

C++11の文法と機能(C++11: Syntax and Feature)

Copyright (C) 2013 江添亮.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

筆者について

名前:江添亮

Mail : boostcpp@gmail.com

Blog : <http://cpplover.blogspot.jp/>

GitHub: <https://github.com/EzoeRyou/cpp-book>

序

本書はC++11のコア言語の文法と機能を、標準規格書に従って解説したものである。正式なC++規格書として発行された後の、ひとつ後のドラフト規格、N3337を参考している。ドラフト規格を参考にした理由は、正式なC++規格書は、個人での入手が煩わしいためである。読者に入手が困難な資料を元に記述された参考書は価値がない。そのため、読者が容易に入手できるドラフト規格のうち、正式なC++規格書とほとんどかわらないN3337を参考にした。

本書の対象読者は、C++を記述するものである。C++実装者ではない。そのため、サンプルコードを増やし、冗長な解説を増やし、C++コンパイラーを実装するための詳細な定義は省いた。そもそも本書はC++規格の翻訳ではなく、C++規格の検証に使うことはできない。

- 1 概要(General)
 - 1.1 C++とは
 - 1.2 用語(Definitions)
 - 1.2.1 実引数(argument)と、仮引数(parameter)
 - 1.2.2 静的な型(static type)と、動的な型(dynamic type)
 - 1.2.3 シグネチャー(signature)
 - 1.2.4 ill-formedプログラムと、well-formedプログラム
 - 1.2.5 実装可能な機能(conditionally-supported)
 - 1.2.6 実装定義の動作(implementation-defined behavior)
 - 1.2.7 未定義の動作(undefined behavior)
 - 1.2.8 未規定の動作(unspecified behavior)
 - 1.3 文法記法(Syntax notation)
 - 1.4 C++メモリーモデル(The C++ memory model)
 - 1.5 C++オブジェクトモデル(The C++ object model)
 - 1.6 プログラムの実行(Program execution)
- 2 字句規約(Lexical conventions)
 - 2.1 翻訳単位(Separate translation)
 - 2.2 ソースファイルの変換(Phase of translation)
 - 2.3 文字セット(Character sets)
 - 2.3.1 基本ソース文字セット
 - 2.3.2 ユニバーサル文字名
 - 2.4 トーケン(Tokens)
 - 2.5 コメント(Comments)

- 2.6 識別子(Identifiers)
 - 2.6.1 予約語
- 2.7 キーワード(Keywords)
- 2.8 リテラル(Literals)
 - 2.8.1 整数リテラル(Integer literals)
 - 2.8.2 浮動小数点数リテラル(Floating literals)
 - 2.8.3 文字リテラル(Character literals)
 - 2.8.3.1 エスケープシーケンス
 - 2.8.4 文字列リテラル(String literals)
 - 2.8.4.1 エンコード方式
 - 2.8.4.2 文字列リテラルの型の要素数
 - 2.8.4.3 生文字列リテラル(Raw String Literal)
 - 2.8.5 boolリテラル(Boolean literals)
 - 2.8.6 ポインタリテラル(Pointer Literals)
 - 2.8.7 ユーザー定義リテラル(User-defined literals)
- 3 基本事項(Basic concepts)
 - 3.1 宣言と定義(Declarations and definitions)
 - 3.1.1 宣言(Declaration)と定義(Definition)の違い
 - 3.1.2 定義ではない宣言
 - 3.2 ODR(One definition rule)
 - 3.3 スコープ(Scope)
 - 3.3.1 宣言領域とスコープ(Declarative regions and scopes)
 - 3.3.2 宣言場所(Point of declaration)
 - 3.3.3 ブロックスコープ(Block scope)
 - 3.3.4 関数プロトタイプのスコープ(Function prototype scope)
 - 3.3.5 関数のスコープ(Function scope)
 - 3.3.6 名前空間のスコープ Namespace scope
 - 3.3.6.1 グローバル名前空間(Global namespace)
 - 3.3.7 クラスのスコープ(Class scope)
 - 3.3.8 enumのスコープ(Enumeration scope)
 - 3.3.9 テンプレート仮引数のスコープ(Template Parameter Scope)
 - 3.3.10 名前隠し(Name hiding)
 - 3.4 名前探索(Name lookup)
 - 3.4.1 Qualified 名前探索(Qualified name lookup)
 - 3.4.2 Unqualified 名前探索(Unqualified name lookup)
 - 3.4.3 ADL(Argument-dependent name lookup)
 - 3.4.3.1 関連クラスと関連名前空間
 - 3.4.3.2 ADLが適用される条件
 - 3.5 プログラムとリンクエージ(Program and linkage)
 - 3.6 プログラムの開始と終了(Start and termination)
 - 3.6.1 main関数(Main function)
 - 3.6.2 非ローカル変数の初期化(Initialization of non-local objects)
 - 3.6.3 終了(Termination)
 - 3.7 ストレージ(storage duration)
 - 3.7.1 確保関数(allocation function)
 - 3.7.2 解放関数(deallocation function)
 - 3.7.3 安全なポインター(Safely-derived pointers)
 - 3.7.4 サブオブジェクトの有効期間
 - 3.8 オブジェクトの寿命(Object lifetime)
 - 3.9 型(Types)
 - 3.9.1 基本型(Fundamental types)
 - 3.9.2 複合型(Compound types)
 - 3.9.3 CV修飾子(CV-qualifiers)
 - 3.9.4 lvalueとrvalue(Lvalues and rvalues)

- 3.9.5 アライメント(Alignment)
- 4 標準型変換(Standard conversions)
 - 4.1 lvalueからrvalueへの型変換(Lvalue-to-rvalue conversion)
 - 4.2 配列からポインターへの型変換(Array-to-pointer conversion)
 - 4.3 関数からポインターへの型変換(Function-to-pointer conversion)
 - 4.4 CV修飾子の型変換(Qualification conversions)
 - 4.5 整数の変換順位(Integer conversion rank)
 - 4.6 整数のプロモーション(Integral promotions)
 - 4.7 整数の型変換(Integral conversions)
 - 4.8 浮動小数点数のプロモーション(Floating point promotion)
 - 4.9 浮動小数点数の型変換(Floating point conversions)
 - 4.10 浮動小数点数と整数の間の型変換(Floating-integral conversions)
 - 4.11 ポインターの型変換(Pointer conversions)
 - 4.12 メンバーへのポインターの型変換(Pointer to member conversions)
 - 4.13 boolの型変換(Boolean conversions)
- 5 式(Expressions)
 - 5.1 優先順位と評価順序
 - 5.2 未評価オペランド(unevaluated operand)
 - 5.3 一次式(Primary expressions)
 - 5.3.1 ::演算子
 - 5.3.2 括弧式(parenthesized expression)
 - 5.4 ラムダ式(Lambda expressions)
 - 5.4.1 ラムダ式の基本的な使い方
 - 5.4.1.1 変数のキャプチャー
 - 5.4.2 ラムダ式の詳細
 - 5.4.2.1 クロージャーオブジェクト(closure object)
 - 5.4.3 thisのキャプチャー
 - 5.4.4 パラメータパックのキャプチャー
 - 5.5 後置式(Postfix expressions)
 - 5.5.1 添字(Subscripting)
 - 5.5.2 関数呼び出し(Function call)
 - 5.5.3 関数形式の明示的型変換(Explicit type conversion (functional notation))
 - 5.5.4 疑似デストラクター呼び出し(Pseudo destructor call)
 - 5.5.5 クラスメンバーアクセス(Class member access)
 - 5.5.6 インクリメントとデクリメント(Increment and decrement)
 - 5.5.7 Dynamic cast(Dynamic cast)
 - 5.5.8 型識別(Type identification)
 - 5.5.9 Static cast(Static cast)
 - 5.5.10 Reinterpret cast
 - 5.5.11 Const cast
 - 5.6 単項式(Unary expressions)
 - 5.6.1 単項演算子(Unary operators)
 - 5.6.1.1 *演算子と&演算子
 - 5.6.1.2 単項演算子の+と-
 - 5.6.1.3 !演算子
 - 5.6.1.4 ~演算子
 - 5.6.2 インクリメントとデクリメント(Increment and decrement)
 - 5.6.3 sizeof(Sizeof)
 - 5.6.4 new
 - 5.6.5 delete
 - 5.6.6 alignof
 - 5.6.7 noexcept演算子(noexcept operator)
 - 5.7 キャスト形式による明示的型変換(Explicit type conversion (cast

- notation))
 - 5.8 メンバーへのポインター演算子(Pointer-to-member operators)
 - 5.9 乗除算の演算子(Multiplicative operators)
 - 5.10 加減算の演算子(Additive operators)
 - 5.11 シフト演算子(Shift operators)
 - 5.12 関係演算子(Relational operators)
 - 5.13 等価演算子(Equality operators)
 - 5.14 ビット列論理積演算子(Bitwise AND operator)
 - 5.15 ビット列排他的論理和演算子(Bitwise exclusive OR operator)
 - 5.16 ビット列論理和演算子(Bitwise inclusive OR operator)
 - 5.17 論理積演算子(Logical AND operator)
 - 5.18 論理和演算子(Logical OR operator)
 - 5.19 条件演算子(Conditional operator)
 - 5.20 代入と複合代入演算子(Assignment and compound assignment operators)
 - 5.21 コンマ演算子(Comma operator)
 - 5.22 定数式(Constant expressions)
- 6 文(Statements)
 - 6.1 ラベル文(Labeled statement)
 - 6.2 式文(Expression statement)
 - 6.3 複合文、ブロック(Compound statement or block)
 - 6.4 選択文(Selection statements)
 - 6.4.1 if文(The if statement)
 - 6.4.2 switch文(The switch statement)
 - 6.5 繰り返し文(Iteration statements)
 - 6.5.1 while文(The while statement)
 - 6.5.2 do文(The do statement)
 - 6.5.3 for文(The for statement)
 - 6.5.4 range-based for文(The range-based for statement)
 - 6.5.4.1 range-based forの基本
 - 6.5.4.2 range-based forの詳細
 - 6.5.5 ジャンプ文(Jump statements)
 - 6.5.5.1 break文(The break statement)
 - 6.5.5.2 continue文(The continue statement)
 - 6.5.5.3 return文(The return statement)
 - 6.5.5.4 goto文(The goto statement)
 - 6.6 宣言文(Declaration statement)
 - 6.7 暗昧解決(Ambiguity resolution)
 - 7 宣言(Declarations)
 - 7.1 単純宣言(simple-declaration)
 - 7.2 static_assert宣言(static_assert-declaration)
 - 7.3 指定子(Specifiers)
 - 7.3.1 ストレージクラス指定子(Storage class specifiers)
 - 7.3.1.1 register指定子
 - 7.3.1.2 thread_local指定子
 - 7.3.1.3 static指定子
 - 7.3.1.4 extern指定子
 - 7.3.1.5 mutable指定子
 - 7.3.2 関数指定子(Function specifiers)
 - 7.3.2.1 inline指定子
 - 7.3.2.2 virtual指定子
 - 7.3.2.3 explicit指定子
 - 7.3.3 typedef指定子(The typedef specifier)
 - 7.3.4 friend指定子(The friend specifier)

- 7.3.5 `constexpr`指定子(The `constexpr` specifier)
 - 7.3.6 型指定子(Type specifiers)
 - 7.3.6.1 CV修飾子(The cv-qualifiers)
 - 7.3.6.2 単純型指定子(Simple type specifiers)
 - 7.3.6.3 複雑型指定子(Elaborated type specifiers)
 - 7.3.6.4 `auto`指定子(`auto` specifier)
 - 7.3.6.5 `decltype`指定子(`decltype` specifier)
 - 7.4 `enum`の宣言(Enumeration declarations)
 - 7.4.1 unscoped enum
 - 7.4.2 scoped enum
 - 7.4.3 `enum`基底(enum-base)
 - 7.4.4 `enum`宣言(opaque-enum-declaration)
 - 7.5 名前空間(Namespaces)
 - 7.5.1 名前空間の定義 Namespace definition
 - 7.5.1.1 `inline`名前空間
 - 7.5.1.2 無名名前空間(Unnamed namespaces)
 - 7.5.1.3 名前空間のメンバーの定義 Namespace member definitions)
 - 7.5.2 名前空間エイリアス Namespace alias
 - 7.5.3 `using`宣言(The `using` declaration)
 - 7.5.4 `using`ディレクティブ(Using directive)
 - 7.6 リンケージ指定(Linkage specifications)
 - 7.7 アトリビュート(Attributes)
 - 7.7.1 アライメント指定子(alignment specifier)
 - 7.7.2 `noreturn`アトリビュート(Noreturn attribute)
- 8 宣言子(Declarators)
 - 8.1 型名(Type names)
 - 8.2 暫昧解決(Ambiguity resolution)
 - 8.3 宣言子の意味(Meaning of declarators)
 - 8.3.1 ポインター(Pointers)
 - 8.3.2 リファレンス(References)
 - 8.3.3 メンバーへのポインター(Pointers to members)
 - 8.3.4 配列(Arrays)
 - 8.3.5 関数(Functions)
 - 8.3.6 デフォルト実引数(Default arguments)
 - 8.4 関数の定義(Function definitions)
 - 8.4.1 `default`定義(Explicitly-defaulted functions)
 - 8.4.2 `delete`定義(Deleted definitions)
 - 8.5 初期化子(Initializers)
 - 8.5.1 ゼロ初期化(zero-initialize)
 - 8.5.2 デフォルト初期化(default-initialize)
 - 8.5.3 値初期化(value-initialize)
 - 8.5.4 アグリゲート(Aggregates)
 - 8.5.5 文字配列(Character arrays)
 - 8.5.6 リファレンス(References)
 - 8.5.7 リスト初期化(List-initialization)
 - 9 クラス(Classes)
 - 9.1 トリビアルにコピー可能なクラス(trivially copyable class)
 - 9.2 トリビアルクラス(trivial class)
 - 9.3 標準レイアウトクラス(standard-layout class)
 - 9.4 POD構造体(POD struct)
 - 9.5 クラス名(Class names)
 - 9.6 クラスのメンバー(Class members)
 - 9.6.1 レイアウト互換(layout-compatible)

- 9.7 メンバー関数(Member functions)
 - 9.7.1 非staticメンバー関数(Nonstatic member functions)
 - 9.7.2 thisポインター(The this pointer)
- 9.8 staticメンバー(Static members)
 - 9.8.1 staticメンバー関数(Static member functions)
 - 9.8.2 staticデータメンバー(Static data members)
- 9.9 union(Unions)
 - 9.9.1 無名union(anonymous union)
 - 9.9.2 共用メンバー(variant member)
- 9.10 ビットフィールド(Bit-fields)
- 9.11 クラス宣言のネスト(Nested class declarations)
- 9.12 ローカルクラス宣言(Local class declarations)
- 9.13 型名のネスト(Nested type names)
- 10 派生クラス(Derived classes)
 - 10.1 複数の基本クラス(Multiple base classes)
 - 10.2 メンバーの名前探索(Member name lookup)
 - 10.3 virtual関数(Virtual functions)
 - 10.4 アブストラクトクラス(Abstract classes)
- 11 メンバーのアクセス指定(Member access control)
 - 11.1 アクセス指定子(Access specifiers)
 - 11.2 基本クラスと、基本クラスのメンバーへのアクセス指定(Accessibility of base classes and base class members)
 - 11.3 friend(Friends)
 - 11.4 protectedメンバーアクセス(Protected member access)
 - 11.5 virtual関数へのアクセス(Access to virtual functions)
 - 11.6 複数のアクセス(Multiple access)
 - 11.7 ネストされたクラス(Nested classes)
- 12 特別なメンバー関数(Special member functions)
 - 12.1 コンストラクター(Constructors)
 - 12.2 一時オブジェクト(Temporary objects)
 - 12.3 型変換(Conversions)
 - 12.3.1 コンストラクターによる型変換(Conversion by constructor)
 - 12.3.2 型変換関数(Conversion functions)
 - 12.4 デストラクター(Destructors)
 - 12.5 フリーストア(Free store)
 - 12.6 初期化(Initialization)
 - 12.6.1 明示的な初期化(Explicit initialization)
 - 12.6.2 基本クラスとデータメンバーの初期化(Initializing bases and members)
 - 12.6.2.1 コンストラクターのデリゲート
 - 12.7 生成と破棄(Construction and destruction)
 - 12.8 クラスのコピーとムーブ
 - 12.9 コンストラクター継承(Inheriting constructors)
- 13 オーバーロード(Overloading)
 - 13.1 オーバーロード可能な宣言(Overloadable declarations)
 - 13.2 オーバーロードのその他の注意事項
 - 13.3 オーバーロード解決(Overload resolution)
 - 13.3.1 候補関数(Candidate functions)
 - 13.3.1.1 関数呼び出しの文法(Function call syntax)
 - 13.3.1.2 式中の演算子(Operators in expressions)
 - 13.3.1.3 コンストラクターによる初期化(Initialization by constructor)
 - 13.3.1.4 ユーザー定義型変換によるクラスのコピー初期化(Copy-initialization of class by user-defined conversion)

- 13.3.1.5 変換関数によるクラスではないオブジェクトの初期化(Initialization by conversion function)
- 13.3.1.6 変換関数によるリファレンスの初期化(Initialization by conversion function for direct reference binding)
- 13.3.1.7 リスト初期化による初期化(Initialization by list-initialization)
- 13.4 適切関数(Viable functions)
- 13.5 最適関数(Best viable function)
 - 13.5.1 暗黙の型変換の順序(Implicit conversion sequences)
 - 13.5.1.1 標準型変換(Standard conversion sequences)
- 13.6 オーバーロード関数のアドレス(Address of overloaded function)
- 13.7 オーバーロード演算子(Overloaded operators)
 - 13.7.1 単項演算子(Unary operators)
 - 13.7.2 二項演算子(Binary operators)
 - 13.7.3 代入(Assignment)
 - 13.7.4 関数呼び出し(Function call)
 - 13.7.5 添字(Subscripting)
 - 13.7.6 クラスメンバーアクセス(Class member access)
 - 13.7.7 インクリメントとデクリメント(Increment and decrement)
 - 13.7.8 確保関数と解放関数(allocation function and deallocation function)
 - 13.7.9 ユーザー定義リテラル
- 14 テンプレート(Templates)
 - 14.1 テンプレート仮引数/実引数(Template parameters/arguments)
 - 14.1.1 型テンプレート仮引数/実引数
 - 14.1.2 非型テンプレート仮引数/実引数
 - 14.1.3 テンプレートテンプレート仮引数/実引数
 - 14.1.4 デフォルトテンプレート実引数
 - 14.1.5 可変テンプレート仮引数
 - 14.2 テンプレート特殊化の名前(Names of template specializations)
 - 14.3 型の同一性(Type equivalence)
 - 14.4 テンプレート宣言(Template declarations)
 - 14.4.1 クラステンプレート(Class templates)
 - 14.4.2 メンバーテンプレート(Member Templates)
 - 14.4.3 可変引数テンプレート(Variadic Templates)
 - 14.4.3.1 可変引数テンプレートの使い方
 - 14.4.4 friend
 - 14.4.5 クラステンプレートの部分的特殊化(Class template partial specializations)
 - 14.4.5.1 クラステンプレートの部分的特殊化の一一致度の比較(Matching of class template partial specializations)
 - 14.4.5.2 クラステンプレートの部分的特殊化の半順序(Partial ordering of class template specializations)
 - 14.4.5.3 クラステンプレートの特殊化のメンバー
 - 14.4.6 関数テンプレート(Function templates)
 - 14.4.7 関数テンプレートのオーバーロード(Function Template Overloading)
 - 14.4.8 関数テンプレートの部分的特殊化(Partial ordering of function templates)
 - 14.4.9 エイリアステンプレート(Alias templates)
 - 14.5 名前解決(Name Resolution)
 - 14.5.1 依存(Dependent)
 - 14.5.2 非依存名の名前解決
 - 14.5.3 依存名の名前解決

- 14.6 テンプレートの実体化と特殊化(Template instantiation and specialization)
 - 14.6.1 暗黙の実体化(Implicit instantiation)
 - 14.6.2 明示的実体化(Explicit instantiation)
 - 14.6.3 明示的特殊化(Explicit specialization)
- 14.7 関数テンプレートの特殊化(Function template specializations)
 - 14.7.1 明示的なテンプレート実引数指定(explicit template argument specification)
 - 14.7.2 テンプレートの実引数推定(Template argument deduction)
 - 14.7.3 半順序(partial ordering)
- 15 例外(Exception handling)
 - 15.1 例外を投げる(Throwing an exception)
 - 15.2 コンストラクターとデストラクター(Constructors and destructors)
 - 15.3 例外の捕捉(Handling an exception)
 - 15.4 例外指定(Exception specifications)

1 概要(General)

1.1 C++とは

本書ではC++11のコア言語を解説する。

C++は、1998年に最初のC++の標準規格が正式に制定された。これをC++98と呼ぶ。C++の標準規格は、2003年にマイナーアップデートされた。これをC++03と呼ぶ。C++03では、文面の記述の誤りや矛盾点を修正し、僅かな新機能を追加した。そして、2011年に、始めてのメジャーアップデートとなる大幅な変更を含んだC++11規格が制定された。C++11では、多くのコア言語機能やライブラリが追加された。また、文面の記述も大幅に見なおされた。

将来に目を向けると、2014年には、再びマイナーアップデートであるC++14の正式発行が予定されている。次のメジャーアップデートは、非公式にC++1yと呼ばれているが、2017年を予定している。

1.2 用語(Definitions)

1.2.1 実引数(argument)と、仮引数(parameter)

引数には、ふたつの種類がある。実引数とは、関数やマクロ、throw文、テンプレートに、実際に渡す引数のことをいう。仮引数とは、関数宣言や関数の定義、例外ハンドラーのcatch句、マクロ、テンプレート仮引数のことをいう。たとえば、関数の場合、

```
// xは仮引数
void f( int x ) ;

int main()
```

```
{  
// 0は実引数  
    f(0) ;  
// argは実引数  
    int arg = 0 ;  
    f( arg ) ;  
}
```

このように、関数の宣言や定義などの引数を、仮引数といい、関数の呼び出しの際に指定する引数を、実引数という。仮引数と実引数は、厳密に区別される。

1.2.2 静的な型(static type)と、動的な型(dynamic type)

静的な型とは、実行しなくても、その意味が分かる型のことである。動的な型は、実行しなければ、その意味が決定出来ない型のことである。

1.2.3 シグネチャー(signature)

シグネチャーとは、ある関数に対する、その関数の名前、引数のリストの型、戻り値の型、テンプレート仮引数のことである。また、メンバー関数の場合は、そのクラスや、CV修飾、リファレンス修飾も含まれる。また、その関数の属する名前空間も含まれる。シグネチャーは、その関数を特定するために用いられる。

1.2.4 ill-formedプログラムと、well-formedプログラム

well-formedプログラムとは、文法上正しいプログラムである。ill-formedプログラムとは、well-formedではないプログラム、すなわち、文法上、間違ったプログラムである。多くの実装では、ill-formedプログラムは、コンパイルエラーとなる。

1.2.5 実装可能な機能(conditionally-supported)

実装可能な機能とは、規格上、実装してなくてもよい機能や動作のことである。

1.2.6 実装定義の動作(implementation-defined behavior)

実装定義の動作とは、well-formedではあるが、その意味が、実装によって変わることである。

1.2.7 未定義の動作(undefined behavior)

未定義の動作とは、そのプログラムの意味が、規格上定義されていないということである。その動作は実装によって異なり、あるいはエラーとなるかもしれないし、あるいは問題なく結果が予測できる動作となるかもしれない。一般に、エラーとなる場合が多い。未定義の動作を含むプログラムを書く場合は、そのコードの意味が、目的の実行環境で、明確に定義されているかどうかを確認するべきである。

1.2.8 未規定の動作(unspecified behavior)

未規定の動作も、具体的な意味が、実装によって異なると言う点で、未定義の動作と変わらない。ただし、未規定の動作は、規格上、推奨される動作が決められていることも多く、多くの実装で、エラーにならない、何らかの意味のあるコードになると言う点で、未定義の動作よりは、安全である。ただし、これも、目的の実行環境での意味がどうなるのかを、正しく把握しておく必要がある。

1.3 文法記法(Syntax notation)

本書では、言語の文法を記述するのに、規格に準じた文法記法を用いる。

個々の文法は、文法カテゴリー(syntactic category)として定義される。ある文法カテゴリーは、複数の文法カテゴリーや、具体的な文法を持つことができる。複数の文法カテゴリーは、改行で示される。

文法カテゴリー:

サブ文法カテゴリー category
category

サブ文法カテゴリー:

Syntactic

この例では、以下の二つの文法が定義されることになる。

Syntactic category
category

また、省略可能な文法カテゴリーは、「文法カテゴリー_{opt}」と表記される。

{ 式_{opt} }

この場合、式は書いても書かなくてもよい。

本書では、分かりやすさを重視するため、規格ほどの厳密な文法記法を用いることはない。文法カテゴリーは、常に日本語で表記される。

1.4 C++メモリーモデル(The C++ memory model)

C++では、メモリー領域のことを、ストレージ(storage)と呼ぶ。ストレージの最小単位は、バイトである。1バイトが何ビットであるかは定められていないが、少なくとも、8ビット以上であることが保証されている。最低8ビットである理由は、UTF-8の、ひとつのコード単位を格納できるようにするためである。メモリーは、連続したバイト列で表される。すべてのバイトは、ユニークなアドレスを持つ。

1.5 C++オブジェクトモデル(The C++ object model)

オブジェクトは、メモリー上に構築される。`int`や`float`といった基本型や、ユーザーが定義したクラスも、すべてオブジェクトとして構築される。ただし、関数は、オブジェクトではない。オブジェクトは、変数の定義や、`new`によって生成される。

```
// int型のオブジェクト
int x ;
int * pointer = new int ;

// Fooクラス型のオブジェクト
class Foo { } ;
Foo foo ;
Foo * foo_pointer = new Foo ;
```

オブジェクトは、型や、3.7 [ストレージの有効期間](#)、3.8 [生存期間](#)を持つ。

1.6 プログラムの実行(Program execution)

この項目の多くは、スレッドの競合と同期に関する話なので、省略する。

2 字句規約(Lexical conventions)

ここでは、C++のソースコードを表現する文字について解説する。

2.1 翻訳単位(Separate translation)

プログラムは、ソースファイルという単位に分割される。ソースファイルとは、プリプロセッサが実行された後のソースコードである。

2.2 ソースファイルの変換(Phase of translation)

ソースファイルは、コンパイルにかけられる前に、変換される。この変換の詳しい定義は略すが、特に知っておくべき変換が、いくつかある。

2.3.1 基本ソース文字セットではない文字は、2.3.2 UCNに変換される。

変換前

あ

変換後

\u3042

行末の¥(バックスラッシュ)と、それに続く改行は、取り除かれる。

変換前

```
int \
value \
= 0 ;
```

変換後

```
int value = 0 ;
```

この改行の除去のルールは、注意を要する。例えば、

変換前

```
//  
//  
これはコメント  
int va\  
lue = 0 ;
```

変換後

```
//これはコメント  
int value = 0 ;
```

連続する文字列リテラルのトークンは、連結される。

変換前

```
"aaa"  
"bbb" ;
```

変換後

```
"aaabbb" ;
```

2.3 文字セット(Character sets)

C++は、ソースファイルの文字コードを定めていない。ソースファイル内の文字は、環境依存の文字コードで表現される。

2.3.1 基本ソース文字セット

基本ソース文字セット(basic source character set)とは、ソースファイルで使うことができる文字のことである。印字可能な文字は、以下の通り。

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

上記に加えて、スペース、水平タブ、垂直タブ、フォームフィード、改行も使うことができる。

たとえば、ASCIIでいえば、@(アットマーク)や`(グレイヴ・アクセント)のような文字は、使われていない。

2.3.2 ユニバーサル文字名

ユニバーサル文字名(UCN:universal character name)とは、ISO/IEC 10646で定義されている文字コードのことである。この文字コードは、ユニバーサル文字セットと呼ばれている。また、厳密には同じではないが、俗に、Unicodeと呼ばれることもある。UnicodeはUnicodeで、別の規格があるのだが、実用上、どちらもコードポイントやエンコード方式に、差がないため、よく混同される。本書では、ユニバーサル文字セットや、UTFエンコードに関する解説は行わない。

Nを16進数の一文字とすると、¥uNNNNは、UCSにおけるコードポイント、0000NNNNで表される文字という意味になり、¥UNNNNNNNNは、NNNNNNNNで表される文字と同じ意味になる。

\u3042	あ
\u3043	い
\U00003044	う

このユニバーサル文字名は、リテラルやエスケープシーケンスではなく、もっと根本的に、文字と同等に扱われる。つまり、対応するユニバーサル文字として認識される。ユニバーサル文字名は、ソースコードのあらゆる場所で使うことができる。ソースコードのある場所に、ユニバーサル文字名を記述した場合、その場所に、対応するユニバーサル文字を記述したのと、全く同じ扱いがなされる。

ただし、2.8.4.3 生文字列リテラルの中では、ユニバーサル文字名は使えない。サロゲートの範囲のコードポイント(0xD800–0xDFFF)を指定することはできない。文字列リテラルの中では、ユニバーサル文字は実装により自動的にエンコードされるので、サロゲ

コードポイントを明示的に使う必要はない。もし文字としてサロゲートコードポイントを使う必要があるならば、エスケープシーケンス`\u`が使える。文字リテラルや文字列リテラルの中以外では、コントロール文字の範囲のコードポイント(0x00–0x1F と 0x7F–0x9F)と、基本ソース文字セットに該当するコードポイントを指定することはできない。

```
R"(\u3042)" ; // ユニバーサル文字名ではなく、文字通りに解釈されることに注意
u8"\u00E3\u0081\u0082" ; // OK、u8"あ"と同じ

u"\ud842\udfb7" ; // エラー、サロゲートコードポイントを明示的に使うことはできない
u"\U00020bb7" ; // OK、u"𠮷"と同じ

int \u0041 = 0 ; // エラー、ここで基本ソース文字セットは使えない
int \u3042 = 0 ; // OK
```

2.4 トークン(Tokens)

C++には、5種類のトークン(token)がある。識別子、キーワード、リテラル、演算子、その他の区切り文字である。

空白、水平タブ、垂直タブ、改行、フォームフィード、コメントは無視される。

ただし、空白文字は、トークンを区切るために文法上必要になることもある。

2.5 コメント(Comments)

コメントには、二種類ある。`/* */`で囲まれた、複数行にわたるコメントと、`//`から始まり行末までの、一行コメントである。

```
/* ここからコメント開始
これはコメント
これもコメント
ここでコメント終わり*/
// これは一行コメント。改行までがコメントとなる。
```

コメントは、`/*`で始まり、`*/`で終わる。この形式のコメントは、ネストできない。

```
/* これはコメント*/  
  
/*ここから  
  
ここまでコメント*/  
  
/* /* これはエラー。ネストしている。 */ */
```

//で始まるコメントは、行末までが、コメントとなる。これは、/**/形式のコメントの中にも、書くことができる。

```
// これは一行コメント。行末までがコメントとなる。  
  
/*  
// これは正しいコード。  
*/
```

2.6 識別子(Identifiers)

識別子には、大文字小文字のアルファベット、_(アンダースコア)、数字、ユニバーサル文字名、その他の実装依存の文字が使える。大文字と小文字は区別される。

```
int aaa ;  
int AAA ; // aaaとは別の識別子、大文字と小文字は、区別される。  
int bbb ;  
int this_is_an_identifier ;  
int n1234567890 ;  
int \u3042 ; // ユニバーサル文字名  
int 変数 ;
```

ただし、識別子の先頭は、数字から始まってはならない。

```
// エラー
```

```
int 0aaa ;
```

また、キーワードや予約語は、識別子として使うことができない。

```
// エラー、templateはキーワードである。  
int template ;  
// エラー、andは代替表現である。  
int and ;
```

以下の識別子は、特定の文法上の場所で現れた場合に、特別な意味を持つ。これは俗に、文脈依存キーワードとも呼ばれている。

```
final override
```

特定の文法は、識別子が現れることがない場所なので、この文脈依存キーワードは、通常通り何の問題もなく識別子として使うことができる。

```
int final ; // OK  
int override ; // OK
```

以下のような記述も合法だ。

```
// OK、クラス名finalをfinalに指定  
class final final { } ;  
  
struct base  
{  
    virtual void override() { }  
}  
  
struct derived : base  
{  
    // OK、virtual関数名overrideにoverrideを指定  
    virtual void override() override { }  
} ;
```

2.6.1 予約語

予約語とは、C++の実装や標準ライブラリの実装のために予約されていて、使ってはならない名前のことである。ユーザーコードで予約語を使った場合、プログラムの動作は保証されない。

以下のいずれかの条件に当てはまる名前は、あらゆる利用を予約されている。

- ひとつのアンダースコアに大文字から始まる名前
- アンダースコアが二つ連続している、いわゆるダブルアンダースコアを含む名前

以下のコードは、すべて予約名を使っているので、エラーである。

```
int _A ;      // アンダースコアに大文字から始まる名前は、予約されている。
int __a ;    // ダブルアンダースコアを含む名前は、予約されている。
int ___a ;   // 三つの連続したアンダースコアも、ダブルアンダースコアを含むので、使えない。
int a__b ;  // 先頭以外でも、どこかにダブルアンダースコアが含まれている場合、使えない。
int a__ ; // ダブルアンダースコアを含む
int __ ; // ダブルアンダースコアを含む
```

ひとつのアンダースコアから始まる名前は、グローバル名前空間で、利用を予約されている。ただし、グローバル名前空間との名前の衝突が、時として、意外な結果をもたらすこともあるので、言語のルールの詳細を把握していない限り、利用はおすすめできない。

```
// エラー、グローバル名前空間
int _a ;

// OK
namespace NS { int _a ; }

int main()
{
    int _a ; // OK
}
```

ひとつのアンダースコアだけの名前は、予約語ではない。

```
int _ ; // OK
int __ ; // エラー、ダブルアンダースコア
```

簡単にまとめると、アンダースコアから始まる名前は、使うべきではない。ダブルアンダースコアを含む名前は、あらゆる使用を禁止されている。

名前の衝突を防ぐため、しばしばC++の理解の浅い者によってアンダースコアが用いられるが、予約語と衝突した場合、コード自体の動作が保証されなくなってしまうので、名前の衝突を防ぐ目的でアンダースコアを使ってはならない。名前の衝突を防ぐためには、名前空間という言語機能がある。

2.7 キーワード (Keywords)

C++では、以下のキーワードが使われている。これらのキーワードは、プログラム中で特別な意味を持つので、識別子に使うことはできない。

alignas	continue	friend	register
true			
alignof	decltype	goto	reinterpret_cast
try			
asm	default	if	return
typedef			
auto	delete	inline	short
typeid			
bool	do	int	signed
typename			
break	double	long	sizeof
union			
case	dynamic_cast	mutable	static
unsigned			
catch	else	namespace	static_assert
using			
char	enum	new	static_cast
virtual			
char16_t	explicit	noexcept	struct
void			
char32_t	export	nullptr	switch
volatile			
class	extern	operator	template
wchar_t			
const	false	private	this
while			
constexpr	float	protected	thread_local
const_cast	for	public	throw

