C++17 の新機能

2017-02-28

目次

序

本書が出版される頃には、すでに C++17 規格が正式に制定されているだろう。本書は C++17 の新機能をいち早く解説するために書かれた。

C++ の規格

プログラミング言語 C++ は ISO の傘下で国際規格 ISO/IEC 14882 として制定されている。この規格は数年おきに改定されている。一般に C++ の規格を参照するときは、規格が制定した西暦の下二桁をとって、C++98(1998 年発行) とか C++11(2011 年発行) と呼ばれている。現在発行されている C++ の規格は以下の通り。

C++98

C++98 は 1998 年に制定された最初の C++ の規格である。本来ならば 1994 年か 1995 年には制定させる予定が大幅にずれて、1998 年となった。

C++03

C++03 は C++98 の文面の曖昧な点を修正したマイナーアップデートとして 2003 年に制定された。新機能の追加はほとんどない。

C++11

C++11 は制定途中のドラフト段階では元 C++0x と呼ばれていた。これは、200x 年までに規格が制定される予定だったからだ。予定は大幅に遅れ、ようやく規格が制定された時にはすでに 2011 年の年末になっていた。C++11 ではとても多くの新機能が追加された。

C++14

C++14 は 2014 年に制定された。C++11 の文面の誤りを修正した他、すこし新機能が追加された。本書で解説する。

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。

C++ の将来の規格

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。

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。

SD-6 C++ のための機能テスト推奨

C++17 には機能テストのための C プリプロセッサー機能が追加された。

機能テストマクロ

機能テストというのは、C++ の実装 (C++ コンパイラー) が特定の機能をサポートしているかどうかをコンパイル時に判断できる機能だ。本来、C++17 の規格に準拠した C++ 実装は、C++17 の機能をすべてサポートしているべきだ。しかし、残念ながら現実の C++ コンパイラーの開発はそのようには行われていない。C++17 に対応途中の C++

コンパイラーは将来的にはすべての機能を実装することを目標としつつも、現時点では一部の機能しか実装していないという状態になる。

例えば、C++11 で追加された rvalue リファレンスという機能に現実の C++ コンパイラーが対応しているかどうかをコンパイル時に判定するコードは以下のようになる。

```
#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。あとは、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。

例えば 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時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 ) ;

#ifndef __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。

__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。

例えば、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()
{
    // エラー
    if ( __has_include(<vector>) )
    { }
}
```

___has_cpp_attribute 式

C++ 実装が特定の属性トークンをサポートしているかどうかをしらべるには、___has_cpp_attribute 式が使える。

```
__has_cpp_attribute( 属性トークン )
```

___has_cpp_attribute 式は、属性トークンが存在する場合は属性トークンが標準規格に採択された年と月を表す数値に、存在しない場合は 0 に置換される。

```
// [[nodiscard]] がサポートされている場合は使う
#if __has_cpp_attribute(nodiscard)
[[nodiscard]]
#endif
void * allocate_memory( std::size_t size ) ;
```

___has_include 式と同じく、___has_cpp_attribute 式も#if か#elif の中でしか使えない。#ifdef で___has_cpp_attribute 式の存在の有無を判定できる。

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づけで積極的な新機能の追加は見送られたからだ。

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

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 と 1000'000'000 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。

[[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
// どちらでもよい
[[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 ) ;
```

機能テストマクロは___has_cpp_attribute(deprecated)。値は 201309。

通常の関数の戻り値の型推定

関数の戻り値の型として auto を指定すると、戻り値の型を return 文から推定してくれる。

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

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

return 文の型が一致していないとエラーとなる。

```
auto f()
{
    return 0 ; // エラー、一致してない
    return 0.0 ; // エラー、一致していない
}
```

すでに型が決定できる return 文が存在する場合、関数の戻り値の型を参照するコードも書ける。

```
auto a()
{
    &a ; // エラー、a の戻り値の型が決定していない
    return 0 ;
}

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

関数 a へのポインターを使うには関数 a の型が決定していなければならないが、return 文の前に型は決定できないので関数 a はエラーになる。関数 b は return 文が現れた後な

ので戻り値の型が決定できる。

再帰関数も書ける。

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

このコードも、return 文の順番を逆にすると戻り値の型が決定できずエラーとなるので注意。

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

機能テストマクロは__cpp_return_type_deduction。値は 201304。

decltype(auto): 厳格な auto

警告：この項目は C++ 規格の詳細な知識を解説しているため極めて難解になっている。平均的な C++ プログラマーはこの知識を得てもよりよいコードが書けるようにはならない。この項目は読み飛ばすべきである。

decltype(auto) は auto 指定子の代わりに使える厳格な auto だ。利用には C++ の規格の厳格な理解が求められる。

auto と decltype(auto) は型指定子と呼ばれる文法の一種で、プレイスホルダー型として使う。

わかりやすく言うと、具体的な型を式から決定する機能だ。

```
// a は int
auto a = 0 ;
```

```
// b は int
auto b() { return 0 ; }
```

変数宣言にプレイスホルダー型を使う場合、型を決定するための式は初期化子と呼ばれる部分に書かれる式を使う。関数の戻り値の型推定にプレイスホルダー型を使う場合、return 文の式を使う。

decltype(auto) は auto の代わりに使うことができる。delctype(auto) も型を式から決定する。

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

一見すると auto と decltype(auto) は同じようだ。しかし、この 2 つは式から型を決定する方法が違う。どちらも C++ の規格の極めて難しい規則に基づいて決定される。習得には熟練の魔法使いであることが要求される。

auto が式から型を決定するには、auto キーワードをテンプレートパラメーター名で置き換えた関数テンプレートの仮引数に、式を実引数として渡してテンプレート実引数推定を行わせた場合に推定される型が使われる。

例えば

```
auto x = 0 ;
```

の場合は、

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

のような関数テンプレートに対して、

```
f(0) ;
```

と実引数を渡した時に u の型として推定される型と同じ型になる。

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

の場合には、

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

のような関数テンプレートに

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

と実引数を渡した時に u の型として推定される型と同じ型になる。この場合は int const * になる。

ここまでが auto の説明だ。decltype(auto) の説明は簡単だ。

decltype(auto) の型は、auto を式で置き換えた decltype の型になる。

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

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

上のコードは、下のコードと同じ意味だ。

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

ここまでは簡単だ。そして、これ以降は黒魔術のような C++ の規格の知識が必要になってくる。

auto と decltype(auto) は一見すると同じように見える。型を決定する方法として、auto は関数テンプレートの実引数推定を使い、decltype(auto) は decltype を使う。どちらも式を評価した結果の型になる。一体何が違うというのか。

主な違いは、auto は関数呼び出しを使うということだ。関数呼び出しの際には様々な暗黙の型変換が行われる。

例えば、配列を関数に渡すと、暗黙の型変換の結果、配列の先頭要素へのポインターになる。

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

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

では auto と decltype(auto) を使うとどうなるのか。

```
int array[5] ;
// int *
auto x1 = array ;
// エラー、配列は配列で初期化できない
decltype(auto) x2 = array ;
```

このコードは、以下と同じ意味になる。

```
int array[5] ;
// int *
int * x1 = array ;
// エラー、配列は配列で初期化できない
int x2[5] = array ;
```

auto の場合、型は int *となる。配列は配列の先頭要素へのポインターへと暗黙に変換できるので、結果のコードは正しい。

decltype(auto) の場合、型は int [5] となる。配列は配列で初期化、代入ができないので、このコードはエラーになる。

関数型も暗黙の型変換により関数へのポインター型になる。

```
void f() ;

// 型は void(*)()
auto x1 = f ;
// エラー、関数型は変数にできない
decltype(auto) x2 = f ;
```

auto はトップレベルのリファレンス修飾子を消すが、decltype(auto) は保持する。

```
int & f()
```

```
{
    static int x ;
    return x ;
}

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

auto はトップレベルの CV 修飾子を消すが、decltype(auto) は保持する。

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

リスト初期化は auto では std::initializer_list だが、decltype(auto) では式ではないためエラー

