江添亮の
詳説C++17

江添 亮 著

ASCII
DWANGO

本文中⊠記載⊠⊠⊠⊠⊠社名⊠⊠⊠商品名⊠、一般⊠開発⊠⊠⊠⊠⊠登録商標⊠⊠。
⊠⊠、本文中⊠⊠ ™・©・® 表示⊠明記⊠⊠⊠⊠⊠⊠⊠。

はじめに

本書⊠2017年⊠規格制定⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠言語C++⊠国際規格、ISO/IEC 14882:2017⊠新機能⊠⊠⊠⊠⊠⊠解説⊠⊠⊠⊠。

新⊠⊠C++17⊠不具合⊠修正⊠、⊠⊠⊠⊠⊠⊠⊠日々⊠⊠⊠⊠⊠⊠⊠⊠楽⊠⊠⊠新機能⊠⊠⊠⊠⊠追加⊠⊠⊠。⊠⊠結果、C++⊠特徴⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠静的型付⊠⊠損⊠⊠⊠⊠⊠⊠、近年⊠動的⊠型⊠弱⊠言語⊠匹敵⊠⊠⊠⊠⊠柔軟⊠記述⊠可能⊠⊠⊠⊠⊠。

人⊠⊠⊠⊠⊠、新機能⊠学⊠⊠⊠労多⊠⊠⊠益少⊠⊠⊠考⊠⊠⊠⊠⊠⊠⊠⊠、C++⊠新機能⊠現実⊠問題⊠解決⊠⊠⊠⊠⊠便利⊠道具⊠⊠⊠追加⊠⊠⊠⊠⊠⊠、仮⊠機能⊠使⊠⊠⊠⊠⊠⊠⊠問題⊠⊠⊠⊠⊠⊠⊠⊠⊠、便利⊠道具⊠⊠問題⊠対処⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠、C++⊠機能⊠一般的⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠自然⊠⊠感⊠⊠⊠⊠⊠設計⊠⊠⊠⊠⊠⊠⊠、利用⊠難⊠⊠⊠⊠。⊠⊠C++⊠難⊠⊠⊠感⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠C++⊠解決⊠⊠⊠現実⊠問題⊠難⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠、我々⊠理想⊠⊠程遠⊠歪⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠扱⊠時代⊠生⊠⊠⊠⊠。CPU⊠性能上昇⊠停滞⊠、⊠⊠⊠⊠CPU⊠比⊠⊠遥⊠⊠遅⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠定数時間⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠収⊠⊠局所性⊠持⊠⊠⊠⊠⊠⊠操作⊠無料同然⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠単位⊠⊠⊠⊠MB⊠数⊠⊠⊠⊠⊠⊠。手⊠⊠⊠⊠乗⊠超低電力CPU⊠⊠⊠⊠⊠⊠⊠⊠⊠一般的⊠⊠⊠、並列処理、非同期処理⊠全⊠⊠⊠⊠⊠⊠⊠考慮⊠⊠⊠⊠⊠⊠問題⊠⊠⊠⊠。

⊠⊠⊠⊠⊠時代⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠最良⊠⊠⊠⊠手法⊠⊠⊠価値⊠失⊠、⊠⊠⊠⊠逆⊠悪⊠手法⊠成⊠下⊠⊠。同時⊠昔⊠現実的⊠⊠⊠⊠⊠⊠手法⊠今⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠方法⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠、現在活発⊠使⊠⊠⊠⊠⊠生⊠⊠⊠⊠言語⊠、常⊠時代⊠合⊠⊠⊠機能⊠廃止⊠、必要⊠機能⊠追加⊠⊠必要⊠⊠⊠。C++⊠発展⊠⊠⊠⊠留⊠⊠⊠⊠⊠⊠、今後⊠C++⊠使⊠⊠続⊠⊠限⊠、修正⊠機能追加⊠行⊠⊠⊠⊠⊠⊠⊠⊠。

本書⊠執筆⊠Github上⊠公開⊠⊠行⊠⊠⊠。

https://github.com/EzoeRyou/cpp17book

本書☒☒☒☒☒☒☒ GPLv3 ☒。

本書☒執筆☒☒株式会社☒☒☒☒☒ GitHub 上☒ Pull Request ☒送☒☒☒☒☒多☒☒貢献者☒協力☒☒☒☒、誤☒☒正☒、☒☒良☒記述☒実現☒☒☒。☒☒場☒借☒☒謝意☒表☒☒☒。

本書☒誤☒☒見☒☒☒☒☒☒、Pull Request ☒送☒先☒

https://github.com/EzoeRyou/cpp17book

☒。

江添亮

序

0.1 C++ の規格

⊠⊠⊠⊠⊠⊠⊠言語 C++ ⊠ ISO ⊠傘下⊠国際規格 ISO/IEC 14882 ⊠⊠⊠制定⊠⊠⊠⊠⊠。⊠⊠規格⊠数年⊠⊠⊠改定⊠⊠⊠⊠⊠。一般⊠ C++ ⊠規格⊠参照⊠⊠⊠⊠⊠、規格⊠制定⊠⊠西暦⊠下二桁⊠取⊠⊠、C++98（1998 年発行）⊠⊠ C++11（2011 年発行）⊠呼⊠⊠⊠⊠⊠。現在発行⊠⊠⊠⊠⊠ C++ ⊠規格⊠以下⊠⊠⊠⊠。

0.1.1 C++98

C++98 ⊠ 1998 年⊠制定⊠⊠⊠最初⊠ C++ ⊠規格⊠⊠⊠。本来⊠⊠⊠ 1994 年⊠ 1995 年⊠⊠制定⊠⊠⊠予定⊠大幅⊠⊠⊠⊠、1998 年⊠⊠⊠⊠。

0.1.2 C++03

C++03 ⊠ C++98 ⊠文面⊠曖昧⊠点⊠修正⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2003 年⊠制定⊠⊠⊠。新機能⊠追加⊠⊠⊠⊠⊠⊠⊠。

0.1.3 C++11

C++11 ⊠制定途中⊠⊠⊠⊠⊠段階⊠⊠元 C++0x ⊠呼⊠⊠⊠⊠⊠。⊠⊠⊠、200x 年⊠⊠⊠規格⊠制定⊠⊠⊠予定⊠⊠⊠⊠⊠⊠。予定⊠大幅⊠遅⊠、⊠⊠⊠⊠規格⊠制定⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 2011 年⊠年末⊠⊠⊠⊠⊠⊠。C++11 ⊠⊠⊠⊠⊠多⊠⊠新機能⊠追加⊠⊠⊠。

0.1.4 C++14

C++14 ⊠ 2014 年⊠制定⊠⊠⊠。C++11 ⊠文面⊠誤⊠⊠修正⊠⊠他、少⊠新機能⊠追加⊠⊠⊠。本書⊠解説⊠⊠。

0.1.5 C++17

C++17 ⊠ 2017 年⊠制定⊠⊠⊠⊠⊠⊠予定⊠⊠⊠⊠⊠最新⊠ C++ 規格⊠、本書⊠解説⊠⊠。

0.2 C++ の将来の規格

0.2.1 C++20

C++20 ⊠ 2020 年⊠制定⊠⊠⊠⊠⊠⊠予定⊠⊠⊠⊠⊠次⊠ C++ 規格⊠。⊠⊠規格⊠⊠、⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠、⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠注力⊠⊠⊠⊠⊠予定⊠⊠⊠⊠⊠。

0.3 コア言語とライブラリ

C++ ⊠標準規格⊠、大⊠⊠分⊠⊠、C ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠言語⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

C ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、C++ ⊠ C 言語⊠⊠受⊠継⊠⊠機能⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠列単位⊠分割⊠⊠、⊠⊠⊠⊠列⊠置換⊠⊠⊠⊠。

⊠⊠言語⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠書⊠⊠⊠⊠⊠⊠⊠列⊠文法⊠⊠⊠意味⊠⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠、⊠⊠言語機能⊠使⊠⊠実装⊠⊠⊠⊠⊠⊠、標準⊠提供⊠⊠⊠⊠⊠⊠⊠⊠。標準⊠⊠⊠⊠⊠⊠⊠、純粋⊠⊠⊠言語⊠機能⊠⊠⊠実装⊠⊠⊠⊠⊠⊠、⊠⊠以外⊠実装依存⊠方法⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠必要⊠⊠⊠⊠⊠⊠⊠。

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 __has_include 式：⊠⊠⊠⊠⊠⊠⊠⊠⊠存在⊠判定⊠⊠ 3
- 1.3 __has_cpp_attribute 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⊠文字 5
- 2.3 [[deprecated]] 属性 6
- 2.4 通常⊠関数⊠戻⊠値⊠型推定 8
- 2.5 decltype(auto)：厳格⊠ auto 9
- 2.6 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ 14
- 2.7 初期化⊠⊠⊠⊠⊠⊠⊠⊠⊠ 15
- 2.8 変数⊠⊠⊠⊠⊠⊠ 18
  - 2.8.1 意味⊠同⊠⊠⊠型⊠違⊠定数 21

2.8.2 traits ⊠⊠⊠⊠⊠ 22
2.9 constexpr 関数⊠制限緩和 23
2.10 ⊠⊠⊠⊠初期化子⊠⊠⊠⊠⊠⊠⊠初期化⊠組⊠合⊠⊠ 23
2.11 ⊠⊠⊠付⊠解放関数 25

第 3 章 C++17 のコア言語の新機能 27

3.1 ⊠⊠⊠⊠⊠⊠⊠廃止 27
3.2 16 進数浮動小数点数⊠⊠⊠⊠ 27
3.3 UTF-8 文字⊠⊠⊠⊠ 28
3.4 関数型⊠⊠⊠⊠例外指定 29
3.5 fold 式 30
3.6 ⊠⊠⊠式⊠ *this ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 34
3.7 constexpr ⊠⊠⊠式 37
3.8 文字列⊠⊠ static_assert 39
3.9 ⊠⊠⊠⊠⊠⊠名前空間定義 39
3.10 [[fallthrough]] 属性 40
3.11 [[nodiscard]] 属性 41
3.12 [[maybe_unused]] 属性 43
3.13 演算子⊠⊠⊠⊠⊠⊠⊠評価順序⊠固定 45
3.14 constexpr if 文：⊠⊠⊠⊠⊠時条件分岐 46
3.14.1 実行時⊠条件分岐 46
3.14.2 ⊠⊠⊠⊠⊠⊠時⊠条件分岐 48
3.14.3 ⊠⊠⊠⊠⊠時⊠条件分岐 49
3.14.4 超上級者向⊠解説 52
3.14.5 constexpr if ⊠⊠解決⊠⊠⊠⊠問題 55
3.14.6 constexpr if ⊠解決⊠⊠⊠問題 55
3.15 初期化文付⊠条件文 56
3.16 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実引数推定 59
3.16.1 推定⊠⊠⊠ 59
3.17 auto ⊠⊠⊠非型⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠宣言 61
3.18 using 属性名前空間 62
3.19 非標準属性⊠無視 63
3.20 構造化束縛 63
3.20.1 超上級者向⊠解説 67
3.20.2 構造化束縛宣言⊠仕様 69
3.20.3 初期化子⊠型⊠配列⊠場合 69

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 . . . 88
⊠⊠⊠⊠⊠初期化 . . . 88
⊠⊠⊠初期化 . . . 89
variant⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠値⊠渡⊠⊠場合 . . . 89
in_place_type⊠⊠⊠ emplace 構築 . . . 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 . . . 93
valueless_by_exception⊠⊠⊠⊠関数 . . . 93
index⊠⊠⊠⊠関数 . . . 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⊠返⊠ . . . 96
4.1.12 holds_alternative：variant⊠指定⊠⊠型⊠値⊠保持⊠⊠⊠⊠⊠⊠⊠⊠⊠確認 . . . 96
4.1.13 get<I>(v)：⊠⊠⊠⊠⊠⊠⊠⊠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 . . . 101
同一性⊠比較 . . . 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⊠方 104
4.2.2 any ⊠構築⊠破棄 105
4.2.3 in_place_type ⊠⊠⊠⊠⊠⊠⊠⊠ 105
4.2.4 any ⊠⊠代入 106
4.2.5 any ⊠⊠⊠⊠⊠関数 106
emplace 106
reset：値⊠破棄 106
swap：⊠⊠⊠⊠ 107
has_value：値⊠保持⊠⊠⊠⊠⊠⊠⊠⊠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⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠ 110
4.3.1 使⊠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⊠⊠⊠⊠⊠⊠⊠⊠⊠値⊠返⊠ 116
4.3.11 reset：保持⊠⊠⊠⊠値⊠破棄⊠⊠ 117
4.3.12 optional 同士⊠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> ⊠ in_place_type 構築⊠⊠返⊠ 120

第 5 章　string_view：文字列ラッパー　121
5.1　使⊠方 . . . 121
5.2　basic_string_view . . . 123
5.3　文字列⊠所有、非所有 . . . 123
5.4　string_view ⊠構築 . . . 125
5.4.1　⊠⊠⊠⊠⊠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 ⊠操作 . . . 128
5.5.1　remove_prefix/remove_suffix：先頭、末尾⊠要素⊠削除 . . . 129
5.6　⊠⊠⊠⊠定義⊠⊠⊠⊠ . . . 130

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　133
6.1　⊠⊠⊠⊠⊠⊠⊠⊠ . . . 133
6.1.1　⊠⊠⊠⊠⊠⊠⊠⊠⊠使⊠方 . . . 134
6.1.2　⊠⊠⊠⊠⊠⊠⊠⊠⊠作⊠方 . . . 135
6.2　polymorphic_allocator：動的⊠⊠⊠⊠⊠⊠⊠⊠⊠実現⊠⊠⊠⊠⊠⊠⊠⊠ . . . 137
6.2.1　⊠⊠⊠⊠⊠⊠⊠⊠ . . . 138
6.3　⊠⊠⊠⊠⊠全体⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠取得 . . . 139
6.3.1　new_delete_resource() . . . 139
6.3.2　null_memory_resource() . . . 139
6.3.3　⊠⊠⊠⊠⊠⊠⊠⊠⊠ . . . 140
6.4　標準⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ . . . 140
6.5　⊠⊠⊠⊠⊠⊠⊠ . . . 142
6.5.1　⊠⊠⊠⊠⊠⊠ . . . 142
6.5.2　synchronized/unsynchronized_pool_resource . . . 145
6.5.3　pool_options . . . 146
6.5.4　⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ . . . 146
6.5.5　⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠関数 . . . 147
release() . . . 147
upstream_resource() . . . 147
options() . . . 147
6.6　⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ . . . 147
6.6.1　⊠⊠⊠⊠⊠⊠ . . . 149

6.6.2 ⊠⊠⊠⊠⊠⊠⊠⊠ 150
6.6.3 ⊠⊠他⊠操作 151
release() 151
upstream_resource() 152

第 7 章 並列アルゴリズム 153
7.1 並列実行⊠⊠⊠⊠ 153
7.2 使⊠方 155
7.3 並列⊠⊠⊠⊠⊠⊠詳細 156
7.3.1 並列⊠⊠⊠⊠⊠⊠ 156
7.3.2 ⊠⊠⊠⊠提供⊠⊠関数⊠⊠⊠⊠⊠⊠⊠制約 157
実引数⊠与⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠直接、間接⊠変更⊠⊠⊠⊠⊠⊠⊠ 157
実引数⊠与⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠一意性⊠依存⊠⊠⊠⊠⊠⊠⊠ 158
⊠⊠⊠競合⊠同期 159
7.3.3 例外 160
7.3.4 実行⊠⊠⊠⊠ 161
is_execution_policy traits 161
⊠⊠⊠⊠⊠実行⊠⊠⊠⊠ 161
⊠⊠⊠⊠実行⊠⊠⊠⊠ 162
⊠⊠⊠⊠非⊠⊠⊠⊠⊠実行⊠⊠⊠⊠ 162
実行⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 162

第 8 章 数学の特殊関数群 165
8.1 ⊠⊠⊠⊠多項式（Laguerre polynomials） 166
8.2 ⊠⊠⊠⊠陪多項式（Associated Laguerre polynomials） 166
8.3 ⊠⊠⊠⊠⊠⊠多項式（Legendre polynomials） 166
8.4 ⊠⊠⊠⊠⊠⊠陪関数（Associated Legendre functions） 167
8.5 球面⊠⊠⊠⊠⊠⊠陪関数（Spherical associated Legendre functions） 167
8.6 ⊠⊠⊠⊠⊠多項式（Hermite polynomials） 168
8.7 ⊠⊠⊠関数（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 integroal 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⊠⊠⊠⊠関数（Cylindrical Bessel functions of the first kind） 171
8.15 ⊠⊠⊠⊠関数（Cylindrical Neumann functions） . . . 171
8.16 第 1 種変形⊠⊠⊠⊠関数（Regular modified cylindrical Bessel functions） . . . 172
8.17 第 2 種変形⊠⊠⊠⊠関数（Irregular modified cylindrical Bessel functions） . . . 172
8.18 第 1 種球⊠⊠⊠⊠関数（Spherical Bessel functions of the first kind） 173
8.19 球⊠⊠⊠⊠関数（Spherical Neumann functions） . . . 173
8.20 指数積分（Exponential integral） . . . 174
8.21 ⊠⊠⊠⊠⊠⊠⊠関数（Riemann zeta function） . . . 174

第 9 章 その他の標準ライブラリ 175
9.1 ⊠⊠⊠⊠⊠⊠干渉⊠⊠⊠（⊠⊠⊠⊠⊠⊠⊠⊠） . . . 175
9.2 std::uncaught_exceptions . . . 176
9.3 apply：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⊠⊠⊠⊠⊠⊠ . . . 183
  9.5.1 乱択⊠⊠⊠⊠⊠⊠ . . . 183
  9.5.2 ⊠⊠⊠⊠⊠⊠ S：選択標本、要素数⊠⊠⊠⊠⊠⊠⊠集合⊠⊠⊠標本⊠選択 . . . 186
  9.5.3 ⊠⊠⊠⊠⊠⊠ R：保管標本、要素数⊠⊠⊠⊠⊠⊠集合⊠⊠⊠標本⊠選択 . . . 187
  9.5.4 C++ ⊠ sample . . . 189
9.6 shared_ptr<T[]>：配列⊠対⊠⊠ shared_ptr . . . 192
9.7 as_const：const 性⊠付与 . . . 193
9.8 make_from_tuple：tuple ⊠要素⊠実引数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠呼⊠出⊠ . . . 194
9.9 invoke：指定⊠⊠関数⊠指定⊠⊠実引数⊠呼⊠出⊠ . . . 195
9.10 not_fn：戻⊠値⊠否定⊠⊠⊠⊠ . . . 196

9.11 ☒☒☒☒管理☒☒☒☒☒☒ 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☒☒☒☒☒☒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☒☒☒☒☒ 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☒☒☒☒☒☒☒ splice 操作 209
9.19.1 merge 210
9.19.2 ☒☒☒☒☒☒☒ 211
9.19.3 extract：☒☒☒☒☒☒☒☒取得 213
9.19.4 insert：☒☒☒☒☒☒☒☒☒要素☒追加 215
9.19.5 ☒☒☒☒☒☒☒☒利用例 218
☒☒☒☒☒☒再確保☒☒☒、☒☒☒☒☒☒一部☒要素☒☒別☒☒☒☒☒☒☒移☒ 218
☒☒☒☒☒☒寿命☒超☒☒要素☒存続☒☒☒ 218
map ☒☒☒☒変更☒☒ 219

9.20 ⊠⊠⊠⊠⊠⊠⊠⊠⊠関数 219
9.21 clamp 220
9.22 3 次元 hypot 221
9.23 atomic<T>::is_always_lock_free 221
9.24 scoped_lock：可変長引数 lock_guard 221
9.25 std::byte 222
9.26 最大公約数（gcd）⊠最小公倍数（lcm） 222
  9.26.1 gcd：最大公約数 222
  9.26.2 lcm：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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠全体像 226
10.4 ⊠⊠⊠処理 227
  10.4.1 例外 227
  10.4.2 非例外 228
10.5 path：⊠⊠⊠⊠⊠⊠文字列⊠⊠⊠ 229
  10.5.1 path：⊠⊠⊠⊠⊠⊠⊠文字列 230
  10.5.2 ⊠⊠⊠⊠⊠⊠⊠操作 234
10.6 file_status 236
10.7 directory_entry 238
10.8 directory_iterator 240
  10.8.1 ⊠⊠⊠処理 241
10.9 recursive_directory_iterator 242
  10.9.1 ⊠⊠⊠⊠⊠ 242
  10.9.2 depth：深⊠取得 244
  10.9.3 pop：現在⊠⊠⊠⊠⊠⊠⊠⊠⊠列挙中止 244
  10.9.4 recursion_pending：現在⊠⊠⊠⊠⊠⊠⊠⊠⊠再帰⊠⊠⊠⊠⊠ 245
10.10 ⊠⊠⊠⊠⊠⊠⊠⊠操作関数 248
  10.10.1 ⊠⊠⊠⊠⊠⊠取得 248
    current_path 248
    temp_directory_path 248
  10.10.2 ⊠⊠⊠⊠⊠⊠操作 248
    absolute 248

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 ⊠⊠⊠ 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
⊠⊠⊠⊠⊠⊠⊠⊠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⊠⊠⊠⊠⊠⊠⊠ C ⊠⊠⊠⊠⊠⊠⊠⊠機能⊠追加⊠⊠⊠。

1.1 機能テストマクロ

機能⊠⊠⊠⊠⊠⊠⊠⊠、C++ ⊠実装（C++ ⊠⊠⊠⊠⊠⊠）⊠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⊠⊠⊠現実⊠ C++ ⊠⊠⊠⊠⊠⊠⊠開発⊠⊠⊠⊠⊠⊠⊠行⊠⊠⊠⊠⊠⊠。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
```

☒☒☒☒☒立☒☒☒☒☒☒☒☒☒☒☒現実☒書☒☒☒☒☒。☒☒☒☒☒☒ GCC ☒ MSVC ☒ EDG ☒ Clang ☒☒☒現実☒使☒☒☒☒☒主要☒ 4 ☒☒ C++ ☒☒☒☒☒☒☒対応☒☒ rvalue ☒☒☒☒☒☒☒実装☒☒☒☒☒☒☒☒☒☒判定☒☒機能☒☒☒☒☒☒☒。

☒☒複雑☒☒☒☒☒☒☒☒☒☒解釈☒☒結果、__USE_RVALUE_REFERENCES ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒値☒、☒☒ C++ ☒☒☒☒☒☒☒ rvalue ☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒ 1、☒☒☒☒☒☒☒ 0 ☒☒☒。後☒、☒☒☒☒☒☒☒☒☒☒☒☒☒☒ #if ☒☒☒☒☒☒☒☒☒書☒。

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

#if __USE_RVALUE_REFERENCES == 1
// 文字列☒☒☒☒☒☒処理☒☒☒☒実装☒関数
// C++ ☒☒☒☒☒☒☒ rvalue ☒☒☒☒☒☒☒実装☒☒☒☒☒場合☒☒☒☒☒☒☒
☒☒☒
void process_string( std::string && str ) ;
#endif
```

C++17 ☒☒、上☒☒☒☒☒☒☒立☒☒☒☒☒☒☒☒☒☒☒書☒☒☒☒☒☒☒☒☒☒、標準☒機能☒☒☒☒☒☒☒用意☒☒☒。C++ 実装☒特定☒機能☒☒☒☒☒☒☒☒☒場合、対応☒☒機能☒☒☒☒☒☒☒定義☒☒☒。機能☒☒☒☒☒☒☒値☒、☒☒機能☒ C++ 標準☒採択☒☒☒年☒月☒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☒☒☒☒☒☒☒使☒☒、上☒☒☒☒☒判定☒以下☒☒☒☒書☒☒。

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

#ifdef __cpp_rvalue_references
// 文字列☒☒☒☒☒☒処理☒☒☒☒実装☒関数
// C++ ☒☒☒☒☒☒☒ rvalue ☒☒☒☒☒☒☒実装☒☒☒☒☒場合☒☒☒☒☒☒☒
```

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

機能⊠⊠⊠⊠⊠⊠⊠値⊠通常⊠気⊠⊠⊠必要⊠⊠⊠。機能⊠⊠⊠⊠⊠⊠⊠存在⊠⊠⊠⊠⊠⊠⊠機能⊠有無⊠確認⊠⊠⊠⊠⊠、通常⊠ `#ifdef` ⊠使⊠⊠⊠⊠。

1.2 __has_include 式： ヘッダーファイルの存在を判定する

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

```
__has_include( ⊠⊠⊠⊠名 )
```

`__has_include` 式⊠⊠⊠⊠⊠名⊠存在⊠⊠場合 1 ⊠、存在⊠⊠⊠場合 0 ⊠置換⊠⊠⊠。

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

```
#if __has_include(<filesystem>)
// ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
#include <filesystem>
namespace fs = std::filesystem ;
#else
// 実験的⊠実装⊠使⊠
#include <experimental/filesystem>
namespace fs = std::experimental::filesystem ;
#endif
```

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

```
#ifdef __has_include
// __has_include ⊠⊠⊠⊠⊠⊠⊠⊠⊠
#else
// __has_include ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
#endif
```

`__has_include` 式⊠ `#if` ⊠ `#elif` ⊠中⊠⊠⊠使⊠⊠⊠。

```
int main()
{
        // ⊠⊠⊠
        if ( __has_include(<vector>) )
        { }
    }
```

1.3 __has_cpp_attribute 式

C++ 実装⊠特定⊠属性⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠調⊠⊠⊠⊠、`__has_cpp_attribute` 式⊠使⊠⊠。

```
__has_cpp_attribute( 属性⊠⊠⊠⊠ )
```

`__has_cpp_attribute` 式⊠、属性⊠⊠⊠⊠⊠存在⊠⊠場合⊠属性⊠⊠⊠⊠⊠標準規格⊠採択⊠⊠⊠年⊠月⊠表⊠数値⊠、存在⊠⊠⊠場合⊠ 0 ⊠置換⊠⊠⊠。

```
// [[nodiscard]] ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合⊠使⊠
#if __has_cpp_attribute(nodiscard)
[[nodiscard]]
#endif
void * allocate_memory( std::size_t size ) ;
```

`__has_include` 式⊠同⊠⊠、`__has_cpp_attribute` 式⊠ `#if` ⊠ `#elif` ⊠中⊠⊠⊠使⊠⊠⊠。`#ifdef` ⊠`__has_cpp_attribute` 式⊠存在⊠有無⊠判定⊠⊠⊠。

第 2 章

C++14 のコア言語の新機能

C++14⊠追加⊠⊠⊠新機能⊠少⊠⊠。C++14⊠C++03⊠同⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠位置付⊠⊠積極的⊠新機能⊠追加⊠見送⊠⊠⊠⊠⊠⊠。

2.1 二進数リテラル

二進数⊠⊠⊠⊠⊠整数⊠⊠⊠⊠⊠二進数⊠記述⊠⊠機能⊠。整数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `0B` ⊠⊠⊠⊠ `0b` ⊠書⊠⊠、二進数⊠⊠⊠⊠⊠⊠⊠。整数⊠表現⊠⊠文字⊠0⊠1⊠⊠使⊠⊠⊠。

```
int main()
{
    int x1 = 0b0 ; // 0
    int x2 = 0b1 ; // 1
    int x3 = 0b10 ; // 2
    int x4 = 0b11001100 ; // 204
}
```

二進数⊠⊠⊠⊠⊠浮動小数点数⊠⊠⊠⊠⊠⊠使⊠⊠⊠。

機能⊠⊠⊠⊠⊠⊠⊠ `__cpp_binary_literals`, 値⊠201304。

2.2 数値区切り文字

数値区切⊠文字⊠、整数⊠⊠⊠⊠⊠浮動小数点数⊠⊠⊠⊠⊠数値⊠⊠⊠⊠⊠⊠⊠⊠⊠文字⊠区切⊠⊠⊠⊠⊠⊠⊠機能⊠。区切⊠桁⊠何桁⊠⊠⊠⊠。

```
int main()
{
    int x1 = 123’456’789 ;
    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⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠ 100000000 ⊠ 1000000000 ⊠書⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠大⊠⊠⊠⊠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 ;
```

```
// 関数
// ⊠⊠⊠⊠関数⊠同⊠文法
// ⊠⊠⊠⊠⊠⊠⊠
[[deprecated]] void function_name1() { }
void function_name2 [[deprecated]] () { }


// ⊠⊠⊠
// union ⊠同⊠
class [[deprecated]] class_name
{
// 非 static ⊠⊠⊠⊠⊠⊠⊠
[[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⊠指定⊠⊠⊠名前⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠、C++ ⊠⊠⊠⊠⊠⊠⊠警告⊠⊠⊠⊠⊠⊠出⊠。

`[[deprecated]]` 属性⊠⊠、文字列⊠付⊠加⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠ C++ 実装⊠⊠⊠⊠⊠警告⊠⊠⊠⊠⊠⊠含⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
[[deprecated("Use of f() is deprecated. Use f(int option) instead.")]]
void f() ;
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⊠戻⊠値⊠型⊠⊠⊠ `auto` ⊠指定⊠⊠⊠、戻⊠値⊠型⊠ `return` 文⊠⊠推定⊠⊠⊠⊠⊠。

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

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

`return` 文⊠型⊠一致⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

⊠⊠⊠型⊠決定⊠⊠⊠ `return` 文⊠存在⊠⊠場合、関数⊠戻⊠値⊠型⊠参照⊠⊠⊠⊠⊠⊠書⊠⊠。

```
auto a()
{
    &a ; // ⊠⊠⊠、a ⊠戻⊠値⊠型⊠決定⊠⊠⊠⊠⊠
    return 0 ;
}

auto b()
{
    return 0 ;
    &b ; // OK、戻⊠値⊠型⊠ int
}
```

関数 a ⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠⊠関数 a ⊠型⊠決定⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、return 文⊠前⊠型⊠決定⊠⊠⊠⊠⊠⊠関数 a ⊠⊠⊠⊠⊠⊠⊠。関数 b ⊠ return 文⊠現⊠⊠後⊠⊠⊠戻⊠値⊠型⊠決定⊠⊠⊠。

再帰関数⊠書⊠⊠。

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

⊠⊠⊠⊠⊠⊠、return 文⊠順番⊠逆⊠⊠⊠⊠戻⊠値⊠型⊠決定⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠注意。

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

機能⊠⊠⊠⊠⊠⊠⊠ `__cpp_return_type_deduction`, 値⊠ 201304。

2.5 decltype(auto)：厳格な auto

警告：⊠⊠項目⊠ C++ 規格⊠詳細⊠知識⊠解説⊠⊠⊠⊠⊠⊠極⊠⊠難解⊠⊠⊠⊠⊠⊠。平均的⊠ C++ ⊠⊠⊠⊠⊠⊠⊠⊠⊠知識⊠得⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠書⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠項目⊠読⊠飛⊠⊠⊠⊠⊠⊠⊠。

`decltype(auto)` ⊠ `auto` 指定子⊠代⊠⊠⊠使⊠⊠厳格⊠ `auto` ⊠。利用⊠⊠ C++ ⊠規格⊠厳格⊠理解⊠求⊠⊠⊠⊠。

`auto` ⊠ `decltype(auto)` ⊠型指定子⊠呼⊠⊠⊠文法⊠一種⊠、⊠⊠⊠⊠⊠⊠⊠⊠型⊠⊠⊠使⊠。

⊠⊠⊠⊠⊠⊠言⊠⊠、具体的⊠型⊠式⊠⊠決定⊠⊠機能⊠。

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

```
auto b() { return 0 ; }
```

変数宣言⊠⊠⊠⊠⊠⊠⊠⊠⊠型⊠使⊠場合、型⊠決定⊠⊠⊠⊠⊠式⊠初期化子⊠呼⊠⊠⊠部分⊠書⊠⊠⊠式⊠使⊠。関数⊠戻⊠値⊠型推定⊠⊠⊠⊠⊠⊠⊠⊠⊠型⊠使⊠場合、return 文⊠式⊠使⊠。

decltype(auto) ⊠ auto ⊠代⊠⊠⊠使⊠⊠⊠⊠⊠⊠⊠。decltype(auto) ⊠型⊠式⊠⊠決定⊠⊠。

```
// a ⊠ int
decltype(auto) a = 0 ;
// b ⊠ int
decltype(auto) b() { return 0 ; }
```

一見⊠⊠⊠ auto ⊠ decltype(auto) ⊠同⊠⊠⊠⊠。⊠⊠⊠、⊠⊠ 2 ⊠⊠式⊠⊠型⊠決定⊠⊠方法⊠違⊠。⊠⊠⊠⊠ C++ ⊠規格⊠極⊠⊠難⊠⊠規則⊠基⊠⊠⊠決定⊠⊠⊠。習得⊠⊠熟練⊠魔法使⊠⊠⊠⊠⊠⊠⊠要求⊠⊠⊠。

auto ⊠式⊠⊠型⊠決定⊠⊠⊠⊠、auto ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠名⊠置⊠換⊠⊠関数⊠⊠⊠⊠⊠⊠⊠仮引数⊠、式⊠実引数⊠⊠⊠渡⊠⊠⊠⊠⊠⊠⊠⊠実引数推定⊠行⊠⊠⊠場合⊠推定⊠⊠⊠型⊠使⊠⊠⊠。

⊠⊠⊠⊠

```
auto x = 0 ;
```

⊠場合⊠、

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

⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠⊠対⊠⊠、

```
f(0) ;
```

⊠実引数⊠渡⊠⊠⊠⊠⊠ u ⊠型⊠⊠⊠推定⊠⊠⊠型⊠同⊠型⊠⊠⊠。

```
int i ;
auto const * x = &i ;
```

⊠場合⊠⊠、

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

⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠⊠

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

⊠実引数⊠渡⊠⊠⊠⊠⊠ u ⊠型⊠⊠⊠推定⊠⊠⊠型⊠同⊠型⊠⊠⊠。⊠⊠場合⊠ `int const *` ⊠⊠⊠。

⊠⊠⊠⊠⊠ `auto` ⊠説明⊠。`decltype(auto)` ⊠説明⊠簡単⊠。

`decltype(auto)` ⊠型⊠、`auto` ⊠式⊠置⊠換⊠⊠ `decltype` ⊠型⊠⊠⊠。

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

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

上⊠⊠⊠⊠⊠、下⊠⊠⊠⊠⊠同⊠意味⊠。

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

⊠⊠⊠⊠⊠簡単⊠。⊠⊠⊠、⊠⊠以降⊠黒魔術⊠⊠⊠⊠ C++ ⊠規格⊠知識⊠必要⊠⊠⊠⊠⊠⊠。

`auto` ⊠ `decltype(auto)` ⊠一見⊠⊠⊠同⊠⊠⊠⊠見⊠⊠。型⊠決定⊠⊠方法⊠⊠⊠、`auto` ⊠関数⊠⊠⊠⊠⊠⊠⊠実引数推定⊠使⊠、`decltype(auto)` ⊠ `decltype` ⊠使⊠。⊠⊠⊠⊠式⊠評価⊠⊠結果⊠型⊠⊠⊠。⊠⊠⊠⊠何⊠違⊠⊠⊠⊠⊠⊠。

主⊠違⊠⊠、`auto` ⊠関数呼⊠出⊠⊠使⊠⊠⊠⊠⊠⊠。関数呼⊠出⊠⊠際⊠⊠⊠⊠⊠⊠暗黙⊠型変換⊠行⊠⊠⊠。

⊠⊠⊠⊠、配列⊠関数⊠渡⊠⊠、暗黙⊠型変換⊠結果、配列⊠先頭要素⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