以下は、キーワードではないが、記号と同じ意味を持つので、識別子に使うことはできない。これは、代替表現(alternative representation)と呼ばれている。

```
and and_eq bitand bitor compl not not_eq or or_eq xor xor_eq
```

2.8 リテラル(Literals)

ここでは、リテラルについて解説している。型については、3.9 型(Types)を参照。

2.8.1 整数リテラル(Integer literals)

整数リテラルには、10進数リテラル、8進数リテラル、16進数リテラルがある。C++には、2進数リテラルは存在しない。

十進数リテラルには、0123456789の文字を使うことができる。

8進数リテラルには、01234567の文字を使うことができる。プレフィクス、0から始まる整数リテラルは、8進数リテラルである。

16進数リテラルには、0123456789, abcdef, ABCDEFの文字を使うことができる。大文字と小文字は、区別されない。プレフィクス、0xから始まる整数リテラルは、16進数リテラルである。

```
// 10進数リテラル
1234 ;

// 8進数リテラル
01234 ; // 10進数では、668

// 16進数リテラル
0x1234 ; // 10進数では、4660
```

0xというプレフィクスが、16進数リテラルを表すのは、他のプログラミング言語でも、よくあることだ。しかし、C++では、8進数リテラルのプレフィクスが変わっている。これは、注意を要する。

```
// if ( x == 8 ) と同じ。
if ( x == 010 )
```

C++の8進数リテラルは、10進数と区別しにくいので、気をつける必要がある。

整数リテラルには、後ろにサフィックスをつけることができる。このサフィックスは、大文字小文字が区別されない。

サフィックス、u、Uは、整数リテラルの型が、unsignedであることを示す。

```
// signed int
auto type1 = 0 ;

// unsigned int
auto type2 = 0u ;
auto type3 = 0U ;
```

サフィックス、l、Lは、型がlong intであることを示す。サフィックス、ll、LLは、型がlong long intであることを示す。

```
// long int
auto type1 = 0l ;
auto type2 = 0L ;
// long long int
auto type3 = 0ll ;
auto type4 = 0LL ;
```

unsignedであることを示すサフィックスと、long int、またはlong long intであることを示すサフィックスは、組み合わせができる。順番は、どちらでもいい。

```
// unsigned long int
auto type1 = 0ul ;
auto type2 = 0lu ;
```

整数リテラルの型の決定は、複雑である。もし、10進数の整数リテラルの値が、intの範囲で表現できない場合は、long intが使われる。long intでも表現できない場合は、long long intが使われる。8進数と16進数の整数リテラルの場合は、unsigned int, unsigned long int, unsigned long long intも、考慮に入れられる。

ただし、l、Lというサフィックスが指定されている場合は、long int以降が使われる。u、Uが指定されている場合は、unsignedな整数型に限られる。

いずれの型でも、整数リテラルの値を表現できない場合は、ill-formedである。

2.8.2 浮動小数点数リテラル(Floating literals)

浮動小数点数のリテラルは、10進数で記述する。

```
1.0 ;
123456789.0 ;
3.14 ;
0.00000001 ;
```

浮動小数点数リテラルには、e, Eに続けて、指数(exponent)を指定することができます。指数を指定すると、小数に、10の指数乗をかけた値になる。

```
// 1 × 101 = 1 × 10 = 10
1e1 ;
1E1 ; // Eでも同じ意味
1e+1 ; // 指数には符号を指定できる。

// 0.12345 × 102 = 0.12345 × 100 = 12.345
0.12345e2 ;

// 12345 × 10-2 = 12345 × 0.01 = 123.45
12345e-2

// 123 × 100 = 123 × 1 = 123
123e0 ;
```

浮動小数点数リテラルの型は、非常に簡単である。サフィックス、f, Fが指定されているリテラルの型は、float、サフィックスが指定されていない型は、double、サフィックス、l, Lが指定されている型は、long doubleである。このサフィックスは、大文字小文字が区別されない。

```
// float
auto f1 = 1.0f ;
auto f2 = 1.0F ;

// double
auto d1 = 1.0 ;
auto d2 = 1.0 ;

// long double
```

```
auto l1 = 1.0l ;
auto l2 = 1.0L ;
```

もし、リテラルの値を、リテラルの型で、完全に表現できない場合は、最も近い値に丸められる。値をどのようにして丸めるかは、実装依存である。もし、リテラルの値を、リテラルの型で表現できない場合は、エラーとなる。

2.8.3 文字リテラル(Character literals)

文字リテラルとは、ある一文字を表すリテラルのことである。文字リテラルは、以下のように記述する。

'x'	通常の文字リテラル (ordinary character literal)
u'x'	char16_t型の文字リテラル
U'x'	char32_t型の文字リテラル
L'x'	wchar_t型の文字リテラル。

プレフィクスのついていない、'x'は、通常の文字リテラル(ordinary character literal)である。型は、charである。このリテラル内に書かれている一文字の値が、そのまま、リテラルの値になる。通常の文字リテラルの具体的な値については、実装依存である。

プレフィクス、Lのついている文字リテラル、L'X'は、ワイド文字リテラルである。型は、wchar_tである。このリテラル内に書かれている一文字の値が、そのまま、リテラルの値になる。ワイド文字リテラルの具体的な値については、実装依存である。

プレフィクス、uのついている文字リテラル、u'x'は、char16_t型の文字リテラルである。これは、一文字のユニバーサル文字セット内の、16bitで表現できるコードポイントに当たる文字を使うことができる。このエンコード方式は、UTF-16と呼ばれている。文字リテラルは、16bitで表せる一文字しか使えないなので、サロゲートペアは使えない。もちろん、文字列リテラルでは、サロゲートペアもサポートされている。

プレフィクス、Uのついている文字リテラル、U'x'は、char32_t型の文字リテラルである。これは、一文字の任意のユニバーサル文字セット内の文字を使うことができる。このエンコード方式は、UTF-32と呼ばれている。

```
// 型はchar、値は実装依存。
auto ordinary_c = 'x' ;
// 型はwchar_t、値は実装依存。
auto wide_c = L'x' ;
// 型は、char16_t、値は、0x3042。
auto u16_c = u'あ' ;
// 型は、char32_t、値は、0x00003042。
```

```
auto u32_c = U'あ' ;
```

2.8.3.1 エスケープシーケンス

文字リテラルの中では、バックスラッシュは、特別なエスケープシーケンスとして扱われる。そのため、バックスラッシュを直接使うことはできない。

エスケープシーケンスは、以下の通り。

```
改行 new-line NL(LF) \n
水平タブ horizontal tab HT \t
垂直タブ vertical tab VT \v
バックスペース backspace BS \b
キャリッジリターン carriage return CR \r
フォームフィード form feed FF \f
アラート文字、ベル文字 alert BEL \a
バックスラッシュ backslash \ \\
疑問符 question mark ? \?
单一引用符 single quote ' \
二重引用符 double quote " \"
```

文字リテラルの中では、二重引用符と、疑問符は、そのまま使うことも出来る。

```
'\n' ; // 改行
'\\' ; // バックスラッシュ

// 二重引用符と疑問符は、そのまま使える。
"" ;   '\\"' ;
'\?' ;  '?' ;
```

// エラー、バックスラッシュはエスケープシーケンスの始まりとみなされるため、使えない。
'\' ;

2.8.4 文字列リテラル(String literals)

文字列リテラル(String Literal)には、通常の文字列リテラル、UTF-8文字列リテラル、char16_t文字列リテラル、char32_t文字列リテラルが存在する。また、それぞれの文字列リテラルに対して、生文字列リテラル(Raw String Literal)という書き方ができる。

文字列リテラルの文法は、以下の通りである。

エンコーディングプレフィクス_{opt} " 文字列 "

文字列リテラルは、文字列の内容で初期化される。ただし、文字列の一番最後には、null文字が付加される。

```
// 以下の二行のコードは同じ意味である。
char s[4] = { 'a', 'b', 'c', '\0' } ;
char s[] = "abc" ;
```

文字列リテラルでは、文字リテラルと同じ、エスケープシーケンスが使える。ただし文字リテラルと違い、二重引用符は、そのまま使うことができないので、エスケープシーケンス、¥"で表さなければならない。単一引用符、'は、そのまま使うことができる。それ以外は、文字リテラルのエスケープシーケンスと、違いはない。

文字列リテラルの型は、要素数nの、constな文字型の配列である。要素数は、文字列リテラル内の文字型の数 + 1である。+ 1は、null文字が付加されるためである。エスケープシーケンスや、ユニバーサル文字名は、一文字として認識される。文字型の数と、文字数とは、同じではない。なぜならば、エンコード方式によっては、ひとつの文字を、複数の文字型で表現するからである。これについては、後述する。

2.8.4.1 エンコード方式

エンコード方式には、実装依存のエンコード、UTF-8、UTF-16、UTF-32がある。

"..."のように、エンコーディングプレフィクス(Encoding Prefix)の指定されていない文字列は、通常の文字列リテラル(ordinary string literal)である。これは、実装依存のエンコード方式が使われる。

```
"abcdefg" ;
"hello" ;
```

u8"..."のように、u8というプレフィクスの指定されている文字列リテラルは、UTF-8文字列リテラルである。これは、UTF-8というエンコードが使われる。

通常の文字列リテラルと、UTF-8文字列リテラルの型は、要素数nの、const charの配列である。

```
// const char [4]
"abc" ;
u8"abc" ;
// const char [1]
"" ;
u8"" ;
```

u"..."のように、小文字のuというプレフィクスの指定されている文字列リテラルは、UTF-16文字列リテラルである。これは、UTF-16というエンコードが使われる。

UTF-16文字列リテラルの型は、要素数nの、const char16_tの配列である。

```
// const char16_t [4]
u"abc" ;
// const char16_t [1]
u"" ;
```

U"..."のように、大文字のUというプレフィクスの指定されている文字列リテラルは、UTF-32文字列リテラルである。これは、UTF-32というエンコードが使われる。

UTF-32文字列リテラルの型は、要素数nの、const char32_tの配列である。

```
// const char32_t [4]
U"abc" ;
// const char32_t [1]
U"" ;
```

L"..."のように、Lというプレフィクスの指定されている文字列リテラルは、ワイド文字列リテラルである。これは、実装依存のエンコードが使われる。

ワイド文字列リテラルの型は、要素数nの、const wchar_tの配列である。

```
// const wchar_t [4]
L"abc" ;
// const wchar_t [1]
L"" ;
```

文字列リテラルの型は、配列であるので、標準変換の、4.2 配列からポインターへの型変換で、ポインターへ、暗黙のうちに変換できる。しかし、C++11では、以前のC++にあった、互換性のための機能が、削られているので、注意を要する。

```
// エラー
char * ptr = "abc" ;
// OK
char const * ptr = "abc" ;
```

なぜか。文字列リテラルの型は、constな文字型の配列である。constは、暗黙のうちに消しさることはできない。たとえば、int型の場合、これは、従来のC++でも、エラーである。

```
int const a[4] = {0} ;
// エラー
int * ptr = a ;
```

今まででは、互換性のためだけの理由で、文字型の配列に限り、constを暗黙のうちに消しがることが、特別に許されていた。しかし、そもそも、文字列リテラルへの変更は、未定義である。

```
// エラー、未定義動作
"abc"[0] = 'd' ;
```

このため、C++では、文字列リテラルは、constな文字型の配列であることが、1993年に決定された。本来、この時点で、constではない文字型へのポインターへの変換は、廃止されるべきであった。しかし、char *に文字列リテラルを代入している既存のコードが、あまりにも多いため、仕方なく、特別なルールを付け加えた。それが、この、文字型の配列に限り、constを消すことができるというルールである。

constを暗黙のうちに消し去ることができると、C++の型システムに穴があいてしまう。いままでは、互換性の問題から、仕方なくこのルールがあったが、C++11では、どうとう、この忌々しい穴は塞がれた。もし、どうしても、既存のコードを利用したい場合は、いくつか方法がある。

```
// 既存の汚いコードの関数
void f( char * ) ;

// 配列を確保して、渡す。
char temp[] = "abc" ;
f( temp ) ;
```

```
// const_castを使用する（非推奨）
f( const_cast<char *>("abc") ) ;
```

そもそも、`const`ではないポインターということは、そのポインターの参照先が、変更されるかもしれないということを意味している。つまり、この関数`f()`は、ポインターの参照先を書き換えるかもしれない。しかし、文字列リテラルの変更は、そもそも未定義である。したがって、`f()`には、書き換えられる配列を確保して、そのポインターを渡すのが、正しい。`const_cast`は、`f()`が、既存のコードであり、今更修正できず、なおかつ、ポインターの参照先を変更しないと保証できる場合に限り、`const_cast`を使うべきである。

これから書くコードは、もちろん、`const`をつけるべきである。本来、変更できないオブジェクトを、非`const`なポインターで参照できるということが、そもそも間違っている。C++11では、このような暗黙の型変換は存在しない。

2.8.4.2 文字列リテラルの型の要素数

文字列リテラルの型の要素数については、注意が必要である。エスケープシーケンスや、ユニバーサル文字名は、一文字として認識される。さらに、文字数=要素数ではない。UTF-8によるエンコードは、一文字が、1~4バイトで表される。

たとえば、“abc”という文字列リテラルは、UTF-8では、以下のようにエンコードされる。二行のコードは、全く同じ意味である。

```
char abc[] = u8"abc" ;
char abc[4] = { 0x61, 0x62, 0x63, 0x00 } ;
```

末尾にnull文字が付加されることを除けば、このコードは、それほど難しくもない。いずれも、一文字がひとつの文字型で表現できる。では、“あ”という文字リテラルを、UTF-8で表現するとどうなるか。

```
char str[] = u8"あ" ;
char str[4] = { 0xe3, 0x81, 0x82, 0x00 } ;
```

UTF-8というエンコード方式では、“あ”(U+3042)を表現するのに、三つの要素が必要である。文字数=要素数ではないということは、よく覚えて置かなければならない。

では、UTF-16はどうか。「あ」という文字に関しては、UTF-16は、ひとつの要素で表現できる。

```
char16_t str[] = u"あ" ;
```

```
char16_t str[2] = { 0x3042, 0x0000 } ;
```

しかし、UTF-16にも、サロゲートペアが存在する。例えば、「吉」(U+020bb7)という古い漢字は、以下のようにエンコードされる。

```
char utf8[] = u8"吉" ;
char utf8[5] = { 0xf0, 0xa0, 0xae, 0xb7, 0x00 } ;

char16_t utf16[] = u"吉" ;
char16_t utf16[3] = { 0xd842, 0xdfb7, 0x0000 } ;
```

このように、UTF-16でも、サロゲートペアが必要なコードポイントについては、二つの要素が必要である。またこの字の場合、UTF-8にいたっては、四つもの要素が必要になる。

UTF-32には、このような問題はない。

```
char32_t utf32[] = U"吉" ;
char32_t utf32[2] = { 0x00020bb7, 0x00000000 } ;
```

ユニバーサル文字セットや、UTFのエンコード方式の詳細は、本書の範疇を超えるので、これ以上深くは解説しない。

また、charとwchar_tのエンコード方式については、実装依存である。

2.8.4.3 生文字列リテラル(Raw String Literal)

文字列リテラルには、直接記述することができない文字が存在する。エスケープシーケンスや、ユニバーサル文字セットだ。その他、改行コードなども、直接記述することはできない。たとえば、以下の文字列を、文字列リテラルを使って書く場合、

```
aaa
bbb
ccc
\\\
"""
```

通常の文字列リテラルでは、以下のように書かなければならない。

```
"aaa\nbbb\nccc\n\\\\\\n\"\"\""
```

これは、非常に分かりにくい。そこで、C++には、生文字列リテラル(Raw String Literal)というものがある。これは、文字列を、そのままの形で書くことができる文字列リテラルである。文法は、以下のようになる。

エンコーディングプレフィクス_{opt} R " デリミター_{opt} (文字列) デリミター_{opt} "

実際の例は、以下の通り。

```
// ""と同じ
R"()" ;

// "aaa\nbbb\nccc\n\\\\\\n\"\"\""と同じ
R"(aaa
bbb
ccc
\\\
""") ;
```

生文字列リテラルの文字列には、バックスラッシュを含めて、あらゆる文字を書くことができる。二重引用符ですら書ける。

問題は、文法の都合上、「)」という文字列が使えないことだ。これは、デリミターを指定すれば、使えるようになる。

```
R"delimiter( )" delimiter" ;
```

ただし、この場合、「)delimiter」」という文字列は、使うことができない。このデリミターは、16文字以内の、基本ソース文字セット内の文字からなる文字列でなければならぬ。

```
// エラー、@という字は、基本ソース文字セットには存在しない。
R"@()@" ;
```

```
// エラー、17文字以上書くことはできない。
R"12345678901234567()12345678901234567" ;
```

基本的に、デリミターは、どのような16文字以内の基本ソース文字セット内の文字列ならば、どのような組み合わせでもよいが、左右のデリミターが一致していなければならぬ。しかし、大抵の場合は、デリミターをわざわざ使う必要はない。

生文字列リテラルは、エンコーディングプレフィクスとともに使うことが出来る。

```
u8R"()" ;
uR"()" ;
UR"()" ;
LR"()" ;
```

生文字列リテラルと、文字列リテラルは、エンコードが同じである場合、連結できる。文字列リテラルのトークンの連結については、2.2 ソースファイルの変換を参照のこと。

```
// "1\n2\n3456"
R"(1
2
3)"
"456" ;
```

2.8.5 boolリテラル (Boolean literals)

boolリテラルは、bool型の真偽を表現するリテラルである。trueとfalseという、二つの値がある。

```
bool t = true ;
bool f = false ;
```

2.8.6 ポインタリテラル (Pointer Literals)

C++には、nullポインターを表すリテラルが存在する。nullptrである。これは、あらゆるポインター型の、nullポインターを表す。

```
void * pointer_to_void = nullptr ;  
int * pointer_to_int = nullptr ;
```

従来のC++では、NULLというマクロを使ったり、0を使ったりしてきた。しかし、マクロは問題が多いし、0にも、問題がある。nullptrポインターは、その内部表現の、すべてのビットが0であることを意味するのではない。ポインターが、何も参照していないということを表す、概念上のものである。そこで、nullptrという、nullptrポインターを表すポインターリテラルが存在する。従来のNULLや、単なる0のかわりに、nullptrを使うべきである。

2.8.7 ユーザー定義リテラル(User-defined literals)

ユーザー定義リテラル(User-defined literal)とは、リテラルを、関数として定義できるようにする機能のことである。整数、浮動小数、文字、文字列に対するユーザー定義リテラルが存在する。

```
// ユーザー定義整数リテラル  
123_x ;  
// ユーザー定義浮動小数リテラル  
1.23_x ;  
// ユーザー定義文字リテラル  
'a'_x ;  
// ユーザー定義文字列リテラル  
"abc"_x ;
```

ユーザー定義リテラルは、演算子のオーバーロードとして、定義する。詳しくは、13.7.9 [ユーザー定義リテラル\(User-defined literals\)](#)を参照のこと。

3 基本事項(Basic concepts)

この章では、C++の根本的なルールを解説する。この章の内容は、言語の深い詳細について解説しているので、すべてを理解する必要はない。必要に応じて参考すればよい。

3.1 宣言と定義(Declarations and definitions)

C++には、名前という概念が存在する。変数や関数、クラス等には、名前をつけられる。名前を使うには、必ず、あらかじめ、その名前が宣言されていなければならない。

3.1.1 宣言(Declaration)と定義(Definition)の違い

宣言とは、ある名前が、その翻訳単位で、何を意味するのかということを、明示するためにある。

定義とは、名前の指し示すものを、具体的に記述することである。宣言と定義は、多くの場合、同時に行われることが多いので、あまり意識しづらい。

```
// これは関数の宣言  
void f( ) ;  
  
// これは関数の宣言と定義  
void f( ) { }
```

以下の例は、関数を宣言だけして、具体的な定義をせずに、使っている。

```
// 関数の宣言  
int f( int ) ;  
  
int main()  
{  
    int x = f( 0 ) ;  
}
```

これは、問題がない。なぜならば、関数を使うには、引数や戻り値の型などが決まってさえいればよいからだ。この場合、その関数を指し示す名前として、fが使われている。この関数 int f(int)の、具体的な実装、つまり、「定義」は、同じ翻訳単位になくても構わない。つまり、int f(int)は、別のソースコードで定義されているかもしれない。

3.1.2 定義ではない宣言

すべての定義は、宣言である。宣言は、定義ではない場合もある。定義ではない宣言は、以下の通りである。

関数宣言で、関数の本体がない場合。

```
// 宣言
```

```
void f( int ) ;  
  
// 宣言と定義  
void f( int ) { }
```

extern指定子を使っていて、初期化子も、関数の本体も記述されていない宣言。

```
// 宣言  
extern int x ;  
  
// 宣言と定義  
int x ;
```

リンクエージ指定されていて、初期化子も、関数の本体も記述されていない宣言。ただし、{}の中の宣言には、影響しない。

```
// 宣言  
extern "C" void f() ;  
  
extern "C"  
{  
// これは、宣言と定義  
    void f() { }  
    int x ;  
  
// 宣言  
    void g() ;  
    extern int y ;  
}
```

クラス名の宣言。

```
// クラス名の宣言  
class C ;  
  
// クラスの宣言と定義  
class C { } ;
```

クラス定義の中の、staticなデータメンバーの宣言。

```
class C  
{  
// 宣言  
    static int x ;  
} ;  
  
// 定義  
int C::x ;
```

enum名の宣言

```
// 宣言  
enum E ;  
  
// 宣言と定義  
enum E { up, down } ;
```

typedef宣言

```
// 宣言  
typedef int type ;
```

using宣言と、usingディレクティブ

```
namespace NS { void f(){} }

// 宣言
using NS::f ;
using namespace NS ;
```

また、`static_assert`宣言、アトリビュート宣言、空宣言は、定義ではない。

```
static_assert( true, "" ) ; // 宣言
[[ ]] ; // 宣言
; // 宣言
```

3.2 ODR(One definition rule)

ODR(One definition rule)とは、定義は原則として、ひとつしか書けないというルールである。

多くの場合、同じ宣言は、いくつでも書ける。ただし、変数、関数、クラス型、enum型、テンプレートの、同じ定義は、ひとつしか書くことができない。

```
// 同じ宣言はいくつでも書ける。
void f() ; void f() ; void f() ; void f() ;

// 定義はひとつしか書けない。
void f() { }

// エラー、定義が重複している
void f() { }
```

定義は、プログラムのすべての翻訳単位で、一つでなければならない。なぜ定義はひとつしか書けないのか。定義が複数あると、問題があるからだ。

```
// 定義が二つある。
int x ;
```

```

int x ;
// どっちのx?
x = 0 ;

// 定義が二つある
void f() { }
void f() { }

// どっちのf()?
f() ;

```

このような問題を防ぐために、定義は、原則として一つでなければならないとされている。

原則としてというのは、例外があるのだ。もし、本当に、定義を一箇所でしか書けないと、困ることがある。たとえば、クラスだ。

```

// 翻訳単位1 A.cpp
// 定義
struct C
{
    int x ;
} ;

C c ; // OK

```

```

// 翻訳単位2 B.cpp
// 宣言
struct C ;

C c ; // エラー

```

翻訳単位2で、クラスCの変数を定義するためには、クラスCは、定義されていなければならない。しかし、すでに、別の翻訳単位で、定義は書かれている。B.cppにも定義を書いてしまうと、ODRに違反する。これは一体、どうすればいいのか。

このため、C++では、クラス型、enum型、外部リンクージを持つインライン関数、クラステンプレート、外部リンクージを持つ関数テンプレート、クラステンプレートのstaticデータメンバー、クラステンプレートのメンバー関数、具体的な型を完全に指定していないテンプレートの特殊化に限り、ある条件を満たせば、別の翻訳単位での、定義の重複を認めている。ある条件とは何か。これには、大きく分けて、二つある。

同じ定義のソースコードは、全く同じトークン列であること。

```
// 翻訳単位1 A.cpp
struct C
{
    int x ;
}
```

```
// 翻訳単位2 B.cpp
struct C
{
public : // エラー。
    int x ;
}
```

ここで、翻訳単位2に、`public :`があろうとなかろうと、意味は変わらない。しかし、全く同じトークン列ではないので、このプログラムはエラーである。

全く同じ複数の定義を管理するのは、極めて困難である。そのため、このように翻訳単位ごとに定義しなければならないクラスやテンプレートは、通常、ヘッダーファイルに記述して、必要な翻訳単位ごとに、`#include`される。

```
// ヘッダーファイル C.h
struct C
{
    int x ;
}
```

```
// 翻訳単位1 A.cpp
#include "C.h"

C c ;
```

```
// 翻訳単位2 B.cpp
#include "C.h"
```

```
C c ;
```

定義の意味が、プログラム中のすべての翻訳単位で、同じであること。

定義のソースコードが、全く同じトークン列であるからといって、意味も同じであるとは限らない。

```
// ヘッダーファイル C.h
class C
{
    void member()
    {
        f() ; // fという名前の、何らかの関数を呼び出す。
    }
} ;
```

このクラス、Cは、member()というメンバー関数で、f()という関数を呼び出している。では、このクラスを使うコードが、以下のようにあれば、どうなるか。

```
// 翻訳単位1 A.cpp

namespace A
{ void f() {} }

// f()はA::f()を呼び出す
using A::f ;

#include "C.h"
```

```
// 翻訳単位2 B.cpp

namespace B
{ void f() {} }

// f()はB::f()を呼び出す
using B::f ;
```

```
#include "C.h"
```

ヘッダーファイルによって、クラスCのソースコードのトークン列は、全く同じなのに、この例では、呼び出す関数が翻訳単位ごとに変わってしまう。このようなコードはエラーである。プログラム中の同じ定義は、必ず同じ意味でなければならぬ。

3.3 スコープ(Scope)

3.3.1 宣言領域とスコープ(Declarative regions and scopes)

宣言された名前には、その名前が有効に使える範囲が存在する。これを、宣言範囲(declarative region)、スコープ(scope)という。

```
int x ;

void f()
{
    int y ;

    {
        int z ;
    }
// ここではもう、zは使えない。
}
// ここではもう、yは使えない。

// xは、ここでも使える。
```

ある名前は、スコープの中ならば、必ず同じ意味であるとは限らない。名前は上書きされる場合がある。

```
void f()
{ // ブロック1
    int x ; // #1
    { // ブロック2
        int x ; // #2
        x = 0 ; // #2が使われる。
    }
}
```

```
x = 0 ; // #1が使われる。
}
```

この例では、ブロック1で宣言されたxは、ブロック2では、別の変数を指し示すxに、隠されている。

このように、スコープがネストする場合、外側のスコープの名前が、内側のスコープの名前に隠されてしまうことがある。

3.3.2 宣言場所(Point of declaration)

スコープには、いくつもの種類がある。これを詳しく説明する前に、まず、宣言された名前は、どこから有効なのかということを、明らかにしておかなければならない。この、名前が有効になる始まりの場所を、宣言場所(Point of declaration)という。名前は、宣言のすぐ直後から有効になる。

```
int x ; // 宣言場所
// ここから、xが使える。
```

宣言場所は、初期化子よりも、前である。

```
int x /*ここから名前xは有効*/ = x ;
```

この例では、xという変数を宣言して、その変数の値で初期化している。このコードに実用的な意味はない。初期化子の中から、宣言された名前は使えるということを示すためだけの例である。

```
// エラー
int x[x /*ここでは、まだxは未定義*/] ;
```

この例は、エラーである。なぜなら、配列の要素数を指定する場所では、xは、まだ定義されていないからだ。これらの例は、通常は気にすることはない、些細な詳細である。一般に、宣言文のすぐ後から使えると考えておけばいい。

3.3.3 ブロックスコープ(Block scope)

6.3 ブロックのスコープは、そのブロックの中である。これを、ブロックスコープと呼ぶ。よく、ローカル変数と呼んでいるものは、ブロックスコープの中で宣言された変数のことである。

```
void f()
{ // ブロック1
int x ;
{ // ブロック2
int y ;
{ // ブロック3
int z ;
// x, y, zが使える
}
// x, yが使える。
}
// xが使える。
}

// ここで使える変数名はない。
```

ブロックはネストできるので、ネストされたブロックの中で、外側のスコープと同じ名前の変数を使いたい場合は、注意が必要である。

```
void f()
{
    int x ;
    {
        int x ; // 外側のスコープのxは隠される。
    }
}
```

関数の仮引数名は、関数本体の一番上のブロックスコープの終わりまで、有効である。

```
void f( int x )
{
    // xはここまで有効
}
// これ以降、xは使えない。
```

3.3.4 関数プロトタイプのスコープ(Function prototype scope)

関数のプロトタイプ宣言にも、スコープがある。関数のプロトタイプ宣言のスコープは、その宣言の終わりまでである。

```
auto f( int x ) -> decltype(x) ;
```

この例では、仮引数の名前が、decltypeに使われている。

3.3.5 関数のスコープ(Function scope)

ブロックスコープではなく、関数自体にも、関数のスコープが存在する。これは、ある関数全体のスコープである。ただし、この関数のスコープが適用されるのは、ラベル名だけである。

```
void f()
{
    {
        label : ;
    }

    goto label ; // labelは、ここでも有効
}
```

このように、ラベル名には、関数のスコープが適用される。

3.3.6 名前空間のスコープ Namespace scope

名前空間のスコープというのは、少しややこしい。まず、名前空間の本体は、もちろんスコープである。

```
namespace NS
{
    int x ;
    // xが使える。
```

```
}
```

// ここでは、xは使えない。

この、名前空間の中の名前(上の例では、x)を、名前空間のメンバーネームという。メンバーネームのスコープは、名前空間の終わりまでである。

ところが、名前空間の本体の定義は、複数書くことができる。

```
namespace NS
{
    int x ;
// xが使える。
}
// ここでは、xは使えない。
```

```
namespace NS
{
    // ここでも、xが使える。
    int y = x ;
}
```

メンバーネームは、その宣言された場所から、後続するすべての同名の名前空間の中で使うことができる。この例の場合、二つめの名前空間NSの定義の中でも、一つめの名前空間NSの定義で宣言されたメンバーネームである、xを使うことができる。

名前空間のメンバーは、スコープ解決演算子、::を使って、参照することもできる。

```
namespace NS
{
    using type = int ;
}

// 名前空間NSの、typeという名前を参照している。
NS::type x ;
```

3.3.6.1 グローバル名前空間(Global namespace)

翻訳単位の、一番上の、namespaceで囲まれていない場所も、一種の名前空間として扱われる。これは、グローバル名前空間と呼ばれている。グローバル名前空間で定義

された名前は、グローバル名前空間のスコープに入る。これは、グローバルスコープとも呼ばれている。グローバル名前空間のスコープは、翻訳単位の終わりまでである。

```
// グローバル名前空間
int x ;

namespace NS
{ // 名前空間、NS

}

// ここは、グローバル名前空間

namespace
{ // 無名名前空間

}

// ここも、グローバル名前空間

// xの範囲は、翻訳単位の終わりまで続く。
```

3.3.7 クラスのスコープ(Class scope)

クラスのスコープは、少し変わっている。ブロックスコープなどは、名前の有効な範囲は、名前を宣言した場所から、スコープの終わりまでである。

```
void f()
{
// ここでは、xは使えない。

int x ; // xを宣言

// ここでは、x使える。
}
```

クラスでは、これが変わっている。

先に、名前が宣言されていなくても、クラス内の関数からは、その名前を使うことができ

る。

```
class C
{
    void f()
    { // 関数の中で、名前を使うことができる。
        type x ;
        value = 0 ;
    }

    type y ;      // エラー。typeは宣言されていない。

    using type = int ; // typeの宣言場所

    type z ; // OK

    int value ;           // valueの宣言場所
} ;
```

また、クラスのメンバー関数を、クラスの外部で定義する場合でも、その関数の中から、クラス内で宣言された名前を使うことができる。

```
class C
{
    void f() ;
    int x ;
} ;

void C::f()
{ // クラス外部で定義されたメンバー関数の中で、クラス内で宣言された名前を使える。
    x = 0 ;
}
```

その他にも、クラス内の名前を、クラス外で使うことができる場合が存在する。

```
class C
{
public :
    int x ;
```

```

        using type = int ;
    } ;

int main()
{
    C c ;
    C * p = &c ;
    // クラスのメンバーアクセス演算子の後に続けて、名前を使える。
    c.x = 0 ;
    p->x = 0 ;

    // スコープ解決演算子の後に続けて、名前を使える。
    C::type value ;
}

```

このように、クラススコープの名前は、宣言した場所から、ある区間まで有効というルールではない。このため、クラスのスコープには特別なルールがある。

- クラスのメンバーの宣言が全てわかったあとに、クラス宣言を再評価して、プログラムの意味が変わるとエラー
- クラス内のメンバーの宣言の順番を変えた際に、プログラムの意味が変わると、エラー

これは、例をあげて説明したほうが分かりやすい。今仮に、このルールがないものとする。すると、以下のようなコードが書けてしまう。

```

// コード1
using type = int ; // #1

class C
{
    type x ; // このtypeは、#1の::type
    using type = float ; // #2
}

```

クラスCの宣言の順番を変えると、以下のコードになる。

```

// コード2
using type = int ; // #1

class C
{
    using type = float ; // #2
}

```

```
    type x ; // このtypeは、#2の、C::type
}
```

このように、メンバーの宣言の順番を変えることによって、プログラムの意味が変わってしまうと、意図せぬバグを生む原因となる。そのため、このようなコードは、エラーである。

3.3.8 enumのスコープ(Enumeration scope)

scoped enumは、enumスコープ(enumeration scope)を持つ。このスコープの範囲は、enumの宣言内だけである。

```
enum class E { x, y, z } ;
// ここで、x, y, zは使えない。
x ; // エラー
E::x ; // OK
```

この理由は、scoped enumは、強い型付けを持つenumだからだ。詳しくは、7.4 [enum](#)を参照のこと。

3.3.9 テンプレート仮引数のスコープ(Template Parameter Scope)

テンプレート仮引数にも、スコープがある。テンプレート仮引数のスコープは、それほど意識する必要はない。

```
template <
typename T, // これ以降、Tを使える。
typename U = T >
class C { } ; // テンプレート仮引数のスコープ、ここまで
```

ただし、テンプレート仮引数名は、基本的に、隠すことができない。

```
template < typename T >
class C
```

```
{
    using T = int ; // エラー

    // エラー
    template < typename T >
    void f() ;

}
```

「基本的に」というのは、隠すことができる場合も存在するからだ。

```
struct Base{ using T = type ; } ;

template < typename T >
struct Derived : Base
{
    T x ; // Base::Tが使われる。テンプレート仮引数ではない。
};
```