```
int main()
{
    // std::initializer_list<int>
    auto x1 = { 1,2,3 } ;
    // エラー、decltype({1,2,3}) はできない
    decltype(auto) x2 = { 1,2,3 } ;
}
```

decltype(auto) は単体で使わなければならない。

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

この他にも auto と decltype(auto) には様々な違いがある。すべての違いを列挙するのは煩雑なので省略するが、decltype(auto) は式の型を直接使う。auto は大抵の場合は便利な型の変換が入る。

auto は便利で大抵の場合はうまく行くが暗黙の型の変換が入るため、意図通りの推定をしてくれないことがある。

例えば、引数でリファレンスを受け取り、戻り値でそのリファレンスを返す関数を書くとする。以下のように書くのは間違いだ。

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

なぜならば、戻り値の型は式の型から変化して int になってしまうからだ。ここで decltype(auto) を使うと、

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

式の型をそのまま使ってくれる。

ラムダ式に delctype(auto) を使う場合は以下のように書く。

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

decltype(auto) は主に関数の戻り地の型推定で式の型をそのまま推定してくれるようにするために追加された機能だ。その利用には C++ の型システムの深い理解が必要になる。

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

ジェネリックラムダ

ジェネリックラムダはラムダ式の引数の型を書かなくてもすむようにする機能だ。

通常のラムダ式は以下のように書く。

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

ラムダ式の引数には型が必要だ。しかし、クロージャーオブジェクトの operator () に渡す型はコンパイル時にわかる。コンパイル時にわかるということはわざわざ人間が指定する必要はない。ジェネリックラムダを使えば、引数の型を書くべき場所に auto キーワードを書くだけで型を推定してくれる。

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

ジェネリックラムダ式の結果のクロージャー型には呼出しごとに違う型を渡すことができる。

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

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

仕組みは簡単で、以下のようなメンバーテンプレートの operator () を持ったクロージャーオブジェクトが生成されているだけだ。

```
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。

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 ;
        val ;
    } ;
}
```

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&をつけると、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 は 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全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 ;
    // ムーブ
```

```
    return [str = std::move(str)]{ return str ; } ;
}
```

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

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通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。

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。例えば円周率を 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時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買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> ;
}
```

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。

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 文一つしか書けない。

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 )
{
    // 変数を宣言できる。
    int sum = 0 ;

    // 繰り返し文を書ける。
    for ( int i = 1 ; i < x ; ++i )
    {
        // 変数を変更できる
        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。

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
{
    // メンバー初期化子
    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もつ非 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。

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。

例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。

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。

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。

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 ; // 整数リテラル
0x1.0 ; // エラー、指数部がない
```

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の仮数部は指数部の 2 の階乗で乗算される。

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 (例えば 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。

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 をつける。UTF-8 文字リテラルは UTF-8 のコード単位一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。

```
// エラー
// 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。

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、無例外指定のつ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()
{
    // 無例外指定のついたポインター
    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, &f は例外指定のある関数へのポインター型
void (*p4)() noexdept = &f ;
```

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

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) は 1 番目の引数を x で、残りをパラメーターパック args で受け取る。そして、x + sum(args …) を返す。すると、sum(args …) はまた sum(x, args) に渡されて、1 番目の引数、つまり最初から見て 2 番目の引数が x に入り、また sum が呼ばれる。このような再帰的な処理を繰り返していく。

そして、引数がひとつだけになると、可変長テンプレートではない sum が呼ばれる。これは重要だ。なぜならば可変長テンプレートは 0 個の引数を取ることができるので、そのまま可変長テンプレート版の sum が呼ばれてしまうと、次の sum の呼び出しができずにエラーとなる。これを回避するために、また再起の終了条件のために、引数がひとつの sum のオーバーロード関数を書いておく。

可変長テンプレートでは任意個の引数に対応するために、このような再帰的なコードが必

須になる。

しかし、ここで実現したいこととは N 個あるパラメーターパック args の中身に対して、仮に N 番目を args#N とする表記を使うと、args#0 + args#1 + … + args#N のような展開をしたいだけだ。C++17 の fold 式はパラメーターパックに対して二項演算子を適用する展開を行う機能だ。

fold 式を使うと sum は以下のように書ける。

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

(… + args) は、args#0 + args#1 + … + args#N のように展開される。

fold 式には、単項 fold 式と二項 fold 式がある。そして、演算子の結合順序に合わせて左 fold と右 fold がある。

fold 式は必ず括弧で囲まなければならない。

```
template < typename ... Types >
auto sum( Types ... args )
{
    // fold式
    ( ... + args )  ;
    // エラー、括弧がない
    ... + 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)
    return ( ... + f(args) )  ;
}
```

これは f(args) というパターンが展開される。

fold-operator にはいか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) 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)))) 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
    // (0+(1+(2+(3+(4+5)))))
    auto right = ( args + ... + 0 ) ;
}

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

fold 式はパラメーターパックのそれぞれに二項演算子を適用したい時にわざわざ複雑な再帰的テンプレートを書かずにすむ方法を提供してくれる。

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

ラムダ式で*this のコピーキャプチャー

C++17 ではラムダ式で*this をコピーキャプチャーできるようになった。*this をコピーキャプチャーするには、ラムダキャプチャーに*this と書く。

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

int main()
{
    std::function <int () > f ;
    {
        X x ;
        f = x.get() ;
    }// x の寿命はここまで

    // コピーされているので問題ない
    int data = f() ;
}
```

コピーキャプチャーする*this は lambda 式が書かれた場所の*this だ。

また、以下のようなコードで挙動の違いをみるとわかりやすい。

```
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
        [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。~c++
class closure_object { X * this_ptr ;

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

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

} ; ~

二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 = 1 ;
    }
} ;
```

三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。

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。

```
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 はこのメンバー関数 f を呼び出したオブジェクトへのアドレス
        std::printf("%p\n", this ) ;

        // this はコピーされた別のオブジェクトへのアドレス
        [*this](){  std::printf("%p\n", this) ;  }() ;
   }
} ;

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

この場合、出力される 2 つのポインターの値は異なる。

ラムダ式での*this のコピーキャプチャーは名前通り*this のコピーキャプチャーを提供する提案だ。同等の機能は初期化キャプチャーでも可能だが、表記が冗長で間違いの元だ。

```
struct X
{
    int data ;

    auto f()
    {
        return [ tmp = *this ] { return this->data ; } ;
    }
} ;
```

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

constexpr ラムダ式

C++17 ではラムダ式が constexpr になった。より正確に説明すると、ラムダ式のクロージャーオブジェクトの operator () は条件を満たす場合 constexpr になる。

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

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

constexpr の条件を満たす　ラムダ式はコンパイル時定数を必要とする場所で使うことができる。例えば constexpr 変数や配列の添字や static_assert などだ。

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

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

constexpr の条件を満たすのであれば、キャプチャーもできる。

```
int main()
{
    int a = 0 ; // 実行時の値
    constexpr int b = 0 ; // コンパイル時定数

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

    // エラー、constexpr の条件を満たさない
    constexpr int c = f() ;

    // Ok、constexpr の条件を満たす
    constexpr int d = f() ;
}
```

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。

文字列なし static_assert

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

```
static_assert( true ) ;
```

C++11 で追加された static_asser 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。

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。

[[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 )
{
    swtich ( x )
    {
    case 0 :
        // 処理 0
        break ;
    case 1 :
        // 処理 1
        break ;
    case 2 :
        // 処理 2
        break ;
    default :
        // x がいずれでもない場合の処理
```

```
        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 の時は処理 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欲しい場合、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。

