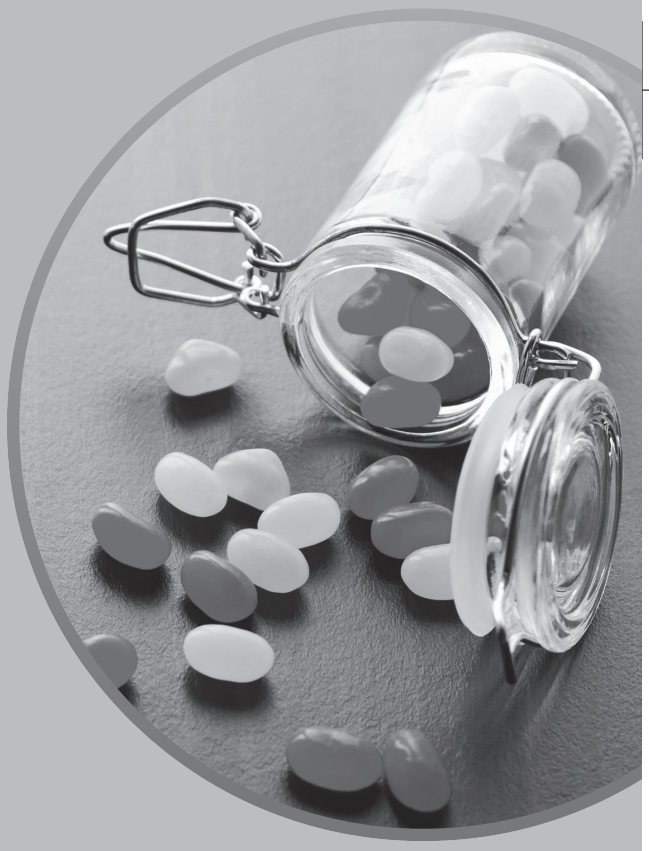
江添亮の詳説C++17

江添 亮 著



ASCII
DWANGO

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 は 2017 年に制定されることが予定されている最新の C++ 規格で、本書で解説する。

0.2 C++ の将来の規格

0.2.1 C++20

C++20 は 2020 年に制定されることが予定されている次の C++ 規格だ。この規格では、モジュール、コンセプト、レンジ、ネットワークに注力することが予定されている。

0.3 コア言語とライブラリ

C++ の標準規格は、大きく分けて、C プリプロセッサーとコア言語とライブラリからなる。

C プリプロセッサーとは、C++ が C 言語から受け継いだ機能だ。ソースファイルをトークン列単位で分割して、トークン列の置換ができる。

コア言語とは、ソースファイルに書かれたトークン列の文法とその意味のことだ。

ライブラリとは、コア言語機能を使って実装されたもので、標準に提供されているものだ。標準ライブラリには、純粋にコア言語の機能のみで実装できるものと、それ以外の実装依存の方法やコンパイラーマジックが必要なものとがある。

目次

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
```

このそびえ立つクソのようなコードは現実に書かれている。このコードは GCC と MSVC と EDG と Clang という現実に使われている主要な 4 つの C++ コンパイラーに対応した rvalue リファレンスが実装されているかどうかを判定する機能テストコードだ。

この複雑なプリプロセッサーを解釈した結果、__USE_RVALUE_REFERENCES というプリプロセッサーマクロの値が、もし C++ コンパイラーが rvalue リファレンスをサポートしているならば 1、そうでなければ 0 となる。後は、このプリプロセッサーマクロで #if ガードしたコードを書く。

```
// 文字列を処理する関数
void process_string( std::string const & str ) ;

#if __USE_RVALUE_REFERENCES == 1
// 文字列をムーブして処理してよい実装の関数
// C++ コンパイラーが rvalue リファレンスを実装していない場合はコンパイルされない
void process_string( std::string && str ) ;
#endif
```

C++17 では、上のようなそびえ立つクソのようなコードを書かなくてもすむように、標準の機能テストマクロが用意された。C++ 実装が特定の機能をサポートしている場合、対応する機能テストマクロが定義される。機能テストマクロの値は、その機能が C++ 標準に採択された年と月を合わせた 6 桁の整数で表現される。

たとえば rvalue リファレンスの場合、機能テストマクロの名前は__cpp_rvalue_referencesとなっている。rvalue リファレンスは 2006 年 10 月に採択されたので、機能テストマクロの値は 200610 という値になっている。将来 rvalue リファレンスの機能が変更されたときは機能テストマクロの値も変更される。この値を調べることによって使っている C++ コンパイラーはいつの時代の C++ 標準の機能をサポートしているか調べることもできる。

この機能テストマクロを使うと、上のコードの判定は以下のように書ける。

```
// 文字列を処理する関数
void process_string( std::string const & str ) ;

#ifdef __cpp_rvalue_references
// 文字列をムーブして処理してよい実装の関数
// C++ コンパイラーが rvalue リファレンスを実装していない場合はコンパイルさ
```

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

機能テストマクロの値は通常は気にする必要がない。機能テストマクロが存在するかどうかで機能の有無を確認できるので、通常は `#ifdef` を使えばよい。

1.2 __has_include 式： ヘッダーファイルの存在を判定する

`__has_include` 式は、ヘッダーファイルが存在するかどうかを調べるための機能だ。

```
__has_include( ヘッダー名 )
```

`__has_include` 式はヘッダー名が存在する場合 1 に、存在しない場合 0 に置換される。

たとえば、C++17 の標準ライブラリにはファイルシステムが入る。そのヘッダー名は `<filesystem>` だ。C++ コンパイラーがファイルシステムライブラリをサポートしているかどうかを調べるには、以下のように書く。

```
#if __has_include(<filesystem>)
// ファイルシステムをサポートしている
#include <filesystem>
namespace fs = std::filesystem ;
#else
// 実験的な実装を使う
#include <experimental/filesystem>
namespace fs = std::experimental::filesystem ;
#endif
```

C++ 実装が`__has_include` をサポートしているかどうかは、`__has_include` の存在をプリプロセッサーマクロのように `#ifdef` で調べることによって判定できる。

```
#ifdef __has_include
// __has_include をサポートしている
#else
// __has_include をサポートしていない
#endif
```

`__has_include` 式は `#if` と `#elif` の中でしか使えない。

```
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 で追加された新機能は少ない。C++14 は C++03 と同じくマイナーアップデートという位置付けで積極的な新機能の追加は見送られたからだ。

2.1 二進数リテラル

二進数リテラルは整数リテラルを二進数で記述する機能だ。整数リテラルのプレフィクスに 0B もしくは 0b を書くと、二進数リテラルになる。整数を表現する文字は 0 と 1 しか使えない。

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

二進数リテラルは浮動小数点数リテラルには使えない。

機能テストマクロは __cpp_binary_literals, 値は 201304。

2.2 数値区切り文字

数値区切り文字は、整数リテラルと浮動小数点数リテラルの数値をシングルクオート文字で区切ることができる機能だ。区切り桁は何桁でもよい。

```
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 ;
}
```

大きな数値を扱うとき、ソースファイルに 100000000 と 1000000000 と書かれていた場合、どちらが大きいのか人間の目にはわかりにくい。人間が読んでわかりにくいコードは間違いの元だ。数値区切りを使うと、100'000'000 □ 1'000'000'000 のように書くことができる。これはわかりやすい。

他には、1 バイト単位で見やすいように区切ることもできる。

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

数値区切りはソースファイルを人間が読みやすくするための機能で、数値に影響を与えない。

2.3 [[deprecated]] 属性

[[deprecated]] 属性は名前とエンティティが、まだ使えるものの利用は推奨されない状態であることを示すのに使える。[[deprecated]] 属性が指定できる名前とエンティティは、クラス、typedef 名、変数、非 static データメンバー、関数、名前空間、enum, enumerator, テンプレートの特殊化だ。

それぞれ以下のように指定できる。

```
// 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
// メンバー関数も同じ文法
// どちらでもよい
[[deprecated]] void function_name1() { }
void function_name2 [[deprecated]] () { }


// クラス
// union も同じ
class [[deprecated]] class_name
{
// 非 static データメンバー
[[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 ; }

// テンプレートの特殊化

template < typename T >
class template_name { } ;

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

[[deprecated]] 属性が指定された名前やエンティティを使うと、C++ コンパイラーは警告メッセージを出す。

[[deprecated]] 属性には、文字列を付け加えることができる。これは C++ 実装によっては警告メッセージに含まれるかもしれない。

```
[[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' ;
    }
} ;
```

機能テストマクロは __cpp_generic_lambdas, 値は 201304。

2.7 初期化ラムダキャプチャー

初期化ラムダキャプチャーはラムダキャプチャーする変数の名前と式を書くことができる機能だ。

ラムダ式は書かれた場所から見えるスコープの変数をキャプチャーする。

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

初期化ラムダキャプチャーはラムダキャプチャーに初期化子を書くことができる機能だ。

```
int main()
{
    int x = 0 ;
    [ x = x, y = x, &ref = x, x2 = x * 2 ]
    {// キャプチャーされた変数を使う
        x ;
        y ;
        ref ;
        x2 ;
    } ;
}
```

初期化ラムダキャプチャーは、“識別子 = expr” という文法でラムダ導入子 [] の中に書く。するとあたかも “auto 識別子 = expr ;” と書いたかのように変数が作られる。これによりキャプチャーする変数の名前を変えたり、まったく新しい変数を宣言することができる。

初期化ラムダキャプチャーの識別子の前に & を付けると、リファレンスキャプチャー扱いになる。

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

    // x ☒ 1
}
```

初期化ラムダキャプチャーが追加された理由には変数の名前を変えたりまったく新しい変数を導入したいという目的の他に、非 static データメンバーをコピーキャプチャーするという目的がある。

以下のコードには問題があるが、わかるだろうか。

```
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 は X::data を返すクロージャーオブジェクトを返す。

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

これを見ると、コピーキャプチャーである [=] を使っているので、data はクロージャーオブジェクト内にコピーされているように思える。しかし、ラムダ式は非 static データメンバーをキャプチャーしてはいない。ラムダ式がキャプチャーしているのは this ポインターだ。上のコードと下のコードは同じ意味になる。

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

さて、main 関数をもう一度見てみよう。

```
int main()
{
    // クロージャーオブジェクトを代入するための変数
    std::function< int() > f ;

    {
        X x ; // x が構築される
        f = x.get_closure_object() ;
        // x が破棄される
    }

    // すでに x は破棄された
    // return &x->data で破棄された x を参照する
    std::cout << f() << std::endl ;
}
```

なんと、すでに破棄されたオブジェクトへのリファレンスを参照してしまっている。これは未定義の動作だ。

初期化ラムダキャプチャーを使えば、非 static データメンバーもコピーキャプチャーできる。

```
auto get_closure_object()
{
```

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

なお、ムーブキャプチャーは存在しない。ムーブというのは特殊なコピーなので初期化ラムダキャプチャーがあれば実現できるからだ。

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

機能テストマクロは __cpp_init_captures, 値は 201304。

2.8　変数テンプレート

変数テンプレートとは変数宣言をテンプレート宣言にできる機能だ。

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

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

これだけではわからないだろうから、順を追って説明する。
C++ ではクラスを宣言できる。

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

C++ ではクラスをテンプレート宣言できる。型テンプレートパラメーターは型として使える。

```
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++ では関数を宣言できる。

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

C++ では関数をテンプレート宣言できる。型テンプレートパラメーターは型として使える。

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

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

C++11 では typedef 名を宣言するためにエイリアス宣言ができる。

```
using type = int ;
```

C++11 ではエイリアス宣言をテンプレート宣言できる。型テンプレートパラメーターは型として使える。

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

int main()
```

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

そろそろパターンが見えてきたのではないだろうか。C++ では一部の宣言はテンプレート宣言できるということだ。このパターンを踏まえて以下を考えてみよう。

C++ では変数を宣言できる。

```
int variable{} ;
```

C++14 では変数宣言をテンプレート宣言できる。型テンプレートパラメーターは型として使える。

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

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

変数テンプレートは名前どおり変数宣言をテンプレート宣言できる機能だ。変数テンプレートはテンプレート宣言なので、名前空間スコープとクラススコープの中にしか書くことができない。

```
// これはグローバル名前空間スコープという特別な名前空間スコープ

namespace ns {
// 名前空間スコープ
}

class
{
// クラススコープ
} ;
```

変数テンプレートの使い道は主に 2 つある。

2.8.1 意味は同じだが型が違う定数

プログラムでマジックナンバーを変数化しておくのは良い作法であるとされている。たとえば円周率を 3.14... などと書くよりも pi という変数名で扱ったほうがわかりやすい。変数化すると、円周率の値が後で変わったときにプログラムを変更するのも楽になる。

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

しかし、円周率を表現する型が複数ある場合どうすればいいのか。よくあるのは名前を分ける方法だ。

```
constexpr float pi_f = 3.1415 ;
constexpr double pi_d = 3.1415926535 ;
constexpr int pi_i = 3 ;
// 任意の精度の実数を表現できるクラスとする
const Real pi_r("3.14159265358979323846264338327 9") ;
```

しかしこれは、使う側で型によって名前を変えなければならない。

```
// 円の面積を計算する関数
template < typename T >
T calc_area( T r )
{
    // T の型によって使うべき名前が変わる
    return r * r * ??? ;
}
```

関数テンプレートを使うという手がある。

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

template < >
Real pi()
{
    return Real("3.14159265358979323846264338327 9") ;
}
```

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

しかし、この場合引数は何もないのに関数呼び出しのための () が必要だ。
変数テンプレートを使うと以下のように書ける。

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

template < >
Real pi<Real>("3.14159265358979323846264338327 9") ;

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

2.8.2　traits のラッパー

値を返す traits で値を得るには ::value と書かなければならない。

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

C++14 では std::integral_constant に constexpr operator bool が追加されたので、以下のようにも書ける。

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

しかしまだ面倒だ。変数テンプレートを使うと traits の記述が楽になる。

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

is_pointer_v<int> ;
```

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

C++ の標準ライブラリでは従来の traits ライブラリを変数テンプレートでラップした _v 版を用意している。

機能テストマクロは __cpp_variable_templates, 値は 201304。

2.9 constexpr 関数の制限緩和

C++11 で追加された constexpr 関数はとても制限が強い。constexpr 関数の本体には実質 return 文 1 つしか書けない。

C++14 では、ほとんど何でも書けるようになった。

```
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 ;
}
```

機能テストマクロは __cpp_constexpr, 値は 201304。

C++11 の constexpr 関数に対応しているが C++14 の constexpr 関数に対応していない C++ 実装では、__cpp_constexpr マクロの値は 200704 になる。

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 ではメンバー初期化子とアグリゲート初期化が組み合わせられるようになった。

メンバー初期化子とはクラスの非 static データメンバーを = で初期化できる C++11 の機能だ。

```
struct S
```

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

アグリゲート初期化とはアグリゲートの条件を満たす型をリスト初期化で初期化できる C++11 の機能だ。

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

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

C++11 ではメンバー初期化子を持つクラスはアグリゲート型の条件を満たさないのでアグリゲート初期化ができない。

C++14 では、この制限が緩和された。

```
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
```

アグリゲート初期化で、メンバー初期化子を持つ非 static データメンバーに対応する値がある場合はアグリゲート初期化が優先される。省略された場合はメンバー初期化子で初期化される。アグリゲート初期化でもメンバー初期化子でも明示的に初期化されていない非 static データメンバーは空の初期化リストで初期化された場合と同じになる。

機能テストマクロは __cpp_aggregate_nsdmi, 値は 201304。

2.11 サイズ付き解放関数

C++14 では operator delete のオーバーロードに、解放すべきストレージのサイズを取得できるオーバーロードが追加された。

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

第二引数は std::size_t 型で、第一引数で指定されたポインターが指す解放すべきストレージのサイズが与えられる。

たとえば以下のように使える。

