_type` は `directory_entry` の、ディレクトリへの入力となる。

`directory_iterator` はディレクトリ (.) の親ディレクトリ (..) を列挙する。

`directory_iterator` はディレクトリの下に存在するファイルやディレクトリの順序を列挙する。未規定。

`directory_iterator` はディレクトリに存在する可能性のある、ディレクトリが存在するかどうかを返す。ディレクトリ、存在するディレクトリを返す。

`directory_iterator` はディレクトリを作成した後、物理的にディレクトリを返す。



`directory_iterator` の挙動は、`directory_options` の値によって変更され、反映される。ただし、`directory_iterator` の挙動は C++17 の標準規格の範囲外である。

`directory_iterator` の挙動は、`directory_options` の値によって指定される。また、`directory_options` の値は、`directory_iterator` の挙動を変更する。ただし、`directory_options` の値は、`directory_iterator` の挙動を変更する。ただし、`directory_options` の値は、`directory_iterator` の挙動を変更する。

### 10.8.1 エラー処理

`directory_iterator` は構築時にエラーが発生する可能性がある。例外として、`error_code` を受取る場合、`error_code` の値は、`error_code` の値に渡される。

```
int main()
{
 using namespace std::filesystem ;

 std::error_code err ;

 directory_iterator iter("this-directory-does-not-exist", err) ;

 if (err)
 {
 // エラー処理
 }
}
```

`recursive_directory_iterator` は構築時にエラーが発生する可能性がある。例外として、`error_code` を受取る場合、`recursive_directory_iterator` の関数 `increment` を呼ぶ。

```
int main()
{
 using namespace std::experimental::filesystem ;

 recursive_directory_iterator iter(".", end) ;

 std::error_code err ;

 for (; iter != end && !err ; iter.increment(err))
 {
 std::cout << *iter << "\n" ;
 }
}
```

第 10 章

```
 }

 if (err)
 {
 // 処理
 }
}
```

10.9 recursive\_directory\_iterator

recursive\_directory\_iterator 指定 下 存在 下 含 、 列挙 。使 方 directory\_iterator 同 。

```
int main()
{
 using namespace std::filesystem ;
 recursive_directory_iterator iter(".", end ;

 std::copy(iter, end,
 std::ostream_iterator<path>(std::cout, "\n")) ;
}
```

関数 options, depth, recursion\_pending, pop, disable\_recursion\_pending 对 呼 出 際 挙動 未定義 。

10.9.1 オプション

recursive\_directory\_iterator 実引数 directory\_options 型 scoped enum 値 取 、 挙動 変更 。directory\_options 型 enum 値 、以下 3 値 規定 。

| 名前                       | 意味   |
|--------------------------|------|
| none                     | 。 違反 |
| follow_directory_symlink | 中 列挙 |
| skip_permission_denied   | 違反   |

## 10.9 recursive\_directory\_iterator

recursive\_directory\_iterator 取得のオプションは、none, follow\_directory\_symlink, skip\_permission\_denied, follow\_directory\_symlink | skip\_permission\_denied の 4 種類がある。

```
int main()
{
 using namespace std::filesystem ;
 recursive_directory_iterator
 iter("/", directory_options::skip_permission_denied), end ;

 std::copy(iter, end,
 std::ostream_iterator<path>(std::cout, "\n")) ;
}
```

follow\_directory\_symlink の、親ディレクトリを再訪する可能性がある場合、ディレクトリ終端に到達する可能性があるので注意が必要。

```
int main()
{
 using namespace std::filesystem ;

 // 自分自身を含むディレクトリに対してシンリンクを作成
 create_symlink(".", "foo") ;

 recursive_directory_iterator
 iter(".", directory_options::follow_directory_symlink), end ;

 // 再訪、終了
 std::copy(iter, end, std::ostream_iterator<path>(std::cout)) ;
}
```

recursive\_directory\_iterator は現在 directory\_options を得る関数、関数 options を呼ぶ。

```
class recursive_directory_iterator {
public :
 directory_options options() const ;
} ;
```

|  |  |
|--|--|
|  |  |
|  |  |

```

recursive_directory_iterator iter("."), end ;

auto const p = canonical("b/a") ;

for (; iter != end ; ++iter)
{
 std::cout << *iter << '\n' ;

 if (canonical(iter->path()) == p)
 iter.pop() ;
}

```

標準出力に指図の順番に以下が出力される。

```

a
b
b/a
c
d

```

“b/a” に到達した時点で pop() を呼び出し、b 以下の再帰的列挙を中止し、親ディレクトリに戻す。

#### 10.9.4 recursion\_pending : 現在のディレクトリーの再帰をスキップ

disable\_recursion\_pending は現在ディレクトリの下で再帰的列挙が行われている機能を無効化する。

```

class recursive_directory_iterator {
public :
 bool recursion_pending() const ;
 void disable_recursion_pending() ;
} ;

```

recursion\_pending() は、直前ディレクトリで再帰的操作の後で disable\_recursion\_pending() を呼び出した場合、true を返す。そうでない場合は false を返す。

言い換えると、disable\_recursion\_pending() を呼び出した直後、ディレクトリで再帰的操作が行われている場合、recursion\_pending() は false を返す。

```

int main()

```

## 第 10 章 再帰的列挙

```

{
 using namespace std ;
 recursive_directory_iterator iter("."), end ;

 // true
 bool b1 = iter.recursion_pending() ;

 iter.disable_recursion_pending() ;
 // false
 bool b2 = iter.recursion_pending() ;

 ++iter ;
 // true
 bool b3 = iter.recursion_pending() ;

 iter.disable_recursion_pending() ;
 // false
 bool b4 = iter.recursion_pending() ;
}

```

現在 `recursive_directory_iterator` が指しているディレクトリを再帰的に再帰的列挙する場合、`iter.recursion_pending()` が `true` を返す場合、`iter.disable_recursion_pending()` を呼び出して再帰的列挙を停止し、`iter.recursion_pending()` が `false` を返す場合、`++iter` を呼び出して次のディレクトリに進む。このようにして、`iter.recursion_pending()` が `true` を返す場合、`iter.disable_recursion_pending()` を呼び出して再帰的列挙を停止し、`iter.recursion_pending()` が `false` を返す場合、`++iter` を呼び出して次のディレクトリに進む。