といっても、これはよほど特殊な例であり、通常は、テンプレート仮引数名は、隠せないと考えても、問題はない。

3.3.10 名前隠し(Name hiding)

ネストされたスコープの内側で、同じ名前が宣言されると、外側の名前は、隠される。

```
void f()
{ // 外側のスコープ
    int x ;
    { // 内側のスコープ
        int x ; // 外側のスコープのxを隠す。
        x = 0 ; // 内側のx
    }
    x = 0 ; // 外側のx
}
```

派生クラスでは、基本クラスの名前は隠される。

```
struct Base { using type = char ; } ;  
  
struct Derived : Base  
{  
    using type = int ;  
  
    type x ; // int  
} ;
```

クラスやenumの名前は、変数やデータメンバーの名前によって、隠される。

```
class ClassName {} ;  
  
void f()  
{  
    ClassName ClassName ; // OK、ClassName型の変数、ClassName  
  
    ClassName x ; // エラー、ClassNameは、ここでは変数名を指す。  
  
    class ClassName x ; // OK、明示的にクラス名であると指定して  
    //いる。  
}
```

このように、クラス名と変数名を同じにするのは、非常に分かりにくい問題を引き起こすので、あまりおすすめできない。

3.4 名前探索 (Name lookup)

あるスコープにおいて、ある名前が使われているとき、その名前が何を意味するのかということを決定するのを、名前探索 (Name lookup) と呼ぶ。これは一見簡単そうに思える。しかし、この名前を決定するというルールは、非常に難しい。

Name lookupには、大きく分けて、三種類ある。Qualified name lookup、Unqualified name lookup、Argument-dependent name lookupだ。

3.4.1 Qualified 名前探索 (Qualified name lookup)

Qualified nameとは、qualified(修飾)という名前通り、スコープ解決演算子 (::) を使った名前のことである。

```

int g ;

namespace NS { int x ; }

struct C { static int x ; } ;
int C::x ;

enum struct E { e } ;

int main()
{
    // これらはQualified name lookup
    NS::x ; // NSという名前空間のx
    C::x ; // Cというクラスのx
    E::e ; // Eというenumのメンバー、e
    ::g ; // グローバル名前空間のg

}

```

このような名前に対する名前探索を、Qualified name lookupという。

スコープ解決演算子(::)の左側には、クラス名か、名前空間名か、enum名を書くことができる。左側に何も書かない場合、グローバル名前空間が使われる。Qualified name lookupでは、名前は、スコープ解決演算子で指定された、クラスや名前空間、enum内の名前から、探索される。

スコープ解決演算子は、ネストできる。

```

namespace N1 { namespace N2 {
    int x ;
} }

N1::N2::x ;

```

3.4.2 Unqualified 名前探索 (Unqualified name lookup)

Unqualified(非修飾) name lookupは、Qualified name lookup以外を指す。これはつまり、スコープ解決演算子を使わない名前に対する、名前探索である。

```

int g ;

namespace NS { int x ; }

int main()
{
    g ; // グローバル変数のg

    int g ;
    g ; // ローカル変数のg

    {
        using namespace NS ;
        x ; // NS::xと同じ
    }

    {
        using NS::x ;
        x ; // NS::xと同じ
    }
}

```

Unqualified nameに対する名前探索を、Unqualified name lookupという。Unqualified name lookupでは、その名前が使われている場所で、明示的に修飾しなくても、見つかる名前が探される。これは、例えばグローバル名前空間内の名前であったり、クラス内であれば、クラスのメンバーであったりする。また、using directiveや、using declarationの影響をうける。

3.4.3 ADL(Argument-dependent name lookup)

Unqualified nameに対して、関数呼び出しをする場合、特別なルールがある。このルールを、ADL(Argument-dependent name lookup)という。

```

namespace NS
{
    class C {} ;
    void f( C ) {}
}

int main()

```

```
{
    NS::C c ;
    f(c) ; // NS::fを呼ぶ
}
```

このコードでは、通常は見つからないはずの、NSという名前空間内の関数であるfが、Unqualified nameなのにもかかわらず、見つかる。これを、実引数に依存する名前探索(Argument-dependent name lookup)と呼ぶ。しばしば、ADLと略される。また、Andrew Koenigさんが、名前空間の導入によって、特に演算子のオーバーロードで、ADLのような必要性を意見したため、Koenig lookupとも呼ばれることがある。Andrew Koenigさんが、ADLの具体的な仕組みを考案したわけではない。誰がADLの原案を考えだしたのかは、歴史に埋もれて忘れ去られているが、そのような歴史的な経緯と誤解により、Koenig lookupと呼ばれている。

このADLというルールは、一見すると、非常に奇妙なルールである。このような仕組みは、非常に厄介な問題を引き起こすのではないか。事実、ADLは時として、問題になることがある。それでもADLが存在するのは、利点があるからだ。

整数を表現するクラス、Integerを考える。名前の衝突を防ぐため、このクラスは、libという名前空間の中に入れたい。また、整数として分かりやすく使うために、演算子をオーバーロードしたい。Integerクラスは、以下のように使えるものとする。

```
int main()
{
    lib::Integer x ;
    // 演算子のオーバーロードによる、分かりやすい加算のコード。
    x + x ;
}
```

さっそく、このIntegerを実装してみよう。

```
namespace lib
{
    // クラス
    class Integer { /*実装*/ } ;
    // 演算子のオーバーロード
    Integer operator + ( Integer const &, Integer const &)
    {
        // 実装
        return Integer() ;
    }
}
```

もしここで、ADLがない場合、operator +()の呼び出しが、困ったことになる。なぜなら、Unqualified lookupでは、lib名前空間の中の名前を探してはくれない。つまり、operator +は、見つからないのである。

```
lib::Integer x ;
// エラー、operator + が見つからない。
x + x ;
```

ではどうするか。これは、Qualified lookupを使うしかない。

```
lib::operator +( x, x ) ;
```

このコードは動く。確かに動くが、これでは、せっかく演算子をオーバーロードした意味がない。そもそも、演算子をオーバーロードする理由とは、 $x + x$ という、分かりやすい使い慣れたコードを書くためだからだ。

このため、Unqualified nameに対する、関数呼び出しには、Unqualified name lookupに加えて、ADLという仕組みで、名前が探索されるようになっている。

3.4.3.1 関連クラスと関連名前空間

ADLは、その名前が示すとおり、「実引数に依存する名前解決」である。どの名前空間から、名前を探すかということは、実引数の型から決定される。また、ADLは、必ず行われるわけではない。ADLが適用される条件というものが存在する。

ADLはどのように行われるか。まず、関数に対する、関連クラス(Associated class)と、関連名前空間(Associated namespace)というものが決定される。ADLは、この関連名前空間の中から、名前を探索する。

関連クラスとは、関数に実引数として渡される型である。関連名前空間とは、関連クラスがメンバーとなっている名前空間である。

```
namespace NS
{
    class A {} ; class B {} ; class C {} ; class D {} ;
    void f( A, B, C, D ) {}

}

int main()
{
    NS::A a ; NS::B b ; NS::C c ; NS::D d ;
    f( a, b, c, d ) ;
```

}

この場合、fの関数呼び出しに対する関連クラスは、A、B、C、Dで、関連名前空間は、NSとなる。

```
namespace A { class C {} ; }
namespace B
{
    class C {};
    void f( A::C, B::C ) {}
}

int main()
{
    A::C ac ; B::C bc ;
    f( ac, bc ) ;
}
```

この場合、fの関数呼び出しに対する関連クラスは、A::C、B::Cで、関連名前空間は、A、Bとなる。

実引数の型のクラスの、基本クラスも、関連クラスになる。

```
namespace NS
{
    class A {};
    class B : A {};
    class C : B {};

    void f( C ) {}
}

int main()
{
    NS::C c ;
    f( c ) ;
}
```

この場合、関数NS::fに対する関連クラスは、A、B、Cで、関連名前空間は、NSとなる。

実引数がクラステンプレートであった場合、そのクラスのテンプレート実引数も、関連クラスになる。

```

namespace NS
{
    template < typename T > class C {} ;
}

namespace lib
{
    class type {} ;

    template < typename T >
    void f( NS::C<T> ) {}
}

int main()
{
    NS::C< lib::type > c ;

    // 関連クラスは、NS::C< lib::type >と、lib::type。
    // 関連名前空間は、NSと、lib。
    f(c) ; // lib::fを呼び出す。
}

```

テンプレート実引数も関連クラスになるというルールは、この例のような、非常に分かりにくいコードのコンパイルを通してしまった。

実引数がクラス以外の場合も、ADLは適用される。

実引数がenumの場合、そのenumが定義されている名前空間が、関連名前空間になる。

```

namespace NS
{
    enum struct E { value } ;
    void f( E ) {}
}

int main()
{
    f( NS::E::value ) ; // NS::fを呼び出す。
}

```

この場合、関数、NS::fの関連名前空間は、NSとなる。

3.4.3.2 ADLが適用される条件

ADLが適用されるには、条件を満たさなければならない。まず、ADLは、Unqualified nameへの関数呼び出しにしか、適用されない。変数としての使用には、ADLは使われない。

```
namespace NS
{
    class C {} ;

    void f( C ) {}
    void g( C ) {}
}

void g( NS::C ) {}

int main()
{
    NS::C c ;

    f(c) ;          // ADLで、NS::fを呼ぶ
    NS::f(c) ;      // Qualified name lookupが行われる

    ::g(c) ;        // Qualified name lookupが行われる
    NS::g(c) ;      // Qualified name lookupが行われる

    g(c) ;          // エラー。::g、NS::gのどちらの名前か、曖昧。
}
```

最後の例は、Unqualified name lookupで、::gが発見され、ADLで、NS::gが発見されるので、どちらの名前を使うのか、曖昧で、エラーになる。

もし、Unqualified name lookupで、関数名以外の名前が見つかった場合、ADLは行われない。

```
namespace NS
{
    class C {} ;
    void f( C ) {}
}
```

```
int f ;  
  
struct Caller  
{  
    void f( NS::C ) {}  
  
    void g()  
    {  
        NS::C c ;  
        f(c) ; // Caller::fが呼ばれる。ADLは行われない。  
    }  
};  
  
int main()  
{  
    NS::C c ;  
  
    f(c) ; // エラー。fはint型の変数  
}
```

ブロックスコープ関数宣言の名前が見つかった場合、ADLは行われない。ただし、using宣言や、usingディレクティブは、影響しない。

```
namespace NS  
{  
class C {} ;  
  
void f( C ) {}  
}  
  
namespace lib { void f( NS::C ) {} }  
  
int main()  
{  
    NS::C c ;  
  
    {  
        void f( NS::C ) ; // ブロックスコープの関数宣言  
        f(c) ; // ::fを呼び出す。ブロックスコープの宣言が見つかったので、ADLは行われない。  
    }  
}
```

```

{
    using namespace lib ;
    f(c) ; // エラー、ADLも行われるので、曖昧になる。
}

{
    using lib::f ;
    f(c) ; // エラー、ADLも行われるので、曖昧になる。
}
}

// ブロックスコープの関数宣言で参照される、グローバル名前空間のf
void f( NS::C ) {}

```

ブロックスコープ内で関数宣言をするということは、言語上は認められているが、現実的には、あまり用いられていない。

using宣言や、usingディレクティブが、ADLの適用を妨げないということは、注意を要する。これにより、不思議なコンパイルエラーになることがある。例えば、上の例の場合、NS名前空間のコードは、他人が書いたものであり、ユーザーはよく知らないとしよう。lib名前空間のコードは、ユーザーが書いたものである。ユーザーは、lib::fを使いたい。main関数内で多用するので、using宣言を使って、簡単に呼び出せるようにした。ところが、NS名前空間の中にも、同名の関数があるので、曖昧エラーになってしまう。

ADLが意図せず適用された際のエラーは、非常に分かりにくい。そのため、ADLを防ぐための方法が用意されている。名前を括弧で囲めば、ADLの適用が阻害される。

```

namespace NS
{
class C {} ;
void f( C ) {}

void f( NS::C ) {}

int main()
{
    NS::C c ;

    f(c) ; // エラー。曖昧

    (f)(c) ; // OK、ADLは適用されない。::fを呼び出す。
}

```

unqualified nameへの関数呼び出しは、通常のunqualified name lookupと、ADLとで見つかった名前の、両方が用いられる。

3.5 プログラムとリンクエージ(Program and linkage)

C++において、プログラム(program)とは、ひとつ以上の翻訳単位(translation unit)が組み合わさったものである。翻訳単位というのは、一連の宣言で構成されている。

よくあるC++の実装としては、翻訳単位は、ソースファイルという単位で分けられている。

同じオブジェクト、リファレンス、関数、型、テンプレート、名前空間、値を指し示す名前が、宣言によって別のスコープに導入された時、その名前はリンクエージ(linkage)を持つ。

- 名前が外部リンクエージ(external linkage)を持つ場合、その名前が指し示すエンティティは、他の翻訳単位の任意のスコープの中や、同じ翻訳単位の別のスコープの中から参照できる。
- 名前が内部リンクエージ(internal linkage)を持つ場合、その名前が指し示すエンティティは、同じ翻訳単位の別のスコープから参照できる。つまり、別の翻訳単位からは参照できないということを意味する。

名前がリンクエージを持たない場合、その名前が指し示すエンティティは、他のスコープから参照できない。

ある名前がリンクエージを持つのか持たないのか、リンクエージを持つ場合、外部リンクエージなのか内部リンクエージなのかというルールは複雑である。

基本的な考え方としては、グローバル名前空間を含む名前空間スコープの名前は、ほとんどが外部リンクエージを持つ。内部リンクエージを持つものは、大抵、明示的な指定子を伴う。リンクエージを持たないエンティティは、そもそも名前を持たない場合が多い。

名前空間スコープの名前で、内部リンクエージを持つものは、以下の通り。

- 明示的にstaticと宣言されている変数、関数、関数テンプレート

```
static int x ;
static void f() { }
template < typename T >
static void f() { }
```

- 明示的にconstかconstexprと宣言されていて、明示的にexternと宣言されておらず、前方で外部リンクエージをもつと宣言されてもいないもの

```
const int x = 0 ;
```

```
constexpr int y = 0 ;
```

- 無名unionのデータメンバー

```
static union { int x ; } ;
```

無名名前空間か、無名名前空間内で、直接的、間接的に宣言された名前空間は、内部リンクエージを持つ。

```
// この無名名前空間は内部リンクエージを持つ
namespace { }

namespace
{
    // 名前空間internalは内部リンクエージを持つ
    namespace internal
    {
        // 名前空間indirectは内部リンクエージを持つ
        namespace indirect { }
    }
}
```

これ以外の名前空間は、すべて外部リンクエージを持つ。例えば、グローバル名前空間も外部リンクエージを持つ。

名前空間スコープを持つ、次に挙げる種類の名前はすべて、属する名前空間スコープと同じリンクエージを持つ。

- 変数
- 関数
- 名前のあるクラス、もしくは、typedef宣言によって定義された無名クラスで、typedef名をもつクラス

```
// グローバル名前空間

// 外部リンクエージを持つ
struct Named { } ;

// 外部リンクエージを持つ
typedef struct { } Typedef_named ;
```

- 名前のあるenum、もしくは、typedef宣言によって定義された無名enumで、typedef名をもつもの
- 列挙子は、その属するenumのリンクエージに従う

```
// fooのリンクエージはEのリンクエージと同じ
enum E { foo } ;
```

- テンプレート

つまり、名前空間のリンクエージに従う。

```
// グローバル名前空間

namespace ns
{
    int foo ; // 外部リンクエージ
}

namespace
{
    int bar ; // 内部リンクエージ
}
```

メンバー関数、staticデータメンバー、クラススコープの名前のあるクラスやenum、クラススコープのtypedef宣言で定義された無名クラスや無名enumでtypedef名をもつものは、その属するクラスが外部リンクエージを持つ場合、外部リンクエージを持つ。

ブロックスコープで定義された、関数名と、extern宣言された変数名は、リンクエージを持つ。

```
void f()
{
    extern void g() ; // リンクエージを持つ
    extern int global ; // リンクエージを持つ

    g() ; // OK
    global = 123 ; // OK
}

// 別の翻訳単位
```

```
void g() { }
int global ;
```

もし、ブロックスコープの外の直前の名前空間スコープで、同じ名前、同じエンティティでリンクエージだけが違う前方宣言がなされていた場合、ブロックスコープのリンクエージ指定は無視され、前方宣言のリンクエージが使われる。

```
// グローバル名前空間
static void f();           // 内部リンクエージ
static int i = 0;           // 内部リンクエージ

void g() {

    extern void f();        // 内部リンクエージ
    int i;                  // リンクエージなし
    {
        extern void f();    // 内部リンクエージ
        extern int i;       // 外部リンクエージ
    }
}
```

この例では、名前*i*のオブジェクトは3つ存在する、それぞれ、内部リンクエージ、リンクエージなし、外部リンクエージをもつ、別のオブジェクトを指し示している。

もし、ブロックスコープの名前と、前方宣言された名前空間スコープの名前で、名前に適合するエンティティが複数あった場合、エラーとなる。そのような重複がない場合、ブロックスコープのextern指定した名前は、外部リンクエージを持つ。

ブロックスコープで宣言された名前でリンクエージを持つものが、すでに前方宣言されていない場合は、ブロックスコープを包む直前の名前空間のメンバーとなる。ただし、名前空間スコープのメンバーの名前として現れることはない。

```
namespace ns
{
    void f()
    {
        extern void h(); // 名前空間nsのメンバーとなる
        h(); // OK
    }

    void g()
    {
        h(); // エラー、名前hは宣言されていない
    }
}
```

```

    void h() { } // ns::hの宣言と定義
}

// これはns::hとは関係がない、別のエンティティ
void h() { }

```

ここまで条件に当てはまらなかった名前は、リンクエージを持たない。特に、ブロックスコープで宣言された名前のほとんどは、すでに上げた条件にあてはまらない限り、リンクエージを持たない。

型のうち、リンクエージを持つものは以下の通り。

- 名前を持つクラス型、もしくはenum型、typedef宣言内で定義された無名クラス型や無名enum型だが、typedef名を持つもので、その名前がリンクエージを持つもの
- リンクエージを持つクラスの中の、無名クラスと無名enumのメンバー
- クラステンプレートの特殊化

```

template < typename T >
void f() { }

void g()
{
    f<int>(); ;
}

```

この例では、f<int>のみがリンクエージを持つ。f<short>やf<double>のような型は、特殊化されていないので、リンクエージを持たない

- 基本型
- クラスとenum以外の複合型で、リンクエージを持つ型が複合しているもの。
たとえば、int *やint **と言った型リンクエージを持つ
- リンクエージをもつ型がCV修飾された型

リンクエージを持たない型は、リンクエージを持つ関数やクラスで使うことはできない。ただし、以下の場合は使うことができる。

- エンティティがC言語リンクエージをもつ場合
- エンティティが無名名前空間で宣言されている場合
- エンティティがODRの文脈で使われていない場合、もしくは、同じ翻訳単位で宣言されている場合

異なるスコープで宣言された同一の二つの名前は、以下の条件をすべて満たした場

合、同じ変数、関数、型、enum、テンプレート、名前空間を指し示す。

- 名前は両方とも、ともに外部リンクを持つか、ともに内部リンクを持つ、同じ翻訳単位で宣言されている。
- 名前は両方とも、同じ名前空間のメンバーか、派生によらない同じクラスのメンバーを指し示している
- 名前が両方とも関数を指し示す場合は、関数の仮引数の型リストが同一であること
- 名前が両方とも関数テンプレートを指し示す場合は、シグネチャが同じであること

NOTE:ここから先のBasic Conceptsは、最新のC++14ドラフト、N3797を参照している。ここから先はC++11とC++14の差が激しく、最新のドラフトを参考することにした。

3.6 プログラムの開始と終了(Start and termination)

3.6.1 main関数(Main function)

プログラム中にある、mainという名前のひとつのグローバル関数から、プログラムは開始する。freestanding環境でmainの定義を必要とするかどうかは実装依存である。ホスト環境と違い、freestanding環境では、開始と終了も実装依存となる。

main関数は特別な扱いを受ける。C++の実装は、mainを予め定義してはならない。main関数はオーバーロードできない。戻り値の型はintであるが、main関数自体の型は実装依存となる。

規格では、C++の実装は、以下の二つの形のmain関数を受け付けるように規定されている。

- 関数で、仮引数が()で、intを返すもの

```
int main() ;
auto main() -> int ;
```

- 関数で、仮引数が(int, charへのポインターへのポインター)で、intを返すもの

```
int main( int, char ** ) ;
// 仮引数に配列型を書くと、ポインター型になる。
int main( int, char *[] ) ;
```

2つ目の形では、慣習的に、関数の1つ目の仮引数はargcと呼ばれていて、2つ目の仮引数はargvと呼ばれている。argcはプログラムの実行される環境で、プログラムに渡された引数の数を表す。argcがゼロでなければ、引数はargv[0]から、argv[argc-1]まで、

null終端されたマルチバイト文字列の先頭文字へのポインターを指す。argv[0]は、プログラムが実行された時のプログラムの名前か、あるいは""となる。argcの値が負数になることはない。argv[argc]の値は0となる。

```
#include <iostream>

int main( int argc, char ** argv )
{
    std::cout << "Number of arguments: " << argc << std::endl
;
    std::cout << "program name: " << argv[0] << std::endl ;

    // 残りの引数
    for ( std::size_t i = 1 ; argv[i] != nullptr ; ++i )
    {
        std::cout << argv[i] << std::endl ;
    }
}
```

main関数は、プログラム中で使われてはならない。「使う」というのは、呼び出すことはもちろん、アドレスやリファレンスを取ることも含まれる。

```
auto ptr = &main ; // エラー
```

mainのリンクエラーは実装依存である。mainをdelete定義したり、inline, static, constexprなどと宣言することはエラーとなる。

mainという名前は、これ以外には予約されていない。つまり、クラス名やenum名や、クラスのメンバーをmainと名付けることは問題ないし、グローバル名前空間以外の名前空間でmainという名前を使うことも問題はない。

例えば、以下のコードは完全に合法なコードである。

```
class main { } ;
namespace ns
{
    int main() { return 0 ; }
}

int main( ) { }
```

3つのmainという名前は、それぞれ別のエンティティを指す。

`std::exit`などの方法を使い、ブロックから抜け出さずにプログラムを終了させた場合、自動ストレージ上のオブジェクトは破棄されない。もし、`std::exit`が`static`やスレッドストレージ上のオブジェクトを破棄中に呼ばれたならば、プログラムの挙動は未定義である。

`main`関数の中の`return`文は、プログラムからの離脱の効果があり、自動ストレージ上のオブジェクトは破棄され、`return`文のオペランドの値が、`std::exit`への実引数として渡される。もし、`main`関数の最後に到達しても、`return`文がない場合は、以下の文を実行したものと同等の効果になる。

```
return 0 ;
```

これは`main`関数だけの特別な仕様である。

3.6.2 非ローカル変数の初期化 (Initialization of non-local objects)

非ローカル変数には二種類ある。`static`ストレージ上の変数と、スレッドストレージ上の変数である。`static`ストレージ上の非ローカル関数は、プログラムの開始にともなって初期化される。スレッドストレージ上の非ローカル関数は、スレッドの実行にともなって初期化される。初期化は以下の手順で行われる。

`static`ストレージ上とスレッドストレージ上の変数は、他の初期化に先んじて、必ずゼロ初期化される。

変数に定数初期化子(constant initializer)がある場合、定数初期化(constant initialization)が行える場合は、行われる。定数初期化子とは、初期化子が定数式となる式の場合である。

ゼロ初期化と定数初期化をまとめて、静的初期化(static initialization)と呼ぶ。これ以外の初期化はすべて、動的初期化(dynamic initialization)である。静的初期化は、動的初期化の前に行われる。

`static`ストレージ上の非ローカル変数の動的初期化の順序があるものと、順序のないものがある。

明示的に特殊化されたクラステンプレートの`static`データメンバーの初期化は、順序がある。暗黙、明示的に実体化された特殊化の`static`データメンバーの初期化は、順序がない。

```
template < typename T >
struct S
{
    static int data ;
```

```

} ;

// 実行時関数
int init()
{
    static int count = 0 ;
    return count++ ;
}

// 動的初期化される
template < typename T >
int S<T>::data = init() ;

// 明示的実体化
extern template struct S< int > ;
extern template struct S< short > ;

// 明示的特殊化
template < >
int S<double>::data = init() ;
template < >
int S<float>::data = init() ;

int main( )
{
    S<long> s1 ;
    S<long long int> s2 ;
}

```

クラステンプレートSのshort, int, long, long long intに対する特殊化は、暗黙や明示的に実体化されているので、staticデータメンバーdataの初期化の順序はない。そのため、値がどうなるかは、規格上定めることができない。

一方、floatとdoubleについての特殊化は、明示的に特殊化されているので、順序があり、doubleの特殊化がfloatの特殊化より先んじることが保証されている。

この他のstaticストレージ上の非ローカル変数は順序がある。

順序がある変数は、ひとつの翻訳単位の中では、定義されている順番通りに初期化される。

```

int init()
{
    static int count = 0 ;

```

```

    return count++ ;
}

int x = init() ; // 0
int y = init() ; // 1
int z = init() ; // 2

```

この他、規格では、挙動が変わらない場合、実装は動的初期化を静的初期化にしてもよいであるとか、main関数に処理が移った時点で初期化が完了している必要はないなどということを規定している。これは実装の選択や最適化を許すためのもので、コードから見える挙動は変わらない。

3.6.3 終了(Termination)

main関数からreturnするか、std::exitを呼び出した場合、staticストレージ上の初期化されたオブジェクトのデストラクターが呼び出される。スレッドの初期関数からreturnするか、std::exitを呼び出した場合、スレッドストレージ上の初期化されたオブジェクトのデストラクターが呼び出される。

終了時のオブジェクトの破棄中に、すでに破棄されたオブジェクトを参照した場合、挙動は未定義である。

3.7 ストレージ(storage duration)

規格上、ストレージという用語は正しくなく、ストレージの有効期間(storage duration)という用語が正しいのだが、本書では多くの箇所で、簡単にするために、単にストレージという言葉を使っている。

ストレージの有効期間には、様々なものがある。その具体的な実装方法は規定されていない。規格はC++の実装の選択肢となるべく制限しないように書かれているので、細部は規定していない。たとえば、規格はC++のインタープリター実装を禁止していない。ここでは、よくある古典的な実装の一例が、各ストレージをどのように実装しているかを、簡単に記述するが、規格の規定ではないことに注意。

静的ストレージとも呼ばれるstaticストレージの有効期間(static storage duration)は、staticキーワードをつけて宣言したローカル変数やデータメンバー、thread_localをつけて宣言した名前空間スコープの変数などが該当する。

既存のよくあるC++の実装では、静的ストレージは、プログラムのコードなどと一緒にバイナリ上に配置され、そのままメモリ上に読み込まれている。

スレッドストレージの有効期間(thread storage duration)は、thread_localキーワードをつけて宣言した変数が該当する。この変数は、スレッドごとに異なるオブジェクトが割り当てられる。スレッドの解説は本書の範疇ではないので、詳しくは説明しない。

既存のよくあるC++の実装では、スレッドストレージは、TLS(Thread Local Storage)などと呼ばれる、実装依存のスレッドごとに割り当てられた何らかのストレージ、あるいはストレージを指し示すハンドルなどから確保している。

自動ストレージの有効期間(automatic storage duration)は、ブロックスコープで明示的にregisterつきで宣言された変数、あるいは、staticやexternつきで宣言されていない変数が該当する。thread_localつきの変数宣言は暗黙的にstaticでもあるので、該当しない。

既存のよくあるC++の実装では、自動ストレージの有効期間は、スタックと呼ばれる特別なメモリから割り当てられる。

動的ストレージの有効期間(dynamic storage duration)を持つストレージ上のオブジェクトは、new式によって動的に作成され、delete式によって破棄される。

C++実装は、オブジェクトを構築する動的ストレージを確保するためのグローバルな確保関数(allocation function)と、動的ストレージを解放するための解放関数(deallocation function)を提供している。

確保関数は、operator newとoperator new[]で、解放関数はoperator deleteとoperator delete[]となる。

C++の標準ライブラリは、デフォルトの確保関数と解放関数の定義を提供している。これはオーバーロードして、独自の定義を与えることもできる。

確保関数、解放関数のシグネチャは、以下の通り。

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

最後の二つの解放関数は、二番目の仮引数に、確保したストレージのサイズが渡される。

3.7.1 確保関数(allocation function)

確保関数は、クラスのメンバー関数か、グローバル関数でなければならない。グローバル名前空間以外の名前空間で確保関数を宣言したり、グローバル名前空間でstaticとして確保関数を宣言すると、プログラムはエラーとなる。

確保関数の戻り値の型はvoid *でなければならない。最初の仮引数は、std::size_t型で、デフォルト実引数があつてはならない。最初の引数には、確保関数が確保すべきストレージのサイズが渡される。

確保関数はテンプレート宣言できるが、戻り値の型と最初の仮引数は、前述通りでな

ければならない。確保関数のテンプレートは、二つ以上の仮引数を持たねばならない。

確保関数は、要求されたサイズ以上のストレージを確保できたならば、そのストレージの先頭アドレスを返す。ストレージの中身の値については、未規定である。

確保関数の返すポインターは、どの基本アライメント要求も満たすアドレスでなければならない。

確保関数がストレージの確保に失敗した場合、登録されているnewハンドラーがあれば呼び出す。確保関数が無例外指定されている場合は、nullポインターを返す。そうでなければstd::bad_alloc型の例外がthrowされる。

3.7.2 解放関数(deallocation function)

解放関数はクラスのメンバー関数か、グローバル関数でなければならない。グローバル名前空間以外の名前空間で解放関数を宣言したり、グローバル名前空間でstaticとして解放関数を宣言すると、プログラムはエラーとなる。

解放関数の戻り値の型はvoid、最初の仮引数は、void *でなければならない。

C++14では、解放関数が2個の仮引数を持ち、第二引数がstd::size_t型である場合、2つの仮引数には、確保したストレージのサイズが渡される。

解放関数から例外を投げて抜けだした場合、挙動は未定義である。

解放関数の1つ目の実引数がnullポインターでなければ、解放関数はポインターの指示するストレージの解放を行う。

3.7.3 安全なポインター(Safely-derived pointers)

C++11にはガベージコレクションはないが、将来ガベージコレクションを追加することを見越して、安全なポインター(safely-derived pointer)というものを定義している。

ガベージコレクションとは、動的ストレージの明示的な解放をしなくても良くなる言語機能である。

ガベージコレクションの実装方法としては、プログラム中のポインターの値を検証し、どこからも参照されていない動的ストレージを探しだす方法がある。もし、あるストレージを参照する方法がなければ、そのストレージはもはや使われていないのだから、解放しても問題がないということになる。

しかし、C++は、ポインターに対する低級な操作を提供している。厳密には未定義の挙動になるが、ポインターの値をreinterpret_castでそのまま整数型に型変換して、整数型として演算したりできる。このような、ポインターの内部表現を変更して、後から元の内部表現に戻すような処理と、プログラム中の全ポインターの値を検証して、どこからも参照されていないストレージを探しだすというガベージコレクションの機能は、相性が

悪い。

そのため、C++11では、どういうポインターの操作は安全なのかということについて、色々と定義している。その詳細は、本書では解説しない。

3.7.4 サブオブジェクトの有効期間

サブオブジェクトのストレージの有効期間は、その完全なオブジェクトの有効期間に同じ。

3.8 オブジェクトの寿命(Object lifetime)

オブジェクトの寿命(lifetime)の始まりは、オブジェクトの型のサイズとアライメントに適切に対応したストレージが確保された後、もしオブジェクトが非トリビアル初期化を持つならば、その初期化が終わった時点である。

非トリビアル初期化(non-trivial initialization)とは、オブジェクトの型がクラスかアグリゲートで、そのメンバーがひとつでもトリビアルデフォルトコンストラクター以外で初期化されるものをいう。

オブジェクトの寿命の終わりは、オブジェクトが非トリビアルデストラクターを持つのならば、デストラクター呼び出しを開始した時点、そうでなければ、オブジェクトの占めるストレージが解放されるか再利用された時点である。

3.9 型(Types)

トリビアルにコピー可能な型のオブジェクトは、その内部表現のバイト列を、charかunsigned charの配列にコピーできる。コピーされたcharかunsigned charの配列を、再びオブジェクトにコピーしなおした場合、オブジェクトは元の値を保持する。

```
// トリビアルにコピー可能な型Tのオブジェクト
T object ;
// Tを表現するバイト列のサイズ
constexpr std::size_t size = sizeof(T) ;
// T型と同じサイズの配列
unsigned char buffer[size] ;

// 配列にコピー
std::memcpy( buffer, &object, size ) ;
```

```
// 元のオブジェクトにコピーしなおす
std::memcpy( &object, buffer, size ) ;

// objectの値は元のまま
```

トリビアルにコピー可能な型Tのオブジェクトを指し示すポインターが二つあるとして、オブジェクトは基本クラスとしてのサブオブジェクトではない場合、内部表現のバイト列をポインターを経由してコピーすると、コピーされた方はコピーした方の値になる。

```
// Tはトリビアルにコピー可能な型
// ポインターはオブジェクトを指し示しているとする
T * ptr1 ;
T * ptr2 ;

std::memcpy( ptr1, ptr2, sizeof(T) ) ;

// ptr1の指し示すオブジェクトは、ptr2の指し示すオブジェクトと同じ
// 値になる
```

ある型Tのオブジェクトのオブジェクト表現(object representation)は、Nをsizeof(T)とすると、N個のunsigned char型のオブジェクトになる。オブジェクトの値表現(value representation)は、T型の値を保持するためのビット列である。トリビアルにコピー可能な型の場合、値表現は、オブジェクト表現のビット列の値とすることができます。この値は実装依存のビット列の値である。

宣言されているが定義されていないクラス、一部のenum、大きさの分からない配列や不完全な要素型の配列は、不完全に定義されたオブジェクト型である。このような不完全に定義されたオブジェクト型は、そのサイズやストレージ上のレイアウトが、まだ定まっていない。オブジェクトは、不完全な型を持ったまま定義されてはならない。

```
struct incomplete ; // 不完全型

struct error
{
    incomplete i ; // エラー、不完全な型をもったまま定義
} ;
```

クラス型は、翻訳単位のある箇所では不完全で、後に完全になることができる。その場合でも、クラスの型は変わらない。

配列の型は、要素として不完全なクラス型を含み、不完全となることがある。もし、クラスが後に完全になったならば、そのような配列型も、その時点で完全になる。

```

class X ; // 不完全型
using type = X [10] ; // OK、不完全型

type a ; // エラー、配列の要素型が不完全

// クラスXを完全にする
class X { } ;

type b ; // OK、配列の要素型は完全