```
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) ;
}
```

機能テストマクロは __cpp_sized_deallocation, 値は 201309。

第 3 章
C++17 のコア言語の新機能

C++14 の新機能のおさらいが終わったところで、いよいよ C++17 のコア言語の新機能を解説していく。

C++17 のコア言語の新機能には、C++11 ほどの大きなものはない。

3.1 トライグラフの廃止

C++17 ではトライグラフが廃止された。

トライグラフを知らない読者はこの変更を気にする必要はない。トライグラフを知っている読者はなおさら気にする必要はない。

3.2 16 進数浮動小数点数リテラル

C++17 では浮動小数点数リテラルに 16 進数を使うことができるようになった。

16 進数浮動小数点数リテラルは、プレフィクス `0x` に続いて仮数部を 16 進数（`0123456789abcdefABCDEF`）で書き、`p` もしくは `P` に続けて指数部を 10 進数で書く。

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

指数部は `e` ではなく `p` か `P` を使う。

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

16 進数浮動小数点数リテラルでは、指数部を省略できない。

```
int a = 0x1 ; // 整数□□□□
```

```
0x1.0 ; // □□□、指数部□□□
```

指数部は 10 進数で記述する。16 進数浮動小数点数リテラルは仮数部に 2 の指数部乗を掛けた値になる。つまり、

```
0xNpM
```

という浮動小数点数リテラルの値は

$$N \times 2^M$$

となる。

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

16 進数浮動小数点数リテラルには浮動小数点数サフィックスを記述できる。

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

16 進数浮動小数点数リテラルは、浮動小数点数が表現方法の詳細を知っている環境（たとえば IEEE–754）で、正確な浮動小数点数の表現が記述できるようになる。

機能テストマクロは `__cpp_hex_float`, 値は 201603。

3.3　UTF-8 文字リテラル

C++17 では UTF-8 文字リテラルが追加された。

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

UTF-8 文字リテラルは文字リテラルにプレフィクス `u8` を付ける。UTF-8 文字リテラルは UTF-8 のコード単位 1 つで表現できる文字を扱うことができる。UCS の規格としては、C0 制御文字と基本ラテン文字 Unicode ブロックが該当する。UTF-8 文字リテラルに書かれた文字が複数の UTF-8 コード単位を必要とする場合はエラーとなる。

```
// □□□
```

```
// U+3042 は UTF-8 は 0xE3, 0x81, 0x82 という 3 つのコード単位で表現する必要が
// あるため
u8'あ' ;
```

機能テストマクロはない。

3.4 関数型としての例外指定

C++17 では例外指定が関数型に組み込まれた。

例外指定とは noexcept のことだ。noexcept と noexcept(true) が指定された関数は例外を外に投げない。

C++14 ではこの例外指定は型システムに入っていなかった。そのため、無例外指定の付いた関数へのポインター型は型システムで無例外を保証することができなかった。

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

int main()
{
    // 無例外指定の付いたポインター
    void (*p)() noexcept = &f ;

    // 無例外指定があるにもかかわらず例外を投げる
    p() ;
}
```

C++17 では例外指定が型システムに組み込まれた。例外指定のある関数型を例外指定のない関数へのポインター型に変換することはできる。逆はできない。

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

// OK
// p1, &f は例外指定のない関数へのポインター型
void (*p1)() = &f ;
```

```
// OK
// 例外指定のある関数へのポインター型&g を例外指定のない関数へのポインター型p2
// に変換できる
void (*p2)() = &g ; // OK

// エラー
// 例外指定のない関数へのポインター型&f は例外指定のある関数へのポインター型p3
// に変換できない
void (*p3)() noexcept = &f ;

// OK
// p4, &g は例外指定のある関数へのポインター型
void (*p4)() noexcept = &g ;
```

機能テストマクロは __cpp_noexcept_function_type, 値は 201510。

3.5 fold 式

C++17 には fold 式が入った。fold は元は数学の概念で畳み込みとも呼ばれている。

C++ における fold 式とはパラメーターパックの中身に二項演算子を適用するための式だ。

今、可変長テンプレートを使って受け取った値をすべて加算した合計を返す関数 sum を書きたいとする。

```
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
}
```

このような関数テンプレート sum は以下のように実装することができる。

```
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 式の文法は以下のいずれかになる。

```
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 には未展開のパラメーターパックが入っていなければならない。

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) )  ;
}
```

これは f(args) というパターンが展開される。
fold-operator には以下のいずれかの二項演算子を使うことができる。

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

fold 式には左 fold と右 fold がある。

左 fold 式の (... op pack) では、展開結果は (((pack#0 op pack#1) op pack #2) ... op pack#N-1)となる。右 fold 式の (pack op ...) では、展開結果は (pack#0 op (pack#1 op (pack#2 op (... op pack#N-1))))となる。

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

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

浮動小数点数のような交換法則を満たさない型に fold 式を適用する際には注意が必要だ。

二項 fold 式の文法は以下のいずれかになる。

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

左右の cast-expression のどちらか片方だけに未展開のパラメーターパックが入っていなければならない。2 つの fold-operator は同じ演算子でなければならない。

(e1 op1 ... op2 e2) という二項 fold 式があったとき、e1 にパラメーターパックがある場合は二項右 fold 式、e2 にパラメーターパックがある場合は二項左 fold 式になる。

```
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 はポインターのキャプチャー
        // data は this ポインターをたどる
        [this]{ data = 1 ; }() ;

        // this->data は 1

        // エラー、*this はコピーされている
        // クロージャーオブジェクトのコピーキャプチャーされた 変数は
        // デフォルトで変更できない
        [*this]{ data = 2 ; } () ;

        // OK、mutable を使っている

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

        // this->data は 1
        // 変更されたのはコピーされたクロージャーオブジェクト 内の*this
   }
} ;
```

最初のラムダ式で生成されるクロージャーオブジェクトは以下のようなものだ。

```
class closure_object
{
    X * this_ptr ;

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

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

2 番目のラムダ式では以下のようなクロージャーオブジェクトが生成される。

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

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

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

これは C++ の文法に従っていないのでやや苦しいコード例だが、コピーキャプチャーされた値を変更しようとしているためエラーとなる。

3 番目のラムダ式では以下のようなクロージャーオブジェクトが生成される。

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

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

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

ラムダ式に mutable が付いているのでコピーキャプチャーされた値も変更できる。

*this をコピーキャプチャーした場合、this キーワードはコピーされたオブジェクトへのポインターになる。

```
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() ;
}
```

なぜならば、これを許してしまうとテンプレート仮引数に対して任意の式や文がテンプレートの Substitution に失敗するかどうかをチェックできてしまうからだ。

上級者向けの解説終わり。

機能テストマクロは __cpp_constexpr, 値は 201603。

__cpp_constexpr マクロの値は、C++11 の時点で 200704、C++14 の時点で 201304 だ。

3.8 文字列なし static_assert

C++17 では static_assert に文字列リテラルを取らないものが追加された。

```
static_assert( true ) ;
```

C++11 で追加された static_assert には、文字列リテラルが必須だった。

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

特に文字列を指定する必要がない場合もあるので、文字列リテラルを取らない static_assert が追加された。

機能テストマクロは __cpp_static_assert, 値は 201411。

C++11 の時点で __cpp_static_assert の値は 200410。

3.9 ネストされた名前空間定義

C++17 ではネストされた名前空間の定義を楽に書ける。

ネストされた名前空間とは、A::B::C のように名前空間の中に名前空間が入っている名前空間のことだ。

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

```
}
```

C++17 では、上記のコードと同じことを以下のように書ける。

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

機能テストマクロは `__cpp_nested_namespace_definitions`, 値は 201411。

3.10　[[fallthrough]] 属性

`[[fallthrough]]` 属性は `switch` 文の中の `case` ラベルを突き抜けるというヒントを出すのに使える。

`switch` 文では対応する `case` ラベルに処理が移る。通常、以下のように書く。

```
void f( int x )
{
    switch ( x )
    {
    case 0 :
        // 処理 0
        break ;
    case 1 :
        // 処理 1
        break ;
    case 2 :
        // 処理 2
        break ;
    default :
        // x がい☒れ☒もな☒場合☒処理
        break ;
    }
}
```

この例を以下のように書くと

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

x が 1 のときは処理 1 を実行した後に、処理 2 も実行される。switch 文を書くときはこのような誤りを書いてしまうことがある。そのため、賢い C++ コンパイラーは switch 文の case ラベルで break 文や return 文などで処理が終わらず、次の case ラベルや default ラベルに処理に突き抜けるコードを発見すると、警告メッセージを出す。

しかし、プログラマーの意図がまさに突き抜けて処理してほしい場合、警告メッセージは誤った警告となってしまう。そのような警告メッセージを抑制するため、またコード中に処理が突き抜けるという意図をわかりやすく記述するために、[[fallthrough]] 属性が追加された。

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

[[fallthrough]] 属性を書くと、C++ コンパイラーは処理がその先に突き抜けることがわかるので、誤った警告メッセージを抑制できる。また、他人がコードを読むときに意図が明らかになる。

機能テストマクロは __has_cpp_attribute(fallthrough), 値は 201603。

3.11 [[nodiscard]] 属性

[[nodiscard]] 属性は関数の戻り値が無視されてほしくないときに使うことができる。[[nodiscard]] 属性が付与された関数の戻り値を無視すると警告メッセージが表示される。

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

void g( int ) { }

int main()
{
    // □□□、戻□値□無視□□□□□
    f() ;
```

```
    // OK、戻り値は無視されていない
    int result = f() ;
    g( f ) ;
    f() + 1 ;
    (void) f() ;
}
```

戻り値を無視する、というのは万能ではない。上の例でも、意味的には戻り値は無視されていると言えるが、コンパイラーはこの場合に戻り値が無視されているとは考えない。

[[nodiscard]] の目的は、戻り値を無視してほしくない関数をユーザーが利用したときの初歩的な間違いを防ぐためにある。void 型にキャストするような意図的な戻り値の無視まで防ぐようには作られていない。

[[nodiscard]] 属性を使うべき関数は、戻り値を無視してほしくない関数だ。どのような関数が戻り値を無視してほしくないかというと大きく 2 つある。

戻り値をエラーなどのユーザーが確認しなければならない情報の通知に使う関数。

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

// 失敗するかもしれない処理
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 ;
}

// エラーがいっさい発生しなかったときの処理
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 ) ;
}
```

ここでは reserved という名前はどこからも使われていない。一見すると不必要な名前に見える。優秀な C++ コンパイラーはこのようなどこからも使われていない名前に対して「どこからも使われていない」という警告メッセージを出す。

しかし、コンパイラーから見えているソースコードがプログラムのすべてではない。さまざまな理由で reserved のような一見使われていない変数が必要になる。

たとえば、reserved はスタック破壊を検出するための領域かもしれない。プログラムは C++ 以外の言語で書かれたコードとリンクしていて、そこで使われるのかもしれない。あるいは OS や外部デバイスが読み書きするメモリーとして確保しているのかもしれない。

どのような理由にせよ、名前やエンティティが一見使われていないように見えるが存在が必要であるという意味を表すのに、[[maybe_unused]] 属性を使うことができる。これにより、C++ コンパイラーの「未使用の名前」という警告メッセージを抑制できる。

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

[[maybe_unused]] 属性を適用できる名前とエンティティの宣言は、クラス、typedef 名、変数、非 static データメンバー、関数、enum, enumerator だ。

```
// クラス
class [[maybe_unused]] class_name
{
// 非 static データメンバー
    [[maybe_unused]] int non_static_data_member ;

} ;

// typedef 名
// どちらでもよい
[[maybe_unused]] typedef int typedef_name1 ;
typedef int typedef_name2 [[maybe_unused]] ;

// エイリアス宣言による 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() ;
}
```

この場合、runtime_value が true の場合は関数 do_true_thing が呼ばれ、false の場合は関数 do_false_thing が呼ばれる。

実行時の条件分岐の条件には、コンパイル時定数を指定できる。

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

この場合、賢いコンパイラーは以下のように処理を最適化するかもしれない。

```
do_true_thing() ;
```

なぜならば、条件は常に true だからだ。このような最適化は実行時の条件分岐でもコンパイル時に行える。コンパイル時の条件分岐はこのような最適化が目的ではない。

もう一度コード例に戻ろう。今度は完全なコードを見てみよう。

```
// 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() ; // □□□
}
```

このコードはエラーになる。その理由は、do_false_thing という名前が宣言されていないからだ。C++ コンパイラーは、コンパイル時にコードを以下の形に変形することで最適化することはできるが、

```
void do_true_thing() ;

void f( bool runtime_value )
```

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

最適化の結果失われたものも、依然としてコンパイル時にコードとして検証はされる。コードとして検証されるということは、コードとして誤りがあればエラーとなる。名前 do_false_thing は宣言されていないのでエラーとなる。

3.14.2　プリプロセス時の条件分岐

C++ が C 言語から受け継いだ C プリプロセッサーには、プリプロセス時の条件分岐の機能がある。

```
// 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
}
```

このコードは、プリプロセスの結果、以下のように変換される。

```
void do_true_thing() ;

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

この結果、プリプロセス時の条件分岐では、選択されない分岐はコンパイルされないので、コンパイルエラーになるコードも書くことができる。

プリプロセス時の条件分岐は、条件が整数とか bool 型のリテラルか、リテラルに比較演算子を適用した結果ではうまくいく。しかし、プリプロセス時とはコンパイル時ではないので、コンパイル時計算はできない。

```
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 )
```

```
{
    // ランダムアクセスイテレーターかどうか確認する
    if ( is_random_access_iterator<Iterator> )
    {// ランダムアクセスイテレーターなので速い方法を使う
        return last - first ;
    }
    else
    { // ランダムアクセスイテレーターではないので遅い方法を使う
        typename std::iterator_traits<Iterator>::difference_type n = 0 ;

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

        return n ;
    }
}
```

残念ながら、このコードは動かない。ランダムアクセスイテレーターではないイテレーターを渡すと、`last - first` というコードがコンパイルされるので 、コンパイルエラーになる。コンパイラーは、

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

という部分について、`is_random_access_iterator<Iterator>` の値はコンパイル時に計算できるので、最終的なコード生成の結果としては、`if (true)` か `if (false)` になると判断できる。したがってコンパイラーは選択されない分岐のコード生成を行わないことはできる。しかしコンパイルはするので、コンパイルエラーになる。

`constexpr if` を使うと、選択されない部分の分岐はコンパイルエラーであってもコンパイルエラーとはならなくなる。

```
// distance
template < typename Iterator >
constexpr typename std::iterator_traits<Iterator>::difference_type
distance( Iterator first, Iterator last )
{
    // ランダムアクセスイテレーターかどうか確認する
    if constexpr ( is_random_access_iterator<Iterator> )
    {// ランダムアクセスイテレーターなので速い方法を使う
```

```
        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) では、return x が discarded statement となるため実体化されない。X は int 型に暗黙に変換できないが問題がなくなる。f(0) では return x.get() が discarded statement となるため実体化されない。int 型にはメンバー関数 get はないが問題はなくなる。

discarded statement は実体化されないだけで、もちろんテンプレートのエンティティの一部だ。discarded statement がテンプレートのコードとして文法的、意味的に正しくない場合は、もちろんコンパイルエラーとなる。

```
template < typename T >
void f( T x )
{
    // エラー、名前 g は宣言されていない
    if constexpr ( false )
        g() ;

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

何度も説明しているように、constexpr if はテンプレートの実体化を条件付きで抑制するだけだ。条件付きコンパイルではない。

```
template < typename T >
void f()
{
    if constexpr ( std::is_same_v<T, int> )
    {
        // 常に□□□□ル□□ー
        static_assert( false ) ;
    }
}
```

このコードは常にコンパイルエラーになる。なぜならば、static_assert(false) はテンプレートに依存しておらず、テンプレートの宣言を解釈するときに、依存名ではないから、そのまま解釈される。

このようなことをしたければ、最初から static_assert のオペランドに式を書けば

よい。

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

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

もし、どうしても constexpr 文の条件に合うときにだけ static_assert が使いたい場合もある。これは、constexpr if をネストしたりしていて、その内容を全部 static_assert に書くのが冗長な場合だ。

```
template < typename T >
void f()
{
    if constexpr ( E1 )
        if constexpr ( E2 )
            if constexpr ( E3 )
            {
                // E1 && E2 && E3 のときにコンパイルエラーにしたい
                // 実際には常にコンパイルエラー
                static_assert( false ) ;
            }
}
```