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

⊠⊠ `auto` ⊠ `decltype(auto)` ⊠使⊠⊠⊠⊠⊠⊠⊠⊠。

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

```
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⊠配列⊠先頭要素⊠⊠⊠⊠⊠⊠⊠⊠⊠暗黙⊠変換⊠⊠⊠⊠⊠、結果⊠⊠⊠⊠⊠正⊠⊠。

decltype(auto) ⊠場合、型⊠ int [5] ⊠⊠⊠。配列⊠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⊠

⊠⊠⊠⊠⊠⊠⊠⊠。

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

decltype(auto) ⊠単体⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

⊠⊠他⊠⊠ auto ⊠ decltype(auto) ⊠⊠⊠⊠⊠⊠⊠違⊠⊠⊠⊠。⊠⊠⊠⊠違⊠⊠列挙⊠⊠⊠⊠煩雑⊠⊠⊠省略⊠⊠⊠、decltype(auto) ⊠式⊠型⊠直接使⊠。auto ⊠⊠⊠⊠⊠⊠場合⊠便利⊠型⊠変換⊠入⊠。

auto ⊠便利⊠⊠⊠⊠⊠⊠場合⊠⊠⊠⊠⊠⊠⊠暗黙⊠型⊠変換⊠入⊠⊠⊠、意図⊠⊠⊠⊠推定⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠、引数⊠⊠⊠⊠⊠⊠⊠⊠受⊠取⊠、戻⊠値⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠返⊠関数⊠書⊠⊠⊠⊠。以下⊠⊠⊠⊠書⊠⊠⊠間違⊠⊠。

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

⊠⊠⊠⊠⊠、戻⊠値⊠型⊠式⊠型⊠⊠変化⊠⊠ int ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠ decltype(auto) ⊠使⊠⊠、

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

式⊠型⊠⊠⊠⊠⊠使⊠⊠⊠⊠⊠。

⊠⊠⊠式⊠ decltype(auto) ⊠使⊠場合⊠以下⊠⊠⊠⊠書⊠。

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

decltype(auto) ⊠主⊠関数⊠戻⊠値⊠型推定⊠式⊠型⊠⊠⊠⊠⊠推定⊠⊠⊠⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠追加⊠⊠⊠機能⊠。⊠⊠利用⊠⊠ C++ ⊠型⊠⊠⊠⊠⊠深⊠理解⊠必要⊠⊠⊠。

機能⊠⊠⊠⊠⊠⊠⊠ `__cpp_decltype_auto`, 値⊠ 201304。

2.6　ジェネリックラムダ

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠式⊠引数⊠型⊠書⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠機能⊠。

通常⊠⊠⊠⊠式⊠以下⊠⊠⊠⊠書⊠。

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

⊠⊠⊠式⊠引数⊠⊠型⊠必要⊠。⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `operator ()` ⊠渡⊠型⊠⊠⊠⊠⊠⊠時⊠⊠⊠⊠。⊠⊠⊠⊠⊠時⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠人間⊠指定⊠⊠必要⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠、引数⊠型⊠書⊠⊠⊠場所⊠ `auto` ⊠⊠⊠⊠⊠⊠書⊠⊠⊠⊠型⊠推定⊠⊠⊠⊠⊠。

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

⊠⊠⊠⊠⊠⊠⊠⊠⊠式⊠結果⊠⊠⊠⊠⊠⊠⊠型⊠⊠呼⊠出⊠⊠⊠⊠違⊠型⊠渡⊠⊠⊠⊠⊠⊠⊠。

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

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

仕組⊠⊠簡単⊠、以下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `operator ()` ⊠持⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠生成⊠⊠⊠⊠⊠⊠⊠⊠。

```
struct closure_object
{
        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⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠変数⊠名前⊠式⊠書⊠⊠⊠⊠⊠⊠⊠機能⊠。

⊠⊠⊠式⊠書⊠⊠⊠場所⊠⊠見⊠⊠⊠⊠⊠⊠⊠変数⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

初期化⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠初期化子⊠書⊠⊠⊠⊠⊠⊠⊠機能⊠。

```
int main()
{
    int x = 0 ;
    [ x = x, y = x, &ref = x, x2 = x * 2 ]
    {// ⊠⊠⊠⊠⊠⊠⊠⊠⊠変数⊠使⊠
        x ;
        y ;
        ref ;
        x2 ;
    } ;
}
```

初期化⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、“識別子 `=` `expr`” ⊠⊠⊠文法⊠⊠⊠⊠導入子 `[]` ⊠中⊠書⊠。⊠⊠⊠⊠⊠⊠⊠ “`auto` 識別子 `=` `expr ;`” ⊠書⊠⊠⊠⊠⊠⊠⊠変数⊠作⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠変数⊠名前⊠変⊠⊠⊠、⊠⊠⊠⊠新⊠⊠変数⊠宣言⊠⊠⊠⊠⊠⊠⊠⊠。

初期化⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠識別子⊠前⊠ & ⊠付⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠扱⊠⊠⊠⊠。

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

    // x ⊠ 1
}
```

初期化⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠追加⊠⊠⊠理由⊠⊠変数⊠名前⊠変⊠⊠⊠⊠⊠⊠⊠新⊠⊠変数⊠導入⊠⊠⊠⊠⊠⊠目的⊠他⊠、非 static ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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 ;
}
```

`X::get_closure_object` ⊠ `X::data` ⊠返⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠返⊠。

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

⊠⊠⊠見⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `[=]` ⊠使⊠⊠⊠⊠⊠⊠、`data` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠内⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠思⊠⊠。⊠⊠⊠、⊠⊠⊠式⊠非 `static` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠式⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `this` ⊠⊠⊠⊠⊠⊠。上⊠⊠⊠⊠⊠下⊠⊠⊠⊠⊠同⊠意味⊠⊠⊠。

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

⊠⊠、`main` 関数⊠⊠⊠一度見⊠⊠⊠⊠。

```
int main()
{
    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠代入⊠⊠⊠⊠⊠変数
    std::function< int() > f ;

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

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

⊠⊠⊠、⊠⊠⊠破棄⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠参照⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠未定義⊠動作⊠。

初期化⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠、非 `static` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠存在⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠特殊⊠⊠⊠⊠⊠⊠⊠初期化⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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⊠⊠⊠⊠⊠⊠⊠⊠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⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠型⊠⊠⊠使⊠⊠。

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

```
{
public :
    T member ;
} ;

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

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

C++ ⊠⊠関数⊠宣言⊠⊠⊠。

```
int f( int x )
{ return x ; }
```

C++ ⊠⊠関数⊠⊠⊠⊠⊠⊠⊠宣言⊠⊠⊠。型⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠型⊠⊠⊠使⊠⊠。

```
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⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠宣言⊠⊠⊠⊠。

```
using type = int ;
```

C++11 ⊠⊠⊠⊠⊠⊠⊠宣言⊠⊠⊠⊠⊠⊠⊠宣言⊠⊠⊠。型⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠型⊠⊠⊠使⊠⊠。

```
template < typename T >
using type = T ;

int main()
```

```
{
    type<int> i = 42 ; // int
    type<double> d = 1.0 ; // double
}
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠見⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。C++ ⊠⊠一部⊠宣言⊠⊠⊠⊠⊠⊠⊠宣言⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠踏⊠⊠⊠以下⊠考⊠⊠⊠⊠⊠。

C++ ⊠⊠変数⊠宣言⊠⊠⊠。

```
int variable{} ;
```

C++14 ⊠⊠変数宣言⊠⊠⊠⊠⊠⊠⊠宣言⊠⊠⊠。型⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠型⊠⊠⊠使⊠⊠。

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

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

変数⊠⊠⊠⊠⊠⊠⊠名前⊠⊠⊠変数宣言⊠⊠⊠⊠⊠⊠⊠宣言⊠⊠⊠機能⊠。変数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠宣言⊠⊠⊠、名前空間⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠中⊠⊠⊠書⊠⊠⊠⊠⊠⊠⊠⊠。

```
// ⊠⊠⊠⊠⊠⊠⊠⊠名前空間⊠⊠⊠⊠⊠⊠⊠特別⊠名前空間⊠⊠⊠⊠

namespace ns {
// 名前空間⊠⊠⊠⊠
}

class
{
// ⊠⊠⊠⊠⊠⊠⊠
} ;
```

変数⊠⊠⊠⊠⊠⊠⊠使⊠道⊠主⊠ 2 ⊠⊠⊠。

2.8.1　意味は同じだが型が違う定数

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠変数化⊠⊠⊠⊠⊠⊠良⊠作法⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠円周率⊠ 3.14... ⊠⊠⊠書⊠⊠⊠⊠ pi ⊠⊠⊠変数名⊠扱⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。変数化⊠⊠⊠、円周率⊠値⊠後⊠変⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠変更⊠⊠⊠⊠楽⊠⊠⊠。

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

⊠⊠⊠、円周率⊠表現⊠⊠型⊠複数⊠⊠場合⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠名前⊠分⊠⊠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⊠⊠⊠⊠⊠⊠⊠⊠⊠
const Real pi_r("3.141592653589793238462643383279") ;
```

⊠⊠⊠⊠⊠⊠、使⊠側⊠型⊠⊠⊠⊠名前⊠変⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

関数⊠⊠⊠⊠⊠⊠⊠使⊠⊠⊠⊠手⊠⊠⊠。

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

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

```
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⊠⊠⊠⊠⊠⊠⊠使⊠⊠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 ⊠⊠ std::integral_constant ⊠ 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++ ⊠標準⊠⊠⊠⊠⊠⊠⊠従来⊠ `traits` ⊠⊠⊠⊠⊠⊠変数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `_v` 版⊠用意⊠⊠⊠⊠。

機能⊠⊠⊠⊠⊠⊠⊠ `__cpp_variable_templates`, 値⊠ 201304。

2.9 constexpr 関数の制限緩和

C++11 ⊠追加⊠⊠⊠ `constexpr` 関数⊠⊠⊠⊠制限⊠強⊠。`constexpr` 関数⊠本体⊠⊠実質 `return` 文 1 ⊠⊠⊠書⊠⊠⊠。

C++14 ⊠⊠、⊠⊠⊠⊠何⊠⊠書⊠⊠⊠⊠⊠⊠⊠⊠。

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

    // 繰⊠返⊠文⊠書⊠⊠
    for ( int i = 1 ; i < x ; ++i )
    {
        // 変数⊠変更⊠⊠⊠
        sum += i ;
    }

    return sum ;
}
```

機能⊠⊠⊠⊠⊠⊠⊠ `__cpp_constexpr`, 値⊠ 201304。

C++11 ⊠ `constexpr` 関数⊠対応⊠⊠⊠⊠ C++14 ⊠ `constexpr` 関数⊠対応⊠⊠⊠⊠⊠ C++ 実装⊠⊠、`__cpp_constexpr` ⊠⊠⊠⊠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 ⊠⊠⊠⊠⊠⊠初期化子⊠⊠⊠⊠⊠⊠⊠初期化⊠組⊠合⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠初期化子⊠⊠⊠⊠⊠⊠⊠非 `static` ⊠⊠⊠⊠⊠⊠⊠⊠ `=` ⊠初期化⊠⊠⊠ C++11 ⊠機能⊠。

```
struct S
```

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

⊠⊠⊠⊠⊠⊠初期化⊠⊠⊠⊠⊠⊠⊠⊠⊠条件⊠満⊠⊠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⊠⊠明示的⊠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 ⊠⊠ 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⊠⊠。

```
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 ⊠新機能⊠⊠⊠⊠⊠⊠終⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠ C++17 ⊠⊠⊠言語⊠新機能⊠解説⊠⊠⊠⊠。

C++17 ⊠⊠⊠言語⊠新機能⊠⊠、C++11 ⊠⊠⊠大⊠⊠⊠⊠⊠⊠⊠。

3.1　トライグラフの廃止

C++17 ⊠⊠⊠⊠⊠⊠⊠⊠⊠廃止⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠知⊠⊠⊠読者⊠⊠⊠変更⊠気⊠⊠⊠必要⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠知⊠⊠⊠⊠読者⊠⊠⊠⊠⊠気⊠⊠⊠必要⊠⊠⊠。

3.2　16 進数浮動小数点数リテラル

C++17 ⊠⊠浮動小数点数⊠⊠⊠⊠⊠ 16 進数⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

16 進数浮動小数点数⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠ `0x` ⊠続⊠⊠仮数部⊠ 16 進数（`0123456789abcdefABCDEF`）⊠書⊠、`p` ⊠⊠⊠⊠ `P` ⊠続⊠⊠指数部⊠ 10 進数⊠書⊠。

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

指数部⊠ `e` ⊠⊠⊠⊠ `p` ⊠ `P` ⊠使⊠。

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

16 進数浮動小数点数⊠⊠⊠⊠⊠⊠、指数部⊠省略⊠⊠⊠⊠。

```
int a = 0x1 ; // 整数⊠⊠⊠⊠
```

```
0x1.0 ; // ⊠⊠⊠、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⊠⊠⊠⊠⊠⊠⊠。

```
// ⊠⊠⊠
// U+3042 ⊠ UTF-8 ⊠ 0xE3, 0x81, 0x82 ⊠⊠⊠ 3 ⊠⊠⊠⊠⊠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⊠⊠⊠⊠⊠⊠⊠
    void (*p)() noexcept = &f ;

    // 無例外指定⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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 ;
```

```
// 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) ⊠ 1 番目⊠引数⊠ x ⊠、残⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ args ⊠受⊠取⊠。⊠⊠⊠、x + sum(args ...) ⊠返⊠。⊠⊠⊠、sum(args ...) ⊠⊠⊠ sum(x, args) ⊠渡⊠⊠⊠、1 番目⊠引数、⊠⊠⊠最初⊠⊠見⊠ 2 番目⊠引数⊠ x ⊠入⊠、⊠⊠ sum ⊠呼⊠⊠⊠。⊠⊠⊠⊠⊠再帰的⊠処理⊠繰⊠返⊠⊠⊠⊠。

⊠⊠⊠、引数⊠ 1 ⊠⊠⊠⊠⊠⊠⊠、可変長⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ sum ⊠呼⊠⊠⊠。⊠⊠⊠重要⊠。⊠⊠⊠⊠⊠可変長⊠⊠⊠⊠⊠⊠⊠ 0 個⊠引数⊠取⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠可変長⊠⊠⊠⊠⊠⊠版⊠ sum ⊠呼⊠⊠⊠⊠⊠⊠⊠、次⊠ sum ⊠呼⊠出⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠回避⊠⊠⊠⊠⊠、⊠⊠再帰⊠終了条件⊠⊠⊠⊠、引数⊠ 1 ⊠⊠ sum ⊠⊠⊠⊠⊠⊠⊠⊠関数⊠書⊠⊠⊠⊠。

可変長⊠⊠⊠⊠⊠⊠⊠⊠任意個⊠引数⊠対応⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠再帰的⊠⊠⊠⊠⊠必須⊠⊠⊠。

⊠⊠⊠、⊠⊠⊠実現⊠⊠⊠⊠⊠⊠⊠ N 個⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ args ⊠中身⊠対⊠⊠、仮⊠ N 番目⊠ args#N ⊠⊠⊠表記⊠使⊠⊠、args#0 + args#1 + ... + args#N-1 ⊠⊠⊠⊠展開⊠⊠⊠⊠⊠⊠⊠。C++17 ⊠ fold 式⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠対⊠⊠二項演算子⊠適用⊠⊠展開⊠行⊠機能⊠。

fold 式⊠使⊠⊠ sum ⊠以下⊠⊠⊠⊠書⊠⊠。

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

(... + args) ⊠、args#0 + args#1 + ... + args#N-1 ⊠⊠⊠⊠展開⊠⊠⊠。

fold 式⊠⊠、単項 fold 式⊠二項 fold 式⊠⊠⊠。⊠⊠⊠、演算子⊠結合順序⊠合⊠⊠⊠左 fold ⊠右 fold ⊠⊠⊠。

fold 式⊠必⊠括弧⊠囲⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

単項 fold 式⊠文法⊠以下⊠⊠⊠⊠⊠⊠⊠⊠。

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

例：

```
template < typename ... Types >
void f( Types ... args )
{
    // 単項左 fold
    ( ... + args )  ;
    // 単項右 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⊠⊠左 fold ⊠右 fold ⊠⊠⊠。

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

```
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⊠⊠⊠⊠交換法則⊠満⊠⊠⊠⊠型⊠ fold 式⊠適用⊠⊠際⊠⊠注意⊠必要⊠。

二項 fold 式⊠文法⊠以下⊠⊠⊠⊠⊠⊠⊠⊠。

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

左右⊠ `cast-expression` ⊠⊠⊠⊠⊠片方⊠⊠⊠未展開⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠入⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。2 ⊠⊠ `fold-operator` ⊠同⊠演算子⊠⊠⊠⊠⊠⊠⊠⊠⊠。

`( e1 op1 ... op2 e2 )` ⊠⊠⊠二項 fold 式⊠⊠⊠⊠⊠⊠、e1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合⊠二項右 fold 式、e2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合⊠二項左 fold 式⊠⊠⊠。

```
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)))))
    auto right = ( args + ... + 0 ) ;
}

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

fold 式⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠二項演算子⊠適用⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠複雑⊠再帰的⊠⊠⊠⊠⊠⊠⊠書⊠⊠⊠⊠⊠方法⊠提供⊠⊠⊠⊠⊠。

機能⊠⊠⊠⊠⊠⊠⊠ `__cpp_fold_expressions`, 値⊠ 201603。

3.6　ラムダ式で *this のコピーキャプチャー

C++17 ⊠⊠⊠⊠⊠式⊠ `*this` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。`*this` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `*this` ⊠書⊠。

```
struct X
{
   int data = 42 ;
   auto get()
   {
       return [*this]() { return this->data ; } ;
   }
} ;

int main()
{
    std::function < int () > f ;
    {
        X x ;
        f = x.get() ;
    }// x ⊠寿命⊠⊠⊠⊠⊠

    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠問題⊠⊠
    int data = f() ;
}
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `*this` ⊠⊠⊠⊠式⊠書⊠⊠⊠場所⊠ `*this` ⊠。⊠⊠、以下⊠⊠⊠⊠⊠⊠⊠⊠挙動⊠違⊠⊠見⊠⊠⊠⊠⊠⊠⊠⊠。

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

        // this->data ⊠ 1

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

        // OK、mutable ⊠使⊠⊠⊠⊠

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

        // this->data ⊠ 1
        // 変更⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠内⊠*this
   }
} ;
```

最初⊠⊠⊠⊠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⊠⊠⊠⊠式⊠⊠以下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠生成⊠⊠⊠。

```
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 ;
    }
} ;
```

⊠⊠⊠ C++ ⊠文法⊠従⊠⊠⊠⊠⊠⊠⊠⊠⊠苦⊠⊠⊠⊠⊠例⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠値⊠変更⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

3番目⊠⊠⊠⊠式⊠⊠以下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠生成⊠⊠⊠。

```
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 ;
    }
} ;
```

⊠⊠⊠式⊠ mutable ⊠付⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠値⊠変更⊠⊠⊠。

*this ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合、this ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

```
    {
        // this ⊠⊠⊠⊠⊠⊠⊠関数 f ⊠呼⊠出⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
        std::printf("%p\n", this) ;

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

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

⊠⊠場合、出力⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠値⊠異⊠⊠。

⊠⊠⊠式⊠⊠ `*this` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠名前⊠⊠⊠ `*this` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠提供⊠⊠提案⊠。同等⊠機能⊠初期化⊠⊠⊠⊠⊠⊠⊠⊠可能⊠⊠、表記⊠冗長⊠間違⊠⊠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 ⊠⊠⊠⊠⊠式⊠ `constexpr` ⊠⊠⊠⊠。⊠⊠正確⊠説明⊠⊠⊠、⊠⊠⊠式⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `operator ()` ⊠条件⊠満⊠⊠場合 `constexpr` ⊠⊠⊠。

```
int main()
{
    auto f = []{ return 42 ; } ;
    constexpr int value = f() ; // OK
}
```

constexpr ⊠条件⊠満⊠⊠⊠⊠⊠式⊠⊠⊠⊠⊠⊠時定数⊠必要⊠⊠⊠場所⊠使⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠ constexpr 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⊠満⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
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⊠満⊠⊠⊠⊠
    constexpr int c = f() ;

    // OK、constexpr ⊠条件⊠満⊠⊠
    constexpr int d = g() ;
}
```

以下⊠内容⊠上級者向⊠⊠解説⊠⊠⊠、通常⊠読者⊠理解⊠⊠必要⊠⊠⊠。

constexpr ⊠⊠⊠式⊠ SFINAE ⊠文脈⊠使⊠⊠⊠⊠⊠⊠⊠⊠。

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

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

⊠⊠⊠⊠⊠、⊠⊠⊠許⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠仮引数⊠対⊠⊠任意⊠式⊠文⊠⊠⊠⊠⊠⊠⊠⊠ Substitution ⊠失敗⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

上級者向⊠⊠解説終⊠⊠。

機能⊠⊠⊠⊠⊠⊠⊠ `__cpp_constexpr`, 値⊠ 201603。

`__cpp_constexpr` ⊠⊠⊠⊠値⊠、C++11 ⊠時点⊠ 200704、C++14 ⊠時点⊠ 201304 ⊠。

3.8　文字列なし static_assert

C++17 ⊠⊠ `static_assert` ⊠文字列⊠⊠⊠⊠⊠取⊠⊠⊠⊠⊠⊠追加⊠⊠⊠。

```
static_assert( true ) ;
```

C++11 ⊠追加⊠⊠⊠ `static_assert` ⊠⊠、文字列⊠⊠⊠⊠⊠必須⊠⊠⊠。

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

特⊠文字列⊠指定⊠⊠必要⊠⊠⊠場合⊠⊠⊠⊠⊠、文字列⊠⊠⊠⊠⊠取⊠⊠⊠ `static_assert` ⊠追加⊠⊠⊠。

機能⊠⊠⊠⊠⊠⊠⊠ `__cpp_static_assert`, 値⊠ 201411。

C++11 ⊠時点⊠ `__cpp_static_assert` ⊠値⊠ 200410。

3.9　ネストされた名前空間定義

C++17 ⊠⊠⊠⊠⊠⊠⊠⊠名前空間⊠定義⊠楽⊠書⊠⊠。

⊠⊠⊠⊠⊠⊠名前空間⊠⊠、`A::B::C` ⊠⊠⊠⊠名前空間⊠中⊠名前空間⊠入⊠⊠⊠⊠名前空間⊠⊠⊠⊠。

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

C++17 ⊠⊠、上記⊠⊠⊠⊠⊠同⊠⊠⊠⊠以下⊠⊠⊠⊠書⊠⊠。

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

機能⊠⊠⊠⊠⊠⊠⊠ `__cpp_nested_namespace_definitions`, 値⊠ 201411。

3.10　[[fallthrough]] 属性

`[[fallthrough]]` 属性⊠ `switch` 文⊠中⊠ `case` ⊠⊠⊠⊠突⊠抜⊠⊠⊠⊠⊠⊠⊠⊠⊠出⊠⊠⊠使⊠⊠。

`switch` 文⊠⊠対応⊠⊠ `case` ⊠⊠⊠⊠処理⊠移⊠。通常、以下⊠⊠⊠⊠書⊠。

```
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 ⊠⊠⊠⊠⊠⊠⊠⊠場合⊠処理
        break ;
    }
}
```

⊠⊠例⊠以下⊠⊠⊠⊠書⊠⊠

```
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☒☒書☒☒☒☒☒☒☒☒☒☒。☒☒☒☒、賢☒ C++ ☒☒☒☒☒☒☒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☒☒警告☒☒☒☒☒☒☒。☒☒☒☒☒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☒☒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☒値☒無視☒☒☒☒☒☒☒☒☒☒☒使☒☒☒☒☒☒。[[nodiscard]] 属性☒付与☒☒☒関数☒戻☒値☒無視☒☒☒警告☒☒☒☒☒☒表示☒☒☒。

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

void g( int ) { }

int main()
{
    // ☒☒☒、戻☒値☒無視☒☒☒☒☒
    f() ;
    // OK、戻⊠値⊠無視⊠⊠⊠⊠⊠⊠
    int result = f() ;
    g( f ) ;
    f() + 1 ;
    (void) f() ;
}
```

戻⊠値⊠無視⊠⊠、⊠⊠⊠⊠⊠万能⊠⊠⊠⊠。上⊠例⊠⊠、意味的⊠⊠戻⊠値⊠無視⊠⊠⊠⊠⊠⊠言⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠場合⊠戻⊠値⊠無視⊠⊠⊠⊠⊠⊠⊠考⊠⊠⊠。

`[[nodiscard]]` ⊠目的⊠、戻⊠値⊠無視⊠⊠⊠⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠利用⊠⊠⊠⊠⊠初歩的⊠間違⊠⊠防⊠⊠⊠⊠⊠⊠。`void` 型⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠意図的⊠戻⊠値⊠無視⊠⊠防⊠⊠⊠⊠⊠作⊠⊠⊠⊠⊠⊠。

`[[nodiscard]]` 属性⊠使⊠⊠⊠関数⊠、戻⊠値⊠無視⊠⊠⊠⊠⊠⊠⊠関数⊠。⊠⊠⊠⊠⊠関数⊠戻⊠値⊠無視⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠大⊠⊠ 2 ⊠⊠⊠。

戻⊠値⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確認⊠⊠⊠⊠⊠⊠⊠⊠⊠情報⊠通知⊠使⊠関数。

```
enum struct error_code
{
    no_error, some_operations_failed,  serious_error
} ;

// 失敗⊠⊠⊠⊠⊠⊠⊠⊠処理
error_code do_something_that_may_fail()
{
    // 処理

    if ( is_error_condition() )
        return error_code::serious_error ;

    // 処理

    return error_code::no_error ;
}

// ⊠⊠⊠⊠⊠⊠⊠⊠発生⊠⊠⊠⊠⊠⊠⊠⊠処理
int do_something_on_no_error() ;

int main()
{
    // ⊠⊠⊠⊠確認⊠⊠⊠⊠⊠
    do_something_that_may_fail() ;

    // ⊠⊠⊠⊠⊠⊠前提⊠次⊠処理⊠⊠⊠⊠⊠⊠⊠
    do_something_on_no_error() ;
}
```

関数⊠ [[nodiscard]] 属性⊠付与⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠側⊠初歩的⊠⊠⊠⊠確認⊠欠如⊠警告⊠⊠⊠⊠⊠⊠出⊠⊠。

[[nodiscard]] 属性⊠、⊠⊠⊠⊠ enum ⊠⊠付与⊠⊠⊠⊠⊠⊠⊠⊠。

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

[[nodiscard]] ⊠付与⊠⊠⊠⊠⊠⊠⊠ enum ⊠戻⊠値⊠型⊠⊠⊠関数⊠[[nodiscard]] ⊠付与⊠⊠⊠扱⊠⊠⊠⊠。

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

X f() { return X{} ; }

int main()
{
    // 警告、戻⊠値⊠無視⊠⊠⊠⊠⊠
    f() ;
}
```

機能⊠⊠⊠⊠⊠⊠⊠ `__has_cpp_attribute(nodiscard)`, 値⊠ 201603。

3.12 [[maybe_unused]] 属性

[[maybe_unused]] 属性⊠名前⊠⊠⊠⊠⊠⊠⊠⊠意図的⊠使⊠⊠⊠⊠⊠⊠⊠示⊠⊠⊠使⊠⊠。

現実⊠ C++ ⊠⊠⊠⊠⊠⊠、宣言⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠考慮⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠見⊠⊠名前⊠⊠⊠⊠⊠⊠⊠⊠存在⊠⊠。

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

void f()
{
    int x[5] ;
    char reserved[1024] = { } ;
    int y[5] ;

    do_something( x, y ) ;
}
```

⊠⊠⊠⊠ `reserved` ⊠⊠⊠名前⊠⊠⊠⊠⊠⊠使⊠⊠⊠⊠⊠⊠。一見⊠⊠⊠不必要⊠名前⊠見⊠⊠。優秀⊠ C++ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠⊠⊠⊠⊠名前⊠対⊠⊠「⊠⊠⊠⊠⊠使⊠⊠⊠⊠⊠⊠」⊠⊠⊠警告⊠⊠⊠⊠⊠⊠出⊠。

⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠見⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠理由⊠ `reserved` ⊠⊠⊠⊠一見使⊠⊠⊠⊠⊠⊠変数⊠必要⊠⊠⊠。

⊠⊠⊠⊠、`reserved` ⊠⊠⊠⊠⊠破壊⊠検出⊠⊠⊠⊠⊠領域⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠ C++ 以外⊠言語⊠書⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠ OS ⊠外部⊠⊠⊠⊠⊠読⊠書⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠⊠理由⊠⊠⊠、名前⊠⊠⊠⊠⊠⊠⊠⊠一見使⊠⊠⊠⊠⊠⊠⊠⊠⊠見⊠⊠⊠存在⊠必要⊠⊠⊠⊠⊠⊠意味⊠表⊠⊠⊠、`[[maybe_unused]]` 属性⊠使⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠、C++ ⊠⊠⊠⊠⊠⊠⊠「未使用⊠名前」⊠⊠⊠警告⊠⊠⊠⊠⊠⊠抑制⊠⊠⊠。

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

`[[maybe_unused]]` 属性⊠適用⊠⊠⊠名前⊠⊠⊠⊠⊠⊠⊠⊠宣言⊠、⊠⊠⊠、`typedef` 名、変数、非 `static` ⊠⊠⊠⊠⊠⊠⊠、関数、`enum`, `enumerator` ⊠。

```
// ⊠⊠⊠
class [[maybe_unused]] class_name
{
// 非 static ⊠⊠⊠⊠⊠⊠⊠
    [[maybe_unused]] int non_static_data_member ;

} ;

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

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

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

// 関数
// ⊠⊠⊠⊠関数⊠同⊠文法
// ⊠⊠⊠⊠⊠⊠⊠
[[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⊠⊠⊠⊠⊠⊠⊠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
```

⊠⊠⊠、

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

⊠書⊠⊠場合、関数 f ⊠⊠⊠先⊠呼⊠出⊠⊠⊠、次⊠関数 g ⊠呼⊠出⊠⊠⊠⊠⊠⊠保証⊠⊠⊠。

関数呼⊠出⊠⊠実引数⊠⊠⊠⊠⊠⊠ b1, b2, b3 ⊠評価順序⊠未規定⊠⊠⊠⊠。

⊠⊠⊠⊠⊠、既存⊠未定義⊠動作⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠挙動⊠定⊠⊠。

3.14　constexpr if 文：コンパイル時条件分岐

constexpr if 文⊠⊠⊠⊠⊠⊠時⊠条件分岐⊠⊠⊠⊠機能⊠。

constexpr if 文⊠、通常⊠ if 文⊠ if constexpr ⊠置⊠換⊠⊠。

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

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

constexpr if 文⊠⊠⊠名前⊠⊠、実際⊠記述⊠⊠⊠⊠⊠ if constexpr ⊠。

⊠⊠⊠⊠⊠時⊠条件分岐⊠⊠何⊠意味⊠⊠⊠⊠。以下⊠ constexpr if ⊠行⊠⊠⊠⊠⊠⊠一覧⊠。

- 最適化
- 非⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠挙動⊠変化

⊠⊠⊠⊠⊠時⊠条件分岐⊠機能⊠理解⊠⊠⊠⊠、⊠⊠ C++ ⊠既存⊠条件分岐⊠⊠⊠⊠理解⊠⊠必要⊠⊠⊠。

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⊠、実行時⊠条件分岐⊠行⊠。

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

⊠⊠場合、runtime_value ⊠ true ⊠場合⊠関数 do_true_thing ⊠呼⊠⊠、false ⊠場合⊠関数 do_false_thing ⊠呼⊠⊠⊠。

実行時⊠条件分岐⊠条件⊠⊠、⊠⊠⊠⊠⊠時定数⊠指定⊠⊠⊠。

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

⊠⊠場合、賢⊠⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠処理⊠最適化⊠⊠⊠⊠⊠⊠⊠⊠。

```
do_true_thing() ;
```

⊠⊠⊠⊠⊠、条件⊠常⊠ true ⊠⊠⊠⊠。⊠⊠⊠⊠⊠最適化⊠実行時⊠条件分岐⊠⊠⊠⊠⊠⊠⊠時⊠行⊠⊠。⊠⊠⊠⊠⊠時⊠条件分岐⊠⊠⊠⊠⊠⊠最適化⊠目的⊠⊠⊠⊠。

⊠⊠一度⊠⊠⊠例⊠戻⊠⊠。今度⊠完全⊠⊠⊠⊠⊠見⊠⊠⊠⊠。

```
// 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() ; // ⊠⊠⊠
}
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠理由⊠、do_false_thing ⊠⊠⊠名前⊠宣言⊠⊠⊠⊠⊠⊠⊠⊠⊠。C++ ⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠時⊠⊠⊠⊠⊠以下⊠形⊠変形⊠⊠⊠⊠⊠最適化⊠⊠⊠⊠⊠⊠⊠⊠⊠、

```
void do_true_thing() ;

void f( bool runtime_value )
```

```
{
    do_true_thing() ;
}
```

最適化⊠結果失⊠⊠⊠⊠⊠⊠、依然⊠⊠⊠⊠⊠⊠⊠⊠時⊠⊠⊠⊠⊠⊠⊠検証⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠検証⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠誤⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。名前 `do_false_thing` ⊠宣言⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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⊠⊠⊠⊠⊠⊠⊠時⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠時計算⊠⊠⊠⊠⊠。

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

void do_true_thing() ;

int main()
{
// ⊠⊠⊠
// 名前 f ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
#if f()
    do_true_thing() ;
#else
    do_false_thing() ;
#endif
}
```

3.14.3 コンパイル時の条件分岐

⊠⊠⊠⊠⊠時⊠条件分岐⊠⊠、分岐⊠条件⊠⊠⊠⊠⊠⊠時計算⊠結果⊠使⊠、⊠⊠、選択⊠⊠⊠⊠分岐⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠含⊠⊠⊠⊠⊠⊠、使⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠条件分岐⊠⊠⊠⊠。

⊠⊠⊠⊠、`std::distance` ⊠⊠⊠標準⊠⊠⊠⊠⊠⊠実装⊠⊠⊠⊠⊠。`std::distance(first, last)` ⊠、⊠⊠⊠⊠⊠⊠ `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` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合⊠⊠⊠動⊠⊠⊠。入力⊠⊠⊠⊠⊠⊠⊠対応⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `last` ⊠等⊠⊠⊠⊠⊠⊠比較⊠⊠実装⊠必要⊠⊠⊠。

```
template < typename Iterator >
constexpr typename std::iterator_traits<Iterator>::difference_type
distance( Iterator first, Iterator last )
{
    typename std::iterator_traits<Iterator>::difference_type n = 0 ;

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

    return n ;
}
```

残念⊠⊠⊠、⊠⊠実装⊠ Iterator ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠渡⊠⊠⊠⊠⊠効率⊠悪⊠。

⊠⊠⊠必要⊠実装⊠、Iterator ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ last - first ⊠使⊠、⊠⊠⊠⊠⊠⊠⊠地道⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠遅⊠実装⊠使⊠⊠⊠⊠。Iterator ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、以下⊠⊠⊠⊠⊠使⊠⊠、is_random_access_iterator<iterator> ⊠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⊠⊠⊠⊠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 )
```

```
{
    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確認⊠⊠
    if ( is_random_access_iterator<Iterator> )
    {// ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠速⊠方法⊠使⊠
        return last - first ;
    }
    else
    { // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠遅⊠方法⊠使⊠
        typename std::iterator_traits<Iterator>::difference_type n = 0 ;

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

        return n ;
    }
}
```

残念⊠⊠⊠、⊠⊠⊠⊠⊠⊠動⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠渡⊠⊠、`last - first` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠、

```
if ( is_random_access_iterator<Iterator> )
```

⊠⊠⊠部分⊠⊠⊠⊠、`is_random_access_iterator<Iterator>` ⊠値⊠⊠⊠⊠⊠⊠時⊠計算⊠⊠⊠⊠⊠、最終的⊠⊠⊠⊠生成⊠結果⊠⊠⊠⊠、`if (true)` ⊠ `if (false)` ⊠⊠⊠⊠判断⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠選択⊠⊠⊠⊠分岐⊠⊠⊠⊠生成⊠行⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

`constexpr if` ⊠使⊠⊠、選択⊠⊠⊠⊠部分⊠分岐⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
// distance
template < typename Iterator >
constexpr typename std::iterator_traits<Iterator>::difference_type
distance( Iterator first, Iterator last )
{
    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確認⊠⊠
    if constexpr ( is_random_access_iterator<Iterator> )
    {// ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠速⊠方法⊠使⊠
```