`iter.disable_recursion_pending()`、現在指しているディレクトリを再帰的に再帰的列挙する機能を提供する。

再帰的列挙、再帰的列挙のディレクトリ以下に再帰的に再帰的列挙する、再帰的列挙のディレクトリ以下に再帰的に再帰的列挙する環境の場合、

```

a
b
b/a
b/c
b/d
c
d

```

以下に再帰的列挙を実行する、

## 10.9 recursive\_directory\_iterator

```
int main()
{
 std::filesystem ;

 recursive_directory_iterator iter(".", end ;

 auto const p = canonical("b/a") ;

 for (; iter != end ; ++iter)
 {
 std::cout << *iter << '\n' ;

 if (iter->is_directory())
 iter.disable_recursion_pending() ;
 }
}
```

標準出力指XXXXXXXXXX順番以下XXXXXXXXX。

A blank coordinate plane with a horizontal x-axis and a vertical y-axis intersecting at the origin. The axes are represented by thin black lines.

必 `disable_recursion_pending()` 呼、再歸的列挙行、結果の動作 `directory_iterator` 同。

`disable_recursion_pending` 呼出、選択の再帰の列举。

## 第 10 章 ファイルシステム操作関数

## 10.10 ファイルシステム操作関数

## 10.10.1 ファイルパス取得

**current\_path**

```
path current_path();
path current_path(error_code& ec);
```

現在の作業ディレクトリ (current working directory) の絶対パスを返す。

**temp\_directory\_path**

```
path temp_directory_path();
path temp_directory_path(error_code& ec);
```

一時作業ディレクトリを作成する最適な一時作業ディレクトリ (temporary directory) の絶対パスを返す。

## 10.10.2 ファイルパス操作

**absolute**

```
path absolute(const path& p);
path absolute(const path& p, error_code& ec);
```

p の絶対パスを返す。p が指すディレクトリが存在しない場合挙動は未規定。

**canonical**

```
path canonical(const path& p, const path& base = current_path());
path canonical(const path& p, error_code& ec);
path canonical(const path& p, const path& base, error_code& ec);
```

存在するパス p が、相対パス、絶対パス、または親ディレクトリ (..) の存在する絶対パスを返す。

**weakly\_canonical**

```
path weakly_canonical(const path& p);
path weakly_canonical(const path& p, error_code& ec);
```



`path::relative(const path& p, error_code& ec)` 解决、正规化 `path` 并返回。相对 `path` 正规化 `path` 定义长 `path` 省略。

### relative

```
path relative(const path& p, error_code& ec);
path relative(const path& p, const path& base = current_path());
path relative(const path& p, const path& base, error_code& ec);
```

`base` 是 `p` 的相对 `path` 返回。

### proximate

```
path proximate(const path& p, error_code& ec);
path proximate(const path& p, const path& base = current_path());
path proximate(const path& p, const path& base, error_code& ec);
```

`base` 是 `p` 的相对 `path` 空 `path` 相对 `path` 返回。相对 `path` 空 `path` `p` 返回。

## 10.10.3 作成

### create\_directory

```
bool create_directory(const path& p);
bool create_directory(const path& p, error_code& ec) noexcept;
```

`p` 是指 `1` 作成。新作成作成場合 `true`、作成場合 `false` 返回。`p` 既存 `path` 指 `path` 新作成作成場合 `path` 作成場合 `path` 作成。单 `false` 返回。

```
bool create_directory(
 const path& p, const path& existing_p);

bool create_directory(
 const path& p, const path& existing_p,
 error_code& ec) noexcept;
```

新作成 `p` 既存 `existing_p` 同 `path`。

### create\_directories

## 第 10 章 標準ライブラリ

```
bool create_directories(const path& p);
bool create_directories(const path& p, error_code& ec) noexcept;
```

`create_directories` は `p` 中に存在しないディレクトリを作成する。  
以下に示すディレクトリ、`a` 以下のディレクトリ `b`、`c` を作成する。途中、`a`、`b` が存在する場合、作成は失敗する。

```
int main()
{
 using namespace std::filesystem;
 create_directories("./a/b/c");
}
```

戻り値、`error_code` が作成成功の場合 `true`、失敗の場合 `false`。

**create\_directory\_symlink**

```
void create_directory_symlink(
 const path& to, const path& new_symlink);
void create_directory_symlink(
 const path& to, const path& new_symlink,
 error_code& ec) noexcept;
```

`create_directory_symlink` は `to` を解決したディレクトリ `new_symlink` を作成する。  
一部 OS では、ディレクトリシンリンクを作成する際に明示的な区別が必要である。この場合、`create_symlink` ではなく `create_directory_symlink` を使用する。

一部 OS では、ディレクトリシンリンクを作成する際に明示的な区別が必要である。この場合、`create_symlink` ではなく `create_directory_symlink` を使用する。

**create\_symlink**

```
void create_symlink(
 const path& to, const path& new_symlink);
void create_symlink(
 const path& to, const path& new_symlink,
 error_code& ec) noexcept;
```

`create_symlink` は `to` を解決したディレクトリ `new_symlink` を作成する。

create\_hard\_link

```
void create_hard_link(
 const path& to, const path& new_hard_link);
void create_hard_link(
 const path& to, const path& new_hard_link,
 error_code& ec) noexcept;
```

to を解決したパス new\_hard\_link 作成。

10.10.4 コピー

copy\_file

```
bool copy_file(const path& from, const path& to);
bool copy_file(const path& from, const path& to,
 error_code& ec) noexcept;
bool copy_file(const path& from, const path& to,
 copy_options options);
bool copy_file(const path& from, const path& to,
 copy_options options,
 error_code& ec) noexcept;
```

from から to へコピー。

copy\_options は挙動変数 enum 型、以下 enum 値。

| 名前                 | 意味            |
|--------------------|---------------|
| none               | 、が存在する場合      |
| skip_existing      | 既存を上書き。エラーを報告 |
| overwrite_existing | 既存を上書き        |
| update_existing    | 既存を上書きする古い上書き |

copy

```
void copy(const path& from, const path& to);
void copy(const path& from, const path& to,
 error_code& ec) noexcept;
void copy(const path& from, const path& to,
```