現実には、E1, E2, E3 は複雑な式なので、static_assert(E1 && E2 && E3) と書くのは冗長だ。同じ内容を二度書くのは間違いの元だ。

このような場合、static_assert のオペランドをテンプレート引数に依存するようにすると、constexpr if の条件に合うときにだけ発動する static_assert が書ける。

```
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> ) ;
            }
}
```

このように false_v を使うことで、static_assert をテンプレート引数 T に依存させる。その結果、static_assert の発動をテンプレートの実体化まで遅延させることができる。

constexpr if は非テンプレートコードでも 書くことができるが、その場合は普通の if 文と同じだ。

3.14.5 constexpr if では解決できない問題

constexpr if は条件付きコンパイルではなく、条件付きテンプレート実体化の抑制なので、最初の問題の解決には使えない。たとえば以下のコードはエラーになる。

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

// do_false_thing の宣言は存在しない

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

理由は、名前 do_false_thing は非依存名なのでテンプレートの宣言時に解決されるからだ。

3.14.6 constexpr if で解決できる問題

constexpr if は依存名が関わる場合で、テンプレートの実体化がエラーになる場合に、実体化を抑制させることができる。

たとえば、特定の型に対して特別な操作をしたい場合

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

```
};

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

    int value{} ;

    // T の型□ X な□ば特別な処理を行□たい
    if constexpr ( std::is_same<T, X>{} )
    {
        value = t.get_value() ;
    }
    else
    {
        value = static_cast<int>(t) ;
    }
}
```

もし constexpr if がなければ、T の型が X ではないときも t.get_value() という式が実体化され、エラーとなる。

再帰的なテンプレートの特殊化をやめさせたいとき

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

もし constexpr if がなければ、factorial<N-1> が永遠に実体化されコンパイル時ループが停止しない。

機能テストマクロは __cpp_if_constexpr, 値は 201606。

3.15　初期化文付き条件文

C++17 では、条件文に初期化文を記述できるようになった。

```
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 ) ;
}
```

```
// これ以降 file は使わない

auto int_ptr = std::make_unique<int>(42) ;
if ( ptr )
{
    // 処理
}
// これ以降 int_ptr は使わない
```

上記のコードには問題がある。これ以降変数は使わないが、変数自体は使えるからだ。

```
auto ptr = std::make_unique<int>(42) ;
if ( ptr )
{
    // 処理
}
// これ以降 ptr は使わない

// でも使える
int value = *ptr ;
```

変数を使えないようにするには、ブロックスコープで囲むことで、変数をスコープから外してやればよい。

```
{
    auto int_ptr = std::make_unique<int>(42) ;
    if ( ptr )
    {
        // 処理
    }
    // ptr は破棄される
}
// これ以降 ptr は使わないし使えない
```

このようなパターンは頻出するので、初期化文付きの条件文が追加された。

```
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 ではクラステンプレートのコンストラクターの実引数からテンプレート実引数の推定が行えるようになった。

```
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 *>
}
```

これは関数テンプレートが実引数からテンプレート実引数の推定が行えるのと同じだ。

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

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

3.16.1 推定ガイド

クラステンプレートのコンストラクターからの実引数は便利だが、クラスのコンストラクターはクラステンプレートのテンプレートパラメーターに一致しない場合もある。そのような場合はそのままでは実引数推定ができない。

```
// コンテナー風のクラス
template < typename T >
class Container
```

```
{
    std::vector<T> c ;
public :
    // 初期化にイテレーターのペアを取る
    // Iterator は T ではない
    // T は推定できない
    template < typename Iterator >
    Container( Iterator first, Iterator last )
        : c( first, last )
    { }
} ;


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

    // エラー
    // T を推定できない
    Container c( std::begin(a), std::end(a) ) ;
}
```

このため、C++17 には推定ガイドという機能が提供されている。

```
テンプレート名( 引数リスト ) -> テンプレート id ;
```

これを使うと、以下のように書ける。

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

C++ コンパイラーはこの推定ガイドを使って、`Container<T>::Container( Iterator, Iterator)`からは、`T` を `std::iterator_traits<Iterator>::value_type` として推定すればいいのだと判断できる。

たとえば、初期化リストに対応するには以下のように書く。

```
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++ コンパイラーはこの推定ガイドから、`Container<T>::Container( std::initializer_list<T> )`の場合は `T` を `T` として推定すればよいことがわかる。

機能テストマクロは `__cpp_deduction_guides`, 値は 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 では非型テンプレートパラメーターの宣言に `auto` を使うことができるようになった。

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

void f() { }

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

これは C++14 までであれば、以下のように書かなければならなかった。

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

```
void f() { }

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

機能テストマクロは __cpp_template_auto, 値は 201606。

3.18　using 属性名前空間

C++17 では、属性名前空間に using ディレクティブのような記述ができるようになった。

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

属性トークンには、属性名前空間を付けることができる。これにより、独自拡張の属性トークンの名前の衝突を避けることができる。

たとえば、ある C++ コンパイラーには独自拡張として foo, bar という属性トークンがあり、別の C++ コンパイラーも同じく独自拡張として foo, bar という属性トークンを持っているが、それぞれ意味が違っている場合、コードの意味も違ってしまう。

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

このため、C++ には属性名前空間という文法が用意されている。注意深い C++ コンパイラーは独自拡張の属性トークンには属性名前空間を設定していることだろう。

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

問題は、これをいちいち記述するのは面倒だということだ。

C++17 では、using 属性名前空間という機能により、using ディレクティブのような名前空間の省略が可能になった。文法は using ディレクティブと似ていて、属性の中で using name : ... と書くことで、コロンに続く属性トークンに、属性名前空間 name を付けたものと同じ効果が得られる。

3.19 非標準属性の無視

C++17 では、非標準の属性トークンは無視される。

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

属性は C++ コンパイラーによる独自拡張を C++ の規格に準拠する形で穏便に追加するための機能だ。その属性のためにコンパイルエラーになった場合、けっきょく C プリプロセッサーを使うか、わずらわしさから独自の文法が使われてしまう。そのためこの機能は必須だ。

3.20 構造化束縛

C++17 で追加された構造化束縛は多値を分解して受け取るための変数宣言の文法だ。

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

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

C++ では、さまざまな方法で多値を扱うことができる。たとえば配列、クラス、tuple, pair だ。

```
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 ;
}
```

構造化束縛を使うと、以下のように書ける。

```
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 ;
}
```

変数の型はそれぞれ対応する多値の型になる。この場合、a, b, c はそれぞれ int, double, std::string 型になる。

tuple だけではなく、pair も使える。

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

    auto [a,b] = p ;

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

構造化束縛は if 文と switch 文、for 文でも使える。

```
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 ;
}
```

構造化束縛でクラスを使う場合は、非 static データメンバーはすべて 1 つのクラスの public なメンバーでなければならない。

構造化束縛は constexpr にはできない。

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

    // エ□□
    constexpr auto [a,b] = expr ;
}
```

3.20.1 超上級者向け解説

構造化束縛は、変数の宣言のうち、構造化束縛宣言 (structured binding declaration) に分類される文法で記述する。構造化束縛宣言となる宣言は、単純宣言（simple-declaration）と for-range 宣言（for-range-declaration）のうち、[識別子リスト] があるものだ。

```
単純宣言:
    属性 auto CV 修飾子（省略可） リファレンス修飾子（省略可）
    [ 識別子リスト ] 初期化子 ;

for-range 宣言:
    属性 auto CV 修飾子（省略可） リ□ァレ□ス修飾子（省略可）
    [ 識別子□□ト ] ;

識別子□ス□:
```

□□□□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

構造化束縛の構造化束縛宣言は以下のように解釈される。

構造化束縛宣言によって宣言される変数の数は、初期化子の多値の数と一致していなければならない。

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

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

構造化束縛宣言で宣言されるそれぞれの変数名について、記述されたとおりの属性、CV 修飾子、リファレンス修飾子の変数が宣言される。

3.20.3 初期化子の型が配列の場合

初期化子が配列の場合、それぞれの変数はそれぞれの配列の要素で初期化される。

リファレンス修飾子がない場合、それぞれの変数はコピー初期化される。

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

これは、以下と同じ意味になる。

```
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

上記以外の場合、構造化束縛宣言の初期化子の型 E はクラス型で、すべての非 static データメンバーは public の直接のメンバーであるか、あるいは単一の曖昧ではない public 基本クラスのメンバーである必要がある。E に匿名 union メンバーがあってはならない。

以下は型 E として適切なクラスの例である。

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

struct B : A { } ;
```

以下は型 E として不適切なクラスの例である。

```
// public 以外の非 static データメンバーがある
struct A
{
public :
    int a ;
private :
    int b ;
} ;


struct B
{
    int a ;
} ;
// クラスにも基本クラスにも非static データメンバーがある
struct C : B
{
    int b ;
} ;

// 匿名 union メンバーがある
struct D
```

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

型 E の非 static データメンバーは宣言された順番で多値として認識される。
以下のコードは、

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

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

以下のコードと意味的に等しい。

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

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

構造化束縛はビットフィールドに対応している。

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

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

機能テストマクロは __cpp_structured_bindings, 値は 201606。

3.21 inline 変数

C++17 では変数に inline キーワードを指定できるようになった。

```
inline int variable ;
```

このような変数を inline 変数と呼ぶ。その意味は inline 関数と同じだ。

3.21.1 inline の歴史的な意味

今は昔、本書執筆から 30 年以上は昔に、inline キーワードが C++ に追加された。

inline の現在の意味は誤解されている。

inline 関数の意味は、「関数を強制的にインライン展開させるための機能」ではない。

大事なことなのでもう一度書くが、inline 関数の意味は、「関数を強制的にインライン展開させるための機能」ではない。

確かに、かつて inline 関数の意味は、関数を強制的にインライン展開させるための機能だった。

関数のインライン展開とは、たとえば以下のようなコードがあったとき、

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

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

    // a と b のうち小さい方を選ぶ
    int value = min( a, b ) ;
}
```

この関数 min は十分に小さく、関数呼び出しのコストは無視できないオーバーヘッドになるため、以下のような最適化が考えられる。

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

```
    int value = a < b ? a : b ;
}
```

このように関数の中身を展開することを、関数のインライン展開という。

人間が関数のインライン展開を手で行うのは面倒だ。それにコードが読みにくい。“min(a,b)” と “a<b?a:b” のどちらが読みやすいだろうか。

幸い、C++ コンパイラーはインライン展開を自動的に行えるので人間が苦労する必要はない。

インライン展開は万能の最適化ではない。インライン展開をすると逆に遅くなる場合もある。

たとえば、ある関数をコンパイルした結果のコードサイズが 1K バイトあったとして、その関数を呼んでいる箇所がプログラム中に 1000 件ある場合、プログラム全体のサイズは 1M バイト増える。コードサイズが増えるということは、CPU のキャッシュを圧迫する。

たとえば、ある関数の実行時間が関数呼び出しの実行時間に比べて桁違いに長いとき、関数呼び出しのコストを削減するのは意味がない。

したがって関数のインライン展開という最適化を適用すべきかどうかを決定するには、関数のコードサイズが十分に小さいとき、関数の実行時間が十分に短いとき、タイトなループの中など、さまざまな条件を考慮しなければならない。

昔のコンパイラー技術が未熟だった時代の C++ コンパイラーは関数をインライン展開するべきかどうかの判断ができなかった。そのため `inline` キーワードが追加された。インライン展開してほしい関数を `inline` 関数にすることで、コンパイラーはその関数がインライン展開するべき関数だと認識する。

3.21.2　現代の inline の意味

現代では、コンパイラー技術の発展により C++ コンパイラーは十分に賢くなったので、関数をインライン展開させる目的で `inline` キーワードを使う必要はない。実際、現代の C++ コンパイラーでは `inline` キーワードはインライン展開を強制しない。関数をインライン展開すべきかどうかはコンパイラーが判断できる。

`inline` キーワードにはインライン展開以外に、もう 1 つの意味がある。ODR (One Definition Rule、定義は 1 つの原則) の回避だ。

C++ では、定義はプログラム中に 1 つしか書くことができない。

```
void f() ; // OK、宣言
void f() ; // OK、再宣言
```

```
void f() { } // OK、定義

void f() { } // □□□、再定義
```

通常は、関数を使う場合には宣言だけを書いて使う。定義はどこか 1 つの翻訳単位に書いておけばよい。

```
// f.h

void f() ;

// f.cpp

void f() { }

// main.cpp

#include "f.h"

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

しかし、関数のインライン展開をするには、コンパイラーの実装上の都合で、関数の定義が同じ翻訳単位になければならない。

```
inline void f() ;

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

しかし、翻訳単位ごとに定義すると、定義が重複して ODR に違反する。

C++ ではこの問題を解決するために、inline 関数は定義が同一であれば、複数の翻訳単位で定義されてもよいことにしている。つまり ODR に違反しない。

```
// 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
}
```

この例で関数 a, b の中の変数 data は同じ変数を指している。変数 data は static ストレージ上に構築された変数なのでプログラムの開始時にゼロで初期化される。2 回インクリメントされるので 値は 2 となる。

これにより、クラスの非 static データメンバーの定義を書かなくてすむようになる。

C++17 以前の C++ では、以下のように書かなければならなかったが、

```
// S.h

struct S
{
    static int data ;
} ;

// S.cpp

int S::data ;
```

C++17 では、以下のように書けばよい。

```
// S.h

struct S
{
    inline static int data ;
} ;
```

S.cpp に変数 S::data の定義を書く必要はない。

機能テストマクロは __cpp_inline_variables, 値は 201606。

3.22 可変長 using 宣言

この機能は超上級者向けだ。

C++17 では using 宣言をコンマで区切ることができるようになった。

```
int x, y ;

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

これは、C++14 で

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

と書くのと等しい。

C++17 では、using 宣言でパック展開ができるようになった。この機能に正式な名前は付いていないが、可変長 using 宣言（Variadic using declaration）と呼ぶのがわかりやすい。

```
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 は一部の演算子がオーバーロードされているので、通常の整数型のように使うことができる。ただし、バイトをビット列演算するのに使う一部の演算子だけだ。

具体的には、以下に示すシフト、ビット OR, ビット列 AND, ビット列 XOR, ビット列 NOT だ。

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

四則演算などの演算子はサポートしていない。

std::byte ☒ std::to_intenger<IntType>(std::byte) により、IntType 型の整数型に変換できる。

```
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 では型安全に値を格納するライブラリとして、variant, any, optional が追加された。

4.1 variant : 型安全な union

4.1.1 使い方

ヘッダーファイル <variant> で定義されている variant は、型安全な union として使うことができる。

```
#include <variant>

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

    // int, double, std::string のいずれかを格納する variant
    // コンストラクターは最初の型をデフォルト構築
    std::variant< int, double, std::string > x ;

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

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

```
    // 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) ☒

$$\texttt{sizeof(U)} = \max\{\texttt{sizeof(int)}, \texttt{sizeof(double)}, \texttt{sizeof(std::string)}\} + ☒☒☒☒☒☒☒☒$$

☒☒☒。sizeof(S) ☒、

$$\texttt{sizeof(S)} = \texttt{sizeof(int)} + \texttt{sizeof(double)} + \texttt{sizeof(std::string)} + ☒☒☒☒☒☒☒☒$$

☒☒☒。

union ☒☒☒☒☒効率☒☒☒。union ☒ variant ☒違☒型非安全☒。☒☒型☒値☒保持☒☒☒☒☒☒☒☒情報☒保持☒☒☒☒☒、☒☒☒☒☒適切☒管理☒☒☒☒☒☒☒☒。

試☒☒、冒頭☒☒☒☒☒ union ☒書☒☒、以下☒☒☒☒☒☒。

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

    // コンストラクター
    // int 型をデフォルト初期化する
    U() : i{} { }
    // デストラクター
    // 何もしない。オブジェクトの破棄はユーザーの責任に任せる
    ~U() { }
} ;

// デストラクター呼び出し
template < typename T >
void destruct ( T & x )
{
    x.~T() ;
}

int main()
{
    U u ;