```

宣言された配列は要素数を不定とすることができます、その時点では不完全な型となる。そのような配列は、後に完全に定義することができる。配列の型は、要素数が指定される前と後とで異なる。要素の型をTとすると、要素数が指定される前の型は、「T型の要素数不定の配列型」であり、要素数がNに定まった後は、「T型のN要素数の配列」となる。

```

extern int a[] ; // int型の要素数不定の配列型

int a[10] ; // int型の要素数10の配列型

```

オブジェクト型(object type)という用語は、CV修飾されているかもしれない型で、関数型、リファレンス型、void型以外の型のことである。

演算型(Arithmetic type)、enum型、ポインター型、メンバーへのポインター型、std::nullptr_t型、そしてこれらの型のCV修飾された型をひっくるめて、スカラー型(scalar type)と呼ぶ。スカラー型とPODクラス、またそのような型の配列と、そのような型がCV修飾された型をひっくるめて、POD型と呼ぶ。スカラー型とトリビアルにコピー可能なクラス型、そのような型の配列、非volatileでconst修飾されたそれらの型をひっくるめて、トリビアルにコピー可能な型(trivially copyable type)と呼ぶ。スカラー型、トリビアルクラス型、そのような型の配列、そのような型のCV修飾された型をひっくるめて、トリビアル型(trivial type)と呼ぶ。スカラー型、標準レイアウトクラス型、そのような型の配列、そのような型のCV修飾された型をひっくるめて、標準レイアウト型(standard layout type)と呼ぶ。

リテラル型(literal type)とは、以下の条件のいずれかひとつを満たす型のことである。

- void
- スカラー型
- リファレンス型
- リテラル型の配列
- クラス型で、以下の条件をすべて満たすもの
 - トリビアルデストラクターを持つ
 - アグリゲート型か、少なくとも一つのconstexprコンストラクターか、constexprコンストラクターテンプレートを持ち、そのコンストラクターとコンストラクター

- テンプレートが、コピーやムーブのコンストラクターではない
- 非staticデータメンバーと基本クラスはすべて、非volatileなりテラル型である

T1型とT2型が同じであるならば、T1とT2は、レイアウト互換型(layout-compatible type)である。

3.9.1 基本型(Fundamental types)

文字(char)のオブジェクトは、基本文字セットをすべて表現できる大きさを持つ。基本文字セットの文字が文字オブジェクトに格納されている場合、文字オブジェクトの整数の値と、その文字の文字リテラルの値は、等しくなる。

```
char c = 'A' ;
c == 'A' ; // true
```

char型が負数を表現できるかどうかは、実装依存である。

```
char c = static_cast<char>(-1) ; // 実装依存
```

文字型は、明示的にunsignedかsignedで宣言できる。char, signed char, unsigned charは、それぞれ別の型として区別される。この3つの型を、ナロー文字型(狭い文字型、narrow character type)と呼ぶ。

ひとつのchar, signed char, unsigned char型のオブジェクトは同じ大きさのストレージを占め、アライメント要求も同じである。つまり、3つの型が同じオブジェクト表現を持つ。

ナロー文字型では、オブジェクト表現を構成するすべてのビット列が、値表現として使われる。符号なしなナロー文字型では、オブジェクト表現のすべてのビット列が、値表現の数値を表現するのに使われる。この点で、ナロー文字型は、他の型にはない独自の特徴を持つ。慣習的にナロー文字型は任意のバイト列を表現するのに使われている。

char型のオブジェクトの値は、signed charかunsigned charのどちらかの値と等しい。どちらと等しくなるのかは、実装に委ねられている。

```
char c = 'A' ;
signed char sc = 'A' ;
unsigned char uc = 'A' ;
```

この例では、規格準拠の実装では、cの値は、scかucのどちらかと必ず等しくなる。どちらと等しいのかは規定されていない。

標準符号つき整数型(standard signed integer type)には、5種類ある。

```
signed char
short int
int
long int
long long int
```

この並び順は、小さい順である。標準符号つき整数型は、少なくともこの順序における、前の型の値と同じかそれ以上の大きさのストレージを占める。

3.9.1 基本型(Fundamental types)

この他に、実装依存の拡張符号つき整数型(extended signed integer type)がある。これは、具体的には規格で定義されないが、実装が独自に提供する符号付きの整数型を指す。標準符号つき整数型と、拡張符号つき整数型をひっくるめて、符号つき整数型(signed integer type)と呼ぶ。

素のintは、実行環境のアーキテクチャにとって自然なサイズとなる。

それぞれの標準符号つき整数型に対応する、標準符号なし整数型(standard unsigned integer type)が存在する。

```
unsigned char
unsigned short int
unsigned int
unsigned long int
unsigned long long int
```

それぞれ、対応する標準符号つき整数型と、同じ大きさのストレージ、同じアライメント要求を持つ。つまり、符号つきの整数型と対応する符号なしの整数型は、それぞれおなじオブジェクト表現を持つ。

```
sizeof( int ) == sizeof( unsigned int ) ; // true
alignof( int ) == alignof( unsigned int ) ; // true
```

標準符号つき整数型と同じように、標準符号なし整数型にも、実装依存の拡張符号なし整数型(extended unsigned integer type)が存在する。これも、それぞれ対応する拡張符号つき整数型とおなじ大きさのストレージとアライメント要求を持つ。

標準符号なし整数型と拡張符号なし整数型をひっくるめて、符号なし整数型(unsigned integer type)と呼ぶ。

符号つき整数型と対応する符号なし整数の型の、値表現は同じである。

標準符号つき整数型と標準符号なし整数型をひっくるめて、標準整数型(standard

integer type)と呼ぶ。拡張符号つき整数型と拡張符号なし整数型をひっくるめて、拡張整数型(extended integer type)と呼ぶ。

C++の整数型は、C言語の標準規格で定義されている要件と同じ要件を満たす。つまり、CとC++はこの点において互換性がある。

符号なし整数は、モジュロの 2^n の法(laws of arithmetic modulo 2^n)に従う。 n は整数型の値表現のビット数である。これは、符号なし整数型は、絶対にオーバーフローしないことが規格上保証されていることを意味する。何故ならば、もある符号なし整数型で表現できる値を超えたとしたならば、その結果は、表現できる最大値での剰余になるからだ。

例えば、規格準拠のC++実装では、以下のコードのnとmの値は、保証されている。

```
int main()
{
    // nの値は、unsigned intで表現できる最大値
    unsigned int max = std::numeric_limits<unsigned
int>::max();

    unsigned int n = max + 1; // nの値は0
    unsigned int m = max + 2; // mの値は1
}
```

wchar_tは、内部型(underlying type)と呼ばれる、何らかの整数型と同じサイズ、符号、アライメント要求を持つ。この内部型は、実装に委ねられている。

wchar_t型のひとつのオブジェクトは、実装がサポートするロケールの文字セットの任意の一文字を表現できる。

wchar_tは、少なくとも、規格上はそうなっている。しかし、現実的には、そのような固定長の文字コードは、サポートするロケールと文字セットを大幅に限定しなければ、存在しない。たとえば、ある実装では、wchar_tは16bitのUTF-16の1単位を表現するようになっている。しかし、UTF-16の1単位は、Unicodeの文字セットの任意の一文字を表現できない。ある実装では32bitのUTF-32の1単位を表現するようになっているが、UTF-32も、1単位で任意の1文字を表せる文字のエンコード方式ではない。wchar_tの内部型が規格で規定されていないことにより、wchar_tの仕様は、移植性の問題を引き起こす。

char16_t型は内部型uint_least16_t型と、char32_t型は内部型uint_least32_t型と、それぞれ等しい大きさ、符号、アライメント要求を持つ。

bool型の取り得る値は、trueかfalseである。bool型には、signed, unsigned short, longといった変種は存在しない。

bool, char, char16_t, char32_t, wchar_t、符号付き整数型と符号なし整数型をひっくるめて、整数型(integral type)もしくはinteger type)と呼ぶ。

整数型の内部表現は、何らかの純粋な二進数であると規定されている。規格は、整数

型の内部表現について実装依存を認めている。例えば、現実の例をだすと、2の補数だとか1の補数だとか、符号の表現方法だとか、エンディアンなど、整数をビット列で表現するには、様々な実装方法が考えられる。C++の規格は、その詳細を規定しない。

浮動小数点数型(floating point type)には、float, double, long doubleがある。doubleは少なくともfloatと同等以上の精度があり、long doubleはdoubleと同等以上の精度がある。

浮動小数点数型が値を表現する方法は、実装依存である。整数型と浮動小数点数型をひっくるめて、演算型(arithmetic type)と呼ぶ。

void型は、空の値を持つ。値を表現しない型のようなものだ。void型は常に不完全な型であり、完全にする方法はない。

void型は、関数が戻り値を返さない場合に、戻り値の型として使われる。

```
// 戻り値を返さない関数
void f() { }
```

あらゆる式は、明示的にvoid型か、CV修飾子つきのvoid型に型変換できる。

```
// int型の値0をvoid型の空の値に変換
static_cast<void>(0) ;
```

void型の式が使える場所は、以下の通り。

- 式文
- コンマ式のオペランド
- 条件演算子の2つ目と3つ目のオペランド
- typeidのオペランド
- noexcept
- decltype
- 戻り値の型がvoidの関数の本体のreturn文の中の式
- void型、CV修飾子つきのvoid型への型変換のオペランド

```
#include <typeinfo>

// 呼び出すとvoid型の式になる関数
void f() { }

void g()
{
    f() ; // 式文
    f() , f() ; // コンマ式のオペランド
```

```

true ? f() : f() ; // 条件演算子
typeid( f() ) ; // typeid
noexcept( f() ) ; // noexcept演算子
using type = decltype( f() ) ; // decltype
return f() ; // return文
static_cast<void>( f() ) ; // void型への型変換
}

```

`std::nullptr_t`型の値は、`null`ポインター一定数である。`null`ポインター一定数には、`nullptr`という特別なキーワードのリテラルがある。`sizeof(std::nullptr_t)`は`sizeof(void*)`と等しくなる。

たとえ、あるC++の実装で、これらの異なる型として認識される基本型の内部表現が同じだったとしても、型は異なるものと認識される。

3.9.2 複合型(Compound types)

複合型とは、ポインターや配列やポインターの配列のように、複数の型が組み合わさって成り立っている型のことである。

複合型は以下の通り。

- 配列
- 関数
- ポインター
- リファレンス
- クラス
- union
- enum
- 非staticなクラスのメンバーへのポインター

これらの複合型は、再帰的に適用できる。例えば、ある型へのポインター、ある型へのポインターへのポインター、ある型へのポインターへのポインターの配列、などといったように。構築された型のオブジェクトのバイト数が、`std::size_t`で表現可能な範囲を超える場合は、エラーとなる。

`void`へのポインターと、オブジェクト型へのポインターをひっくるめて、オブジェクトポインター型(object pointer type)と呼ぶ。

ある型Tのオブジェクトへのポインターのことを、「T型へのポインター」という。C++の規格の文面で単に「ポインター」という場合、メンバーへのポインターは含まない。ただし、`static`メンバーへのポインターは、メンバーへのポインターではなく、ポインターに含まれることに注意。

不完全な型へのポインターは使えるが、そのポインターを使ってできることは制限される。

オブジェクトポインター型の有効な値は、メモリー上のある1バイトへのアドレスを表現し

ているか、nullポインターである。たとえば、T型のオブジェクトが、アドレスAに配置されていて、アドレスAの値となるポインターがあった場合、そのポインターは、オブジェクトを指し示している(ポイントしている)と呼ばれる。

ポインター型の値の表現方法は実装依存である。レイアウト互換な型へのCV修飾されたポインターとCV修飾されていないポインターは、同じ値表現と同じアライメント要求を持つ。

CV修飾されているか、CV修飾されていない、void型へのポインターは、型の分からないオブジェクトを指し示すのに使うことができる。void型へのポインターは特別な扱いになっていて、どのようなオブジェクトポインターをも保持できると規定されている。cv void *型のオブジェクトは、cv char *と同じ値の表現とアライメント要求を持つ。

3.9.3 CV修飾子(CV-qualifiers)

3.9.2 複合型(Compound types)と3.9.1 基本型(Fundamental types)で説明されている型は、CV非修飾型(cv-unqualified type)である。CV非修飾な完全型、不完全型、voidには、三種類のCV修飾された型がある。const修飾された型、volatile修飾された型、const-volatile修飾された型だ。

- constオブジェクトとは、オブジェクトの型が、const Tとなるか、あるいはそのようなオブジェクトのサブオブジェクトである。
- volatileオブジェクトとは、オブジェクトの型が、volatile Tとなるか、あるいはそのようなオブジェクトのサブオブジェクトである。
- const volatileオブジェクトとは、オブジェクトの型が、const volatile Tとなるか、あるいはそのようなオブジェクトのサブオブジェクトである。

ある型のCV修飾された型と、CV非修飾の型は、異なる型である。ただし、同じ表現とアライメント要求を持つ。

CV修飾子には、半順序が存在する。これは、よりCV修飾されている型を比較して決定できる。

その順序は以下の通り。

CV非修飾	<	const
CV非修飾	<	volatile
CV非修飾	<	const volatile
const	<	const volatile
volatile	<	const volatile

3.9.4 lvalueとrvalue(Lvalues and rvalues)

`lvalue`と`rvalue`という用語は、C++の祖先であるC言語のそのまた祖先である、BCPLの頃から、慣習的に使われていた用語である。その本来の意味は、代入式の左右のオペランドに記述することができる値という意味であった。`lvalue`は代入式の左(left)に書くことができる値(value)であるからして、`left value`であり、`rvalue`は右(right)に書けるので`right value`ということだ。

```
int x ;  
x = 0 ;
```

この例で、`x`は代入式の左辺に書けるので`lvalue`であり、`0`は右辺に書けるので`rvalue`である。

今日では、`lvalue`と`rvalue`は、その本来の意味を失い、全く別の意味で使われるようになっている。式の値を分類する用語として使われている。

C++11では、式は、`glvalue`と`rvalue`の二種類に分けることができる。これは更に細分化でき、値は三種類の値に分類することができる。`lvalue`と`xvalue`と`prvalue`である。`glvalue`は`lvalue`と`xvalue`に細分化できる。`rvalue`は`prvalue`と`xvalue`に細分化できる。

分類名の意味は、以下のとおりである。

`lvalue`

`lvalue`は、関数かオブジェクトである。

`lvalue`は、名前付きの変数が指し示すオブジェクトや、ポインターを経由して指し示すオブジェクトなどが該当する。

`lvalue`の名前の由来は、歴史的経緯で`left value`(左辺値)であるが、C++では歴史的経緯の意味とは関係がない。

`xvalue`

`xvalue`は、オブジェクトである。`xvalue`のオブジェクトは大抵、その寿命が近いか、あるいは寿命に関心がないことを表現するために使われる。これにより、もしオブジェクトが`xvalue`であるならば、ムーブしても問題はないということを表現するために使われる。

`xvalue`は、一部の式の結果や、`rvalue`リファレンスへの明示的なキャストなどが該当する。

`xvalue`の名前の由来は、`Expiring value`(消失値)である。これは、`xvalue`というのは寿命が近かったり、寿命に関心がなく、消失しても問題のない値であるという意味から名付けられた。

`glvalue`

glvalueは、lvalueとxvalueの総称である。

glvalueの名前の由来は、generalized lvalue(一般化lvalue)である。

rvalue

rvalueは、xvalueとprvalueの総称である。xvalueの他には、一時オブジェクトや、リテラルの値(123や3.14やtrueなど)や、特定のオブジェクトに関連付けられていない値などが該当する。

rvalueの名前の由来は、歴史的経緯で、right value(右辺値)であるが、C++では歴史的経緯の意味とは関係がない。

prvalue

prvalue(pure rvalue)は、rvalueのうちxvalueではないものである。これには、一時オブジェクトやリテラルの値や、特定のオブジェクトに関連付けられていない値などが該当する。例えば、関数呼び出しの戻り値で、型がリファレンスではないものもある。

C++03までは、リファレンスは、単に「リファレンス」と呼ばれていた。C++11でいうlvalueリファレンスを意味した。C++03のリファレンスには、rvalueは束縛できなかつた。

```
int f() { return 0 ; }

int main()
{
    int & lvalue_ref = f() ; // エラー
}
```

ただし、constなlvalueリファレンスは、rvalueを束縛できるという例外的なルールがある。

```
int f() { return 0 ; }

int main()
{
    int const & lvalue_ref = f() ; // OK
}
```

C++にムーブの概念を持ち込むにあたって、rvalueを非constなリファレンスで束縛したいという需要が生まれた。そのため、従来のリファレンスを、lvalueリファレンスとし、新しくrvalueリファレンスを追加することになった。

```
int f() { return 0 ; }

int main()
{
    int && lvalue_ref = f() ; // OK
}
```

rvalueリファレンスは、rvalueのみを束縛できるリファレンスである。rvalueである以上、寿命がすぐに尽きるか、あるいは、プログラマーはそのオブジェクトの寿命に关心を持たないと明示的に意思表示したとみなすことができる。

そのため、rvalueリファレンスで束縛できたということは、その値の保持する所有権を横取りしても問題がないということになる。

```
class owner
{
private :
    int * ptr ;
public :
    owner( int value )
        : ptr( new int( value ) )
    { }

    // コピーコンストラクター
    owner( owner const & lref )
        : ptr( new int( *lref.ptr ) )
    { }

    // ムーブコンストラクター
    owner( owner && rref )
        : ptr ( rref.ptr )
    {
        rref.ptr = nullptr ;
    }

    ~owner( )
    {
        delete ptr ;
    }
} ;
```

```

owner f()
{
    return owner(123) ;
}

int main()
{
    owner o = f() ;
}

```

rvalueリファレンスの導入により、ストレージなどの確保、解放が必要なリソースや、あるいはファイルやスレッドなどのコピーという概念が存在しないリソースの所有権を、ムーブ(移動)することが可能になった。

このコピーと対をなすムーブという新しい概念は、ムーブセマンティクス(Move Semantics)と呼ばれるプログラミング技法として知られている。プログラミング技法は本書の範疇ではないので、詳しくは解説しない。

3.9.5 アライメント(Alignment)

オブジェクト型には、アライメント要求(alignment requirements)というものが存在する。これは、オブジェクトが構築されるストレージのアドレスに対する制約である。

アライメント(alignment)とは、メモリ上で連続したオブジェクトを構築するときのアドレスの値に対する、実装依存の整数値である。

7.7.1 アライメント指定子を使うことによって、より厳格なアライメントを要求することができる。詳細は7.7.1 アライメント指定子を参照。

```

// アライメント8を要求
alignas( 8 ) char[64] ;

```

5.6.6 [alignof式](#)を使うことによって、型のアライメント要求を得ることができる。詳細は5.6.6 [alignof式](#)を参照。

```

// int型のアライメント要求を取得
constexpr std::size_t align_of_int = alignof( int ) ;

```

以下の例は、連続したメモリ上に確保された二つのint型のオブジェクトの先頭アドレスを表示している。結果は実装により異なる。

```
#include <cstdio>
int main()
{
    int ai[2] ;
    std::printf(
        "a[0]: %p\n"
        "a[1]: %p",
        &ai[0], &ai[1] ) ;
}
```

基本アライメント(fundamental alignment)とは、実装がどのような文脈でもサポートしているアライメントの最大値であり、その値は、`alignof(std::max_align_t)`に等しい。

以下の例は、基本アライメントの数値を出力するコードである。結果は実装により異なる。

```
#include <cstddef>
#include <iostream>

int main()
{
    std::cout << "fundamental alignment is "
        << alignof( std::max_align_t )
        << std::endl ;
}
```

ある型のアライメント要求は、その型が完全なオブジェクトとして使われるか、あるいはサブオブジェクトとして使われるかで、変わる可能性がある。

```
struct B { long double d ; } ;
struct D : virtual B { char c ; } ;
```

たとえば、上の例のDを、完全なオブジェクトとして使った場合、サブオブジェクトとしてBを含むので、`long double`のアライメント要求も考慮してアラインされる。しかし、もしDが、別のオブジェクトのサブオブジェクトであり、その別のオブジェクトが、Bをvirtual基本クラスとして持つ場合、

```
D2 : virtual B, D { char c ; } ;
```

ある実装では、Bのサブオブジェクトは別のオブジェクトのサブオブジェクトとなるかもしれません。DはBをサブオブジェクトとして持たないかもしれない。そのような場合、サブオブジェクトとしてのDのアライメント要求は、Bのアライメント要求に影響されないかも知れない。

これは実装による。alignofの結果は、オペランドの型が完全なオブジェクトとして使われた場合のアライメント要求を返す。

拡張アライメント(extended alignment)は、alignof(std::max_align_t)よりも大きいアライメントである。拡張アライメントがサポートされるかは実装依存である。また、サポートされたとしても、すべての文脈でサポートされないかもしれない。もし、ある文脈で拡張アライメントがサポートされない場合、実装はそのようなコードをエラーにしなければならない。

```
#include <cstddef>
void f()
{
    // この文脈でこの拡張アライメントがサポートされる場合、OK
    // サポートされない場合、エラー
    alignas( alignof( std::max_align_t ) * 2 ) char buf[64];
}
```

拡張アライメントを持つ型のことを、アライン超過型(over-aligned type)という。

```
// アライン超過型の例
struct
alignas( alignof( std::max_align_t ) * 2 )
S { } ;

// これもアライン超過型
struct S2
{
    S s ;
};
```

アライメントは、std::size_t型の値で表現される。妥当なアライメントは、alignof式で返される基本型のアライメントと、実装依存のアライメントである。実装依存のアライメントはサポートされていない可能性もある。アライメントは、必ず、負数ではない2の乗数でなければならない。1, 2, 4, 8, 16のような数値は、実装がサポートしていれば、妥当なアライメントである。3, 5, 6, 7, 9のような数値は、妥当なアライメントではない。

アライメントには、順序がある。この順序は、低い方は「より弱い」(weaker)アライメントといい、高い方は「より強い」(stronger)アライメントとか、「より厳格」(stricter)なアライメントという。より厳格なアライメントは、アライメントの数値としての値が高い。あるアライメント要求を満たすアドレスは、そのアライメント要求より弱いアライメント要求も満たす。

完全型のアライメント要求は、5.6.6 alignof式のオペランドに型を与えることで取得できる。

狭い文字型(char, signed char, unsigned char)は、もっとも弱いアライメント要求を持つ。これにより、狭い文字型を、アラインされたメモリ領域のための内部型として使うことができる。

```
#include <new>

struct S
{
    int i ;
    double d ;
} ;

void f()
{
    // Sを構築するメモリ領域
    alignas(S) char buf [ sizeof(S) ] ;

    // placement newでSを構築
    S * ptr = new( buf ) S ;

    // 疑似デストラクター呼び出し
    ptr->~S() ;
}
```

アライメントは比較することができ、その結果は常識的なものである。

- 二つのアライメントの数値が等しい場合、アライメントは等しい
- 二つのアライメントの数値が異なる場合、アライメントは等しくない
- 二つのアライメントのうち、数値の大きいほうが、より厳格なアライメントである

標準ライブラリには、バッファー上で指定されたアライメント要求を満たすアドレスのポインターを返すとか、指定したアライメント要求を満たすアドレスのストレージを確保するライブラリがある。標準ライブラリは本書の範疇ではないので解説しない。

もある実装で、拡張アライメントがその文脈でサポートされない場合、プログラムはエラーとなる。アライメント要求を指定して動的ストレージを確保する標準ライブラリは、指定されたアライメントに従えない場合、その挙動は確保失敗になる。

4 標準型変換(Standard conversions)

標準型変換(Standard conversion)は、暗黙の型変換とも呼ばれている。C++には、多

の組み込み型があるが、異なる型なのにもかかわらず、キャストを使わず、暗黙的に型を変換できる場合がある。この機能のことを、標準型変換という。

```
short a = 0 ;
int b = a ; // shortからintへ
long c = b ; // intからlongへ
```

この例では、shortからintへ、intからlongへと、型を変換している。すべての標準型変換が、このように分かりやすくて安全だとは限らない。

```
int a = 123456789 ;
float b = a ; // intからfloatへ
b = 0.12345 ;
a = b ; // floatからintへ
```

float型が、int型で表現できる整数の桁をすべて表現できるとは限らない。int型は、整数を表す型であるので、小数点数を正しく表現することはできない。もし、整数と浮動小数点数間で、値を完全に表現できない場合、実装依存の方法で、近い値が使われる。

標準型変換は、人間にとて、できるだけ自然になるように、設計されている。しかし、この標準型変換は、Cから受け継いだ、歴史のある汚い機能なので、どうしても、安全ではない。ここでは、どのような標準型変換があるかを、詳しく説明する。

本書では、普段、「暗黙の型変換」と簡単に呼んでいる標準型変換に、どのようなものがあるのかということを取りあげる。

4.1 lvalueからrvalueへの型変換(Lvalue-to-rvalue conversion)

本書では、煩雑を避けるために省略しているが、多くの標準型変換は、ある型のprvalueの値を、別の型のprvalueの値に変換するようになっている。そのため、標準型変換の際には、必要な場合、glvalueが、自動的にprvalueに変換される。これを、lvalueからrvalueへの型変換という。変換できるglvalueは、関数と配列以外である。

この変換は、通常、まず意識することがない。

4.2 配列からポインターへの型変換(Array-to-pointer conversion)

配列とポインターは、よく混同される。その理由の一つに、配列名が、あたかもポインターのように振舞うことがある。

```

int a[10] ;
// pは、aの先頭要素を指す。
int * p = a ;

// どちらも、配列aの先頭要素に0を代入する
*a = 0 ;
*p = 0 ;

```

これは、配列からポインターへの型変換によるものである。配列名は、配列の先頭要素へのポインターとして扱われる。

```

int a[10] ;

int * p1 = a ; // &a[0]と同じ
int (* p2 )[10] = &a ; // int [10]へのポインター

```

ここで、変数aの型は、int [10]であって、int *ではない。ただし、int *に暗黙のうちに型変換されるので、あたかもポインターのように振舞う。

多くの人は、これを暗黙の型変換としては意識していない。配列からポインターへの型変換は、非常によく使われる変換であって、多くの式では、配列名は、自動的に、配列の先頭要素へのポインターに型変換される。

4.3 関数からポインターへの型変換 (Function-to-pointer conversion)

関数の名前は、その関数へのポインターに型変換される。

```

void f( void ) {}

int main()
{
    // typeは関数ポインターの型
    using type = void (*) (void) ;

    // 同じ意味。
    type p1 = f ;
    type p2 = &f ;
}

```

`f`の型は、関数であって、関数ポインターではない。関数ポインターとは、`&f`である。しかし、関数は、暗黙のうちに、関数ポインターに型変換されるので、関数名`f`は、関数ポインターとしても使うことができる。

この型変換も、非常によく使われる。多くの場合は、自動的に、関数は関数ポインターに変換される。

ただし、この型変換は、非`static`なメンバー関数には適用されない。ただし、`static`なメンバー関数は、この標準変換が適用される。

```
struct C
{
    void f(void) {}
    static void g(void) {}
} ;

// エラー
void ( C::* error )(void) = C::f ;
// OK
void ( C::* ok )(void) = &C::f ;

// staticなメンバー関数は、普通の関数と同じように、変換できる
void (*ptr)(void) = C::g ;
void (*ptr2)(void) = &C::g ; // ただし、こちらの方が分かりやすい
```

このような暗黙の型変換があるとはいえ、通常、関数ポインターを扱う際には、明示的に5.6.1 [単項演算子](#)である`&`演算子を使ったほうが、分かりやすい。

4.4 CV修飾子の型変換 (Qualification conversions)

ある型Tへのポインターは、ある`const`または`volatile`付きの型Tへのポインターに変換できる。

```
int * p ;
int const * cp = p ;
int volatile * vp = p ;
int const volatile * cvp = p ;

cvp = cp ;
```

```
cvp = vp ;
```

これは、より少ないCV修飾子へのポインターから、より多いCV修飾子へのポインターに、暗黙のうちに型変換できるということである。

ただし、ポインターのポインターの場合は、注意を要する。

```
int ** p ;

// エラー
int const ** cp = p ;

// これはOK
int const * const * cp = p ;
```

なぜか。実は、この型変換を認めてしまうと、const性に穴が空いてしまうのだ。

```
int main()
{
    int const x = 0 ;
    int * p ;

    // これはエラー。
    p = &x ;

    // もしこれが認められていたとする。
    // 実際はエラー。
    int const ** cpp = &p ;

    // cppを経由して、pを書き換えることができてしまう。
    *cpp = &x ;

    // pは、xを参照できてしまう。
    *p = 0 ;
}
```

このため、ある型をTとした場合、T **から、T const **への型変換は、認められていない。T **から、T const * const *への変換はできる。

```
int * p = nullptr ;
```

```
int const * const * p2 = &p ; // OK
```

4.5 整数の変換順位 (Integer conversion rank)

整数型には、変換順位というものが存在する。これは、標準型変換や、リスト初期化で考慮される、整数型の優先順位である。これは、それほど複雑な順位ではない。基本的には、型のサイズの大小によって決定される。もっとも、多くの場合、型のサイズというのは、実装依存なのだが。

基本的な変換順位は、以下のようになる。

```
signed char < short int < int < long int < long long int
```

`unsigned`な整数型の順位は、対応する`signed`な型と同じである。

この他にも、いくつか細かいルールがある。

`char`と`signed char`と、`unsigned char`は、同じ順位である。

`bool`は、最も低い順位となる。

`char16_t`、`char32_t`、`wchar_t`の順位は、実装が割り当てる内部的な型に依存する。従って、これらの変換順位は、実装依存である。

拡張整数型、つまり、実装が独自に定義する整数型は、実装依存の順位になる。

4.6 整数のプロモーション (Integral promotions)

整数のプロモーションとは、変換順位の低い型から、高い型へ、型変換することである。ただし、単に順位が低い型から高い型への型変換なら、何でもいいというわけではない。

`bool`、`char16_t`、`char32_t`、`wchar_t`以外の整数型で、`int`より変換順位の低い整数型、つまり、`char`、`short`、その他の実装独自の拡張整数型は、もし、`int`型が、その値をすべて表現できる場合、`int`に変換できる。

```
short s = 0 ;
int i = s ; // 整数のプロモーション
long l = s ; // これは、整数の型変換
```

`int`より低い順位の整数型から、`int`型への変換ということに注意しなければならない。`long`や`long long`への変換、または、`char`から`short`への変換などは、プロモーションではなく、4.7 整数の型変換に分類される。

`char16_t`、`char32_t`、`wchar_t`は、実装の都合による内部的な整数型に変換できる。内部的な整数型というのは、`int`、`unsigned int`、`long int`、`unsigned long int`、`long long int`、`unsigned long long int`のいずれかである。もし、これらのどの型でも、すべての値を表現できないならば、実装依存の整数型に変換することができる。

今、`int`型で、`char16_t`と`char32_t`の取りうるすべての値が表現できるものとすると、

```
char16_t c16 = u'あ' ;
char32_t c32 = U'あ' ;
wchar_t wc = L'あ' ;

int x = 0 ;
x = c16 ; // xの値は0x3042
x = c32 ; // xの値は0x3042
x = wc ; // xの値は実装依存
```

`int`型と`wchar_t`型のサイズは、実装により異なるので、このコードは、実際のC++の実装では、動く保証はない。

基底型が指定されていない`unscoped enum`型は、`int`、`unsigned int`、`long int`、`unsigned long int`、`long long int`、`unsigned long long int`のうち、`enum`型のすべての値を表現できる最初の型に変換できる。もし、どの標準整数型でもすべての値を表現できない場合、すべての値を表現できる実装依存の拡張整数型のうち、もっとも変換順位の低い型が選ばれる。もし、順位の同じ整数型が二つある場合、つまり、`signed`と`unsigned`とが違う場合、`signed`な整数型の方が選ばれる。

基底型が指定されている`unscoped enum`型は、指定された基底型に変換できる。その場合で、さらに整数のプロモーションが適用できる場合も、プロモーションとみなされる。例えば、

```
enum E : short { value } ;

short s = value ; // これは整数のプロモーション
int i = value ; // これも整数のプロモーション
```

このように、`enum`の場合は、`int`型以外への変換でも、プロモーションになる。

`int`型への代入では、`enum`型が、基底型である`short`に変換された後、さらに`int`に変換されている。これは、どちらもプロモーションである。

9.10 ビットフィールドは、すべての値を表現できる場合、`int`に変換できる。

```

struct A
{
    int x:8 ;
};

int main()
{
    A a = {0} ;
    int x = a.x ; // 整数のプロモーション
}

```

もし、ビットフィールドの値が、intより大きいが、unsigned int型で表現できる場合は、unsigned intに変換できる。値がunsigned intより大きい場合は、整数のプロモーションは行われない。整数の型変換が行われる。

bool型の値は、int型に変換できる。falseは0となり、trueは1となる。

```

int a = true ; // aは1
int b = false ; // bは0

```

以上が、整数のプロモーションである。これに当てはまらない整数型同士の型変換は、すべて、次に述べる4.7 [整数の型変換](#)である。

4.7 整数の型変換(Integral conversions)

整数型は、他の整数型に型変換できる。ほんの一例を示すと、

```

short s = 0 ;
int i = s ; // shortからintへの変換
s = i ; // intからshortへの変換

unsigned int ui = i ; // intからunsigned intへの変換
i = ui ; // unsigned intからintへの変換

long l = s ; // shortからlongへの変換
long long ll = l ; // longからlong longへの変換。

```

4.6 [整数のプロモーション](#)以外の整数の型変換は、すべて、整数の型変換になる。この

違いは、オーバーロード解決などに影響するので、重要である。

整数の型変換は、危険である。変換先の型が、変換元の値を表現できない場合がある。

例えば、今、signed charは8ビットで、intは16ビットだと仮定する。

```
#include <limits>

int main()
{
    int i = std::numeric_limits<signed char>::max() + 1 ;
    signed char c = i ; // どうなる？
}
```

signed charは、intの取りうる値をすべて表現できるわけではない。この場合、どうなってしまうのか。

変換先の整数型がunsignedの場合、結果の値は、変換元の対応する下位桁の値である。

具体的な例を示して説明する。

```
// unsigned charが8ビット、unsigned intが16ビットとする

int main()
{
    unsigned int ui = 1234 ;
    unsigned char uc = ui ; // 210
}
```

この場合、unsigned int型は、16ビット、uiの値は、2進数で0000010011010010である。unsigned char型は8ビット。つまり、この場合の対応する下位桁の値は、2進数で11010010(uiの下位8ビット)である。よって、ucは、10進数で210となる。

unsignedの場合、変換先の型が、変換元の値を表現できないとしても、その値がどうなるかだけは、保証されている。もっとも、値を完全に保持できないので、危険なことには変わりないのだが。

変換先の整数型がsignedの場合は、非常に危険である。変換先の整数型が、変換元の値を表現できる場合、値は変わらない。表現できない場合、その値は実装依存である。

今仮に、int型は、signed char型の取りうる値をすべて表現できるが、signed char型は、int型の取りうる値をすべて表現することはできないとする。また、signed charは8ビット、intは16ビットとする。signed charの最小値は-127、最大値は127。intの最小値