## 第 10 章 標準ライブラリ

```

 copy_options options);
void copy(const path& from, const path& to,
 copy_options options,
 error_code& ec) noexcept;

```

`from` から `to` にコピーする。  
`copy_options` は挙動変換型 `enum` 型、以下 `enum` 値。

- 関数指定

| 名前                     | 意味 |
|------------------------|----|
| <code>none</code>      | 、  |
| <code>recursive</code> | 中身 |

- 関数指定

| 名前                         | 意味  |
|----------------------------|-----|
| <code>none</code>          | 、   |
| <code>copy_symlinks</code> | 。直接 |
| <code>skip_symlinks</code> | 無視  |

- 方法関数指定

| 名前                             | 意味            |
|--------------------------------|---------------|
| <code>none</code>              | 、下中身          |
| <code>directories_only</code>  | 構造。非          |
| <code>create_symlinks</code>   | 作成。優先する場合、元絶対 |
| <code>create_hard_links</code> | 作成            |

**copy\_symlink**

```

void copy_symlink(const path& existing_symlink,

```

```

 const path& new_symlink);
void copy_symlink(const path& existing_symlink,
 const path& new_symlink,
 error_code& ec) noexcept;

existing_symlink から new_symlink へコピー。

```

### 10.10.5 削除

#### remove

```

bool remove(const path& p);
bool remove(const path& p, error_code& ec) noexcept;

```

`p` が存在する場合、`p` を削除。削除に失敗した場合、`error_code` を通知し、`ec` が `error_code::value_type::value` 以外の値の場合、`remove` は `false` を返す。

戻り値は、削除に成功した場合 `true` を返す。削除に失敗した場合 `false` を返す。`error_code` は、削除に失敗した場合、`error_code` を通知し、`ec` が `error_code::value_type::value` 以外の値の場合、`remove` は `false` を返す。

#### remove\_all

```

uintmax_t remove_all(const path& p);
uintmax_t remove_all(const path& p, error_code& ec) noexcept;

```

`p` が存在する場合、`p` を削除。削除に失敗した場合、`error_code` を通知し、`ec` が `error_code::value_type::value` 以外の値の場合、`remove_all` は `0` を返す。

`p` が存在する場合、`p` がディレクトリの場合、`p` の下にあるすべてのファイルとディレクトリを削除。削除に失敗した場合、`error_code` を通知し、`ec` が `error_code::value_type::value` 以外の値の場合、`remove_all` は `0` を返す。

`p` が存在する場合、`p` がファイルの場合、`p` を削除。

戻り値は、削除したファイルの個数を返す。`error_code` は、削除に失敗した場合、`error_code` を通知し、`ec` が `error_code::value_type::value` 以外の値の場合、`remove_all` は `static_cast<uintmax_t>(-1)` を返す。

10.10.6 变更

permissions

```
void permissions(const path& p, perms prms,
 perm_options opts=perm_options::replace);
void permissions(const path& p, perms prms,
 error_code& ec) noexcept;
void permissions(const path& p, perms prms,
 perm_options opts,
 error_code& ec);
```

`permissions` `p` 文件/目录变更。

`opts` `perm_options` 型 `enum` 值、`replace`, `add`, `remove` 或 `nofollow` 指定。省略时 `replace`。

存在 `foo` 文件、对 `foo` 实行权限附加、以下 `perms` 书。

```
int main()
{
 using namespace std::filesystem ;

 permissions("./foo", perms(0111), perm_options::add) ;
}
```

`perm_options` 以下 `enum` 值持。

| 名前                    | 意味                                                          |
|-----------------------|-------------------------------------------------------------|
| <code>replace</code>  | <code>perms</code> 置換                                       |
| <code>add</code>      | <code>perms</code> 指定追加                                     |
| <code>remove</code>   | <code>perms</code> 指定取除                                     |
| <code>nofollow</code> | 場合、 <code>permissions</code> 先、 <code>permissions</code> 变更 |

`replace`、`add`、`remove`、`nofollow` 置換、追加、取除、`nofollow` 場合、

```
perm_options opts = perm_options::replace | perm_options::nofollow ;
```

書。

**rename**

```
void rename(const path& old_p, const path& new_p);
void rename(const path& old_p, const path& new_p,
 error_code& ec) noexcept;
```

old\_p new\_p をリネームする。

old\_p new\_p が同一存在を指す場合、何もしない。

```
int main()
{
 using namespace std::filesystem ;

 // 何もしない
 rename("foo", "foo") ;
}
```

例外以外の場合、例外を伴う以下挙動が発生する。

例、リネーム前 new\_p が既存を指す場合、例外を伴って new\_p を削除する。

```
int main()
{
 using namespace std::experimental::filesystem ;

 {
 std::ofstream old_p("old_p"), new_p("new_p") ;

 old_p << "old_p" ;
 new_p << "new_p" ;
 }

 // old_p の内容を "old_p"
 // new_p の内容を "new_p"

 // old_p を new_p にリネーム
 // new_p を削除
 rename("old_p", "new_p") ;

 std::ifstream new_p("new_p") ;

 std::string text ;
```

## 第 10 章 標準ライブラリ

```

new_p >> text ;

// "old_p"
std::cout << text ;
}

```

、`new_p` が既存の空ディレクトリを指している場合、POSIX 準拠 OS では、ディレクトリに伴って `new_p` を削除する。他の OS での動作は保証されない。

```

int main()
{
 using namespace std::experimental::filesystem ;

 create_directory("old_p") ;
 create_directory("new_p") ;

 // POSIX 準拠環境での動作を保証
 rename("old_p", "new_p") ;
}

```

`old_p` がディレクトリの場合、ディレクトリを削除する前に、ディレクトリ内のファイルやディレクトリを削除する必要がある。

## resize\_file

```

void resize_file(const path& p, uintmax_t new_size);
void resize_file(const path& p, uintmax_t new_size,
 error_code& ec) noexcept;

```

ディレクトリ `path` を指しているディレクトリを `new_size` に変更する。

POSIX では `truncate()` を行い、ディレクトリを振替える。ディレクトリが小さい場合、余計なデータを捨てる。ディレクトリが大きい場合、増分を `null` (0) に設定する。ディレクトリが最終的に日時を更新する。