```
        return last - first ;
    }
    else
    { // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠遅⊠方法⊠使⊠
        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 ⊠、実⊠⊠⊠⊠⊠⊠時条件分岐⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠実体化時⊠、選択⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実体化⊠抑制⊠行⊠機能⊠。

constexpr if ⊠⊠⊠⊠選択⊠⊠⊠⊠文⊠ discarded statement ⊠⊠⊠。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 ;
    f( x ) ; // return x.get()
    f( 0 ) ; // return x
}
```

f(x) ⊠⊠、return x ⊠ discarded statement ⊠⊠⊠⊠⊠実体化⊠⊠⊠⊠。X ⊠ int 型⊠暗黙⊠変換⊠⊠⊠⊠⊠問題⊠⊠⊠⊠⊠。f(0) ⊠⊠ return x.get() ⊠ discarded statement ⊠⊠⊠⊠⊠実体化⊠⊠⊠⊠。int 型⊠⊠⊠⊠⊠⊠関数 get ⊠⊠⊠⊠問題⊠⊠⊠⊠⊠。

discarded statement ⊠実体化⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠一部⊠。discarded statement ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠文法的、意味的⊠正⊠⊠⊠⊠場合⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
template < typename T >
void f( T x )
{
    // ⊠⊠⊠、名前 g ⊠宣言⊠⊠⊠⊠⊠⊠
    if constexpr ( false )
        g() ;

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

何度⊠説明⊠⊠⊠⊠⊠⊠⊠、constexpr if ⊠⊠⊠⊠⊠⊠⊠⊠実体化⊠条件付⊠⊠抑制⊠⊠⊠⊠⊠。条件付⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

⊠⊠⊠⊠⊠⊠常⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠、static_assert(false) ⊠⊠⊠⊠⊠⊠⊠⊠依存⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠宣言⊠解釈⊠⊠⊠⊠⊠、依存名⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠解釈⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、最初⊠⊠ static_assert ⊠⊠⊠⊠⊠⊠⊠式⊠書⊠⊠

⊠⊠。

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

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

⊠⊠、⊠⊠⊠⊠⊠ constexpr 文⊠条件⊠合⊠⊠⊠⊠⊠⊠ static_assert ⊠使⊠⊠⊠場合⊠⊠⊠。⊠⊠⊠、constexpr if ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠内容⊠全部 static_assert ⊠書⊠⊠⊠冗長⊠場合⊠。

```
template < typename T >
void f()
{
    if constexpr ( E1 )
        if constexpr ( E2 )
            if constexpr ( E3 )
            {
                // E1 && E2 && E3 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
                // 実際⊠⊠常⊠⊠⊠⊠⊠⊠⊠⊠⊠
                static_assert( false ) ;
            }
}
```

現実⊠⊠、E1, E2, E3 ⊠複雑⊠式⊠⊠⊠、static_assert(E1 && E2 && E3) ⊠書⊠⊠⊠冗長⊠。同⊠内容⊠二度書⊠⊠⊠間違⊠⊠元⊠。

⊠⊠⊠⊠⊠場合、static_assert ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠引数⊠依存⊠⊠⊠⊠⊠⊠⊠⊠、constexpr if ⊠条件⊠合⊠⊠⊠⊠⊠⊠発動⊠⊠ static_assert ⊠書⊠⊠。

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

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

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

⊠⊠⊠⊠⊠ `false_v` ⊠使⊠⊠⊠⊠、`static_assert` ⊠⊠⊠⊠⊠⊠⊠引数 `T` ⊠依存⊠⊠⊠。⊠⊠結果、`static_assert` ⊠発動⊠⊠⊠⊠⊠⊠⊠⊠実体化⊠⊠遅延⊠⊠⊠⊠⊠⊠⊠⊠⊠。

`constexpr if` ⊠非⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠書⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠場合⊠普通⊠ `if` 文⊠同⊠⊠。

3.14.5 constexpr if では解決できない問題

`constexpr if` ⊠条件付⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、条件付⊠⊠⊠⊠⊠⊠⊠実体化⊠抑制⊠⊠⊠、最初⊠問題⊠解決⊠⊠使⊠⊠⊠。⊠⊠⊠⊠以下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
// do_true_thing ⊠宣言
void do_true_thing() ;

// do_false_thing ⊠宣言⊠存在⊠⊠⊠

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

理由⊠、名前 `do_false_thing` ⊠非依存名⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠宣言時⊠解決⊠⊠⊠⊠⊠⊠。

3.14.6 constexpr if で解決できる問題

`constexpr if` ⊠依存名⊠関⊠⊠場合⊠、⊠⊠⊠⊠⊠⊠⊠実体化⊠⊠⊠⊠⊠⊠⊠場合⊠、実体化⊠抑制⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠、特定⊠型⊠対⊠⊠特別⊠操作⊠⊠⊠⊠場合

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

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

    int value{} ;

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

⊠⊠ `constexpr if` ⊠⊠⊠⊠⊠、`T` ⊠型⊠ `X` ⊠⊠⊠⊠⊠⊠⊠ `t.get_value()` ⊠⊠⊠式⊠実体化⊠⊠、⊠⊠⊠⊠⊠⊠。

再帰的⊠⊠⊠⊠⊠⊠⊠⊠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>() ; }
}
```

⊠⊠ `constexpr if` ⊠⊠⊠⊠⊠、`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 ⊠⊠、条件文⊠初期化文⊠記述⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

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

⊠⊠⊠、以下⊠⊠⊠⊠⊠同⊠意味⊠⊠⊠。

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

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

⊠⊠⊠⊠⊠⊠⊠機能⊠追加⊠⊠⊠⊠⊠⊠⊠⊠、変数⊠宣言⊠、`if` 文⊠条件⊠変数⊠使⊠、`if` 文⊠実行後⊠変数⊠使用⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠現実⊠⊠⊠⊠⊠頻出⊠⊠⊠⊠⊠。

```
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 ) ;
}
```

```
// ⊠⊠以降 file ⊠使⊠⊠⊠

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

上記⊠⊠⊠⊠⊠⊠⊠問題⊠⊠⊠。⊠⊠以降変数⊠使⊠⊠⊠⊠⊠、変数自体⊠使⊠⊠⊠⊠⊠。

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

// ⊠⊠使⊠⊠
int value = *ptr ;
```

変数⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠囲⊠⊠⊠⊠、変数⊠⊠⊠⊠⊠⊠⊠外⊠⊠⊠⊠⊠⊠⊠。

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

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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

C++17⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実引数⊠⊠⊠⊠⊠⊠⊠⊠実引数⊠推定⊠行⊠⊠⊠⊠⊠⊠⊠⊠。

```
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⊠⊠⊠⊠⊠⊠⊠実引数⊠⊠⊠⊠⊠⊠⊠⊠実引数⊠推定⊠行⊠⊠⊠⊠同⊠⊠。

```
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

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実引数⊠便利⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠一致⊠⊠⊠場合⊠⊠⊠。⊠⊠⊠⊠⊠場合⊠⊠⊠⊠⊠⊠⊠実引数推定⊠⊠⊠⊠⊠。

```
// ⊠⊠⊠⊠⊠風⊠⊠⊠⊠
template < typename T >
class Container
```

```
{
    std::vector<T> c ;
public :
    // 初期化⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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) ) ;
}
```

⊠⊠⊠⊠、C++17 ⊠⊠推定⊠⊠⊠⊠⊠⊠機能⊠提供⊠⊠⊠⊠⊠。

```
⊠⊠⊠⊠⊠⊠名( 引数⊠⊠⊠ ) -> ⊠⊠⊠⊠⊠⊠ id ;
```

⊠⊠⊠使⊠⊠、以下⊠⊠⊠⊠書⊠⊠。

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

C++ ⊠⊠⊠⊠⊠⊠⊠⊠⊠推定⊠⊠⊠⊠使⊠⊠、`Container<T>::Container( Iterator, Iterator)`⊠⊠⊠、T ⊠ `std::iterator_traits<Iterator>::value_type` ⊠⊠⊠推定⊠⊠⊠⊠⊠⊠⊠⊠判断⊠⊠⊠。

⊠⊠⊠⊠、初期化⊠⊠⊠⊠対応⊠⊠⊠⊠以下⊠⊠⊠⊠書⊠。

```
template < typename T >
class Container
{
    std::vector<T> c ;
public :
    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++ ⊠⊠⊠⊠⊠⊠⊠⊠⊠推定⊠⊠⊠⊠⊠、`Container<T>::Container( std::initializer_list<T> )`⊠場合⊠ `T` ⊠ `T` ⊠⊠⊠推定⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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 ⊠⊠非型⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠宣言⊠ `auto` ⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
template < auto x >
struct X { } ;

void f() { }

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

⊠⊠⊠ C++14 ⊠⊠⊠⊠⊠⊠、以下⊠⊠⊠⊠書⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

```
void f() { }

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

機能⊠⊠⊠⊠⊠⊠⊠ `__cpp_template_auto`, 値⊠ 201606。

3.18　using 属性名前空間

C++17 ⊠⊠、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⊠避⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠、⊠⊠ C++ ⊠⊠⊠⊠⊠⊠⊠⊠独自拡張⊠⊠⊠ `foo`, `bar` ⊠⊠⊠属性⊠⊠⊠⊠⊠⊠⊠、別⊠ C++ ⊠⊠⊠⊠⊠⊠⊠同⊠⊠独自拡張⊠⊠⊠ `foo`, `bar` ⊠⊠⊠属性⊠⊠⊠⊠⊠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⊠ 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 ⊠⊠、`using` 属性名前空間⊠⊠⊠機能⊠⊠⊠、`using` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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 ⊠⊠、非標準⊠属性⊠⊠⊠⊠⊠無視⊠⊠⊠。

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

属性⊠ C++ ⊠⊠⊠⊠⊠⊠⊠⊠⊠独自拡張⊠ C++ ⊠規格⊠準拠⊠⊠形⊠穏便⊠追加⊠⊠⊠⊠⊠機能⊠。⊠⊠属性⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠C ⊠⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠独自⊠文法⊠使⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠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" } ;
```

```
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⊠⊠⊠⊠⊠⊠⊠。

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

多値⊠受⊠取⊠⊠⊠、⊠⊠⊠⊠⊠多値⊠固⊠⊠⊠⊠⊠受⊠取⊠⊠、⊠⊠⊠⊠⊠⊠分解⊠⊠受⊠取⊠⊠⊠⊠⊠⊠⊠。

多値⊠固⊠⊠⊠受⊠取⊠⊠⊠以下⊠⊠⊠⊠書⊠。

```
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⊠⊠⊠⊠⊠⊠⊠受⊠取⊠⊠⊠以下⊠⊠⊠⊠書⊠。

```
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⊠使⊠⊠、以下⊠⊠⊠⊠書⊠⊠。

```
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⊠⊠⊠⊠⊠対応⊠⊠多値⊠型⊠⊠⊠。⊠⊠場合、a, b, c ⊠⊠⊠⊠⊠ `int`, `double`, `std::string` 型⊠⊠⊠。

`tuple` ⊠⊠⊠⊠⊠⊠、`pair` ⊠使⊠⊠。

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

    auto [a,b] = p ;

    // a ⊠ int 型、値⊠ 1
    // b ⊠ int 型、値⊠ 2
}
```

構造化束縛⊠ `if` 文⊠ `switch` 文、`for` 文⊠⊠使⊠⊠。

```
int main()
{
    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⊠⊠使⊠⊠。

```
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` ⊠要素型 `std::pair<const std::string, std::string>` ⊠構造化束縛 `[key, value]` ⊠受⊠⊠⊠⊠。

構造化束縛⊠配列⊠⊠使⊠⊠。

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

構造化束縛⊠⊠⊠⊠⊠⊠使⊠⊠。

```
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⊠⊠⊠⊠⊠使⊠場合⊠、非 `static` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠⊠⊠⊠⊠ `public` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

構造化束縛⊠ `constexpr` ⊠⊠⊠⊠⊠⊠。

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

    // ⊠⊠⊠
    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）⊠分類⊠⊠⊠文法⊠記述⊠⊠。構造化束縛宣言⊠⊠⊠宣言⊠、単純宣言（simple-declaration）⊠ `for-range` 宣言（for-range-declaration）⊠⊠⊠、[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⊠⊠⊠:
    ⊠⊠⊠⊠区切⊠⊠⊠識別子

初期化子:
    = 式
    { 式 }
    ( 式 )
```

以下⊠単純宣言⊠⊠⊠⊠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⊠使⊠例
    auto const & [p,q,r] = e1 ;
}
```

以下⊠ `for-range` 宣言⊠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⊠構造化束縛宣言⊠以下⊠⊠⊠⊠解釈⊠⊠⊠。

構造化束縛宣言⊠⊠⊠⊠宣言⊠⊠⊠変数⊠数⊠、初期化子⊠多値⊠数⊠一致⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
int main()
{
    // 2個⊠値⊠持⊠
    int expr[] = {1,2} ;

    // ⊠⊠⊠、変数⊠少⊠⊠⊠⊠
    auto[a] = expr ;
    // ⊠⊠⊠、変数⊠多⊠⊠⊠
    auto[b,c,d] = expr ;
}
```

構造化束縛宣言⊠宣言⊠⊠⊠⊠⊠⊠⊠⊠変数名⊠⊠⊠⊠、記述⊠⊠⊠⊠⊠⊠⊠属性、CV 修飾子、⊠⊠⊠⊠⊠⊠修飾子⊠変数⊠宣言⊠⊠⊠。

3.20.3　初期化子の型が配列の場合

初期化子⊠配列⊠場合、⊠⊠⊠⊠⊠変数⊠⊠⊠⊠⊠⊠配列⊠要素⊠初期化⊠⊠⊠。⊠⊠⊠⊠⊠⊠修飾子⊠⊠⊠場合、⊠⊠⊠⊠⊠変数⊠⊠⊠⊠初期化⊠⊠⊠。

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

⊠⊠⊠⊠⊠⊠修飾子⊠⊠⊠場合、変数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
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⊠場合、配列⊠要素⊠⊠⊠⊠⊠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⊠初期化子⊠型 E ⊠配列⊠⊠⊠⊠場合⊠、std::tuple_size<E> ⊠完全形⊠名前⊠⊠⊠場合、構造化束縛宣言⊠初期化子⊠型⊠ E, ⊠⊠値⊠ e ⊠⊠⊠。構造化束縛宣言⊠宣言⊠⊠⊠ 1 ⊠目⊠変数⊠ 0, 2 ⊠目⊠変数⊠ 1, . . . , ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ i ⊠⊠⊠。

std::tuple_size<E>::value ⊠整数⊠⊠⊠⊠⊠⊠時定数式⊠、⊠⊠値⊠初期化子⊠値⊠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 ⊠ 3
}
```

⊠⊠⊠⊠⊠値⊠取得⊠⊠⊠⊠⊠、非修飾名 get ⊠型 E ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠探⊠⊠⊠。get ⊠見⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠変数⊠初期化子⊠ e.get<i>() ⊠⊠⊠。

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

⊠⊠⊠構造化束縛宣言⊠、以下⊠意味⊠⊠⊠。

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

⊠⊠⊠⊠⊠ get ⊠宣言⊠見⊠⊠⊠⊠⊠場合、初期化子⊠ get<i>(e) ⊠⊠⊠。⊠⊠場合、get ⊠連想名前空間⊠⊠探⊠⊠⊠。通常⊠非修飾名前検索⊠行⊠⊠⊠⊠。

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

構造化束縛宣言⊠宣言⊠⊠⊠変数⊠型⊠以下⊠⊠⊠⊠決定⊠⊠⊠。

変数⊠型 type ⊠ “std::tuple_element<i, E>::type” ⊠⊠⊠。

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

```
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⊠場合、構造化束縛宣言⊠初期化子⊠型 E ⊠⊠⊠⊠型⊠、⊠⊠⊠⊠非 static ⊠⊠⊠⊠⊠⊠⊠⊠ public ⊠直接⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠単一⊠曖昧⊠⊠⊠⊠ public 基本⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠必要⊠⊠⊠。E ⊠匿名 union ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

以下⊠型 E ⊠⊠⊠適切⊠⊠⊠⊠⊠例⊠⊠⊠。

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

struct B : A { } ;
```

以下⊠型 E ⊠⊠⊠不適切⊠⊠⊠⊠⊠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
```

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

型 E ⊠非 static ⊠⊠⊠⊠⊠⊠⊠⊠宣言⊠⊠⊠順番⊠多値⊠⊠⊠認識⊠⊠⊠。以下⊠⊠⊠⊠⊠、

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

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

以下⊠⊠⊠⊠⊠意味的⊠等⊠⊠。

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

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

構造化束縛⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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 ;
    int value = a < b ? a : b ;
}
```

⊠⊠⊠⊠⊠関数⊠中身⊠展開⊠⊠⊠⊠⊠、関数⊠⊠⊠⊠⊠⊠展開⊠⊠⊠。

人間⊠関数⊠⊠⊠⊠⊠⊠展開⊠手⊠行⊠⊠⊠面倒⊠。⊠⊠⊠⊠⊠⊠⊠読⊠⊠⊠⊠。“`min(a,b)`” ⊠ “`a<b?a:b`” ⊠⊠⊠⊠⊠読⊠⊠⊠⊠⊠⊠⊠⊠。

幸⊠、C++ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠展開⊠自動的⊠行⊠⊠⊠⊠人間⊠苦労⊠⊠必要⊠⊠⊠。

⊠⊠⊠⊠⊠展開⊠万能⊠最適化⊠⊠⊠⊠。⊠⊠⊠⊠⊠展開⊠⊠⊠⊠逆⊠遅⊠⊠⊠場合⊠⊠⊠。

⊠⊠⊠⊠、⊠⊠関数⊠⊠⊠⊠⊠⊠⊠⊠結果⊠⊠⊠⊠⊠⊠⊠⊠ 1K ⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠関数⊠呼⊠⊠⊠⊠箇所⊠⊠⊠⊠⊠⊠中⊠ 1000 件⊠⊠場合、⊠⊠⊠⊠⊠全体⊠⊠⊠⊠⊠ 1M ⊠⊠⊠増⊠⊠。⊠⊠⊠⊠⊠⊠⊠増⊠⊠⊠⊠⊠⊠⊠⊠、CPU ⊠⊠⊠⊠⊠⊠⊠圧迫⊠⊠。

⊠⊠⊠⊠、⊠⊠関数⊠実行時間⊠関数呼⊠出⊠⊠実行時間⊠比⊠⊠桁違⊠⊠長⊠⊠⊠、関数呼⊠出⊠⊠⊠⊠⊠⊠削減⊠⊠⊠⊠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⊠⊠⊠⊠⊠⊠⊠技術⊠未熟⊠⊠⊠時代⊠ C++ ⊠⊠⊠⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠展開⊠⊠⊠⊠⊠⊠⊠⊠⊠判断⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠ `inline` ⊠⊠⊠⊠⊠⊠追加⊠⊠⊠。⊠⊠⊠⊠⊠展開⊠⊠⊠⊠⊠関数⊠ `inline` 関数⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠展開⊠⊠⊠⊠関数⊠⊠認識⊠⊠。

3.21.2　現代の inline の意味

現代⊠⊠、⊠⊠⊠⊠⊠⊠技術⊠発展⊠⊠⊠ C++ ⊠⊠⊠⊠⊠⊠⊠十分⊠賢⊠⊠⊠⊠⊠⊠、関数⊠⊠⊠⊠⊠⊠展開⊠⊠⊠目的⊠ `inline` ⊠⊠⊠⊠⊠⊠使⊠必要⊠⊠⊠。実際、現代⊠ C++ ⊠⊠⊠⊠⊠⊠⊠⊠ `inline` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠展開⊠強制⊠⊠⊠。関数⊠⊠⊠⊠⊠⊠展開⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠判断⊠⊠⊠。

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

C++ ⊠⊠、定義⊠⊠⊠⊠⊠⊠中⊠ 1 ⊠⊠⊠書⊠⊠⊠⊠⊠⊠⊠⊠。

```
void f() ; // OK、宣言
void f() ; // OK、再宣言
void f() { } // OK、定義

void f() { } // ⊠⊠⊠、再定義
```

通常⊠、関数⊠使⊠場合⊠⊠宣言⊠⊠⊠書⊠⊠使⊠。定義⊠⊠⊠⊠ 1 ⊠⊠翻訳単位⊠書⊠⊠⊠⊠⊠⊠⊠。

```
// f.h

void f() ;

// f.cpp

void f() { }

// main.cpp

#include "f.h"

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

⊠⊠⊠、関数⊠⊠⊠⊠⊠⊠展開⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠実装上⊠都合⊠、関数⊠定義⊠同⊠翻訳単位⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
inline void f() ;

int main()
{
    // ⊠⊠⊠、定義⊠⊠⊠
    f() ;
}
```

⊠⊠⊠、翻訳単位⊠⊠⊠定義⊠⊠⊠、定義⊠重複⊠⊠ ODR ⊠違反⊠⊠。

C++ ⊠⊠⊠⊠問題⊠解決⊠⊠⊠⊠⊠、`inline` 関数⊠定義⊠同一⊠⊠⊠⊠、複数⊠翻訳単位⊠定義⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠ ODR ⊠違反⊠⊠⊠。

```
// a.cpp

inline void f() { }
```

```
void a()
{
    f() ;
}

// b.cpp

// OK、inline 関数
inline void f() { }

void b()
{
    f() ;
}
```

⊠⊠⊠例⊠⊠⊠⊠同一⊠ inline 関数⊠直接記述⊠⊠⊠⊠⊠、inline 関数⊠定義⊠同一性⊠保証⊠⊠⊠⊠⊠、通常⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠書⊠⊠ #include ⊠⊠使⊠。

3.21.3　inline 変数の意味

inline 変数⊠、ODR ⊠違反⊠⊠変数⊠定義⊠重複⊠認⊠⊠。同⊠名前⊠ inline 変数⊠同⊠変数⊠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⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
int x, y ;

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

⊠⊠⊠、C++14 ⊠

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

⊠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 ⊠⊠、⊠⊠⊠⊠表現⊠⊠型⊠入⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠言語⊠特別⊠型⊠⊠⊠扱⊠⊠⊠⊠⊠。

⊠⊠⊠⊠⊠ 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⊠⊠⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠数⊠ `<climits>` ⊠定義⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、`CHAR_BIT` ⊠知⊠⊠⊠⊠⊠⊠⊠。

C++17 ⊠⊠、1 ⊠⊠⊠⊠ UTF-8 ⊠ 8 ⊠⊠⊠⊠ 1 ⊠⊠⊠単位⊠⊠⊠⊠表現⊠⊠⊠⊠規定⊠⊠⊠⊠。

`std::byte` 型⊠、生⊠⊠⊠⊠列⊠表⊠⊠⊠⊠型⊠⊠⊠使⊠⊠⊠⊠⊠⊠⊠。生⊠ 1 ⊠⊠⊠⊠表⊠⊠⊠ `unsigned char` 型⊠慣習的⊠使⊠⊠⊠⊠⊠⊠、`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⊠、`<cstddef>` ⊠以下⊠⊠⊠⊠定義⊠⊠⊠⊠⊠。

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

`std::byte` ⊠⊠⊠⊠⊠⊠⊠⊠⊠ `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()
{
    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⊠他⊠数値型⊠⊠⊠暗黙⊠型変換⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠型⊠取⊠違⊠⊠⊠⊠⊠型⊠他⊠型⊠演算⊠⊠⊠⊠⊠⊠⊠⊠防⊠⊠⊠⊠⊠⊠⊠。

```
int main()
{
    // ⊠⊠⊠、() ⊠⊠⊠初期化⊠ int 型⊠⊠⊠暗黙⊠変換⊠入⊠
    std::byte b1(1) ;

    // ⊠⊠⊠、=⊠⊠⊠初期化⊠ int 型⊠⊠⊠暗黙⊠変換⊠入⊠
    std::byte b2 = 1 ;

    std::byte b{} ;

    // ⊠⊠⊠、operator =⊠⊠⊠ int 型⊠代入⊠暗黙⊠変換⊠入⊠
    b = 1 ;
    // ⊠⊠⊠、operator =⊠⊠⊠ double 型⊠代入⊠暗黙⊠変換⊠入⊠
    b = 1.0 ;
}
```

std::byte 型⊠ {} ⊠⊠⊠⊠初期化⊠⊠⊠、縮小変換⊠禁止⊠⊠⊠⊠⊠⊠⊠⊠、std::byte 型⊠表現⊠⊠⊠値⊠範囲⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠、今 std::byte ⊠ 8 ⊠⊠⊠⊠、最小値⊠ 0、最大値⊠ 255 ⊠環境⊠⊠⊠⊠。

```
int main()
{
    // ⊠⊠⊠、表現⊠⊠⊠値⊠範囲⊠⊠⊠⊠
    std::byte b1{-1} ;
    // ⊠⊠⊠、表現⊠⊠⊠値⊠範囲⊠⊠⊠⊠
    std::byte b2{256} ;
}
```

std::byte ⊠内部⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠単位⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、規格上 char ⊠同⊠⊠⊠⊠配慮⊠行⊠⊠⊠⊠⊠。

```
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⊠⊠⊠⊠使⊠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_intenger<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 ⊠⊠型安全⊠値⊠格納⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、variant, any, optional ⊠追加⊠⊠⊠。

4.1 variant : 型安全な union

4.1.1 使い方

⊠⊠⊠⊠⊠⊠⊠⊠ <variant> ⊠定義⊠⊠⊠⊠⊠ variant ⊠、型安全⊠ union ⊠⊠⊠使⊠⊠⊠⊠⊠⊠⊠。

```
#include <variant>

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

    // int, double, std::string ⊠⊠⊠⊠⊠⊠格納⊠⊠ variant
    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠最初⊠型⊠⊠⊠⊠⊠⊠構築
    std::variant< int, double, std::string > x ;

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

    // int ⊠入⊠⊠⊠⊠⊠確認
    // false ⊠返⊠
    bool has_int = std::holds_alternative<int>( x ) ;
    // std::string ⊠入⊠⊠⊠⊠⊠確認
    // true ⊠返⊠
    bool has_string = std::holds_alternative<std::string> ( x ) ;

    // 入⊠⊠⊠⊠値⊠得⊠
    // "hello"
    std::string str = std::get<std::string>(x) ;
}
```

4.1.2　型非安全な古典的 union

C++ ⊠従来⊠⊠持⊠⊠⊠⊠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)` ⊠

$$\begin{aligned}\texttt{sizeof(U)} = {} & \max\{\texttt{sizeof(int)}, \texttt{sizeof(double)}, \texttt{sizeof(std::string)}\} \\ & + \text{⊠⊠⊠⊠⊠⊠⊠}\end{aligned}$$

⊠⊠⊠。`sizeof(S)` ⊠、

$$\begin{aligned}\texttt{sizeof(S)} = {} & \texttt{sizeof(int)} + \texttt{sizeof(double)} + \texttt{sizeof(std::string)} \\ & + \text{⊠⊠⊠⊠⊠⊠⊠}\end{aligned}$$

⊠⊠⊠。

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⊠⊠⊠⊠⊠⊠。

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

    // ⊠⊠⊠⊠⊠⊠⊠⊠
    // int 型⊠⊠⊠⊠⊠⊠初期化⊠⊠
    U() : i{} { }
    // ⊠⊠⊠⊠⊠⊠⊠
    // 何⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠破棄⊠⊠⊠⊠⊠⊠責任⊠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
```

```
        // 破棄⊠必要
        destruct( u.s ) ;
    }
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠書⊠⊠⊠⊠⊠。variant ⊠使⊠⊠、⊠⊠⊠⊠⊠面倒⊠冗長⊠⊠⊠⊠⊠書⊠⊠⊠、型安全⊠ union ⊠同等機能⊠実現⊠⊠⊠。

4.1.3　variant の宣言

variant ⊠⊠⊠⊠⊠⊠⊠実引数⊠保持⊠⊠⊠型⊠与⊠⊠。

```
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

⊠⊠⊠⊠⊠初期化

variant ⊠⊠⊠⊠⊠⊠構築⊠⊠⊠、最初⊠与⊠⊠型⊠値⊠⊠⊠⊠⊠⊠構築⊠⊠保持⊠⊠。

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

variant ⊠⊠⊠⊠⊠⊠構築⊠⊠⊠⊠型⊠最初⊠与⊠⊠⊠、variant ⊠⊠⊠⊠⊠⊠構築⊠⊠⊠⊠。

```
// ⊠⊠⊠⊠⊠構築⊠⊠⊠⊠型
struct non_default_constructible
{
    non_default_constructible() = delete ;
} ;

// ⊠⊠⊠
// ⊠⊠⊠⊠⊠構築⊠⊠⊠⊠
std::variant< non_default_constructible > v ;
```

⊠⊠⊠⊠⊠構築⊠⊠⊠⊠型⊠⊠⊠保持⊠⊠ variant ⊠⊠⊠⊠⊠⊠構築⊠⊠⊠⊠⊠⊠、最初⊠型⊠⊠⊠⊠⊠⊠構築可能⊠型⊠⊠⊠⊠⊠⊠。

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

struct Empty { } ;


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

⊠⊠⊠⊠⊠場合⊠、Empty ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠独自⊠定義⊠⊠⊠⊠面倒⊠⊠⊠、標準⊠⊠⊠⊠⊠⊠⊠ std::monostate ⊠⊠⊠⊠以下⊠⊠⊠⊠定義⊠⊠⊠⊠⊠。

```
namespace std {
    struct monostate { } ;
}
```

⊠⊠⊠⊠⊠、上⊠例⊠以下⊠⊠⊠⊠書⊠⊠。

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

std::monostate ⊠ variant ⊠最初⊠⊠⊠⊠⊠⊠⊠実引数⊠⊠⊠使⊠⊠⊠⊠ variant ⊠⊠⊠⊠⊠⊠構築可能⊠⊠⊠⊠⊠⊠型⊠。⊠⊠以上⊠意味⊠⊠⊠。

⊠⊠⊠初期化

variant ⊠同⊠型⊠ variant ⊠渡⊠⊠、⊠⊠⊠/⊠⊠⊠⊠⊠。

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

variant ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠値⊠渡⊠⊠場合

variant ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠上記以外⊠値⊠渡⊠⊠場合、variant ⊠⊠⊠⊠⊠⊠⊠実引数⊠指定⊠⊠型⊠中⊠⊠、⊠⊠⊠⊠⊠⊠⊠解決⊠⊠⊠最適⊠型⊠選⊠⊠、⊠⊠型⊠値⊠変換⊠⊠、値⊠保持⊠⊠。

```
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⊠優先的⊠変換⊠⊠⊠
    val d( 'a' ) ;
}
```

in_place_type ⊠⊠⊠ emplace 構築

variant ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠第一引数⊠ std::in_place_type<T> ⊠渡⊠⊠⊠⊠⊠⊠、T 型⊠要素⊠構築⊠⊠⊠⊠⊠ T 型⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠渡⊠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⊠⊠⊠⊠⊠動⊠⊠⊠。

```
struct X
```

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

int main()
{
    // X ⊠構築
    X x( 1, 2, 3 ) ;
    // ⊠⊠⊠、X ⊠⊠⊠⊠⊠⊠⊠⊠
    std::variant<X> v( x ) ;
}
```

⊠⊠⊠⊠⊠場合、variant ⊠内部⊠ X ⊠構築⊠⊠際⊠、構築⊠必要⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実引数⊠渡⊠⊠、variant ⊠ X ⊠構築⊠⊠⊠必要⊠⊠⊠。⊠⊠⊠⊠⊠ std::in_place_type<T> ⊠使⊠⊠。T ⊠構築⊠⊠⊠型⊠指定⊠⊠第一引数⊠⊠、第二引数以降⊠ T ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠渡⊠値⊠⊠⊠。

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

int main()
{
    // X ⊠値⊠構築⊠⊠保持
    std::variant<X> v( std::in_place_type<X>, 1, 2, 3 ) ;
}
```

4.1.5　variant の破棄

variant ⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠保持⊠⊠⊠⊠値⊠適切⊠破棄⊠⊠⊠⊠⊠。

```
int main()
{
    std::vector<int> v ;
    std::list<int> l ;
    std::deque<int> d ;
    std::variant<
        std::vector<int>,
        std::list<int>,
        std::deque<int>
    > val ;

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

    // variant ⊠⊠⊠⊠⊠⊠⊠⊠⊠ deque<int>⊠破棄⊠⊠
}
```

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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠。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⊠⊠⊠⊠場合、false ⊠返⊠。

```
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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠
    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⊠。

```
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⊠場合⊠、std::variant_npos ⊠返⊠。

```
// variant ⊠値⊠持⊠⊠⊠⊠⊠⊠⊠⊠確認⊠⊠関数
template < typename ... Types  >
void has_value( std::variant< Types ... > && v )
{
    return v.index() != std::variant_npos ;

    // ⊠⊠⊠⊠⊠⊠
    // 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⊠渡⊠⊠、variant ⊠保持⊠⊠⊠型⊠数⊠返⊠⊠⊠⊠⊠。

```
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 t2_size = std::variant_size<t3>::size ;
```

variant_size ⊠ integral_constant ⊠基本⊠⊠⊠⊠持⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠構築⊠⊠結果⊠⊠⊠⊠⊠定義変換⊠⊠⊠⊠⊠⊠値⊠取⊠出⊠⊠。

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

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

variant_size ⊠以下⊠⊠⊠⊠⊠⊠⊠⊠⊠変数⊠⊠⊠⊠⊠⊠⊠用意⊠⊠⊠⊠⊠。

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

⊠⊠⊠使⊠⊠、以下⊠⊠⊠⊠⊠書⊠⊠。

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

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

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

`std::variant_alternative<I, T>` ⊠ T 型⊠ `variant` ⊠保持⊠⊠⊠型⊠⊠⊠、`I` 番目⊠型⊠⊠⊠⊠⊠⊠⊠型名 `type` ⊠返⊠。

```
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` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠定義⊠⊠⊠⊠⊠。

```
template <size_t I, class T>
    using variant_alternative_t
        = typename variant_alternative<I, T>::type ;
```

⊠⊠⊠使⊠⊠、以下⊠⊠⊠⊠⊠書⊠⊠。

```
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)` ⊠、`variant v` ⊠ `T` 型⊠値⊠保持⊠⊠⊠⊠⊠⊠⊠⊠⊠確認⊠⊠。保持⊠⊠⊠⊠⊠⊠⊠⊠⊠ `true` ⊠、⊠⊠⊠⊠⊠⊠⊠ `false` ⊠返⊠。

```
int main()
{
    // int 型⊠値⊠構築
    std::variant< int, double > v ;
    // true
    bool has_int = std::holds_alternative<int>(v) ;
    // false
    bool has_double = std::holds_alternative<double>(v) ;
}
```

型 T ⊠実引数⊠与⊠⊠⊠⊠ variant ⊠保持⊠⊠⊠型⊠⊠⊠⊠⊠⊠⊠⊠⊠。以下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

    // ⊠⊠⊠
    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) ⊠、variant v ⊠型⊠⊠⊠⊠⊠⊠⊠⊠⊠ I 番目⊠型⊠値⊠返⊠。⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠⊠。

```
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) ;
}
```

I ☒☒☒☒☒☒☒☒範囲☒超☒☒☒☒☒☒☒☒☒☒☒。

```
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) ) ;
}
```

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⊠。

4.1.15　get_if：値を保持している場合に取得

get_if<I>(vp) ⊠ get_if<T>(vp) ⊠、variant ⊠⊠⊠⊠⊠⊠⊠ vp ⊠実引数⊠取⊠、*vp ⊠⊠⊠⊠⊠⊠⊠ I, ⊠⊠⊠⊠型 T ⊠値⊠保持⊠⊠⊠⊠場合、⊠⊠値⊠⊠⊠⊠⊠⊠⊠⊠返⊠。

```
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<1>( &v ) ;

    v = "hello" ;
    // std::string
    auto c = std::get_if<2>( &v ) ;

}
```

⊠⊠、vp ⊠ nullptr ⊠場合、⊠⊠⊠⊠ *vp ⊠指定⊠⊠⊠値⊠保持⊠⊠⊠⊠⊠場合⊠、nullptr ⊠返⊠。

```
int main()
{
    // int 型⊠値⊠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⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠比較⊠⊠⊠。variant 同士⊠比較⊠、一般⊠⊠⊠⊠⊠⊠⊠⊠自然⊠⊠思⊠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⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠、variant v, w ⊠対⊠⊠、式 `get<i>(v) == get<i>(w)` ⊠⊠⊠⊠⊠ i ⊠対⊠⊠妥当⊠⊠⊠⊠⊠⊠⊠⊠⊠。

variant v, w ⊠同一性⊠比較⊠、`v == w` ⊠場合、以下⊠⊠⊠⊠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 ⊠⊠ variant ⊠別⊠型⊠保持⊠⊠⊠⊠場合⊠等⊠⊠⊠⊠。⊠⊠⊠値⊠⊠⊠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⊠値⊠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() )
```

```
        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⊠⊠考⊠⊠⊠⊠。

大小比較

variant ⊠大小⊠比較⊠⊠⊠⊠⊠、variant ⊠⊠⊠⊠⊠⊠⊠実引数⊠与⊠⊠型⊠自分自身⊠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() ⊠⊠⊠、false
2. ⊠⊠以外⊠場合、v.valueless_by_exception() ⊠⊠⊠、true
3. ⊠⊠以外⊠場合、v.index() < w.index() ⊠⊠⊠、true
4. ⊠⊠以外⊠場合、v.index() > w.index() ⊠⊠⊠、false
5. ⊠⊠以外⊠場合、get<i>(v) < get<i>(w)。⊠⊠⊠ i ⊠ v.index()

値⊠⊠⊠ variant ⊠最⊠小⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠小⊠⊠⊠⊠⊠小⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠同⊠型⊠値⊠⊠⊠⊠⊠⊠⊠⊠、値同士⊠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
    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠比較
    b < a ;
}
```

operator < ⊠以下⊠⊠⊠⊠実装⊠⊠⊠。

```
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⊠⊠大小比較⊠同⊠方法⊠比較⊠⊠⊠。

4.1.17　visit：variant が保持している値を受け取る

std::visit ⊠、variant ⊠保持⊠⊠⊠⊠型⊠実引数⊠関数⊠⊠⊠⊠⊠⊠⊠呼⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
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 ) ⊠呼⊠⊠⊠
    std::visit( visitor, v, w ) ;
    // visitor( 3.14, 42 ) ⊠呼⊠⊠⊠
    std::visit( visitor, w, v ) ;
}
```