は-32767、最大値は32767とする。

```
int main()
{
    signed char c = 100 ;
    int i = c ; // iの値は100

    signed char value = 1000 ; // 値は実装依存

}
```

iの値は、100である。なぜなら、今仮定した環境では、int型は100を表現できるからである。valueの値は、実装依存であり、分からぬ。なぜならば、signed char型は、1000を表現できないからだ。その場合、変換先のsignedな整数型の値は、実装依存である。

4.8 浮動小数点数のプロモーション(Floating point promotion)

float型の値は、double型の値に変換できる。このとき、値は変わらない。つまり、floatからdoubleへの変換は、まったく同じ値が表現できることを意味している。

```
float f = 3.14 ;
double d = f ; // dの値は3.14
```

この変換を、浮動小数点数のプロモーションという。

4.9 浮動小数点数の型変換(Floating point conversions)

浮動小数点数のプロモーション以外の、浮動小数点数同士の型変換を、浮動小数点数の型変換という。

```
double d = 0.0 ;
float f = 0.0 ;
long double ld = 0.0 ;

f = d ; // doubleからfloatへの型変換
ld = f ; // floatからlong doubleへの型変換
```

```
ld = d ; // doubleからlong doubleへの型変換
```

もし、変換先の型が、変換元の型の値を、すべて表現できるのならば、値は変わらない。値を正確に表現できない場合は、最も近い値が選ばれる。この近似値がどのように選ばれるかは、実装依存である。近似値すら表現できない場合の挙動は、未定義である。

4.10 浮動小数点数と整数の間の型変換(Floating-integral conversions)

浮動小数点数型は、整数型に変換できる。このとき、小数部分は切り捨てられる。小数部分を切り捨てた後の値が、変換先の整数型で表現できない場合、挙動は未定義である。

```
int x = 1.9 ; // xの値は、1
int y = 1.9999 ; // yの値は、1
int z = 0.9999 ; // zの値は、0
```

整数型、あるいはunscoped enum型は、浮動小数点数型に変換できる。結果は、可能であれば、まったく同じ値になる。近似値で表現できる場合、実装依存の方法によって、近似値が選ばれる。値を表現できない場合の挙動は、未定義である。

```
float f = 1 ; // fの値は、1.0f
```

4.11 ポインターの型変換(Pointer conversions)

nullポインター定数とは、整数型定数で、0であるものか、std::nullptr_t型である。

```
0 ; // nullポインター定数
1 ; // これはnullポインター定数ではない
nullptr ; // nullポインター定数。型はstd::nullptr_t
```

0がnullポインター定数として扱われる原因是、歴史的な理由である。

nullポインター定数は、どんなポインター型にでも変換できる。この値を、nullポインター値(null pointer value)という。nullポインター定数同士を比較すると、等しいと評価され

る。

```
int * a = nullptr ;
char * c = nullptr ;
int ** pp = nullptr ;

bool b = (nullptr == nullptr) ; // true
```

nullポインター一定数を、CV修飾付きの型へのポインターに変換する場合、このポインターの型変換のみが行われる。CV修飾子の型変換ではない。

```
// ポインターの型変換のみが行われる。
// CV修飾子の型変換は行われない。
int const * p = nullptr ;
```

整数型定数のnullポインター一定数は、std::nullptr_t型に変換できる。結果の値は、nullポインターである。

```
std::nullptr_t null = 0 ;
```

あるオブジェクトへのポインター型は、voidへのポインターに変換できる。

```
int x = 0 ;
int * int_pointer = &x ;
void * void_pointer = int_pointer ; // int *からvoid *に変換できる
```

この時、CV修飾子が付いていた場合、消すことはできない。

```
int x = 0 ;
int const * int_pointer = &x ;

void * error = int_pointer ; // エラー
void const * ok = int_pointer ; // OK
```

void *に変換した場合、ポインターの値は、変換元のオブジェクトのストレージの、先頭を指し示す。値がnullポインターの場合は、変換先の型のnullポインターになる。

派生クラスのポインターから、基本クラスのポインターに変換することができる。

```
struct Base { } ;
struct Derived : Base { } ;

Derived * p = nullptr ;
Base * bp = p ; // OK。Derived *からBase *への変換
```

もし、基本クラスにアクセス出来ない場合や、曖昧な場合は、エラーとなる。

```
// 基本クラスにアクセス出来ない場合
struct Base { } ;
struct Derived : private Base { } ;

Derived * d = nullptr ;
Base * b = d ; // エラー。Baseにはアクセス出来ないので、変換できない
```

```
// 曖昧な場合
struct Base { } ;
struct Wrap1 : Base { } ;
struct Wrap2 : Base { } ;

// Derivedは、基本クラスとしてふたつのBaseを持っている。
struct Derived : Wrap1, Wrap2 { } ;

Derived * ptr = nullptr ;

// エラー
// Wrap1::Baseと、Wrap2::Baseのどちらなのかが曖昧
Base * ambiguous_base = ptr ;

// OK
// Wrap1::Base
Base * Wrap1_base = static_cast<Wrap1 *>(ptr) ;
```

派生クラスのポインターから基本クラスポインターへの変換の結果は、派生クラスの中の、基本クラス部分を指す。これは、変換の結果、ポインターの値が変わることの可能性がある。実装に依存するので、あまり具体的な例を挙げたくないが、例えば、以下のようなコードは、多くの実装で、ポインターの値が変わる。

```
#include <cstdio>

struct Base1 { int x ; } ;
struct Base2 { int x ; } ;
struct Derived : Base1, Base2 { } ;

int main()
{
    Derived d ;

    // dへのポインター
    Derived * d_ptr = &d ;
    std::printf("d_ptr : %p\n", d_ptr) ;

    // 基本クラスのポインターへ型変換
    Base1 * b1_ptr = d_ptr ;
    Base2 * b2_ptr = d_ptr ;

    // 多くの実装では、
    // b1_ptrとb2_ptrのどちらかが、d_ptrと同じ値ではない。
    std::printf("b1_ptr : %p\n", b1_ptr) ;
    std::printf("b2_ptr : %p\n", b2_ptr) ;

    // 派生クラスへキャスト（標準型変換の逆変換）
    Derived * d_ptr_from_b1 = static_cast<Derived *>(b1_ptr)
    ;
    Derived * d_ptr_from_b2 = static_cast<Derived *>(b2_ptr)
    ;

    // 多くの実装では、
    // d_ptrと同じ値になる。
    std::printf("d_ptr_from_b1 : %p\n", d_ptr_from_b1) ;
    std::printf("d_ptr_from_b2 : %p\n", d_ptr_from_b2) ;
}
```

このように、基本クラスと派生クラスの間のポインターのキャストは、ポインターの値の変わることもある。このような型変換には、単に値をそのまま使う、5.5.10 [reinterpret_cast](#)は使えない。

変換元のポインターの値がnullポインターの場合は、変換先の型のnullポインターになる。

4.12 メンバーへのポインターの型変換(Pointer to member conversions)

nullポインター定数は、メンバーへのポインターにも変換できる。変換された結果の値を、nullメンバーpointer値(null member pointer value)という。

```
struct C { int data ; } ;

int C::* ptr = nullptr ;
```

nullメンバーpointer値は、他のメンバーへのポインターの値と比較できる。

```
struct C { int data ; } ;

int C::* ptr1 = nullptr ;
int C::* ptr2 = &C::data ;

bool b = ( ptr1 == ptr2 ) ; // false
```

4.13 boolの型変換(Boolean conversions)

整数、浮動小数点数、unscoped enum、ポインター、メンバーへのポインターは、boolに変換できる。ゼロ値、nullポインター値、nullメンバーpointer値は、falseに変換される。それ以外の値はすべて、trueに変換される。

```
bool b1 = 0 ; // false
bool b2 = 1 ; // true
bool b3 = -1 ; // true

bool b4 = nullptr ; // false

int x = 0 ;
bool b5 = &x ; // true
```

5 式(Expressions)

式(expression)とは、演算子(operator)とオペランド(operand)を組み合わせたものである。オペランドとは、言わば、演算子を適用する引数である。式は、何らかの挙動をし、結果を返す。式の結果は、lvalueかxvalueかprvalueになる。

```
// 演算子は+
// オペランドは1と2
1 + 2 ;
```

組み込み型以外の型に対しては、演算子はオーバーロードされている可能性がある。その場合の挙動については、オーバーロード関数次第である。

式を評価した際、結果が数学的に定義されていない場合や、型の表現できる範囲を超えた場合の挙動は、未定義である。数学的に定義されていない場合というのは、例えばゼロ除算がある。

多くの二項演算子は、数値型やenumをオペランドに取る。この時、二つのオペランドの型が、それぞれ違う場合、型変換が行われる。この型変換のルールは、以下のようになる。

オペランドにscoped enum型がある場合、変換は行われない。もう片方のオペランドが、同じ型でない場合は、エラーになるからだ。

片方のオペランドが浮動小数点数型の場合、もう片方のオペランドは、その浮動小数点数型に変換される。浮動小数点数型の間の優先順位は、long double > double > floatである。

```
// オペランドは、long doubleとdouble
// long doubleに変換される
1.01 + 1.0 ;
1.0 + 1.01 ;

// オペランドは、doubleとint
// doubleに変換される
1.0 + 1 ;
1 + 1.0 ;
```

オペランドが浮動小数点数型ではない場合。つまり、整数型か、unscoped enum型の場合、まず、両方のオペランドに対して、整数のプロモーションが適用される。つまり、int型より変換順位の低い型は、int型に変換される。オペランドがunsignedな整数型の場合は、unsigned intに変換される。その後、両方のオペランドのうち、変換順位が高い方の型に合わせられる。

```
short s = 0 ;
```

```
auto type = s + s ;
```

この場合、オペランドであるsは、両方とも、int型に変換される。その結果、両方のオペランドは同じ型になるので、結果の型はintになる。

```
short s = 0 ;
long l = 0 ;
auto type2 = l + s ;
```

この場合、sはまずint型に変換される。longとintでは、longの方が、変換順位が高いので、結果の型はlongになる。

この時、符号の違う整数型がオペランドになると、非常に複雑な変換が行われるが、本書では解説しない。

5.1 優先順位と評価順序

式には、優先順位と評価順序がある。

優先順位とは、ある式の中で、異なる式が複数使われた場合、どちらが先に評価されるのかという順位である。この優先順位は、人間にとて自然になるように設計されているので、通常、式の優先順位を気にする必要はない。

```
// 1 + (2 * 3)
// operator *が優先される
1 + 2 * 3 ;

int x ;
// operator +が優先される
x = 1 + 1 ;
```

評価順序とは、ある式の中で、優先順位の同じ式が複数使われた場合、どちらを先に評価するかという順序である。これは、式ごとに、「左から右(Left-To-Right)」、あるいは「右から左(Right-To-Left)」のどちらかになる。

たとえば、operator +は、「左から右」である。

```
// (1 + 1) + 1 と同じ
1 + 1 + 1 ;
```

一方、operator = は、「右から左」である。

```
int x ; int y ;
// x = (y = 0) と同じ
x = y = 0 ;
```

これも、人間にとて自然になるように設計されている。通常、気にする必要はない。

5.2 未評価オペランド(unevaluated operand)

5.5.8 [typeid演算子](#)、5.6.3 [sizeof演算子](#)、5.6.7 [noexcept演算子](#)、7.3.6.2 [decltype型指定子](#)では、あるいは未評価オペランド(unevaluated operand)というものが使われる。このオペランドは文字通り、評価されない式である。

オペランドの式は評価されないが、式を評価した結果の型は、通常の評価される式と何ら変わりない。

```
int f()
{
    std::cout << "hello" << std::endl ;
    return 0 ;
}

int main()
{
    // int x ; と同等
    // 関数fは呼ばれない
    decltype( f() ) x ;
}
```

この例では、オペランドの式の結果の型を、変数xとして宣言、定義している。関数呼び出しの結果の型は、関数の戻り値の型になるので、型はintである。ただし、式自体は評価されないので、実行時に関数が呼ばれる事はない。つまり、標準出力にhelloと出力されることはない。

この未評価式は、評価されないということを除けば、通常の式と全く同じように扱われる。例えば、オーバーロード解決やテンプレートのインスタンス化なども、通常の式と同じように働く。

```
// 関数の宣言だけでいい。定義は必要ない
double f(int) ;
int f(double) ;
```

```

int main()
{
    // double x ; と同等
    decltype( f(0) ) x ;
    // int y ; と同等
    decltype( f(0.0) ) y ;
}

```

この例では、関数fは、宣言だけされていて、定義がない。しかし、これは全く問題がない。なぜならば、未評価式は評価されないので、関数fが呼ばれることがない。呼ばれることがなければ、定義も必要はない。

5.3 一次式(Primary expressions)

一次式には、多くの細かな種類がある。例えば、リテラルや名前も一次式である。ここでは、一次式の中でも、特に重要なものを説明する。

5.3.1 :: 演算子

::演算子は、ある名前のスコープを指定する演算子である。このため、非公式に「スコープ解決演算子」とも呼ばれている。しかし、公式の名前は、::演算子(operator ::)である。::に続く名前のスコープは、::の前に指定されたスコープになる。

```

// スコープはグローバル
int x = 0;

// スコープはNS名前空間
namespace NS { int x = 0; }

// スコープはCクラス
struct C { static int x ; } ;
int C::x = 0 ;

int main()
{ // スコープはmain()関数のブロック
    int x = 0 ;

    x ; // ブロック

```

```

::x ; // グローバル
NS::x ; // NS名前空間
C::x ; // クラス
}

```

このように、`::`に続く名前のスコープを指定することができる。スコープが省略された場合は、グローバルスコープになる。

式の結果は、`::`に続く名前が、関数か変数の場合は`lvalue`に、それ以外は`prvalue`になる。

5.3.2 括弧式(parenthesized expression)

括弧式とは、括弧である。これは、式を囲むことができる。括弧式の結果は、括弧の中の式とまったく同じになる。これは主に、ひとつの式の中で、評価する順序を指定するような場合に用いられる。あるいは、単にコードを分かりやすく、強調するために使っても構わない。

(0) ; // 括弧式

```

// 1 + (2 * 3) = 1 + 6 = 7
1 + 2 * 3 ;
// 3 * 3 = 9
(1 + 2 ) * 3

```

ほとんどの場合、括弧式の有無は、括弧の中の式の結果に影響を与えない。ただし、括弧式の有無によって意味が変わる場合もある。例えば、`decltype`指定子だ。

5.4 ラムダ式(Lambda expressions)

ラムダ式(lambda expression)は、関数オブジェクトを簡単に記述するための式である。以下のような文法になる。

```
[ ラムダキャプチャopt ] ( 仮引数 ) mutableopt 例外指定opt -> 戻り値の型opt
```

5.4.1 ラムダ式の基本的な使い方

ラムダ式を、通常の関数のように使う方法を説明する。まず、ラムダ式の構造は、以下のようにになっている。

```
[ /*ラムダキャプチャー*/ ] // ラムダ導入子
( /*仮引数リスト*/ ) // 省略可能
-> void // 戻り値の型、省略可能
{} // 複合文
```

これを、通常の関数定義と比較してみる。

```
auto // 関数宣言
func // 関数名
() // 引数リスト
-> void // 戻り値の型
{} // 関数の定義
```

ラムダ式は、関数オブジェクトである。通常の関数のように、引数もあれば、戻り値もある。もちろん、通常の関数のように、何も引数に取らないこともできるし、戻り値を返さないこともできる。

```
// 通常の関数
auto f() -> void {}
// ラムダ式
[]() -> void {} ;
```

ラムダ式を評価した結果は、prvalueの一時オブジェクトになる。この一時オブジェクトを、クロージャーオブジェクト(closure object)と呼ぶ。クロージャーオブジェクトの型は、クロージャー型(closure type)である。クロージャー型はユニークで、名前がない。これは実装依存の型であり、ユーザーは具体的な型を知ることができない。このクロージャーオブジェクトは、関数オブジェクトと同じように振舞う。

```
auto f = []() ->void {} ;
```

ラムダ式は関数オブジェクトなので、通常の関数と同じように、operator ()を適用することで呼び出すことができる。

```
// 通常の関数
auto f() -> void {}

int main()
{
    f(); // 関数の呼び出し

    // ラムダ式
    auto g = []() -> void {} ;
    g(); // ラムダ式の呼び出し

    // ラムダ式を直接呼び出す
    []() -> void {}() ;
}
```

仮引数リストと、戻り値の型は、省略できる。従って、最小のラムダ式は、以下の通りになる。

[]{ }

仮引数リストを省略した場合は、引数を取らないということを意味する。戻り値の型を省略した場合は、ラムダ式の複合文の内容によって、戻り値の型が推測される。複合文が以下の形になっている場合、

{ return 式 ; }

戻り値の型は、式に lvalue から rvalue への型変換、配列からポインターへの型変換、関数からポインターへの型変換を適用した結果の型になる。

それ以外の場合は、void型になる。

注意しなければならないことは、戻り値の型を推測させるためには、複合文は必ず、{ return 式 ; } の形でなければならない。つまり、return 文ひとつでなければならないということだ。return 文以外に、複数の文がある場合、戻り値の型は void である。

```
// エラー、戻り値の型はvoidだが、値を返している
[]
{
    int x = 0 ;
    return x ;
}();
```

```
// OK、戻り値の型を、明示的に指定している。  
[] -> int  
{  
    int x = 0 ;  
    return x ;  
}();
```

いくつか例を挙げる。

```
// 戻り値の型はint  
auto type1 = []{ return 0 ; }();  
  
// 戻り値の型はdouble  
auto type2 = []{ return 0.0 ; }();  
  
// 戻り値の型はvoid  
[]{}();  
[]  
{  
    int x = 0 ;  
    x = 1 ;  
}();
```

ラムダ式の引数は、通常の関数と同じように記述できる。

```
int main()  
{  
    auto f = []( int a, float b ) { return a ; }  
    f( 123, 3.14f ) ;  
}
```

複合文は、通常の関数の本体と同じように扱われる。

```
int main()  
{  
    // 通常の関数と同じように文を書ける  
    auto f =
```

```

[] {
    int x = 0 ;
    ++x ;
};

f() ;

auto g = []
{ //もちろん、複数の文を書ける
    int x = 0 ;
    ++x ; ++x ; ++x ;
} ;

g() ;
}

```

クロージャーオブジェクトは、変数として保持できる。

```

#include <functional>

int main()
{
    // auto指定子を使う方法
    auto f = []{} ;
    f() ;

    // std::functionを使う方法
    std::function< void (void) > g = []{} ;
    g() ;
}

```

ラムダ式は、テンプレート引数にも渡せる。

```

template < typename Func >
void f( Func func )
{
    func() ; // 関数オブジェクトを呼び出す
}

int main()
{
    f( []{ std::cout << "hello" << std::endl ; } ) ;
}

```

}

ラムダ式の使い方の例を示す。例えば、`std::vector`の全要素を、標準出力に出力したいとする。

```
#include <iostream>
#include <vector>

struct Print
{
    void operator () ( int value ) const
    { std::cout << value << std::endl ; }
} ;

int main()
{
    std::vector<int> v = { 1, 2, 3, 4, 5 } ;
    std::for_each( v.begin(), v.end(), Print() ) ;
}
```

この例では、本質的にはたった一行のコードを書くのに、わざわざ関数オブジェクトを、どこか別の場所に定義しなければならない。ラムダ式を使えば、その場に書くことができる。

```
#include <vector>
#include <algorithm>

int main()
{
    std::vector<int> v = { 1, 2, 3, 4, 5 } ;
    std::for_each( v.begin(), v.end(),
        [] (int value){ std::cout << value << std::endl ; } )
};
```

5.4.1.1 変数のキャプチャー

関数内に関数を書くことができるのは、確かに手軽で便利だ。しかし、ラムダ式は、単にその場に関数を書くだけでは終わらない。ラムダ式は、関数のローカル変数をキャプ

チャーできる。

```
#include <iostream>

template < typename Func >
void call( Func func )
{
    func() ; // helloと表示する
}

int main()
{
    std::string str = "hello" ;
    // main関数のローカル変数strを、ラムダ式の中で使う
    auto f = [=]{ std::cout << str << std::endl ; } ;
    f() ;
    //もちろん、他の関数に渡せる。
    call( f ) ;
}
```

このように、ラムダ式が定義されている関数のブロックスコープの中のローカル変数を、使うことができる。この機能を、変数のキャプチャという。

この、ラムダ式で、定義されている場所のローカル変数を使えるというのは、一見、奇妙に思えるかもしれない。しかし実のところ、これは単なるシンタックスシュガーにすぎない。同じことは、従来の関数オブジェクトでも行える。詳しくは後述する。

もちろん、クロージャーオブジェクトがどのように実装されるかは、実装により異なる。しかし基本的に、ラムダ式は、このような関数オブジェクトへの、シンタックスシュガーに過ぎない。

□の中身を、ラムダキャプチャという。ラムダキャプチャの中には、キャプチャーリストを記述できる。変数のキャプチャをするには、キャプチャーリストに、どのようにキャプチャをするかを指定しなければならない。変数のキャプチャには、二種類ある。コピーでキャプチャするか、リファレンスでキャプチャするかの違いである。

```
int main()
{
    int x = 0 ;

    // コピーキャプチャ
    [=] { x ; }
    // リファレンスキャプチャ
    [&] { x ; }
```

}

キャプチャリストに=を記述すると、コピーキャプチャーになる。&を記述すると、リファレンスキャプチャーになる。

コピーキャプチャーの場合、変数はクロージャオブジェクトのデータメンバーとして、コピーされる。リファレンスキャプチャーの場合は、クロージャオブジェクトに、変数への参照が保持される。

コピーキャプチャーの場合は、ラムダ式から、その変数を書き換えることができない。

```
int main()
{
    int x = 0 ;

    [=]
    { // コピーキャプチャー
        int y = x ; // OK、読むことはできる
        x = 0 ; // エラー、書き換えることはできない
    } ;

    [&]
    { // リファレンスキャプチャー
        int y = x ; // OK
        x = 0 ; // OK
    } ;
}
```

これは、クロージャオブジェクトのoperator()が、const指定されているためである。ラムダ式にmutableが指定されていた場合、operator()は、const指定されないので、書き換えることができる。

```
int main()
{
    int x = 0 ;

    [=]() mutable
    {
        int y = x ; // OK
        x = 0 ; // OK
    } ;
}
```

リファレンスキャプチャーの場合は、変数の寿命に気をつけなければならない。

```
#include <functional>

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

    {
        int x = 0 ;
        f = [&]{ x ; } ; // リファレンスキャプチャー
        g = [=]{ x ; } ; // コピーキャプチャー
    }

    f() ; // エラー、xの寿命は、すでに尽きている。

    g() ; // OK
}
```

ローカル変数の寿命は、そのブロックスコープ内である。この例で、fを呼び出すときには、すでに、xの寿命は尽きているので、エラーになる。

ラムダ式がキャプチャーできるのは、ラムダ式が記述されている関数の、最も外側のブロックスコープ内である。

```
int main()
{ // 関数の最も外側のブロックスコープ
    int x ;
    {
        int y ;

        // xもyもキャプチャーできる。
        [=]{ x ; y ; } ;
    }
}
```

関数の最も外側のブロックスコープ以外のスコープ、例えばグローバル変数などは、キャプチャーせずにアクセス出来る。

```
// グローバルスコープの変数
int x = 0 ;

int main()
{
    // キャプチャ―する必要はない
    []{ x ; } ;
}
```

変数ごとに、キャプチャ―方法を指定できる。

```
int main()
{
    int a = 0 ;
    int b = 0 ;

    [ a, &b ]{} ;
}
```

変数のキャプチャ―方法を、それぞれ指定する場合、キャプチャーリストの中に、変数名を記述する。その時、単に変数名だけを記述した場合、コピーキャプチャ―になり、変数名の前に&をつけた場合、リファレンスキャプチャ―になる。

キャプチャ―したい変数がたくさんある場合、いちいち名前をすべて記述するのは面倒であるので、デフォルトのキャプチャ―方法を指定できる。これをデフォルトキャプチャ―(default capture)という。この時、デフォルトキャプチャ―に続けて、個々の変数名のキャプチャ―方法を指定できる。

```
int main()
{
    int a = 0 ; int b = 0 ; int c = 0 ; int d = 0 ;

    // デフォルトはコピーキャプチャ―
    [=]{ a ; b ; c ; d ; } ;
    // デフォルトはリファレンスキャプチャ―
    [&]{ a ; b ; c ; d ; } ;

    // aのみリファレンスキャプチャ―
    [=, &a]{} ;

    // aのみコピーキャプチャ―
    [&, a]{} ;
```

```
// a, bのみリファレンスキャプチャー
[=, &a, &b]{} ;

// デフォルトキャプチャーを使わない
[a]{} ;

}
```

このとき、デフォルトキャプチャーと同じキャプチャー方法を、個々のキャプチャーで指定することはできない。

```
int main()
{
    int a = 0 ; int b = 0 ;

    // エラー、デフォルトキャプチャーと同じ
    [=, a]{} ;
    // OK
    [=, &a]{} ;

    // エラー、デフォルトキャプチャーと同じ
    [&, &a]{} ;
    // OK
    [&, a]{} ;
}
```

キャプチャーリスト内で、同じ名前を複数書くことはできない。

```
int main()
{
    int x = 0 ;
    // エラー
    [x, x]{} ;
}
```

たとえ、キャプチャーメソッドが同じであったとしても、エラーになる。

デフォルトキャプチャーが指定されているラムダ式の関数の本体で、キャプチャーできる変数を使った場合、その変数は、暗黙的にキャプチャーされる。

変数のキャプチャーメソッドの具体的な使用例を示す。今、vectorの各要素の合計を求めるプログラムを書くとする。関数オブジェクトで実装をすると、以下のようになる。

```

struct Sum
{
    int sum ;
    Sum() : sum(0) { }
    void operator ()( int value ) { sum += value ; }
} ;

int main()
{
    std::vector<int> v = {1,2,3,4,5} ;
    Sum sum = std::for_each( v.begin(), v.end(), Sum() ) ;

    std::cout << sum.sum << std::endl ;
}

```

これは、明らかに分かりにくい。`sum += value`という短いコードのために、関数オブジェクトを定義しなければならないし、その取扱も面倒である。このため、多くのプログラマは、STLのアルゴリズムを使うより、自前のループを書きたがる。

```

int main()
{
    std::vector<int> v = {1,2,3,4,5} ;
    int sum = 0 ;
    for ( auto iter = v.begin() ; iter != v.end() ; ++iter )
    {
        sum += *iter ;
    }

    std::cout << sum << std::endl ;
}

```

しかし、ループを手書きするのは分かりにくいし、間違いの元である。ラムダ式のキャプチャーは、この問題を解決してくれる。

```

int main()
{
    std::vector<int> v = {1,2,3,4,5} ;
    int sum = 0 ;
    std::for_each( v.begin(), v.end(),
        [&]( int value ){ sum += value ; } )
}

```

```
) ;

    std::cout << sum << std::endl ;
}
```

これで、コードは分かりやすくなる。また、ループを手書きしないので、間違いも減る。

5.4.2 ラムダ式の詳細

5.4.2.1 クロージャーオブジェクト(closure object)

ラムダ式が評価された結果は、クロージャーオブジェクト(closure object)になる。これは、一種の関数オブジェクトで、その型は、ユニークで無名な実装依存のクラスであるとされている。この型は、非常に限定的にしか使えない。例えば、ラムダ式は、未評価式の中で使うことが出来ない。これは、decltypeやsizeofの中で使うことが出来ないということを意味する。

```
using type = decltype([]{}) ; // エラー
sizeof([]{}) ; // エラー

// OK
auto f = []{} ;
```

クロージャーオブジェクトがどのように実装されるかは、実装依存である。しかし、今、説明のために、実装の一例を示す。

```
int main()
{
    int a = 0 ; int b = 0 ;
    auto f = [a, &b](){ a ; b ; } ;
    f() ;
}
```

例えば、このようなコードがあったとすると、例えば、以下のように実装できる。

```
class Closure // 本来、ユーザー側から使える名前は存在しない
```

```

{
private :
    // aはコピーキャプチャー、bはリファレンスキャプチャー
    int a ; int & b ;
public :
    Closure(int & a, int & b )
        : a(a), b(b) { }

    // コピーコンストラクターが暗黙的に定義される
    Closure( Closure const & ) = default ;
    // ムーブコンストラクターが暗黙的に定義される可能性がある
    Closure( Closure && ) = default ;

    // デフォルトコンストラクターはdelete定義される
    Closure() = delete ;
    // コピーダイアゴノカル演算子はdelete定義される
    Closure & operator = ( Closure const & ) = delete ;

    inline void operator () ( void ) const
    { a ; b ; }
} ;

int main()
{
    int a = 0 ; int b = 0 ;
    auto f = Closure(a, b) ;
    f() ;
}

```

クロージャーオブジェクトは、メンバー関数として、operator ()を持つ。これにより、関数呼び出しの演算子で、関数のように呼び出すことができる。キャプチャーした変数は、データメンバーとして持つ。このoperator ()は、inlineである。また、mutable指定されていない場合、const指定されている。これにより、コピーキャプチャーした変数は、書き換えることができない。mutableが指定されている場合、constではなくなるので、書き換えることができる。

```

int main()
{
    int x = 0 ;
    // エラー
    [x]() { x = 0 ; } ;
}

```

```
// OK
[]() mutable { x = 0; } ;
}
```

ラムダ式の仮引数リストには、デフォルト引数を指定できない。

```
// エラー
[](int x = 0){} ;
```

ラムダ式は、例外指定できる。

```
[]() noexcept {} ;
```

ラムダ式に例外指定をすると、クロージャーオブジェクトのoperator ()に、同じ例外指定がなされたものと解釈される。

クロージャーオブジェクトには、コピー構造体が暗黙的に定義される。ムーブ構造体は、可能な場合、暗黙的に定義される。デフォルト構造体と、コピー代入演算子は、delete定義される。これはつまり、初期化はできるが、コピー代入はできないということを意味する。

```
// 初期化はできる。
auto f = []{} ;

// OK fはラムダ式ではないので可能
using closure_type decltype(f) ;
// OK 初期化はできる
closure_type g = f ;

// エラー、デフォルト構造体は存在しない
closure_type h ;
// エラー、コピー代入演算子は存在しない。
h = f ;
```

関数ポインターへの変換

ラムダキャプチャを使わないラムダ式のクロージャオブジェクトは、同一の引数と戻り値の関数ポインターへの変換関数を持つ。

```
void (*ptr1)(void) = []{} ;
auto (*ptr2)(int, int, int) -> int = [](int a, int b, int c)
-> int { return a + b + c ; };

// 呼び出す。
ptr1() ; ptr2(1, 2, 3) ;
```

ラムダキャプチャを使っているクロージャオブジェクトは、関数ポインターに変換できない。

```
int main()
{
    int x = 0 ;
    // エラー、変換できない
    auto (*ptr1)(void) -> int = [=] -> int{ return x ; } ;
    auto (*ptr2)(void) -> int = [&] -> int{ return x ; } ;
}
```

変数をキャプチャしないラムダ式というのは、関数オブジェクトではなく、単なる関数に置き換えることができるので、このような機能が提供されている。この機能は、まだテンプレートを使っていない既存のコードやライブラリとの相互利用のために用意されている。

ラムダ式のネスト

ラムダ式はネストできる。

```
[]{ // 外側のラムダ式
    []{} ; // 内側のラムダ式
};
```

この時、問題になるのは、変数のキャプチャだ。内側のラムダ式は、外側のラムダ式のブロックスコープから見える変数しか、キャプチャすることはできない。

```
int main()
{
```

```

int a = 0 ; int b = 0 ;
[b]{ // 外側のラムダ式
    int c = 0 ;
    [=]{ // 内側のラムダ式
        a ; // エラー、aはキャプチャできない。
        b ; // OK
        c ; // OK
    } ;
} ;
}

```

外側のラムダ式が、デフォルトキャプチャによって、暗黙的に変数をキャプチャしている場合は、内側のラムダも、その変数をキャプチャできる。

```

int main()
{
    int a = 0 ;
    [=]{ // 外側のラムダ式
        [=]{ // 内側のラムダ式
            a ; // OK
        } ;
    } ;
}

```

5.4.3 thisのキャプチャ

基本的にラムダ式は、そのラムダ式が使われているブロックスコープのローカル変数しかキャプチャできない。しかし、実は、データメンバーを使うことができる。

```

struct C
{
    int x ;
    void f()
    {
        [=]{ x ; } ; // OK、ただし、これはキャプチャではないこ
        とに注意
    }
};

```

このように、非staticなメンバー関数のラムダ式では、データメンバを使うことができる。しかし、これは、データメンバーをキャプチャーしているわけではない。その証拠に、データメンバーを直接キャプチャーしようとすると、エラーになる。

```
struct C
{
    int x ;
    void f()
    {
        [x]{} ; // エラー、データメンバーはキャプチャーできない
    }
} ;
```

では、どうしてデータメンバーが使えるのか。一体何をキャプチャーしているのか。実は、これはthisをキャプチャーしているのである。ラムダ式は、thisをキャプチャーできる。

```
struct C
{
    int x ;
    void f()
    {
        [this]{ this->x ; } ;
    }
} ;
```

ラムダ式の関数の本体では、thisは、クロージャーオブジェクトへのポインターではなく、ラムダ式が使われている非staticなメンバー関数のthisをキャプチャーしたものと解釈される。thisは、必ずコピーキャプチャーされる。というのも、そもそもthisはポインターなので、リファレンスキャプチャーしても、あまり意味はない。

```
struct C
{
    int x ;
    void f()
    {
        [this]{} ; // OK
        [&this]{} ; // エラー、thisはリファレンスキャプチャーで
        // きない
    }
} ;
```

ラムダ式にデフォルトキャプチャが指定されていて、データメンバーが使われている場合、thisは暗黙的にキャプチャされる。デフォルトキャプチャがコピーでもリファレンスでも、thisは必ずコピーキャプチャされる。

```
struct C
{
    int x ;
    void f()
    {
        [=]{ x ; } ; // thisをコピーキャプチャする
        [&]{ x ; } ; // thisをコピーキャプチャする
    }
} ;
```

thisのキャプチャは、注意を要する。すでに述べたように、データメンバーは、キャプチャできない。ラムダ式でデータメンバーを使うということは、thisをキャプチャすることである。データメンバーは、thisを通して使われる。これは、データメンバーは参照で使われるということを意味する。ということは、もし、クロージャーオブジェクトのoperator()が呼ばれた際に、thisを指示すオブジェクトが無効になっていた場合、エラーとなってしまう。

```
struct C
{
    int x ;
    std::function< int (void) > f()
    {
        return [this]{ return x ; } ;
    }
} ;

int main()
{
    std::function< int (void) > f ;
    {
        C c ;
        f = c.f() ;
    } // cの寿命はすでに終わっている

    f() ; // エラー
}
```