## 10.10.7 情報取得

## ディレクトリ判定

ディレクトリを表現する `file_type` 型の enum、この enum 値は以下に示す。



| 名前                     | 意味            |
|------------------------|---------------|
| <code>none</code>      | 決定できない        |
| <code>not_found</code> | 発見できないを示す疑似例外 |
| <code>regular</code>   | 通常ファイル        |
| <code>directory</code> | ディレクトリ        |
| <code>symlink</code>   | シンリンク         |
| <code>block</code>     | ブロックデバイス      |
| <code>fifo</code>      | FIFO デバイス     |
| <code>socket</code>    | ソケット          |
| <code>unknown</code>   | 存在しないか決定できない  |

他、実装依存の追加の可能性。

、`file_status` 関数 `type` 戻り値を調べる。

以下、存在しない `foo` を調べる。

```
int main()
{
 using namespace std::filesystem ;

 auto s = status("./foo") ;
 bool b = s.type() == file_type::directory ;
}
```

、`status` path を決定する。判定 `is_directory` 用意。

```
int main()
{
 using namespace std::filesystem ;

 bool b1 = is_directory("./foo") ;

 auto s = status("./foo") ;
 bool b2 = is_directory(s) ;
}
```

`file_status` 情報、物理状態変更、同対何度情報取得場合、

第 10 章 文件系统

`file_status` 使用 `is_x` 形式函数、以下形式取。

```
bool is_x(file_status s) noexcept;
bool is_x(const path& p);
bool is_x(const path& p, error_code& ec) noexcept;
```

以下函数名前、判定表。

| 名前                           | 意味        |
|------------------------------|-----------|
| <code>is_regular_file</code> | 通常ファイル    |
| <code>is_directory</code>    | ディレクトリ    |
| <code>is_symlink</code>      | シンリンク     |
| <code>is_block</code>        | ブロックデバイス  |
| <code>is_fifo</code>         | FIFO デバイス |
| <code>is_socket</code>       | ソケット      |

、単一関数調以下名前関数存在。

| 名前                    | 意味                                                                   |
|-----------------------|----------------------------------------------------------------------|
| <code>is_other</code> | 存在、通常ファイル、ディレクトリ、シンリンク、ブロックデバイス、FIFO デバイス、ソケット                       |
| <code>is_empty</code> | 場合、空 <code>true</code> 返。非空場合、 <code>0</code> 返 <code>true</code> 返。 |

**status**

```
file_status status(const path& p);
file_status status(const path& p, error_code& ec) noexcept;
```

関数 `status` は `p` が指すファイルの情報を格納した `file_status` を返す。  
`p` が存在しない場合、`ec` がエラーコードを格納し、`file_status` を返す。

**status\_known**

```
bool status_known(file_status s) noexcept;

s.type() != file_type::none を返す。
```

**symlink\_status**

```
file_status symlink_status(const path& p);
file_status symlink_status(const path& p, error_code& ec) noexcept;

status が同様に、p がシンボリックリンクの場合、シンボリックリンクの

status を返す。
```

**equivalent**

```
bool equivalent(const path& p1, const path& p2);
bool equivalent(const path& p1, const path& p2,
 error_code& ec) noexcept;

p1 が p2 と物理的に等しい場合、同一ファイルの場合、true を返す。

異なる場合 false を返す。


```

**exists**

```
bool exists(file_status s) noexcept;
bool exists(const path& p);
bool exists(const path& p, error_code& ec) noexcept;

s, p が指すファイルが存在する場合 true を返す。存在しない場合 false を返す。


```

**file\_size**

```
uintmax_t file_size(const path& p);
uintmax_t file_size(const path& p, error_code& ec) noexcept;
```

## 第 10 章 文件系统

`p` 指存在情况返回。  
 存在情况。通常情况、  
 返回。以外情况、  
 通知 `error_code` 受取函数、戻値  
`static_cast<uintmax_t>(-1)`。

**hard\_link\_count**

```
uintmax_t hard_link_count(const path& p);
uintmax_t hard_link_count(const path& p, error_code& ec) noexcept;
```

`p` 指数。  
 通知 `error_code` 受取函数、戻値  
`static_cast<uintmax_t>(-1)`。

**last\_write\_time**

```
file_time_type last_write_time(const path& p);
file_time_type last_write_time(const path& p,
 error_code& ec) noexcept;
```

`p` 指最終更新日時。  
`void last_write_time( const path& p, file_time_type new_time);`  
`void last_write_time( const path& p, file_time_type new_time,`  
`error_code& ec) noexcept;`  
`p` 指最終更新日時 `new_time`。  
`last_write_time(p, new_time)` 呼出後、`last_write_time(p) == new_time` 保証。  
 物理、  
 分解能品質問題。  
`file_time_type`、`std::chrono::time_point` 特殊化以下定義。

```
namespace std::filesystem {
 using file_time_type = std::chrono::time_point< trivial_clock > ;
}
```

`trivial_clock`、`TrivialClock` 要件満、  
 値正確表現。  
 具体的型実装依存、完全。

时间操作函数非常困难。现在时刻设定、差分时间设定。

```
int main()
{
 using namespace std::experimental::filesystem ;
 using namespace std::chrono ;
 using namespace std::literals ;

 // 最终更新时间取得
 auto timestamp = last_write_time("foo") ;

 // 时刻 1 时间进
 timestamp += 1h ;
 // 更新
 last_write_time("foo", timestamp) ;

 // 现在时刻取得
 auto now = file_time_type::clock::now() ;

 last_write_time("foo", now) ;
}
```

多实现 `file_time_type`、`time_point<std::chrono::system_clock>` 使用。 `file_time_type::clock` 与 `system_clock`、`system_clock::to_time_t` 与 `system_clock::from_time_t` 与 `time_t` 型相互变换、。

```
// file_time_type::clock 与 system_clock 场合

int main()
{
 using namespace std::experimental::filesystem ;
 using namespace std::chrono ;

 // 最终更新时间文字列得
 auto time_point_value = last_write_time("foo") ;
 time_t time_t_value =
 system_clock::to_time_t(time_point_value) ;
 std::cout << ctime(&time_t_value) << '\n' ;
}
```