[[nodiscard]] 属性

[[nodiscard]] 属性は関数の戻り値が無視されてほしくない時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()
{
    // エラー、戻り値が無視されている
    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時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
int do_something_that_may_fail()
{
    // 処理

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

```
    // 処理

    return error_code::no_error ;
}

// エラーが一切発生しなかった時の処理
int do_something_on_no_error() ;

int main()
{
    // エラーを確認していない
    do_soemthing() ;

    // エラーがない前提で次の処理をしようとする
    do_something_on_no_error() ;
}
```

関数に [[nodiscard]] 属性を付与しておけば、このようなユーザー側の初歩的なエラー確認の欠如に警告メッセージを出せる。

[[nodiscard]] 属性は、クラスと enum にも付与することができる。

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

[[nodiscard]] が付与されたクラスか enum が戻り値の型である関数は [[nodiscard]] が付与された扱いとなる。

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

X f() { return X{} ; }

int main()
{
    // 警告、戻り値が無視されている
    f() ;
}
```

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

[[maybe_unused]] 属性

[[maybe_unused]] 属性は名前やエンティティが意図的に使われないことを示すのに使える。

現実の C++ のコードでは、宣言されているのにソースコードだけを考慮するとどこからも使われていないように見える名前やエンティティが存在する。

```
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全てではない。様々な理由で reserved のような一見使われていない変数が必要になる。

例えば、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
// どちらでもよい
[[maybe_unused]] int variable_name1{};
int variable_name2 [[maybe_unused]] { } ;

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

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

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

演算子のオペランドの評価順序の固定

C++17 では演算子のオペランドの評価順序が固定された。

以下の式は、a, b, c, d の順番に評価されることが規格上保証される。

```
a.b
```

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

つまり、

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

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

と書いた場合、関数 f がまず先に呼び出されて、次に関数 g が呼び出されることが保証される。

関数呼び出しの実引数のオペランド b1, b2, b3 の評価順序は未規定のままだ。

これにより、既存の未定義の挙動となっていたコードの挙動が定まる。

constexpr if 文 : コンパイル時条件分岐

constexpr if 文はコンパイル時の条件分岐ができる機能だ。

constexpr if 文は、通常の if 文を if constexpr で置き換える。

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

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

constexpr if 文という名前だが、実際に記述するときは if constexpr だ。

コンパイル時の条件分岐とは何を意味するのか。以下は constexpr if が**行わないもの**の一覧だ。

- 最適化
- 非テンプレートコードにおける挙動の変化

コンパイル時の条件分岐の機能を理解するには、まず C++ の既存の条件分岐について理解する必要がある。

実行時の条件分岐

通常の実行時の条件分岐は、実行時の値を取り、実行に条件分岐を行う。

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

この場合、runtime_value が true の場合は関数 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。こんどは完全なコードをみてみよう。

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

// do_false_thing の宣言は存在しない

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

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。

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 の宣言
void do_true_thing() ;

// do_false_thing の宣言は存在しない

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()
{
// エラー
// 名前 f はプリプロセッサーマクロではない。
# if f()
    do_true_thing() ;
#else
    do_false_thing() ;
#endif
}
```

コンパイル時の条件分岐

コンパイル時の条件分岐とは、分岐の条件にコンパイル時計算の結果を使い、かつ、選択されない分岐にコンパイルエラーが含まれていても、使われないのでコンパイルエラーにはならない条件分岐のことだ。

たとえば、std::distance という標準ライブラリを実装してみよう。std::distance(first, last) は、イテレーター first と last の距離を返す。

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

残念ながら、この実装は Iterator がランダムアクセスイテレーターの場合にしか動かない。入力イテレーターに対応させるには、イテレーターを一つづつインクリメントして last と等しいかどうか比較する実装が必要になる。

```
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 ;
}
```

残念ながら、この実装は Iterator にランダムアクセスイテレーターを渡した時に効率が悪い。

ここで必要な実装は、Iterator がランダムアクセスイテレーターならば last - first を使い、そうでなければ地道にインクリメントする遅い実装を使うことだ。Iterator がランダ

ムアクセスイテレーターかどうかは、以下のコードを使えば、is_random_access_iterator<iterator>で確認できる。

```
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 > ;
```

すると、distance は以下のように書けるのではないか。

```
// ランダムアクセスイテレーターかどうかを判定するコード
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。コンパイラはー、

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

超上級者向け解説

constexpr if は、実はコンパイル時条件分岐ではない。テンプレートの実体化時に、選択されないブランチのテンプレートの実体化の抑制を行う機能だ。

constexpr if によって選択されない文は discarded statement となる。discarded statement はテンプレートの実体化の際に実体化されなくなる。

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

template < typename T >
int f(T x)
{
    if constexpr ( std::is_same_v<T, std::decay_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> )
    {
        // 常にコンパイルエラー
        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 文の条件に会うときにだけ 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 の条件に会うときにだけ発動する static_assert が書ける。

```
template  <typename ... >
using false_t = false ;

template < typename T >
void f()
{
    if constexpr ( E1 )
        if constexpr ( E2 )
            if constepxr ( E3 )
            {
                static_assert( false_t<T> ) ;
            }
}
```

このように false_t 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。

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。例えば以下のコードはエラーになる

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

// do_false_thing の宣言は存在しない

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

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。

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。

例えば、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 の型が X ならば特別な処理を行いたい
```

```
    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 の階乗を返す
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 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。

初期化文つき条件文

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 :
        /*... */;
}
```

これは、以下のコードと同じ意味になる。

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

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

なぜこのような機能が追加されたかというと、変数を宣言し、if 文の条件に変数を使い、if 文を実行後は変数を使用しない、というパターンは現実のコードで頻出するからだ。

```
void * ptr = std::malloc(10) ;
if ( ptr != nullptr )
{
    // 処理
    std::free(ptr) ;
}
// これ以降 ptr は使わない

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、初期化文つきの条件文が追加された。

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

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 *>
}
```

実引数ガイド

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。

例えば、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。

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

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。

```
// [[extention::foo, extention::bar]] と同じ
[ using extention : foo, bar ]] int x ;
```

属性トークンには、属性名前空間をつけることができる。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。

例えば、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。

```
[[ extention::foo, extention::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。

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、無視される
[[ 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、結局 C プリプロセッサーを使うか、煩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。

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 [a,b,c] = a ;

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

C++ では、様々な方法で多値を扱うことができる。例えば配列、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++ の関数は配列以外の多値を返すことができる。

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

多値を受け取るには、これまでは多値を塊として受け取るか、ライブラリで分解して受け取るしかなかった。

多値を塊で受け取るには以下のように書く。

```
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 ;
}
```

多値をライブラリで受け取るには以下のように書く。

```
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 は int 型、値は 1
    // b は int 型、値は 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 ; )
    { }
}
```

構造化束縛は range-based for 文にも使える。

```
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' ;
    }
}
```

これは、map の要素型 std::pair<const std::string, std::string>を構造化束縛 [key, value] で受けている。

構造化束縛は配列にも使える。

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

構造化束縛はクラスにも使える。

```
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ひとつのクラスの 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()
{
    int expr[] = { 1,2 } ;

    // エラー
    constexpr auto [a,b] = expr ;
}
```

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） リファレンス修飾子（省略可） [識別子リスト] ;

識別子リスト:
    コンマで区切られた識別子

初期化子:
    = 式
    { 式 }
    (式)

以下は単純宣言のコード例だ。

```
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"の例
    auto [a,b,c] = e1 ;
    auto [d,e,f] = e2 ;
    auto [g,h,i] = e3 ;

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

    // CV 修飾子とリファレンス修飾子を使う例
    auto const & [p,q,r] = e1 ;
}
```

以下は for-range 宣言の例だ。

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

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

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個の値を持つ
    int expr[] = {1,2} ;