データメンバーをコピー・キャプチャーする方法はない。そもそも、何度も述べているように、データメンバーはキャプチャーできない。では、上の例で、どうしてもデータメンバーの値を使いたい場合はどうすればいいのか。この場合、一度ローカル変数にコピーするという方法がある。

```
struct C
{
    int x ;
    std::function< int (void) > f()
    {
        int x = this->x ;
        return [x]{ return x ; } ; // xはローカル変数のコピー
    }
} ;
```

もちろん、同じ名前にするのが紛らわしければ、名前を変えてもいい。

ラムダ式でデータメンバーを使う際には、キャプチャーしているのは、実はthisなのだとということに注意しなければならない。

5.4.4 パラメーターパックのキャプチャー

可変引数テンプレートの関数パラメーターパックも、キャプチャーリストに書くことができる。その場合、通常と同じように、パック展開になる。

```
template < typename ... Types > void g( Types ... args ) ;

template < typename ... Types >
void f( Types ... args )
{
    // 明示的なキャプチャー
    [args...]{ g( args... ) ; } ;
    [&args...]{ g( args... ) ; } ;

    // 暗黙的なキャプチャー
    [=]{ g( args... ) ; } ;
    [&]{ g( args... ) ; } ;
}
```

5.5 後置式(Postfix expressions)

後置式は、主にオペランドの後ろに演算子を書くことから、そう呼ばれている。後置式の評価順序はすべて、「左から右」である。

5.5.1 添字(Subscripting)

```
式 [ 式 ]
式 [ 初期化リスト ]
```

operator []は、添字と呼ばれる式である。これは、配列の要素にアクセスするために用いられる。どちらか片方の式は、Tへのポインター型でなければならず、もう片方は、unscoped enumか、整数型でなければならない。式の結果は、lvalueのTとなる。式、E1[E2] は、*(E1)+(E2)) と書くのに等しい。

```
int x[3] ;
// *(x + 1)と同じ
x[1] ;
```

この場合、xには、4.2 配列からポインターへの型変換が適用されている。

「どちらか片方の式」というのは、文字通り、どちらか片方である。たとえば、x[1]とすべきところを、1[x]としても、同じ意味になる。

```
int x[3] ;
// どちらも同じ意味。
x[1] ;
1[x] ;
```

もっとも、通常は、一つめの式をポインター型にして、二つ目の式を整数型にする。ユーザー定義のoperator []では、このようなことはできない。

ユーザー定義のoperator []の場合、[]の中の式に、初期化リストを渡すことができる。これは、どのように使ってもいいが、例えば以下のように使える。

```
struct C
{
    int data[10][10][10] ;
    int & operator []( std::initializer_list<std::size_t>
```

```

list )
{
    if ( list.size() != 3 ) { /* エラー処理 */ }

    auto iter = list.begin() ;
    std::size_t const i = *iter ; ++iter ;
    std::size_t const j = *iter ; ++iter ;
    std::size_t const k = *iter ;

    return data[i][j][k] ;
}
} ;

int main()
{
    C c ;
    c[{1, 2, 3}] = 0 ;
}

```

初期化リストを使えば、オーバーロードされたoperator []に、複数の引数を渡すことができる。

5.5.2 関数呼び出し(Function call)

関数呼び出し(function call)の文法は、以下の通り。

式（引数のリスト）

関数呼び出しには、通常の関数呼び出しと、メンバー関数呼び出しがある。

通常の関数を呼び出す場合、式には、関数へのlvalueか、関数へのポインターが使える。

```

void f( void ) { }
void g( int ) { }
void h( int, int, int ) { }

int main()
{

```

```

// 「関数へのlvalue」への関数呼び出し
f( ) ;
g( 0 ) ;
h( 0, 0, 0 ) ;

// 関数への参照
void (&ref)(void) = f ;

// 「関数へのlvalue」への関数呼び出し
ref() ;

// 関数ポインター
void (*ptr)(void) = &f ;

// 関数ポインターへの関数呼び出し
ptr() ;
}

```

staticなメンバー関数は、通常の関数呼び出しになる。

```

struct C { static void f(void) {} } ;

int main()
{
    void (*ptr)(void) = &C::f ;
    ptr() ; // 通常の関数呼び出し
}

```

メンバー関数を呼び出す場合、式には、関数のメンバーの名前か、メンバー関数へのポインター式が使える。

```

struct C
{
    void f(void) {}

    void g(void)
    {
        // メンバー関数の呼び出し
        f() ;
        this->f() ;
        (*this).f() ;
    }
}

```

```

    // メンバー関数へのポインター
    void (C::* ptr)(void) = &C::f ;
    // 関数呼び出し
    (this->*ptr)();
}
}

```

関数呼び出し式の結果の型は、式で呼び出した関数の戻り値の型になる。

```

void f() ;
int g() ;

// 式の結果の型はvoid
f() ;
// 式の結果の型はint
g() ;

```

関数呼び出しの結果の型は、戻り値の型になる。これはtypeidやsizeofやdecltypeのオペランドの中でも、同様である。

```

// 関数fの戻り値の型はint
// すなわち、fを関数呼び出しした結果の型はint
int f() { return 0 ; }

int main()
{
    // sizeof(int)と同じ
    sizeof( f() ) ;
    // typeid(int)と同じ
    typeid( f() ) ;
    // int型の変数xの宣言と定義。
    decltype( f() ) x ;
}

```

関数が呼ばれた際、仮引数は対応する実引数で初期化される。非staticメンバー関数の場合、this仮引数もメンバー関数を呼び出した際のオブジェクトへのポインターで初期化される。

仮引数に対して、具体的な一時オブジェクトが生成されるかどうかは、実装依存である。たとえば、実装は最適化のために、一時オブジェクトの生成を省略するかもしれない。

仮引数が参照の場合をのぞいて、呼ばれた関数の中で仮引数を変更しても、実引数は変更されない。ただし、型がポインターの場合、参照を通して参照先のオブジェクトが変更される可能性がある。

```
void f( int x, int & ref, int * ptr )
{
    x = 1 ; // 実引数は変更されない
    ref = 1 ; // 実引数が変更される
    *ptr = 1 ; // 実引数は、ポインターの参照を通して変更される
    ptr = nullptr ; // 実引数のポインターは変更されない
}

int main()
{
    int x = 0 ; // 実引数
    int * ptr = &x ;
    f( x, x, ptr ) ;
}
```

実引数の式が、どのような順番で評価されるかは決められていない。ただし、呼び出された関数の本体に入る際には、式はすべて評価されている。

```
#include <iostream>

int f1(){ std::cout << "f1" << std::endl ; return 0 ; }
int f2(){ std::cout << "f2" << std::endl ; return 0 ; }
int f3(){ std::cout << "f3" << std::endl ; return 0 ; }

void g( int, int, int ){ }

int main( )
{
    g( f1(), f2(), f3() ) ; // f1, f2, f3関数呼び出しの順番は分
    らない
}
```

この例では、関数f1, f2, f3がどの順番で呼ばれるのかが分からない。したがって、標準出力にどのような順番で文字列が出力されるかも分からない。ただし、関数gの本体に入る際には、f1, f2, f3は、すべて呼び出されている。

関数は、自分自身を呼び出すことができる。これを再帰呼び出しという。

```
void f()
{
    f() ; // 自分自身を呼び出す、無限ループ
}
```

ただし、main関数だけは特別で、再帰呼び出しをすることができない。

```
int main()
{
    main() ; // エラー
}
```

5.5.3 関数形式の明示的型変換(Explicit type conversion (functional notation))

型名 (式リスト)
型名 初期化リスト

関数形式の明示的型変換(Explicit type conversion (functional notation))とは、関数呼び出しのような文法による、一種のキャストである。

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

int main()
{
    int() ;
    int{} ;

    S( 0 ) ;
    S( 1, 2 ) ;
}
```

型名として使えるのは、7.3.6.2 [単純型指定子](#)か、`typename`指定子である。単純型指定子でなければならないということには、注意しなければならない。たとえば、ポインター やリファレンス、配列などを直接書くことはできない。ただし、`typedef`名は使える。

```
int x = 0 ;
// これらはエラー
int *(&x) ;
int &(x) ;

// typedef名は使える
using type = int * ;
type(&x) ;
```

単純型指定子の中でも、`auto`と`decltype`は、注意が必要である。まず、`auto`は使えない。

```
auto(0) ; // エラー
```

`decltype`は使える。ただし、非常に使いづらいので、使うべきではない。

```
// int型をint型にキャスト
// int(0)と同じ
decltype(0)(0) ;
```

たとえば、以下のコードはエラーである。

```
int x ;
decltype(x)(x) ; // エラー
```

これは、文法が曖昧だからだ。詳しくは、6.7 [曖昧解決](#)を参照。何が起こっているかというと、`decltype(x)(x)`は、キャストではなく、変数の宣言だとみなされている。`decltype(x)`は、`int`という型である。

```
// decltype(x)(x)と同じ
// decltype(x)(x) → int (x) → int x
int x ;
```

このため、`decltype`を関数形式のキャストで使うのは、問題が多い。使うならば、`typedef`名をつけてから使うか、`static_cast`を使うべきである。

```
int x ;
using type = decltype(x) ;
type(x) ;

static_cast< decltype(x) >(x) ;
```

`typename`指定子も使うことができる。

```
template < typename T >
void f()
{
    typename T::type() ; // OK
}
```

式リストが、たったひとつの式である場合、5.7 キャストと同じ意味になる。

```
// int型からshort型へのキャスト
short(0) ;
// int型からdouble型へのキャスト
double(0) ;

struct C { C(int) {} } ;
// 変換関数による、int型からC型へのキャスト
C(0) ;
```

型名がクラス名である場合、`T(x1, x2, x3)`という形の式は、`T t(x1, x2, x3)`という形と同じ意味を持つ一時オブジェクトを生成し、その一時オブジェクトを、prvalueの結果として返す。型名がクラス名でも、式リストがひとつしかない場合は、キャストである。もっとも、その場合も、ユーザー定義のコンストラクターが、変換関数として呼び出されることになるので、意味はあまり変わらない。

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

```
int main()
{
    C(0) ; // これはキャスト、意味としては、あまり違いはない
    C(1, 2) ;
    C(1, 2, 3) ;
}
```

式リストが空の場合、つまり、`T()`という形の式の場合。まず、`T`は配列型であってはならない。`T`は完全な型か、`void`でなければならない。式の結果は、値初期化された型のprvalueの値になる。値初期化については、8.5 [初期化子](#)を参照。

```
int() ; // int型の0で初期化された値
double() ; // double型の0で初期化された値

struct C {} ;
C() ; // デフォルトコンストラクターが呼ばれたCの値
```

`void`型の場合、値初期化はされない。式の結果の型は`void`である。

```
void() ; // 結果はvoid
```

括弧で囲まれた式リストではなく、初期化リストの場合、式の結果は、指定された型の、初期化リストによって直接リスト初期化されたprvalueの一時オブジェクトになる。

```
#include <initializer_list>

struct C
{
    C( std::initializer_list<int> ) { }
} ;

int main()
{
    C{1,2,3} ;
}
```

5.5.4 疑似デストラクター呼び出し(Pseudo destructor call)

疑似デストラクター呼び出しとは、デストラクターを明示的に呼び出すことができる一連の式である。使い方は、operator .. operator ->に続けて、疑似デストラクターネームを書き、さらに関数呼び出しのoperator ()を書く。この一連の式を、疑似デストラクター呼び出しという。このような疑似デストラクターネームに続けては、関数呼び出し式しか適用することができない。式の結果はvoidになる。

```
// このコードは、疑似デストラクター呼び出しの文法を示すためだけの
// 例である
struct C {} ;

int main()
{
    C c ;
    c.~C() ; // 疑似デストラクター呼び出し
    C * ptr = &c
    ptr->~C() ; // 疑似デストラクター呼び出し
}
```

注意すべきことは、デストラクターを明示的に呼び出したとしても、暗黙的に呼び出されるデストラクターは、依然として呼び出されるということである。

```
#include <iostream>

struct C
{
    ~C() { std::cout << "destructed." << std::endl ; }
} ;

int main()
{
    {
        C c ;
        c.~C() ; // デストラクターを呼び出す
    } // ブロックスコープの終りでも、デストラクターは暗黙的に呼ばれる

    C * ptr = new C ;
    ptr->~C() ; // デストラクターを呼び出す

    delete ptr ; // デストラクターが暗黙的に呼ばれる。
```

}

このように、通常は、デストラクターの呼び出しが重複してしまう。二重にデストラクターを呼び出すのは、大抵の場合、プログラム上のエラーである。では、疑似デストラクタ一呼び出しは何のためにあるのか。具体的な用途としては、placement newと組み合わせて使うことがある。

```
struct C { } ;

int main()
{
    // placement new用のストレージを確保
    void * storage = operator new( sizeof(C) ) ;
    // placement new
    C * ptr = new(storage) C ;
    // デストラクターを呼び出す
    ptr->~C() ;
    // placement new用のストレージを解放
    operator delete( storage ) ;
}
```

この疑似デストラクターには、decltypeを使うことができる。

```
struct C {} ;

int main()
{
    C c ;
    c.~decltype(c) ;
    C * ptr = &c
    ptr->~decltype(c) ;
}
```

テンプレート引数の場合、型がスカラー型であっても、疑似デストラクター呼び出しができる。

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

```
t.~T() ;  
}  
  
int main()  
{  
    f<int>();  
}
```

これにより、ジェネリックなテンプレートコードが書きやすくなる。

5.5.5 クラスメンバーアクセス (Class member access)

クラスのオブジェクト . メンバーネーム
クラスのポインター -> メンバーネーム

クラスメンバーアクセスは、名前の通り、クラスのオブジェクトか、クラスのオブジェクトへのポインターのメンバーにアクセスするための演算子である。

. 演算子の左側の式は、クラスのオブジェクトでなければならない。-> 演算子の左側の式は、クラスのオブジェクトへのポインターでなければならない。演算子の右側は、そのクラスか、基本クラスのメンバーネームでなければならない。-> 演算子を使った式、E1->E2は、(*E1).E2という式とおなじになる。

```
struct Object  
{  
    int x ;  
    static int y ;  
    void f() {}  
};  
  
int Object::y ;  
  
int main()  
{  
    Object obj ;  
    // . 演算子  
    obj.x = 0 ;  
    obj.y = 0 ;  
    obj.f() ;
```

```

Object * ptr = &obj ;
// -> 演算子
ptr->x = 0 ;
ptr->y = 0 ;
ptr->f() ;
}

```

もし、クラスのオブジェクト、クラスのオブジェクトへのポインターを表す式が依存式であり、メンバーメンバーテンプレートであり、テンプレート引数を明示的に指定したい場合、メンバーメンバーテンプレートを使わなければならない。

```

struct Object
{
    template < typename T >
    void f() {}
} ;

template < typename T >
void f()
{
    T obj ;
    obj.f<int>() ; // エラー
    obj.template f<int>() ; // OK
}

int main()
{
    f<Object>() ;
}

```

これは、<演算子や、>演算子と、文法が曖昧になるためである。この問題については、14.2 テンプレート特殊化の名前でも、解説している。

派生によって、クラスのメンバーメンバーテンプレートが曖昧な場合、エラーになる。

```

struct Base1 { int x ; } ;
struct Base2 { int x ; } ;

struct Derived : Base1, Base2
{ } ;

```

```
int main()
{
    Derived d ;

    d.x ; // エラー
    d.Base1::x ; // OK
    d.Base2::x ; // OK
}
```

5.5.6 インクリメントとデクリメント(Increment and decrement)

ここでは、後置式のインクリメントとデクリメントについて解説する。前置式のインクリメントとデクリメントについては、5.6.2 単項式のインクリメントとデクリメントを参照。

式 ++
式 --

後置式の++演算子の式の結果は、オペラントの式の値になる。オペラントは、変更可能なlvalueでなければならない。オペラントの型は、数値型か、ポインター型でなければならない。式が評価されると、オペラントに1を加算する。ただし、式の結果は、オペラントに1を加算する前の値である。

```
int x = 0 ;
int result = x++ ;

// ここで、result == 0, x == 1
```

式の結果の値は、オペラントの値と変わりがないが、オペラントには、1を加算されるということに注意しなければならない。

後置式の--演算子は、オペラントから1を減算する。それ以外は、++演算子と全く同じように動く。

```
int x = 0 ;
int result = x-- ;

// ここで、result == 0, x == -1
```

5.5.7 Dynamic cast(Dynamic cast)

```
dynamic_cast < 型名 > ( 式 )
```

`dynamic_cast<T>(v)`という式は、`v`という式を`T`という型に変換する。便宜上、`v`を`dynamic_cast`のオペランド、`T`を`dynamic_cast`の変換先の型とする。変換先の型はクラスへのポインターかリファレンス、あるいは、`void`へのポインター型でなければならぬ。オペランドは、変換先の型が、ポインターの場合はポインター、リファレンスの場合はリファレンスでなければならない。

```
struct C {} ;

int main()
{
    C c ;

    // 変換先の型がポインターの場合は、オペランドもポインター
    // 変換先の型がリファレンスの場合は、オペランドもリファレンス
    // でなければならない
    dynamic_cast<C &>(c) ; // OK
    dynamic_cast<C *>(&c) ; // OK

    // ポインターかリファレンスかが、一致していない
    dynamic_cast<C *>(c) ; // エラー
    dynamic_cast<C &>(&c) ; // エラー
}
```

dynamic_castの機能

今、`Derived`クラスが、`Base`クラスから派生されていたとする。

```
struct Base {} ;
struct Derived : Base {} ;
```

この時、`static_cast`を使えば、`Base`へのポインターやリファレンスから、`Derived`へのポインターやリファレンスに変換することができる。

```

int main()
{
    Derived d ;

    Base & base_ref = d ;
    Derived & derived_ref = static_cast<Derived &>
(base_ref) ;

    Base * base_ptr = &d ;
    Derived * derived_ptr = static_cast<Derived *>
(base_ptr) ;
}

```

この例では、ポインターやリファレンスが指す、本当のオブジェクトは、Derivedクラスのオブジェクトだということが分かりきっているので安全である。しかし、ポインターやリファレンスを使う場合、常にオブジェクトの本当のクラス型が分かるわけではない。

```

void f( Base & base )
{
    // baseがDerivedを参照しているかどうかは、分からぬ。
    Derived & d = static_cast<Derived &>(base) ;
}

int main()
{
    Derived derived ;
    f(derived) ; // ok

    Base base ;
    f(base) ; // エラー
}

```

このように、ポインターやリファレンスの指し示すオブジェクトの本当のクラス型は、実行時にしか分からない。しかし、オブジェクトの型によって、特別な処理をしたいことも、よくある。

```

void f( Base & base )
{
    if ( /* baseの指すオブジェクトがDerivedクラスの場合 */ )
    {
        // 特別な処理
    }
}

```

```

    }

    // 共通の処理
}

```

本来、このような処理は、virtual関数で行うべきである。しかし、現実には、どうしても、このような泥臭くて汚いコードを書かなければならない場合もある。そのようなどうしようもない場合のために、C++には、基本クラスへのポインターやりファレンスが、実は派生クラスのオブジェクトを参照している場合に限り、キャストできるという機能が提供されている。それが、dynamic_castである。

動的な型チェックを使うためには、dynamic_castのオペランドのクラスは、ポリモーフィック型でなければならない。つまり、少なくともひとつのvirtual関数を持っていなければならない。ポリモーフィック型の詳しい定義については、10.3 [virtual関数](#)を参照。

もし、オペランドの参照するオブジェクトが、変換先の型として指定されている派生クラスのオブジェクトであった場合、変換することができる。

```

struct Base { virtual void f() {} } ;
struct Derived : Base {} ;

void f(Base & base)
{ // baseはDerivedを指しているとする
    Derived & ref = dynamic_cast<Derived &>(base) ;
    Derived * ptr = dynamic_cast<Derived *>(&base) ;
}

```

実引数に、変換先の型ではないオブジェクトを渡した場合、dynamic_castの変換は失敗する。変換が失敗した場合、変換先の型がリファレンスの場合、std::bad_castがthrowされる。変換先の型がポインターの場合、nullポインターが返される。

```

struct Base { virtual void f() {} } ;
struct Derived : Base {} ;

int main()
{
    Base base ;

    // リファレンスの場合
    try
    {
        Derived & ref = dynamic_cast<Derived &>(base) ;
    }

```

```

catch ( std::bad_cast )
{
    // 変換失敗
    // リファレンスの場合、std::bad_castがthrowされる
}

// ポインターの場合
Derived * ptr = dynamic_cast<Derived *>(&base) ;

if ( ptr == nullptr )
{
    // 変換失敗
    // ポインターの場合、nullptrポインターが返される
}
}

```

基本クラスのポインター やリファレンスが、実際は何を指しているかは、実行時にしか分からない。そのため、常に変換に失敗する可能性がある。そのため、dynamic_castを使う場合は、常に変換が失敗するかもしれないという前提のもとに、コードを書かなければならぬ。

失敗せずに変換できる場合というのは、オペランドの指すオブジェクトの本当の型が、変換先の型のオブジェクトである場合で、しかもアクセスできる場合である。オブジェクトである(is a)場合というのは、例えば、

```

struct A { virtual void f(){} } ;
struct B : A {} ;
struct C : B {} ;
struct D : C {} ;

```

このようなクラスがあった場合、Dは、Cであり、Bであり、Aである。従って、Dのオブジェクトを、Aへのリファレンスで保持していた場合、D、C、Bのいずれにも変換できる。

```

int main()
{
    D d ;
    A & ref = d ;

    // OK
    // refの指しているオブジェクトは、Dなので、変換できる。
    dynamic_cast<D &>(ref) ;
    dynamic_cast<C &>(ref) ;
}

```

```
    dynamic_cast<B &>(ref) ;
}
```

アクセスできる場合というのは、変換先の型から、publicで派生している場合である。

```
struct Base1 { virtual void f(){} } ;
struct Base2 { virtual void g(){} } ;
struct Base3 { virtual void h(){} } ;

struct Derived
: public Base1,
  public Base2,
  private Base3
{ } ;

int main()
{
    Derived d ;
    Base1 & ref = d ;

    // OK、Base2はpublicなので、アクセス出来る
    dynamic_cast<Base2 &>(ref) ;
    // 実行時エラー、Base3はprivateなので、アクセス出来ない
    // std::bad_castがthrowされる。
    dynamic_cast<Base3 &>(ref) ;
}
```

この例の場合、refが参照するオブジェクトは、Derived型であるので、Base3型のサブオブジェクトも持っているが、Base3からは、privateで派生されているために、アクセスすることはできない。そのため、変換することが出来ず、std::bad_castがthrowされる。

変換先の型は、void型へのポインターとすることもできる。その場合、オペランドの指す本当のオブジェクトの、もっとも派生されたクラスを指すポインターが、voidへのポインター型として、返される。

```
struct Base { virtual void f(){} } ;
struct Derived1 : Base {} ;
struct Derived2 : Derived1 {} ;

int main()
{
    Derived1 d1 ;
```

```

Base * d1_ptr = &d1 ;

// Derived1を指すポインターの値が、void *として返される
void * void_ptr1 = dynamic_cast<void *>(d1_ptr) ;

Derived1 d2 ;
Base * d2_ptr = &d2;

// Derived2を指すポインターの値が、void *として返される
void * void_ptr2 = dynamic_cast<void *>(d2_ptr) ;
}

```

一般に、この機能はあまり使われることがないだろう。

dynamic_castのその他の機能

dynamic_castは、その主目的の機能の他にも、クラスへのポインターやリファレンスに限って、キャストを行うことができる。この機能は、4.11 標準型変換のポインターの型変換に、ほぼ似ている。このキャストは、static_castでも行える。以下の機能に関しては、実行時のコストは発生しない。

オペランドの型が、変換先の型と同じ場合、式の結果の型は、変換先の型になる。この時、constとvolatileを付け加えることはできるが、消し去ることは出来ない。

```

// 型と式が同じ場合の例
struct C { } ;

int main()
{
    C v ;

    dynamic_cast<C &>(v) ;
    dynamic_cast<C const &>(v) ; // constを付け加える

    C const cv ;
    dynamic_cast<C &>(cv) ; // エラー、constを消し去ることは出来ない

    // ポインターの場合
    C * ptr = &v ;
    dynamic_cast<C *>(ptr) ;

```

```

    dynamic_cast<C const *>(ptr) ;
}

```

変換先の型が基本クラスへのリファレンスで、オペランドの型が、派生クラスへのリファレンスの場合、`dynamic_cast`の結果は、派生クラスのうちの基本クラスを指すリファレンスになる。ポインターの場合も、同様である。

```

struct Base {} ; // 基本クラス
struct Derived : Base {} ; // 派生クラス

int main()
{
    Derived d ;

    Base & base_ref = dynamic_cast<Base &>(d) ;
    Base * base_ptr = dynamic_cast<Base *>(&d) ;
}

```

5.5.8 型識別 (Type identification)

```

typeid ( 式 )
typeid ( 型名 )

```

`typeid`とは、式や型名の、型情報を得るための式である。型情報は、`const std::type_info`のリファレンスという形で返される。`std::type_info`についての詳細は、[RTTI \(Run Time Type Information\)](#)を参照。`typeid`を使うには、必ず、`<typeinfo>`ヘッダーを#includeしなければならない。ただし、本書のサンプルコードは、紙面の都合上、必要なヘッダーのincludeを省略していることがある。

`typeid`のオペランドは、`sizeof`に似ていて、式と型名の両方を取ることができる。

```

#include <typeinfo>

int main()
{
    // 型名の例
    typeid(int) ;
    typeid( int * ) ;
}

```

```
// 式の例
int x = 0 ;
typeid(x) ;
typeid(&x) ;
typeid( x + x ) ;

}
```

typeidのオペランドの式が、ポリモーフィッククラス型のglvalueであった場合、実行時チェックが働き、結果のstd::type_infoが表す型情報は、実行時に決定される。型情報は、オブジェクトの最も派生したクラスの型となる。

```
struct Base { virtual void f() {} } ;
struct Derived : Base {} ;

int main()
{
    Derived d ;
    Base & ref = d ;

    // オブジェクトの、実行時の本当の型を表すtype_infoが返される
    std::type_info const & ti = typeid(d) ;

    // true
    ti == typeid(Derived) ;

    // Derivedを表す、人間の読める実装依存の文字列
    std::cout << ti.name() << std::endl ;
}
```

オペランドの式の型がポリモーフィッククラス型のglvalueの場合で、nullポインターを参照した場合は、std::bad_typeidが投げられる。

```
struct Base { virtual void f() {} } ;
struct Derived : Base {} ;

int main()
{
    // ptrの値はnullポインター
    Base * ptr = nullptr ;
```

```

try
{
    typeid( *ptr ) ; // 実行時エラー
}
catch ( std::bad_typeid )
{
    // 例外が投げられて、ここに処理が移る
}
}

```

オペランドの式の型が、ポリモーフィッククラス型でない場合は、`std::type_info`が表す型情報は、コンパイル時に決定される。

```

// int型を表すtype_info
typeid(0) ;
// double型を表すtype_info
typeid(0.0) ;

int f() {}
// int型を表すtype_info
typeid( f() ) ;

```

この際、4.1 [lvalueからrvalueへの型変換](#)、4.2 [配列からポインターへの型変換](#)、4.3 [関数からポインターへの型変換](#)は行われない。

```

// 配列からポインターへの型変換は行われない
int a[10] ;

// 型情報は、int [10]
// int *ではない
typeid(a) ;

// 関数からポインターへの型変換は行われない
void f() {}

// 型情報は、void (void)
// void (*) (void)ではない。
typeid(f) ;

```

これらの標準型変換は、C++では、非常に多くの場所で、暗黙のうちに行われているので、あまり意識しない。たとえば、テンプレートの実引数を推定する上では、これらの変

換が行われる。

```
// 実引数の型を、表示してくれるはずの便利な関数
template < typename T >
void print_type_info(T)
{
    std::cout << typeid(T).name() << std::endl ;
}

void f() { }

int main()
{
    int a[10] ;
    // int [10]
    std::cout << typeid(a).name() << std::endl ;
    // int *
    print_type_info(a) ;

    // void (void)
    std::cout << typeid(f).name() << std::endl ;
    // void (*) (void)
    print_type_info(f) ;
}
```

`std::type_info::name()`の返す文字列は実装依存だが、今、C++の文法と同じように型を表示すると仮定すると、このような出力になる。C++では、多くの場面で、暗黙のうちに、これら三つの型変換が行われるので、このような差異が生じる。

オペランドが、型名の場合は、`std::type_info`は、その型を表す。ほんの一例をあげると。

```
int main()
{
    typeid( int ) ;           // int型
    typeid( int * ) ;         // intへのポインター型
    typeid( int & ) ;         // intへのlvalueリファレンス型
    typeid( int [2] ) ;       // 配列型
    typeid( int (* )[2] ) ;   // 配列へのポインター型
    typeid( int (int) ) ;     // 関数型
    typeid( int (*)(int) ) ;  // 関数へのポインター型
}
```

オペランドの式や型名の、トップレベル(top-level)のCV修飾子は、無視される。

```
int main()
{
    // トップレベルのCV修飾子は無視される
    typeid(const int) ; // int
    // 当然、型情報は等しい
    typeid(const int) == typeid(int) ; // true

    // 型名も式も同じ
    int i = 0 ; const int ci = 0;
    typeid(ci) ; // int
    typeid(ci) == typeid(i) ; // true

    // これはトップレベルのCV修飾子
    typeid(int * const) ; // int *

    // 以下はトップレベルのCV修飾子ではない
    typeid(const int *) ; // int const *
    typeid(int const *) ; // int const *
}
```

5.5.9 Static cast(Static cast)

`static_cast< 型名 >(式)`

`static_cast`は、実に多くの静的な変換ができる。その概要は、標準型変換とその逆変換、ユーザー定義の変換、リファレンスやポインターにおける変換など、比較的安全なキャストである。以下に`static_cast`の行える変換を列挙するが、これらを丸暗記しておく必要はない。もし、どのキャストを使うか迷った場合は、`static_cast`を使っておけば、まず間違いはない。`static_cast`がコンパイルエラーとなるキャストは、大抵、実装依存で危険なキャストである。

`static_cast`によるキャストがどのように行われるかは、おおむね、以下の順序で判定される。条件に合う変換方法が見つかった時点で、それより先に行くことはない。これは、完全な`static_cast`の定義ではない。分かりやすさのため省いた挙動もある。

`static_cast<T>(v)`の結果は、オペランドvを変換先の型Tに変換したものとなる。変換先の型がlvalueリファレンスならば結果はlvalue、rvalueリファレンスならば結果はxvalue、それ以外の結果はprvalueとなる。`static_cast`は、`const`と`volatile`を消し去ることはでき

ない。

オペランドの型が基本クラスで、変換先の型が派生クラスへのリファレンスの場合。もし、標準型変換で、派生クラスのポインターから、基本クラスのポインターへと変換できる場合、キャストできる。

```
struct Base {} ;
struct Derived : Base {} ;

void f(Base & base)
{
    // Derived *からBase *に標準型変換で変換できるので、キャスト
    // できる
    Derived & derived = static_cast<Derived &>(base) ;
}
```

ただし、これには実行時チェックがないので、baseが本当にDerivedのオブジェクトを参照していなかった場合、動作は未定義である。

lvalueのオペランドは、rvalueリファレンスに型変換できる。

```
int main()
{
    int x = 0 ;
    static_cast<int &&>(x) ;
}
```

もし、`static_cast<T>(e)` という式で、`T t(e);` という宣言ができる場合、オペランドの式`e`は、変換先の型`T`に変換できる。その場合、一時オブジェクトを宣言して、それをそのまま使うのと、同じ意味になる。

```
// short t(0) は可能なので変換できる
static_cast<short>(0) ;

// float t(0) は可能なので変換できる
static_cast<float>(0) ;
```

4 標準型変換の逆変換を行うことができる。ただし、いくつかの変換は、逆変換を行えない。これには、4.1 lvalueからrvalueへの型変換、4.2 配列からポインターへの型変換、4.3 関数からポインターへの型変換(Function-to-pointer conversion)、ポインターや関数ポインターからnullポインターへの変換、がある。

変換先の型に、voidを指定することができる。その場合、static_castの結果はvoidである。オペランドの式は評価される。

```
int main()
{
    static_cast<void>( 0 ) ;
    static_cast<void>( 1 + 1 ) ;
}
```

整数型とscoped enum型は、static_castを使うことで、明示的に変換することができる。その場合、変換先の型で、オペランドの値を表現できる場合は、値が保持される。値を表現できない場合の挙動は、規定されていない。

```
int main()
{
    enum struct A { value = 1 } ;
    enum struct B { value = 1 } ;

    int x = static_cast<int>( A::value ) ;
    A a = static_cast<A>(1) ;
    B b = static_cast<B>( A::value ) ;
}
```

派生クラスへのポインターから、基本クラスへのポインターにキャストできる。

```
struct Base {} ;
struct Derived : Base {} ;

Derived d ;
Base * ptr = static_cast<Base *>(&d) ;
```

voidへのポインターは、他の型へのポインターに変換できる。ある型へのポインターから、voidへのポインターにキャストされ、そのまま、ある型へのポインターにキャストされなおされた場合、その値は保持される。

```
int main()
{
    int x = 0 ;
    int * ptr = &x ;
```

```

// void *への変換は、標準型変換で行えるので、キャストはなくて
もよい。
void * void_ptr = static_cast<void *>(ptr) ;

// キャストが必要
ptr = static_cast<int *>(void_ptr) ;

*ptr ; // ポインターの値は保持されるので、xを正しく参照する
}

```

5.5.10 Reinterpret cast

`reinterpret_cast < 型名 > (式)`

`reinterpret_cast`は、式の値をそのまま、他の型に変換するキャストである。ポインターと整数の間の変換や、ある型へのポインターを全く別の型へのポインターに変換するといったことができる。`reinterpret_cast`を使えば、値をそのままにして、型を変換することができる。変換した結果、その値が、変換先の型としてそのまま使えるかどうかなどといったことは、ほとんど規定されていない。元の値をそのまま保持できるかどうかも分からぬ。それ故、`reinterpret_cast`は、危険なキャストである。

多くの実装では、`reinterpret_cast`には、何らかの具体的で実用的な意味がある。現実のC++が必要とされる環境では、`reinterpret_cast`を使わなければならないことも、多くある。しかし、`reinterpret_cast`を使った時点で、そのコードは実装依存であり、具体的に意味が定義されたその環境でしか動かないということを、常に意識するべきである。

`reinterpret_cast`では、`const`や`volatile`を消し去ることはできない。

`reinterpret_cast`ができる変換を、以下に列挙する。

ポインター型と整数型の間の型変換

ある型へのポインター型から整数型へのキャスト、あるいはその逆に、整数型や`enum`型からポインター型へのキャストを行える。整数型は、ポインターの値をそのまま保持できるほど大きくなければならない。どの整数型ならば十分に大きいのか。もし整数型が十分に大きくなればどうなるのかなどということは、定義されていない。

`int main()`