## 第 10 章 時間と時刻

```

// 最終更新日時 2017-10-12 19:02:58 設定
tm struct_tm{} ;
struct_tm.tm_year = 2017 - 1900 ;
struct_tm.tm_mon = 10 ;
struct_tm.tm_mday = 12 ;
struct_tm.tm_hour = 19 ;
struct_tm.tm_min = 2 ;
struct_tm.tm_sec = 58 ;

time_t timestamp = std::mktime(&struct_tm) ;
auto tp = system_clock::from_time_t(timestamp) ;

last_write_time("foo", tp) ;
}

```

このコードは、C++ の現在時刻を利用し、C++ 風の時間と時刻の表現を、問題点を将来に規格改定で改善することを示す。

**read\_symlink**

```

path read_symlink(const path& p);
path read_symlink(const path& p, error_code& ec);

```

この関数は、p を解決し、先容量を取得する。p が存在しない場合は、error\_code を返す。

**space**

```

space_info space(const path& p);
space_info space(const path& p, error_code& ec) noexcept;

```

この関数は、p を指し、先容量を取得する。space\_info は以下に定義されている。

```

struct space_info {
 uintmax_t capacity;
 uintmax_t free;
 uintmax_t available;
};

```

関数、POSIX 関数 `statvfs` 関数呼出結果 `struct statvfs` `f_blocks`, `f_bfree`, `f_bavail`、`f_frsize` 乗、`space_info` `capacity`, `free`, `available` 返。値決定。 `static_cast<uintmax_t>(-1)` 代入。

通知 `error_code` 返関数場合、`space_info` `static_cast<uintmax_t>(-1)` 代入。

`space_info` 意味説明、以下表。

| 名前                     | 意味      |
|------------------------|---------|
| <code>capacity</code>  | 総容量     |
| <code>free</code>      | 空容量     |
| <code>available</code> | 権限使用空容量 |

# 索引

\*this, 34  
 ::value, 22  
 <algorithm>, 153, 156, 183, 220  
 <any>, 104  
 <atomic>, 221  
 <chrono>, 262  
 <cmath>, 165, 221  
 <cstdint>, 81  
 <execution>, 156, 161  
 <filesystem>, 225  
 <functional>, 179, 195, 196  
 <iterator>, 220  
 <memory>, 196  
 <memory\_resource>, 133  
 <new>, 176  
 <numeric>, 222  
 <optional>, 110  
 <system\_error>, 227  
 <tuple>, 194  
 <type\_traits>, 201  
 <utility>, 193  
 <variant>, 85  
 [[deprecated]] 属性, 6  
 [[fallthrough]] 属性, 40  
 [[maybe\_unused]] 属性, 43  
 [[nodiscard]] 属性, 41  
 \_\_cpp\_aggregate\_nsdmi, 24  
 \_\_cpp\_binary\_literals, 5  
 \_\_cpp\_capture\_star\_this, 37  
 \_\_cpp\_constexpr, 23, 39  
 \_\_cpp\_decltype\_auto, 14  
 \_\_cpp\_deduction\_guides, 61  
 \_\_cpp\_fold\_expressions, 34  
 \_\_cpp\_generic\_lambdas, 15  
 \_\_cpp\_hex\_float, 28  
 \_\_cpp\_if\_constexpr, 56  
 \_\_cpp\_init\_captures, 18  
 \_\_cpp\_inline\_variables, 79  
 \_\_cpp\_nested\_namespace\_definitions, 40  
 \_\_cpp\_noexcept\_function\_type, 30  
 \_\_cpp\_return\_type\_deduction, 9  
 \_\_cpp\_rvalue\_references, 2  
 \_\_cpp\_sized\_deallocation, 25  
 \_\_cpp\_static\_assert, 39  
 \_\_cpp\_structured\_bindings, 75  
 \_\_cpp\_template\_auto, 62  
 \_\_cpp\_variable\_templates, 23  
 \_\_cpp\_variadic\_using, 81  
 \_\_has\_cpp\_attribute(deprecated), 8  
 \_\_has\_cpp\_attribute(fallthrough), 41  
 \_\_has\_cpp\_attribute(maybe\_unused), 45  
 \_\_has\_cpp\_attribute(nodiscard), 43  
 \_\_has\_cpp\_attribute 式, 4  
 \_\_has\_include 式, 3  
 \_\_USE\_RVALUE\_REFERENCES, 2  
 \_v 版, 23, 201  
 0B, 5  
 0b, 5  
 0x, 27  
 16 進数浮動小数点数 式, 27  
 3 次元 hypot, 221  
 absolute, 248  
 addressof, 196  
 all\_of, 153, 155  
 allocate, 134, 149  
 any, 85, 104  
   any\_cast<T>, 109  
   emplace, 105, 106  
   has\_value, 107  
   make\_any<T>, 108  
   reset, 106  
   std::in\_place\_type<T>, 105  
   swap, 107  
   type, 108  
   構築, 105  
   代入, 106  
   破棄, 105  
 any\_cast<T>, 109  
 The Art of Computer Programming, 183  
 as\_const, 193  
 auto, 8, 61  
   厳格~, 9  
 basic\_string\_view, 123



BinaryOperation, 157  
 BinaryOperation1, 157  
 BinaryOperation2, 157  
 BinaryPredicate, 157  
 bool, 115  
 bool\_constant, 201  
 Boyer–Moore–Horspool 検索関数, 182  
 Boyer–Moore 文字列検索関数, 180  
  