⊠⊠⊠⊠⊠、variant ⊠⊠⊠型⊠値⊠保持⊠⊠⊠⊠⊠⊠扱⊠⊠⊠⊠⊠⊠⊠。

std::visit ⊠以下⊠⊠⊠⊠宣言⊠⊠⊠⊠⊠。

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

第一引数⊠関数⊠⊠⊠⊠⊠⊠⊠渡⊠、第二引数以降⊠ variant ⊠渡⊠。⊠⊠⊠、`vis( get<i>(vars)... )` ⊠⊠⊠⊠呼⊠⊠⊠。

```
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> ⊠定義⊠⊠⊠⊠⊠ std::any ⊠、⊠⊠⊠⊠⊠⊠⊠型⊠値⊠⊠保持⊠⊠⊠⊠⊠⊠⊠。

```
#include <any>

int main()
{
    std::any a ;

    a = 0 ; // int
    a = 1.0 ; // double
    a = "hello" ; // char const *
    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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ emplace ⊠⊠⊠⊠⊠⊠ in_place_type ⊠使⊠⊠。

```
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 ) ;
}
```

4.2.4　any への代入

any ⊠⊠代入⊠普通⊠⊠⊠⊠⊠⊠⊠⊠期待⊠⊠⊠⊠動⊠⊠⊠⊠。

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

    // a ⊠ int 型⊠値 42 ⊠保持⊠⊠
    a = 42 ;
    // b ⊠ int 型⊠値 42 ⊠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 ⊠ emplace ⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

int main()
{
    std::any a ;

    // 型 X ⊠ X(1, 2) ⊠構築⊠⊠結果⊠値⊠保持⊠⊠
    a.emplace<X>( 1, 2 ) ;
}
```

reset : 値⊠破棄

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

any ⊠ reset ⊠⊠⊠⊠関数⊠、any ⊠保持⊠⊠⊠⊠値⊠破棄⊠⊠。reset ⊠呼⊠出⊠⊠後⊠ any ⊠値⊠保持⊠⊠⊠。

```
int main()
{
    // a ⊠値⊠保持⊠⊠⊠
    std::any a ;
    // a ⊠ int 型⊠値⊠保持⊠⊠
    a = 0 ;

    // a ⊠値⊠保持⊠⊠⊠
    a.reset() ;
}
```

swap：⊠⊠⊠⊠

any ⊠ swap ⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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

    // a ⊠ int 型⊠値⊠保持
    // b ⊠ double 型⊠値⊠保持

    a.swap(b) ;

    // a ⊠ double 型⊠値⊠保持
    // b ⊠ int 型⊠値⊠保持
}
```

has_value：値⊠保持⊠⊠⊠⊠⊠⊠⊠⊠調⊠⊠

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

any ⊠ has_value ⊠⊠⊠⊠関数⊠ any ⊠値⊠保持⊠⊠⊠⊠⊠⊠⊠⊠⊠調⊠⊠。値⊠保持⊠⊠⊠⊠⊠⊠⊠ true ⊠、保持⊠⊠⊠⊠⊠⊠⊠⊠ false ⊠返⊠。

```
int main()
{
    std::any a ;
    // 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 ⊠ typeid(T) ⊠返⊠。値⊠保持⊠⊠⊠⊠⊠場合、typeid(void) ⊠返⊠。

```
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>
```

```
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...) ⊠ T 型⊠⊠⊠⊠⊠⊠⊠⊠⊠実引数 args... ⊠構築⊠⊠値⊠保持⊠⊠ any ⊠返⊠。

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

int main()
{
    // int 型⊠値⊠保持⊠⊠ any
    auto a = std::make_any<int>( 0 ) ;
    // double 型⊠値⊠保持⊠⊠ any
    auto b = std::make_any<double>( 0.0 ) ;

    // X 型⊠値⊠保持⊠⊠ any
    auto c = std::make_any<X>( 1, 2 ) ;
}
```

any_cast：保持⊠⊠⊠⊠値⊠取⊠出⊠

```
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) ⊠ operand ⊠保持⊠⊠⊠⊠値⊠返⊠。

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

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

any_cast<T> ⊠指定⊠⊠ T 型⊠、any ⊠保持⊠⊠⊠⊠型⊠⊠⊠⊠場合、std::bad_any_cast ⊠ throw ⊠⊠⊠。

```
int main()
```

```
{
    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 ⊠⊠⊠⊠⊠⊠⊠⊠渡⊠⊠、T ⊠⊠⊠⊠⊠⊠⊠型⊠返⊠⊠⊠。any ⊠ T 型⊠保持⊠⊠⊠⊠場合⊠ T 型⊠参照⊠⊠⊠⊠⊠⊠⊠⊠返⊠。保持⊠⊠⊠⊠⊠場合⊠、nullptr ⊠返⊠。

```
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⊠⊠。

```
int divide( int a, int b )
{
    if ( b == 0 )
    {
        // ⊠⊠⊠処理
    }
    else
        return a / b ;
}
```

⊠⊠⊠除算⊠⊠⊠⊠⊠⊠⊠、b⊠値⊠0⊠場合、⊠⊠関数⊠値⊠用意⊠⊠⊠⊠⊠⊠⊠⊠⊠。問題⊠、int型⊠⊠⊠⊠⊠値⊠通常⊠除算結果⊠⊠⊠使⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠示⊠特別⊠値⊠返⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠⊠場合⊠⊠⊠⊠⊠値⊠通知⊠⊠方法⊠⊠⊠、過去⊠⊠⊠⊠⊠⊠方法⊠考案⊠⊠⊠。⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実引数⊠⊠⊠受⊠取⊠方法、⊠⊠⊠⊠⊠変数⊠使⊠方法、例外⊠。

`optional`⊠⊠⊠⊠⊠⊠値⊠用意⊠⊠⊠⊠場合⊠使⊠⊠共通⊠方法⊠提供⊠⊠。

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

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

    // ⊠⊠除算
    auto fail = divide( 10, 0 ) ;

    // false、値⊠保持⊠⊠⊠⊠⊠
    bool has_value = fail.has_value() ;

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

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⊠⊠、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 ) ;
    // int ⊠⊠ double ⊠⊠型変換⊠行⊠⊠⊠
    // int 型⊠値 1 ⊠保持⊠⊠
    std::optional<int> c ( 1.0 ) ;
}
```

T 型⊠⊠ U 型⊠型変換⊠⊠⊠⊠⊠、optional<T> ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ optional<U> ⊠渡⊠⊠ U ⊠⊠ T ⊠型変換⊠⊠⊠ T 型⊠値⊠保持⊠⊠ optional ⊠⊠⊠。

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

    // long 型⊠値 42 ⊠保持⊠⊠
    std::optional<long> b ( a ) ;
}
```

optional ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠第一引数⊠ std::in_place_type<T> ⊠渡⊠⊠、後続⊠引数⊠使⊠⊠ T 型⊠⊠⊠⊠⊠⊠⊠⊠ emplace 構築⊠⊠⊠。

```
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⊠⊠⊠⊠挙動⊠⊠⊠。std::nullopt ⊠代入⊠⊠⊠値⊠保持⊠⊠⊠ optional ⊠⊠⊠。

4.3.5　optional の破棄

optional ⊠破棄⊠⊠⊠⊠⊠、保持⊠⊠⊠⊠値⊠⊠⊠⊠、適切⊠破棄⊠⊠⊠。

```
struct X
```

```
{
    ~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 ⊠ 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 ⊠値⊠保持⊠⊠⊠⊠場合、true ⊠返⊠。

```
int main()
{
    std::optional<int> a ;
    // false
    bool b1 = a.has_value() ;
    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確認する

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

optional ⊠文脈上 bool ⊠変換⊠⊠⊠、値⊠保持⊠⊠⊠⊠場合⊠⊠⊠ true ⊠⊠⊠評価⊠⊠⊠。

```
int main()
{
    std::optional<bool> a = some_function();
    // OK、文脈上 bool ⊠変換
    if ( a )
    {
        // 値⊠保持
    }
    else
    {
        // 値⊠不保持
    }

    // ⊠⊠⊠、暗黙⊠型変換⊠行⊠⊠⊠⊠
    bool b1 = a ;
    // OK、明示的⊠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 ⊠⊠⊠⊠関数⊠ optional ⊠値⊠保持⊠⊠⊠⊠場合、値⊠⊠⊠⊠⊠⊠⊠⊠⊠

返⊠。値⊠保持⊠⊠⊠⊠⊠場合、`std::bad_optional_access` ⊠ `throw` ⊠⊠⊠。

```
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⊠⊠⊠⊠⊠⊠⊠
    }
}
```

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)` ⊠⊠⊠⊠関数⊠、`optional` ⊠値⊠保持⊠⊠⊠⊠場合⊠⊠⊠値⊠、保持⊠⊠⊠⊠⊠場合⊠ `v` ⊠返⊠。

```
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) ;
}
```

4.3.11 reset：保持している値を破棄する

reset ⊠⊠⊠⊠関数⊠呼⊠出⊠⊠、保持⊠⊠⊠⊠値⊠⊠⊠場合破棄⊠⊠。reset ⊠⊠⊠⊠関数⊠呼⊠出⊠⊠後⊠ optional ⊠値⊠保持⊠⊠⊠状態⊠⊠⊠。

```
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> ⊠比較⊠⊠⊠⊠⊠⊠、T 型⊠⊠⊠⊠⊠⊠⊠同士⊠比較⊠⊠⊠必要⊠⊠⊠。

同一性⊠比較

値⊠保持⊠⊠⊠ 2 ⊠⊠ optional ⊠等⊠⊠。片方⊠⊠⊠値⊠保持⊠⊠⊠⊠ optional ⊠等⊠⊠⊠⊠。両方⊠⊠値⊠保持⊠⊠⊠⊠ optional ⊠値⊠⊠⊠比較⊠⊠⊠。

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

    // true
    // ⊠⊠⊠⊠値⊠保持⊠⊠⊠ optional
    bool b1 = a == b ;

    a = 0 ;

    // false
    // a ⊠⊠値⊠保持
    bool b2 = a == b ;
    b = 1 ;

    // false
    // ⊠⊠⊠⊠値⊠保持。値⊠⊠⊠比較
    bool b3 = a == b ;
}
```

大小比較

optional 同士⊠大小比較⊠、`a < b` ⊠場合

1. `b` ⊠値⊠保持⊠⊠⊠⊠⊠⊠⊠ `false`
2. ⊠⊠以外⊠場合⊠、`a` ⊠値⊠保持⊠⊠⊠⊠⊠⊠⊠ `true`
3. ⊠⊠以外⊠場合、`a` ⊠ `b` ⊠保持⊠⊠⊠⊠値同士⊠比較

⊠⊠⊠。

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

    // false
    // b ⊠値⊠⊠
    bool b1 = a < b ;

    b = 0 ;

    // true
    // b ⊠値⊠⊠⊠ a ⊠値⊠⊠
    bool b2 = a < b ;

    a = 1 ;

    // false
    // ⊠⊠⊠⊠⊠値⊠⊠⊠⊠⊠値同士⊠比較
    // 1 < 0 ⊠ false
    bool b3 = a < b ;
}
```

4.3.13　optional と std::nullopt との比較

optional ⊠ std::nullopt ⊠⊠比較⊠、std::nullopt ⊠値⊠持⊠⊠⊠⊠⊠ optional ⊠⊠⊠扱⊠⊠⊠。

4.3.14　optional<T> と T の比較

optional<T> ⊠ T 型⊠比較⊠⊠、optional<t> ⊠値⊠保持⊠⊠⊠⊠⊠場合 false ⊠返⊠。⊠⊠以外⊠場合、optional ⊠保持⊠⊠⊠⊠値⊠ T ⊠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 ) ;
    // 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⊠⊠⊠⊠場合、in_place_type 構築⊠⊠⊠⊠⊠⊠⊠⊠⊠関数⊠選⊠⊠⊠。

```
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

string_view ⊠、文字型（char, wchar_t, char16_t, char32_t）⊠連続⊠⊠配列⊠表現⊠⊠⊠文字列⊠対⊠⊠共通⊠文字列⊠⊠⊠⊠提供⊠⊠。文字列⊠所有⊠⊠⊠。

5.1 使い方

連続⊠⊠文字型⊠配列⊠使⊠⊠文字列⊠表現方法⊠⊠⊠⊠⊠⊠⊠⊠。C++ ⊠⊠最⊠基本的⊠文字列⊠表現方法⊠⊠⊠、null 終端⊠⊠⊠文字型⊠配列⊠⊠⊠。

```
char str[6] = { 'h', 'e', 'l', 'l', 'o', '\0' } ;
```

⊠⊠⊠⊠、文字型⊠配列⊠文字数⊠表現⊠⊠⊠⊠⊠⊠⊠。

```
// size ⊠文字数
std::size_t size
char * ptr ;
```

⊠⊠⊠⊠⊠表現⊠⊠⊠⊠⊠管理⊠⊠⊠⊠面倒⊠⊠⊠、⊠⊠⊠⊠包⊠⊠⊠⊠⊠⊠。

```
class string_type
{
    std::size_t size ;
    char *ptr
} ;
```

⊠⊠⊠⊠⊠文字列⊠表現⊠⊠方法⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠対応⊠⊠⊠⊠⊠、表現⊠数⊠⊠関数⊠⊠⊠⊠⊠⊠⊠⊠⊠追加⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
// null 終端文字列用
void process_string( char * ptr ) ;
// 配列⊠⊠⊠⊠⊠⊠⊠⊠文字数
```

```
void process_string( char * ptr, std::size_t size ) ;
// std::string ⊠⊠⊠
void process_string( std::string s ) ;
// 自作⊠ string_type ⊠⊠⊠
void process_string( string_type s ) ;
// 自作⊠ my_string_type ⊠⊠⊠
void process_string( my_string_type s ) ;
```

string_view ⊠⊠⊠⊠⊠⊠表現⊠文字列⊠対⊠⊠共通⊠ view ⊠提供⊠⊠⊠⊠⊠、⊠⊠問題⊠解決⊠⊠⊠。⊠⊠関数⊠⊠⊠⊠⊠⊠⊠⊠⊠大量⊠追加⊠⊠必要⊠⊠⊠。

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

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

// ⊠⊠ 1 ⊠⊠⊠⊠⊠⊠
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 ⊠ std::basic_string< CharT, Traits > ⊠対⊠⊠ std::basic_string<char> ⊠⊠⊠⊠⊠⊠、std::string_view ⊠、⊠⊠実態⊠ 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 ⊠⊠⊠ typedef 名⊠使⊠⊠⊠⊠⊠⊠。本書⊠⊠ string_view ⊠⊠⊠解説⊠⊠⊠、⊠⊠他⊠ typedef 名⊠文字型⊠違⊠⊠⊠⊠同⊠⊠。

5.3 文字列の所有、非所有

string_view ⊠文字列⊠所有⊠⊠⊠。所有⊠⊠⊠⊠⊠、文字列⊠表現⊠⊠⊠⊠⊠⊠⊠⊠確保⊠破棄⊠責任⊠持⊠⊠⊠⊠⊠⊠⊠。所有⊠⊠⊠⊠⊠⊠意味⊠説明⊠⊠⊠⊠⊠、⊠⊠文字列⊠所有⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠説明⊠⊠。

std::string ⊠文字列⊠所有⊠⊠。std::string 風⊠⊠⊠⊠⊠実装⊠、⊠⊠⊠⊠以下⊠⊠⊠⊠⊠⊠。

```
class string
{
    std::size_t size ;
    char * ptr ;

public :
    // 文字列⊠表現⊠⊠⊠⊠⊠⊠⊠⊠動的確保
    string ( char const * str )
    {
        size = std::strlen( str ) ;
        ptr = new char[size+1] ;
        std::strcpy( ptr, str ) ;
    }

    // ⊠⊠⊠
    // 別⊠⊠⊠⊠⊠⊠⊠動的確保
    string ( string const & r )
        : size( r.size ), ptr ( new char[size+1] )
    {
        std::strcpy( ptr, r.ptr ) ;
    }

    // ⊠⊠⊠
    // 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⊠⊠⊠⊠⊠⊠⊠⊠解放
    ~string()
    {
        delete[] ptr ;
    }

} ;
```

`std::string` ⊠文字列⊠表現⊠⊠⊠⊠⊠⊠⊠⊠動的⊠確保⊠、所有⊠⊠。⊠⊠⊠⊠別⊠⊠⊠⊠⊠⊠⊠確保⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠所有権⊠移⊠。⊠⊠⊠⊠⊠⊠⊠⊠所有⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠破棄⊠⊠。

`std::string_view` ⊠文字列⊠所有⊠⊠⊠。`std::string_view` 風⊠⊠⊠⊠⊠実装⊠、⊠⊠⊠⊠以下⊠⊠⊠⊠⊠⊠。

```
class string_view
{
    std::size_t size ;
    char const * ptr ;
public :

    // 所有⊠⊠⊠
    // str ⊠参照先⊠寿命⊠呼⊠出⊠側⊠責任⊠持⊠
    string_view( char const * str ) noexcept
        : size( std::strlen(str) ), ptr( str )
    { }

    // ⊠⊠⊠
    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ default 化⊠⊠⊠⊠⊠⊠⊠
    string_view( string_view const & r ) noexcept = default ;

    // ⊠⊠⊠⊠⊠⊠⊠⊠同⊠
    // 所有⊠⊠⊠⊠⊠所有権⊠移動⊠⊠⊠

    // 破棄
    // 何⊠解放⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
} ;
```

string_view ⊠渡⊠⊠連続⊠⊠文字型⊠配列⊠⊠⊠⊠⊠⊠⊠⊠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 ⊠構築⊠⊠ 4 種類⊠⊠。

- ⊠⊠⊠⊠⊠構築
- null 終端⊠⊠⊠文字型⊠配列⊠⊠⊠⊠⊠⊠⊠
- 文字型⊠配列⊠⊠⊠⊠⊠⊠⊠⊠文字数
- 文字列⊠⊠⊠⊠⊠⊠変換関数

5.4.1　デフォルト構築

```
constexpr basic_string_view() noexcept;
```

string_view ⊠⊠⊠⊠⊠⊠構築⊠、空⊠ string_view ⊠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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、null 終端⊠⊠⊠文字型⊠⊠⊠⊠⊠⊠⊠⊠受⊠取⊠。

```
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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、文字型⊠配列⊠⊠⊠⊠⊠⊠⊠⊠文字数⊠受⊠取⊠。⊠⊠⊠⊠⊠⊠ null 終端⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
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⊠文字列⊠⊠⊠⊠⊠ string_view ⊠作⊠⊠⊠、変換関数⊠使⊠。string_view ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠⊠。

std::string ⊠ string_view ⊠⊠変換関数⊠⊠⊠⊠⊠⊠⊠⊠⊠。独自⊠文字列⊠⊠⊠⊠ string_view ⊠対応⊠⊠⊠⊠⊠変換関数⊠使⊠。⊠⊠⊠⊠以下⊠⊠⊠⊠実装⊠⊠。

```
class string
{
    std::size_t size ;
    char * ptr ;
public :
    operator std::string_view() const noexcept
    {
        return std::string_view( ptr, size ) ;
    }
} ;
```

⊠⊠⊠⊠⊠、std::string ⊠⊠ string_view ⊠⊠変換⊠可能⊠⊠⊠。

```
int main()
{
    std::string s = "hello" ;
    std::string_view sv = s ;
}
```

⊠⊠⊠同⊠方法⊠使⊠⊠、独自⊠文字列⊠⊠⊠⊠ string_view ⊠対応⊠⊠⊠⊠⊠⊠⊠⊠⊠。

std::string ⊠ string_view ⊠受⊠取⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠持⊠⊠⊠⊠⊠⊠、string_view ⊠⊠ string ⊠⊠変換⊠⊠⊠⊠。

```
int main()
{
    std::string_view sv = "hello" ;

    // ⊠⊠⊠⊠⊠⊠
    std::string s = sv ;
}
```

5.5 string_view の操作

string_view ⊠既存⊠標準⊠⊠⊠⊠⊠⊠ string ⊠⊠⊠同⊠操作性⊠提供⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠取⊠⊠⊠⊠⊠⊠⊠⊠、operator [] ⊠要素⊠⊠⊠⊠⊠⊠⊠⊠⊠、size() ⊠要素数⊠返⊠⊠、find() ⊠検索⊠⊠⊠⊠。

```
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 ⊠文字列⊠所有⊠⊠⊠⊠⊠、文字列⊠書⊠換⊠⊠方法⊠提供⊠⊠⊠⊠⊠。

```
int main()
{
    std::string s = "hello" ;
    s[0] = 'H' ;
    s += ",world" ;

    std::string_view sv = s ;

    // ⊠⊠⊠
    // string_view ⊠書⊠換⊠⊠⊠⊠⊠
    sv[0] = 'h' ;
    s += ".\n" ;
}
```

string_view ⊠文字列⊠所有⊠⊠、⊠⊠参照⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
int main()
{
    std::string s = "hello" ;
    std::string_view sv = s ;

    // "hello"
    std::cout << sv ;

    s = "world" ;

    // "world"
    // string_view ⊠参照⊠⊠⊠⊠⊠⊠
    std::cout << sv ;
}
```

string_view ⊠ string ⊠⊠⊠互換性⊠⊠⊠⊠⊠⊠⊠⊠持⊠⊠⊠⊠⊠、一部⊠文字列⊠変更⊠⊠⊠⊠⊠⊠⊠削除⊠⊠⊠⊠⊠。

5.5.1 remove_prefix/remove_suffix：先頭、末尾の要素の削除

string_view ⊠先頭⊠末尾⊠⊠ n 個⊠要素⊠削除⊠⊠⊠⊠⊠⊠関数⊠提供⊠⊠⊠⊠。

```
constexpr void remove_prefix(size_type n);
constexpr void remove_suffix(size_type n);
```

string_view ⊠⊠⊠⊠、先頭⊠末尾⊠⊠ n 個⊠要素⊠削除⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠ n 個⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠文字列⊠所有⊠⊠⊠ string_view ⊠⊠行⊠⊠操作⊠。

```
int main()
```

```
{
    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 ;
    // 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

⊠⊠⊠⊠⊠⊠⊠⊠ `<memory_resource>` ⊠定義⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、動的⊠⊠⊠⊠⊠⊠確保⊠⊠⊠⊠⊠ C++17 ⊠追加⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠特徴⊠以下⊠⊠⊠⊠。

- ⊠⊠⊠⊠⊠⊠⊠変⊠⊠新⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
- ⊠⊠⊠⊠⊠⊠⊠⊠⊠振⊠舞⊠⊠可能⊠⊠⊠⊠⊠⊠⊠⊠⊠
- 標準⊠提供⊠⊠⊠⊠⊠⊠⊠⊠特性⊠持⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実装

6.1　メモリーリソース

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠変⊠⊠新⊠⊠⊠⊠⊠⊠確保⊠解放⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠抽象⊠⊠⊠⊠。⊠⊠⊠⊠⊠時⊠挙動⊠変⊠⊠静的⊠⊠⊠⊠⊠⊠⊠⊠設計⊠⊠⊠⊠⊠⊠⊠⊠違⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠実行時⊠挙動⊠変⊠⊠動的⊠⊠⊠⊠⊠⊠⊠⊠設計⊠⊠⊠⊠⊠⊠。

```
void f( memory_resource * mem )
{
    // 10 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保
    auto ptr = mem->allocate( 10 ) ;
    // 確保⊠⊠⊠⊠⊠⊠⊠⊠解放
    mem->deallocate( ptr ) ;
}
```

⊠⊠⊠ `std::pmr::memory_resource` ⊠宣言⊠以下⊠⊠⊠⊠。

```
namespace std::pmr {
```

```
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⊠⊠⊠。

6.1.1　メモリーリソースの使い方

`memory_resource` ⊠使⊠⊠⊠簡単⊠。`memory_resource` ⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠⊠、⊠⊠⊠⊠関数 `allocate( bytes, alignment )` ⊠⊠⊠⊠⊠⊠⊠確保⊠⊠。⊠⊠⊠⊠関数 `deallocate( p, bytes, alignment )` ⊠⊠⊠⊠⊠⊠⊠解放⊠⊠。

```
void f( std::pmr::memory_resource * mem )
{
    // 100 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保
    void * ptr = mem->allocate( 100 ) ;
    // ⊠⊠⊠⊠⊠⊠解放
    mem->deallocate( ptr, 100 ) ;
}
```

2 ⊠⊠ `memory_resource` ⊠⊠⊠⊠⊠⊠⊠ a, b ⊠⊠⊠⊠⊠、一方⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠一方⊠⊠⊠⊠⊠⊠⊠⊠解放⊠⊠⊠⊠⊠、`a.is_equal( b )` ⊠ `true` ⊠返⊠。

```
void f( std::pmr::memory_resource * a, std::pmr::memory_resource * b )
{
    void * ptr = a->allocate( 1 ) ;
    // 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 メモリーリソースの作り方

独自⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `memory_resource` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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;
};
```

do_allocate(bytes, alignment) ⊠少⊠⊠⊠⊠ alignment ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ bytes ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠返⊠。⊠⊠⊠⊠⊠⊠確保⊠⊠⊠⊠⊠⊠場合⊠、適切⊠例外⊠ throw ⊠⊠。

do_deallocate(p, bytes, alignment) ⊠事前⊠同⊠ *this ⊠⊠呼⊠出⊠⊠⊠ allocate(bytes, alignment) ⊠返⊠⊠⊠⊠⊠⊠⊠⊠ p ⊠解放⊠⊠。⊠⊠⊠解放⊠⊠⊠⊠⊠⊠⊠⊠ p ⊠渡⊠⊠⊠⊠⊠⊠⊠。例外⊠投⊠⊠⊠。

do_is_equal(other) ⊠、*this ⊠ other ⊠互⊠⊠一方⊠確保⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠一方⊠解放⊠⊠⊠場合⊠ true ⊠返⊠。

⊠⊠⊠⊠、malloc/free ⊠使⊠⊠ memory_resource ⊠実装⊠以下⊠⊠⊠⊠。

```
// malloc/free ⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠
class malloc_resource : public std::pmr::memory_resource
{
public :
    //
    ~malloc_resource() { }
private :
    // ⊠⊠⊠⊠⊠⊠確保
    // 失敗⊠⊠場合 std::bad_alloc ⊠ 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
```

```
  {
    return dynamic_cast< const malloc_resource * >
          ( &other ) != nullptr ;
  }

} ;
```

do_allocate ⊠ malloc ⊠⊠⊠⊠⊠⊠⊠確保⊠、do_deallocate ⊠ free ⊠⊠⊠⊠⊠⊠⊠解放⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠⊠⊠⊠⊠⊠⊠⊠規定⊠⊠⊠⊠⊠、malloc ⊠挙動⊠任⊠⊠。malloc ⊠ 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠⊠⊠⊠⊠⊠⊠、C11 ⊠⊠規定⊠⊠⊠。POSIX ⊠⊠ null ⊠⊠⊠⊠⊠⊠返⊠⊠、free ⊠解放可能⊠何⊠⊠⊠⊠⊠⊠⊠⊠返⊠⊠⊠⊠⊠⊠⊠⊠。

do_is_equal ⊠、malloc_resource ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠解放⊠⊠⊠⊠⊠、*this ⊠ malloc_resource ⊠⊠⊠⊠⊠⊠⊠⊠dynamic_cast ⊠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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠動的⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠振⊠舞⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

従来⊠⊠⊠⊠⊠⊠⊠⊠、静的⊠⊠⊠⊠⊠⊠⊠⊠⊠実現⊠⊠⊠⊠⊠設計⊠⊠⊠⊠⊠。⊠⊠⊠⊠独自⊠ custom_int_allocator 型⊠使⊠⊠⊠場合⊠以下⊠⊠⊠⊠書⊠。

```
std::vector< int, custom_int_allocator > v ;
```

⊠⊠⊠⊠⊠時⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠決定⊠⊠⊠場合⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、実行時⊠⊠⊠⊠⊠⊠⊠⊠選択⊠⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠引数⊠取⊠設計⊠問題⊠⊠⊠。

⊠⊠⊠⊠、C++17 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠引数⊠取⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠実行時⊠⊠⊠⊠⊠⊠⊠⊠⊠振⊠舞⊠⊠⊠⊠std::pmr::polymorphic_allocator ⊠追加⊠⊠⊠。

⊠⊠⊠⊠、標準入力⊠⊠ true ⊠ false ⊠入力⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、monotonic_buffer_resource ⊠実行時⊠切⊠替⊠⊠⊠⊠、以下⊠⊠⊠⊠書⊠⊠。

```
int main()
{
    bool b;

    std::cin >> b ;

    std::pmr::memory_resource * mem ;
    std::unique_ptr< memory_resource > mono ;

    if ( b )
    { // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠使⊠
        mem = std::pmr::get_default_resource() ;
    }
    else
    { // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠使⊠
        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⊠⊠⊠⊠宣言⊠⊠⊠⊠⊠。

```
namespace std::pmr {

template <class T>
class polymorphic_allocator ;

}
```

⊠⊠⊠⊠⊠⊠実引数⊠⊠ `std::allocator<T>` ⊠同⊠⊠、確保⊠⊠型⊠与⊠⊠。

6.2.1　コンストラクター

```
polymorphic_allocator() noexcept;
polymorphic_allocator(memory_resource* r);
```

`std::pmr::polymorphic_allocator` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠ `std::pmr::get_default_resource()` ⊠取得⊠⊠。

`memory_resource *` ⊠引数⊠取⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、渡⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠使⊠。`polymorphic_allocator` ⊠生存期間中、⊠⊠⊠⊠⊠⊠⊠⊠

⊠⊠⊠⊠⊠⊠⊠⊠妥当⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
int main()
{
    // p1( std::pmr::get_default_resource () ) ⊠同⊠
    std::pmr::polymorphic_allocator<int> p1 ;

    std::pmr::polymorphic_allocator<int> p2(
        std::pmr::get_default_resource() ) ;
}
```

後⊠通常⊠⊠⊠⊠⊠⊠⊠⊠同⊠⊠⊠⊠振⊠舞⊠。

6.3 プログラム全体で使われるメモリーリソースの取得

C++17 ⊠⊠、⊠⊠⊠⊠⊠全体⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠取得⊠⊠⊠⊠⊠⊠⊠⊠。

6.3.1 new_delete_resource()

```
memory_resource* new_delete_resource() noexcept ;
```

関数 `new_delete_resource` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠返⊠。参照⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠確保⊠ `::operator new` ⊠使⊠、⊠⊠⊠⊠⊠⊠解放⊠ `::operator delete` ⊠使⊠。

```
int main()
{
    auto mem = std::pmr::new_delete_resource() ;
}
```

6.3.2 null_memory_resource()

```
memory_resource* null_memory_resource() noexcept ;
```

関数 `null_memory_resource` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠返⊠。参照⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `allocate` ⊠必⊠失敗⊠、`std::bad_alloc` ⊠ `throw` ⊠⊠。`deallocate` ⊠何⊠⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠確保⊠失敗⊠⊠場合⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠目的⊠使⊠⊠。

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⊠指定⊠⊠⊠⊠⊠⊠⊠⊠⊠場合⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠利用⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。初期値⊠ `new_delete_resource()` ⊠戻⊠値⊠⊠⊠⊠⊠⊠。

現在⊠⊠⊠⊠⊠⊠・⊠⊠⊠⊠⊠⊠⊠⊠・⊠⊠⊠⊠⊠⊠取得⊠⊠⊠⊠⊠⊠、関数 `get_default_resource` ⊠使⊠。⊠⊠⊠⊠⊠・⊠⊠⊠⊠⊠⊠⊠⊠・⊠⊠⊠⊠⊠⊠独自⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠差⊠替⊠⊠⊠⊠、関数 `set_default_resource` ⊠使⊠。

```
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 ) ;
}
```

⊠⊠⊠、残念⊠⊠⊠現実⊠⊠⊠⊠⊠⊠⊠⊠ OS ⊠⊠⊠⊠⊠管理⊠、⊠⊠⊠⊠⊠柔軟⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ OS ⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠呼⊠⊠⊠単位⊠⊠⊠確保⊠⊠⊠⊠。⊠⊠⊠最小⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ 4K ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠低級⊠⊠⊠⊠⊠管理⊠使⊠⊠上⊠⊠⊠⊠⊠実装⊠⊠⊠⊠⊠⊠⊠、47 ⊠⊠⊠程度⊠⊠⊠⊠⊠⊠使⊠⊠⊠ 3K ⊠⊠⊠超⊠無駄⊠生⊠⊠⊠⊠⊠⊠⊠。

他⊠⊠⊠⊠⊠⊠⊠⊠⊠問題⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠適切⊠⊠⊠⊠⊠⊠⊠⊠配置⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、著⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠落⊠⊠⊠⊠⊠⊠⊠。

`malloc` ⊠ `operator new` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、低級⊠⊠⊠⊠⊠管理⊠隠匿⊠、小⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠効率的⊠行⊠⊠⊠⊠実装⊠⊠⊠⊠⊠。

一般的⊠⊠、大⊠⊠連続⊠⊠⊠⊠⊠⊠空間⊠⊠⊠⊠⊠⊠確保⊠、⊠⊠中⊠管理用⊠⊠⊠⊠構造⊠作⊠、⊠⊠⊠⊠⊠必要⊠⊠⊠⊠⊠切⊠出⊠。

```
// 実装⊠⊠⊠⊠

// ⊠⊠⊠⊠⊠⊠分割⊠⊠管理⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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 ; // ⊠⊠⊠⊠⊠⊠先頭⊠⊠⊠⊠⊠⊠⊠
    std::size_t size ; // ⊠⊠⊠⊠⊠⊠⊠⊠⊠
    std::mutex m ; // 同期用
public :

    memory_allocator()
    {
        // 大⊠⊠連続⊠⊠⊠⊠⊠⊠⊠⊠確保
    }

    virtual void *
    do_allocate( std::size_t bytes, std::size_t alignment ) override
    {
        std::scoped_lock lock( m ) ;
        // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、十分⊠大⊠⊠⊠未使用領域⊠探⊠、⊠⊠⊠⊠⊠⊠構造体⊠
        // 構築⊠⊠返⊠
        // ⊠⊠⊠⊠⊠⊠要求⊠注意
    }

    virtual void *
    do_allocate( std::size_t bytes, std::size_t alignment ) override
    {
        std::scoped_lock lock( m ) ;
        // ⊠⊠⊠⊠⊠⊠⊠⊠該当⊠⊠部分⊠削除
    }

    virtual bool
    do_is_equal( const memory_resource & other )
        const noexcept override
    {
    // *this ⊠ other ⊠相互⊠⊠⊠⊠⊠⊠⊠解放⊠⊠⊠⊠⊠⊠⊠⊠返⊠
    }
} ;
```

6.5　プールリソース

⊠⊠⊠⊠⊠⊠⊠⊠ C++17 ⊠標準⊠⊠⊠⊠⊠⊠提供⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実装⊠。`synchronized_pool_resource` ⊠ `unsynchronized_pool_resource` ⊠ 2 ⊠⊠⊠⊠。

6.5.1　アルゴリズム

⊠⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠特徴⊠持⊠。

- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠破棄⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `allocate` ⊠確保⊠⊠⊠⊠⊠⊠⊠⊠、明示的⊠ `deallocate` ⊠呼⊠⊠⊠⊠解放⊠⊠⊠。

```
void f()
{
    std::pmr::synchronized_pool_resource mem ;
    mem.allocate( 10 ) ;

    // 確保⊠⊠⊠⊠⊠⊠⊠⊠破棄⊠⊠⊠
}
```

- ⊠⊠⊠⊠⊠⊠⊠⊠構築時⊠、上流⊠⊠⊠⊠⊠⊠⊠⊠⊠与⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠上流⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠。

```
int main()
{
    // get_default_resource() ⊠使⊠⊠⊠
    std::pmr::synchronized_pool_resource m1 ;

    // 独自⊠上流⊠⊠⊠⊠⊠⊠⊠⊠⊠指定
    custom_memory_resource mem ;
    std::pmr::synchronized_pool_resource m2( &mem ) ;

}
```

- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠上流⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠呼⊠⊠⊠複数⊠⊠⊠⊠⊠⊠⊠確保⊠⊠。⊠⊠⊠⊠複数⊠⊠⊠⊠⊠⊠保持⊠⊠⊠⊠。⊠⊠⊠⊠⊠複数⊠同一⊠⊠⊠⊠⊠⊠⊠⊠⊠保持⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠対⊠⊠ `do_allocate(size, alignment)` ⊠、少⊠⊠⊠⊠ `size` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠割⊠当⊠⊠⊠⊠。
  ⊠⊠、最大⊠⊠⊠⊠⊠⊠⊠⊠⊠超⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠⊠⊠⊠⊠場合、上流⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠⊠。

```
// 実装⊠⊠⊠⊠

namespace std::pmr {

// ⊠⊠⊠⊠⊠実装
template < size_t block_size >
class chunk
{
    blocks<block_size> b ;
}

// ⊠⊠⊠⊠実装
template < size_t block_size >
class pool : public memory_resource
{
    chunks<block_size> c ;
} ;

class pool_resource : public memory_resource
{
    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
    pool<8> pool_8bytes ;
    pool<16> pool_16bytes ;
    pool<32> pool_32bytes ;

    // 上流⊠⊠⊠⊠⊠⊠⊠⊠
    memory_resource * mem ;


    virtual void * do_allocate( size_t bytes, size_t alignment ) override
    {
        // 対応⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
        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⊠⊠⊠⊠⊠⊠⊠⊠超⊠⊠⊠⊠上流⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
            return mem->allocate( bytes, alignment ) ;
    }
} ;

}
```

- ⊠⊠⊠⊠⊠⊠⊠⊠構築時⊠ `pool_options` ⊠渡⊠⊠⊠⊠⊠⊠、最大⊠⊠⊠⊠⊠⊠⊠⊠最大⊠⊠⊠⊠⊠⊠⊠⊠設定⊠⊠⊠。
- ⊠⊠⊠⊠⊠⊠⊠⊠⊠呼⊠出⊠⊠⊠安全⊠同期⊠取⊠ `synchronized_pool_resource` ⊠、同期⊠取⊠⊠⊠ `unsynchronized_pool_resource` ⊠⊠⊠。

6.5.2 synchronized/unsynchronized_pool_resource

⊠⊠⊠⊠⊠⊠⊠⊠⊠、`synchronized_pool_resource` ⊠ `unsynchronized_pool_resource`⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠名以外⊠同⊠⊠⊠⊠使⊠⊠。⊠⊠⊠、`synchronized_pool_resource` ⊠複数⊠⊠⊠⊠⊠⊠⊠同時⊠実行⊠⊠⊠使⊠⊠⊠⊠⊠内部⊠同期⊠取⊠⊠⊠⊠⊠⊠⊠対⊠、`unsynchronized_pool_resource` ⊠同期⊠行⊠⊠⊠。`unsynchronized_pool_resource` ⊠複数⊠⊠⊠⊠⊠⊠⊠同時⊠呼⊠出⊠⊠⊠⊠⊠⊠⊠⊠。

```
// 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 ) ;
    }
} ;
}
```

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;
};

}
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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⊠⊠⊠。実装⊠指定⊠⊠⊠小⊠⊠値⊠使⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠別⊠値⊠使⊠⊠⊠⊠⊠⊠⊠。

`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⊠⊠⊠。実装⊠指定⊠⊠⊠大⊠⊠値⊠使⊠⊠⊠⊠⊠⊠⊠。

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) {}
```

```
explicit pool_resource(const pool_options& opts)
: pool_resource(opts, get_default_resource()) {}
```

pool_options ⊠ memory_resource * ⊠指定⊠⊠。指定⊠⊠⊠場合⊠⊠⊠⊠⊠⊠値⊠使⊠⊠⊠。

6.5.5 プールリソースのメンバー関数

release()

```
void release();
```

確保⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠解放⊠⊠。⊠⊠⊠明示的⊠ deallocate ⊠呼⊠出⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠解放⊠⊠。

```
int main()
{
    synchronized_pool_resource mem ;
    void * ptr = mem.allocate( 10 ) ;

    // ptr ⊠解放⊠⊠⊠
    mem.release() ;

}
```

upstream_resource()

```
memory_resource* upstream_resource() const;
```

構築時⊠渡⊠⊠上流⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠返⊠。

options()

```
pool_options options() const;
```

構築時⊠渡⊠⊠ pool_options ⊠⊠⊠⊠⊠⊠⊠同⊠値⊠返⊠。

6.6 モノトニックバッファーリソース

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ C++17 ⊠標準⊠⊠⊠⊠⊠⊠追加⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実装⊠。⊠⊠⊠名⊠ monotonic_buffer_resource。

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠高速⊠⊠⊠⊠⊠⊠確保⊠、一気⊠解放⊠⊠⊠⊠

⊠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⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠、⊠⊠後確保⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠解放⊠⊠⊠場合⊠考⊠⊠。通常⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠片⊠解放⊠⊠⊠⊠⊠⊠⊠⊠⊠全体⊠構築⊠⊠⊠⊠⊠⊠構造⊠⊠⊠⊠、⊠⊠⊠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
    }
} ;

}
```

☒☒☒☒☒、基本的☒実装☒☒☒☒、do_allocate ☒☒☒☒☒☒☒加算☒☒管理☒☒☒☒☒。☒☒☒☒☒解放処理☒☒☒☒☒☒☒、個々☒☒☒☒☒☒☒片☒管理☒☒☒☒☒☒☒☒構造☒構築☒☒必要☒☒☒。do_deallocate ☒何☒☒☒☒。☒☒☒☒☒☒☒☒☒☒☒☒☒全体☒解放☒☒。

6.6.1 アルゴリズム

☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒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
    mem.allocate( 1 ) ;
    // 上流⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠切⊠出⊠⊠確保
    mem.allocate( 100 ) ;
    // 前回⊠⊠⊠⊠⊠⊠確保⊠空⊠⊠⊠⊠⊠⊠⊠⊠⊠
    // ⊠⊠⊠⊠新⊠⊠上流⊠⊠確保⊠⊠切⊠出⊠
    mem.allocate( 100 ) ;
}
```

- 1 ⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠⊠⊠前提⊠設計⊠⊠⊠⊠⊠。`allocate` ⊠ `deallocate` ⊠同期⊠⊠⊠。
- ⊠⊠⊠⊠⊠⊠⊠⊠⊠破棄⊠⊠⊠⊠確保⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠解放⊠⊠⊠。明示的⊠ `deallocate` ⊠呼⊠⊠⊠⊠⊠⊠⊠。

6.6.2　コンストラクター

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
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⊠⊠⊠⊠⊠⊠取⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠。

```
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()) {}
```

```
explicit monotonic_buffer_resource(size_t initial_size)
    : monotonic_buffer_resource(initial_size,
                                get_default_resource()) {}
```

initial_size ⊠、上流⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠最初⊠確保⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠(初期⊠⊠⊠）⊠⊠⊠⊠⊠⊠⊠。実装⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠実装依存⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠確保⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠上流⊠⊠⊠⊠⊠⊠⊠⊠⊠ std::pmr_get_default_resource()⊠与⊠⊠⊠⊠同⊠挙動⊠⊠⊠。

size_t1 ⊠⊠⊠⊠取⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、初期⊠⊠⊠⊠⊠⊠与⊠⊠後⊠⊠⊠⊠⊠⊠⊠扱⊠⊠⊠⊠。

初期⊠⊠⊠⊠⊠⊠取⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠。

```
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⊠⊠⊠⊠⊠⊠先頭⊠⊠⊠⊠⊠ void * 型⊠渡⊠、⊠⊠⊠⊠⊠⊠ size_t 型⊠渡⊠。

6.6.3 その他の操作

release()

```
void release() ;
```

⊠⊠⊠⊠関数 release ⊠、上流⊠⊠⊠⊠⊠⊠確保⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠解放⊠⊠。明示的⊠ deallocate ⊠呼⊠出⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠解放⊠⊠⊠。

```
int main()
{
    std::pmr::monotonic_buffer_resource mem ;

    mem.allocate( 10 ) ;

    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠解放⊠⊠⊠
    mem.release() ;

}
```

upstream_resource()

```
memory_resource* upstream_resource() const;
```

⊠⊠⊠⊠関数 `upstream_resource` ⊠、構築時⊠与⊠⊠⊠⊠上流⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠返⊠。

第 7 章

並列アルゴリズム

並列⊠⊠⊠⊠⊠⊠⊠ C++17 ⊠追加⊠⊠⊠新⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠既存⊠ `<algorithm>` ⊠、並列実行版⊠追加⊠⊠。

7.1 並列実行について

C++11 ⊠⊠、⊠⊠⊠⊠⊠同期処理⊠追加⊠⊠、複数⊠実行媒体⊠同時⊠実行⊠⊠⊠⊠⊠⊠概念⊠ C++ 標準規格⊠入⊠⊠。

C++17 ⊠⊠、既存⊠⊠⊠⊠⊠⊠⊠⊠、並列実行版⊠追加⊠⊠⊠。

⊠⊠⊠⊠、`all_of(first, last, pred)` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、`[first,last)` ⊠区間⊠空⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `i` ⊠対⊠⊠ `pred(*i)` ⊠ `true` ⊠返⊠⊠⊠、`true` ⊠返⊠。⊠⊠以外⊠場合⊠ `false` ⊠返⊠。

⊠⊠⊠⊠値⊠ 100 未満⊠⊠⊠⊠⊠⊠⊠⊠調⊠⊠⊠⊠、以下⊠⊠⊠⊠書⊠。

```
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 ) ;
    std::cout << "result : " << result << std::endl ;
}
```

本書⊠執筆時点⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠一般的⊠⊠⊠、同時⊠複数⊠⊠⊠⊠⊠⊠実行⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠処理⊠2⊠⊠⊠⊠⊠⊠⊠並列化⊠⊠⊠⊠⊠。

```
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() ;
}
```

⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

筆者⊠⊠⊠⊠⊠⊠⊠⊠⊠CPU⊠2⊠⊠物理⊠⊠、4⊠⊠論理⊠⊠⊠持⊠⊠⊠⊠⊠⊠、4⊠⊠⊠⊠⊠⊠同時⊠並列実行⊠⊠⊠。読者⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠高性能⊠⊠⊠⊠多⊠⊠⊠⊠⊠⊠⊠同時⊠実行可能⊠⊠⊠。実行時⊠最大⊠効率⊠出⊠⊠⊠⊠⊠⊠⊠⊠⊠頑張⊠⊠⊠⊠⊠。

```
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⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧対⌧⌧自前⌧実装⌧⌧⌧⌧面倒⌧。⌧⌧⌧、C++17 ⌧⌧標準⌧並列実行⌧⌧⌧⌧⌧並列⌧⌧⌧⌧⌧⌧（Parallelism）⌧追加⌧⌧⌧。

7.2 使い方

並列⌧⌧⌧⌧⌧⌧⌧既存⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧追加⌧⌧⌧⌧⌧。以下⌧既存⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ `all_of` ⌧宣言⌧。

```
template <class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last, Predicate pred);
```

並列⊠⊠⊠⊠⊠⊠版⊠ all_of ⊠以下⊠⊠⊠⊠宣言⊠⊠⊠。

```
template <  class ExecutionPolicy, class ForwardIterator,
            class Predicate>
bool all_of(ExecutionPolicy&& exec, ForwardIterator first,
            ForwardIterator last, Predicate pred);
```

並列⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠仮引数⊠⊠⊠ ExecutionPolicy ⊠追加⊠⊠⊠⊠⊠第一引数⊠取⊠。⊠⊠⊠実行時⊠⊠⊠⊠⊠呼⊠。

実行時⊠⊠⊠⊠⊠ `<execution>` ⊠定義⊠⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠用⊠⊠⊠型⊠、`std::execution::seq`, `std::execution::par`, `std::execution::par_unseq` ⊠⊠⊠。

複数⊠⊠⊠⊠⊠⊠⊠⊠並列実行⊠行⊠⊠⊠、`std::execution::par` ⊠使⊠。

```
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` ⊠渡⊠⊠既存⊠⊠⊠⊠⊠⊠⊠⊠同⊠⊠⊠⊠⊠⊠⊠⊠実行⊠⊠⊠。`std::execution::par` ⊠渡⊠⊠⊠⊠⊠⊠実行⊠⊠⊠。`std::execution::par_unseq` ⊠並列実行⊠⊠⊠⊠⊠⊠実行⊠⊠⊠。

C++17 ⊠⊠実行⊠⊠⊠⊠⊠受⊠取⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠関数⊠追加⊠⊠⊠⊠⊠。

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⊠⊠⊠⊠⊠⊠（parallel algorithm）⊠⊠、`ExecutionPolicy`（実行⊠⊠⊠⊠）⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。既存⊠ `<algorithm>` ⊠ C++14 ⊠追加⊠⊠⊠一部⊠関数⊠⊠⊠⊠⊠⊠⊠、並列⊠⊠⊠⊠⊠⊠⊠対応⊠⊠⊠⊠。

並列⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、仕様上定⊠⊠⊠⊠操作、⊠⊠⊠⊠⊠提供⊠⊠関数⊠⊠⊠⊠⊠⊠⊠⊠⊠操作、仕様上定⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠⊠対⊠⊠操作⊠⊠⊠

⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠関数群⊠、要素⊠⊠⊠⊠関数（element access functions）⊠呼⊠。

⊠⊠⊠⊠、`std::sort` ⊠以下⊠⊠⊠⊠要素⊠⊠⊠⊠関数⊠持⊠。

- ⊠⊠⊠⊠⊠⊠実引数⊠与⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
- 要素⊠対⊠⊠ `swap` 関数⊠適用
- ⊠⊠⊠⊠提供⊠⊠⊠ `Compare` 関数⊠⊠⊠⊠⊠⊠

並列⊠⊠⊠⊠⊠⊠⊠使⊠要素⊠⊠⊠⊠関数⊠、並列実行⊠伴⊠⊠⊠⊠⊠⊠制約⊠満⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

7.3.2 ユーザー提供する関数オブジェクトの制約

並列⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠名⊠、`Predicate`, `BinaryPredicate`, `Compare`, `UnaryOperation`, `BinaryOperation`, `BinaryOperation1`, `BinaryOperation2` ⊠⊠⊠⊠⊠⊠⊠⊠⊠、関数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠提供⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠提供⊠関数⊠⊠⊠⊠⊠⊠⊠⊠、並列⊠⊠⊠⊠⊠⊠⊠渡⊠際⊠制約⊠⊠⊠。

- 実引数⊠与⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠直接、間接⊠変更⊠⊠⊠⊠⊠⊠⊠
- 実引数⊠与⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠一意性⊠依存⊠⊠⊠⊠⊠⊠⊠
- ⊠⊠⊠競合⊠同期

一部⊠特殊⊠⊠⊠⊠⊠⊠⊠⊠⊠例外⊠⊠⊠⊠、⊠⊠⊠⊠⊠並列⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠制約⊠満⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

実引数⊠与⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠直接、間接⊠変更⊠⊠⊠⊠⊠⊠⊠

⊠⊠⊠⊠提供⊠関数⊠⊠⊠⊠⊠⊠⊠実引数⊠与⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠直接、間接⊠変更⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠、以下⊠⊠⊠⊠⊠⊠⊠⊠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⊠⊠⊠⊠⊠⊠⊠実引数⊠ `lvalue` ⊠⊠⊠⊠⊠⊠⊠受⊠取⊠⊠変更⊠⊠⊠⊠⊠⊠、並列⊠⊠⊠⊠⊠⊠⊠制約⊠満⊠⊠⊠⊠。

std::for_each ⊠⊠⊠⊠⊠⊠⊠⊠変更可能⊠要素⊠返⊠場合、⊠⊠⊠⊠提供⊠関数⊠⊠⊠⊠⊠⊠⊠実引数⊠変更⊠⊠⊠⊠⊠可能⊠。

```
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
}
```

⊠⊠⊠、for_each ⊠仕様上⊠⊠⊠⊠⊠定⊠⊠⊠⊠⊠⊠⊠⊠⊠。

実引数⊠与⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠一意性⊠依存⊠⊠⊠⊠⊠⊠⊠

⊠⊠⊠⊠提供⊠関数⊠⊠⊠⊠⊠⊠⊠実引数⊠与⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠一意性⊠依存⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠実引数⊠渡⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠取得⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠渡⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠同⊠⊠⊠⊠⊠⊠⊠期待⊠⊠⊠⊠⊠⊠⊠⊠⊠書⊠⊠⊠⊠⊠⊠⊠⊠。

```
int main()
{
    std::vector<int> c = { 1,2,3,4,5 } ;

    // 最後⊠要素⊠⊠⊠⊠⊠⊠⊠
    int * ptr = &c[4] ;

    std::all_of( std::execution::par, std::begin(c), std::end(c),
        [=]( auto & x ){
            if ( ptr == &x )
            {
                // 最後⊠要素⊠⊠⊠特別⊠処理
                // ⊠⊠⊠
            }
        } ) ;
}
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、並列⊠⊠⊠⊠⊠⊠⊠⊠⊠並列処理⊠一環⊠⊠⊠、要素⊠⊠⊠⊠⊠作成⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠提供⊠関数⊠⊠⊠⊠⊠⊠⊠渡⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
// 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☒☒☒☒☒☒書☒☒☒☒☒☒☒☒。

☒☒☒競合☒同期

`std::execution::sequenced_policy` ☒渡☒☒並列☒☒☒☒☒☒☒☒☒要素☒☒☒☒関数☒呼☒出☒☒呼☒出☒側☒☒☒☒☒実行☒☒☒。☒☒☒☒実行☒☒☒☒。

`std::execution::parallel_policy` ☒渡☒☒並列☒☒☒☒☒☒☒☒☒要素☒☒☒☒関数☒呼☒出☒☒、呼☒出☒側☒☒☒☒☒、☒☒☒☒☒側☒作☒☒☒☒☒☒☒☒☒☒☒☒☒実行☒☒☒。☒☒☒☒☒要素☒☒☒☒関数☒呼☒出☒☒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
}
```

⊠⊠⊠⊠⊠、⊠⊠⊠⊠提供⊠関数⊠⊠⊠⊠⊠⊠⊠複数⊠⊠⊠⊠⊠⊠⊠同時⊠呼⊠出⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

`std::execution::parallel_unsequenced_policy` ⊠実行⊠変⊠⊠⊠⊠⊠。未規定⊠⊠⊠⊠⊠⊠⊠同期⊠⊠⊠⊠実行⊠許⊠⊠⊠⊠⊠。⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠実行⊠想定⊠⊠⊠⊠実行媒体⊠⊠⊠⊠⊠⊠⊠⊠⊠強⊠実行保証⊠⊠⊠実行媒体⊠⊠⊠⊠、SIMD ⊠ GPGPU ⊠⊠⊠⊠極⊠⊠軽⊠実行媒体⊠⊠⊠⊠⊠⊠。

⊠⊠結果、要素⊠⊠⊠⊠関数⊠通常⊠⊠⊠⊠競合⊠⊠⊠⊠⊠⊠⊠⊠防⊠⊠⊠⊠手段⊠⊠取⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠実行⊠途中⊠中断⊠⊠別⊠処理⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠、以下⊠⊠⊠⊠⊠動⊠⊠⊠。

```
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 ;
        } ) ;
    // ⊠⊠⊠
}
```

⊠⊠⊠⊠⊠⊠ `parallel_policy` ⊠⊠⊠、非効率的⊠⊠⊠⊠⊠問題⊠⊠同期⊠⊠⊠⊠⊠⊠競合⊠⊠動⊠⊠⊠⊠⊠。⊠⊠⊠、`parallel_unsequenced_policy` ⊠⊠動⊠⊠⊠。⊠⊠⊠⊠⊠、`mutex` ⊠ `lock` ⊠⊠⊠同期⊠⊠⊠関数⊠呼⊠出⊠⊠⊠⊠。

C++ ⊠⊠、⊠⊠⊠⊠⊠⊠確保解放以外⊠同期⊠⊠標準⊠⊠⊠⊠⊠⊠関数⊠⊠⊠⊠、⊠⊠⊠⊠化非安全（vectorization-unsafe）⊠分類⊠⊠⊠⊠。⊠⊠⊠⊠化非安全⊠関数⊠ `std::execution::parallel_unsequenced_policy` ⊠要素⊠⊠⊠⊠関数内⊠呼⊠出⊠⊠⊠⊠⊠⊠⊠⊠。

7.3.3　例外

並列⊠⊠⊠⊠⊠⊠⊠実行中⊠、一時⊠⊠⊠⊠⊠確保⊠必要⊠⊠⊠⊠⊠確保⊠⊠⊠⊠場合、`std::bad_alloc` ⊠ `throw` ⊠⊠⊠。

並列⊠⊠⊠⊠⊠⊠⊠実行中⊠、要素⊠⊠⊠⊠関数⊠外⊠例外⊠投⊠⊠⊠⊠場合、`std::terminate` ⊠呼⊠⊠⊠。

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⊠⊠⊠⊠⊠⊠⊠⊠返⊠ traits ⊠。

```
// false
constexpr bool b1 = std::is_execution_policy_v<int> ;
// true
constexpr bool b2 =
    std::is_execution_policy_v<std::execution::sequenced_policy> ;
```

⊠⊠⊠⊠⊠実行⊠⊠⊠⊠

```
namespace std::execution {

class sequenced_policy ;
inline constexpr sequenced_policy seq { } ;

}
```

⊠⊠⊠⊠⊠実行⊠⊠⊠⊠⊠、並列⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実行⊠行⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠実行⊠⊠⊠⊠⊠渡⊠⊠⊠場合、処理⊠呼⊠出⊠元⊠⊠⊠⊠⊠⊠⊠⊠行⊠⊠⊠。

⊠⊠⊠⊠実行⊠⊠⊠⊠

```
namespace std::execution {

class parallel_policy ;
inline constexpr parallel_policy par { } ;

}
```

⊠⊠⊠⊠実行⊠⊠⊠⊠⊠、並列⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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 ⊠ GPGPU ⊠⊠⊠⊠⊠⊠⊠⊠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{ };
}
```

実行⊠⊠⊠⊠⊠型⊠直接書⊠⊠⊠面倒⊠。

```
std::for_each( std::execution::parallel_policy{}, ... ) ;
```

⊠⊠⊠⊠、標準⊠⊠⊠⊠⊠⊠実行⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠用意⊠⊠⊠⊠。`seq` ⊠ `par` ⊠ `par_unseq` ⊠⊠⊠。

```
std::for_each( std::execution::par, ... ) ;
```

並列⊠⊠⊠⊠⊠⊠⊠使⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠並列⊠⊠⊠⊠⊠⊠⊠第一引数⊠渡⊠⊠⊠⊠⊠⊠。

第 8 章

数学の特殊関数群

C++17 ⌧⌧数学⌧特殊関数群（mathematical special functions）⌧⌧⌧⌧⌧⌧⌧⌧⌧ `<cmath>` ⌧追加⌧⌧⌧。

数学⌧特殊関数⌧、⌧⌧⌧⌧実引数⌧取⌧⌧、規定⌧計算⌧⌧、結果⌧浮動小数点数型⌧戻⌧値⌧⌧⌧返⌧。

数学⌧特殊関数⌧ `double`, `float`, `long double` 型⌧ 3 ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧。⌧⌧⌧⌧、関数名⌧最後⌧、何⌧⌧⌧、`f`, `l` ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧表現⌧⌧⌧⌧⌧。

```
double      function_name() ;   // 何⌧⌧⌧
float       function_namef() ;  // f
long double function_namel() ;  // l
```

数学⌧特殊関数⌧説明⌧、関数⌧宣言、効果、戻⌧値、注意⌧⌧⌧。

⌧⌧、数学⌧特殊関数⌧渡⌧⌧実引数⌧ NaN（Not a Number）⌧⌧⌧場合、関数⌧戻⌧値⌧ NaN ⌧⌧⌧。⌧⌧⌧定義域⌧⌧⌧⌧起⌧⌧⌧⌧。

⌧⌧以外⌧場合⌧、関数⌧定義域⌧⌧⌧⌧返⌧⌧⌧⌧⌧⌧、

- 関数⌧戻⌧値⌧記述⌧、定義域⌧示⌧⌧⌧⌧⌧実引数⌧示⌧⌧⌧定義域⌧超⌧⌧⌧⌧
- 実引数⌧対応⌧⌧数学関数⌧結果⌧値⌧非⌧⌧⌧虚数部⌧含⌧⌧⌧
- 実引数⌧対応⌧⌧数学関数⌧結果⌧値⌧数学的⌧定義⌧⌧⌧⌧⌧⌧⌧⌧

別途示⌧⌧⌧⌧⌧⌧場合、関数⌧⌧⌧⌧⌧有限⌧値、負⌧無限大、正⌧無限大⌧対⌧⌧⌧定義⌧⌧⌧⌧⌧。

数学関数⌧与⌧⌧⌧⌧実引数⌧値⌧対⌧⌧定義⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧、⌧⌧⌧以下⌧⌧⌧⌧⌧⌧⌧⌧。

- 実引数⌧値⌧集合⌧対⌧⌧明示的⌧定義⌧⌧⌧⌧⌧
- 計算方法⌧依存⌧⌧⌧極限値⌧存在⌧⌧

⊠⊠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⊠⊠⊠⊠⊠⊠多項式（Laguerre polynomials）⊠計算⊠⊠。
戻⊠値：

$$\mathsf{L}_n(x) = \frac{e^x}{n!}\frac{\mathsf{d}^n}{\mathsf{d}x^n}\,(x^n e^{-x}), \quad \text{for } x \geq 0$$

n ⊠ `n`, x ⊠ `x` ⊠⊠⊠。
注意： `n` >= 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⊠⊠⊠⊠⊠⊠陪多項式（Associated Laguerre polynomials）⊠計算⊠⊠。
戻⊠値：

$$\mathsf{L}_n^m(x) = (-1)^m \frac{\mathsf{d}^m}{\mathsf{d}x^m}\,\mathsf{L}_{n+m}(x), \quad \text{for } x \geq 0$$

n ⊠ `n`, m ⊠ `m`, x ⊠ `x` ⊠⊠⊠。
注意： `n` >= 128 ⊠⊠⊠⊠ `m` >= 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⊠⊠⊠⊠⊠⊠⊠⊠多項式 (Legendre polynomials) ⊠計算⊠⊠。

戻⊠値：

$$\mathsf{P}_\ell(x) = \frac{1}{2^\ell\,\ell!}\frac{\mathsf{d}^\ell}{\mathsf{d}x^\ell}\,(x^2-1)^\ell, \quad \text{for } |x| \le 1$$

l ⊠ `l`, x ⊠ `x` ⊠⊠⊠。

注意： `l` >= 128 ⊠⊠⊠⊠関数⊠呼⊠出⊠⊠効果⊠実装定義⊠⊠⊠。

8.4　ルジャンドル陪関数（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 `l`, `m`, `x` ⊠対⊠⊠⊠⊠⊠⊠⊠⊠陪関数（Associated Legendre functions）⊠計算⊠⊠。

戻⊠値：

$$\mathsf{P}_\ell^m(x) = (1-x^2)^{m/2}\,\frac{\mathsf{d}^m}{\mathsf{d}x^m}\,\mathsf{P}_\ell(x), \quad \text{for } |x| \le 1$$

l ⊠ `l`, m ⊠ `m`, x ⊠ `x` ⊠⊠⊠。

注意： `l` >= 128 ⊠⊠⊠⊠関数呼⊠出⊠⊠効果⊠実装定義⊠⊠⊠。

8.5　球面ルジャンドル陪関数（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 `l`, `m`, `theta`（`theta` ⊠単位⊠⊠⊠⊠⊠）⊠対⊠⊠球面⊠⊠⊠⊠⊠⊠陪関数（Spherical associated Legendre functions）⊠計算⊠⊠。

戻⊠値：

$$\mathsf{Y}_\ell^m(\theta, 0)$$

⊠⊠⊠⊠、

$$\mathsf{Y}_\ell^m(\theta,\phi) = (-1)^m \left[\frac{(2\ell+1)}{4\pi}\frac{(\ell-m)!}{(\ell+m)!}\right]^{1/2} \mathsf{P}_\ell^m(\cos\theta)e^{im\phi}, \quad \text{for } |m| \le \ell$$

l ⊠ `l`, m ⊠ `m`, θ ⊠ `theta` ⊠⊠⊠。

注意： `l` >= 128 ⊠⊠⊠⊠関数⊠呼⊠出⊠⊠効果⊠実装定義⊠⊠⊠。

球面調和関数（Spherical harmonics） $\mathsf{Y}_\ell^m(\theta, \phi)$ ⊠、以下⊠⊠⊠⊠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⊠⊠⊠⊠⊠⊠⊠多項式（Hermite polynomials）⊠計算⊠⊠。

戻⊠値：

$$\mathsf{H}_n(x) = (-1)^n e^{x^2} \frac{\mathsf{d}^n}{\mathsf{d}x^n} e^{-x^2}$$

n ⊠ `n`, x ⊠ `x` ⊠⊠⊠。

注意： `n` >= 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⊠⊠⊠⊠⊠関数（Beta function）⊠計算⊠⊠。

戻⊠値：

$$\mathsf{B}(x, y) = \frac{\Gamma(x)\,\Gamma(y)}{\Gamma(x+y)}, \quad \text{for } x > 0,\ y > 0$$

x ⊠ `x`, y ⊠ `y` ⊠⊠⊠。

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` ☒対☒☒第 1 種完全楕円積分（Complete elliptic integral of the first kind）☒計算☒☒。

戻☒値：

$$\mathsf{K}(k) = \mathsf{F}(k, \pi/2), \quad \text{for } |k| \leq 1$$

k ☒ `k` ☒☒☒。

第 1 種不完全楕円積分☒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` ☒対☒☒第 2 種完全楕円積分（Complete elliptic integral of the second kind）☒計算☒☒。

戻☒値：

$$\mathsf{E}(k) = \mathsf{E}(k, \pi/2), \quad \text{for } |k| \leq 1$$

k ☒ `k` ☒☒☒。

第 2 種不完全楕円積分☒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);
```

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`, ν ⊠ `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：

$$\mathsf{F}(k, \phi) = \int_0^{\phi} \frac{\mathrm{d}\theta}{\sqrt{1 - k^2 \sin^2 \theta}}, \quad \text{for } |k| \leq 1$$

k ⊠ `k`, ϕ ⊠ `phi` ⊠⊠⊠。

8.12　第 2 種不完全楕円積分（Incomplete elliptic integroal 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：

$$\mathsf{E}(k, \phi) = \int_0^{\phi} \sqrt{1 - k^2 \sin^2 \theta}\, \mathrm{d}\theta, \quad \text{for } |k| \leq 1$$

k ⊠ `k`, ϕ ⊠ `phi` ⊠⊠⊠。

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` ☒単位☒☒☒☒☒）☒対☒☒第 3 種不完全楕円積分（Incomplete elliptic integral of the third kind）☒計算☒☒。
戻☒値：

$$\Pi(\nu, k, \phi) = \int_0^{\phi} \frac{\mathrm{d}\theta}{(1-\nu\sin^2\theta)\sqrt{1-k^2\sin^2\theta}}, \quad \text{for } |k| \le 1$$

ν ☒ `nu`, k ☒ `k`, ϕ ☒ `phi` ☒☒☒。

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`, `k` ☒対☒☒第 1 種☒☒☒☒関数（Cylindrical Bessel functions of the first kind, Bessel functions of the first kind）☒計算☒☒。
戻☒値：

$$\mathsf{J}_{\nu}(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{\nu+2k}}{k!\,\Gamma(\nu+k+1)}, \quad \text{for } x \ge 0$$

ν ☒ `nu`, x ☒ `x` ☒☒☒。
注意： `nu` >= 128 ☒☒☒☒関数☒呼☒出☒☒効果☒実装定義☒☒☒。

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);
```

効果：実引数 `nu`, `x` ⊠対⊠⊠⊠⊠⊠⊠関数（Cylindrical Neumann functions, Neumann functions）、⊠⊠⊠名⊠第 2 種⊠⊠⊠⊠関数（Cylindrical Bessel functions of the second kind, Bessel functions of the second kind）⊠計算⊠⊠。
戻⊠値：

$$\mathsf{N}_\nu(x) = \begin{cases} \dfrac{\mathsf{J}_\nu(x)\cos\nu\pi - \mathsf{J}_{-\nu}(x)}{\sin\nu\pi}, & \text{for } x \geq 0 \text{ and non-integral } \nu \\[2ex] \lim\limits_{\mu\to\nu} \dfrac{\mathsf{J}_\mu(x)\cos\mu\pi - \mathsf{J}_{-\mu}(x)}{\sin\mu\pi}, & \text{for } x \geq 0 \text{ and integral } \nu \end{cases}$$

ν ⊠ `nu`, x ⊠ `x` ⊠⊠⊠。
注意： `nu` >= 128 ⊠⊠⊠⊠⊠関数⊠呼⊠出⊠⊠効果⊠実装定義⊠⊠⊠。
第 1 種⊠⊠⊠⊠関数⊠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⊠⊠⊠⊠関数（Regular modified cylindrical Bessel functions, Modified Bessel functions of the first kind）⊠計算⊠⊠。
戻⊠値：

$$\mathsf{I}_\nu(x) = \mathrm{i}^{-\nu}\mathsf{J}_\nu(\mathrm{i}x) = \sum_{k=0}^{\infty} \frac{(x/2)^{\nu+2k}}{k!\,\Gamma(\nu+k+1)}, \quad \text{for } x \geq 0$$

ν ⊠ `nu`, x ⊠ `x` ⊠⊠⊠。
注意： `nu` >= 128 ⊠⊠⊠⊠⊠関数⊠呼⊠出⊠⊠効果⊠実装定義⊠⊠⊠。
第 1 種⊠⊠⊠⊠関数⊠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⊠⊠⊠⊠関数（Irregular modified cylindrical Bessel functions, Modified Bessel functions of the second kind）⊠計算⊠⊠。

戻☒値：

$$
\begin{aligned}
\mathsf{K}_\nu(x) &= (\pi/2)\mathrm{i}^{\nu+1}(\mathsf{J}_\nu(\mathrm{i}x) + \mathrm{i}\mathsf{N}_\nu(\mathrm{i}x)) \\
&= \begin{cases} \dfrac{\pi}{2}\dfrac{\mathsf{I}_{-\nu}(x) - \mathsf{I}_\nu(x)}{\sin \nu\pi}, & \text{for } x \geq 0 \text{ and non-integral } \nu \\[2ex] \dfrac{\pi}{2}\lim\limits_{\mu\to\nu}\dfrac{\mathsf{I}_{-\mu}(x) - \mathsf{I}_\mu(x)}{\sin \mu\pi}, & \text{for } x \geq 0 \text{ and integral } \nu \end{cases}
\end{aligned}
$$

ν ☒ `nu`, x ☒ `x` ☒☒☒。

注意： `nu` >= 128 ☒☒☒☒関数☒呼☒出☒☒効果☒実装定義☒☒☒。
第 1 種変形☒☒☒☒関数、第 1 種☒☒☒☒関数、☒☒☒☒関数☒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` ☒対☒☒第 1 種球☒☒☒☒関数（Spherical Bessel functions of the first kind）☒計算☒☒。
戻☒値：

$$\mathsf{j}_n(x) = (\pi/2x)^{1/2}\mathsf{J}_{n+1/2}(x), \quad \text{for } x \geq 0$$

注意： `n` >= 128 ☒☒☒☒関数☒呼☒出☒☒効果☒実装定義☒☒☒。
第 1 種☒☒☒☒関数☒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` ☒対☒☒球☒☒☒☒関数（Spherical Neumann functions）、☒☒☒名☒第 2 種球☒☒☒☒関数（Spherical Bessel functions of the second kind）☒計算☒☒。
戻☒値：

$$\mathsf{n}_n(x) = (\pi/2x)^{1/2}\mathsf{N}_{n+1/2}(x), \quad \text{for } x \geq 0$$

n ☒ `n`, x ☒ `x` ☒☒☒。

注意： `n` >= 128 ⊠⊠⊠⊠関数⊠呼⊠出⊠⊠効果⊠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` ⊠対⊠⊠指数積分（Exponential integral）⊠計算⊠⊠。
戻⊠値：

$$\mathsf{Ei}(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t}\,\mathsf{d}t$$

x ⊠ `x` ⊠⊠⊠。

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` ⊠対⊠⊠⊠⊠⊠⊠⊠⊠⊠関数（Riemann zeta function）⊠計算⊠⊠。
戻⊠値：

$$\zeta(x) = \begin{cases} \displaystyle\sum_{k=1}^{\infty} k^{-x}, & \text{for } x > 1 \\ \displaystyle\frac{1}{1-2^{1-x}} \sum_{k=1}^{\infty} (-1)^{k-1} k^{-x}, & \text{for } 0 \le x \le 1 \\ 2^x \pi^{x-1} \sin(\frac{\pi x}{2})\,\Gamma(1-x)\,\zeta(1-x), & \text{for } x < 0 \end{cases}$$

x ⊠ `x` ⊠⊠⊠。

第 9 章

その他の標準ライブラリ

⊠⊠章⊠⊠ C++17 ⊠追加⊠⊠⊠細⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠干渉⊠⊠⊠⊠取得⊠⊠⊠⊠⊠⊠⊠⊠⊠入⊠⊠。⊠⊠⊠⊠⊠⊠⊠干渉⊠⊠⊠⊠⊠、俗⊠⊠⊠⊠⊠⊠⊠⊠⊠（cache line）⊠⊠呼⊠⊠⊠⊠⊠概念⊠。

残念⊠⊠⊠、⊠⊠ 2017 年⊠⊠、⊠⊠⊠⊠⊠極⊠⊠遅⊠。⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠高速⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠用意⊠⊠⊠⊠。⊠⊠⊠⊠⊠対⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠程度⊠⊠⊠⊠⊠⊠⊠⊠⊠数単位⊠行⊠⊠⊠。⊠⊠単位⊠何⊠⊠⊠⊠⊠⊠⊠⊠⊠実装依存⊠。C++17 ⊠⊠⊠⊠⊠⊠⊠⊠取得⊠⊠⊠⊠⊠⊠⊠⊠⊠入⊠⊠。

⊠⊠⊠⊠⊠⊠干渉⊠⊠⊠⊠知⊠⊠⊠理由⊠ 2 ⊠⊠⊠。2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠同一⊠局所性⊠持⊠⊠⊠⊠⊠⊠⊠載⊠⊠⊠⊠⊠場合⊠載⊠⊠⊠場合⊠。

2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、一方⊠頻繁⊠変更⊠、⊠⊠一方⊠⊠⊠⊠⊠変更⊠⊠⊠場合⊠、2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠同⊠局所性⊠持⊠⊠⊠⊠⊠⊠⊠載⊠⊠⊠⊠場合、⊠⊠変更⊠⊠⊠⊠⊠⊠⊠⊠⊠変更⊠⊠⊠⊠⊠、⊠⊠⊠⊠変更⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠同期⊠発生⊠⊠。