    // 基本型はそのまま代入できる
    // 破棄も考えなくてよい
    u.i = 0 ;
    u.d = 0.0 ;

    // 非トリビアルなコンストラクターを持つ型
    // placement new が必要
    new(&u.s) std::string("hello") ;

    // ユーザーはどの型を入れたか別に管理しておく必要がある
    bool has_int = false ;
    bool has_string = true ;

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

    // 非トリビアルなデストラクターを持つ型
```

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

このようなコードは書きたくない。variant を使えば、このような面倒で冗長なコードを書かずに、型安全に union と同等機能を実現できる。

4.1.3 variant の宣言

variant はテンプレート実引数で保持したい型を与える。

```
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

デフォルト初期化

variant はデフォルト構築すると、最初に与えた型の値をデフォルト構築して保持する。

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

variant にデフォルト構築できない型を最初に与えると、variant もデフォルト構築できない。

```
// デフォルト構築できない型
struct non_default_constructible
{
    non_default_constructible() = delete ;
} ;

// エラー
// デフォルト構築できない
std::variant< non_default_constructible > v ;
```

デフォルト構築できない型だけを保持する variant をデフォルト構築するためには、最初の型をデフォルト構築可能な型にすればよい。

```
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> は T 型の variant の保持できる型のうち、I 番目の型をネストされた型名 type で返す。

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

// char
using t0 = std::variant_alternative< 0, type >::type ;
// short
using t1 = std::variant_alternative< 1, type >::type ;
// int
using t2 = std::variant_alternative< 2, type >::type ;
```

variant_alternative_t というテンプレートエイリアスが以下のように定義されている。

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

これを使えば、以下のようにも書ける。

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

// char
using t0 = std::variant_alternative_t< 0, type > ;
// short
using t1 = std::variant_alternative_t< 1, type > ;
// int
using t2 = std::variant_alternative_t< 2, type > ;
```

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

holds_alternative<T>(v) は、variant v が T 型の値を保持しているかどうかを確認する。保持しているのであれば true を、そうでなければ false を返す。

```
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

```
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 ) ;
}
```

このように、variant にどの型の値が保持されていても扱うことができる。

std::visit は以下のように宣言されている。

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

第一引数に関数オブジェクトを渡し、第二引数以降に variant を渡す。すると、vis(get<i>(vars)...) のように呼ばれる。

```
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

ヘッダーファイル <any> で定義されている std::any は、ほとんどどんな型の値でも保持できるクラスだ。

```
#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 の第一引数が T 型ではない場合、in_place_type 構築するオーバーロード関数が選ばれる。

```
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 は、文字型（char, wchar_t, char16_t, char32_t）の連続した配列で表現された文字列に対する共通の文字列ビューを提供する。文字列は所有しない。

5.1 使い方

連続した文字型の配列を使った文字列の表現方法にはさまざまある。C++ では最も基本的な文字列の表現方法として、null 終端された文字型の配列がある。

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

あるいは、文字型の配列と文字数で表現することもある。

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

このような表現をいちいち管理するのは面倒なので、クラスで包むこともある。

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

このように文字列を表現する方法はさまざまある。これらのすべてに対応していると、表現の数だけ関数のオーバーロードが追加されていくことになる。

```
// 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 のデフォルト構築は、空の string_view を作る。

```
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);
```

この string_view のコンストラクターは、null 終端された文字型へのポインターを受け取る。

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

5.4.3　文字型へのポインターと文字数

```
constexpr basic_string_view(const charT* str, size_type len);
```

この string_view のコンストラクターは、文字型の配列へのポインターと文字数を受け取る。ポインターは null 終端されていなくてもよい。

```
int main()
{
    char str[] = {'h', 'e', 'l', 'l', 'o'} ;

    std::string_view s( str, 5 ) ;
}
```

5.4.4 文字列クラスからの変換関数

他文字列 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) ;
}
```

このメンバー関数は既存の std::string にも追加されている。

5.6　ユーザー定義リテラル

std::string と std::string_view にはユーザー定義リテラルが追加されている。

```
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;
```

以下のように使う。

```
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

ヘッダーファイル <memory_resource> で定義されているメモリーリソースは、動的ストレージを確保するための C++17 で追加されたライブラリだ。その特徴は以下のとおり。

- アロケーターに変わる新しいインターフェースとしてのメモリーリソース
- ポリモーフィックな振る舞いを可能にするアロケーター
- 標準で提供されるさまざまな特性を持ったメモリーリソースの実装

6.1 メモリーリソース

メモリーリソースはアロケーターに変わる新しいメモリー確保と解放のためのインターフェースとしての抽象クラスだ。コンパイル時に挙動を変える静的ポリモーフィズム設計のアロケーターと違い、メモリーリソースは実行時に挙動を変える動的ポリモーフィズム設計となっている。

```
void f( memory_resource * mem )
{
    // 10 バ□トの□ト□ー□□確保
    auto ptr = mem->allocate( 10 ) ;
    // 確保したストレージを解放
    mem->deallocate( ptr ) ;
}
```

クラス std::pmr::memory_resource の宣言は以下のとおり。

```
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;
};

}
```

クラス memory_resource は std::pmr 名前空間スコープの中にある。

6.1.1　メモリーリソースの使い方

memory_resource を使うのは簡単だ。memory_resource のオブジェクトを確保したら、メンバー関数 allocate(bytes, alignment) でストレージを確保する。メンバー関数 deallocate(p, bytes, alignment) でストレージを解放する。

```
void f( std::pmr::memory_resource * mem )
{
    // 100 バイトのストレージを確保
    void * ptr = mem->allocate( 100 ) ;
    // ストレージを解放
    mem->deallocate( ptr, 100 ) ;
}
```

2 つの memory_resource のオブジェクト a, b があるとき、一方のオブジェクトで確保したストレージをもう一方のオブジェクトで解放できるとき、a.is_equal(b) は true を返す。

```
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 は malloc でストレージを確保し、do_deallocate は free でストレージを解放する。メモリーリソースで 0 バイトのストレージを確保しようとしたときの規定はないので、malloc の挙動に任せる。malloc は 0 バイトのメモリーを確保しようとしたとき、C11 では規定がない。POSIX では null ポインターを返すか、free で解放可能な何らかのアドレスを返すものとしている。

do_is_equal は、malloc_resource でさえあればどのオブジェクトから確保されたストレージであっても解放できるので、*this が malloc_resource であるかどうかを dynamic_cast で確認している。

6.2 polymorphic_allocator：動的ポリモーフィズムを実現するアロケーター

std::pmr::polymorphic_allocator はメモリーリソースを動的ポリモーフィズムとして振る舞うアロケーターにするためのライブラリだ。

従来のアロケーターは、静的ポリモーフィズムを実現するために設計されていた。たとえば独自の custom_int_allocator 型を使いたい場合は以下のように書く。

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

コンパイル時に使うべきアロケーターが決定できる場合はこれでいいのだが、実行時にアロケーターを選択したい場合、アロケーターをテンプレート引数に取る設計は問題になる。

そのため、C++17 ではメモリーリソースをコンストラクター引数に取り、メモリーリソースからストレージを確保する実行時ポリモーフィックの振る舞いをする std::pmr::polymorphic_allocator が追加された。

たとえば、標準入力から true か false が入力されたかによって、システムのデフォルトのメモリーリソースと、monotonic_buffer_resource を実行時に切り替えるには、以下のように書ける。

```
int main()
{
```

```
    bool b;

    std::cin >> b ;

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

    if ( b )
    { // デフォルトのメモリーリソースを使う
        mem = std::pmr::get_default_resource() ;
    }
    else
    { // モノトニックバッファーを使う
        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 は以下のように宣言されている。

```
namespace std::pmr {

template <class T>
class polymorphic_allocator ;

}
```

テンプレート実引数には std::allocator<T> と同じく、確保する型を与える。

6.2.1 コンストラクター

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

std::pmr::polymorphic_allocator のデフォルトコンストラクターは、メモリーリソースを std::pmr::get_default_resource() で取得する。

memory_resource * を引数に取るコンストラクターは、渡されたメモリーリソースをストレージ確保に使う。polymorphic_allocator の生存期間中、メモリーリソース

へのポインターは妥当なものでなければならない。

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

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

後は通常のアロケーターと同じように振る舞う。

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 では、プログラム全体で使われるメモリーリソースへのポインターを取得することができる。

6.3.1 new_delete_resource()

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

関数 new_delete_resource はメモリーリソースへのポインターを返す。参照されるメモリーリソースは、ストレージの確保に ::operator new を使い、ストレージの解放に ::operator delete を使う。

```
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 はメモリーリソースへのポインターを返す。参照されるメモリーリソースの allocate は必ず失敗し、std::bad_alloc を throw する。deallocate は何もしない。

このメモリーリソースは、ストレージの確保に失敗した場合のコードをテストする目的で使える。

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 ;
```

デフォルト・メモリーリソース・ポインターとは、メモリーリソースを明示的に指定することができない場合に、システムがデフォルトで利用するメモリーリソースへのポインターのことだ。初期値は new_delete_resource() の戻り値となっている。

現在のデフォルト・メモリーリソース・ポインターと取得するためには、関数 get_default_resource を使う。デフォルト・メモリーリソース・ポインターを独自のメモリーリソースに差し替えるには、関数 set_default_resource を使う。

```
int main()
{
    // 現在のデフォルトのメモリーリソースへのポインター
    auto init_mem = std::pmr::get_default_resource() ;

    std::pmr::synchronized_pool_resource pool_mem ;

    // デフォルトのメモリーリソースを変更する
    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

標準ライブラリはメモリーリソースの実装として、プールリソースとモノトニックリソースを提供している。このメモリーリソースの詳細は後に解説するが、ここではそのための事前知識として、汎用的なメモリーアロケーター一般の解説をする。

プログラマーはメモリーを気軽に確保している。たとえば 47 バイトとか 151 バイトのような中途半端なサイズのメモリーを以下のように気軽に確保している。

```
int main()
{
```

```
    auto mem = std::get_default_resource() ;

    auto p1 = mem->allocate( 47 ) ;
    auto p2 = mem->allocate( 151 ) ;

    mem->deallocate( p1 ) ;
    mem->deallocate( p2 ) ;
}
```

しかし、残念ながら現実のハードウェアや OS のメモリー管理は、このように柔軟にはできていない。たとえば、あるアーキテクチャーと OS では、メモリーはページサイズと呼ばれる単位でしか確保できない。そして最小のページサイズですら 4K バイトであったりする。もしシステムの低級なメモリー管理を使って上のコードを実装しようとすると、47 バイト程度のメモリーを使うのに 3K バイト超の無駄が生じることになる。

他にもアライメントの問題がある。アーキテクチャーによってはメモリーアドレスが適切なアライメントに配置されていないとメモリーアクセスができないか、著しくパフォーマンスが落ちることがある。

malloc や operator new などのメモリーアロケーターは、低級なメモリー管理を隠匿し、小さなサイズのメモリー確保を効率的に行うための実装をしている。

一般的には、大きな連続したアドレス空間のメモリーを確保し、その中に管理用のデータ構造を作り、メモリーを必要なサイズに切り出す。

```
// 実装イメージ

// ストレージを分割して管理するためのリンクリストデータ構造
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 ; // ストレージの先頭へのポインター
    std::size_t size ; // ストレージのサイズ
    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□。

- プールリソースのオブジェクトが破棄されるとき、そのオブジェクトから allocate で確保したストレージは、明示的に deallocate を呼ばずとも解放される。

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

    // 確保したストレージは破棄される
}
```

- プールリソースの構築時に、上流メモリーリソースを与えることができる。プールリソースは上流メモリーリソースからチャンクのためのストレージを確保する。

```
int main()
{
    // get_default_resource() が使われる
    std::pmr::synchronized_pool_resource m1 ;

    // 独自の上流メモリーリソースを指定
    custom_memory_resource mem ;
    std::pmr::synchronized_pool_resource m2( &mem ) ;

}
```

- プールリソースはストレージを確保する上流メモリーリソースから、プールと呼ばれる複数のストレージを確保する。プールは複数のチャンクを保持している。チャンクは複数の同一サイズのブロックを保持している。プールリソースに対する do_allocate(size, alignment) は、少なくとも size バイトのブロックサイズのプールのいずれかのチャンクのブロックが割り当てられる。
もし、最大のブロックサイズを超えるサイズのストレージを確保しようとした場合、上流メモリーリソースから確保される。

```
// 実装イメージ

namespace std::pmr {

// チャンクの実装
```

```
template < size_t block_size >
class chunk
{
    blocks<block_size> b ;
}

// プールの実装
template < size_t block_size >
class pool : public memory_resource
{
    chunks<block_size> c ;
} ;

class pool_resource : public memory_resource
{
    // それぞれのブロックサイズのプール
    pool<8> pool_8bytes ;
    pool<16> pool_16bytes ;
    pool<32> pool_32bytes ;

    // 上流メモリーリソース
    memory_resource * mem ;


    virtual void * do_allocate( size_t bytes, size_t alignment ) override
    {
        // 対応するブロックサイズのプールにディスパッチ
        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
        // 最大ブロックサイズを超えたので上流メモリーリソースにディスパッチ
            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 はプールリソースの挙動を指定するためのクラスで、以下のように定義されている。

```
namespace std::pmr {

struct pool_options {
    size_t max_blocks_per_chunk = 0;
    size_t largest_required_pool_block = 0;
};

}
```

このクラスのオブジェクトをプールリソースのコンストラクターに与えることで、プールリソースの挙動を指定できる。ただし、pool_options による指定はあくまでも目安で、実装には従う義務はない。

max_blocks_per_chunk は、上流メモリーリソースからプールのチャンクを補充する際に一度に確保する最大のブロック数だ。この値がゼロか、実装の上限より大きい場合、実装の上限が使われる。実装は指定よりも小さい値を使うことができるし、またプールごとに別の値を使うこともできる。

largest_required_pool_block はプール機構によって確保される最大のストレージのサイズだ。この値より大きなサイズのストレージを確保しようとすると、上流メモリーストレージから直接確保される。この値がゼロか、実装の上限よりも大きい場合、実装の上限が使われる。実装は指定よりも大きい値を使うこともできる。

6.5.4 プールリソースのコンストラクター

プールリソースの根本的なコンストラクターは以下のとおり。synchronized と unsynchronized どちらも同じだ。

```
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 と memory_resource * を指定する。指定しない場合はデフォルト値が使われる。

6.5.5 プールリソースのメンバー関数

release()

```
void release();
```

確保したストレージすべてを解放する。たとえ明示的に deallocate を呼び出されていないストレージも解放する。

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

    // ptr は解放される
    mem.release() ;

}
```

upstream_resource()

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

構築時に渡した上流メモリーリソースへのポインターを返す。

options()

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

構築時に渡した pool_options オブジェクトと同じ値を返す。

6.6 モノトニックバッファーリソース

モノトニックバッファーリソースは C++17 で標準ライブラリに追加されたメモリーリソースの実装だ。クラス名は monotonic_buffer_resource。

モノトニックバッファーリソースは高速にメモリーを確保し、一気に解放するとい

う用途に特化した特殊な設計をしている。モノトニックバッファーリソースはメモリー解放をせず、メモリー使用量がモノトニックに増え続けるので、この名前が付いている。

たとえばゲームで 1 フレームを描画する際に大量に小さなオブジェクトのためのストレージを確保し、その後確保したストレージをすべて解放したい場合を考える。通常のメモリーアロケーターでは、メモリー片を解放するためにメモリー全体に構築されたデータ構造をたどり、データ構造を書き換えなければならない。この処理は高くつく。すべてのメモリー片を一斉に解放してよいのであれば、データ構造をいちいちたどったり書き換えたりする必要はない。メモリーの管理は、単にポインターだけでよい。

```
// 実装イメージ

namespace std::pmr {

class monotonic_buffer_resource : public memory_resource
{
    // 連続した長大なストレージの先頭へのポインター
    void * ptr ;
    // 現在の未使用ストレージの先頭へのポインター
    std::byte * current ;

    virtual void *
    do_allocate( size_t bytes, size_t alignment ) override
    {
        void * result = static_cast<void *>(current) ;
        current += bytes ; // 必要であればアライメント調整
        return result ;
    }

    virtual void
    do_deallocate( void * ptr, size_t bytes, size_t alignment ) override
    {
        // 何もしない
    }

public :
    ~monotonic_buffer_resource()
    {
        // ptr の解放
```

```
    }
} ;

}
```