 C++03, v  
 C++11, v  
 C++14, v, 5  
   言語, 5  
 C++17, vi, 1, 27  
   言語, 27  
 C++20, vi  
 C++98, v  
 c\_str, 232  
 canonical, 248  
 char, 121, 230  
 CHAR\_BIT, 81  
 char16\_t, 121, 230  
 char32\_t, 121, 230  
 clamp, 220  
 clear, 228  
 Compare, 157  
 conjunction, 202  
 constexpr, 23, 37  
 constexpr if 文, 46  
   解決関数問題, 55  
   解決関数問題, 55  
 copy, 251  
 copy\_file, 251  
 copy\_options, 251, 252  
 copy\_symlink, 252  
 create\_directories, 249  
 create\_directory, 249  
 create\_directory\_symlink, 250  
 create\_hard\_link, 251  
 create\_symlink, 250  
 current\_path, 248  
 C 関数, vi, 1  
  
 data, 220  
 deallocate, 134, 149  
 decltype(auto), 9  
 delete, 25  
 depth, 244  
 destroy, 199  
 directory\_entry, 236, 238  
 directory\_iterator, 238, 240  
   error\_code, 241  
   increment, 241  
 directory\_options, 241, 242  
 disable\_recursion\_pending, 245  
 discarded statement, 52  
  
 disjunction, 203  
 do\_allocate, 135, 143, 149  
 do\_deallocate, 135, 149  
 do\_is\_equal, 135  
  
 emplace, 90, 92, 105, 106, 208  
   戻値, 206  
 emplace\_back, 206  
 emplace\_front, 206  
 empty, 220  
 equivalent, 259  
 error\_code, 228, 241  
 ExecutionPolicy, 156  
 exists, 259  
 extract, 213  
  
 false\_type, 204  
 file\_size, 259  
 file\_status, 236, 257, 258  
 file\_time\_type, 260  
 file\_type, 236, 256  
 filesystem\_error, 227  
 fold 式, 30  
   単項~, 31, 32  
   二項~, 31, 33  
   左~, 31, 33  
   右~, 31, 33  
 follow\_directory\_symlink, 243  
 for-range 宣言, 67  
 free, 136  
  
 gcd, 222  
 generic\_string, 234  
 get<I>(v), 97  
 get<T>(v), 99  
 get\_default\_resource, 138, 140  
 get\_if<I>(vp), 100  
 get\_if<T>(vp), 100  
  
 hard\_link\_count, 260  
 has\_unique\_object\_representations<T>, 205  
 has\_value, 107, 114  
 holds\_alternative<T>(v), 96  
 hypot, 221  
  
 if constexpr, 46  
 in\_place\_type, 120  
 increment, 241  
 index, 94  
 inline  
   関数, 75  
   展開, 75  
   変数, 75, 78  
 insert, 215

## 索引

insert\_or\_assign, 208  
 insert\_return\_type, 216  
 integral\_constant, 201  
 IntType, 83  
 INVOKE, 195  
 invoke, 195  
 is\_always\_lock\_free, 221  
 is\_directory, 257  
 is\_equal, 134  
 is\_invocable, 204  
 is\_invocable\_r, 204  
 is\_lock\_free, 221  
 is\_nothrow\_invocable, 204  
 is\_nothrow\_invocable\_r, 204  
 is\_nothrow\_swappable<T>, 206  
 is\_nothrow\_swappable\_with<T, U>, 206  
 is\_x, 258  
 ISO/IEC 14882, v  
  
 key, 212  
  
 largest\_required\_pool\_block, 146  
 last\_write\_time, 260  
 lcm, 223  
 lock, 160  
 lock\_guard, 221  
  
 make\_any<T>, 108  
 make\_from\_tuple, 194  
 make\_optional<T, Args ...>, 120  
 make\_optional<T>, 119  
 malloc, 136, 141  
 map, 207, 209  
   key, 212  
   mapped, 212  
 mapped, 212  
 max\_blocks\_per\_chunk, 146  
 memory\_resource, 133  
   allocate, 134  
   deallocate, 134  
   do\_allocate, 135, 143  
   do\_deallocate, 135  
   do\_is\_equal, 135  
   free, 136  
   get\_default\_resource, 140  
   is\_equal, 134  
   malloc, 136  
   new\_delete\_resource, 139  
   null\_memory\_resource, 139  
   set\_default\_resource, 140  
 merge, 210  
 monotonic\_buffer\_resource, 137, 147  
 multi\_map, 207  
 multimap, 209  
 multiset, 209  
 mutable, 36  
  
 mutex, 160  
  
 native, 232  
 negation, 203  
 new, 141  
 new\_delete\_resource, 139  
 node\_type, 211  
 noexcept, 29  
 not\_fn, 196  
 not1, 196  
 not2, 196  
 null\_memory\_resource, 139  
 null 終端, 121, 126  
  
 ODR (One Definition Rule) , 76  
 operator (), 14, 37  
 operator delete, 25  
 optional, 85, 110  
   bool, 115  
   has\_value, 114  
   in\_place\_type, 120  
   make\_optional<T, Args ...>, 120  
   make\_optional<T>, 119  
   reset, 117  
   std::bad\_optional\_access, 116  
   std::in\_place\_type<T>, 113  
   std::nullopt, 112, 119  
   swap, 114  
   value, 115  
   value\_or, 116  
   構築, 112  
   代入, 113  
   ☐☐☐☐☐☐索引数, 112  
   破棄, 113  
   比較, 117  
 options, 147, 243  
  
 parallel\_policy, 159, 160  
 parallel\_unsequenced\_policy, 160  
 path, 229, 235  
 path::string\_type, 233  
 path::value\_type, 230  
 perm\_options, 254  
 permissions, 236, 254  
 perms, 236  
 polymorphic\_allocator, 137  
   ☐☐☐☐☐☐☐☐, 138  
 pool\_options, 145, 146  
 pop, 244  
 Predicate, 157  
 preferred\_separator, 230  
 proximate, 249  
  