    // エラー、変数が少なすぎる
    auto[a] = expr ;
    // エラー、変数が多すぎる
```

```
    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、記述された通りの属性、CV 修飾子、リファレンス修飾子の変数が宣言される。

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] ;
}
```

リファレンス修飾子がある場合、変数はリファレンスとなる。

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

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

もし、変数の型が配列の場合、配列の要素はそれぞれ対応する配列の要素で初期化される。

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

これは、以下と同じ意味になる。

```
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] } ;
}
```

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

構造化束縛宣言の初期化子の型 E が配列ではない場合で、std::tuple_size<E>が完全形の名前である場合、

構造化束縛宣言の初期化子の型を E、その値を e とする。構造化束縛宣言で宣言されるひとつ目の変数を 0, ふたつ目の変数を 1...とインクリメントされていくインデックスを i とする。

std::tuple_size<E>::value は整数のコンパイル時定数式で、その値は初期化子の値の数

でなければならない。

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

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

それぞれの値を取得するために、非修飾名 get が型 E のクラススコープから探される。get が見つかった場合、それぞれの変数の初期化子は e.get<i>() となる。

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

という構造化束縛宣言は、以下の意味になる。

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

そのような get の宣言が見つからない場合、初期化子は get<i>(e) となる。この場合、get は連想名前空間から探される。通常の非修飾名前検索は行われない。

```
// ただし通常の非修飾名前検索は行われない。
type a = get<0>(e) ;
type b = get<1>(e) ;
type c = get<2>(e) ;
```

構造化束縛宣言で宣言される変数の型は以下のように決定される。

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

```
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) ;
```

以下のコードは、

```
int main()
{
```

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

以下とほぼ同等の意味になる。

```
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) ;
}
```

以下のコードは、

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

以下のような意味になる。

```
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)) ;
}
```

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。

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。

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、例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。

例えば、ある関数をコンパイルした結果のコードサイズが 1KB 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 1MB 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。

例えば、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時、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時、関数の実行時間が十分に短い時、タイトなループの中など、様々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。

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++ コンパイラーは inline キーワードを無視する。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、もうひとつの意味がある。ODR(One Definition Rule、定義はひとつの原則) の回避だ。

C++ では、定義はプログラム中にひと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() { } // エラー、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。定義はどこかひと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()
{
    // エラー、定義がない
    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() ;
}
```

これは例のために同一の inline 関数を直接記述しているが、inline 関数は定義を同一性を保証させるため、通常はヘッダーファイルに書いて#include して使う。

inline 変数の意味

inline 変数は、ODR に違反せず変数の定義の重複を認める。同じ名前の inline 変数は同じ変数を指す。

```
// 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。

可変長 using 宣言

この機能は超上級者向けだ。

C++17 では using 宣言をカ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ついていないが、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()
}
```

機能テストマクロは___cpp_variadic_using, 値は 201611。

std::byte バイトを表現する型

C++17 では、バイトを表現する型が入った。std::byte は<cstddef>で定義されている。ライブラリでもあるのだがコア言語で特別な型として扱われている。

バイトとは C++ のメモリモデルにおけるストレージの単位で、C++ においてユニークなアドレスが付与される最小単位だ。C++ の規格は未だに 1 バイトが具体的に何ビットであるのかを規定していない。これは過去にバイトのサイズが 8 ビットではないアーキテクチャが存在したためだ。

バイトのビット数は<climits>で定義されているプリプロセッサーマクロ、CHAR_BIT で知ることができる。

C++17 では、1 バイトは UTF-8 の 8 ビットの 1 コード単位をすべて表現できると規定している。

C++ では、1 バイトのストレージは unsigned char 型か std::byte 型で表現できる。複数バイトが連続するストレージは、unsigned char の配列型、もしくは std::byte の配列型として表現できる。

std::byte 型は、<cstddef>で以下のように定義されている。

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

std::byte はライブラリとして scoped enum 型で定義されている。これにより、他の整数

型からの暗黙の型変換が行えない。

値 0x12 の std::byte 型の変数は以下のように定義できる。

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

std::byte 型の値が欲しい場合は、以下のように書くことができる。

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

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

std::byte 型は他の数値型からは暗黙に型変換できない。これによりうっかりと型を取り違えてバイト型と他の型を演算してしまうことを防ぐことができる。

```
int main()
{
    // エラー、() による初期化は int 型からの暗黙の変換が入る
    std::byte b1(1) ;

    // エラー、=による初期化は int 型からの暗黙の変換が入る
    std::byte b2 = 1 ;

    std::byte b{} ;

    // エラー、operator =による int 型の代入は暗黙の変換が入る
    b = 1 ;
    // エラー、operator =による double 型の代入は暗黙の変換が入る
    b = 1.0 ;
}
```

std::byte 型は{}によって初期化するが、縮小変換を禁止するルールにより、std::byte 型が表現できる値の範囲でなければエラーとなる。

例えば、今 std::byte が 8 ビットで、最小値が 0、最大値が 255 の環境だとする。

```
int main()
{
    // エラー、表現できる値の範囲ではない
    std::byte b1{-1} ;
    // エラー、表現できる値の範囲ではない
    std::byte b2{256} ;
}
```

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 は 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 型の値は 42
    auto i = std::to_integer<int>(b) ;
}
```

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。

variant : 型安全な union

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::hold_alternative<int>( x ) ;
    // std::string が入っているか確認
    // true を返す
    bool has_string = std::hold_alternative<std::string> ( x ) ;

    // 入っている値を得る
    // "hello"
    std::string str = std::get<std::string>(x) ;
}
```

型非安全な古典的 union

C++ が従来から持っている古典的な union は、複数の型のいずれかひとつだけの値を格納する型だ。union のサイズはデータメンバーのいずれかの型をひとつ表現できるだけのサイズとなる。

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

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

この場合、sizeof(U) のサイズは

$$\text{sizeof(U)} = \max\{\text{sizeof(int)}, \text{sizeof(double)}, \text{sizeof(std::string)}\} + \text{パディングなど}$$

になる。sizeof(S) のサイズは、

$$\text{sizeof(S)} = \text{sizeof(int)} + \text{sizeof(double)} + \text{sizeof(std::string)} + \text{パディングなど}$$

になる。

union はメモリ効率がよい。union は variant と違い型非安全だ。どの型の値を保持しているかという情報は保持しないので、利用者が適切に管理しなければならない。

試しに、冒頭のコードを union で書くと、以下のようになる。

```
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利用者の責任に任せる
    ~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
    // 破棄も考えなくて良い
    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") ;

    // 利用者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。

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 ;
```

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 を保持
    std::variant< Empty, A, B, C > v ;
}
```

このような場合に、Empty のようなクラスをわざわざ独自に定義するのは面倒なので、標準ライブラリには std::monostate クラスが以下のように定義されている。

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

したがって、上の例は以下のように書ける。

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

std::monostate は variant の最初のテンプレート実引数として使うことで variant をデフォルト構築可能にするための型だ。それ以上の意味はない。

■コピー初期化　variant に同じ型の variant を渡すと、コピー/ムーブする。

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

■variant のコンストラクターに値を渡した場合　variant のコンストラクターに上記以外の値を渡した場合、variant のテンプレート実引数に指定した型の中から、オーバーロード解決により最適な型が選ばれ、その型の値に変換され、値を保持する。

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

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

    // std::string
    // char const *型は std::string 型に変換される。
    val c("hello") ;

    // int
    // char 型は Integral promotion により int 型に優先的に変換される
    val d( 'a' ) ;
}
```

■in_place_type による emplace 構築　variant のコンストラクターの第一引数に std::in_place_type<T>を渡すことにより、T 型の要素を構築するために T 型のコンストラクターに渡す実引数を指定できる。