```
struct Data
{
    int counter ;
    int status ;
} ;
```

⊠⊠⊠、`counter` ⊠頻繁⊠変更⊠⊠⊠、`status` ⊠⊠⊠⊠⊠変更⊠⊠⊠場合、`counter` ⊠ `status` ⊠間⊠適切⊠⊠⊠⊠⊠⊠⊠挿入⊠⊠⊠⊠⊠、2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠同⊠局所性⊠持⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠場合⊠⊠、`std::hardware_destructive_interference_size` ⊠使⊠⊠。

```
struct Data
{
    int counter ;
    std::byte padding[
        std::hardware_destructive_interference_size - sizeof(int)
    ] ;
    int status ;
} ;
```

反対⊠、2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠同一⊠局所性⊠持⊠⊠⊠⊠⊠⊠⊠載⊠⊠⊠場合、`std::hardware_constructive_interference_size` ⊠使⊠⊠。

⊠⊠⊠⊠⊠⊠干渉⊠⊠⊠⊠ `<new>` ⊠⊠⊠⊠⊠以下⊠⊠⊠⊠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 ⊠⊠⊠、⊠⊠ `catch` ⊠⊠⊠⊠⊠⊠例外⊠⊠⊠場合⊠、`bool std::uncaught_exception()`⊠判定⊠⊠⊠⊠⊠⊠⊠⊠。

```
struct X
{
    ~X()
    {
        if ( std::uncaught_exception() )
        {
            // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠中⊠呼⊠⊠⊠
        }
        else
        {
            // 通常⊠破棄
        }
    }
} ;

int main()
{
    {
        X x ;
    }// 通常⊠破棄

    {
        X x ;
        throw 0 ;
    }// ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠中

}
```

`bool std::uncaught_exception()` ⊠、C++17 ⊠⊠非推奨扱⊠⊠⊠⊠⊠。⊠⊠⊠廃止⊠⊠⊠見込⊠⊠。

廃止⊠理由⊠⊠⊠⊠、単⊠以下⊠⊠⊠⊠例⊠役⊠立⊠⊠⊠⊠⊠⊠。

```
struct X
{
    ~X()
    {
        try {
            // true
            bool b = std::uncaught_exception() ;
        } catch( ... ) { }
    }
} ;
```

⊠⊠⊠⊠、`int std::uncaught_exceptions()` ⊠新⊠⊠追加⊠⊠⊠。⊠⊠関数⊠現在 `catch` ⊠⊠⊠⊠⊠⊠例外⊠個数⊠返⊠。

```
struct X
{
    ~X()
    {
        try {
            if ( int x = std::uncaught_exceptions() ; x > 1 )
            {
                // ⊠⊠⊠⊠⊠⊠例外
            }
        } 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 ⊠ tuple ⊠⊠⊠⊠⊠⊠要素⊠順番⊠実引数⊠渡⊠⊠関数⊠呼⊠出⊠⊠⊠⊠⊠関数⊠。

⊠⊠要素数 N ⊠ tuple t ⊠関数⊠⊠⊠⊠⊠⊠ f ⊠対⊠⊠、apply(f, t) ⊠、f(get<0>(t), get<1>(t), ... , get<N-1>(t)) ⊠⊠⊠⊠ f ⊠関数呼⊠出⊠⊠⊠。

例：

```
template < typename ... Types >
void f( Types ... args ) { }

int main()
{
    // int, int, int
    std::tuple t1( 1,2,3 ) ;

    // f( 1, 2, 3 ) ⊠関数呼⊠出⊠
    std::apply( f, t1 ) ;

    // int, double, const char *
    std::tuple t2( 123, 4.56, "hello" ) ;

    // f( 123, 4.56, "hello" ) ⊠関数呼⊠出⊠
    std::apply( f, t2 ) ;
}
```

9.4 Searcher：検索

C++17 ⊠⊠ `<functional>` ⊠ `searcher` ⊠⊠⊠⊠⊠⊠⊠⊠⊠追加⊠⊠⊠。⊠⊠⊠順序⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠集合⊠、⊠⊠部分集合（⊠⊠⊠⊠）⊠含⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠検索⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠最⊠一般的⊠応用例⊠文字列検索⊠⊠⊠。

`searcher` ⊠基本的⊠設計⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠構築⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠検索⊠⊠⊠部分集合（⊠⊠⊠⊠）⊠与⊠、`operator ()` ⊠部分集合⊠含⊠⊠⊠⊠⊠⊠⊠検索⊠⊠⊠集合⊠与⊠⊠。

⊠⊠設計⊠⊠⊠⊠⊠⊠⊠追加⊠⊠⊠理由⊠、⊠⊠⊠⊠⊠検索⊠⊠⊠⊠何⊠⊠⊠事前⊠準備⊠状態⊠⊠⊠保持⊠⊠⊠⊠⊠⊠検索⊠⊠⊠⊠⊠⊠⊠実装⊠⊠⊠⊠⊠。

9.4.1 default_searcher

⊠⊠⊠ `std::default_searcher` ⊠以下⊠⊠⊠⊠宣言⊠⊠⊠⊠⊠。

```
template <  class ForwardIterator1,
            class BinaryPredicate = equal_to<> >
class default_searcher {
public:
    // ⊠⊠⊠⊠⊠⊠⊠⊠
    default_searcher(
        ForwardIterator1 pat_first, ForwardIterator1 pat_last
        , BinaryPredicate pred = BinaryPredicate() ) ;

    // operator ()
    template <class ForwardIterator2>
    pair<ForwardIterator2, ForwardIterator2>
    operator()(ForwardIterator2 first, ForwardIterator2 last) const ;
} ;
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠部分集合⊠受⊠取⊠。`operator ()` ⊠集合⊠受⊠取⊠、部分集合（⊠⊠⊠⊠）⊠一致⊠⊠場所⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠返⊠。見⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `[last, last)` ⊠⊠⊠⊠⊠⊠。

以下⊠⊠⊠⊠使⊠。

```
int main()
{
    std::string pattern("fox") ;
    std::default_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 ) ;
  }
```

`default_searcher` ⊠検索⊠、内部的⊠ `std::search` ⊠使⊠⊠⊠。

9.4.2 boyer_moore_searcher

`std::boyer_moore_searcher` ⊠ Boyer–Moore 文字列検索⊠⊠⊠⊠⊠⊠⊠使⊠⊠部分集合⊠検索⊠行⊠。

Boyer–Moore 文字列検索⊠⊠⊠⊠⊠⊠⊠極⊠⊠効率的⊠文字列検索⊠⊠⊠⊠⊠⊠⊠⊠。Boyer–Moore ⊠⊠⊠⊠⊠⊠⊠ Bob Boyer ⊠ Strother Moore ⊠⊠⊠⊠発明⊠⊠、1977 年⊠ Communications of the ACM ⊠発表⊠⊠⊠。⊠⊠内容⊠以下⊠ URL ⊠読⊠⊠⊠⊠⊠⊠⊠。

http://www.cs.utexas.edu/~moore/publications/fstrpos.pdf

愚直⊠実装⊠⊠文字列検索⊠⊠⊠⊠⊠⊠⊠検索⊠⊠⊠部分文字列（⊠⊠⊠⊠）⊠検索対象⊠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⊠⊠。⊠⊠⊠⊠⊠⊠効率的⊠文字列検索⊠実現⊠⊠⊠⊠。

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⊠検索前⊠準備時間⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠、⊠⊠効率的⊠検索⊠⊠⊠相殺⊠⊠⊠。特⊠、⊠⊠⊠⊠⊠長⊠場合⊠効果的⊠。

C++17 ⊠入⊠ Boyer–Moore ⊠⊠⊠⊠⊠⊠⊠基⊠⊠検索⊠、⊠⊠⊠⊠⊠⊠⊠使⊠⊠汎用的⊠ `char` 型⊠⊠⊠⊠状態数⊠少⊠⊠型⊠対⊠⊠⊠実装⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠構造⊠使⊠⊠⊠⊠⊠⊠、任意⊠型⊠対応⊠⊠⊠汎用的⊠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:
    // ⊠⊠⊠⊠⊠⊠⊠⊠
    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 ⊠、文字列以外⊠⊠適用⊠⊠⊠汎用的⊠設計⊠⊠⊠、⊠⊠⊠⊠関数⊠取⊠。char 型⊠⊠⊠⊠取⊠⊠⊠状態数⊠少⊠⊠型以外⊠渡⊠⊠⊠場合⊠、std::unordered_map ⊠⊠⊠⊠⊠⊠⊠⊠使用量⊠削減⊠⊠⊠何⊠⊠⊠⊠⊠⊠構造⊠使⊠⊠⊠⊠⊠⊠⊠構築⊠⊠。

使⊠方⊠ default_searcher ⊠⊠⊠⊠⊠変⊠⊠⊠⊠。

```
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 ) ;
}
```

9.4.3 boyer_moore_horspool_searcher

`std::boyer_moore_horspool_searcher` ⊠ Boyer–Moore–Horspool 検索⊠⊠⊠⊠⊠⊠⊠使⊠⊠部分集合⊠検索⊠行⊠。Boyer–Moore–Horspool ⊠⊠⊠⊠⊠⊠⊠ Nigel Horspool ⊠⊠⊠⊠ 1980 年⊠発表⊠⊠⊠。

参考：“Practical fast searching in strings” 1980

Boyer–Moore–Horspool ⊠⊠⊠⊠⊠⊠⊠内部⊠⊠⊠⊠⊠使⊠⊠⊠⊠⊠使用量⊠削減⊠⊠⊠⊠⊠、最悪計算量⊠点⊠⊠⊠⊠⊠⊠ Boyer–Moore ⊠⊠⊠⊠⊠⊠⊠⊠劣⊠⊠⊠⊠。⊠⊠⊠、実行時間⊠増大⊠犠牲⊠⊠⊠⊠⊠⊠⊠使用量⊠削減⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠言⊠⊠。

⊠⊠⊠ `boyer_moore_horspool_searcher` ⊠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:
    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠
    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⊠方⊠ `boyer_moore_horspool_searcher` ⊠変⊠⊠⊠⊠。

```
int main()
{
    std::string pattern("fox") ;
    std::boyer_moore_horspool_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 ) ;
}
```

9.5 sample：乱択アルゴリズム

```
template <  class PopulationIterator, class SampleIterator,
            class Distance, class UniformRandomBitGenerator >

SampleIterator sample(
    PopulationIterator first, PopulationIterator last,
    SampleIterator out,
    Distance n, UniformRandomBitGenerator&& g) ;
```

C++17 ⊠ `<algorithm>` ⊠追加⊠⊠⊠ `std::sample` ⊠、標本⊠確率的⊠選択⊠⊠⊠⊠⊠乱択⊠⊠⊠⊠⊠⊠⊠。

`[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`。

⊠⊠要素⊠集合⊠⊠、`n` 個⊠要素⊠確率的⊠公平⊠選択⊠⊠⊠場合⊠使⊠⊠⊠⊠⊠⊠⊠。

9.5.1 乱択アルゴリズム

`std::sample` ⊠使⊠前⊠、⊠⊠正⊠⊠乱択⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠学⊠必要⊠⊠⊠。乱択⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠詳⊠⊠⊠、Donald E. Knuth ⊠ *The Art of Computer Programming*（以下 TAOCP、邦訳⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠同名⊠書名⊠出版⊠⊠⊠⊠⊠）⊠参照。

⊠⊠⊠⊠⊠⊠⊠入力、計測⊠⊠気象情報⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、世⊠中⊠⊠膨大⊠量⊠⊠⊠⊠⊠存在⊠⊠。⊠⊠⊠⊠膨大⊠⊠⊠⊠⊠⊠⊠⊠処理⊠⊠⊠⊠⊠⊠⊠、標本⊠採取⊠⊠⊠⊠⊠⊠⊠⊠、統計的⊠⊠⊠⊠⊠⊠信頼⊠⊠⊠確率⊠正⊠⊠全体⊠⊠⊠⊠⊠推定⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠ n 個⊠標本⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠方法⊠選択⊠⊠必要⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠方法⊠ n 個⊠標本⊠取⊠出⊠⊠⊠、集合⊠先頭⊠⊠ n 個⊠⊠、1 ⊠⊠⊠⊠ n 個⊠⊠⊠⊠方法⊠選択⊠⊠

⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠値⊠集合⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ n 個⊠標本⊠得⊠⊠⊠、集合⊠⊠⊠⊠⊠値⊠等⊠⊠確率⊠選⊠⊠⊠上⊠ n 個⊠選択⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

`std::sample` ⊠使⊠⊠、100 個⊠値⊠⊠ 10 個⊠標本⊠得⊠⊠⊠、以下⊠⊠⊠⊠書⊠⊠⊠⊠可能⊠。

```
int main()
{
    // 100 個⊠値⊠集合
    std::vector<int> pop(100) ;
    std::iota( std::begin(pop), std::end(pop), 0 ) ;

    // 標本⊠格納⊠⊠⊠⊠⊠⊠⊠
    std::vector<int> out(10) ;

    // 乱数生成器
    std::array<std::uint32_t, sizeof(std::knuth_b)/4> a ;
    std::random_device r ;
    std::generate( std::begin(a), std::end(a), [&]{ return r() ; } ) ;
    std::seed_seq seed( std::begin(a), std::end(a) ) ;
    std::knuth_b g( seed ) ;

    // 10 個⊠標本⊠得⊠
    sample( std::begin(pop), std::end(pop), std::begin(out), 10, g ) ;

    // 標本⊠出力
    std::copy(  std::begin(out), std::end(out),
                std::ostream_iterator<int>(std::cout, ", ") ) ;
}
```

集合⊠含⊠⊠⊠値⊠数⊠ N 個⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠値⊠⊠⊠⊠ n/m ⊠確率⊠選⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。100 個中 10 個⊠選⊠⊠⊠⊠⊠⊠、1/10 ⊠確率⊠⊠⊠⊠⊠⊠値⊠標本⊠⊠⊠選択⊠⊠⊠⊠⊠⊠⊠。

⊠⊠考⊠⊠基⊠⊠乱択⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠⊠⊠。

1. 集合⊠要素数⊠ N、選択⊠⊠⊠標本⊠数⊠ n, i ⊠ 0 ⊠⊠⊠。
2. 0 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ i 番目⊠値⊠ n/m ⊠確率⊠標本⊠⊠⊠選択⊠⊠。
3. i ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。
4. $i\ != N$ ⊠⊠⊠ goto 2。

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠書⊠⊠以下⊠⊠⊠⊠⊠⊠。

```
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 ⊠ true ⊠返⊠⊠⊠⊠⊠⊠分布
    double probability = double(n)/double(N) ;
    std::bernoulli_distribution d( probability ) ;

    // ⊠⊠⊠⊠⊠値⊠対⊠⊠
    std::for_each( first, last,
        [&]( auto && value )
        {
            if ( d(g) )
            {// n/N ⊠確率⊠標本⊠⊠⊠選択⊠⊠
                *out = value ;
                ++out ;
            }
        } ) ;

    return out ;
}
```

残念⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠正⊠⊠動⊠⊠⊠。⊠⊠例⊠⊠、100 個⊠値⊠集合⊠⊠ 10 個⊠標本⊠選択⊠⊠⊠。⊠⊠⊠、選⊠⊠⊠標本⊠数⊠⊠⊠⊠⊠⊠⊠実行⊠⊠⊠異⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠、標本⊠数⊠平均的⊠ 10 個選⊠⊠⊠⊠⊠⊠期待⊠⊠⊠⊠、運⊠悪⊠⊠ 0 個⊠ 100 個⊠標本⊠選⊠⊠⊠⊠⊠⊠可能性⊠⊠⊠。

⊠⊠⊠⊠、TAOCP Vol. 2 ⊠⊠⊠⊠、⊠⊠⊠⊠選⊠⊠⊠標本数⊠標準偏差⊠ $\sqrt{n(1-n/N)}$ ⊠⊠⊠。

正⊠⊠⊠⊠⊠⊠⊠⊠⊠、要素⊠集合⊠⊠⊠⊠ $(t+1)$ 番目⊠要素⊠、⊠⊠⊠ m 個⊠要素⊠標本⊠⊠⊠選⊠⊠⊠⊠⊠、$(n-m)(N-t)$ ⊠確率⊠選⊠⊠⊠⊠。

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 ⊠ TAOCP Vol. 2 ⊠⊠、⊠⊠⊠⊠⊠⊠ S ⊠称⊠⊠、要素数⊠⊠⊠⊠⊠⊠⊠集合⊠⊠標本⊠選択⊠⊠方法⊠解説⊠⊠⊠⊠。

⊠⊠⊠⊠⊠⊠ S ⊠以下⊠⊠⊠⊠。

$0 < n \le N$ ⊠⊠⊠、N 個⊠集合⊠⊠ n 個⊠標本⊠⊠⊠⊠⊠⊠選択⊠⊠。

1. t, m ⊠ 0 ⊠⊠⊠。t ⊠⊠⊠⊠⊠⊠処理⊠⊠要素数、m ⊠標本⊠⊠⊠選択⊠⊠要素数⊠⊠⊠。
2. $0 \le U \le N - t$ ⊠範囲⊠乱数 U ⊠生成⊠⊠。
3. $U \ge n - m$ ⊠⊠⊠⊠ goto 5。
4. 次⊠要素⊠標本⊠⊠⊠選択⊠⊠。m ⊠ t ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。$m < n$ ⊠⊠⊠⊠、goto 2。⊠⊠⊠⊠⊠⊠⊠標本⊠完了⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠終了⊠⊠。
5. 次⊠要素⊠標本⊠⊠⊠選択⊠⊠⊠。t ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。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.
```

```
            ++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

⊠⊠⊠⊠⊠⊠ S ⊠集合⊠要素数⊠ N 個⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合⊠、n 個⊠標本⊠選択⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠、⊠⊠ N ⊠⊠⊠⊠⊠⊠場合⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

現実⊠⊠ N ⊠⊠⊠⊠⊠⊠状況⊠⊠⊠⊠⊠。

- ⊠⊠⊠⊠⊠⊠⊠入力
- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠提供⊠⊠⊠⊠⊠全部読⊠込⊠⊠⊠⊠⊠要素数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠入力
- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠入力

⊠⊠⊠⊠⊠要素数⊠⊠⊠⊠⊠⊠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⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠、1 段階⊠要素⊠巡回⊠⊠⊠済⊠⊠⊠⊠。要素数⊠⊠⊠⊠⊠⊠⊠入力⊠処理⊠⊠、⊠⊠時点⊠公平⊠選択⊠⊠⊠標本⊠得⊠⊠。

⊠⊠⊠⊠⊠⊠ R ⊠⊠⊠⊠⊠⊠状況⊠使⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠⊠⊠ R ⊠⊠、要素数⊠⊠⊠⊠⊠⊠要素⊠集合⊠⊠ n 個⊠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 ⊠以下⊠⊠⊠⊠（⊠⊠⊠⊠⊠⊠⊠⊠⊠ Knuth 本⊠⊠違⊠)。

$n > 0$ ⊠⊠⊠、$size \geq n$ ⊠⊠⊠未確定⊠ $size$ 個⊠要素数⊠持⊠入力⊠⊠、n 個⊠標本⊠⊠⊠⊠⊠⊠選択⊠⊠。標本⊠候補⊠ n 個⊠⊠保管⊠⊠⊠。$1 \leq j \leq n$ ⊠⊠⊠ $I[j]$ ⊠保管⊠⊠⊠標本⊠指⊠。

1. 入力⊠⊠最初⊠ n 個⊠標本⊠⊠⊠選択⊠、保管⊠⊠。$1 \leq j \leq n$ ⊠範囲⊠ $I[j]$ ⊠ j 番目⊠標本⊠保管⊠⊠。t ⊠値⊠ n ⊠⊠⊠。$I[1], \ldots, I[n]$ ⊠現在⊠標本⊠指⊠。t ⊠現在処理⊠⊠入力⊠個数⊠指⊠。
2. 入力⊠終⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠終了⊠⊠。
3. t ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。$1 \leq M \leq t$ ⊠範囲⊠乱数 M ⊠生成⊠⊠。$M > n$ ⊠⊠⊠ goto 5。
4. 次⊠入力⊠ $I[M]$ ⊠保管⊠⊠。goto 2。
5. 次⊠入力⊠保管⊠⊠⊠。goto 2。

実装⊠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) ;
```

```
            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⊠⊠⊠⊠⊠⊠⊠場合⊠⊠⊠⊠⊠⊠⊠ S（選択標本）⊠、入力⊠要素数⊠⊠⊠⊠⊠⊠場合⊠⊠⊠⊠⊠⊠⊠ R（保管標本）⊠。

⊠⊠⊠、C++ ⊠追加⊠⊠⊠乱択⊠⊠⊠⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠⊠宣言⊠、⊠⊠⊠⊠説明⊠⊠⊠⊠⊠以下⊠ 1 ⊠⊠⊠⊠⊠。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` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠⊠⊠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⊠

PopulationIterator [first, last) ⊠⊠要素数⊠得⊠必要⊠⊠⊠、⊠⊠⊠⊠⊠⊠ PopulationIterator ⊠前方⊠⊠⊠⊠⊠⊠⊠満⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠場合、選択⊠⊠標本⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠出力⊠⊠⊠⊠⊠⊠⊠、出力先⊠ SampleIterator ⊠出力⊠⊠⊠⊠⊠⊠⊠満⊠⊠⊠⊠⊠⊠⊠。

⊠⊠入力元⊠ PopulationIterator ⊠入力⊠⊠⊠⊠⊠⊠⊠⊠満⊠⊠⊠⊠場合、⊠⊠場合⊠ PopulationIterator ⊠ [first, last) ⊠⊠要素数⊠得⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、要素数⊠⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠⊠⊠⊠⊠⊠⊠ R（保管標本）⊠選択⊠⊠⊠⊠得⊠⊠。⊠⊠場合、入力⊠処理⊠⊠⊠連⊠⊠、新⊠⊠選択⊠⊠標本⊠既存⊠標本⊠上書⊠⊠⊠⊠⊠、出力先⊠ SampleIterator ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠必要⊠⊠⊠。

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

    std::knuth_b g ;

    // PopulationIterator ⊠前方⊠⊠⊠⊠⊠⊠⊠満⊠⊠
    // SampleIterator ⊠出力⊠⊠⊠⊠⊠⊠⊠⊠⊠
    std::sample(    std::begin(input), std::end(input),
                    std::ostream_iterator<int>(std::cout), 100
                    g ) ;

    std::vector<int> sample(100) ;

    // PopulationIterator ⊠入力⊠⊠⊠⊠⊠⊠⊠⊠満⊠⊠⊠⊠
    // SampleIterator ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠必要
    std::sample(
        std::istream_iterator<int>(std::cin),
        std::istream_iterator<int>{},
        std::begin(sample), 100, g ) ;

}
```

注意⊠必要⊠⊠⊠⊠⊠⊠、C++ ⊠ sample ⊠入力元⊠ PopulationIterator ⊠前方⊠⊠⊠⊠⊠⊠以上⊠満⊠⊠場合⊠、必⊠⊠⊠⊠⊠⊠⊠ S（選択標本）⊠使⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠、要素数⊠得⊠⊠⊠⊠ std::distance(first, last) ⊠行⊠⊠⊠⊠⊠⊠⊠⊠⊠意味⊠⊠。⊠⊠⊠⊠処理⊠非効率的⊠⊠⊠⊠⊠⊠⊠⊠渡⊠⊠場合、必要以上⊠非効率的⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠以下⊠⊠⊠⊠⊠、

```
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⊠⊠⊠⊠意味⊠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⊠得⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠回⊠
    // 非効率的
    for( auto && e : input )
    { ++count ; }

    // 標本⊠選択⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠回⊠
    for ( auto && e : input )
    {/* 標本⊠選択 */}
}
```

std::list ⊠⊠⊠⊠⊠関数 size ⊠定数時間⊠⊠⊠⊠⊠⊠保証⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠回⊠⊠⊠⊠⊠1回⊠抑⊠⊠⊠⊠。⊠⊠⊠、std::sample ⊠要素数⊠渡⊠実引数⊠⊠⊠⊠⊠⊠要素数⊠⊠⊠⊠⊠⊠⊠⊠全走査⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合⊠⊠、非効率的⊠処理⊠行⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠未満、前方⊠⊠⊠⊠⊠⊠以上⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠範囲⊠⊠標本⊠選択⊠⊠⊠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 >
```

```
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 □□、`shared_ptr` □配列□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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ <utility> ⊠定義⊠⊠⊠⊠⊠。

```
template <class T> constexpr add_const_t<T>& as_const(T& t) noexcept
{
    return t ;
}
```

as_const ⊠引数⊠⊠⊠渡⊠⊠ lvalue ⊠⊠⊠⊠⊠⊠⊠ const ⊠ lvalue ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠関数⊠。const 性⊠付与⊠⊠手軽⊠⊠⊠⊠⊠関数⊠⊠⊠使⊠⊠⊠⊠⊠⊠⊠。

```
// 1
template < typename T >
void f( T & ) {}
// 2、⊠⊠⊠⊠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
    // 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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ <tuple> ⊠定義⊠⊠⊠⊠⊠。

```
template <class T, class Tuple>
constexpr T make_from_tuple(Tuple&& t);
```

apply ⊠ tuple ⊠要素⊠実引数⊠関数⊠呼⊠出⊠⊠⊠⊠⊠⊠⊠⊠、make_from_tuple ⊠ tuple ⊠要素⊠実引数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠呼⊠出⊠⊠⊠⊠⊠⊠⊠。

⊠⊠型 T ⊠要素数 N ⊠ tuple t ⊠対⊠⊠、make_from_tuple<T>(t) ⊠、T 型⊠ T(get<0>(t), get<1>(t), ... , get<N-1>(t)) ⊠⊠⊠⊠構築⊠⊠、構築⊠⊠ T 型⊠⊠⊠⊠⊠⊠⊠⊠返⊠。

```
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 ) ;
}
```

9.9 invoke：指定した関数を指定した実引数で呼び出す

invoke ⊠⊠⊠⊠⊠⊠⊠⊠⊠ <functional> ⊠定義⊠⊠⊠⊠⊠。

```
template <class F, class... Args>
invoke_result_t<F, Args...> invoke(F&& f, Args&&... args)
noexcept(is_nothrow_invocable_v<F, Args...>);
```

invoke(f, t1, t2, ... , tN) ⊠、関数 f ⊠ f(a1, a2, ... , aN) ⊠⊠⊠⊠呼⊠出⊠。

⊠⊠正確⊠⊠、C++ 標準規格⊠ INVOKE(f, t1, t2, ... , tN) ⊠同⊠規則⊠呼⊠出⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠規則⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠場合⊠与⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ reference_wrapper ⊠⊠⊠⊠⊠⊠異⊠⊠。⊠⊠詳細⊠⊠⊠⊠⊠解説⊠⊠⊠。

INVOKE ⊠ std::function ⊠ std::bind ⊠⊠使⊠⊠⊠⊠⊠規則⊠⊠⊠、標準⊠⊠⊠⊠⊠⊠同⊠挙動⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠覚⊠⊠⊠⊠⊠⊠⊠。

例：

```
void f( int ) { }

struct S
{
    void f( int ) ;
    int data ;
} ;

int main()
{
    // f( 1 )
    std::invoke( f, 1 ) ;

    S s ;

    // (s.*&S::f)(1)
    std::invoke( &S::f, s, 1 ) ;
    // ((*&s).*&S::f)(1)
    std::invoke( &S::f, &s, 1 ) ;
    // s.*&S::data
    std::invoke( &S::data, s ) ;
}
```

9.10 not_fn：戻り値の否定ラッパー

`not_fn` ⊠⊠⊠⊠⊠⊠⊠⊠⊠ `<functional>` ⊠定義⊠⊠⊠⊠⊠。

```
template <class F> unspecified not_fn(F&& f);
```

関数⊠⊠⊠⊠⊠⊠ `f` ⊠対⊠⊠ `not_fn(f)` ⊠呼⊠出⊠⊠、戻⊠値⊠⊠⊠何⊠⊠⊠関数⊠⊠⊠⊠⊠⊠⊠返⊠⊠⊠⊠。⊠⊠関数⊠⊠⊠⊠⊠⊠⊠呼⊠出⊠⊠、実引数⊠ `f` ⊠渡⊠⊠ `f` ⊠関数呼⊠出⊠⊠⊠、戻⊠値⊠ `operator !` ⊠否定⊠⊠返⊠。

```
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` ⊠代替品。

9.11 メモリー管理アルゴリズム

C++17 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `<memory>` ⊠⊠⊠⊠⊠管理用⊠⊠⊠⊠⊠⊠⊠⊠追加⊠⊠⊠。

9.11.1 addressof

```
template <class T> constexpr T* addressof(T& r) noexcept;
```

`addressof` ⊠ C++17 以前⊠⊠⊠⊠⊠。`addressof(r)` ⊠ `r` ⊠⊠⊠⊠⊠⊠⊠取得⊠⊠。⊠⊠⊠、`r` ⊠型⊠ `operator &`⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠正⊠⊠⊠⊠⊠⊠⊠⊠取

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` ☒☒ `n` 個☒範囲☒未初期化☒☒☒☒☒☒ `typename iterator_traits<ForwardIterator>::value_type` ☒☒☒☒☒☒初期化☒☒。2 ☒目☒☒☒☒☒☒☒☒ `first` ☒☒ `n` 個☒☒☒☒☒☒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() ) ;
    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⊠。⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠初期化⊠⊠⊠⊠値初期化⊠⊠。

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) ⊠範囲、⊠⊠⊠⊠ first ⊠⊠ n 個⊠範囲⊠値⊠、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() ) ;
    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⊠方⊠ `uninitialized_copy` ⊠同⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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)` ⊠範囲、⊠⊠⊠⊠ `first` ⊠⊠ n 個⊠範囲⊠未初期化⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠実引数 x ⊠与⊠⊠構築⊠⊠。

9.11.7 destroy

```
template <class T>
void destroy_at(T* location);
```

`location->~T()` ⊠呼⊠出⊠。

```
template <class ForwardIterator>
void destroy(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Size>
ForwardIterator destroy_n(ForwardIterator first, Size n);
```

`[first, last)` ⊠範囲、⊠⊠⊠⊠ `first` ⊠⊠ n 個⊠範囲⊠ `destroy_at` ⊠呼⊠出⊠。

9.12　shared_ptr::weak_type

C++17 ⊠⊠ `shared_ptr` ⊠ `weak_type` ⊠⊠⊠⊠⊠⊠⊠⊠⊠型名⊠追加⊠⊠⊠。⊠⊠⊠ `shared_ptr` ⊠対⊠⊠ `weak_ptr` ⊠ `typedef` 名⊠⊠⊠⊠⊠⊠。

```
namespace std {

template < typename T >
class shared_ptr
{
    using weak_type = weak_ptr<T> ;
} ;

}
```

使⊠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> ⊠定義⊠⊠⊠⊠⊠ void_t ⊠以下⊠⊠⊠⊠定義⊠⊠⊠⊠⊠。

```
namespace std {

template < class ... >
using void_t = void ;

}
```

void_t ⊠任意個⊠型⊠⊠⊠⊠⊠⊠⊠実引数⊠⊠⊠受⊠取⊠ void 型⊠。⊠⊠性質⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠便利⊠⊠⊠、標準⊠⊠⊠⊠⊠⊠追加⊠⊠⊠。

9.14 bool_constant

⊠⊠⊠⊠⊠⊠⊠⊠ <type_traits> ⊠ bool_constant ⊠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 ⊠使⊠⊠⊠⊠場面⊠特⊠ bool ⊠⊠⊠必要⊠場面⊠⊠、C++17 以降⊠単⊠ std::true_type ⊠ std::false_type ⊠書⊠⊠⊠⊠⊠⊠⊠⊠。

9.15 type_traits

C++17 ⊠⊠ <type_traits> ⊠機能追加⊠行⊠⊠⊠。

9.15.1 変数テンプレート版 traits

C++17 ⊠⊠、既存⊠ traits ⊠変数⊠⊠⊠⊠⊠⊠⊠利用⊠⊠ _v 版⊠追加⊠⊠⊠。

⊠⊠⊠⊠、is_integral<T>::value ⊠書⊠代⊠⊠⊠ is_integral_v<T> ⊠書⊠⊠⊠⊠⊠⊠⊠。

```
template < typename T >
void f( T x )
{
    constexpr bool b1 = std::is_integral<T>::value ; // ⊠⊠⊠⊠⊠⊠⊠
    constexpr bool b2 = std::is_integral_v<T> ; // 変数⊠⊠⊠⊠⊠⊠
    constexpr bool b3 = std::is_integral<T>{} ; // operator bool()
}
```

9.15.2　論理演算 traits

C++17 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `conjunction`, `disjunction`, `negation` ⊠追加⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠論理積、論理和、否定⊠手軽⊠扱⊠⊠⊠⊠ `traits` ⊠。

conjunction：論理積

```
template<class... B> struct conjunction;
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠ `conjunction<B1, B2, ..., BN>` ⊠⊠⊠⊠⊠⊠⊠実引数 `B1`, `B2`, `...`, `BN` ⊠論理積⊠適用⊠⊠。`conjunction` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実引数 `Bi` ⊠対⊠⊠、`bool(Bi::value)` ⊠ `false` ⊠⊠⊠最初⊠型⊠基本⊠⊠⊠⊠持⊠⊠、⊠⊠⊠⊠最後⊠ `BN` ⊠基本⊠⊠⊠⊠持⊠。

```
int main()
{
    using namespace std ;

    // is_void<void>⊠基本⊠⊠⊠⊠持⊠
    using t1 =
        conjunction<
            is_same<int, int>, is_integral<int>,
            is_void<void> > ;

    // is_integral<double>⊠基本⊠⊠⊠⊠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 ⊠論理和⊠適用⊠⊠。disjunction ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実引数 Bi ⊠対⊠⊠、bool(Bi::value) ⊠ true ⊠⊠⊠最初⊠型⊠基本⊠⊠⊠⊠持⊠⊠、⊠⊠⊠⊠最後⊠ BN ⊠基本⊠⊠⊠⊠持⊠。

```
int main()
{
    using namespace std ;

    // is_same<int,int>⊠基本⊠⊠⊠⊠持⊠
    using t1 =
        disjunction<
            is_same<int, int>, is_integral<int>,
            is_void<void> > ;

    // is_void<int>⊠基本⊠⊠⊠⊠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 ⊠否定⊠適用⊠⊠。negation ⊠基本⊠⊠⊠⊠⊠⊠ bool_constant<!bool(B::value)> ⊠持⊠。