 queue, 206  
  
 read\_symlink, 262

```

recursion_pending, 245
recursive_directory_iterator, 238, 242
 depth, 244
 directory_options, 242
 disable_recursion_pending, 245
 follow_directory_symlink, 243
 options, 243
 pop, 244
 recursion_pending, 245
refresh, 240
relative, 249
release, 147, 151
remove, 253
remove_all, 253
remove_prefix, 129
remove_suffix, 129
rename, 255
reset, 106, 117
resize_file, 256
rvalue 常量表达式, 1

scoped_enum, 81
searcher, 179
sequenced_policy, 159
set, 209
 value, 212
set_default_resource, 140
SFINAE, 38
shared_ptr::weak_type, 200
shared_ptr<T[]>, 192
size, 220
space, 262
space_info, 262, 263
splice, 209
stack, 206
static_assert
 文字列化, 39
status, 236, 259
status_known, 259
statvfs, 263
std::any, 104
std::apply, 178
std::bad_alloc, 160
std::bad_optional_access, 116
std::basic_string, 123
std::basic_string_view, 123
std::boyer_moore_horspool_searcher, 182
std::boyer_moore_searcher, 180
std::byte, 81, 222
std::chrono_time_point, 260
std::default_searcher, 179
std::error_code, 227, 228
std::execution::par, 156
std::execution::par_unseq, 156
std::execution::seq, 156
std::false_type, 201
std::filesystem, 225
std::filesystem::filesystem_error, 227
std::filesystem::path, 229
std::for_each, 158
std::hardware_constructive_interference_size, 176
std::hardware_destructive_interference_size, 176
std::in_place_type<T>, 90, 105, 113
std::integral_constant, 22
std::is_execution_policy<T>, 161
std::monostate, 89
std::nullopt, 112, 119
std::pmr::memory_resource, 133
std::pmr::polymorphic_allocator, 137
std::sample, 183
std::scoped_lock, 221
std::size_t, 25
std::string, 123
 常量表达式, 130
std::string_view, 123
 常量表达式, 130
std::terminate, 161
std::true_type, 201
std::tuple_size<E>, 71
std::uncaught_exception, 176
std::uncaught_exceptions, 176, 177
std::variant_alternative<I, T>, 96
std::variant_size<T>, 95
std::visit, 103
string, 233
string_type, 230, 233
string_view, 121
 remove_prefix, 129
 remove_suffix, 129
 構築, 125
 操作, 128
 変換関数, 127
swap, 94, 107, 114
symlink_status, 259
synchronized_pool_resource, 142, 145

temp_directory_path, 248
traits, 22
 変数テンプレート版, 201
 論理演算, 202
trivial_clock, 261
true_type, 204
try_emplace, 207
tuple, 178, 194
type, 108, 236
typedef 名, 19

u16string, 233
u16string_view, 123
u32string, 233

```

## 索引

u8, 28  
 u8path, 231  
 UnaryOperation, 157  
 uninitialized\_copy, 198  
 uninitialized\_default\_construct, 197  
 uninitialized\_fill, 199  
 uninitialized\_move, 199  
 uninitialized\_value\_construct, 198  
 union, 85, 86  
   型安全, 85  
   型非安全, 86  
 unordered\_map, 207, 209  
 unordered\_multi\_map, 207  
 unordered\_multimap, 209  
 unordered\_multiset, 209  
 unordered\_set, 209  
 unsigned char, 81  
 unsynchronized\_pool\_resource, 142, 145  
 upstream\_resource, 147, 152  
 using 属性名前空間, 62  
 UTF-16 字元, 230  
 UTF-32 字元, 230  
 UTF-8 字元, 230  
 UTF-8 文字, 230  
  
 value, 115, 212  
 value\_or, 116  
 value\_type, 230  
 valueless\_by\_exception, 93  
 variant, 85  
   emplace, 90, 92  
   get<I>(v), 97  
   get<T>(v), 99  
   get\_if<I>(vp), 100  
   get\_if<T>(vp), 100  
   holds\_alternative<T>(v), 96  
   index, 94  
   std::variant\_alternative<I, T>, 96  
   std::variant\_size<T>, 95  
   std::visit, 103  
   swap, 94  
   valueless\_by\_exception, 93  
   字元初期化, 89  
   初期化, 88  
   宣言, 88  
   大小比較, 102  
   代入, 92  
   字元字串初期化, 88  
   同一性比較, 101  
   破棄, 91  
 void\_t, 201  
  
 wchar\_t, 121, 230  
 weak\_type, 200  
 weakly\_canonical, 248  
 wstring, 233  
  
 字元字串初期化, 23  
 字元字串, 141  
 字元字串 R (保管標本), 187  
 字元字串 S (選擇標本), 186  
 字元字串, 133  
 字元字串宣言, 19  
 字元字串多項式, 168  
 演算子字元字串評估順序, 45  
  
 型  
   IntType, 83  
   scoped enum, 81  
   std::byte, 81  
   字元字串, 81  
 型安全字元字串  
   union, 85  
 型字元字串字元字串, 21  
 型指定子, 9  
 型非安全, 86  
 可變長 using 宣言, 79  
 可變長字元字串字元字串, 30  
 函數型 (例外指定), 29  
 函數宣言, 19  
 函數字元字串字元字串, 21  
 函數字元字串字元字串字元字串型推定, 8  
 完全型, 206  
 機能字元字串, 1  
 機能字元字串字元字串, 2  
 字元字串字元字串字元字串, 175  
 字元字串字元字串字元字串, 175  
 球字元字串字元字串, 173  
 球面字元字串字元字串字元字串, 167  
 字元字串宣言, 18  
 字元字串字元字串字元字串字元字串, 14, 37  
 字元字串字元字串型, 14  
 嚴格字元字串 auto, 9  
 字元字串言語, vi  
 構造化束縛, 63  
   完全形字元字串名前, 71  
   字元字串, 73  
   仕様, 69  
   配列, 69  
   非 static 字元字串字元字串字元字串, 74  
   字元字串字元字串字元字串, 74  
 構造化束縛宣言, 67  
 古典的字元字串 union, 86  
 字元字串字元字串字元字串, 16, 34  
 字元字串字元字串字元字串字元字串, 220  
   data, 220  
   empty, 220  
   size, 220  
 字元字串字元字串時條件分歧, 46, 49  
 字元字串字元字串時定數, 47  
  
 最小公倍数, 223  
 字元字串字元字串字元字串字元字串, 25

最大公約数, 222  
 異種型変換, 14  
 異種型変換実行, 161  
 指数積分, 174  
 実行型, 162  
 実行時, 156  
     std::execution::par, 156  
     std::execution::par\_unseq, 156  
     std::execution::seq, 156  
 条件文  
     初期化文付, 56  
 条件分岐  
     型時, 46, 49  
     実行時, 46  
     型時, 48  
 初期化文付条件文, 56  
 初期化型変換, 15  
 型変換, 250  
 推定, 59  
 数学特殊関数, 165  
     型多項式, 168  
     球関数, 173  
     球面関数陪関数, 167  
     指数積分, 174  
     第1種完全楕円積分, 169  
     第1種球関数, 173  
     第1種不完全楕円積分, 170  
     第1種型関数, 171  
     第1種変形型関数, 172  
     第2種完全楕円積分, 169  
     第2種不完全楕円積分, 170  
     第2種変形型関数, 172  
     第3種完全楕円積分, 169  
     第3種不完全楕円積分, 171  
     型関数, 171  
     型関数, 168  
     型多項式, 166  
     型陪多項式, 166  
     型関数, 174  
     型多項式, 166  
     型陪関数, 167  
 数値区切文字, 5  
 型, 154  
 選択標本, 186  
 属性, 4, 62  
 属性名前空間, 62  
  