このように、基本的な実装としては、`do_allocate`はポインターを加算して管理するだけだ。なぜならば解放処理がいらないため、個々のストレージ片を管理するためのデータ構造を構築する必要がない。`do_deallocate`は何もしない。デストラクターはストレージ全体を解放する。

6.6.1　アルゴリズム

モノトニックバッファーリソースは以下のような特徴を持つ。

- `deallocate`呼び出しは何もしない。メモリー使用量はリソースが破棄されるまでモノトニックに増え続ける。

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

    void * ptr = mem.allocate( 10 ) ;
    // 何もしない
    // ストレージは解放されない
    mem.deallocate( ptr ) ;

    // mem が破棄される際に確保したストレージはすべて破棄される
}
```

- メモリー確保に使う初期バッファーを与えることができる。ストレージ確保の際に、初期バッファーに空きがある場合はそこから確保する。空きがない場合は上流メモリーリソースからバッファーを確保して、バッファーから確保する。

```
int main()
{
    std::byte initial_buffer[10] ;
    std::pmr::monotonic_buffer_resource
        mem( initial_buffer, 10, std::pmr::get_default_resource() ) ;

    // 初期バッファーから確保
```

```
    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` は、上流メモリーリソースから最初に確保するバッファーのサイズ（初期サイズ）のヒントとなる。実装はこのサイズか、あるいは実装依存のサイズをバッファーとして確保する。

デフォルトコンストラクターは上流メモリーリソースに `std::pmr_get_default_resource()` を与えたのと同じ挙動になる。

`size_t` 1 つだけを取るコンストラクターは、初期サイズだけを与えて後はデフォルトの扱いになる。

初期バッファーを取るコンストラクターは以下のとおり。

```
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()) {}
```

初期バッファーは先頭アドレスを `void *` 型で渡し、そのサイズを `size_t` 型で渡す。

6.6.3　その他の操作

release()

```
void release() ;
```

メンバー関数 `release` は、上流リソースから確保されたストレージをすべて解放する。明示的に `deallocate` を呼び出していないストレージも解放される。

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

    mem.allocate( 10 ) ;

    // ストレージはすべて解放される
    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 ;
}
```

本書の執筆時点では、コンピューターはマルチコアが一般的になり、同時に複数のスレッドを実行できるようになった。さっそくこの処理を 2 つのスレッドで並列化してみよう。

```
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() ;
}
```

なるほど、とてもわかりにくいコードだ。

筆者のコンピューターの CPU は 2 つの物理コア、4 つの論理コアを持っているので、4 スレッドまで同時に並列実行できる。読者の使っているコンピューターは、より高性能でさらに多くのスレッドを同時に実行可能だろう。実行時に最大の効率を出すようにできるだけ頑張ってみよう。

```
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 ;
}
```

もうわけがわからない。

このような並列化をそれぞれのアルゴリズムに対して自前で実装するのは面倒だ。そこで、C++17 では標準で並列実行してくれる並列アルゴリズム（Parallelism）が追加された。

7.2 使い方

並列アルゴリズムは既存のアルゴリズムのオーバーロードとして追加されている。以下は既存のアルゴリズムである `all_of` の宣言だ。

```
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□□□

て、オブジェクトにアクセスする。そのような関数群を、要素アクセス関数（element access functions）と呼ぶ。

たとえば、std::sort は以下のような要素アクセス関数を持つ。

- テンプレート実引数で与えられたランダムアクセスイテレーター
- 要素に対する swap 関数の適用
- ユーザー提供された Compare 関数オブジェクト

並列アルゴリズムが使う要素アクセス関数は、並列実行に伴うさまざまな制約を満たさなければならない。

7.3.2 ユーザー提供する関数オブジェクトの制約

並列アルゴリズムのうち、テンプレートパラメーター名が、Predicate, BinaryPredicate, Compare, UnaryOperation, BinaryOperation, BinaryOperation1, BinaryOperation2 となっているものは、関数オブジェクトとしてユーザーがアルゴリズムに提供するものである。このようなユーザー提供の関数オブジェクトには、並列アルゴリズムに渡す際の制約がある。

- 実引数で与えられたオブジェクトを直接、間接に変更してはならない
- 実引数で与えられたオブジェクトの一意性に依存してはならない
- データ競合と同期

一部の特殊なアルゴリズムには例外もあるが、ほとんどの並列アルゴリズムではこの制約を満たさなければならない。

実引数で与えられたオブジェクトを直接、間接に変更してはならない

ユーザー提供の関数オブジェクトは実引数で与えられたオブジェクトを直接、間接に変更してはならない。

つまり、以下のようなコードは違法だ。

```
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 ; } ) ;
    // □□□
}
```

これは、ユーザー提供の関数オブジェクトが実引数を lvalue リファレンスで受け取って変更しているので、並列アルゴリズムの制約を満たさない。

std::for_each はイテレーターが変更可能な要素を返す場合、ユーザー提供の関数オブジェクトが実引数を変更することが可能だ。

```
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
}
```

これは、for_each は仕様上そのように定められているからだ。

実引数で与えられたオブジェクトの一意性に依存してはならない

ユーザー提供の関数オブジェクトは実引数で与えられたオブジェクトの一意性に依存してはならない。

これはどういうことかというと 、たとえば実引数で渡されたオブジェクトのアドレスを取得して、そのアドレスがアルゴリズムに渡したオブジェクトのアドレスと同じであることを期待するようなコードを書くことができない。

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

    // 最後の要素へのポインター
    int * ptr = &c[4] ;

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

これはなぜかというと、並列アルゴリズムはその 並列処理の一環として、要素のコピーを作成し、そのコピーをユーザー提供の関数オブジェクトに渡すかもしれないからだ。

```
// 実装例メモ

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 ) ;
    }
}
```

このため、オブジェクトの一意性に依存したコードを書くことはできない。

データ競合と同期

std::execution::sequenced_policy を渡した並列アルゴリズムによる要素アクセス関数の呼び出しは呼び出し側スレッドで実行される。パラレル実行ではない。

std::execution::parallel_policy を渡した並列アルゴリズムによる要素アクセス関数の呼び出しは、呼び出し側スレッドか、ライブラリ側で作られたスレッドのいずれかで実行される。それぞれの要素アクセス関数の呼び出しの同期は定められていない。そのため、要素アクセス関数はデータ競合やデッドロックを起こさないようにしなければならない。

以下のコードはデータ競合が発生するのでエラーとなる。

```
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 ; } ) ;
    // エラー、データ競合
}
```

なぜならば、ユーザー提供の関数オブジェクトは複数のスレッドから同時に呼び出されるかもしれないからだ。

std::execution::parallel_unsequenced_policy の実行は変わっている。未規定のスレッドから同期されない実行が許されている。これは、パラレルベクトル実行で想定している実行媒体がスレッドのような強い実行保証のある実行媒体ではなく、SIMD や GPGPU のような極めて軽い実行媒体であるからだ。

その結果、要素アクセス関数は通常のデータ競合やデッドロックを防ぐための手段すら取れなくなる。なぜならば、スレッドは実行の途中に中断して別の処理をしたりするからだ。

たとえば、以下のコードは動かない。

```
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 ;
        } ) ;
    // □□□
}
```

このコードは parallel_policy ならば、非効率的ではあるが問題なく同期されてデータ競合なく動くコードだ。しかし、parallel_unsequenced_policy では動かない。なぜならば、mutex の lock という同期をする関数を呼び出すからだ。

C++ では、ストレージの確保解放以外の同期する標準ライブラリの関数をすべて、ベクトル化非安全（vectorization-unsafe）に分類している。ベクトル化非安全な関数は std::execution::parallel_unsequenced_policy の要素アクセス関数内で呼び出すことはできない。

7.3.3　例外

並列アルゴリズムの実行中に、一時メモリーの確保が必要になったが確保できない場合、std::bad_alloc □ throw される。

並列アルゴリズムの実行中に、要素アクセス関数の外に例外が投げられた場合、std::terminate が呼ばれる。

7.3.4 実行ポリシー

実行ポリシーはヘッダーファイル <execution> で定義されている。その定義は以下のようになっている。

```
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> は T が実行ポリシー型であるかどうかを返す traits だ。

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

シーケンス実行ポリシー

```
namespace std::execution {

class sequenced_policy ;
```

```
inline constexpr sequenced_policy seq { } ;

}
```

シーケンス実行ポリシーは、並列アルゴリズムにパラレル実行を行わせないためのポリシーだ。この実行ポリシーが渡された場合、処理は呼び出し元のスレッドだけで行われる。

パラレル**実行**ポリシー

```
namespace std::execution {

class parallel_policy ;
inline constexpr parallel_policy par { } ;

}
```

パラレル実行ポリシーは、並列アルゴリズムにパラレル実行を行わせるためのポリシーだ。この実行ポリシーが渡された場合、処理は呼び出し元のスレッドと、ライブラリが作成したスレッドを用いる。

パラレル**非**シーケンス**実行**ポリシー

```
namespace std::execution {

class parallel_unsequenced_policy ;
inline constexpr parallel_unsequenced_policy par_unseq { } ;

}
```

パラレル非シーケンス実行ポリシーは、並列アルゴリズムにパラレル実行かつベクトル実行を行わせるためのポリシーだ。この実行ポリシーが渡された場合、処理は複数のスレッドと、SIMD や GPGPU のようなベクトル実行による並列化を行う。

実行ポリシーオブジェクト

```
namespace std::execution {

inline constexpr sequenced_policy seq{ };
inline constexpr parallel_policy par{ };
inline constexpr parallel_unsequenced_policy par_unseq{ };
```

```
}
```

実行ポリシーの型を直接書くのは面倒だ。

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

そのため、標準ライブラリは実行ポリシーのオブジェクトを用意している。seq と par と par_unseq がある。

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

並列アルゴリズムを使うには、このオブジェクトを並列アルゴリズムの第一引数に渡すことになる。

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：

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

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：

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

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：

$$\Pi(\nu, k) = \Pi(\nu, k, \pi/2), \quad \text{for } |k| \le 1$$

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：

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

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：

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

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：

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

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：

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

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：

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

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：

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

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

この章では C++17 で追加された細かいライブラリをまとめて 解説する。

9.1 ハードウェア干渉サイズ（キャッシュライン）

C++17 にはハードウェアの干渉サイズを取得するライブラリが入った。ハードウェアの干渉サイズとは、俗にキャッシュライン（cache line）とも呼ばれている概念だ。

残念ながら、この 2017 年では、メモリーは極めて遅い。そのため、プロセッサーはより高速にアクセスできるキャッシュメモリーを用意している。メモリーに対するキャッシュはある程度のまとまったバイト数単位で行われる。この単位が何バイトであるのかは実装依存だ。C++17 にはこのサイズを取得できるライブラリが入った。

ハードウェア干渉サイズを知りたい理由は 2 つある。2 つのオブジェクトを同一の局所性を持つキャッシュに載せたくない場合と載せたい場合だ。

2 つのオブジェクトのうち、一方は頻繁に変更し、もう一方はめったに変更しない場合で、2 つのオブジェクトが同じ局所性を持つキャッシュに載っている場合、よく変更するオブジェクトを変更しただけで、めったに変更しないオブジェクトも、メモリーとの同期が発生する。

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

ここで、counter は頻繁に変更するが、status はめったに変更しない場合、counter と status の間に適切なパディングを 挿入することで、2 つのオブジェクトが同じ局所性を持たないようにしたい。

□□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 では <functional> に searcher というライブラリが追加された。これは順序のあるオブジェクトの集合に、ある部分集合（パターン）が含まれているかどうかを検索するためのライブラリだ。その最も一般的な応用例は文字列検索となる。

searcher の基本的な設計としては、クラスのオブジェクトを構築して、コンストラクターで検索したい部分集合（パターン）を与え、operator () で部分集合が含まれているかを検索したい集合を与える。

この設計のライブラリが追加された理由は、パターンの検索のために何らかの事前の準備を状態として保持しておきたい検索アルゴリズムを実装するためだ。

9.4.1 default_searcher

クラス std::default_searcher は以下のように宣言されている。

```
template <  class ForwardIterator1,
            class BinaryPredicate = equal_to<> >
class default_searcher {
public:
    // コンストラクター
    default_searcher(
        ForwardIterator1 pat_first, ForwardIterator1 pat_last
        , BinaryPredicate pred = BinaryPredicate() ) ;

    // operator ()
    template <class ForwardIterator2>
    pair<ForwardIterator2, ForwardIterator2>
    operator()(ForwardIterator2 first, ForwardIterator2 last) const ;
} ;
```

コンストラクターで部分集合を受け取る。operator () で集合を受け取り、部分集合（パターン）と一致した場所をイテレーターのペアで返す。見つからない場合、イテレーターのペアは [last, last) になっている。

以下のように使う。

```
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 の検索は、内部的に std::search が使われる。

9.4.2 boyer_moore_searcher

std::boyer_moore_searcher は Boyer–Moore 文字列検索アルゴリズムを使って部分集合の検索を行う。

Boyer–Moore 文字列検索アルゴリズムは極めて効率的な文字列検索のアルゴリズムだ。Boyer–Moore アルゴリズムは Bob Boyer と Strother Moore によって発明され、1977 年の Communications of the ACM で発表された。その内容は以下の URL で読むことができる。

http://www.cs.utexas.edu/~moore/publications/fstrpos.pdf

愚直に実装した文字列検索アルゴリズムは検索すべき部分文字列（パターン）を検索対象の文字列（コーパス）から探す際、パターンの先頭の文字をコーパスの先頭から順に探していき、見つかれば 2 文字目以降も一致するかどうかを調べる。

Boyer–Moore アルゴリズムはパターンの末尾の文字から調べる。文字が一致しなければ、パターンから絶対に不一致であるとわかっている長さだけの文字を比較せずに読み飛ばす。これによって効率的な文字列検索を実現している。

Boyer–Moore アルゴリズムは事前にパターンのどの文字が不一致ならば何文字比較せずに読み飛ばせるかという情報を計算した 2 つのテーブルを生成する必要がある。このため、Boyer–Moore アルゴリズムはメモリー使用量と検索前の準備時間というコストがかかる。そのコストは、より効率的な検索により相殺できる。特に、パターンが長い場合は効果的だ。

C++17 に入る Boyer–Moore アルゴリズムに基づく検索は、テンプレートを使った汎用的な char 型のような状態数の少ない型に対しての実装だけではなく、ハッシュを使ったハッシュマップのようなデータ構造を使うことにより、任意の型に対応できる汎用的な設計になっている。

クラス boyer_moore_searcher は以下のように宣言されている。

```
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 は Boyer–Moore–Horspool 検索アルゴリズムを使って部分集合の検索を行う。Boyer–Moore–Horspool アルゴリズムは Nigel Horspool によって 1980 年に発表された。

参考：“Practical fast searching in strings” 1980

Boyer–Moore–Horspool アルゴリズムは内部テーブルに使うメモリー使用量を削減しているが、最悪計算量の点でオリジナルの Boyer–Moore アルゴリズムには劣っている。つまり、実行時間の増大を犠牲にしてメモリー使用量を削減したトレードオフなアルゴリズムと言える。

クラス boyer_moore_horspool_searcher の宣言は以下のとおり。

```
template <
    class RandomAccessIterator1,
    class Hash = hash<
        typename iterator_traits<RandomAccessIterator1>::value_type>,
    class BinaryPredicate = equal_to<> >
class boyer_moore_horspool_searcher {
public:
    // コンストラクター
    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;
} ;
```

使い方は boyer_moore_horspool_searcher と変わらない。

```
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 で <algorithm> に追加された std::sample は、標本を確率的に選択するための乱択アルゴリズムだ。

[first, last) は標本を選択する先の集合を指すイテレーター。out は標本を出力する先のイテレーター。n は選択する標本の個数。g は標本を選択するのに使う乱数生成器。戻り値は out。