```
{
    int x = 0 ;
    int * ptr = &x ;

    // ポインターから整数へのキャスト
    int value = reinterpret_cast<int>(ptr) ;

    // 整数からポインターへのキャスト
    ptr = reinterpret_cast<int *>(value) ;
}
```

これらのキャストについての挙動は、ほとんどが実装依存であり、あまり説明できることはない。

もし、変換先の整数型が、ポインターの値をすべて表現できるとするならば、再びポインター型にキャストし直した時、ポインターは同じ値を保持すると規定されている。しかし、int型がポインターの値をすべて表現できるという保証はない。
unsigned intであろうと、long intであろうとlong long intであろうと、そのような保証はない。従って、上記のコードで、ptrが同じ値を保つかどうかは、実装依存である。

異なるポインター型の間の型変換

ある型へのポインターは、まったく別の型へのポインターに変換できる。たとえば、int *からshort *などといった変換ができる。

```
int main()
{
    int x = 0 ;
    int * int_ptr = &x ;

    // int * からshort *へのキャスト
    short * short_ptr = reinterpret_cast<short *>
(int_ptr) ;
```

これについても、挙動は実装依存であり、特に説明できることはない。たとえば、上記のコードで、short_ptrを参照した場合どうなるのかということも、全く規定されていない。ある実装では、問題なく、int型のストレージを、あたかもshort型のストレージとして使うことができるかもしれない。ある実装では、参照した瞬間にプログラムがクラッシュするかもしれない。

異なるリファレンス型の間の型変換

異なるポインター型の間の型変換に似ているが、異なるリファレンス型の変換をすることができます。例えば、int &からshort &などといった変換ができる。

```
int main()
{
    int x = 0 ;

    // int & からshort &へのキャスト
    short & short_ref = reinterpret_cast<short &>(x) ;
}
```

異なるポインター型の間の型変換と同じで、これについても、具体的な意味は実装依存である。

異なるメンバーポインターの間の型変換

異なるメンバーポインターへ変換することができる。

```
struct A { int value ; } ;
struct B { int value ; } ;

int main()
{
    int B::* ptr = reinterpret_cast<int B::*>(&A::value)
;
```

意味は、実装依存である。

異なる関数ポインター型の間の型変換

ある関数ポインターは、別の型の関数ポインターにキャストできるかもしれない。意味は実装依存である。「かもしれない」というのは実に曖昧な表現だが、たとえ完全に規格準拠な実装であっても、この機能をサポートする義務がないという意味である。

```

void f(int) {}

int main()
{
    // void (short)な関数へのポインター型
    using type = void (*)(short) ;

    // 関数ポインターの型変換
    type ptr = reinterpret_cast<type>(&f) ;
}

```

この変換がどういう意味を持つのか。例えば、変換した結果の関数ポインターは、関数呼び出しできるのか。できるとして、一体どういう意味になるのか、などということは一切規定されていない。

reinterpret_castには、できないこと

reinterpret_castが行えるキャストは、上記にすべて列挙した。それ以外の変換は、reinterpret_castでは行うことができない。これは、そもそもreinterpret_castの目的が、危険で実装依存なキャストのみを分離するという目的にあるので、それ以外の変換は、あえて行えないようになっている。

```

int main()
{
    short value = 0 ;

    // OK、標準型変換による暗黙の型変換
    int a = value ;
    // OK、static_castによる明示的な型変換
    int b = static_cast<int>(value) ;

    // エラー
    // reinterpret_castでは、この型変換をサポートしていない
    int c = reinterpret_cast<int>(value) ;
}

```

5.5.11 Const cast

```
const_cast < 型名 > ( 式 )
```

const_castは、constとvolatileが異なる型の間の型変換を行う。constやvolatileを取り除くことや、付け加えることができる。

```
int main()
{
    int const x = 0 ;

    // エラー、constを取り除くことはできない。
    int * error1 = &x ;
    int * error2 = static_cast<int *>(&x) ;

    // OK、ポインターの例
    int * ptr = const_cast<int *>(&x) ;
    // OK、リファレンスの例
    int & ref = const_cast<int &>(x) ;

    // constを付け加えることもできる。
    int y = 0 ;
    int const * cptr = const_cast<int const *>(&y) ;
}
```

ポインターへのポインターであっても、それぞれのconstを取り除くことができる。

```
int main()
{
    int const * const * const * const c = nullptr ;
    int *** ptr = const_cast<int ***>(c) ;
}
```

const_castは、constやvolatileのみを取り除く、または付け加えるキャストのみを行える。それ以外の型変換を行うことはできない。

```
int main()
{
    int const x = 0 ;
    // エラー、const以外の型変換を行っている。
    short * ptr = const_cast<short *>(&x) ;
```

}

では、`const`を取り除くと同時に、他の型変換も行ないたい場合はどうするかというと、`static_cast`や、`reinterpret_cast`を併用する。

```
int main()
{
    int const x = 0 ;

    short * ptr1 =
        static_cast<short *>(
            static_cast<void *>(
                const_cast<int *>(&x)
            )
        ) ;

    short * ptr2 = reinterpret_cast<short *>(const_cast<int
*>(&x)) ;
}
```

`const_cast`は、基本的に、ほとんどの`const`をキャストすることができるが、キャストできない`const`も存在する。たとえば、関数ポインターやメンバー関数ポインターに関する`const`を取り除くことはできない。関数へのリファレンスも同様である。

```
void f( int ) {}

int main()
{
    using type = void (*)(int) ;
    type const ptr = nullptr ;

    // エラー、関数ポインターはキャストできない
    type p = const_cast<type>(ptr) ;
}
```

もちろん、関数の仮引数に対する`const`をキャストすることや、`const`なメンバー関数を非`const`なメンバー関数にキャストすることなどもできない。

5.6 単項式 (Unary expressions)

単項式は、オペランドをひとつしか取らないことより、そう呼ばれている。単項式の評価順序はすべて、「右から左」である。

5.6.1 単項演算子(Unary operators)

単項演算子というカテゴリーには、六つの異なる演算子がまとめられている。`*`、`&`、`+`、`-`、`!`、`~`である。

5.6.1.1 * 演算子と& 演算子

* 単項演算子は、参照(indirection)である。オペランドは、オブジェクトへのポインターでなければならない。オペランドの型が、「Tへのポインター」であるとすると、式の結果は、lvalueのTである。

& 演算子は、オペランドのポインターを得る。オペランドの型がTであるとすると、結果は、prvalueのTへのポインターである。& 演算子は、オブジェクトだけではなく、関数にも適用できる。

```
int main()
{
    int x = 0 ;

    // & 演算子
    // 変数xのオブジェクトへのポインターを得る。
    int * ptr = &x ;

    // * 演算子
    // ポインターを参照する
    *ptr ;
}
```

5.6.1.2 単項演算子の+と-

単項演算子の+と-は、オペランドの符号を指定する演算子である。

+ 単項演算子は、オペランドの値を、そのまま返す。オペランドの型には、数値型、非scoped enum型、ポインター型が使える。結果はprvalueである。

```
int main()
{
    int x = +0 ; // xは0
    +x ; // 結果は0

    int * ptr = nullptr ;
    +ptr ; // 結果はptrの値
}
```

ただし、オペランドには、整数のプロモーションが適用されるので、オペランドの型がcharやshort等の場合、int型になる。

```
short x = 0 ;
+x ; // int型の0
```

- 単項演算子は、オペランドの値を、負数にして返す。オペランドの型には、数値型と非scoped enum型が使える。+ 単項演算子と同じく、オペランドには整数のプロモーションが適用される。

```
-0 ; // 0
-1 ; // -1
- -1 ; // +1
```

- 単項演算子が、unsignedな整数型に使われた場合の挙動は、明確に定義されている。オペランドのunsignedな整数型のビット数をnとする。式の結果は、 2^n から、オペランドの値を引いた結果の値になる。

具体的な例を挙げるために、今、unsigned int型を16ビットだと仮定する。

```
// unsigned int型は16bitであるとする。
unsigned int x = 1 ;
// result =  $2^{16} - 1 = 65536 - 1 = 65535$ 
unsigned int result = -x ;

x = 100 ;
// result =  $2^{16} - 100 = 65536 - 100 = 65436$ 
result = -x ;
```

5.6.1.3 ! 演算子

! 演算子は、オペランドをboolに変換し、その否定を返す。つまり、オペランドがtrueの場合はfalseに、falseの場合はtrueになる。

```
!true ; // false
!false ; // true
```

```
int x = 0 ;
!x ; // 0はfalseに変換される。その否定なので、結果はtrue
```

5.6.1.4 ~ 演算子

~ 演算子は、ビット反転とも呼ばれている。オペランドには、整数型と非scoped enum型が使える。式の結果は、オペランドの1の補数となる。すなわち、オペランドの各ビットが反転された値となる。オペランドには整数のプロモーションが適用される。

```
int x = 0 ;
// ビット列の各ビットを反転する
~x ;
```

5.6.2 インクリメントとデクリメント(Increment and decrement)

++ 式
-- 式

ここでは、前置式のインクリメントとデクリメントについて解説する。5.5.6 後置式のインクリメントとデクリメントも参照。

前置式の++ 演算子は、オペランドに1を加算して、その結果をそのまま返す。オペランドは数値型かポインター型で、変更可能なlvalueでなければならない。式の結果はlvalueになる。

前置式の-- 演算子は、オペランドから1を減算する。それ以外は、++演算子と同じよう

に動く。

```
int x = 0 ;
int result = ++x ;
// ここで、result = 1, x = 1
```

5.6.3 sizeof(Sizeof)

```
sizeof ( 未評価式 )
sizeof ( 型名 )
sizeof ... ( 識別子 )
```

sizeofとは、オペランドを表現するオブジェクトのバイト数を返す演算子である。オペランドは、未評価式か型名になる。

オペランドに型名を指定した場合、sizeof演算子は、型のオブジェクトのバイト数を返す。sizeof(char)、sizeof(signed char)、sizeof(unsigned char)は、1を返す。それ以外のあらゆる型のサイズは、実装によって定義される。たとえば、sizeof(bool)やsizeof(char16_t)やsizeof(char32_t)のサイズも、規格では決められていない。

```
// 1
sizeof(char) ;
// int型のオブジェクトのサイズ
sizeof(int) ;
```

オペランドに式を指定した場合、その式の結果の型のオブジェクトのバイト数を返す。式は評価されない。[4.1 lvalueからrvalueへの型変換](#)、[4.2 配列からポインターへの型変換](#)、[4.3 関数からポインターへの型変換](#)は行われない。

```
int f() ;

// int型のオブジェクトのサイズ
sizeof( f() ) ;
// int型のオブジェクトのサイズ
sizeof( 1 + 1 ) ;
```

関数呼び出しの式の結果の型は、関数の戻り値の型になる。

オペランドには、関数と不完全型を使うことはできない。関数は、そもそもオブジェクトではないし、不完全型は、そのサイズを決定できないからだ。「関数」は使えないが、関数呼び出しが「関数」ではないので使える。また、関数ポインターにも使える。

```
int f() ;
struct Incomplete ;

// 関数呼び出しが「関数」ではない
// sizeof(int)と同じ
sizeof( f() ) ;
// 関数ポインターはオブジェクトであるので、使える
// sizeof( int (*)(void) )と同じ
sizeof( &f ) ;

// エラー、関数を使うことはできない
sizeof( f ) ;
// エラー、不完全型を使うことはできない
sizeof( Incomplete ) ;
```

オペランドがリファレンス型の場合、参照される型のオブジェクトのサイズになる。

```
void f( int & ref )
{
// sizeof(int)と同じ
sizeof( int & ) ;
sizeof( int && ) ;
sizeof( ref ) ;
}
```

オペランドがクラス型の場合、クラスのオブジェクトのバイト数になる。これには、アライメントの調整や、配列の要素として使えるようにするための実装依存のパディングなども含まれる。クラス型のサイズは、必ず1以上になる。これは、サイズが0では、ポインターの演算などに差し支えるからである。

オペランドが配列型の場合、配列のバイト数になる。これはつまり、要素の型のサイズ × 要素数となる。

```
// sizeof(int) * 10と同じ
sizeof( int [10] ) ;
```

```

char a[10] ;
// sizeof(char) * 10 = 1 * 10 = 10
sizeof( a ) ;

// この型は配列ではなく、char
// sizeof(char)と同じ
sizeof( a[0] ) ;

```

`sizeof...`は、オブジェクトのバイト数とは、何の関係もない。`sizeof...`のオペランドには、パラメーターパックの識別子を指定できる。`sizeof...`演算子は、オペランドのパラメーターパックの引数の数を返す。`sizeof...`演算子の結果は定数で、型は`std::size_t`である。

```

#include <cstddef>
#include <iostream>

template < typename... Types >
void f( Types... args )
{
    std::size_t const t = sizeof...(Types) ;
    std::size_t const a = sizeof...(args) ;

    std::cout << t << ", " << a << std::endl ;
}

int main()
{
    f() ; // 0, 0
    f(1,2,3) ; // 3, 3
    f(1,2,3,4,5) ; // 5, 5
}

```

5.6.4 new

確保関数と解放関数の具体的な実装方法については、[動的メモリー管理](#)を参照。

```

::opt new 型名 new初期化子 opt
::opt new ( 式リスト ) 型名 new初期化子 opt

```

`new`に必要な宣言の一部は、`<new>`ヘッダーで定義されているので、使う際は、これをincludeしなければならない。

`new`式は、型名のオブジェクトを生成する。`new`される型は、完全型でなければならぬ。ただし、抽象クラスは`new`できない。リファレンスはオブジェクトではないため、`new`できない。`new`式の結果は、型が配列以外の場合は、生成されたオブジェクトへのポインターを返す。型が配列の場合は、配列の先頭要素へのポインターを返す。

```
class C {} ;
int main()
{
    // int型のオブジェクトを生成する
    int * i = new int ;
    // C型のオブジェクトを生成する
    C * c = new C ;
}
```

`new`が、オブジェクトのためのストレージの確保に失敗した場合、`std::bad_alloc`例外がthrowされる。

```
int main()
{
    try
    {
        new int ;
    }
    catch ( std::bad_alloc )
    {
        // newが失敗した
    }
}
```

詳細なエラーについては、後述する。

`new`によって生成されるオブジェクトは、**動的ストレージの有効期間**を持つ。つまり、`new`によって作られたオブジェクトを破棄するためには、明示的に`delete`を使わなければならない。

```
int main()
{
    int * ptr = new int ; // 生成
    delete ptr ; // 破棄
```

}

new式の評価

new式に、new初期化子が指定されている場合、その式を評価する。次に、確保関数(allocation function)を呼び出して、オブジェクトの生成に必要なストレージを確保する。初期化を行ない、確保したストレージ上に、オブジェクトを構築する。そして、オブジェクトへのポインターを返す。

配列の生成

newで配列を生成する場合、要素数は、定数でなくても構わない。

```
void f( int n )
{
    // 5個のint型の配列を生成する
    // 要素数は定数
    new int[5] ;

    // n個のint型の配列を生成する
    // 要素数は定数ではない
    new int[n] ;
}
```

配列の配列、つまり多次元配列を生成する場合、配列型の最初に指定する要素数は、定数でなくても構わない。残りの要素数は、すべて定数でなければならぬ。

```
void f( int n )
{
    // 要素数はすべて定数
    new int[5][5][5] ;

    // OK
    // 最初の要素数は定数ではない
    // 残りはすべて定数
    new int[n][5][5] ;
```

```

new int [n] // 最初の要素数は定数でなくてもよい
            [5][5] ; // 残りの要素数は定数でなければならない

// エラー
// 最初以外の要素数が定数ではない
new int[n][n][n] ;
new int[5][n][n] ;
new int[5][n][5] ;
}

```

配列の要素数が0の場合、`new`は、0個の配列を生成する。配列の要素数が負数であった場合の挙動は未定義である。

```

int main()
{
    // OK
    int * ptr = new int[0] ;
    // もちろんdeleteしなければならない
    delete [] ptr ;

    // エラー
    new int[-1] ;
}

```

もし、配列型の定数ではない要素数が、実装の制限以上の大きさである場合、ストレージの確保は失敗する。その場合、`std::bad_array_new_length`例外がthrowされる。要素数が定数であった場合は、通常通り、`std::bad_alloc`例外がthrowされる。

```

// int[n]のストレージを確保できないとする。
int main()
{
    try
    {
        std::size_t n =
std::numeric_limits<std::size_t>::max() ;
        new int[n] ; // 要素数は定数ではない
    }
    catch ( std::bad_array_new_length )
    {
        // ストレージを確保できなかった場合
    }
}

```

```
try
{
    // numeric_limitsのメンバー関数maxはconstexpr関数なので、定数になる。
    std::size_t const n =
std::numeric_limits<std::size_t>::max() ;
    new int[n] ; // 要素数は定数
}
catch ( std::bad_alloc )
{
    // ストレージを確保できなかった場合
}
}
```

要素数が定数でない場合で、ストレージが確保できない場合のみ、`std::bad_array_new_length`がthrowされる。要素数が定数の場合は、通常通り、`std::bad_alloc`がthrowされる。

オブジェクトの初期化

生成するオブジェクトの初期化は、`new`初期化子によって指定される。`new`初期化子とは、(式リスト)か、初期化リストのいずれかである。`new`初期化子が指定された場合、オブジェクトは、直接初期化される。`new`初期化子が省略された場合、デフォルト初期化される。

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

int main()
{
    // new初期化子が省略されている
    // デフォルト初期化
    new C ;

    // 直接初期化
    new C(0) ;
```

```

new C(0, 0) ;

// 初期化リスト
new C{0} ;
new C{0,0} ;
}

```

組み込み型に対するデフォルト初期化は、「初期化しない」という挙動なので、注意を要する。初期化についての詳しい説明は、8.5 [初期化子](#)を参照。

型名としてのauto

`new`の型名が`auto`の場合、`new`初期化子は、(代入式)の形を取らなければならぬ。オブジェクトの型は、代入式の結果の型となる。オブジェクトは代入式の結果の値で初期化される。

```

int f() { return 0 ; }
int main()
{
    // int型、値は0
    new auto( 0 ) ;
    // double型、値は0.0
    new auto( 0.0 ) ;
    // float型、値は0.0f
    new auto( 0.0f ) ;
    // int型、値は関数fの戻り値
    new auto( f() ) ;
}

```

これは、`auto`指定子とよく似ている。

placement new

`placement new`とは、確保関数に追加の引数を渡すことができる`new`式の文法である。これは、対応する`new`演算子のオーバーロード関数を呼び出す。

```

void * operator new( std::size_t size, int )
throw(std::bad_alloc)
{ return operator new(size) ; }

```

```

void * operator new( std::size_t size, int, int )
throw(std::bad_alloc)
{ return operator new(size) ; }
void * operator new( std::size_t size, int, int, int )
throw(std::bad_alloc)
{ return operator new(size) ; }

int main()
{
    new(1) int ; // operator new( sizeof(int), 1 )
    new(1,2) int ; // operator new( sizeof(int), 1, 2 )
    new(1,2,3) int ; // operator new( sizeof(int), 1, 2,
3 )
}

```

このように、newと型名の間に、通常の関数の実引数のリストのように、追加の引数を指定することができる。追加の引数は、operator newの二番目以降の引数に渡される。placement newの追加の引数は、ストレージを確保する方法を確保関数に指定するなどの用途に使える。

特殊なplacement new

C++には、あらかじめplacement newが二つ定義されている。operator new(std::size_t, const std::nothrow_t &) throw()と、operator new(std::size_t, void *) throw()である。

operator new(std::size_t, const std::nothrow_t &) throw()は、ストレージの確保に失敗しても例外を投げない特別な確保関数である。これには通常、std::nothrowが渡される。

```

// デフォルトで実装により定義される確保関数
// void * operator new(std::size_t, const std::nothrow_t
&) throw() ;

int main()
{
    // 失敗しても例外を投げない
    int * ptr = new(std::nothrow) int ;

    if ( ptr != nullptr )
    {
        // オブジェクトの生成に成功
        // 参照できる
    }
}

```

```

        *ptr = 0 ;
    }

    delete ptr ;
}

```

nothrow版のnew演算子のオーバーロードは、ストレージの確保に失敗しても、例外を投げない。かわりに、nullポインターを返す。これは、newは使いたいが、どうしても例外を使いたくない状況で使うことができる。nothrow版のnewを呼び出した場合は、戻り値がnullポインターであるかどうかを確認しなければならない。

std::nothrow_tは、単にオーバーロード解決のためのタグに過ぎない。また、引数として渡しているstd::nothrowは、単に便利な変数である。

```

// 実装例
namespace std {
    struct nothrow_t {} ;
    extern const nothrow_t noexcept ;
}

```

operator new(std::size_t, void *) throw()は、非常に特別な確保関数である。この形のnew演算子はオーバーロードできない。このnew演算子は、ストレージを確保する代わりに、第二引数に指定されたポインターの指すストレージ上に、オブジェクトを構築する。第二引数のポインターは、オブジェクトの構築に必要なサイズやアライメント要求などの条件を満たしていなければならない。

一般に、placement newといえば、この特別なnew演算子の呼び出しを意味する。ただし、正式なplacement newという用語の意味は、追加の実引数を指定するnew式の文法である。

```

struct C
{
    C(){ std::cout << "constructed." << std::endl ; }
    ~C(){ std::cout << "destructed." << std::endl ; }
} ;

int main()
{
    // ストレージを自前で確保する
    // operator newの返すストレージは、あらゆるアライメント要求を満たす
    void * storage = operator new( sizeof(C) ) ;

    // placement newによって、ストレージ上にオブジェクトを構
}

```

築

```
C * ptr = new( storage ) C ;

// ストレージの解放の前に、デストラクターを呼び出す
ptr->~C() ;

// ストレージを自前で解放する
operator delete( storage ) ;
}
```

ストレージは自前で確保しなければならないので、通常通りdelete式を使うことはできない。デストラクターを自前で呼び出し、その後に、ストレージを自前で解放しなければならない。

ストレージは、動的ストレージでなくても構わない。ただし、アライメント要求には注意しなければならない。

```
struct C
{
    int x ;
    double y ;
} ;

int main()
{
    // ストレージは自動変数
    char storage [[align(C)]] [sizeof(C)] ;

    // placement newによって、ストレージ上にオブジェクトを構
築
    C * ptr = new( storage ) C ;

    // デストラクターはtrivialなので呼ぶ必要はない。
    // ストレージは自動変数なので、解放する必要はない
}
```

この例では、`sizeof(C)`の大きさのchar配列の上にオブジェクトを構築している。アトリビュートを使い、アライメントを指定していることに注意。

このplacement newは、STLのアロケーターを実装するのにも使われている。

ストレージの確保に失敗した場合のエラー処理

確保関数がストレージの確保に失敗した場合、`std::bad_alloc`例外が`throw`される。placement newの`std::nothrow_t`を引数に取る確保関数の場合は、戻り値のポインターが、`null`ポインターとなる。

```
int main()
{
    try
    {
        new int ;
    }
    catch ( std::bad_alloc )
    {
        // エラー処理
    }

    int * ptr = new(std::nothrow) int ;

    if ( ptr == nullptr )
    {
        // エラー処理
    }
}
```

初期化に失敗した場合のエラー処理

`new`が失敗する場合は、二つある。ストレージが確保に失敗した場合と、オブジェクトの初期化に失敗した場合である。

たとえストレージが確保できたとしても、オブジェクトの初期化は、失敗する可能性がある。なぜならば、初期化の際に、コンストラクターが例外を投げるかもしれませんからだ。

```
// 例外を投げるコンストラクターを持つクラス
struct Fail
{
    Fail() { throw 0 ; }
} ;

int main()
{
    try
```

```
{  
    new Fail ; // 必ず初期化に失敗する  
}  
catch ( int ) { }  
}
```

コンストラクターが例外を投げた場合、newは、確保したストレージを、対応する解放関数(deallocation function)を呼び出して解放する。そして、コンストラクターの投げた例外を、そのまま外に伝える。

対応する解放関数とは何か。通常は、operator delete(void *)である。しかし、placement newを使っている場合は、最初の引数を除く、残りの引数の数と型が一致するoperator deleteになる。

```
// placement new  
void * operator new( std::size_t size, int, int, int )  
throw(std::bad_alloc)  
{ return operator new(size) ; }  
  
// placement delete  
void operator delete( void * ptr, int, int, int ) throw()  
{  
    std::cout << "placement delete" << std::endl ;  
    operator delete(ptr) ;  
}  
  
// 例外を投げるかもしれないクラス  
struct Fail  
{  
    Fail() noexcept(false) ; // 例外を投げる可能性がある  
} ;  
  
int main()  
{  
    // コンストラクターが例外を投げた場合、  
    // operator delete( /*ストレージへのポインター*/, 1, 2,  
    3 )が呼ばれる  
    Fail * ptr = new(1, 2, 3) Fail ;  
  
    // operator delete(void *)が呼ばれる  
    delete ptr ;  
}
```

初期化が失敗した場合のplacement deleteの呼び出しには、placement newに渡された追加の引数と、全く同じ値が渡される。

なお、delete式は通常通り、operator delete(void *)を呼び出す。たとえplacement newで確保したオブジェクトであっても、delete式では対応する解放関数は呼ばれない。あくまで、初期化の際に呼ばれるだけである。また、delete式から、placement deleteを呼び出す文法も存在しない。これは、「newの際に指定した情報を、deleteの際にまで保持しておくのは、ユーザー側にとっても実装側にとっても困難である」という思想に基づく。

確保関数の選択

new式が呼び出す確保関数は、以下の方法で選択される。

生成するクラスのメンバー関数に、operator newのオーバーロードがある場合、メンバー関数が選ばれる。メンバー関数によってオーバーロードされていない場合、グローバルスコープのoperator newが選ばれる。new式が、「::new」で始まる場合、たとえメンバー関数によるオーバーロードがあっても、グローバルスコープのoperator newが選ばれる。

```
// オーバーロードあり
struct A
{
    void * operator new( std::size_t size )
throw( std::bad_alloc ) ;
} ;

// オーバーロードなし
struct B { } ;

int main()
{
    // A::operator newが選ばれる
    new A ;
    // ::operator newが選ばれる
    new B ;

    // ::operator newが選ばれる
    ::new A ;
}
```

配列の場合も同様である。配列の場合メンバー関数は、配列の要素のクラス型のメンバーから探される。

```
// オーバーロードあり
struct A
{
    void * operator new[]( std::size_t size )
throw( std::bad_alloc ) ;
} ;

// オーバーロードなし
struct B { } ;

int main()
{
    // A::operator new[]が選ばれる
    new A[1] ;
    // ::operator new[]が選ばれる
    new B[1] ;

    // ::operator new[]が選ばれる
    ::new A[1] ;
}
```

placement newの場合、追加の引数が、オーバーロード解決によって考慮され、最も最適なオーバーロード関数が選ばれる。

```
void * operator new( std::size_t size, int ) throw(
std::bad_alloc ) ;
void * operator new( std::size_t size, double ) throw(
std::bad_alloc ) ;
void * operator new( std::size_t size, int, int ) throw(
std::bad_alloc ) ;

int main()
{
    // operator new( std::size_t size, int )
    new(0) int ;
    // operator new( std::size_t size, double )
    new(0.0) int ;
    // operator new( std::size_t size, int, int )
    new(1, 2) int ;
}
```

CV修飾されている型のnew

CV修飾子のある型もnewできる。特に変わることはない。

```
int main()
{
    int const * ptr = new int const(0) ;
    delete ptr ;
}
```

5.6.5 delete

確保関数と解放関数の具体的な実装方法については、[動的メモリー管理](#)を参照。

```
::opt delete 式  
::opt delete [ ] 式
```

new式によって確保したオブジェクトの寿命は、スコープにはとらわれない。オブジェクトを破棄したければ、delete式で解放しなければならない。

deleteのオペランドの値は、new式によって返されたポインターでなければならない。オブジェクトが配列ではない場合は、deleteを、配列の場合は、delete []を使う。delete式の結果の型は、voidである。

```
int main()
{
    int * ptr = new int ;
    int * array_ptr = new int[1] ;

    delete ptr ;
    delete[] array_ptr ;
}
```

配列であるかどうかで、deleteとdelete[]を使い分けなければならない。これは間違えやすいので注意すること。

deleteのオペランドがクラスのオブジェクトであった場合、非explicitなユーザー定義の

変換が定義されている場合、オブジェクトへのポインターに変換される。

```
struct C
{
    operator int *() { return new int ; }
} ;

int main()
{
    C c ;
    // C::operator int *()を呼び出し、
    // 戻り値を解放する。
    delete c ;
}
```

`delete`式は、まず、ポインターの指し示すオブジェクトのデストラクターを呼び出す。次に、解放関数を呼び出して、ストレージを解放する。オブジェクトの指す型が、メンバー関数として`operator delete`のオーバーロードを持つ場合、メンバー関数が呼ばれる。オーバーロードされたメンバー関数が存在しない場合、グローバルスコープの`operator delete`を呼び出す。`delete`式が、「`::operator delete`」で始まる場合、メンバー関数のオーバーロードの有無にかかわらず、グローバルスコープの`operator delete`を呼び出す。

```
// オーバーロードあり
struct A
{
    void operator delete( void * ) throw() ;
} ;

// オーバーロードなし
struct B { } ;

int main()
{
    A * a = new A ;
    // A::operator delete(void*)を呼び出す
    delete a ;

    B * b = new B ;
    // ::operator delete(void*)を呼び出す
    delete b ;

    a = new A ;
    // ::operator delete(void*)を呼び出す
```

```
::delete a ;
}
```

オブジェクトが、placement newで確保されたとしても、呼び出す解放関数は、必ず operator delete(void *)、もしくはoperator delete[](void *)となる。delete式では、 placement deleteは呼び出されない。また、delete式には、placement deleteを呼び出すための文法も存在しない。どうしてもplacement deleteを呼び出したい場合は、手動でデストラクターを呼び出し、さらに手動でplacement deleteを呼び出すしかない。

```
// placement delete
void operator delete( void *, int ) throw() ;

struct C
{
    C() {}
    ~C(){}
} ;

void f()
{
    C * ptr = new C ;

    // これでは、operator delete( void * )が呼び出される
    delete ptr ;

    // 疑似デストラクター呼び出し
    ptr->~C() ;
    // operator deleteの明示的な呼び出し
    operator delete( ptr, 0 ) ;
}
```

5.6.6 alignof

alignof (型名)

alignof式は、オペランドの型のアライメント要求を返す。オペランドの型は、完全なオブジェクト型か、その配列もしくはリファレンスでなければならない。式の結果は、std::size_t型の定数になる。

オペランドが、リファレンス型の場合、結果は参照される型のアライメント要求になる。
配列の場合、結果は配列の要素の型のアライメント要求になる。

```
struct C
{
    char c ; int i ; double d ;
} ;

void f()
{
    // char型のアライメント要求を返す
    alignof( char ) ;
    // int型のアライメント要求を返す
    alignof( int ) ;
    // double型のアライメント要求を返す
    alignof( double ) ;
    // C型のアライメント要求を返す
    alignof( C ) ;
}
```

5.6.7 noexcept演算子 (noexcept operator)

`noexcept (未評価式)`

noexcept演算子は、オペランドの式が、例外を投げる可能性のある式を含むかどうかを返す。noexcept演算子の結果の型はboolの定数で、例外を投げる可能性のある式を含まない場合trueを、含む場合falseを返す。オペランドの式は、評価されない。

結果がfalseとなる場合、すなわち、例外を投げる可能性のある式とは、以下の通りである。

throw式。

```
// false
noexcept( throw 0 ) ;
```

dynamic_cast式、dynamic_cast<T>(v)において、Tがリファレンス型で、実行時チェックが必要な場合。

```

struct Base { virtual void f() {} } ;
struct Derived : Base { } ;

void f( Base & ref )
{
    // false
    noexcept( dynamic_cast<Derived & >( ref ) ) ;
}

```

typeid式において、オペランドがglvalueで、実行時チェックが必要な場合。

```

struct Base { virtual void f() {} } ;
struct Derived : Base { } ;

void f( Base * ptr )
{
    // false
    noexcept( typeid( *ptr ) ) ;
}

```

関数、メンバー関数、関数ポインター、メンバー関数ポインターを呼び出す式において、呼び出す関数の例外指定が、無例外(non-throwing)でないもの。

```

void a() ;
void b() noexcept ; // non-throwing
void c() noexcept(true) ; // non-throwing
void d() noexcept(false) ;
void e() throw() ; // non-throwing
void f() throw(int) ;

int main()
{
    noexcept( a() ) ; // false
    noexcept( b() ) ; // true
    noexcept( c() ) ; // true
    noexcept( d() ) ; // false
    noexcept( e() ) ; // true
    noexcept( f() ) ; // false
}

```

関数を、「呼び出す式」というのは、関数を間接的に呼び出す場合も該当する。たとえば、new式は確保関数を呼び出すので、関数を呼び出す式である。その場合の結果は、呼び出される確保関数の例外指定に依存する。

```
int main()
{
    // ::operator new( std::size_t ) throw( std::bad_alloc )
    // を呼び出す
    std::cout << noexcept( new int ) ; // false

    // ::operator new( std::size_t, std::nothrow_t ) throw()
    // を呼び出す
    std::cout << noexcept( new(std::nothrow) int ) ; // true
}
```

もちろん、演算子のオーバーロード関数も、「関数」である。従って、演算子のオーバーロード関数を呼び出す式は、関数を呼び出す式である。

```
struct C
{
    C operator +( C ) ;
    C operator -( C ) noexcept ;
} ;

int main()
{
    int i = 0 ;
    noexcept( i + i ) ; // true

    C c ;
    noexcept( c + c ) ; // false
    noexcept( c - c ) ; // true
}
```

その他にも、関数を間接的に呼び出す可能性のある式というのは、非常に多いので、注意しなければならない。

関数のオーバーロード解決は静的に行われる所以、当然、呼び出される関数に応じて結果も変わる。

```

void f(int) noexcept ;
void f(double) ;

int main()
{
    noexcept( f(0) ) ; // true
    noexcept( f(0.0) ) ; // false
}

```

例外を投げる可能性のある式を「含む」というのは、たとえその式が絶対に評価されないでも、例外を投げる可能性があるとみなされる。例えば、

```
noexcept( true ? 0 : throw 0 ) ; // false
```

この`noexcept`のオペランドの式は、もし評価された場合、決して例外を投げることがない。しかし、例外を投げる可能性のある式を含んでいるので、`noexcept`の結果は`false`となる。

上記以外の場合、`noexcept`の結果は`true`となる。

```

struct Base { } ;
struct Derived : Base { } ;

int main()
{
    noexcept( 0 ) ; // true

    Derived d ;
    noexcept( d ) ; // true
    noexcept( dynamic_cast<Base &>( d ) ) ; // true
    noexcept( typeid( d ) ) ; // true
}