 第1種完全楕円積分, 169  
 第1種球関数, 173  
 第1種不完全楕円積分, 170  
 第1種型関数, 171  
 第1種変形型関数, 172  
 第2種完全楕円積分, 169  
 第2種不完全楕円積分, 170  
 第2種変形型関数, 172  
 第3種完全楕円積分, 169  
  
 第3種不完全楕円積分, 171  
 量込, 30  
 多値, 63  
 単項 fold 式, 31, 32  
 単純宣言, 67  
 定義 1 原則, 76  
 定数  
     型違, 21  
     型時, 47  
 型  
     create\_directories, 249  
     create\_directory, 249  
     create\_directory\_symlink, 250  
 型区切文字, 230  
 型競合, 159  
 型, 159  
 型実引数推定, 59  
 型宣言, 18, 19, 20  
 動的型, 133  
 動的型, 137  
 型, 27  
  
 名前空間  
     型, 39  
 二項 fold 式, 31, 33  
 二進数, 5  
 型, 230  
 型, 230  
 型名前空間, 39  
 型関数, 171  
 型, 211  
     extract, 213  
     insert, 215  
     insert\_return\_type, 216  
     key, 212  
     mapped, 212  
     node\_type, 211  
     value, 212  
     取得, 213  
     要素追加, 215  
  
 型, 81  
     型数, 81  
 型干渉, 175  
 型, 251  
 型, 236, 254  
 型, 30  
 型実行, 162  
 型非型実行, 162  
 非 static 型, 23  
 非型型, 61  
 非順序連想型, 209  
     merge, 210  
 左 fold, 31, 33  
 非標準属性, 63  
 型, 225

## 索引

- 绝对路径, 256
  - 相对路径, 236
  - 相对路径, 236
  - 绝对, 248
  - canonical, 248
  - current\_path, 248
  - proximate, 249
  - relative, 249
  - temp\_directory\_path, 248
  - weakly\_canonical, 248
  - 取得, 248
  - 操作, 234, 248
- 不完全型, 206
- 浮動小数点数, 28
- 浮動小数点数, 27
- largest\_required\_pool\_block, 146
  - max\_blocks\_per\_chunk, 146
  - options, 147
  - pool\_options, 145, 146
  - release, 147
  - synchronized\_pool\_resource, 142, 145
  - unsynchronized\_pool\_resource, 142, 145
  - upstream\_resource, 147
- 型, 9
  - 言語 C++, v
- 並列
  - BinaryOperation, 157
  - BinaryOperation1, 157
  - BinaryOperation2, 157
  - BinaryPredicate, 157
  - Compare, 157
  - ExecutionPolicy, 156
  - lock, 160
  - mutex, 160
  - parallel\_policy, 159, 160
  - parallel\_unsequenced\_policy, 160
  - Predicate, 157
  - sequenced\_policy, 159
  - std::bad\_alloc, 160
  - std::is\_execution\_policy<T>, 161
  - std::terminate, 161
  - UnaryOperation, 157
  - 実行, 161
    - 実行時, 156
    - 競合, 159
    - , 159
    - 実行, 162
  - 非実行, 162
    - 要素関数, 157
    - 例外, 160
  - 関数, 168
  - 存在, 3
  - 変数宣言, 20
  - 変数, 18, 22
  - 保管標本, 187
- fold, 31, 33
  - , 18
  - , 140
  - 管理, 196
  - , 133, 140
  - 取得, 139
  - , 142
  - 初期化子, 147
  - 初期化子, 23
  - 文字型, 121
  - ~配列, 121
  - 文字列, 230
  - 文字列 static\_assert, 39
  - 文字列, 121
  - 文字列, 121
  - allocate, 149
    - deallocate, 149
    - do\_allocate, 149
    - do\_deallocate, 149
    - monotonic\_buffer\_resource, 147
    - release, 151
    - upstream\_resource, 152
    - , 150
- 定義, 130
- 要素関数, 157
- vi
- 多項式, 166
- 陪多項式, 166
- , 15
- 式, 14, 15, 34, 37
- 乱択, 183
- , 16
- 関数, 174
- 多項式, 166
- 陪関数, 167
- 例外指定, 29
- 連想, 209
- merge, 210
- 論理演算 traits, 202

●本書は対価を問合せず、電子メール (info@asciidwango.jp) 宛てに願ひを述べらるゝ。  
但し、本書の記述内容越えの質問に答へることを断つて承知願ふ。

## 江添亮のC++17入門（仮）

2018 年 x 月 x 日 初版発行

著 者 江添 亮  
発行者 川上量生  
発 行 株式会社アスキー  
〒 104-0061  
東京都中央区銀座 4-12-15 歌舞伎座ビル  
編集 03-3549-6153  
電子メール info@asciidwango.jp  
<http://asciidwango.jp/>  
  
発 売 株式会社 KADOKAWA  
〒 102-8177  
東京都千代田区富士見 2-13-3  
営業 0570-002-301 (フリーダイヤル・全国対応)  
受付時間 9:00~17:00 (土日 祝日 年末年始除く)  
<http://www.kadokawa.co.jp/>

印刷・製本 株式会社アスキー

Printed in Japan

落丁・乱丁本を取り替へる。下記 KADOKAWA 読者係に連絡する。  
送料小社負担を取り替へる。  
但し、古書店で本書を購入した場合取り替へる。  
電話 049-259-1100 (9:00-17:00/土日、祝日、年末年始除く)  
〒354-0041 埼玉県入間郡三芳町藤久保 550-1  
定価は表示する。

ISBN: 978-4-04-xxxxx-x

アスキー編集部  
編 集 星野浩章