ある要素の集合から、n 個の要素を確率的に公平に選択したい場合に使うことができる。

9.5.1 乱択アルゴリズム

std::sample を使う前に、まず正しい乱択アルゴリズムについて学ぶ必要がある。乱択アルゴリズムについて詳しくは、Donald E. Knuth の *The Art of Computer Programming* （以下 TAOCP、邦訳はアスキードワンゴから同名の書名で出版されている）を参照。

ユーザーからの入力、計測した気象情報のデータ、サーバーへのアクセスログなど、世の中には膨大な量のデータが存在する。これらの膨大なデータをすべて処理するのではなく、標本を採取することによって、統計的にそれなりに信頼できる確率で正しい全体のデータを推定することができる。そのためには n 個の標本をバイアスがかからない方法で選択する必要がある。バイアスのかからない方法で n 個の標本を取り出すには、集合の先頭から n 個とか、1 つおきに n 個といった方法で選択して

はならない。それはバイアスがかかっている。

ある値の集合から、バイアスのかかっていない n 個の標本を得るには、集合のすべての値が等しい確率で選ばれた上で n 個を選択しなければならない。いったいどうすればいいのだろうか。

std::sample を使えば、100 個の値から 10 個の標本を得るのは、以下のように書くことが可能だ。

```
int main()
{
    // 100 個の値□集合
    std::vector<int> pop(100) ;
    std::iota( std::begin(pop), std::end(pop), 0 ) ;

    // 標本を格納□□コン□ナー
    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 個の標本を得る
    sample( std::begin(pop), std::end(pop), std::begin(out), 10, g ) ;

    // 標本を出力
    std::copy(  std::begin(out), std::end(out),
                std::ostream_iterator<int>(std::cout, ", ") ) ;
}
```

集合に含まれる値の数が N 個だとわかっているならば、それぞれの値について n/m の確率で選ぶというのはどうだろうか。100 個中 10 個を選ぶのであれば、$1/10$ の確率でそれぞれの値を標本として選択することになる。

この考えに基づく乱択アルゴリズムは以下のようになる。

1. 集合の要素数を N、選択すべき標本の数を n, i を 0 とする。
2. 0 ベースインデックスで i 番目の値を n/m の確率で標本として選択する。
3. i をインクリメントする。
4. $i\ != N$ ならば 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 の TAOCP Vol. 2 では、アルゴリズム S と称して、要素数のわかっている集合から標本を選択する方法を解説している。

アルゴリズム S は以下のとおり。

$0 < n \leq N$ のとき、N 個の集合から n 個の標本をランダムに選択する。

1. t, m を 0 とする。t はこれまでに処理した要素数、m は標本として選択した要素数とする。
2. $0 \leq U \leq N - t$ の範囲の乱数 U を生成する。
3. $U \geq n - m$ であれば goto 5。
4. 次の要素を標本として選択する。m と t をインクリメントする。$m < n$ であれば、goto 2。そうでなければ標本は完了したのでアルゴリズムは終了する。
5. 次の要素を標本として選択しない。t をインクリメントする。goto 2。

実装は以下のようになる。

```
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

アルゴリズム S は集合の要素数が N 個であるとわかっている場合に、n 個の標本を選択するアルゴリズムだ。では、もし N がわからない場合はどうすればいいのだろうか。

現実には N がわからない状況がよくある。

- ユーザーからの入力
- シーケンシャルアクセスしか提供しておらず全部読み込まなければ要素数のわからないテープデバイスからの入力
- ガイガーカウンターからの入力

このような要素数のわからない入力にアルゴリズム S を適用するには、まず一度全部入力を得て、全体の要素数を確定させた上で、全要素に対してアルゴリズム S を適用させるという 2 段階の方法を使うことができる。

しかし、1 段階の要素の巡回だけで済ませたい。要素数のわからない入力を処理して、その時点で公平に選択された標本を得たい。

アルゴリズム R はそのような状況で使えるアルゴリズムだ。

アルゴリズム R では、要素数のわからない要素の集合から n 個の標本を選択する。そのために標本として選択した要素を保管しておき、新しい入力が与えられるたびに、標本として選択するかどうかの判断をし、選択をするのであれば、保管しておいた既存の標本と置き換える。

アルゴリズム R は以下のとおり（このアルゴリズムは Knuth 本とは違う）。

$n > 0$ のとき、$size \geq n$ である未確定の $size$ 個の要素数を持つ入力から、n 個の標本をランダムに選択する。標本の候補は n 個まで保管される。$1 \leq j \leq n$ のとき $I[j]$ は保管された標本を指す。

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

ここまで説明したように、乱択アルゴリズムには 2 種類ある。入力の要素数がわかっている場合のアルゴリズム S（選択標本）と、入力の要素数がわからない場合のアルゴリズム R（保管標本）だ。

しかし、C++ に追加された乱択アルゴリズムの関数テンプレートの宣言は、はじめに説明したように以下の 1 つしかない。並列アルゴリズムには対応していない。

```
template<
    class PopulationIterator, class SampleIterator,
    class Distance, class UniformRandomBitGenerator >
SampleIterator
sample(
    PopulationIterator first, PopulationIterator last,
    SampleIterator out,
    Distance n, UniformRandomBitGenerator&& g) ;
```

[first, last) は標本を選択する先の集合を指すイテレーター。out は標本を出力する先のイテレーター。n は選択する標本の個数。g は標本を選択するのに使う乱数生成器。戻り値は out。

sample は PopulationIterator と SampleIterator のイテレーターカテゴリーによって、どちらのアルゴリズムを使うべきか判断している。

アルゴリズム S（選択標本）を使う場合、PopulationIterator は前方イテレーター、SampleIterator は出力イテレーターを満たさなければならない。

アルゴリズム R（保管標本）を使う場合、PopulationIterator は入力イテレーター、SampleIterator はランダムアクセスイテレーターを満たさなければならない。

これはどういうことかというと、要素数の取得のためには、入力元の

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 ) ;
}
```

以下のような意味を持つ。

```
int main()
{
    std::list<int> input(10000) ;
    std::list<int> sample(100) ;
    std::knuth_b g ;

    std::size_t count = 0 ;

    // 要素数の得るためにイテレーターを回す
    // 非効率的
    for( auto && e : input )
    { ++count ; }

    // 標本の選択のためにイテレーターを回す
    for ( auto && e : input )
    {/* 標本の選択 */}
}
```

std::list のメンバー関数 size は定数時間であることが保証されているため、このコードにおけるイテレーターを回すループは 1 回に抑えられる。しかし、std::sample は要素数を渡す実引数がないために要素数がイテレーターを全走査しなくてもわかっている場合でも、非効率的な処理を行わなければならない。

もしランダムアクセスイテレーター未満、前方イテレーター以上のイテレーターカテゴリーのイテレーターの範囲から標本を選択したい場合で、イテレーターの範囲の指す要素数があらかじめわかっている場合は、自前でアルゴリズム S を実装したほうが効率がよい。

```
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 はヘッダーファイル <tuple> で定義されている。

```
template <class T, class Tuple>
constexpr T make_from_tuple(Tuple&& t);
```

apply は tuple の要素を実引数に関数を呼び出すライブラリだが、make_from_tuple は tuple の要素を実引数にコンストラクターを呼び出すライブラリだ。

ある型 T と要素数 N の tuple t に対して、make_from_tuple<T>(t) は、T 型を T(get<0>(t), get<1>(t), ... , get<N-1>(t)) のように構築して、構築した T 型のオブジェクトを返す。

```
class X
{
    template < typename ... Types >
    T( Types ... ) { }
} ;

int main()
{
    // int, int, int
    std::tuple t1(1,2,3) ;

    // X(1,2,3)
    X x1 = std::make_from_tuple<X>( t1 )

    // int, double, const char *
    std::tuple t2( 123, 4.56, "hello" ) ;

    // X(123, 4.56, "hello")
    X x2 = std::make_from_tuple<X>( t2 ) ;
}
```

9.9 invoke : 指定した関数を指定した実引数で呼び出す

invoke はヘッダーファイル <functional> で定義されている。

```
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) は、関数 f を f(a1, a2, ... , aN) のように呼び出す。

より正確には、C++ 標準規格の INVOKE(f, t1, t2, ... , tN) と同じ規則で呼び出す。これにはさまざまな規則があり、たとえばメンバー関数へのポインターやデータメンバーへのポインター、またその場合に与えるクラスへのオブジェクトがリファレンスかポインターか reference_wrapper かによっても異なる。その詳細はここでは解説しない。

INVOKE は std::function や std::bind でも使われている規則なので、標準ライブラリと同じ挙動ができるようになると覚えておけばよい。

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 はヘッダーファイル <functional> で定義されている。

```
template <class F> unspecified not_fn(F&& f);
```

関数オブジェクト f に対して not_fn(f) を呼び出すと、戻り値として何らかの関数オブジェクトが返ってくる。その関数オブジェクトを呼び出すと、実引数を f に渡して f を関数呼び出しして、戻り値を operator ! で否定して返す。

```
int main()
{

    auto r1 = std::not_fn( []{ return true ; } ) ;

    r1() ; // false

    auto r2 = std::not_fn( []( bool b ) { return b ; } ) ;

    r2(true) ; // false
}
```

すでに廃止予定になった not1, not2 の代替品。

9.11　メモリー管理アルゴリズム

C++17 ではヘッダーファイル <memory> にメモリー管理用のアルゴリズムが追加された。

9.11.1　addressof

```
template <class T> constexpr T* addressof(T& r) noexcept;
```

addressof は C++17 以前からもある。addressof(r) は r のポインターを取得する。たとえ、r の型が operator &をオーバーロードしていても正しいポインターを取

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

ヘッダーファイル <type_traits> で定義されている void_t は以下のように定義されている。

```
namespace std {

template < class ... >
using void_t = void ;

}
```

void_t は任意個の型をテンプレート実引数として受け取る void 型だ。この性質はテンプレートメタプログラミングにおいてとても便利なので、標準ライブラリに追加された。

9.14 bool_constant

ヘッダーファイル <type_traits> に bool_constant が追加された。

```
template <bool B>
using bool_constant = integral_constant<bool, B>;

using true_type = bool_constant<true>;
using false_type = bool_constant<false>;
```

今まで integral_constant を使っていた場面で特に bool だけが必要な場面では、C++17 以降は単に std::true_type か std::false_type と書くだけでよくなる。

9.15 type_traits

C++17 では <type_traits> に機能追加が行われた。

9.15.1 変数テンプレート版 traits

C++17 では、既存の traits に変数テンプレートを利用した _v 版が追加された。

たとえば、is_integral<T>::value と書く代わりに is_integral_v<T> と書くことができる。

```
template < typename T >
void f( T x )
{
    constexpr bool b1 = std::is_integral<T>::value ; // データメンバー
    constexpr bool b2 = std::is_integral_v<T> ; // 変数テンプレート
    constexpr bool b3 = std::is_integral<T>{} ; // operator bool()
}
```

9.15.2　論理演算 traits

C++17 ではクラステンプレート conjunction, disjunction, negation が追加された。これはテンプレートメタプログラミングで論理積、論理和、否定を手軽に扱うための traits だ。

conjunction : 論理積

```
template<class... B> struct conjunction;
```

クラステンプレート conjunction<B1, B2, ..., BN> はテンプレート実引数 B1, B2, ..., BN に論理積を適用する。conjunction はそれぞれのテンプレート実引数 Bi に対して、bool(Bi::value) が false となる最初の型を基本クラスに持つか、あるいは最後の BN を基本クラスに持つ。

```
int main()
{
    using namespace std ;

    // is_void<void>を基本クラスに持つ
    using t1 =
        conjunction<
            is_same<int, int>, is_integral<int>,
            is_void<void> > ;

    // is_integral<double>を基本クラスに持つ
    using t2 =
        conjunction<
            is_same<int, int>, is_integral<double>,
            is_void<void> > ;

}
```

disjunction : 論理和

```
template<class... B> struct disjunction;
```

クラステンプレート disjunction<B1, B2, ..., BN>はテンプレート実引数 B1, B2, ..., BN に論理和を適用する。disjunction はそれぞれのテンプレート実引数 Bi に対して、bool(Bi::value) が true となる最初の型を基本クラスに持つか、あるいは最後の BN を基本クラスに持つ。

```
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;
```

クラステンプレート negation<B> は B に否定を適用する。negation は基本クラスとして bool_constant<!bool(B::value)> を持つ。

```
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> は、T 型がトリビアルにコピー可能で、かつ T 型の同値である 2 つのオブジェクトの内部表現が同じ場合に、true を返す。

false を返す例としては、オブジェクトがパディング（padding）と呼ばれるアライメント調整などのための値の表現に影響しないストレージ領域を持つ場合だ。パディングビットの値は同値に影響しないので、false を返す。

たとえば以下のようなクラス X は、

```
struct X
{
    std::uint8_t a ;
    std::uint32_t b ;
} ;
```

ある実装においては、4 バイトにアライメントする必要があり、そのオブジェクトの本当のレイアウトは以下のようになっているかもしれない。

```
struct X
{
    std::uint8_t a ;

    std::byte unused_padding[3] ;

    std::uint32_t b ;
} ;
```

この場合、unused_padding の値には意味がなく、クラス X の同値比較には用いられない。この場合、std::has_unique_representations_v<X> は false になる。

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> は T 型が swap で例外を投げないときに true を返す。

is_nothrow_swappable_with<T, U> は、T 型と U 型を相互に swap するときに例外を投げないときに true を返す。

9.16 コンテナーで不完全型のサポート

注意：この説明は上級者向けだ。

C++17 では以下のコードが合法になった。このコードの挙動は C++14 までは実装依存であった。

```
struct X
{
    std::vector<X> v ;
    std::list<X> l ;
    std::forward_list<X> f ;
} ;
```

クラスはクラス定義の終了である } を持って完全型となる。クラススコープに注入されたクラス名は、クラス定義の中ではまだ完全型ではない。不完全型をコンテナーの要素型に指定した場合の挙動は、C++14 までは規定されていなかった。

C++17 では、vector, list, forward_list に限り、要素型に一時的に不完全型を許すようになった。実際にコンテナーを使う際には完全型になっていなければならない。

9.17 emplace の戻り値

C++17 ではシーケンスコンテナーの emplace_front/emplace_back, queue と stack の emplace が構築した要素へのリファレンスを返すように変更された。

☒☒☒☒、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

である。

```
int main()
{
    std::map< int, std::unique_ptr<int> > m ;

    // すでに要素が存在する
    m[0] = nullptr ;

    auto ptr = std::make_unique<int>(0) ;
    // emplace は失敗する
    auto [iter, is_emplaced] = m.emplace( 0, std::move(ptr) ) ;

    // 結果は実装により異なる
    // ptr はムーブされているかもしれない
    bool b = ( ptr != nullptr ) ;
}
```

この場合、実際に `map` に要素は追加されていないのに、`ptr` はムーブされてしまうかもしれない。

このため、C++17 では、要素が追加されなかった場合 `args` はムーブされないことが保証される `try_emplace` が追加された。

```
int main()
{
    std::map< int, std::unique_ptr<int> > m ;

    // すでに要素が存在する
    m[0] = nullptr ;

    auto ptr = std::make_unique<int>(0) ;
    // emplace は失敗する
    auto [iter, is_emplaced] = m.emplace( 0, std::move(ptr) ) ;

    // true であることが保証される
    // ptr はムーブされていない
    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 は key に連想された要素が存在する場合は要素を代入し、存在しない場合は要素を追加する。operator [] との違いは、要素が代入されたか追加されたかが、戻り値の pair の bool でわかるということだ。

```
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 では、連想コンテナーと非順序連想コンテナーで splice 操作がサポートされた。

対象のコンテナーは map, set, multimap, multiset, unordered_map, unordered_set, unordered_multimap, unordered_multiset だ。

splice 操作とは list で提供されている操作で、アロケーター互換の list のオブ

ジェクトの要素をストレージと所有権ごと別のオブジェクトに移動する機能だ。