ほとんどの型はコピーかムーブができる。

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

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

しかし、もし型 X がコピーもムーブもできない型だったとしたら、上記のコードは動かない。

```
struct X
```

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

int main()
{
    // X を構築
    X x( 1, 2, 3 ) ;
    // エラー、X はコピーできない
    std::variant<X> v( x ) ;
}
```

このような場合、variant が内部で X を構築する際に、構築に必要なコンストラクターの実引数を渡して、variant に X を構築させる必要がある。そのために std::in_place_type<T> が使える。T に構築したい型を指定して第一引数とし、第二引数以降を T のコンストラクターに渡す値にする。

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

int main()
{
    // X の値を構築して保持
    std::variant<X> v( std::in_place_type<X>, 1, 2, 3 ) ;
}
```

variant の破棄

variant のデストラクターは、そのときに保持している値を適切に破棄してくれる。

```
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 のデストラクターは deque<int>を破棄する
}
```

variant のユーザーは何もする必要がない。

variant の代入

variant の代入はとても自然だ。variant を渡せばコピーするし、値を渡せばオーバーロード解決に従って適切な型の値を保持する。

variant の emplace

variant は emplace をサポートしている。variant の場合、構築すべき型を知らせる必要があるので、emplace<T>の T で構築すべき型を指定する。

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

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

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

variant に値が入っているかどうかの確認

■valueless_by_exception メンバー関数

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

valueless_by_exception メンバー関数は、variant が値を保持している場合、false を返す。

```
void f( std::variant<int> & v )
{

    if ( v.valueless_by_exception() )
    { // true
        // v は値を保持していない
    }
    { // false
        // v は値を保持している
    }
}
```

variant はどの値も保持しない状態になることがある。例えば、std::string はコピーにあたって動的なメモリ確保を行うかもしれない。variant が std::string をコピーする際に、動的メモリ確保に失敗した場合、コピーは失敗する。なぜならば、variant は別の型の値を構築する前に、以前の値を破棄しなければならないからだ。variant は値を持たない状態になりうる。

```
int main()
{
    std::variant< int, std::string > v ;
    try {
        std::string s("hello") ;
        v = s ; // 動的メモリ確保が発生するかもしれない
    } catch( std::bad_alloc e )
    {
    // 動的メモリ確保が失敗するかもしれない
    }

    // 動的メモリ確保の失敗により
    // true になるかもしれない
    bool b = v.valueless_by_exception() ;
}
```

■index メンバー関数

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

index メンバー関数は、variant に指定したテンプレート実引数のうち、現在 variant が保持している値の型を 0 ベースのインデックスで返す。

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

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

もし variant が値を保持しない場合、つまり valueless_by_exception() が true を返す場合は、std::variant_npos を返す。

```
// variant が値を持っているかどうか確認する関数
template < typename ... Types  >
void has_value( std::variant< Types ... > && v )
{
    return v.index() != std::variant_npos ;

    // これでもいい
    // return v.valueless_by_exception() == false ;
}
```

std::variant_npos の値は-1 だ。

swap

variant は swap に対応している。

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

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

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

std::variant_size は、T に variant 型を渡すと、variant が保持できる型の数を返してくれる。

```
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 は integral_constant を基本クラスに持つクラスなので、デフォルト構築した結果をユーザー定義変換することでも値を取り出せる。

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

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

variant_size を以下のようにラップした変数テンプレートも用意されている

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

これを使えば、以下のようにも書ける。

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

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

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 ;
// ing
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;
```

これをつかえば、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 > ;
// ing
using t2 = std::variant_alternative_t<2, type > ;
```

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 型の値を構築
    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 は実引数に与えられた variant が保持できる型でなければならない。以下のようなコードはエラーとなる。

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

    // エラー
    std::holds_alternative<double>(v) ;
}
```

get<I>(v) : インデックスから値の取得

get<I>(v) は、variant v の型のインデックスから I 番目の型の値を返す。インデックスは 0 ベースだ。

```
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 がインデックスの範囲を超えているとエラーとなる。

```
int main()
{
    // インデックスは 0, 1, 2まで
    std::variant< int, double, std::string > v ;

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

もし、variant が値を保持していない場合、つまり v.index() != I の場合は、std::bad_variant_access が throw される。

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

    try {
        // double 型の値を要求
        auto d = std::get<1>(v) ;
    } catch ( std::bad_variant_access & e )
    {
        // double は保持していなかった
    }
}
```

get の実引数に渡す variant が lvalue の場合は、戻り値は lvalue リファレンス、rvalue の場合は戻り値は rvalue リファレンスになる。

```
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 の実引数に渡す variant が CV 修飾されている場合、戻り値の型も実引数と同じく CV

修飾される。

```
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 ) ;
}
```

get<T>(v) : 型から値の取得

get<T>(v) は、variant v の保有する型 T の値を返す。型 T の値を保持していない場合、std::bad_variant_access が throw される。

```
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 ) ;
}
```

その他はすべて get<I>と同じ。

get_if : 値を保持している場合に取得

get_if<I>(vp) と get_if<T>(vp) は、variant へのポインター vp を実引数にとり、*vp がインデックス I、もしくは型 T の値を保持している場合、その値へのポインターを返す。

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

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

    v = 3.14 ;
    // double *
    auto b = std::get_if<1>( &v ) ;

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

}
```

もし、vp が nullptr の場合、もしくは*vp が指定された値を保持していない場合は、nullptr を返す。

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

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

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

variant の比較

variant は比較演算子がオーバーロードされているため比較できる。variant 同士の比較は、一般のプログラマーは自然だと思う結果になるように実装されている。

■同一性の比較　variant の同一性の比較のためには、variant のテンプレート実引数に与える型は自分自身と比較可能でなければならない。

つまり、variant v, w に対して、式 get<i>(v) == get<i>(w) がすべての i に対して妥当でなければならない。

variant v, w の同一性の比較は、v == w の場合、以下のように行われる。

1. v.index() != v.index ならば、false
2. それ以外の場合、v.value_less_by_exception() ならば、true
3. それ以外の場合、get<i>(v) == get<i>(w)。ただし i は v.index()

二つの variant が別の型を保持している場合は等しくない。ともに値なしの状態であれば等しい。それ以外は保持している。

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

    // true
    // 同じ型の同じ値を保持している。
    a == b ;

    a = 1.0 ;

    // false
    // 型が違う
    a == b ;
}
```

例えば operator ==は以下のような実装になる。

```
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 !=はこの逆だと考えてよい。

■大小比較　variant の大小の比較のためには、variant のテンプレート実引数に与える型は自分自身と比較可能でなければならない。

つまり、operator < の場合、variant v, w に対して、式 get<i>(v) < get<i>(w) がすべての i に対して妥当でなければならない。

variant v, w の大小比較は、v < w の場合、以下のように行われる。

1. w.valueless_by_exception() ならば、false
2. それ以外の場合、v.valueless_by_exception() ならば、true
3. それ以外の場合、v.index() < w.index() ならば、true
4. それ以外の場合、v.index > w.index() ならば、false
5. それ以外の場合、get<i>(v) < get<i>(w)。ただし i は v.index()

値なしの variant は最も小さいとみなされる。インデックスの小さいほうが小さいとみなされる。どちらも同じ型の値があるのであれば、値同士の比較となる。

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

    // false
    // 同じ型の同じ値を比較
    a < b ;

    a = 1.0 ;

    // false
    // インデックスによる比較
    a < b ;
    // true
```

```
    // インデックスによる比較
    b < a ;
}
```

operator <は以下のような実装になる。

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

残りの大小比較も同じ方法で比較される。

visit：variant が保持している値を受け取る。

std::visit は、variant の保持している型を実引数に関数オブジェクトを呼んでくれるライブラリだ。

```
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 ) が呼ばれる
    std::visit( visitor, v, w ) ;
    // visitor( 3.14, 42 ) が呼ばれる
    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 ) ;
}
```

any : どんな型の値でも保持できるクラス

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>
```

```
    // 保持している std::vector<int>のコピー
    auto value = std::any_cast< std::vector<int> >( a ) ;
}
```

any が保持できない型は、コピー構築できない型だ。

any の構築と破棄

クラス any はテンプレートではない。そのため宣言は単純だ。