```
int main()
{
    using namespace std ;

    // false
    constexpr bool b1 = negation< true_type >::value ;
    // true
    constexpr bool b2 = negation< false_type >::value ;
}
```

9.15.3　is_invocable：呼び出し可能か確認する 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 ⌧⌧⌧⌧⌧⌧⌧実引数⌧与⌧⌧⌧⌧型 Fn ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧ ArgTypes ⌧⌧⌧⌧展開⌧⌧結果⌧実引数⌧関数呼⌧出⌧⌧⌧⌧⌧⌧⌧⌧、⌧⌧⌧⌧⌧戻⌧値⌧ R ⌧暗黙変換⌧⌧⌧⌧⌧⌧⌧⌧確認⌧⌧ traits ⌧。呼⌧出⌧⌧⌧⌧⌧⌧⌧ true_type, ⌧⌧⌧⌧⌧⌧⌧ false_type ⌧基本⌧⌧⌧⌧持⌧。

is_invocable ⌧関数呼⌧出⌧⌧⌧結果⌧戻⌧値⌧型⌧⌧⌧⌧⌧問⌧⌧⌧。

is_invocable_r ⌧呼⌧出⌧可能性⌧加⌧⌧、関数呼⌧出⌧⌧⌧結果⌧戻⌧値⌧型⌧ R ⌧暗黙変換⌧⌧⌧⌧⌧⌧確認⌧⌧⌧。

is_nothrow_invocable ⌧ is_nothrow_invocable_r ⌧、関数呼⌧出⌧（⌧⌧⌧戻⌧値型 R ⌧⌧暗黙変換）⌧無例外保証⌧⌧⌧⌧⌧⌧⌧⌧確認⌧⌧。

```
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>` ⊠、`T` 型⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠可能⊠、⊠⊠ `T` 型⊠同値⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠⊠⊠⊠内部表現⊠同⊠場合⊠、`true` ⊠返⊠。

`false` ⊠返⊠例⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠（padding）⊠呼⊠⊠⊠⊠⊠⊠⊠⊠⊠調整⊠⊠⊠⊠⊠⊠値⊠表現⊠影響⊠⊠⊠⊠⊠⊠⊠⊠領域⊠持⊠場合⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠値⊠同値⊠影響⊠⊠⊠⊠⊠、`false` ⊠返⊠。

⊠⊠⊠⊠以下⊠⊠⊠⊠⊠⊠⊠ `X` ⊠、

```
struct X
{
    std::uint8_t a ;
    std::uint32_t b ;
} ;
```

⊠⊠実装⊠⊠⊠⊠⊠、4 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠必要⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠本当⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
struct X
{
    std::uint8_t a ;

    std::byte unused_padding[3] ;

    std::uint32_t b ;
} ;
```

⊠⊠場合、`unused_padding` ⊠値⊠⊠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 可能か確認する 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 コンテナーで不完全型のサポート

注意：⊠⊠説明⊠上級者向⊠⊠。

C++17 ⊠⊠以下⊠⊠⊠⊠⊠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⊠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⊠⊠⊠。

⊠⊠⊠⊠、C++14 ⊠⊠以下⊠⊠⊠⊠書⊠⊠⊠⊠⊠⊠⊠⊠、

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

    v.emplace_back(0) ; // void
    int value = v.back() ;
}
```

以下⊠⊠⊠⊠書⊠⊠⊠⊠⊠⊠⊠⊠。

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

    int value = v.emplace_back(0) ;
}
```

9.18 map と unordered_map の変更

map ⊠ `unordered_map` ⊠、`try_emplace` ⊠ `insert_or_assign` ⊠⊠⊠ 2 ⊠⊠⊠⊠⊠⊠関数⊠入⊠⊠。⊠⊠⊠⊠⊠⊠関数⊠ `multi_map` ⊠ `unordered_multi_map` ⊠⊠追加⊠⊠⊠⊠⊠⊠。

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⊠ `emplace` ⊠、⊠⊠⊠対応⊠⊠要素⊠存在⊠⊠⊠場合、要素⊠ `args` ⊠⊠ `emplace` 構築⊠⊠⊠追加⊠⊠⊠。⊠⊠、⊠⊠⊠対応⊠⊠要素⊠存在⊠⊠場合、要素⊠追加⊠⊠⊠⊠。要素⊠追加⊠⊠⊠⊠⊠⊠、`args` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実装定義

⊠⊠⊠。

```
int main()
{
    std::map< int, std::unique_ptr<int> > m ;

    // ⊠⊠⊠要素⊠存在⊠⊠
    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⊠⊠⊠異⊠⊠
    // ptr ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
    bool b = ( ptr != nullptr ) ;
}
```

⊠⊠場合、実際⊠ map ⊠要素⊠追加⊠⊠⊠⊠⊠⊠⊠⊠、ptr ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠、C++17 ⊠⊠、要素⊠追加⊠⊠⊠⊠⊠⊠場合 args ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠保証⊠⊠⊠ try_emplace ⊠追加⊠⊠⊠。

```
int main()
{
    std::map< int, std::unique_ptr<int> > m ;

    // ⊠⊠⊠要素⊠存在⊠⊠
    m[0] = nullptr ;

    auto ptr = std::make_unique<int>(0) ;
    // emplace ⊠失敗⊠⊠
    auto [iter, is_emplaced] = m.emplace( 0, std::move(ptr) ) ;

    // true ⊠⊠⊠⊠⊠⊠保証⊠⊠⊠
    // ptr ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
    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 ☒ key ☒連想☒☒☒要素☒存在☒☒場合☒要素☒代入☒、存在☒☒☒場合☒要素☒追加☒☒。operator [] ☒☒違☒☒、要素☒代入☒☒☒☒追加☒☒☒☒☒、戻☒値☒ pair ☒ bool ☒☒☒☒☒☒☒☒☒☒。

```
int main()
{
    std::map< int, int > m ;
    m[0] = 0 ;

    {
        // 代入
        // is_inserted ☒ false
        auto [iter, is_inserted] = m.insert_or_assign( 0, 1 ) ;
    }

    {
        // 追加
        // is_inserted ☒ true
        auto [iter, is_inserted] = m.insert_or_assign( 1, 1 ) ;
    }
}
```

9.19　連想コンテナーへの splice 操作

C++17 ☒☒、連想☒☒☒☒☒☒非順序連想☒☒☒☒☒☒ splice 操作☒☒☒☒☒☒☒☒。

対象☒☒☒☒☒☒☒ map, set, multimap, multiset, unordered_map, unordered_set, unordered_multimap, unordered_multiset ☒。

splice 操作☒☒ list ☒提供☒☒☒☒☒操作☒、☒☒☒☒☒☒互換☒ list ☒☒☒

⊠⊠⊠⊠⊠要素⊠⊠⊠⊠⊠⊠⊠所有権⊠⊠別⊠⊠⊠⊠⊠⊠⊠⊠移動⊠⊠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⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠仕組⊠⊠用⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠要素⊠所有権⊠⊠⊠⊠⊠⊠⊠外⊠出⊠仕組⊠⊠、`splice` 操作⊠行⊠。

9.19.1 merge

⊠⊠⊠⊠連想⊠⊠⊠⊠⊠⊠非順序連想⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠関数 `merge` ⊠持⊠⊠⊠⊠。⊠⊠⊠⊠⊠ a, b ⊠⊠⊠⊠⊠⊠⊠互換⊠⊠⊠、`a.merge(b)` ⊠、⊠⊠⊠⊠⊠ b ⊠要素⊠所有権⊠⊠⊠⊠⊠⊠⊠⊠⊠ a ⊠移⊠。

```
int main()
{
    std::set<int> a = {1,2,3} ;
    std::set<int> b = {4,5,6} ;

    // b ⊠要素⊠⊠⊠⊠ a ⊠移⊠
    a.merge(b) ;

    // a ⊠{1,2,3,4,5,6}
    // b ⊠{}
}
```

⊠⊠、⊠⊠⊠重複⊠許⊠⊠⊠⊠⊠⊠⊠⊠⊠場合⊠、値⊠重複⊠⊠場合、重複⊠⊠要素⊠移動⊠⊠⊠。

```
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 ⊠⊠⊠⊠移動⊠⊠⊠要素⊠指⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、要素⊠移動後⊠妥当⊠⊠⊠。⊠⊠⊠、所属⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠変⊠⊠。

```
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

⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠要素⊠構築⊠⊠⊠⊠⊠⊠⊠⊠所有権⊠切⊠離⊠機能⊠。

⊠⊠⊠⊠⊠⊠⊠⊠型⊠、各⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠型名 node_type ⊠⊠⊠。⊠⊠⊠⊠ std::set<int> ⊠⊠⊠⊠⊠⊠⊠⊠型⊠、std::set<int>::node_type ⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠⊠⊠⊠⊠⊠持⊠⊠⊠⊠。

```
class node_handle
{
public :
    // ⊠⊠⊠⊠⊠⊠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

    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
    // ⊠⊠⊠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

    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
    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()
{
    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}}

}
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠切⊠離⊠、所有権⊠得⊠。⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠得⊠⊠⊠⊠⊠、元⊠⊠⊠⊠⊠⊠⊠⊠独立⊠、元⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠破棄⊠際⊠⊠破棄⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠破棄時⊠破棄⊠⊠⊠。⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠持⊠。

```
int main()
{
    std::set<int>::node_type n ;

    {
        std::set<int> c = { 1,2,3 } ;
        // 所有権⊠移動
        n = c.extract( std::begin(c) ) ;
        // c ⊠破棄⊠⊠⊠
    }

    // OK
    // ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠所有権⊠移動⊠⊠⊠⊠
    int x = n.value() ;

    // n ⊠破棄⊠⊠⊠
}
```

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 ) ;
```

連想⊠⊠⊠⊠⊠⊠非順序連想⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠関数 extract ⊠、⊠⊠⊠⊠⊠⊠⊠⊠取得⊠⊠⊠⊠⊠⊠⊠⊠⊠関数⊠。

⊠⊠⊠⊠関数 `extract(position)` ⊠、⊠⊠⊠⊠⊠⊠⊠ position ⊠指⊠要素⊠、⊠⊠⊠⊠⊠⊠⊠除去⊠⊠、⊠⊠要素⊠所有⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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}

}
```

⊠⊠⊠⊠関数 `extract(x)` ⊠、⊠⊠ x ⊠⊠⊠⊠⊠⊠⊠存在⊠⊠場合、⊠⊠要素⊠⊠⊠⊠⊠⊠⊠⊠除去⊠⊠、⊠⊠要素⊠所有⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠返⊠。存在⊠⊠⊠場合、空⊠⊠⊠⊠⊠⊠⊠⊠⊠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、複数⊠⊠⊠⊠⊠ 1 ⊠⊠所有権⊠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⌧⌧⌧⌧⌧⌧⌧⌧⌧場合
insert_return_type  insert(node_type&& nh);
// ⌧⌧⌧重複⌧許⌧multi ⌧⌧⌧⌧⌧⌧場合
iterator  insert(node_type&& nh);

// ⌧⌧⌧付⌧⌧ insert
iterator            insert(const_iterator hint, node_type&& nh);
```

⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧関数 insert ⌧実引数⌧渡⌧⌧、⌧⌧⌧⌧⌧⌧⌧⌧⌧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⌧場合、何⌧起⌧⌧⌧⌧。

```
int main()
{
    std::set<int> c ;
    std::set<int>::node_type n ;

    // 何⌧起⌧⌧⌧⌧
    c.insert( std::move(n) ) ;
}
```

⊠⊠⊠重複⊠許⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠存在⊠⊠⊠⊠⊠等⊠⊠値⊠所有⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `insert` ⊠⊠⊠⊠⊠⊠⊠、`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` ⊠受⊠取⊠ `insert` ⊠挙動⊠、従来⊠ `insert` ⊠同⊠⊠。要素⊠ `hint` ⊠直前⊠追加⊠⊠⊠⊠⊠⊠⊠⊠償却定数時間⊠処理⊠終⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠実引数⊠受⊠取⊠ `insert` ⊠戻⊠値⊠型⊠、⊠⊠⊠重複⊠許⊠ `multi` ⊠⊠⊠⊠⊠⊠場合 `iterator`。⊠⊠⊠重複⊠許⊠⊠⊠⊠⊠⊠⊠⊠⊠場合、`insert_return_type` ⊠⊠⊠。

`multi` ⊠⊠⊠⊠⊠⊠場合、戻⊠値⊠追加⊠⊠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⊠⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠型名 `insert_return_type` ⊠戻⊠値⊠型⊠⊠⊠。⊠⊠⊠⊠ `set<int>`⊠場合、`set<int>::insert_return_type` ⊠⊠⊠。

`insert_return_type` ⊠具体的⊠名前⊠規格上規定⊠⊠⊠⊠⊠⊠。`insert_return_type`⊠以下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠持⊠型⊠⊠⊠⊠⊠⊠。

```
struct insert_return_type
{
    iterator position ;
    bool inserted ;
    node_type node ;
} ;
```

position ⊠ insert ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠所有権⊠移動⊠⊠追加⊠⊠⊠要素⊠指⊠⊠⊠⊠⊠⊠⊠、inserted ⊠要素⊠追加⊠行⊠⊠⊠場合⊠ true ⊠⊠⊠ bool, node ⊠要素⊠追加⊠失敗⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠所有権⊠移動⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

insert ⊠渡⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠空⊠⊠⊠、inserted ⊠ false, position ⊠ end(), node ⊠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 ⊠成功⊠⊠⊠⊠、inserted ⊠ true, position ⊠追加⊠⊠⊠要素⊠指⊠、node ⊠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 ⊠失敗⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠同一⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠存在⊠⊠⊠⊠、inserted ⊠ false, node ⊠ insert ⊠呼⊠出⊠前⊠⊠⊠⊠⊠⊠⊠⊠⊠値、position ⊠

⊠⊠⊠⊠⊠⊠中⊠追加⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠等⊠⊠要素⊠指⊠。`insert` ⊠渡⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠未規定⊠値⊠⊠⊠。

```
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⊠⊠⊠場合⊠ n ⊠値⊠⊠⊠⊠規定⊠⊠⊠⊠⊠⊠、最⊠⊠⊠⊠⊠実装⊠⊠⊠⊠、n ⊠ node ⊠⊠⊠⊠⊠⊠⊠⊠⊠、n ⊠空⊠⊠⊠、⊠⊠⊠後⊠状態⊠⊠⊠。

9.19.5　ノードハンドルの利用例

⊠⊠⊠⊠⊠⊠⊠⊠典型的⊠使⊠方⊠以下⊠⊠⊠⊠。

⊠⊠⊠⊠⊠⊠再確保⊠⊠⊠、⊠⊠⊠⊠⊠⊠一部⊠要素⊠⊠別⊠⊠⊠⊠⊠⊠⊠移⊠

```
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⊠⊠必要⊠⊠⊠。⊠⊠⊠⊠動的⊠⊠⊠⊠⊠⊠⊠解放⊠確保⊠必要⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠、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 コンテナーアクセス関数

⊠⊠⊠⊠⊠⊠⊠⊠ `<iterator>` ⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠関数⊠⊠⊠、⊠⊠⊠関数版⊠ size, empty, data ⊠追加⊠⊠⊠。⊠⊠⊠⊠、⊠⊠⊠⊠関数⊠ size, empty, data ⊠

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⊠⊠。

```
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⊠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 ;
    {
        // a,b,c,d ⊠ lock ⊠⊠
        std::scoped_lock l( a, b, c, d ) ;
        // a,b,c,d ⊠ unlock ⊠⊠
    }
}
```

std::scoped_lock ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠複数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠取⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠起⊠⊠⊠⊠方法⊠⊠⊠⊠⊠関数 lock ⊠呼⊠出⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠関数 unlock ⊠呼⊠出⊠。

9.25 std::byte

C++17 ⊠⊠⊠⊠⊠⊠表現⊠⊠型⊠⊠⊠ std::byte ⊠⊠⊠⊠⊠⊠⊠追加⊠⊠⊠。⊠⊠⊠、⊠⊠言語⊠一部⊠⊠⊠、別項⊠詳⊠⊠解説⊠行⊠⊠⊠⊠。

9.26 最大公約数（gcd）と最小公倍数（lcm）

C++17 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ <numeric> ⊠最大公約数（gcd）⊠最小公倍数（lcm）⊠追加⊠⊠⊠。

```
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)
{
    if ( n == 0 )
        return m ;
    else
        return gcd( n, std::abs(m) % std::abs(n) ) ;
}
```

gcd(m, n) ⊠ m ⊠ n ⊠⊠⊠⊠⊠⊠⊠場合⊠⊠⊠返⊠。⊠⊠以外⊠場合、$|m|$ ⊠ $|n|$ ⊠最大公約数（Greatest Common Divisor）⊠返⊠。

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) ⊠、m ⊠ n ⊠⊠⊠⊠⊠⊠⊠⊠⊠場合⊠⊠⊠返⊠。⊠⊠以外⊠場合、$|m|$ ⊠ $|n|$ ⊠最小公倍数（Least Common Multiple）⊠返⊠。

第 10 章

ファイルシステム

⊠⊠⊠⊠⊠⊠⊠⊠ `<filesystem>` ⊠定義⊠⊠⊠⊠⊠標準⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠属性⊠扱⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

一般⊠「⊠⊠⊠⊠⊠⊠⊠⊠」⊠⊠⊠⊠場合、⊠⊠⊠⊠ Linux ⊠ ext4, Microsoft Windows ⊠ FAT ⊠ NTFS, Apple Mac ⊠ HFS+ ⊠ APFS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠属性⊠表現⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠上⊠⊠⊠⊠構造⊠意味⊠⊠。C++ ⊠標準⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実現⊠⊠⊠⊠⊠構造⊠操作⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠構造⊠抽象化⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠属性、⊠⊠⊠付随⊠⊠要素、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠操作⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、「⊠⊠⊠⊠」⊠⊠⊠用語⊠単⊠通常⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠、FIFO（名前付⊠⊠⊠⊠）、⊠⊠⊠⊠⊠⊠⊠特殊⊠⊠⊠⊠⊠⊠含⊠。

本書⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠詳細⊠解説⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠量⊠膨大⊠⊠⊠、特定⊠関数⊠意味⊠⊠⊠⊠⊠、C++ ⊠⊠⊠⊠⊠⊠⊠付属⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠参照⊠⊠⊠⊠⊠。

10.1　名前空間

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `std::filesystem` 名前空間⊠⊠⊠⊠⊠下⊠宣言⊠⊠⊠⊠⊠。

```
int main()
{
    std::filesystem::path p("/bin") ;
}
```

⊠⊠名前空間⊠長⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠⊠⊠、関数⊠⊠⊠⊠⊠⊠⊠⊠⊠単位⊠ using ⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠、名前空間⊠⊠⊠⊠⊠⊠使⊠⊠短⊠別名⊠付⊠⊠⊠⊠⊠。

```
void using_directive()
{
    // using ⊠⊠⊠⊠⊠⊠⊠
    using namespace std::filesystem ;

    path p("/etc") ;
}

void namespace_alias()
{
    // 名前空間⊠⊠⊠⊠⊠
    namespace fs = std::filesystem ;

    fs::path p("/usr") ;
}
```

10.2　POSIX 準拠

C++ ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠操作⊠挙動⊠、POSIX 規格⊠従⊠。実装⊠⊠⊠⊠⊠ POSIX ⊠規定⊠⊠⊠挙動⊠提供⊠⊠⊠⊠場合⊠⊠⊠。⊠⊠場合⊠制限⊠範囲内⊠、⊠⊠⊠⊠⊠ POSIX ⊠近⊠挙動⊠行⊠。実装⊠⊠⊠⊠⊠⊠意味⊠⊠⊠挙動⊠提供⊠⊠⊠⊠場合、⊠⊠⊠⊠通知⊠⊠⊠。

10.3　ファイルシステムの全体像

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠全体像⊠簡単⊠箇条書⊠⊠⊠⊠以下⊠⊠⊠⊠。

- ⊠⊠⊠ `path` ⊠⊠⊠⊠⊠⊠⊠文字列⊠扱⊠
- 例外⊠⊠⊠ `filesystem_error` ⊠⊠⊠⊠ `error_code` ⊠⊠⊠⊠通知
- ⊠⊠⊠ `file_status` ⊠⊠⊠⊠⊠⊠情報⊠⊠⊠⊠⊠⊠⊠⊠⊠取得、設定
- ⊠⊠⊠ `directory_entry` ⊠⊠⊠⊠⊠⊠⊠⊠⊠情報⊠取得、設定
- ⊠⊠⊠ `directory_iterator` ⊠⊠⊠⊠⊠⊠⊠⊠構造⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
- 多数⊠⊠⊠⊠関数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠操作

10.4 エラー処理

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠発生⊠⊠場合、⊠⊠⊠⊠通知方法⊠⊠ 2 種類⊠方法⊠⊠⊠。例外⊠使⊠方法⊠、⊠⊠⊠⊠⊠⊠⊠⊠ `<system_error>` ⊠定義⊠⊠⊠⊠⊠⊠⊠⊠通知用⊠⊠⊠⊠ `std::error_code` ⊠⊠⊠⊠⊠⊠⊠⊠⊠実引数⊠⊠⊠渡⊠⊠⊠⊠⊠内容⊠受⊠取⊠方法⊠。

⊠⊠⊠処理⊠方法⊠、⊠⊠⊠⊠起⊠⊠期待度⊠⊠⊠⊠選択⊠⊠⊠。一般⊠、⊠⊠⊠⊠⊠⊠⊠⊠起⊠⊠⊠⊠場合、⊠⊠⊠⊠起⊠⊠⊠⊠予期⊠⊠⊠⊠⊠場合、⊠⊠⊠処理⊠⊠例外⊠使⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠頻繁⊠起⊠⊠場合、⊠⊠⊠⊠起⊠⊠⊠⊠⊠予期⊠⊠⊠場合、⊠⊠⊠処理⊠⊠例外⊠使⊠⊠⊠⊠⊠⊠⊠⊠。

10.4.1 例外

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠関数⊠⊠⊠、`std::error_code &` 型⊠実引数⊠取⊠⊠⊠関数⊠、以下⊠⊠⊠⊠⊠⊠⊠通知⊠行⊠。

- OS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠操作⊠⊠⊠⊠⊠⊠⊠⊠発生⊠⊠場合、`std::filesystem::filesystem_error` 型⊠例外⊠ `throw` ⊠⊠⊠。1 ⊠⊠ `path` ⊠実引数⊠取⊠関数⊠場合、`filesystem_error` ⊠⊠⊠⊠⊠関数 `path1` ⊠実引数⊠ `path` ⊠得⊠⊠⊠。2 ⊠⊠ `path` ⊠実引数⊠取⊠関数⊠場合、`filesystem_error` ⊠⊠⊠⊠⊠関数 `path1`, `path2` ⊠第一、第二引数⊠⊠⊠⊠⊠得⊠⊠⊠。`filesystem_error` ⊠⊠⊠⊠内容⊠応⊠⊠ `error_code` ⊠持⊠
- ⊠⊠⊠⊠⊠⊠確保⊠失敗⊠⊠場合、既存⊠例外⊠⊠⊠通知⊠行⊠⊠⊠
- ⊠⊠⊠⊠⊠⊠⊠⊠例外⊠投⊠⊠⊠

例外⊠使⊠⊠⊠⊠⊠処理⊠以下⊠⊠⊠⊠。

```
int main()
{
    using namespace std::filesystem ;

    try {
        // ⊠⊠⊠⊠名⊠⊠同⊠⊠⊠⊠⊠名⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
        path file("foobar.txt") ;
        std::ofstream{ file } ;
        copy_file( file, file ) ;

    } catch( filesystem_error & e )
    { // ⊠⊠⊠⊠場合
```

```
        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⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
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

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠関数⊠⊠⊠、std::error_code & 型⊠実引数⊠取⊠関数⊠、以下⊠⊠⊠⊠⊠⊠⊠通知⊠行⊠。

- OS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠操作⊠⊠⊠⊠⊠⊠⊠⊠発生⊠⊠場合、error_code & 型⊠実引数⊠⊠⊠⊠内容⊠応⊠⊠設定⊠⊠⊠。⊠⊠⊠⊠⊠⊠場合、error_code & 型⊠実引数⊠対⊠⊠⊠⊠⊠⊠関数 clear() ⊠呼⊠⊠⊠。

```
int main()
{
    using namespace std::filesystem ;

    // ⊠⊠⊠⊠名⊠⊠同⊠⊠⊠⊠⊠名⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
    path file("foobar.txt") ;
```

```
        std::ofstream{ file } ;
        std::error_code error_code;
        copy_file( file, file, error_code ) ;

        if ( error_code )
        { // ⌧⌧⌧⌧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` ⌧⌧⌧⌧⌧⌧⌧⌧文字列⌧表現⌧⌧⌧⌧⌧⌧⌧⌧⌧。文字列⌧表現⌧⌧⌧⌧⌧⌧⌧⌧ C++ ⌧⌧⌧⌧⌧ `std::string` ⌧⌧⌧⌧、⌧⌧⌧⌧⌧⌧⌧⌧⌧文字列⌧表現⌧⌧⌧⌧⌧、別⌧専用⌧⌧⌧⌧作⌧⌧⌧。

⌧⌧⌧ `path` ⌧以下⌧機能⌧提供⌧⌧。

- ⌧⌧⌧⌧⌧⌧文字列⌧表現
- ⌧⌧⌧⌧⌧⌧文字列⌧操作

`path` ⌧⌧⌧⌧⌧⌧⌧文字列⌧表現⌧操作⌧⌧⌧提供⌧⌧⌧⌧⌧⌧⌧、物理⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧変更⌧⌧⌧⌧⌧⌧⌧⌧⌧。

⌧⌧⌧⌧⌧⌧文字列⌧⌧⌧⌧⌧⌧表現⌧⌧⌧⌧⌧⌧⌧実装⌧⌧⌧異⌧⌧。POSIX 環境⌧⌧文字型⌧ `char` 型⌧⌧⌧ UTF-8 ⌧⌧⌧⌧⌧⌧表現⌧⌧ OS ⌧多⌧⌧、Microsoft Windows ⌧⌧本書執筆現在、文字型⌧ `wchar_t` ⌧⌧⌧ UTF-16 ⌧⌧⌧⌧⌧⌧表現⌧⌧慣習⌧⌧⌧⌧⌧⌧。

⌧⌧、OS ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧大文字小文字⌧区別⌧⌧⌧⌧⌧⌧、区別⌧⌧⌧⌧無視⌧⌧⌧⌧⌧⌧実装⌧⌧⌧。

⌧⌧⌧ `path` ⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧文字列⌧差異⌧吸収⌧⌧⌧⌧⌧。

⌧⌧⌧ `path` ⌧⌧以下⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧型名⌧⌧⌧。

```
namespace std::filesystem {
    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` ⊠ `string_type` ⊠ `path` ⊠内部⊠⊠⊠⊠⊠⊠⊠文字列⊠表現⊠⊠⊠⊠使⊠文字⊠文字列⊠型⊠。`preferred_separator` ⊠、推奨⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠区切⊠文字⊠。⊠⊠⊠⊠ POSIX 互換環境⊠⊠ / ⊠用⊠⊠⊠⊠⊠、Microsoft Windows ⊠⊠ \ ⊠使⊠⊠⊠⊠⊠。

10.5.1　path：ファイルパスの文字列

⊠⊠⊠⊠⊠⊠⊠文字列⊠表現⊠⊠。C++ ⊠文字列⊠⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠⊠⊠。

- `char`: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
- `wchar_t`: ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
- `char`: UTF-8 ⊠⊠⊠⊠⊠
- `char16_t`: UTF-16 ⊠⊠⊠⊠⊠
- `char32_t`: UTF-32 ⊠⊠⊠⊠⊠

`path::value_type` ⊠⊠⊠文字型⊠使⊠、⊠⊠文字列⊠⊠⊠⊠⊠⊠使⊠⊠⊠⊠⊠⊠実装依存⊠。`path` ⊠⊠⊠文字列⊠⊠⊠⊠⊠⊠渡⊠⊠⊠⊠、`path::value_type` ⊠文字型⊠文字⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠自動的⊠変換⊠行⊠⊠⊠。

```
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" ) ;
}
```

⊠⊠⊠、⊠⊠文字列⊠⊠⊠⊠⊠⊠渡⊠⊠⊠動⊠。

C++ ⊠⊠ UTF-8 ⊠⊠⊠⊠⊠⊠文字型⊠ `char` ⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠文字型⊠同⊠⊠⊠⊠、型⊠⊠⊠⊠⊠⊠⊠⊠区別⊠⊠⊠⊠。⊠⊠⊠⊠、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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

環境⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠使⊠⊠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⊠必要⊠。⊠⊠⊠⊠、環境⊠⊠⊠⊠⊠大文字小文字⊠区別⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠、`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⊠⊠⊠⊠。

⊠⊠⊠⊠⊠⊠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⊠⊠⊠⊠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
    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;
    std::u16string u16string() const;
    std::u32string u32string() const;
} ;
```

⊠⊠⊠⊠、⊠⊠⊠⊠関数 `string` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `std::string`, ⊠⊠⊠⊠関数 `u8string` ⊠ UTF-8 ⊠⊠⊠⊠⊠⊠⊠⊠ `std::string` ⊠返⊠。

```
int main()
{
    using namespace std::filesystem ;

    path name("hello.txt") ;
    std::ofstream file( name.string() ) ;
    file << "hello" ;
}
```

⊠⊠⊠⊠⊠⊠文字列⊠⊠⊠⊠⊠⊠⊠⊠変換⊠⊠返⊠ `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⊠方⊠⊠⊠⊠⊠⊠⊠文字列⊠返⊠ `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

作⊠提供⊠⊠⊠⊠。

operator /, operator /= ⊠⊠⊠⊠⊠⊠⊠⊠区切⊠⊠⊠⊠⊠⊠⊠⊠文字列⊠追加⊠行⊠。

```
int main()
{
    using namespace std::filesystem ;

    path p("/") ;

    // "/usr"
    p /= "usr" ;
    // "/usr/local/include"
    p = p / "local" / "include" ;
}
```

operator += ⊠単⊠⊠文字列⊠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⊠⊠⊠。

⊠⊠他⊠⊠、path ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠文字列⊠対⊠⊠操作⊠提供⊠⊠⊠⊠。以下⊠⊠⊠一例⊠。

```
int main()
{
    using namespace std::filesystem ;

    path p( "/home/cpp/src/main.cpp" ) ;
    // "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⊠対⊠⊠⊠⊠行⊠文字列処理⊠提供⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠名⊠⊠抜⊠出⊠処理、拡張子⊠⊠抜⊠出⊠処理、拡張子⊠変⊠⊠処理⊠⊠⊠。

10.6　file_status

⊠⊠⊠ file_status ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠保持⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠文字列⊠指定⊠⊠取得⊠⊠方法⊠別途⊠⊠⊠、⊠⊠方法⊠⊠毎回物理⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠発生⊠⊠。file_status ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠情報⊠保持⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠役割⊠果⊠⊠。

file_status ⊠物理⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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⊠。

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠権限⊠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⊠⊠⊠⊠使⊠。

```
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" ;
}
```

⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠通常⊠⊠⊠⊠⊠⊠数⊠、実行可能⊠⊠⊠⊠⊠⊠数⊠表示⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠表現⊠⊠ enum 型 perms ⊠、⊠⊠⊠⊠⊠⊠⊠⊠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⊠⊠

⊠、明示的⊠⊠⊠⊠⊠⊠必要⊠。

```
// ⊠⊠⊠
std::filesystem::perms a = 0755 ;

// OK
std::filesystem::perms b = std::filesystem::perms(0755) ;
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠書⊠換⊠⊠⊠⊠⊠⊠関数⊠以下⊠⊠⊠⊠。

```
void type(file_type ft) noexcept;
void permissions(perms prms) noexcept;
```

⊠⊠⊠、file_status ⊠⊠⊠⊠⊠単⊠⊠⊠⊠⊠⊠⊠用⊠⊠⊠⊠⊠⊠⊠、file_status ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠「書⊠換⊠⊠」⊠⊠⊠⊠⊠、単⊠ file_status ⊠⊠⊠⊠⊠⊠⊠⊠保持⊠⊠⊠⊠⊠値⊠書⊠換⊠⊠⊠⊠⊠、物理⊠⊠⊠⊠⊠⊠⊠⊠⊠反映⊠⊠⊠⊠⊠⊠⊠⊠⊠。物理⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠書⊠換⊠⊠⊠⊠、⊠⊠⊠関数⊠ permissions ⊠使⊠。

10.7 directory_entry

⊠⊠⊠ directory_entry ⊠⊠⊠⊠⊠⊠⊠文字列⊠保持⊠、⊠⊠⊠⊠⊠⊠⊠指⊠示⊠⊠⊠⊠⊠⊠情報⊠取得⊠⊠⊠⊠⊠⊠⊠。

物理⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠情報⊠毎回読⊠⊠⊠非効率的⊠。directory_entry ⊠⊠⊠⊠⊠⊠⊠⊠情報⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠用途⊠持⊠。

directory_entry ⊠物理⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠情報⊠読⊠込⊠⊠⊠⊠、変更⊠⊠⊠⊠⊠⊠⊠⊠⊠。

directory_entry ⊠構築⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠引数⊠⊠⊠ path ⊠与⊠⊠他、directory_iterator ⊠ recursive_directory_iterator ⊠⊠⊠得⊠⊠⊠⊠⊠⊠⊠。

```
int main()
{
    using namespace std::filesystem ;

    path p(".") ;

    // ⊠⊠⊠⊠⊠⊠文字列⊠⊠得⊠
    directory_entry e1(p) ;

    // ⊠⊠⊠⊠⊠⊠⊠⊠得⊠
    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⊠⊠用意⊠⊠⊠⊠⊠。`directory_entry`⊠使⊠⊠、⊠⊠⊠⊠情報⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、同⊠⊠⊠⊠⊠⊠⊠⊠対⊠⊠、物理⊠⊠⊠⊠⊠⊠⊠⊠⊠変更⊠⊠⊠⊠⊠⊠複数回⊠⊠⊠⊠⊠情報取得⊠行⊠⊠⊠効率的⊠⊠⊠。

```
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⊠
```

```
        entry.replace_filename("bar") ;
    }
```

directory_entry ⊠⊠⊠⊠⊠⊠用⊠⊠⊠⊠⊠、自動的⊠物理⊠⊠⊠⊠⊠⊠⊠⊠⊠変更⊠追随⊠⊠⊠⊠⊠、最新⊠情報⊠取得⊠⊠⊠⊠、明示的⊠⊠⊠⊠⊠関数 refresh ⊠呼⊠出⊠必要⊠⊠⊠。

10.8 directory_iterator

directory_iterator ⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠下⊠存在⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠形式⊠列挙⊠⊠⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠列挙⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠⊠⊠。

```
    int main()
    {
        using namespace std::filesystem ;
        directory_iterator iter("."), end ;
        std::copy( iter, end,
            std::ostream_iterator<path>(std::cout, "\n") ) ;
    }
```

directory_iterator ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ path ⊠渡⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠下⊠最初⊠⊠⊠⊠⊠⊠相当⊠⊠ directory_entry ⊠返⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠指定⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠下⊠⊠⊠⊠⊠⊠存在⊠⊠⊠場合、終端⊠⊠⊠⊠⊠⊠⊠⊠⊠。

directory_iterator ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠終端⊠⊠⊠⊠⊠⊠⊠⊠⊠。終端⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

directory_iterator::value_type ⊠ directory_entry ⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠入力⊠⊠⊠⊠⊠⊠⊠⊠⊠。

directory_iterator ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠（.）⊠親⊠⊠⊠⊠⊠⊠⊠（..）⊠列挙⊠⊠⊠。

directory_iterator ⊠⊠⊠⊠⊠⊠⊠⊠下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠順番⊠列挙⊠⊠⊠⊠未規定⊠。

directory_iterator ⊠⊠⊠⊠返⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠存在⊠⊠⊠可能性⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠存在⊠⊠⊠⊠⊠当⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠、存在⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

directory_iterator ⊠⊠⊠⊠⊠⊠⊠⊠作成⊠⊠⊠後⊠物理⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

⊠⊠⊠変更⊠、反映⊠⊠⊠⊠⊠⊠⊠未規定⊠⊠⊠。

directory_iterator ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠列挙時⊠動作⊠指定⊠⊠⊠ directory_options ⊠実引数⊠受⊠取⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠、C++17 ⊠標準規格⊠範囲⊠⊠ directory_iterator ⊠挙動⊠変更⊠⊠ directory_options ⊠規定⊠⊠⊠⊠⊠⊠。

10.8.1 エラー処理

directory_iterator ⊠構築時⊠⊠⊠⊠⊠発生⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠例外⊠⊠⊠⊠ error_code ⊠受⊠取⊠⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠⊠⊠実引数⊠ error_code ⊠⊠⊠⊠⊠⊠⊠⊠⊠渡⊠。