```
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}

}
```

連想コンテナーでは、ノードハンドルという仕組みを用いて、コンテナーのオブジェクトから要素の所有権をコンテナーの外に出す仕組みで、splice 操作を行う。

9.19.1 merge

すべての連想コンテナーと非順序連想コンテナーは、メンバー関数 merge を持っている。コンテナー a, b がアロケーター互換のとき、a.merge(b) は、コンテナー b の要素の所有権をすべてコンテナー a に移す。

```
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 □{}
}
```

もし、キーの重複を許さないコンテナーの場合で、値が重複した場合、重複した要素は移動しない。

```
int main()
{
    std::set<int> a = {1,2,3} ;
    std::set<int> b = {1,2,3,4,5,6} ;

    a.merge(b) ;

    // a は{1,2,3,4,5,6}
    // b は{1,2,3}

}
```

merge によって移動された要素を指すポインターとイテレーターは、要素の移動後も妥当である。ただし、所属するコンテナーのオブジェクトが変わる。

```
int main()
{
    std::set<int> a = {1,2,3} ;
    std::set<int> b = {4,5,6} ;


    auto iterator = std::begin(b) ;
    auto pointer = &*iterator ;

    a.merge(b) ;

    // iterator と pointer はまだ妥当
    // ただし要素は a に所属する
}
```

9.19.2 ノードハンドル

ノードハンドルとは、コンテナーオブジェクトから要素を構築したストレージの所有権を切り離す機能だ。

ノードハンドルの型は、各コンテナーのネストされた型名 node_type となる。たとえば std::set<int> のノードハンドル型は、std::set<int>::node_type となる。

ノードハンドルは以下のようなメンバーを持っている。

```
class node_handle
{
public :
```

```
    // ネストされた型名
    using value_type = ... ;        // set 限定、要素型
    using key_type = ... ;          // map 限定、キー型
    using mapped_type = ... ;       // map 限定、マップ型
    using allocator_type = ... ;    // アロケーターの型

    // デフォルトコンストラクター
    // ムーブコンストラクター
    // ムーブ代入演算子


    // 値へのアクセス
    value_type & value() const ;   // set 限定
    key_type & key() const ;        // map 限定
    mapped_type & mapped() const ;  // map 限定

    // アロケーターへのアクセス
    allocator_type get_allocator() const ;

    // 空かどうかの判定
    explicit operator bool() const noexcept ;
    bool empty() const noexcept ;

    void swap( node_handle & ) ;
} ;
```

set のノードハンドルはメンバー関数 value で値を得る。

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

    auto n = c.extract(2) ;

    // n.value() == 2
    // c は{1,3}
}
```

map のノードハンドルはメンバー関数 key と mapped でそれぞれの値を得る。

```
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 は{{1,1},{3,3}}

}
```

ノードハンドルはノードをコンテナーから切り離し、所有権を得る。そのため、ノードハンドルによって得たノードは、元のコンテナーから独立し、元のコンテナーオブジェクトの破棄の際にも破棄されない。ノードハンドルのオブジェクトの破棄時に破棄される。このため、ノードハンドルはアロケーターのコピーも持つ。

```
int main()
{
    std::set<int>::node_type n ;

    {
        std::set<int> c = { 1,2,3 } ;
        // 所有権の移動
        n = c.extract( std::begin(c) ) ;
        // c が破棄される
    }

    // OK
    // ノードハンドルによって所有権が移動している
    int x = n.value() ;

    // n が破棄される
}
```

9.19.3 extract : ノードハンドルの取得

```
node_type extract( const_iterator position ) ;
node_type extract( const key_type & x ) ;
```

連想コンテナーと非順序連想コンテナーのメンバー関数 extract は、ノードハンドルを取得するためのメンバー関数だ。

メンバー関数 extract(position) は、イテレーターの position が指す要素を、コンテナーから除去して、その要素を所有するノードハンドルを返す。

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

    auto n1 = c.extract( std::begin(c) ) ;

    // c は{2,3}

    auto n2 = c.extract( std::begin(c) ) ;

    // c □{3}

}
```

メンバー関数 extract(x) は、キー x がコンテナーに存在する場合、その要素をコンテナーから除去して、その要素を所有するノードハンドルを返す。存在しない場合、空のノードハンドルを返す。

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

    auto n1 = c.extract( 1 ) ;
    // c □{2,3}

    auto n2 = c.extract( 2 ) ;
    // c は{3}

    // キー 4 □存在□ない
    auto n3 = c.extract( 4 ) ;
    // c は{3}
    // n3.empty() == true
}
```

キーの重複を許すコンテナーの場合、複数あるうちの 1 つの所有権が解放される。

```
int main()
```

```
{
    std::multiset<int> c = {1,1,1} ;
    auto n = c.extract(1) ;
    // c は{1,1}
}
```

9.19.4 insert : ノードハンドルから要素の追加

```
// キーの重複を許さないコンテナーの場合
insert_return_type  insert(node_type&& nh);
// キーの重複を許すmulti コンテナーの場合
iterator  insert(node_type&& nh);

// ヒント付きの insert
iterator            insert(const_iterator hint, node_type&& nh);
```

ノードハンドルをコンテナーのメンバー関数 insert の実引数に渡すと、ノードハンドルから所有権をコンテナーに移動する。

```
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
}
```

ノードハンドルが空の場合、何も起こらない。

```
int main()
{
    std::set<int> c ;
    std::set<int>::node_type n ;

    // 何も起こらない
    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 は insert によってコンテナーに所有権を移動して追加された要素を指すイテレーター、inserted は要素の追加が行われた場合に true となる bool, node は要素の追加が失敗したときにノードハンドルの所有権が移動されるノードハンドルとなる。

insert に渡したノードハンドルが空のとき、inserted は false, position は end(), node は空になる。

```
    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 が成功したとき、inserted は true, position は追加された要素を指し、node は空になる。

```
    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 が失敗したとき、つまりすでに同一のキーがコンテナーに存在したとき、inserted は false, node は insert を呼び出す前のノードハンドルの値、position は

□□□□□□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 が破棄される
    }

    // コンテナーの破棄後も存続する
    int value = n.value() ;
}
```

map のキーを変更する

map ではキーは変更できない。キーを変更したければ、元の要素は削除して、新しい要素を追加する必要がある。これには動的なストレージの解放と確保が必要になる。

ノードハンドルを使えば、既存の要素のストレージに対して、所有権を map から引き剥がした上で、キーを変更して、もう一度 map に差し戻すことができる。

```
int main()
{
    std::map< std::string, std::string > m =
    {
        {"cat", "meow"},
        {"DOG", "bow"}, // キーを間違えたので変更したい
        {"cow", "moo"}
    } ;

    // 所有権を引き剥がす
    auto n = m.extract("DOG") ;
    // キーを変更
    n.key() = "dog" ;
    // 差し戻す
    m.insert( std::move(n) ) ;
}
```

9.20 コンテナーアクセス関数

ヘッダーファイル <iterator> に、コンテナーアクセス関数として、フリー関数版の size, empty, data が追加された。それぞれ、メンバー関数の size, empty, data を

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 のコンストラクターは複数のロックのオブジェクトのリファレンスを取り、それぞれにデッドロックを起こさない方法でメンバー関数 lock を呼び出す。デストラクターはメンバー関数 unlock を呼び出す。

9.25　std::byte

C++17 ではバイトを表現する型として std::byte がライブラリに追加された。これは、コア言語の一部であり、別項で詳しく解説を行っている。

9.26　最大公約数（gcd）と最小公倍数（lcm）

C++17 ではヘッダーファイル <numeric> に最大公約数（gcd）と最小公倍数（lcm）が追加された。

```
int main()
{
    int a, b ;

    while( std::cin >> a >> b )
    {
        std::cout
            << "gcd: " << gcd(a,b)
            << "\nlcm: " << lcm(a,b) << '\n' ;
    }
}
```

9.26.1　gcd：最大公約数

```
template <class M, class N>
constexpr std::common_type_t<M,N> gcd(M m, N n)
{
```

```
    if ( n == 0 )
        return m ;
    else
        return gcd( n, std::abs(m) % std::abs(n) ) ;
}
```

gcd(m, n) は m と n のいずれかが 0 の場合は 0 を返す。0 以外の場合、$|m|$ と $|n|$ の最大公約数（Greatest Common Divisor）を返す。

9.26.2 lcm：最小公倍数

```
template <class M, class N>
constexpr std::common_type_t<M,N> lcm(M m, N n)
{
    if ( m == 0 || n == 0 )
        return 0 ;
    else
        return std::abs(m) / gcd( m, n ) * std::abs(n) ;
}
```

lcm(m,n) は、m と n のいずれかが 0 の場合は 0 を返す。0 以外の場合、$|m|$ と $|n|$ の最小公倍数（Least Common Multiple）を返す。

第 10 章
ファイルシステム

ヘッダーファイル <filesystem> で定義されている標準ライブラリのファイルシステムは、ファイルやディレクトリーとその属性を扱うためのライブラリだ。

一般に「ファイルシステム」といった場合、たとえば Linux の ext4, Microsoft Windows の FAT や NTFS, Apple Mac の HFS+ や APFS といったファイルとその属性を表現するためのストレージ上のデータ構造を意味する。C++ の標準ライブラリのファイルシステムとは、そのようなファイルシステムを実現するデータ構造を操作するライブラリではない。ファイルシステムというデータ構造で抽象化された、ファイルやディレクトリーとその属性、それに付随する要素、たとえばパスやファイルやディレクトリーを操作するためのライブラリのことだ。

また、ファイルシステムライブラリでは、「ファイル」という用語は単に通常のファイルのみならず、ディレクトリー、シンボリックリンク、FIFO（名前付きパイプ）、ソケットなどの特殊なファイルも含む。

本書ではファイルシステムライブラリのすべてを詳細に解説していない。ファイルシステムライブラリは量が膨大なので、特定の関数の意味については、C++ コンパイラーに付属のリファレンスマニュアルなどを参照するとよい。

10.1 名前空間

ファイルシステムライブラリは std::filesystem 名前空間スコープの下に宣言されている。

```
int main()
{
    std::filesystem::path p("/bin") ;
}
```

この名前空間は長いので、ファイルシステムライブラリを使うときは、関数のブロックスコープ単位で using ディレクティブを使うか、名前空間エイリアスを使って短い別名を付けるとよい。

```
void using_directive()
{
    // using ディ□ク□ィブ
    using namespace std::filesystem ;

    path p("/etc") ;
}

void namespace_alias()
{
    // 名前空間エイリアス
    namespace fs = std::filesystem ;

    fs::path p("/usr") ;
}
```

10.2　POSIX 準拠

C++ のファイルシステムのファイル操作の挙動は、POSIX 規格に従う。実装によっては POSIX に規定された挙動を提供できない場合もある。その場合は制限の範囲内で、できるだけ POSIX に近い挙動を行う。実装がどのような意味のある挙動も提供できない場合、エラーが通知される。

10.3　ファイルシステムの全体像

ファイルシステムライブラリの全体像を簡単に箇条書きすると以下のとおり。

- クラス path でファイルパス文字列を扱う
- 例外クラス filesystem_error とクラス error_code でエラー通知
- クラス file_status でファイルの情報とパーミッションの取得、設定
- クラス directory_entry でディレクトリーの情報の取得、設定
- クラス directory_iterator でディレクトリー構造をイテレーターとしてたどる
- 多数のフリー関数でファイルとディレクトリーの操作

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 )
    { // エラー場合
        auto path1 = file ; // 第一引数
        auto path2 = file ; // 第二引数

        std::cout
            << "error number: " << error_code.value ()
            << "\nerror message: " << error_code.message()
            << "\npath1: " << path1
            << "\npath2: " << path2 << '\n' ;
    }
}
```

10.5 path : ファイルパス文字列クラス

std::filesystem::path ファイルパス文字列表現ライブラリ。文字列表現ライブラリ C++ 標準 std::string 標準、ファイルパス文字列表現標準、別専用標準作成。

path 以下機能提供。

- ファイルパス文字列表現
- ファイルパス文字列操作

path ファイルパス文字列表現操作標準提供、物理ファイル変更。

ファイルパス文字列ファイル表現実装異。POSIX 環境 文字型 char 型 UTF-8 ファイル表現 OS 多、Microsoft Windows 本書執筆現在、文字型 wchar_t UTF-16 ファイル表現慣習。

、OS ファイル大文字小文字区別、区別無視実装。

path ファイル文字列差異吸収。

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" ) ;
}
```

なので、どの文字列エンコードで渡しても動く。

C++ では UTF-8 エンコードの文字型は char で、これはネイティブナローエンコードの文字型と同じなので、型システムによって区別できない。そのため、UTF-8 文字列リテラルを渡すと、ネイティブナローエンコードとして認識される。

```
int main()
{
    using namespace std::filesystem ;

    // ネイティブナローエンコードとして 解釈される
    path p( u8"ファイル名" ) ;
}
```

このコードは、ネイティブナローエンコードが UTF-8 ではない場合、動く保証のない移植性の低いコードだ。UTF-8 エンコードを移植性の高い方法でファイルパスとして使いたい場合、u8path を使うとよい。

```
int main()
{
    using namespace std::filesystem ;

    // UTF-8 エンコードとして解釈される
    // 実装の使う文字エンコードに変換される
    path = u8path( u8"ファイル名" ) ;
}
```

u8path(Source) は Source を UTF-8 エンコードされた文字列として扱うので、通常の文字列リテラルを渡すと、ネイティブナローエンコードが UTF-8 ではない環境では問題になる。

```
int main()
{
    using namespace std::filesystem ;

    // UTF-8 エンコードとして解釈される
    // ネイティブナローエンコードが UTF-8 ではない場合、問題になる
    path = u8path( "ファイル名" ) ;
}
```

u8path を使う場合は、文字列は必ず UTF-8 エンコードしなければならない。

環境によっては、ファイルパスに使える文字に制限があり、また特定の文字列は特別な意味を持つ予約語になっていることもあるので、移植性の高いプログラムの作成に当たってはこの点でも注意が必要だ。たとえば、環境によっては大文字小文字の区別をしないかもしれない。また、CON や AUX のような文字列が特別な意味を持つかもしれない。

path に格納されているファイルパス文字列を取得する方法は、環境依存の文字列エンコードとファイルパスの表現方法の差異により、さまざまな方法が用意されている。

ファイルパス文字列のフォーマットには以下の 2 つがある。

- ネイティブ： 実装依存のフォーマット
- ジェネリック：汎用的な標準のフォーマット

POSIX 準拠の環境においては、ネイティブとジェネリックはまったく 同じだ。POSIX 準拠ではない環境では、ネイティブとジェネリックは異なるフォーマットを持つ可能性がある。

たとえば、Microsoft Windows では、ネイティブのファイルパス文字列はディレクトリーの区切り文字に POSIX 準拠の / ではなく \ を使っている。

まずメンバー関数 native と c_str がある。

```
class path {
{
public :
    const string_type& native() const noexcept;
    const value_type* c_str() const noexcept;
} ;
```

これはクラス path が内部で使っている実装依存のネイティブな文字列型をそのまま返すものだ。

```
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() ;

}
```

このメンバー関数を使うコードは移植性に注意が必要だ。

str の型は path::string_type で、ptr の型は実装依存の path::value_type const * だ。path::value_type ☒ path::string_type は、char や wchar_t, std::string や std::wstring のような C++ が標準で定義する型ではない可能性がある。

そして、path::string_type への変換関数 operator string_type() がある。

```
int main()
{
    using namespace std::experimental::filesystem ;

    auto p = current_path() ;

    // 暗黙☒型変換
    path::string_type str = p ;
}
```

path の operator string_type() は、ネイティブの文字列型を既存のファイルストリームライブラリでオープンできる形式に変換して返す。たとえば空白文字を含むファイルパスのために、二重引用符で囲まれている文字列に変換されるかもしれない。

```
int main()
{
    using namespace std::filesystem ;

    path name("foo bar.txt") ;
    std::basic_ofstream<path::value_type> file( name ) ;
    file << "hello" ;
}
```

ネイティブのファイルパス文字列を string, wstring, u16string, u32string に変換して取得するメンバー関数に以下のものがある。

```
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;
} ;
```