```
int main()
{
    // 値を保持しない
    std::any a ;
    // int 型の値を保持する
    std::any b( 0 ) ;
    // double 型の値を保持する
    std::any c( 0.0 ) ;
}
```

any が保持する型を事前に指定する必要はない。

クラス any を破棄すると、その時保持していた値が適切に破棄される。

in_place_type コンストラクター

any のコンストラクターで emplace をするために in_place_type が使える。

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

int main()
{
    // 型 X を X(1, 2) で構築した結果の値を保持する
    std::any a( std::in_place_type<X>, 1, 2 ) ;
}
```

any への代入

any への代入も普通のプログラマーの期待通りの動きをする。

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

    // a は int 型の値 42 を保持する。
    a = 42 ;
    // b は int 型の値 42 を保持する
    b = a ;

}
```

any のメンバー関数

■emplace

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

any は emplace メンバー関数をサポートしている。

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

int main()
{
    std::any a ;

    // 型 X を X(1, 2) で構築した結果の値を保持する
    a.emplace( 1, 2 ) ;
}
```

■reset： 値の破棄

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

any の reset メンバー関数は、any の保持してある値を破棄する。reset を呼び出した後の any は値を保持しない。

```
int main()
{
    // a は値を保持しない
    std::any a ;
    // a は int 型の値を保持する
    a = 0 ;

    // a は値を保持しない
    a.reset() ;
}
```

■swap：スワップ　any は swap メンバー関数をサポートしている。

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

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

    a.swap(b) ;

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

■has_value：値を保持しているかどうか調べる

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

any の has_value メンバー関数は any が値を保持しているかどうかを調べる。値を保持しているならば true を、保持していないならば false を返す。

```
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 : 保持している型の type_info を得る

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

type メンバー関数は、保持している型 T の typeid(T) を返す。値を保持していない場合、typeid(void) を返す。

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

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

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

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

any のフリー関数

■make_any<T>：T 型の any を作る

```
emplate <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...) は T 型をコンストラクター実引数 args...で構築した値を保持する any を返す。

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

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

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

■any_cast：保持している値の取り出し

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

any_cast<T>(operand) は operand が保持している値を返す。

```
int main()
{
```

```
    std::any a(0) ;

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

any_cast<T>で指定した T 型が、any が保持している型ではない場合、std::bad_any_cast が throw される。

```
int main()
{
    try {
        std::any a ;
        std::any_cast<int>(a) ;
    } catch( std::bad_any_cast e )
    {
        // 型を保持していなかった。
    }

}
```

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

any_cast<T>に any へのポインターを渡すと、T へのポインター型が返される。any が T 型を保持している場合は T 型を参照するポインターが返る。保持していない場合は、nullptr が返る。

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

    // int 型の値を参照するポインター
    int * p1 = std::any_cast<int>( &a ) ;

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

optional：値を保有しているか、していないクラス

使い方

ヘッダーファイル<optional>で定義されている optional<T>は、T 型の値を保有しているか、保有していないライブラリだ。

条件次第で値が用意できない場合が存在する。例えば割り算の結果の値を返す関数を考える。

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

ゼロで除算はできないので、b の値が 0 の場合、この関数は値を用意することができない。問題は、int 型のすべての値は通常の除算結果として使われるので、エラーであることを示す特別な値を返すこともできない。

このような場合にエラーや値を通知する方法として、過去に様々な方法が考案された。例えば、ポインターやリファレンスを実引数として受け取る方法、グローバル変数を使う方法、例外だ。

optional はこのような値が用意できない場合に使える共通の方法を提供する。

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

int main()
{
    auto result = divide( 10, 2 ) ;
```

```
    // 値の取得
    auto value = result.value() ;

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

    // false、値を保持していない
    bool has_value = fail.has_value() ;

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

optional のテンプレート実引数

optional<T>は T 型の値を保持するか、もしくは保持しない状態を取る。

```
int main()
{
    // int 型の値を保持するかしない optional
    using a = std::optional<int> ;
    // double 型の値を保持するかしない ottional
    using b = std::optional<double> ;
}
```

optional の構築

optional をデフォルト構築すると、値を保持しない optional になる。

```
int main()
{
    // 値を保持しない
    std::optional<int> a ;
}
```

コンストラクターの実引数に std::nullopt を渡すと、値を保持しない optional になる。

```
int main()
{
    // 値を保持しない
```

```
    std::optional<int> a( std::nullopt ) ;
}
```

optional<T>のコンストラクターの実引数に T 型に変換できる型を渡すと、T 型の値に型変換して保持する。

```
int main()
{
    // int 型の値 42 を保持する
    std::optional<int> a(42) ;

    // double 型の値 1.0 を保持する
    std::optional<double> b( 1.0 ) ;

    // int から double への型変換が行われる
    // int 型の値 1 を保持する
    std::optional<int> c ( 1.0 ) ;
}
```

T 型から U 型に型変換できるとき、optional<T>のコンストラクターに optional<U>を渡すと U から T に型変換されて T 型の値を保持する optional になる。

```
int main()
{
    // int 型の値 42 を保持する
    std::optional<int> a( 42 ) ;

    // long 型の値 42 を保持する
    std::optional<long> b ( a ) ;
}
```

optional のコンストラクターの第一引数に std::in_place_type<T>を渡すと、後続の引数を使って T 型のオブジェクトが emplace 構築される。

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

int main()
{
```

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

optional の代入

通常のプログラマーの期待通りの挙動をする。std::nullopt を代入すると値を保持しない optional になる。

optional の破棄

optional が破棄されるとき、保持している値があれば、適切に破棄される。

```
struct X
{
    ~X() { }
} ;

int main()
{
    {
        // 値を保持する
        std::optional<X> o ( X{} ) ;
        // X のデストラクターが呼ばれる。
    }

    {
        // 値を保持しない
        std::optional<X> o ;
        // X のデストラクターは呼ばれない。
    }
}
```

swap

optional は swap に対応している。

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

```
    a.swap(b) ;
}
```

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 メンバー関数は optional が値を保持している場合、true を返す。

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

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

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 メンバー関数は optional が値を保持している場合、値へのリファレンスを返す。値を保持していない場合、std::bad_optional_access が throw される。

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

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

    try {
        std::optional<int> b ;
```

```
        int y = b.get() ;
    } catch( std::bad_optional_access e )
    {
        // 値を保持していなかった
    }
}
```

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) メンバー関数は、optional が値を保持している場合はその値を、保持していない場合は v を返す。

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

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

    std::optional<int> b ;

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

reset：保持している値を破棄する

reset メンバー関数を呼び出すと、保持している値がある場合破棄する。reset メンバー関数を呼び出した後の optional は値を保持しない状態になる。

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

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

```
    a.reset() ;

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

optional 同士の比較

optional<T>を比較するためには、T 型のオブジェクト同士が比較できる必要がある。

■同一性の比較　値を保持しない二つの optional は等しい。片方のみが値を保持している optional は等しくない。両方とも値を保持している optional は値による比較になる。

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

    // true
    // どちらも値を保持しない optional
    bool b1 a == b ;

    a = 0 ;

    // false
    // a のみ値を保持
    bool b2 = a == b ;

    b = 1 ;