```
int main()
{
    using namespace std::filesystem ;

    std::error_code err ;

    directory_iterator iter("this-directory-does-not-exist", err) ;

    if ( err )
    {
        // ⊠⊠⊠処理
    }
}
```

directory_iterator ⊠⊠⊠⊠⊠⊠⊠⊠時⊠⊠⊠⊠⊠発生⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠例外⊠⊠⊠⊠ error_code ⊠受⊠取⊠⊠⊠場合、⊠⊠⊠⊠関数 increment ⊠呼⊠出⊠。

```
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" ;
        }

        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` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠対⊠⊠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⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

⊠⊠⊠⊠取⊠⊠⊠組⊠合⊠⊠⊠、`none`, `follow_directory_symlink`, `skip_permission_denied`, `follow_directory_symlink | skip_permission_denied` ⊠ 4 種類⊠⊠⊠。

```
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` ⊠、親⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠存在⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠終端⊠⊠⊠⊠⊠⊠⊠到達⊠⊠⊠可能性⊠⊠⊠⊠⊠注意⊠⊠⊠⊠。

```
int main()
{
    using namespace std::filesystem ;

    // 自分自身⊠含⊠⊠⊠⊠⊠⊠⊠⊠⊠対⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
    create_symlink(".", "foo") ;

    recursive_directory_iterator
        iter(".", directory_options::follow_directory_symlink), end ;

    // ⊠⊠⊠、⊠⊠⊠⊠終了⊠⊠⊠
    std::copy( iter, end, std::ostream_iterator<path>(std::cout) ) ;
}
```

`recursive_directory_iterator` ⊠現在⊠ `directory_options` ⊠得⊠⊠⊠、⊠⊠⊠⊠関数 `options` ⊠呼⊠。

```
class recursive_directory_iterator {
public :
    directory_options options() const ;
} ;
```

10.9.2　depth：深さ取得

recursive_directory_iterator ⊠現在列挙⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠深⊠⊠知⊠⊠⊠、⊠⊠⊠⊠関数 depth ⊠呼⊠。

```
class recursive_directory_iterator {
public :
    int depth() const ;
} ;
```

最初⊠⊠⊠⊠⊠⊠⊠⊠⊠深⊠⊠ 0 ⊠、次⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠深⊠⊠ 1、⊠⊠以降⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠同様⊠続⊠。

10.9.3　pop：現在のディレクトリーの列挙中止

⊠⊠⊠⊠関数 pop ⊠呼⊠⊠、現在列挙中⊠⊠⊠⊠⊠⊠⊠⊠⊠列挙⊠取⊠⊠⊠、親⊠⊠⊠⊠⊠⊠⊠⊠戻⊠。現在⊠⊠⊠⊠⊠⊠⊠⊠⊠初期⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠ depth() == 0 ⊠場合⊠、終端⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
class recursive_directory_iterator {
public :
    void pop();
    void pop(error_code& ec);
} ;
```

⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠以下⊠書⊠⊠⊠順番⊠⊠⊠⊠⊠⊠列挙⊠⊠環境⊠場合、

```
a
b
b/a
b/c
b/d
c
d
```

以下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠実行⊠⊠⊠、

```
int main()
{
    std::filesystem ;
    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⊠指⊠⊠⊠⊠⊠⊠⊠⊠順番⊠以下⊠⊠⊠⊠⊠⊠。

```
a
b
b/a
c
d
```

“b/a” ⊠到達⊠⊠時点⊠ pop() ⊠呼⊠⊠⊠⊠⊠、⊠⊠以上⊠⊠⊠⊠⊠⊠⊠⊠ b 下⊠列挙⊠中止⊠⊠、親⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠戻⊠。

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 ⊠現在⊠⊠⊠⊠⊠⊠⊠⊠⊠下⊠再帰的⊠列挙⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠機能⊠。

```
class recursive_directory_iterator {
public :
    bool recursion_pending() const ;
    void disable_recursion_pending() ;
} ;
```

recursion_pending() ⊠、直前⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠操作⊠後⊠disable_recursion_pending() ⊠呼⊠⊠⊠⊠⊠⊠場合、true ⊠返⊠。⊠⊠⊠⊠⊠場合⊠ false ⊠返⊠。

言⊠換⊠⊠⊠、disable_recursion_pending() ⊠呼⊠⊠直後⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠操作⊠⊠⊠⊠⊠⊠場合、recursion_pending() ⊠ false ⊠返⊠。

```
int main()
```

```
{
    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` ☒指☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒場合、☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒、☒☒☒☒☒☒☒☒☒下☒再帰的☒列挙☒☒☒☒☒☒☒。☒☒☒、`recursion_pending()` ☒ `false` ☒返☒場合、☒☒☒☒☒☒☒☒最適的☒列挙☒☒☒☒☒☒☒☒。☒☒☒☒☒☒☒操作☒行☒☒☒後☒ `recursion_pending()` ☒結果☒ `true` ☒戻☒。

☒☒☒、`disable_recursion_pending` ☒、現在指☒☒☒☒☒☒☒☒☒☒☒下☒再帰的☒列挙☒☒☒☒☒☒☒☒☒☒☒機能☒提供☒☒。

☒☒☒☒、☒☒☒☒☒☒☒☒☒☒☒☒以下☒☒☒☒☒☒☒☒☒☒☒☒☒☒☒、☒☒☒☒☒☒☒以下☒書☒☒☒順番☒☒☒☒☒☒列挙☒☒環境☒場合、

```
a
b
b/a
b/c
b/d
c
d
```

以下☒☒☒☒☒☒☒☒☒☒実行☒☒☒、

```
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⊠⊠⊠⊠⊠⊠⊠⊠順番⊠以下⊠⊠⊠⊠⊠⊠。

```
a
b
c
d
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠必⊠ `disable_recursion_pending()` ⊠呼⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠再帰的⊠列挙⊠行⊠⊠⊠、結果的⊠動作⊠ `directory_iterator` ⊠同⊠⊠⊠⊠。

`disable_recursion_pending` ⊠呼⊠出⊠⊠⊠⊠⊠⊠⊠、選択的⊠⊠⊠⊠⊠⊠⊠⊠⊠再帰的⊠列挙⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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);
```

⌧⌧⌧⌧・⌧⌧⌧⌧⌧・⌧⌧⌧⌧⌧⌧⌧（current working directory）⌧⌧絶対⌧⌧⌧返⌧。

temp_directory_path

```
path temp_directory_path();
path temp_directory_path(error_code& ec);
```

一時⌧⌧⌧⌧⌧作成⌧⌧⌧⌧最適⌧一時⌧⌧⌧⌧⌧⌧⌧（temporary directory）⌧⌧⌧⌧⌧⌧⌧⌧⌧返⌧。

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 ⌧⌧絶対⌧⌧⌧返⌧。p ⌧指⌧⌧⌧⌧⌧⌧存在⌧⌧⌧場合⌧挙動⌧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⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧p ⌧⌧、⌧⌧⌧⌧⌧⌧⌧⌧⌧、⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧⌧（.）、親⌧⌧⌧⌧⌧⌧⌧（..）⌧存在⌧⌧⌧絶対⌧⌧⌧返⌧。

weakly_canonical

```
path weakly_canonical(const path& p);
path weakly_canonical(const path& p, error_code& ec);
```

⊠⊠⊠⊠⊠⊠ p ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠解決⊠⊠、正規化⊠⊠⊠⊠⊠⊠返⊠。⊠⊠⊠⊠⊠⊠⊠正規化⊠⊠⊠⊠⊠定義⊠長⊠⊠⊠⊠⊠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 ⊠⊠⊠⊠⊠⊠⊠⊠ p ⊠対⊠⊠相対⊠⊠⊠返⊠。

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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠ p ⊠対⊠⊠相対⊠⊠⊠空⊠⊠⊠⊠⊠⊠⊠相対⊠⊠⊠返⊠。相対⊠⊠⊠空⊠⊠⊠⊠⊠ p ⊠返⊠。

10.10.3 作成

create_directory

```
bool create_directory(const path& p);
bool create_directory(const path& p, error_code& ec) noexcept;
```

p ⊠指⊠⊠⊠⊠⊠⊠⊠⊠⊠ 1 ⊠作成⊠⊠。新⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠作成⊠⊠⊠場合⊠ true ⊠、作成⊠⊠⊠⊠⊠⊠場合⊠ false ⊠返⊠。p ⊠既存⊠⊠⊠⊠⊠⊠⊠⊠⊠指⊠⊠⊠⊠新⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠作成⊠⊠⊠⊠⊠⊠場合⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。単⊠ false ⊠返⊠。

```
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⊠⊠⊠⊠⊠⊠。

create_directories

```
bool create_directories(const path& p);
bool create_directories(const path& p, error_code& ec) noexcept;
```

⊠⊠⊠⊠⊠⊠ p ⊠中⊠⊠⊠⊠⊠⊠⊠⊠⊠存在⊠⊠⊠⊠⊠⊠⊠⊠⊠作成⊠⊠。

以下⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠下⊠⊠⊠⊠⊠⊠⊠⊠ a ⊠下⊠⊠⊠⊠⊠⊠⊠⊠ b ⊠下⊠⊠⊠⊠⊠⊠⊠⊠ c ⊠作成⊠⊠。⊠⊠、途中⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ a, b ⊠存在⊠⊠⊠場合、⊠⊠⊠作成⊠⊠。

```
int main()
{
    using namespace std::filesystem ;
    create_directories("./a/b/c") ;
}
```

戻⊠値⊠、⊠⊠⊠⊠⊠⊠⊠⊠作成⊠⊠場合 `true`, ⊠⊠⊠⊠⊠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;
```

⊠⊠⊠⊠⊠⊠⊠ `to` ⊠解決⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `new_symlink` ⊠作成⊠⊠。

一部⊠ OS ⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠作成時⊠明示的⊠区別⊠⊠必要⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠作成⊠⊠⊠⊠⊠⊠ `create_symlink` ⊠⊠⊠⊠ `create_directory_symlink` ⊠使⊠⊠⊠⊠⊠⊠。

一部⊠ OS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠注意⊠⊠⊠⊠⊠⊠。

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;
```

⊠⊠⊠⊠⊠⊠ `to` ⊠解決⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ `new_symlink` ⊠作成⊠⊠。

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 ⊠解決⊠⊠⊠⊠⊠⊠⊠⊠⊠ 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 ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ to ⊠⊠⊠⊠⊠⊠。

copy_options ⊠⊠⊠⊠⊠挙動⊠変⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ enum 型⊠、以下⊠ enum 値⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

名前	意味
none	⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠存在⊠⊠場合⊠⊠⊠⊠
skip_existing	既存⊠⊠⊠⊠⊠⊠上書⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠報告⊠⊠⊠
overwrite_existing	既存⊠⊠⊠⊠⊠⊠上書⊠⊠⊠
update_existing	既存⊠⊠⊠⊠⊠⊠上書⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠古⊠⊠⊠上書⊠⊠⊠

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,
                copy_options options);
void copy(  const path& from, const path& to,
            copy_options options,
            error_code& ec) noexcept;
```

⊠⊠⊠⊠⊠⊠ from ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ to ⊠⊠⊠⊠⊠⊠。

copy_options ⊠⊠⊠⊠⊠挙動⊠変⊠⊠⊠⊠⊠⊠⊠⊠⊠型⊠ enum 型⊠、以下⊠ enum 値⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠関⊠⊠指定

名前	意味
`none`	⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`recursive`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠中身⊠⊠⊠⊠⊠⊠

- ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠関⊠⊠指定

名前	意味
`none`	⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`copy_symlinks`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠指⊠⊠⊠⊠⊠⊠直接⊠⊠⊠⊠⊠⊠
`skip_symlinks`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠無視⊠⊠

- ⊠⊠⊠方法⊠関⊠⊠指定

名前	意味
`none`	⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠下⊠中身⊠⊠⊠⊠⊠⊠
`directories_only`	⊠⊠⊠⊠⊠⊠⊠構造⊠⊠⊠⊠⊠⊠⊠⊠。非⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`create_symlinks`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠作成⊠⊠。⊠⊠⊠先⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠元⊠⊠⊠⊠⊠⊠⊠⊠絶対⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`create_hard_links`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠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 ⊠ new_symlink ⊠⊠⊠⊠⊠⊠。

10.10.5 削除

remove

```
    bool remove(const path& p);
    bool remove(const path& p, error_code& ec) noexcept;
```

⊠⊠⊠⊠⊠⊠ p ⊠指⊠⊠⊠⊠⊠⊠存在⊠⊠⊠⊠⊠⊠⊠削除⊠⊠。⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠削除⊠⊠⊠。⊠⊠⊠⊠先⊠削除⊠⊠⊠⊠。

戻⊠値⊠⊠⊠、⊠⊠⊠⊠⊠存在⊠⊠⊠場合 false ⊠返⊠。⊠⊠以外⊠場合 true ⊠返⊠。error_code ⊠⊠⊠⊠通知⊠受⊠取⊠関数⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠false ⊠返⊠。

remove_all

```
    uintmax_t remove_all(const path& p);
    uintmax_t remove_all(const path& p, error_code& ec) noexcept;
```

⊠⊠⊠⊠⊠⊠ p ⊠下⊠存在⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠削除⊠⊠後、p ⊠指⊠⊠⊠⊠⊠⊠削除⊠⊠。

⊠⊠⊠、p ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠指⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠下⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠存在⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠削除⊠⊠、⊠⊠⊠⊠⊠⊠⊠ p ⊠削除⊠⊠⊠。

p ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠指⊠場合、p ⊠削除⊠⊠⊠。

戻⊠値⊠⊠⊠、削除⊠⊠⊠⊠⊠⊠⊠個数⊠返⊠。error_code ⊠⊠⊠⊠通知⊠受⊠取⊠関数⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠⊠ static_cast<uintmax_t>(-1) ⊠返⊠。

10.10.6　変更

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);
```

⊠⊠⊠⊠⊠⊠ p ⊠⊠⊠⊠⊠⊠⊠⊠⊠変更⊠⊠。

opts ⊠ perm_options 型⊠ enum 値、replace, add, remove ⊠⊠⊠⊠⊠⊠⊠ 1 ⊠⊠、別途 nofollow ⊠指定⊠⊠⊠⊠⊠⊠⊠⊠。省略⊠⊠場合⊠ replace ⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠存在⊠⊠⊠⊠⊠⊠ foo ⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠対⊠⊠実行権限⊠付加⊠⊠⊠⊠、以下⊠⊠⊠⊠書⊠。

```
int main()
{
    using namespace std::filesystem ;

    permissions( "./foo", perms(0111), perm_options::add ) ;
}
```

perm_options ⊠以下⊠⊠⊠⊠ enum 値⊠持⊠。

名前	意味
replace	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ prms ⊠置⊠換⊠⊠
add	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ prms ⊠指定⊠⊠⊠⊠⊠⊠追加⊠⊠
remove	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ prms ⊠指定⊠⊠⊠⊠⊠⊠取⊠除⊠
nofollow	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠先⊠⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠変更⊠⊠

⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠置⊠換⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠書⊠換⊠⊠⊠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 ⊠⊠⊠⊠⊠⊠⊠。

old_p ⊠ new_p ⊠同⊠存在⊠⊠⊠⊠⊠⊠⊠指⊠場合、何⊠⊠⊠⊠。

```
int main()
{
    using namespace std:filesystem ;

    // 何⊠⊠⊠⊠
    rename("foo", "foo") ;
}
```

⊠⊠以外⊠場合、⊠⊠⊠⊠⊠件⊠⊠以下⊠⊠⊠⊠挙動⊠発生⊠⊠。

⊠⊠、⊠⊠⊠⊠前⊠ new_p ⊠既存⊠⊠⊠⊠⊠⊠指⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠件⊠⊠ new_p ⊠削除⊠⊠⊠。

```
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 ⊠内容⊠"old_p"
    // ⊠⊠⊠⊠ new_p ⊠内容⊠"new_p"

    // ⊠⊠⊠⊠ old_p ⊠ new_p ⊠⊠⊠⊠⊠
    // ⊠⊠⊠⊠⊠ new_p ⊠削除⊠⊠⊠
    rename("old_p", "new_p") ;

    std::ifstream new_p("new_p") ;

    std::string text ;
    new_p >> text ;

    // "old_p"
    std::cout << text ;
}
```

⊠⊠、new_p ⊠既存⊠空⊠⊠⊠⊠⊠⊠⊠⊠指⊠⊠⊠⊠場合、POSIX 準拠 OS ⊠⊠⊠⊠、⊠⊠⊠⊠⊠伴⊠⊠ new_p ⊠削除⊠⊠⊠。他⊠ OS ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
int main()
{
    using namespace std::experimental::filesystem ;

    create_directory("old_p") ;
    create_directory("new_p") ;

    // POSIX 準拠環境⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠保証⊠⊠⊠
    rename("old_p", "new_p") ;
}
```

old_p ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠先⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

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;
```

⊠⊠⊠⊠⊠⊠ path ⊠指⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ new_size ⊠⊠⊠。

⊠⊠⊠⊠⊠ POSIX ⊠ truncate() ⊠行⊠⊠⊠⊠⊠⊠⊠⊠振⊠舞⊠。⊠⊠⊠、⊠⊠⊠⊠⊠小⊠⊠⊠⊠⊠⊠⊠⊠場合、余計⊠⊠⊠⊠⊠捨⊠⊠⊠⊠。⊠⊠⊠⊠⊠大⊠⊠⊠⊠⊠⊠⊠⊠場合、増⊠⊠⊠⊠⊠⊠ null ⊠⊠⊠（\0）⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠最終⊠⊠⊠⊠日時⊠更新⊠⊠⊠。

10.10.7　情報取得

⊠⊠⊠⊠⊠⊠⊠⊠判定

⊠⊠⊠⊠⊠⊠⊠⊠表現⊠⊠ file_type 型⊠ enum ⊠⊠⊠、⊠⊠ enum 値⊠以下⊠⊠⊠⊠⊠⊠⊠⊠⊠。

名前	意味
`none`	⊠⊠⊠⊠⊠⊠⊠⊠決定⊠⊠⊠⊠⊠⊠⊠⊠
`not_found`	⊠⊠⊠⊠⊠発見⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠示⊠疑似⊠⊠⊠⊠⊠⊠⊠
`regular`	通常⊠⊠⊠⊠⊠
`directory`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`symlink`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`block`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`fifo`	FIFO ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`socket`	⊠⊠⊠⊠⊠⊠⊠⊠
`unknown`	⊠⊠⊠⊠⊠存在⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠決定⊠⊠⊠⊠

⊠⊠他⊠、実装依存⊠⊠⊠⊠⊠⊠⊠⊠⊠追加⊠⊠⊠⊠⊠可能性⊠⊠⊠。

⊠⊠⊠⊠⊠⊠⊠⊠調⊠⊠⊠⊠、`file_status` ⊠⊠⊠⊠⊠関数 `type` ⊠戻⊠値⊠調⊠⊠⊠⊠⊠。

以下⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠存在⊠⊠⊠⊠⊠⊠ `foo` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠調⊠⊠⊠⊠⊠⊠。

```
int main()
{
    using namespace std::filesystem ;

    auto s = status("./foo") ;
    bool b = s.type() == file_type::directory ;
}
```

⊠⊠、`status` ⊠⊠⊠⊠ `path` ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠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⊠⊠⊠⊠⊠⊠⊠⊠変更⊠加⊠⊠⊠状態⊠、同⊠⊠⊠⊠⊠⊠対⊠⊠何度⊠⊠⊠⊠⊠情報⊠取得⊠⊠場合⊠、

`file_status` ⊠使⊠⊠⊠⊠⊠⊠⊠。

⊠⊠⊠⊠⊠ `is_x` ⊠⊠⊠形式⊠⊠⊠⊠関数⊠、⊠⊠⊠⊠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⊠⊠⊠⊠関数⊠名前⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠判定⊠⊠表⊠。

名前	意味
`is_regular_file`	通常⊠⊠⊠⊠⊠
`is_directory`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`is_symlink`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`is_block`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`is_fifo`	FIFO ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`is_socket`	⊠⊠⊠⊠⊠⊠⊠⊠

⊠⊠、単一⊠⊠⊠⊠⊠⊠⊠⊠⊠調⊠⊠⊠⊠⊠⊠⊠以下⊠⊠⊠⊠名前⊠⊠⊠⊠関数⊠存在⊠⊠。

名前	意味
`is_other`	⊠⊠⊠⊠⊠存在⊠、通常⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠
`is_empty`	⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠下⊠空⊠⊠⊠⊠ `true` ⊠返⊠。⊠⊠⊠⊠⊠⊠非⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠⊠ 0 ⊠⊠⊠⊠ `true` ⊠返⊠。

status

```
file_status status(const path& p);
file_status status(const path& p, error_code& ec) noexcept;
```

⊠⊠⊠⊠⊠⊠ p ⊠⊠⊠⊠⊠⊠情報⊠格納⊠⊠ file_status ⊠返⊠。
p ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠先⊠⊠⊠⊠⊠⊠ file_status ⊠返⊠。

status_known

```
bool status_known(file_status s) noexcept;
```

s.type() != file_type::none ⊠返⊠。

symlink_status

```
file_status symlink_status(const path& p);
file_status symlink_status(const path& p, error_code& ec) noexcept;
```

status ⊠同⊠⊠⊠、p ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠ status ⊠返⊠。

equivalent

```
bool equivalent(const path& p1, const path& p2);
bool equivalent(const path& p1, const path& p2,
                error_code& ec) noexcept;
```

p1 ⊠ p2 ⊠物理⊠⊠⊠⊠⊠⊠⊠⊠上、同一⊠⊠⊠⊠⊠⊠⊠⊠場合、true ⊠返⊠。⊠⊠⊠⊠⊠場合 false ⊠返⊠。

exists

```
bool exists(file_status s) noexcept;
bool exists(const path& p);
bool exists(const path& p, error_code& ec) noexcept;
```

s, p ⊠指⊠⊠⊠⊠⊠⊠存在⊠⊠⊠⊠⊠⊠⊠ true ⊠返⊠。⊠⊠⊠⊠⊠場合 false ⊠返⊠。

file_size

```
uintmax_t file_size(const path& p);
uintmax_t file_size(const path& p, error_code& ec) noexcept;
```

p ⊠指⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠返⊠。

⊠⊠⊠⊠⊠存在⊠⊠⊠場合⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠通常⊠⊠⊠⊠⊠⊠場合、⊠⊠⊠⊠⊠⊠⊠⊠返⊠。⊠⊠以外⊠場合、挙動⊠実装依存⊠⊠⊠。

⊠⊠⊠通知⊠ `error_code` ⊠受⊠取⊠関数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、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⊠返⊠。

⊠⊠⊠通知⊠ `error_code` ⊠受⊠取⊠関数⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠、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⊠返⊠。

```
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⊠⊠⊠⊠⊠⊠⊠⊠⊠実装⊠起因⊠⊠時刻⊠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` ⊠⊠、⊠⊠⊠⊠（⊠⊠正確⊠⊠ TrivialClock）⊠要件⊠満⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠値⊠正確⊠表現⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠⊠⊠⊠具体的⊠型⊠実装依存⊠⊠⊠、完全⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

⊠⊠⊠⊠⊠⊠⊠時間⊠扱⊠⊠⊠極⊠⊠困難⊠⊠⊠。⊠⊠⊠⊠現在時刻⊠設定⊠⊠⊠⊠、差分⊠時間⊠設定⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
int main()
{
    using namespace std::experimental::filesystem ;
    using namespace std::chrono ;
    using namespace std::literals ;

    // 最終更新日時⊠取得
    auto timestamp = last_write_time( "foo" ) ;

    // 時刻⊠ 1 時間進⊠⊠
    timestamp += 1h ;
    // 更新
    last_write_time( "foo", timestamp ) ;


    // 現在時刻⊠取得
    auto now = file_time_type::clock::now() ;

    last_write_time( "foo", now ) ;
}
```

⊠⊠⊠、多⊠⊠実装⊠⊠ file_time_type ⊠⊠⊠、time_point<std::chrono::system_clock> ⊠使⊠⊠⊠⊠⊠。file_time_type::clock ⊠ system_clock ⊠⊠⊠⊠、system_clock::to_time_t ⊠ system_clock::from_time_t ⊠⊠⊠⊠ time_t 型⊠⊠相互変換⊠⊠⊠⊠⊠⊠⊠⊠、⊠⊠⊠⊠⊠⊠⊠⊠⊠。

```
// file_time_type::clock ⊠ system_clock ⊠⊠⊠場合

int main()
{
    using namespace std::experimental::filesystem ;
    using namespace std::chrono ;

    // 最終更新日時⊠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' ;
    // 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 ) ;
}
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠見⊠⊠⊠⊠、C++ ⊠⊠現在 `<chrono>` ⊠⊠利用⊠⊠⊠ C++ 風⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠。⊠⊠問題⊠将来⊠規格改定⊠改善⊠⊠⊠⊠⊠⊠。

read_symlink

```
path read_symlink(const path& p);
path read_symlink(const path& p, error_code& ec);
```

⊠⊠⊠⊠⊠⊠⊠⊠⊠ p ⊠解決⊠⊠⊠先⊠⊠⊠⊠⊠⊠⊠⊠返⊠。

p ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠場合⊠⊠⊠⊠⊠⊠⊠。

space

```
space_info space(const path& p);
space_info space(const path& p, error_code& ec) noexcept;
```

⊠⊠⊠⊠⊠⊠ p ⊠指⊠先⊠容量⊠取得⊠⊠。

⊠⊠⊠ `space_info` ⊠以下⊠⊠⊠⊠定義⊠⊠⊠⊠⊠。

```
struct space_info {
    uintmax_t capacity;
    uintmax_t free;
    uintmax_t available;
};
```

⊠⊠関数⊠、POSIX ⊠ `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
`capacity`	総容量
`free`	空⊠容量
`available`	権限⊠⊠⊠⊠⊠⊠⊠⊠使⊠⊠空⊠容量

索引

*this, 34
::value, 22
<algorithm>, 153, 156, 183, 220
<any>, 104
<atomic>, 221
<chrono>, 262
<cmath>, 165, 221
<cstddef>, 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☒☒☒☒, 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⊠⊠⊠⊠⊠⊠, 182
- Boyer–Moore 文字列検索⊠⊠⊠⊠⊠⊠, 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⊠⊠⊠⊠問題, 55
  - 解決⊠⊠⊠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 ⊠⊠⊠⊠⊠⊠⊠⊠, 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
  - ⊠⊠⊠⊠⊠, 75
  - 展開, 75
  - 変数, 75, 78
- insert, 215

- 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
  - ⊠⊠⊠⊠⊠⊠実引数, 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 ⊠⊠⊠⊠⊠⊠, 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⊠⊠, 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
  ⊠⊠⊠⊠定義⊠⊠⊠⊠, 130
std::string_view, 123
  ⊠⊠⊠⊠定義⊠⊠⊠⊠, 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⊠⊠⊠⊠⊠⊠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

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 ⊠⊠⊠⊠⊠, 230
UTF-32 ⊠⊠⊠⊠⊠, 230
UTF-8 ⊠⊠⊠⊠⊠, 230
UTF-8 文字⊠⊠⊠⊠, 28

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
    ⊠⊠⊠初期化, 89
    初期化, 88
    宣言, 88
    大小比較, 102
    代入, 92
    ⊠⊠⊠⊠⊠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

⊠⊠⊠⊠⊠⊠初期化, 23
⊠⊠⊠⊠⊠⊠, 141
⊠⊠⊠⊠⊠⊠ R（保管標本）, 187
⊠⊠⊠⊠⊠⊠ S（選択標本）, 186
⊠⊠⊠⊠⊠⊠, 133
⊠⊠⊠⊠⊠宣言, 19
⊠⊠⊠⊠⊠多項式, 168
演算子⊠評価順序, 45

型
    IntType, 83
    scoped enum, 81
    std::byte, 81
    ⊠⊠⊠, 81
型安全⊠
    union, 85
型⊠違⊠定数, 21
型指定子, 9
型非安全, 86
可変長 using 宣言, 79
可変長⊠⊠⊠⊠⊠⊠, 30
関数型（例外指定）, 29
関数宣言, 19
関数⊠⊠⊠⊠⊠⊠, 21
関数⊠戻⊠値⊠型推定, 8
完全型, 206
機能⊠⊠⊠, 1
機能⊠⊠⊠⊠⊠⊠, 2
⊠⊠⊠⊠⊠⊠⊠⊠⊠, 175
⊠⊠⊠⊠⊠⊠⊠⊠, 175
球⊠⊠⊠⊠関数, 173
球面⊠⊠⊠⊠⊠⊠陪関数, 167
⊠⊠⊠宣言, 18
⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, 14, 37
⊠⊠⊠⊠⊠⊠型, 14
厳格⊠ auto, 9
⊠⊠言語, vi
構造化束縛, 63
    完全形⊠名前, 71
    ⊠⊠⊠, 73
    仕様, 69
    配列, 69
    非 static ⊠⊠⊠⊠⊠⊠⊠, 74
    ⊠⊠⊠⊠⊠⊠⊠⊠, 74
構造化束縛宣言, 67
古典的⊠ union, 86
⊠⊠⊠⊠⊠⊠⊠⊠⊠, 16, 34
⊠⊠⊠⊠⊠⊠⊠⊠⊠関数, 220
    data, 220
    empty, 220
    size, 220
⊠⊠⊠⊠⊠時条件分岐, 46, 49
⊠⊠⊠⊠⊠時定数, 47

最小公倍数, 223
⊠⊠⊠付⊠解放関数, 25

最大公約数, 222
⊠⊠⊠⊠⊠⊠⊠⊠⊠, 14
⊠⊠⊠⊠⊠実行⊠⊠⊠⊠, 161
指数積分, 174
実行⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, 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
    ⊠⊠⊠⊠⊠時～, 46, 49
    実行時⊠～, 46
    ⊠⊠⊠⊠⊠⊠時⊠～, 48
初期化文付⊠条件文, 56
初期化⊠⊠⊠⊠⊠⊠⊠⊠⊠, 15
⊠⊠⊠⊠⊠⊠⊠⊠⊠, 250
推定⊠⊠⊠, 59
数学⊠特殊関数, 165
    ⊠⊠⊠⊠⊠多項式, 168
    球⊠⊠⊠⊠関数, 173
    球面⊠⊠⊠⊠⊠⊠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⊠⊠⊠⊠関数, 171
    第 1 種変形⊠⊠⊠⊠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⊠⊠⊠⊠関数, 172
    第 3 種完全楕円積分, 169
    第 3 種不完全楕円積分, 171
    ⊠⊠⊠⊠関数, 171
    ⊠⊠⊠関数, 168
    ⊠⊠⊠⊠多項式, 166
    ⊠⊠⊠⊠陪多項式, 166
    ⊠⊠⊠⊠⊠⊠⊠関数, 174
    ⊠⊠⊠⊠⊠⊠多項式, 166
    ⊠⊠⊠⊠⊠⊠陪関数, 167
数値区切⊠文字, 5
⊠⊠⊠⊠, 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⊠⊠⊠⊠関数, 171
第 1 種変形⊠⊠⊠⊠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⊠⊠⊠⊠関数, 172
第 3 種完全楕円積分, 169
第 3 種不完全楕円積分, 171
畳⊠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
    ⊠⊠⊠⊠⊠時～, 47
⊠⊠⊠⊠⊠⊠⊠
    `create_directories`, 249
    `create_directory`, 249
    `create_directory_symlink`, 250
⊠⊠⊠⊠⊠⊠⊠区切⊠文字, 230
⊠⊠⊠競合, 159
⊠⊠⊠⊠⊠⊠, 159
⊠⊠⊠⊠⊠⊠実引数⊠推定, 59
⊠⊠⊠⊠⊠⊠宣言, 18, 19, 20
動的⊠⊠⊠⊠⊠, 133
動的⊠⊠⊠⊠⊠⊠⊠⊠, 137
⊠⊠⊠⊠⊠⊠, 27

名前空間
    ⊠⊠⊠⊠⊠⊠～, 39
二項 `fold` 式, 31, 33
二進数⊠⊠⊠⊠, 5
⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, 230
⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, 230
⊠⊠⊠⊠⊠⊠名前空間, 39
⊠⊠⊠⊠関数, 171
⊠⊠⊠⊠⊠⊠⊠, 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

⊠⊠⊠, 81
    ⊠⊠⊠数, 81
⊠⊠⊠⊠⊠⊠干渉⊠⊠⊠, 175
⊠⊠⊠⊠⊠⊠, 251
⊠⊠⊠⊠⊠⊠⊠, 236, 254
⊠⊠⊠⊠⊠⊠⊠⊠⊠, 30
⊠⊠⊠⊠実行⊠⊠⊠⊠, 162
⊠⊠⊠⊠非⊠⊠⊠⊠⊠実行⊠⊠⊠⊠, 162
非 `static` ⊠⊠⊠⊠⊠⊠⊠, 23
非型⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, 61
非順序連想⊠⊠⊠⊠⊠, 209
    `merge`, 210
左 `fold`, 31, 33
非標準属性, 63
⊠⊠⊠⊠⊠⊠⊠⊠, 225

⊠⊠⊠⊠⊠⊠⊠, 256
⊠⊠⊠⊠⊠⊠⊠⊠, 236
⊠⊠⊠⊠⊠⊠
    absolute, 248
    canonical, 248
    current_path, 248
    proximate, 249
    relative, 249
    temp_directory_path, 248
    weakly_canonical, 248
    取得, 248
    操作, 234, 248
⊠⊠⊠⊠⊠⊠文字列, 229
不完全型, 206
浮動小数点数⊠⊠⊠⊠⊠⊠, 28
浮動小数点数⊠⊠⊠⊠, 27
⊠⊠⊠⊠⊠⊠⊠, 142
    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
    ⊠⊠⊠⊠⊠⊠⊠⊠, 146
⊠⊠⊠⊠⊠⊠⊠⊠型, 9
⊠⊠⊠⊠⊠⊠⊠言語 C++, v
並列⊠⊠⊠⊠⊠⊠, 153, 155, 156
    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⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, 162
    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

⊠⊠⊠⊠⊠⊠⊠⊠, 21
右 fold, 31, 33
⊠⊠⊠⊠⊠⊠⊠⊠⊠, 18
⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, 140
⊠⊠⊠⊠⊠管理⊠⊠⊠⊠⊠⊠, 196
⊠⊠⊠⊠⊠⊠⊠⊠, 133, 140
    取得, 139
    ⊠⊠⊠⊠⊠⊠⊠, 142
    ⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, 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
⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠, 147
    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⊠対⊠⊠⊠問⊠合⊠⊠⊠、電子⊠⊠⊠（info@asciidwango.jp）⊠⊠⊠願⊠⊠⊠⊠⊠⊠。
但⊠、本書⊠記述内容⊠越⊠⊠⊠質問⊠⊠⊠答⊠⊠⊠⊠⊠⊠⊠⊠、⊠了承⊠⊠⊠⊠。

江添亮⊠ 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⊠⊠⊠⊠
〒 104-0061
東京都中央区銀座 4-12-15 歌舞伎座⊠⊠⊠
編集 03-3549-6153
電子⊠⊠⊠ 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（⊠⊠⊠⊠⊠⊠⊠⊠⊠・⊠⊠⊠⊠⊠⊠）
受付時間 9：00〜17:00（土日 祝日 年末年始⊠除⊠）
http://www.kadokawa.co.jp/

印刷・製本　株式会社⊠⊠⊠⊠⊠⊠⊠

Printed in Japan

落丁・乱丁本⊠⊠取⊠替⊠⊠⊠⊠⊠⊠。下記 KADOKAWA 読者係⊠⊠⊠連絡⊠⊠⊠⊠。
送料小社負担⊠⊠⊠取⊠替⊠⊠⊠⊠⊠⊠。
但⊠、古書店⊠本書⊠購入⊠⊠⊠⊠⊠場合⊠⊠取⊠替⊠⊠⊠⊠⊠⊠。
電話　049-259-1100（9:00-17:00/土日、祝日、年末年始⊠除⊠）
〒354-0041　埼玉県入間郡三芳町藤久保 550-1
定価⊠⊠⊠⊠⊠表示⊠⊠⊠⊠⊠⊠。

ISBN: 978-4-04-xxxxxx-x

⊠⊠⊠⊠⊠⊠⊠⊠編集部
編　集　星野浩章