以下のように使う。

```
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" ;
}
```

このプログラムは、カレントディレクトリーにある通常のファイルの数と、実行可能なファイルの数を表示する。

ファイルパーミッションを表現する enum 型 perms は、パーミッションが不明な場合 perms::unknown になる。この値は 0xFFFF なのでビット演算をする場合には注意が必要だ。

それ以外の perms の値は POSIX に準拠しているが、perms は scoped enum 型なの

で、明示的なキャストが必要だ。

```
// エラー
std::filesystem::perms a = 0755 ;

// OK
std::filesystem::perms b = std::filesystem::perms(0755) ;
```

ファイルのタイプとパーミッションを書き換えるメンバー関数は以下のとおり。

```
void type(file_type ft) noexcept;
void permissions(perms prms) noexcept;
```

ただし、file_status というのは単なるキャッシュ用のクラスなので、file_status のタイプとパーミッションを「書き換える」というのは、単に file_status のオブジェクトに保持されている値を書き換えるだけで、物理ファイルシステムに反映されるものではない。物理ファイルシステムを書き換えるには、フリー関数の permissions を使う。

10.7　directory_entry

クラス directory_entry はファイルパス文字列を保持し、ファイルパスの指し示すファイルの情報を取得できるクラスだ。

物理ファイルシステムからファイルの情報を毎回読むのは非効率的だ。directory_entry はいわばファイル情報のキャッシュとしての用途を持つ。

directory_entry は物理ファイルシステムから情報を読み込むだけで、変更のコミットはしない。

directory_entry の構築は、コンストラクターに引数として path を与える他、directory_iterator と recursive_directory_iterator からも得ることができる。

```
int main()
{
    using namespace std::filesystem ;

    path p(".") ;

    // ファイルパス文字列から得る
    directory_entry e1(p) ;

    // イテレーターから得る
```

```
    directory_iterator i1(p) ;
    directory_entry e2 = *i1 ;

    recursive_directory_iterator i2(p) ;
    directory_entry e3 = *i2 ;
}
```

directory_entry にはさまざまなファイル情報を取得するメンバー関数があるが、これは同じ機能のものがフリー関数でも用意されている。directory_entry を使うと、ファイル情報をキャッシュできるため、同じファイルパスに対して、物理ファイルシステムの変更がないときに複数回のファイル情報取得を行うのが効率的になる。

```
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() ;

    // ファイルサイズを取得
    std::uintmax_t size = entry.file_size() ;

    {
        std::ofstream foo( entry.path() ) ;
        foo << "hello" ;
    }

    // 物理ファイルシステムから情報を更新
    entry.refresh() ;
    // もう一度ファイルサイズを取得
    size = entry.file_size() ;

    // 情報を取得するファイルパスを
    // "/home/cpp/bar"
    // に置き換えて refresh() を呼び出す
```

```
    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⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠⊠

このうち取りうる組み合わせは、none, follow_directory_symlink, skip_permission_denied, follow_directory_symlink | skip_permission_denied の 4 種類になる。

```
int main()
{
    using namespace std::filesystem ;
    recursive_directory_iterator
        iter("/", directory_options::skip_permission_denied), end ;

    std::copy(  iter, end,
                std::ostream_iterator<path>(std::cout, "\n") ) ;
}
```

follow_directory_symlink は、親ディレクトリーへのシンボリックリンクが存在する場合、イテレーターが終端イテレーターに到達しない可能性があるので注意すること。

```
int main()
{
    using namespace std::filesystem ;

    // 自分自身を含むディレクトリーに対するシンボリックリンク
    create_symlink(".", "foo") ;

    recursive_directory_iterator
        iter(".", directory_options::follow_directory_symlink), end ;

    // エラー、もしくは終了しない
    std::copy( iter, end, std::ostream_iterator<path>(std::cout) ) ;
}
```

recursive_directory_iterator の現在の directory_options を得るには、メンバー関数 options を呼ぶ。

```
class recursive_directory_iterator {
public :
    directory_options options() const ;
} ;
```

10.9.2 depth：深さ取得

recursive_directory_iterator が現在列挙しているディレクトリーの深さを知るには、メンバー関数 depth を呼ぶ。

```
class recursive_directory_iterator {
public :
    int depth() const ;
} ;
```

最初のディレクトリーの深さは 0 で、次のサブディレクトリーの深さは 1、それ以降のサブディレクトリーも同様に続く。

10.9.3 pop：現在のディレクトリーの列挙中止

メンバー関数 pop を呼ぶと、現在列挙中のディレクトリーの列挙を取りやめ、親ディレクトリーに戻る。現在のディレクトリーが初期ディレクトリーの場合、つまり depth() == 0 の場合は、終端イテレーターになる。

```
class recursive_directory_iterator {
public :
    void pop();
    void pop(error_code& ec);
} ;
```

たとえば、カレントディレクトリーが以下のようなディレクトリーツリーで、イテレーターが以下に書かれた順番でファイルを列挙する環境の場合、

```
a
b
b/a
b/c
b/d
c
d
```

以下のようなプログラムを実行すると、

```
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() ;
        }
    }
```

標準出力が指すファイルとその順番は以下のようになる。

```
a
b
b/a
c
d
```

“b/a” に到達した時点で pop() が呼ばれるので、それ以上のディレクトリー b 下の列挙が中止され、親ディレクトリーであるカレントディレクトリーに戻る。

10.9.4 recursion_pending：現在のディレクトリーの再帰をスキップ

disable_recursion_pending は現在のディレクトリーの下を再帰的に列挙することをスキップする機能だ。

```
class recursive_directory_iterator {
public :
    bool recursion_pending() const ;
    void disable_recursion_pending() ;
} ;
```

recursion_pending() は、直前のイテレーターのインクリメント操作の後に disable_recursion_pending() が呼ばれていない場合、true を返す。そうでない場合は false を返す。

言い換えれば、disable_recursion_pending() を呼んだ直後で、まだイテレーターのインクリメント操作をしていない場合、recursion_pending() は false を返す。

```
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() ;
    }
}
```

標準出力が指すファイルとその順番は以下のようになる。

```
a
b
c
d
```

このプログラムはディレクトリーであれば必ず disable_recursion_pending() が呼ばれるので、サブディレクトリーの再帰的な列挙は行われず、結果的に動作は directory_iterator と同じになる。

disable_recursion_pending を呼び出すことによって、選択的にディレクトリーの再帰的な列挙をスキップさせることができる。

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);
```

ファイルパス p のシンボリックリンクが解決され、正規化されたパスを返す。ファイルパスの正規化についての定義は長くなるので省略。

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);
```

ファイルパス base からファイルパス p に対する相対パスを返す。

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);
```

ファイルパス base からのファイルパス p に対する相対パスが空パスでなければ相対パスを返す。相対パスが空パスならば p が返る。

10.10.3　作成

create_directory

```
bool create_directory(const path& p);
bool create_directory(const path& p, error_code& ec) noexcept;
```

p の指すディレクトリーを 1 つ作成する。新しいディレクトリーが作成できた場合は true を、作成できなかった場合は false を返す。p が既存のディレクトリーを指していて新しいディレクトリーが作成できなかった場合はエラーにはならない。単に false が返る。

```
bool create_directory(
    const path& p, const path& existing_p);

bool create_directory(
    const path& p, const path& existing_p,
    error_code& ec) noexcept;
```

新しく作成するディレクトリー p のアトリビュートを既存のディレクトリー existing_p と同じものにする。

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;
```

ファイルパス to に解決されるハードリンク new_hard_link を作成する。

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;
```

ファイルパス from のファイルをファイルパス to にコピーする。

copy_options はコピーの挙動を変えるビットマスクの enum 型で、以下の enum 値がサポートされている。

名前	意味
none	デフォルト、ファイルがすでに存在する場合はエラー
skip_existing	既存のファイルを上書きしない。スキップはエラーとして報告しない
overwrite_existing	既存のファイルを上書きする
update_existing	既存のファイルが上書きしようとするファイルより古ければ上書きする

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;
```

ファイルパス from のファイルをファイルパス to にコピーする。

copy_options はコピーの挙動を変えるビットマスク型の enum 型で、以下の enum 値がサポートされている。

- サブディレクトリーに関する指定

名前	意味
none	デフォルト、サブディレクトリーはコピーしない
recursive	サブディレクトリーとその中身もコピーする

- シンボリックリンクに関する指定

名前	意味
none	デフォルト、シンボリックリンクをフォローする
copy_symlinks	シンボリックリンクをシンボリックリンクとしてコピーする。シンボリックリンクが指すファイルを直接コピーしない
skip_symlinks	シンボリックリンクを無視する

- コピー方法に関する指定

名前	意味
none	デフォルト、ディレクトリー下の中身をコピーする
directories_only	ディレクトリー構造のみをコピーする。非ディレクトリーファイルはコピーしない
create_symlinks	ファイルをコピーするのではなく、シンボリックリンクを作成する。コピー先がカレントディレクトリーではない 場合、コピー元のファイルパスは絶対パスでなければならない
create_hard_links	ファイルをコピーするのではなく、ハードリンクを作成する

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 を new_symlink にコピーする。

10.10.5 削除

remove

```
bool remove(const path& p);
bool remove(const path& p, error_code& ec) noexcept;
```

ファイルパス p の指すファイルが存在するのであれば削除する。ファイルがシンボリックリンクの場合、シンボリックリンクファイルが削除される。フォロー先は削除されない。

戻り値として、ファイルが存在しない場合 false を返す。それ以外の場合 true を返す。error_code でエラー通知を受け取る関数オーバーロードでは、エラーならば false が返る。

remove_all

```
uintmax_t remove_all(const path& p);
uintmax_t remove_all(const path& p, error_code& ec) noexcept;
```

ファイルパス p の下の存在するファイルをすべて削除した後、p の指すファイルも削除する。

つまり、p がディレクトリーファイルを指していて、そのディレクトリー下にサブディレクトリーやファイルが存在する場合、それらがすべて削除され、ディレクトリー p も削除される。

p がディレクトリーではないファイルを指す場合、p が削除される。

戻り値として、削除したファイルの個数が返る。error_code でエラー通知を受け取る関数オーバーロードの場合、エラーならば static_cast<uintmax_t>(-1) が返る。

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、□□□□□伴□□以下□□□□挙動□発生□□。
□□、□□□□前□ new_p □既存□□□□□□□指□□□□場合、□□□□□伴□□ 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 ;
}
```

もし、new_p が既存の空ディレクトリーを指していた場合、POSIX 準拠 OS であれば、リネームに伴って new_p は削除される。他の OS ではエラーになるかもしれない。

```
int main()
{
    using namespace std::experimental::filesystem ;

    create_directory("old_p") ;
    create_directory("new_p") ;

    // POSIX 準拠環境であればエラーにならないことが保証される
    rename("old_p", "new_p") ;
}
```

old_p がシンボリックリンクの場合、フォロー先ではなくシンボリックリンクファイルがリネームされる。

resize_file

```
void resize_file(   const path& p, uintmax_t new_size);
void resize_file(   const path& p, uintmax_t new_size,
                    error_code& ec) noexcept;
```

ファイルパス path の指すファイルのファイルサイズを new_size にする。

リサイズは POSIX の truncate() で行われたかのように振る舞う。つまり、ファイルを小さくリサイズした場合、余計なデータは捨てられる。ファイルを大きくリサイズした場合、増えたデータは null バイト（\0）でパディングされる。ファイルの最終アクセス日時も更新される。

10.10.7　情報取得

ファイルタイプの判定

ファイルタイプを表現する file_type 型の enum があり、その enum 値は以下のようになっている。

名前	意味
none	ファイルタイプが決定できないかエラー
not_found	ファイルが発見できなかったことを示す疑似ファイルタイプ
regular	通常のファイル
directory	ディレクトリーファイル
symlink	シンボリックリンクファイル
block	ブロックスペシャルファイル
fifo	FIFO もしくはパイプファイル
socket	ソケットファイル
unknown	ファイルは存在するがファイルタイプは決定できない

この他に、実装依存のファイルタイプが追加されている可能性がある。

ファイルタイプを調べるには、file_status のメンバー関数 type の戻り値を調べればよい。

以下のプログラムは、カレントディレクトリーに存在するファイル foo がディレクトリーかどうかを調べるコードだ。

```
int main()
{
    using namespace std::filesystem ;

    auto s = status("./foo") ;
    bool b = s.type() == file_type::directory ;
}
```

また、status もしくは path からファイルタイプがディレクトリーであるかどうかを判定できる is_directory も用意されている。

```
int main()
{
    using namespace std::filesystem ;

    bool b1 = is_directory("./foo") ;

    auto s = status("./foo") ;
    bool b2 = is_directory(s) ;
}
```

file_status はファイル情報をキャッシュするので、物理ファイルシステムに変更を加えない状態で、同じファイルに対して何度もファイル情報を取得する場合は、

file_status を使ったほうがよい。

このような is_x という形式のフリー関数は、いずれも以下の形式を取る。

```
bool is_x(file_status s) noexcept;
bool is_x(const path& p);
bool is_x(const path& p, error_code& ec) noexcept;
```

以下はフリー関数の名前と、どのファイルタイプであるかを判定する表だ。

名前	意味
is_regular_file	通常のファイル
is_directory	ディレクトリーファイル
is_symlink	シンボリックリンクファイル
is_block	ブロックスペシャルファイル
is_fifo	FIFO もしくはパイプファイル
is_socket	ソケットファイル

また、単一のファイルタイプを調べるのではない以下のような名前のフリー関数が存在する。

名前	意味
is_other	ファイルが存在し、通常のファイルでもディレクトリーでもシンボリックリンクでもないタイプ
is_empty	ファイルがディレクトリーの場合、ディレクトリー下が空であれば true を返す。ファイルが非ディレクトリーの場合、ファイルサイズが 0 であれば true を返す。

status

```
file_status status(const path& p);
file_status status(const path& p, error_code& ec) noexcept;
```

ファイルパス p のファイルの情報を格納する file_status を返す。
p がシンボリックリンクの場合、フォロー先のファイルの file_status を返す。

status_known

```
bool status_known(file_status s) noexcept;
```

s.type() != file_type::none を返す。

symlink_status

```
file_status symlink_status(const path& p);
file_status symlink_status(const path& p, error_code& ec) noexcept;
```

status と同じだが、p がシンボリックリンクの場合、そのシンボリックリンクファイルの status を返す。

equivalent

```
bool equivalent(const path& p1, const path& p2);
bool equivalent(const path& p1, const path& p2,
                error_code& ec) noexcept;
```

p1 と p2 が物理ファイルシステム上、同一のファイルである場合、true を返す。そうでない場合 false を返す。

exists

```
bool exists(file_status s) noexcept;
bool exists(const path& p);
bool exists(const path& p, error_code& ec) noexcept;
```

s, p が指すファイルが存在するのであれば true を返す。そうでない場合 false を返す。

file_size

```
uintmax_t file_size(const path& p);
uintmax_t file_size(const path& p, error_code& ec) noexcept;
```

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

●本書に対するお問い合わせは、電子メール（info@asciidwango.jp）にてお願いいたします。
但し、本書の記述内容を越えるご質問にはお答えできませんので、ご了承ください。

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　株式会社ドワンゴ
〒 104-0061
東京都中央区銀座 4-12-15 歌舞伎座タワー
編集 03-3549-6153
電子メール info@asciidwango.jp
http://asciidwango.jp/

発　売　株式会社 KADOKAWA
〒 102-8177
東京都千代田区富士見 2-13-3
営業 0570-002-301（カスタマーサポート・ナビダイヤル）
受付時間 9：00〜17:00（土日 祝日 年末年始を除く）
http://www.kadokawa.co.jp/

印刷・製本　株式会社リーブルテック

Printed in Japan

落丁・乱丁本はお取り替えいたします．下記 KADOKAWA 読者係までご連絡ください．
送料小社負担にてお取り替えいたします。
但し、古書店で本書を購入されている場合はお取り替えできません．
電話　049-259-1100（9:00-17:00/土日、祝日、年末年始を除く）
〒354-0041　埼玉県入間郡三芳町藤久保 550-1
定価はカバーに表示してあります．

ISBN: 978-4-04-xxxxxx-x

アスキードワンゴ編集部
編　集　星野浩章