    // false
    // どちらも値を保持。値による比較
    bool b3 = a == b ;
}
```

■大小比較　optional 同士の大小比較は、a < b の場合

1. b が値を保持していなければ false
2. それ以外の場合で、a が値を保持していなければ true
3. それ以外の場合、a と b の保持している値同士の比較

となる。

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

    // false
    // b が値なし
    bool b1 = a < b ;

    b = 0 ;

    // true
    // b は値ありで a が値なし
    bool b2 = a < b ;

    a = 1 ;

    // false
    // どちらとも値があるので値同士の比較
    // 1 < 0 は false
    bool b3 = a < b ;
}
```

optional と std::nullpot との比較

optional と std::nullopt との比較は、std::nullopt が値を持っていない optional として扱われる。

optional<T>と T の比較

optional<T>と T 型の比較をする場合、optional は値を保持していなければならない。

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) は optional<T>(t) を返す。

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

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

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 ) ;
}
```

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。

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様々ある。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 は文字数
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様々ある。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 ) ;
// 配列へのポインターと文字数
void process_string( char * ptr, std::size_t size ) ;
// std::string クラス
void process_string( std::string s ) ;
// 自作の string_type クラス
void process_string( string_type s ) ;
// 自作の my_string_type クラス
void process_string( my_string_type s ) ;
```

string_view は様々な表現の文字列に対して共通の view を提供することで、この問題を解決できる。もう関数のオーバーロードを大量に追加する必要はない。

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

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

```
}

// これひとつだけでよい。
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 ) ;
}
```

basic_string_view

std::string が std::basic_string< CharT, Traits >に対する std::basic_string であるように、std::string_view も、その実態は std::basic_string_view の特殊化への typedef 名だ。

```
// 本体
template<class charT, class traits = char_traits<charT>>
class basic_string_view ;

// それぞれの文字型の 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>;
```

なので、通常は basic_string_view ではなく、string_view とか u16string_view などの typedef 名を使うことになる。本書では string_view だけを解説するが、その他の typedef 名も文字型が違うだけで同じだ。

文字列の所有、非所有

string_view は文字列を所有しない。所有というのは、文字列を表現するストレージの確保と破棄に責任を持つということだ。所有しないことの意味を説明するために、まず文字

列を所有するライブラリについて説明する。

std::string は文字列を所有する。std::string 風のクラスの実装は、例えば以下のようになる。

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

public :
    // 文字列を表現するストレージの動的確保
    string ( char const * str )
    {
        size = std::strlen( str ) ;
        ptr = new char[size+1] ;
        std::strcpy( ptr, str ) ;
    }

    // コピー
    // 別のストレージを動的確保
    string ( string const & r )
        : size( r.size ), ptr ( new char[size+1] )
    {
        std::strcpy( ptr, r.ptr ) ;
    }

    // ムーブ
    // 所有権の移動
    string ( string && r )
        : size( r.size ), ptr( r.ptr )
    {
        r.size = 0 ;
        r.ptr = nullptr ;
    }

    // 破棄
    // 動的確保したストレージを解放
    ~string()
    {
        delete[] ptr ;
    }
```

```
} ;
```

std::string は文字列を表現するストレージを動的に確保し、所有する。コピーは別のストレージを確保する。ムーブするときはストレージの所有権を移す。デストラクターは所有しているストレージを破棄する。

std::string_view は文字列を所有しない。std::string_view 風のクラスの実装は、例えば以下のようになる。

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

public :

    // 所有しない
    // str の参照先の寿命は呼び出し側が責任を持つ
    string_view( char const * str ) noexcept
        : size( std::strlen(str) ), ptr( str )
    { }

    // コピー
    // メンバーごとのコピーだけでよいので default 化するだけでよい
    string_view( string_view const & r ) noexcept = default ;

    // ムーブはコピーと同じ
    // 所有しないので所有権の移動もない

    // 破棄
    // 何も開放するストレージはない
    // デストラクターもトリビアルでよい
} ;
```

string_view に渡した連続した文字型の配列へのポインターの寿命は、渡した側が責任を持つ。つまり、以下のようなコードは間違っている。

```
std::string_view get_string()
{
    char str[] = "hello" ;

    // エラー
```

```
    // str の寿命は関数の呼び出し元に戻った時点で尽きている
    return str ;
}
```

string_view の構築

string_view の構築には 4 種類ある。

- デフォルト構築
- null 終端された文字型の配列へのポインター
- 文字方の配列へのポインターと文字数
- 文字列クラスからの変換関数

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()
{
    // 空の string_view
    std::string_view s ;
}
```

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" ) ;
}
```

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 ) ;
}
```

文字列クラスからの変換関数

他の文字列クラスから string_view を作るには、変換関数を使う。string_view のコンストラクターは使わない。

std::string は string_view への変換関数をサポートしている。独自の文字列クラスを string_view に対応させるにも変換関数を使う。例えば以下のように実装する。

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

これにより、std::string から string_view への変換が可能になる。

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

```
    std::string_view sv = s ;
}
```

コレと同じ方法を使えば、独自の文字列クラスも string_view に対応させることができる。

std::string は string_view を受け取るコンストラクターを持っているので、string_view から string への変換もできる。

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

    // コピーされる
    std::string s = sv ;
}
```

string_view の操作

string_view は既存の標準ライブラリの string とほぼ同じ操作性を提供している。例えばイテレーターを取ることができるし、operator [] で要素にアクセスできるし、size() で要素数が返るし、find() で検索もできる。

```
template < typename T >
void f( T  t )
{
    for ( auto c : t )
    {
        std::cout << c ;
    }

    if ( t.size() > 3 )
    {
        auto c = t[3] ;
    }

    auto pos = t.find( "fox" ) ;
}

int main()
```

```
{
    std::string s("quick brown fox jumps over the lazy dog.") ;

    f( s ) ;

    std::string_view sv = s ;

    f( sv ) ;
}
```

string_view は文字列を所有しないので、文字列を書き換える方法を提供していない。

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

    s[0] = 'H' ;
    s += ",world" ;

    std::string_view sv = s ;

    // エラー
    // string_view は書き換えられない
    sv[0] = 'h' ;
    s += ".\n" ;
}
```

string_view は文字列を所有せず、ただ参照しているだけだからだ。

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

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

    s = "world" ;

    // "world"
    // string_view は参照しているだけ
    std::cout << sv ;
```

```
}
```

string_vie は string とほぼ互換性のあるメンバーを持っているが、一部の文字列を変更するメンバーは削除されている。

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

string_view は先頭と末尾から n 個の要素を削除するメンバー関数を提供している。

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

string_view にとって、先頭と末尾から n 個の要素を削除するのは、ポインターを n 個ずらすだけなので、これは文字列を所有しない string_view でも行える操作だ。

```
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。

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 ;
}
```

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。その特徴は以下の通り。

- 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
- 標準で提供される様々な特性を持ったメモリーリソースの実装

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 バイトのストレージを確保
    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 の宣言は以下の通り。

```
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なかにある。

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 ) ;
}
```

二つの 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_resouce * b )
{
    void * ptr = a->allocate( 1 ) ;

    // a で確保したストレージは b で解放できるか？
    if ( a->is_equal( *b ) )
    {// できる
        b->deallocate( ptr, 1 ) ;
    }
    else
    {// できない
        a->deallocate( ptr, 1 ) ;
    }
}
```

is_equal を呼び出す operator ==と operator !=も提供されている。

```
void f( std::pmr::memory_resource * a, std::pmr::memory_resource * b )
{
    bool b1 = ( *a == *b ) ;
    bool b2 = ( *a != *b ) ;
}
```

メモリーリソースの作り方

独自のメモリーアロケーターを memory_resouce のインターフェースに合わせて作るには、memory_resource から派生した上で、do_allocate, do_deallocate, do_is_equal の 3 つの private 純粋 virtual メンバー関数をオーバーライドする。必要に応じてデストラクターもオーバーライドする。

```
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) は少なくとも alignment バイトでアライメントされた bytes バイトのストレージへのポインターを返す。ストレージが確保できなかった場合は、適切な例外を throw する。

do_deallocate(p, bytes, alignment) は事前に同じ*this から呼び出された allocate(bytes, alignment) で返されたポインター p を解放する。すでに解放されたポインター p を渡してはならない。例外は投げない。

do_is_equal(other) は、*this と other が互いに一方で確保したストレージをもう一方で解放できる場合に true を返す。

たとえば、malloc/free を使った memory_resouce の実装は以下の通り。

```
// malloc/free を使ったメモリーリソース
class malloc_resource : public std::pmr::memory_resource
{
public :
    //
    ~malloc_resource() { }
private :
    // ストレージの確保
    // 失敗した場合 std::bad_alloc を throw する
    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 ;
    }

    // ストレージの解放
    virtual void do_deallocate( void * p, std::size_t bytes, std::size_t alignment ) overr
    {
```

```
        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。

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。例えば独自の 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とり、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。

例えば、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

にかける。

```
int main()
{
    bool b;

    std::cin >> b ;

    std::pmr::mempry_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_defa
        mem = mono.get() ;
    }

    std::vector< int, std::pmr::polymorphic_allocator<int> > v( std::pmr::polymorphic_allo
}
```

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。

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 () ) と同じ
    std::pmr::polymorphic_allocator<int> p1 ;

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

後は通常のあロケーターと同じように振る舞う。

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。

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() ;
}
```

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。

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 ;
}
```

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。例えば 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 のメモリ管理は、このように柔軟にはできていない。例えば、あるアーキテクチャーと OS では、メモリ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 4KB であったりする。もしシステムの低級なメモリ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を使うのに 3KB 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。アーキテクチャによってはメモリ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、低級なメモリ管理を隠匿し、小さなサイズのメモリ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を確保し、その中に管理用のデータ構造を作り、メモリ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()
    {
        // 大きな連続したストレージを確保
    }

    virtual void * do_allocate( std::size_t bytes, std::size_t alignment ) override
    {
        std::scoped_lock lock( m ) ;
        // リンクリストをたどり、十分な大きさの未使用領域を探し、リンクリスト構造体を構築して返す
        // アライメント要求に注意
    }

    virtual void * do_allocate( std::size_t bytes, std::size_t alignment ) override
    {
        std::scoped_lock lock( m ) ;
        // リンクリストから該当する部分を削除
    }

    virtual bool do_is_equal( const memory_resource & other ) const noexcept override
    {
    // *this と other で相互にストレージを解放できるかどうか返す
    }
} ;
```

プールリソース

プールリソースは C++17 の標準ライブラリが提供しているメモリーリソースの実装だ。synchronized_pool_resourcce と unsynchronized_pool_resource の二つがある。

アルゴリズム

プールリソースは以下のような特徴を持つ。

- 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 ) ;
    }
} ;

}
```

- プールリソースは構築時に pool_options を渡すことにより、最大ブロックサイズと最大チャンクサイズを設定できる。
- マルチスレッドから呼び出しても安全な同期を取る synchronized_pool_resource と、同期をとらない unsynchronized_pool_resource がある。

synchronized/unsynchronized_pool_resource

プールリソースには、synchronized_pool_resource と unsynchronized_pool_resource がある。どちらもクラス名以外は同じように使える。ただし、synchronized_pool_resource は複数のスレッドから同時に実行しても使えるように内部で同期が取られているのに対し、unsynchronized_pool_resource は同期を行わない。unsyncrhonized_pool_resource は複数のスレッドから同時に呼び出すことはできない。

```
// 実装イメージ

namespace std::pmr {

class synchronized_pool_resource : public memory_resource
{
    std::mutex m ;

    virtual void * do_allocate( size_t size, size_t alignment ) override
    {
        // 同期する
        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
    {
        // 同期しない
        return do_allocate_impl( size, alignment ) ;
    }
} ;

}
```

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。

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通り。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。

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全てを解放する。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。

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、この名前がついている。

例えばゲームで 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辿り、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辿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 はなにもしない。デストラクターはストレージ全体を解放する。

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_resourc

    // 初期バッファーから確保
    mem.allocate( 1 ) ;
    // 上流メモリーリソースからストレージを確保して切り出して確保
    mem.allocate( 100 ) ;
    // 前回のストレージ確保で空きがあればそこから
```

```
    // なければ新たに上流から確保して切り出す。
    mem.allocate( 100 ) ;
}
```

- 一つのスレッドから使うことを前提に設計されている。allocate と deallocate は同期しない。
- メモリーリソースが破棄されると確保されたすべてのストレージも解放される。明示的に deallocate を呼ばなくてもよい。

コンストラクター

モノトニックバッファーリソースには以下のコンストラクターがある。

```
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()) {}
```

初期バッファーを取らないコンストラクターは以下の通り。

```
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 ひと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。

初期バッファーをとるコンストラクターは以下の通り。

```
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。

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;
```

メンバー関数 uptream_resource は、構築時に与えられた上流メモリーリソースへのポインターを返す。

並列アルゴリズム

並列アルゴリズムは C++17 で追加された新しいライブラリだ。このライブラリは既存の <algorithm>に、並列実行版を追加する。

並列実行について

C++11 では、スレッドと同期処理が追加され、複数の実行媒体が同時に実行されるという概念が C++ 標準規格に入った。

C++17 では、既存のアルゴリズムに、並列実行版が追加された。

例えば、all_of(first, last, pred) というアルゴリズムは、[first,last) の区間が空であるか、すべてのイテレーター i に対して pred(*i) が true を返すとき、true を返す。それ以外の場合は false を返す。

すべての値が 100 未満であるかどうかを調べるには、以下のように書く。

```
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::ba

    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二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 は二つの物理コア、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、より高性能で更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。

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);
```

並列アルゴリズム版の all_of は以下のような宣言になる。

```
template <class ExecutionPolicy, class ForwardIterator, class Predicate>
bool any_of(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last, Predicate
```

並列アルゴリズムには、テンプレート仮引数として ExecutionPolicy が追加されていて第一引数に取る。これを実行時ポリシーと呼ぶ。

実行時ポリシーは<execution>で定義されている関数ディスパッチ用のタグ型で、std::execution::seq, std::execution::par, std::execution::par_unseq がある。

複数のスレッドによる並列実行を行うには、std::execution::par を使う。

```
template < typename Container >
bool is_all_of_less_than_100( Container const & input )
```

```
{
    return std::all_of( std::execution::par,
        std::begin(input), std::end(input),
        []( auto x ){ return x < 100 ; } ) ;
}
```

std::execution::seq を渡すと既存のアルゴリズムと同じシーケンシャル実行になる。std::execution::par を渡すとパラレル実行になる。std::execution::par_unseq は並列実行かつベクトル実行になる。

C++17 には実行ポリシーを受け取るアルゴリズムのオーバーロード関数が追加されている。

並列アルゴリズム詳細

並列アルゴリズム

並列アルゴリズム (parallel algorithm) とは、ExecutionPolicy(実行ポリシー) というテンプレートパラメーターのある関数テンプレートのことだ。既存の<algorithm>と C++14 で追加された一部の関数テンプレートが、並列アルゴリズムに対応している。

並列アルゴリズムはイテレーター、仕様上定められた操作、ユーザーの提供する関数オブジェクトによる操作、仕様上定められた関数オブジェクトに対する操作によって、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。

例えば、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、並列実行にともなう様々な制約を満たさなければならない。

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 ; } ) ;
    // エラー
}
```

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 )
                // 最後の要素なので特別な処理
                // エラー
        } ) ;
}
```

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。

```
// 実装イメージ

template < typename ExecutionPolicy, typename ForwardIterator, typename Predicate >
bool all_of( ExecutionPolicy && exec, ForwardIterator first, ForwardIterator last, Predicate p
{
    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。

例えば、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 ;
        } ) ;
    // エラー
}
```

このコードは parallel_unsequenced_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体。

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。

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 が 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。

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。