```

5.7 キャスト形式による明示的型変換(Explicit type conversion (cast notation))

注意:C形式のキャストには様々な問題があるので、使ってはならない。

(型名) 式

これは、悪名高いC形式のキャストである。

```
int main()
{
    int i = 0 ;
    double * ptr = (double *) &i ;
}
```

C形式のキャストは、`static_cast`と`reinterpret_cast`と`const_cast`を組み合わせた働きをする。組み合わせは、以下の順序で決定される。

1. `const_cast`
2. `static_cast`
3. `static_cast`と`const_cast`
4. `reinterpret_cast`
5. `reinterpret_cast`と`const_cast`

上から下に評価していき、変換できる組み合わせが見つかったところで、そのキャストを使って変換する。

ただし、C形式のキャストでは、`static_cast`に特別な変更を三つ加える。クラスのアクセス指定を無視できる機能である。

派生クラスへのポインター やリファレンスから、基本クラスへのポインター やリファレンスに変換できる。文字通り変換できる。アクセス指定などは考慮されない。

```
struct Base { } ;
struct Derived : private Base { } ;

int main()
{
    Derived d ;

    Base & ref1 = (Base &) d ; // OK
    Base & ref2 = static_cast<Base &>(d) ; // ill-formed
}
```

このため、`public`ではない基本クラスにアクセスできてしまう。

派生クラスのメンバーへのポインターから、曖昧ではない非virtualな基本クラスのメンバーへのポインターに変換できる。文字通り変換できる。アクセス指定などは考慮されない。

```
struct Base { } ;
struct Derived : private Base { int x ; } ;

int main()
{
    int Base::* ptr1 = (int Base::*) &Derived::x ; // OK
    int Base::* ptr2 = static_cast<int Base::*>
(&Derived::x) ; // ill-formed
}
```

これも、アクセス指定を無視できてしまう。

曖昧ではなく非virtualな基本クラスのポインターやリファレンスあるいはメンバーへのポインターは、派生クラスのポインターやリファレンスあるいはメンバーへのポインターに変換できる。文字通り変換できる。アクセス指定などは考慮されない。

```
struct Base { int x ; } ;
struct Derived : private Base { } ;

int main()
{
    Derived d ;

    d.x = 0 ; // ill-formed. アクセス指定のため

    int Derived::* ptr = (int Derived::*) &Base::x ; // well-formed.
    d.*ptr = 0 ; // well-formed. C形式のキャストを使ったため、アクセス指定を無視できている
}
```

C形式のキャストでしかできないキャストとは、クラスのアクセス指定を無視し、しかもクラス階層のナビゲーションを行うキャストのことである。

これらのキャストは、`reinterpret_cast`でもできる。ただし、`reinterpret_cast`は、クラス階層のナビゲーションを行わないので、正しく動かない。`static_cast`は、クラス階層のナビゲーションを行うので、正しく動く。

アクセス指定を無視できるキャストをしなければならない場合というのは、現実には存在しないはずである。アクセス指定を無視するぐらいならば、最初からpublicにしておけばいい。

`reinterpret_cast`は必要である。C++が必要とされる環境では、ポインターの内部的な値を、そのまま別の型のポインターとして使わなければならない場合も存在する。また、既存のCのコードとの互換性のため、`const_cast`も残念ながら必要である。しかし、アクセス指定は、C++に新しく追加された概念であるので、互換性の問題も存在しないし、また、アクセス指定を無視しなければならない場合というのも、全く考えられない。従って、アクセス指定を無視できるという理由で、C形式のキャストを使ってはならない。

そもそも、C形式のキャストは根本的に邪悪であるので、使ってはならない。C形式のキャストの問題点は、できることが多すぎるということだ。安全なキャストも、危険なキャストも、全く同じ文法で行うことができる。C++では、この問題を解決するために、キャストを三つに分けた。`static_cast`、`reinterpret_cast`、`const_cast`である。C++では、この新しい形式のキャストを使うべきである。以下にその概要と簡単な使い分けをまとめると。

5.5.9 `static_cast`は、ほとんどが安全なキャストである。`static_cast`は、型変換を安全にするため、値を変えることもある。値を変更するので、`static_cast`は、クラス階層のナビゲーションを行うことができる。派生クラスと基本クラスとの間のポインターの型変換は、ポインターの内部的な値が変わる可能性があるからだ。ポインターの値は、もとより実装依存であるが、最も多くの環境で再現できるコードは、複数の基本クラスを使うものだ。

```

struct Base1 { int x ; } ;
struct Base2 { int x ; } ;

struct Derived : Base1, Base2 { } ;

int main()
{
    Derived d ;
    Derived * ptr = &d ;

    // 基本クラスへのキャスト
    Base1 * base1 = static_cast<Base1 *>( ptr ) ;
    Base2 * base2 = static_cast<Base2 *>( ptr ) ;

    // 派生クラスへのキャスト
    Derived * d1 = static_cast<Derived *>( base1 ) ;
    Derived * d2 = static_cast<Derived *>( base2 ) ;

```

```

// 派生クラスのポインターの値
std::printf( "Derived *: %p\n", ptr ) ;

// 基本クラスのポインターの値は同じか?
std::printf( "Base1 *: %p\n", base1 ) ;
std::printf( "Base2 *: %p\n", base2 ) ;

// 派生クラスに戻した場合はどうか?
std::printf( "from Base1 * to Derived *: %p\n", d1 ) ;
std::printf( "from Base1 * to Derived *: %p\n", d2 ) ;
}

```

複数の基本クラスの場合、基本クラスのサブオブジェクトが複数あるので、派生クラスと基本クラスのポインターの間で、同じ値を使うことができない。従って、基本クラスへのポインターにキャストするには、ストレージ上の、その基本クラスのサブオブジェクトを指すポインターを返さなければならない。また、派生クラスへのポインターにキャストするには、値を戻さなければならない。

このため、クラス階層のナビゲーションには、`static_cast`か`dynamic_cast`を用いなければならない。

5.5.10 `reinterpret_cast`は、危険で愚直なキャストである。`reinterpret_cast`は、値を変えない。ただ、その値の型だけを変更する。`reinterpret_cast`は、クラス階層のナビゲーションができない。

5.5.11 `const_cast`は、CV修飾子を外すキャストである。

もし、どのキャストを使うべきなのか判断できない場合は、まず`static_cast`を使っておけば問題はない。もし、`static_cast`が失敗した場合、本当にそのキャストは安全なのかということを確かめてから、`reinterpret_cast`を使うべきである。`const_cast`は、既存のCのコードの利用以外に使ってはならない。

5.8 メンバーへのポインター演算子(Pointer-to-member operators)

式 . * 式
式 ->* 式

メンバーへのポインター演算子は、「左から右」に評価される。

メンバーへのポインター演算子は、クラスのメンバーへのポインターを使って、クラスのオブジェクトのメンバーにアクセスするための演算子である。クラスのメンバーへのポインターを参照するためには、参照するクラスのオブジェクトが必要である。

*演算子の第一オペランドには、クラスのオブジェクトを指定する。->*演算子の第一オペラントには、クラスへのポインターを指定する。第二オペラントには、クラスのメンバーへのポインターを指定する。

```
struct C
{
    int member ;
} ;

int main()
{
    int C::* mem_ptr = &C::member ;

    C c ;
    c.*mem_ptr = 0 ;

    C * ptr = &c ;
    ptr->*mem_ptr = 0 ;
}
```

メンバー関数の呼び出しの際は、演算子の優先順位に気をつけなければならない。

```
struct C
{
    void member() {}
} ;

int main()
{
    void (C::* mem_ptr)() = &C::member ;

    C c ;
    (c.*mem_ptr)() ;

    C * ptr = &c ;
    (ptr->*mem_ptr)() ;
}
```

なぜならば、メンバーへのポインター演算子の式より、関数呼び出し式の優先順位の方が高いので、c.*mem_ptr()という式は、c.*(&mem_ptr())という式に解釈されてしまう。これは、mem_ptrという名前に対して、関数呼び出し式を適用した後、その結果を、クラスのメンバーへのポインターとして使う式である。このように解釈されることを避けるた

めに、括弧式を使わなければならない。

その他の細かいルールについては、5.5.5 クラスメンバーアクセスと同じである。

5.9 乗除算の演算子 (Multiplicative operators)

```
式 * 式  
式 / 式  
式 % 式
```

乗除算の演算子は、「左から右」に評価される。

*演算子と/演算子のオペランドは、数値型かunscoped enum型でなければならない。%演算子のオペランドは、整数型かunscoped enum型でなければならぬ。オペランドには、通常通り数値に関する標準型変換が適用される。5 式を参照。

*演算子は、乗算を意味する。

/演算子は、除算を意味する。%演算子は、第一オペランドを第二オペランドで割った余りを意味する。第二オペランドの値が0の場合の挙動は未定義である。/演算子の結果の型が整数の場合、小数部分は切り捨てられる。

```
int main()  
{  
    2 * 3 ; // 6  
    10 / 5 ; // 2  
    3 % 2 ; // 1  
  
    3 / 2 ; // 結果は整数型、小数部分が切り捨てられるので、結果は  
    1  
  
    3.0 / 2.0 ; // 結果は浮動小数点数型の1.5  
}
```

以下は間違っている例である。

```
// このコードは間違っている例  
int main()  
{  
    // ゼロ除算
```

```

1 / 0 ;

// %演算子のオペランドに浮動小数点数型は使えない
3.0 % 2.0 ;
}

```

5.10 加減算の演算子(Additive operators)

式 + 式
式 - 式

加減算の演算子は、「左から右」に評価される。

両方のオペランドが数値型の場合

+演算子は、加算を意味する。-演算子は、減算を意味する。-演算子の減算とは、第二オペランドの値を第一オペランドから引くことである。結果の型には、通常通り数値型に関する標準型変換が行われる。

```

int main()
{
    1 + 1 ; // 2
    1 - 1 ; // 0
}

```

オペランドがポインター型の場合

まず、ポインターの型は、完全に定義されたオブジェクトでなければならない。ポインターは、配列の要素を指し示しているものとみなされる。たとえ実際には配列の要素を指していないとしても、配列の要素を指しているものとみなされる。

+演算子の片方のオペランドがポインター型の場合、もう片方は、整数型でなければならない。-演算子は、両方のオペランドが同じポインター型か、左オペランドがポインター型で右オペランドが整数型でなければならない。

```

int main()

```

```
{
    int array[3] ;
    int * ptr = &array[1] ;

    // OK
    ptr + 1 ;
    1 + ptr ;
    ptr + (-1) ;
    (-1) + ptr ;
    ptr - ptr ;
    ptr - 1 ;
    ptr - (-1) ;

    // エラー
    ptr + ptr ; // +演算子の両オペランドがポインターとなって
    いる
    1 - ptr ; // -演算子の左オペランドが整数で右オペランドが
    ポインターとなっている
}
```

ポインターと整数の加減算の結果の型は、ポインターの型である。結果の値は、ポインターが指す要素に対する配列中の添字に、整数を加減算した要素を指すものとなる。もし、ポインターが配列の添字で i 番目の要素を指し示している場合、このポインターに整数 n を加算することは、 $i + n$ 番目の要素を指し示すことになる。同様にして、整数 n を減算することは、 $i - n$ 番目の要素を指し示すことになる。

```
int main()
{
    int array[10] ;
    int * ptr = &array[5] ;

    ptr + 2 ; // &array[5 + 2]と同じ
    ptr - 2 ; // &array[5 - 2]と同じ
}
```

もし、ポインターが、配列の最後の要素を指している場合、これに1を加えると、結果のポインターは配列の最後の要素のひとつ後ろを指すことになる。ポインターが配列の最後の要素のひとつ後ろを指している場合、これから1を引くと、結果のポインターは配列の最後の要素を指すことになる。

```
int main()
{
```

```

int array[10] ;
// 配列の最後の要素を指す
int * ptr = &array[9] ;

// 配列の最後の要素のひとつ後ろを指す
int * one_past_the_last = ptr + 1 ;
// 配列の最後の要素を指す
int * last = one_past_the_last - 1 ;
}

```

配列の最後の要素を指しているポインターに1を加算して、最後の要素の一つ後の要素を指すようにしても、規格上、ポインターの値のオーバーフローは起こらないと保証されている。2つ目以降の要素を指し示した場合、挙動は未定義である。

```

int main()
{
    int a[1] ;
    int * p1 = &a[0] ; // 最後の要素

    int * p2 = p1 + 1 ; // OK、最後の一つ後の要素
    int * p3 = p2 + 1 ; // 挙動は未定義
}

```

上の例で、もし、ポインターp2を参照した場合、挙動は未定義だが、p2自体は未定義ではない。p3は未定義である。

ポインター同士を減算した場合、結果は、ポインターの指す配列の添字の差になる。ポインターPが配列の添字でi番目の要素を差しており、ポインターQが配列の添字でj番目の要素を指している場合、 $P - Q$ は、 $i - j$ となる。配列の添字は、0から始まることに注意。両方のポインターが同じ配列上の要素を差していない場合、挙動は未定義である。

```

int main()
{
    int array[10] ;
    int * P = &array[2] ;
    int * Q = &array[7] ;

    P - Q ; // 2 - 7 = -5
    Q - P ; // 7 - 2 = 5
}

```

ポインター同士の減算の結果の型は、実装依存であるが、`<cstddef>`ヘッダーで定義されている、`std::ptrdiff_t`と同じ型になる。

0という値が、ポインターに足し引きされた場合、結果は、そのポインターの値になる。

```
void f( int * ptr )
{
    ptr == ptr + 0 ; // true
    ptr == ptr - 0 ; // true
}
```

5.11 シフト演算子(Shift operators)

式 \ll 式
式 \gg 式

シフト演算子のオペランドは、整数型かunscoped enum型でなければならない。オペランドには、整数のプロモーションが行われる。結果の型は、整数のプロモーションが行われた後のオペランドの型になる。

左シフト、 $E1 \ll E2$ の結果は、 $E1$ を $E2$ ビット、左にシフトしたものとなる。シフトされた後のビットは、0で埋められる。もし、 $E1$ の型が`unsigned`ならば、結果の値は、 $E1 \times 2^{E2}$ を、 $E1$ の最大値+1で剰余したものとなる。

```
// コメント内の値は2進数である。
int main()
{
    // 1101
    unsigned int bits = 9 ;

    bits << 1 ; // 11010
    bits << 2 ; // 110100
}
```

$E1$ の型が`signed`の場合、 $E1$ が負数でなく、 $E1 \times 2^{E2}$ が表現可能であれば、その値になる。その他の場合は未定義である。これは、`signed`な整数型の内部表現が2の補数であるとは保証していないので、このようになっている。

```
// コメント内の値は2進数である
int main()
{
    // 1101
    int bits = 9 ;

    bits << 1 ; // 11010
    bits << 2 ; // 110100

    -1 << 1 ; // 結果は未定義
}
```

右シフト、 $E_1 \gg E_2$ の結果は、 E_1 を E_2 ビット、右にシフトしたものとなる。もし、 E_1 の型が unsignedか、signedで正の数ならば、結果の値は、 $E_1 \div 2^{E_2}$ の整数部分になる。

```
// コメント内の値は2進数である
int main()
{
    // 1101
    unsigned int value = 9 ;

    value >> 1 ; // 110
    value >> 2 ; // 11

    int signed_value = 9 ;

    signed_value >> 1 ; // 110
    signed_value >> 2 ; // 11
}
```

E_1 の型がsignedで、値が負数の場合、挙動は未定義である。

```
int main()
{
    -1 >> 1 ; // 結果は未定義
}
```

右オペランドの値が負数であったり、整数のプロモーション後の左オペランドのビット数以上の場合の挙動は未定義である。

```
// この環境では、1バイトは8ビット
// sizeof(unsigned int)は2とする。
// すなわち、この環境では、unsigned intは16ビットとなる。
int main()
{
    unsigned int value = 1 ;
    value << -1 ; // 未定義
    value >> -1 ; // 未定義

    value << 16 ; // 未定義
    value >> 16 ; // 未定義

    value << 17 ; // 未定義
    value >> 17 ; // 未定義
}
```

シフト演算には、未定義の部分が非常に多い。ただし、多くの現実の環境では、何らかの具体的な意味が定義されていて、時として、そのような未定義の挙動に依存したコードを書かなければならない場合がある。その場合、特定の環境に依存したコードだという正しい認識を持たなければならない。

5.12 関係演算子 (Relational operators)

```
式 < 式
式 > 式
式 <= 式
式 >= 式
```

関係演算子は「左から右」に評価される。

関係演算子のオペランドには、数値型、enum型、ポインター型を使うことができる。各演算子の意味は、以下のようになっている。

A < B	AはBより小さい
A > B	AはBより大きい
A <= B	AはBより小さいか、等しい
A >= B	AはBより大きいか、等しい

結果の型はboolとなる。両オペランドが数値型かenum型の場合、不等号の関係が正しければtrueを、そうでなければfalseを返す。

```
void f( int a, int b )
{
    a < b ;
    a > b ;
    a <= b ;
    a >= b ;
}
```

式の結果の型はboolである。

ポインター同士の比較に関しては、未規定な部分が多い。ここでは、規格により保証されていることだけを説明する。

同じ型の二つのポインター、pとqが、同じオブジェクトか関数を指している場合、もしくは、同じ配列の最後の要素のひとつ後の要素を指している場合、もしくは、両方ともnullの場合は、 $p \leq q$ と $p \geq q$ はtrueとなり、 $p < q$ と $p > q$ はfalseとなる。

```
int main()
{
    int object = 0 ;
    int * p = &object ;
    int * q = &object ;

    p <= q ; // true
    p >= q ; // true

    p < q ; // false
    p > q ; // false
}
```

同じ型の二つのポインター、pとqが、異なるオブジェクトを差しており、そのオブジェクトは同じオブジェクトのメンバーではなく、また同じ配列内の要素ではなく、異なる関数でもなく、あるいは、どちらか片方の値のみがnullの場合、 $p < q$, $p > q$, $p \leq q$, $p \geq q$ の結果は、未規定である。

```
int main()
{
    int object1 ;
    int object2 ;
```

```

int * p = &object1 ;
int * q = &object2 ;

p <= q ; // 結果は未規定
p >= q ; // 結果は未規定

p < q ; // 結果は未規定
p > q ; // 結果は未規定

p < nullptr ; // 結果は未規定
}

```

同じ型の二つのポインター、pとqが、同じ配列の要素を指している場合、添字の大きい要素の方が、より大きいと評価される。

```

int main()
{
    int array[2] ;

    int * p = &array[0] ;
    int * q = &array[1] ;

    p < q ; // true
    p > q ; // false

    p <= q ; // true
    p >= q ; // false

```

これと同様に、pとqが指しているものが、同じ型の同じクラスのオブジェクトのサブオブジェクトである場合は、同じアクセスコントロール下にある場合、後に宣言されたメンバーの方が、ポインター同士の比較演算では、大きいと評価される。

```

struct S
{
// 同じアクセスコントロール下
    int a ;
    int b ; // bが後に宣言されている
} ;

int main()

```

```
{  
    S object ;  
    // 同じオブジェクトのサブオブジェクト  
    int * p = &object.a ;  
    int * q = &object.b ;  
  
    p < q ; // true  
    p > q ; // false  
};
```

これと似ているが、ただしクラスのメンバーのアクセスコントロールが異なる場合、結果は未規定である。

```
struct S  
{  
public :  
    int a ;  
private :  
    int b ;  
  
    void f()  
    {  
        &a < &b ; // 結果は未規定  
    }  
};
```

二つのポインター、pとqが、unionの同じオブジェクトの非staticなデータメンバーを指している場合、等しいと評価される。

```
union Object  
{  
    int x ;  
    int y ;  
};  
  
int main()  
{  
    Object object ;  
    int * p = &object.x ;  
    int * q = &object.y ;
```

```

p < q ; // false
p > q ; // false

p <= q ; // true
p >= q ; // true

p == q ; // true
}

```

二つのポインターが、同じ配列内の要素を指している場合、添字の大きい要素を指すポインターが、大きいと評価される。また、これはどちらか片方のポインターが、配列の範囲を超えていても、評価できる。

```

int main()
{
    int a[2] ;
    int * p1 = &a[0] ;
    int * p2 = &a[1] ;

    p1 < p2 ; // true

    int * p3 = p2 + 1 ; // p3は配列の範囲外を指す

    p1 < p3 ; // OK、結果はtrue
}

```

voidへのポインター型は、比較することができる。また、片方のオペランドがvoidへのポインター型で、もう片方が別のポインター型である場合、もう片方のオペランドが、標準型変換によってvoidへのポインター型に変換されるので、比較することができる。もし、両方のポインターが、同じアドレスであった場合かnullポインターの場合は、等しいと評価される。それ以外は、未規定である。

```

int main()
{
    int object = 0 ;

    int * ptr = &object ;
    void * p = ptr ;
    void * q = ptr ;

    p < q ; // false
    p > q ; // false
}

```

```

    p <= q ; // true
    p >= q ; // true

    // 標準型変換によって、別のポインター型とも比較できる
    p <= ptr ; // true
}

```

これ以外の比較の結果は、すべて未規定となっている。未定義ではなく、未規定なので、実装によっては、意味のある結果を返すこともある。しかし、実装に依存する挙動なので、移植性に欠ける。

5.13 等価演算子(Equality operators)

```

式 == 式
式 != 式

```

`==`演算子(等しい)と、`!=`演算子(等しくない)は、5.12 関係演算子とオペランドや結果の型、評価の方法は同じである。ただし比較の意味は、「等しい」か、「等しくない」かである。

```

int main()
{
    1 == 1 ; // true
    1 != 1 ; // false

    1 == 2 ; // false
    1 != 2 ; // true
}

```

同じ型のポインターの場合、ともにアドレスが同じか、ともにnullポインターの場合、`true`と評価される。

`==`演算子は、代入演算子である`=`演算子と間違えやすいので、注意しなければならない。

```

void f( int x )
{

```

```
if ( x = 1 ) // 間違い
{
    // 処理
} else {
    // 処理
}
}
```

この例では、if文の条件式の結果は、代入式の結果となってしまう。それは、1であるので、このif文は常にtrueであると評価されてしまう。

5.14 ビット列論理積演算子(Bitwise AND operator)

式 & 式

ビット列論理積演算子は、両オペランドの各ビットごとの論理積(AND)を返す。オペラントは整数型か、unscoped enum型でなければならない。

5.15 ビット列排他的論理和演算子(Bitwise exclusive OR operator)

式 ^ 式

ビット列排他的論理和演算子は、両オペランドの各ビットごとの排他的論理和(exclusive OR)を返す。オペラントは整数型か、unscoped enum型でなければならない。

5.16 ビット列論理和演算子(Bitwise inclusive OR operator)

式 | 式

ビット列論理和演算子は、両オペランドの各ビットごとの論理和(inclusive OR)を返す。

オペランドは整数型か、unscoped enum型でなければならない。

5.17 論理積演算子(Logical AND operator)

式 && 式

&&演算子は「左から右」に評価される。

論理積演算子は、オペランドの論理積を返す演算子である。両オペランドはboolに変換される。結果の型はboolである。両方のオペランドがtrueであれば、結果はtrue。それ以外はfalseとなる。

```
true && true ; // true
true && false ; // false
false && true ; // false
false && false ; // false
```

第一オペランドを評価した結果がfalseの場合、第二オペランドは評価されない。なぜならば、第一オペランドがfalseであれば、第二オペランドを評価するまでもなく、結果はfalseであると決定できるからである。

```
bool f() { return false ; }
bool g() { return true ; }

int main()
{
    // g()は呼ばれない。結果はfalse
    f() && g() ;
}
```

この例では、第一オペランドである関数fの呼び出しはfalseを返すので、第二オペランドの関数gの呼び出しが評価されることはない。つまり、関数gは呼ばれない。

第二オペランドが評価される時、第一オペランドの評価によって生じた値の計算や副作用は、すべて行われている。

```
int main()
{
```

```

int value = 0 ;

++value // 値は1になるので、trueと評価される
&&
value ; // 値はすでに1となっているので、trueと評価される
}

```

5.18 論理和演算子(Logical OR operator)

式 || 式

||演算子は、「左から右」に評価される。

論理和演算子は、オペランドの論理和を返す演算子である。両オペランドはboolに変換される。結果の型はboolである。オペランドが片方でもtrueと評価される場合、結果はtrueとなる。両オペランドがfalseの場合に、結果はfalseとなる。

```

int main()
{
    true || true ; // true
    true || false ; // true
    false || true ; // true
    false || false ; // false
}

```

第一オペランドを評価した結果がtrueの場合、第二オペランドは評価されない。なぜならば、第一オペランドがtrueであれば、第二オペランドを評価するまでもなく、結果はtrueとなるからである。

```

bool f() { return true ; }
bool g() { return false ; }

int main()
{
    // g()は呼ばれない。結果はtrue
    f() || g() ;
}

```

論理積と同じように、第二オペランドが評価される場合、第一オペランドの評価によって生じた値の計算や副作用は、すべて行われている。

5.19 条件演算子(Conditional operator)

式 ? 式 : 代入式

条件演算子は「左から右」に評価される。

条件演算子は、三つのオペランドを取る。C++には他に三つのオペランドを取る演算子がないことから、三項演算子といえば、条件演算子の代名詞のように使われている。しかし、正式名称は条件式であり、演算子の名称は条件演算子である。

条件演算子の第一オペランドはboolに変換される。値がtrueであれば、第二オペランドの式が評価され、その結果が返される。値がfalseであれば、第三オペランドの式が評価され、その結果が返される。第二オペランドと第三オペランドは、どちらか片方しか評価されない。

```
bool cond() ;
int e1() ;
int e2() ;

int main()
{
    true ? 1 : 2 ; // 1
    false ? 1 : 2 ; // 2

    // 関数condの戻り値によって、関数e1、あるいはe2が呼ばれ、その
    戻り値が返される。
    // e1とe2は、どちらか片方しか呼ばれない。
    cond() ? e1() : e2() ;
}
```

実は、条件演算子は見た目ほど簡単ではない。特に、結果の型をどのようにして決定するかということが、非常に難しい。ここでは、結果の型を決定する完全な詳細は説明しないが、特に重要なと思われる事を取りあげる。

条件演算子の第二第三オペランドには、結果がvoid型となる式を使うことができる。

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

```

int main()
{
    true ? f() : f() ;

    int * ptr = new int ;
    true ? delete ptr : delete ptr ;

    true ? throw 0 : throw 0 ;
}

```

片方のオペランドがvoidで、もう片方がvoidではない場合、エラーである。

```

void f() {}
int main()
{
    true ? 0 : f() ; // エラー
    true ? f() : 0 ; // エラー
}

```

ただし、片方のオペランドがthrow式の場合に限り、もう片方のオペランドに、voidではない式でも使うことができる。結果はprvalueの値で、型はthrow式ではない方のオペランドの型になる。

```

void f() {}

int main()
{
    // OK
    // xに0を代入する
    int x = true ? 0 : throw 0 ;

    // エラー
    // 戻り値に123を代入しようとしているが、prvalueには代入できない
    (true ? x : throw 0) = 123 ;

    true ? throw 0 : f() ; // OK
}

```

両オペランドが、ともに同じ値カテゴリーで、同じ型の場合は、条件演算子の結果は、

その値カテゴリと型になる。

```

int f() { return 0 ; }

int main()
{
    int x = 0 ;

    // 両オペランドとも、lvalueのint型
    // 結果はlvalueのint
    ( true ? x : x ) = 0 ; // lvalueなので代入も可能

    // 両オペランドとも、xvalueのint型
    // 結果はxvalueのint
    true ? std::move(x) : std::move(x) ;

    // 両オペランドとも、prvalueのint型
    // 結果はprvalueのint
    true ? f() : f() ;
}

```

もし、オペランドの値カテゴリや型が違う場合、暗黙の型変換によって、お互いの型と値カテゴリを一致させようという試みがなされる。この変換の詳細は、非常に複雑で、通常は意識する必要はないため、本書では省略する。

5.20 代入と複合代入演算子 (Assignment and compound assignment operators)

式 代入演算子 式

代入演算子: 以下のうちどれか

= *= /= %= += -= >>= <<= &= ^= |=

代入演算子(=)と、複合代入演算子は、「右から左」に評価される。

代入演算子は、左側のオペランドに、右側のオペランドの値を代入する。左側のオペラントは変更可能なlvalueでなければならない。結果として、左側のオペランドのlvalueを返す。

```
int main()
{
    int x ;
    x = 0 ;
}
```

初期化と混同しないように注意。

```
int main()
{
    int x = 0 ; // これは初期化
    x = 0 ; // これは代入
}
```

=を代入演算子といい、その他の演算子を、複合代入演算子という。

クラスの代入に関する詳細は、12.8 [クラスオブジェクトのコピーとムーブ](#)や、オーバーロードの13.7.3 [代入](#)を参照。

複合代入演算子の式、 $E1 \text{ op} = E2$ は、 $E1 = E1 \text{ op } E2$ と同じである。ただし、 $E1$ という式は、一度しか評価されない。 op には、任意の複合代入演算子の一部が入る。

```
int main()
{
    int x = 0 ;

    x += 1 ; // x = x + 1と同じ
    x *= 2 ; // x = x * 2と同じ
}
```

右側のオペランドには、初期化リストを使うことができる。

左側のオペランドがスカラー型の場合、ある型Tの変数をxとすると、 $x = \{v\}$ という式は、 $x = T(v)$ という式と同じ意味になる。ただし、初期化リストなので、縮小変換は禁止されている。 $x = \{\}$ という式は、 $x = T()$ という式と同じ意味になる。

```
int main()
{
    int x ;
    x = {1} ; // x = int(1)と同じ
    x = {} ; // x = int()と同じ
```

```

short s ;
s = {x} ; // エラー、縮小変換は禁止されている。
}

```

それ以外の場合は、初期化リストを実引数として、ユーザー定義の代入演算子が呼び出される。

```

struct C
{
    C(){}
    C( std::initializer_list<int> ) {}
};

int main()
{
    C c ;
    c = { 1, 2, 3 } ;
}

```

5.21 コンマ演算子(Comma operator)

式 , 式

コンマ演算子は、「左から右」に評価される。

コンマ演算子は、まず左のオペランドの式が評価され、次に、右のオペランドの式が評価される。左のオペランドの式を評価した結果は破棄され、右のオペランドの結果が、コンマ演算子の結果として、そのまま返される。結果の型や値、値カテゴリーは、右のオペランドの式を評価した結果と全くおなじになる。

```

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

```

右のオペランドの式が評価される前に、左のオペランドの式の値計算や副作用は、す

でに行われている。

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

int main()
{
    int i = 0 ;
    // 左のオペランドのiは、すでにインクリメントされている。
    ++i, i ;
    // 関数gが呼ばれる前に、関数fはすでに呼ばれ終わっている。
    f(), g() ;
}
```

コンマが特別な意味を持つ場面では、コンマ演算子を使うには、明示的に括弧で囲まなければならない。コンマが特別な意味を持つ場面には、例えば、関数の実引数リストや、初期化リストなどがある。

```
void f(int, int, int) {}
int main()
{
    int x ;
    // 括弧が必要
    f( 1, (x=0, x), 2 ) ;
}
```

この例では、関数fは三つの引数を取る。二つめの引数は、括弧式に囲まれたコンマ演算子の式である。これは変数xに0を代入した後、そのxを引数として渡している。

5.22 定数式 (Constant expressions)

定数式 (constant expression) とは、値がコンパイル時に決定できる式のことである。定数式かどうかということは、C++のいくつかの場面で、重要になってくる。例えば、配列を宣言する時、要素数は定数式でなければならない。

```
int main()
{
    // 整数リテラルは定数式
    int a[5] ;
```

```
// const修飾されていて、初期化式が定数式であるオブジェクトは定
// 数式
    int const n = 5 ;
    int b[n] ; // OK

    int m = 5 ; // これは定数式ではない
    int c[m] ; // エラー
}
```

注意:この解説は、C++14のドラフトN3797を参考にしている。正式なC++14規格では変更される可能性がある。また、C++11ではない。

ある式eを評価した際に、以下に挙げる式を評価しない場合、式eはコア定数式(core constant expression)である。

- this、ただし、eの一部として評価されるconstexpr関数かconstexprコンストラクタ一内を除く
- リテラルクラスのconstexprコンストラクター、constexpr関数、トリビアルデストラクターの暗黙の呼び出しを除く、関数の呼び出し
- 未定義のconstexpr関数か未定義のconstexprコンストラクターの呼び出し
- 実装の制約を超える式

実装の制約とは、仮引数やテンプレート仮引数の数の最大値のような、コンピューターが有限であることに起因する様々な制約。

- 未定義の挙動を含む処理

例えば、符号付き整数のオーバーフローや、一部のポインター演算や、ゼロ除算や、一部のシフト演算など。

- 5.4 ラムダ式
- lvalueからrvalueへの変換、ただし以下の場合を除く
 - 非volatileのglvalueの整数かenumの型が、初期化前か定数式による初期化後の非volatileのconstオブジェクトを参照する場合

文字列リテラルによる配列が該当する

- 非volatileのglvalueが、constexprで定義された非volatileオブジェクトか、そのようなオブジェクトの変更可能なサブオブジェクトを参照する場合
- 非volatileのリテラル型のglvalueが、式eの初期化にともなって寿命が開始した非volatileオブジェクトを参照する時
- lvalueからrvalueへの変換か、unionの使われていないメンバーかサブオブジェクトを参照するglvalueの変更
- リファレンス型の変数かデータメンバーを参照する名前、ただし、リファレンスが以下のいずれかの方法で初期化された後の場合を除く。

- 定数式で初期化された場合

- 式eの評価とともに寿命が始まったオブジェクトの非staticデータメンバー

- cv void *からオブジェクトへのポインター型への型変換
- dynamic_cast
- reinterpret_cast
- 疑似デストラクター呼び出し
- オブジェクトの変更、ただし、式eの評価とともに寿命が始まった非volatileオブジェクトを参照する非volatileのリテラル型のlvalueに対して適用されたものを除く
- ポリモーフィックなクラス型のglvalueをオペランドに指定したtypeid式
- new式
- delete式
- 結果が未規定となる比較演算子と等号、不等号演算子
- throw式

式が整数型かスコープなしenum型で、暗黙にprvalueに型変換でき、型変換した式がコア定数式であるような式を、整数定数式(integral constant expression)という。

このような整数定数式は、配列の宣言の添字や、ビットフィールドや、enum初期化子や、アライメントなどに使える。

T型に変換された定数式(converted constant expression)とは、暗黙にT型のprvalueに変換された式のこと、その変換された式がコア定数式で、暗黙の型変換には、ユーザ一定義型変換、lvalueからrvalueへの変換、整数のプロモーション、8.5.7 ナロー変換以外の整数の型変換だけが使われているものを言う。

変換された定数式は、new式、case式、内部型が固定されたenum初期化子、配列の範囲など、整数かenumの非型テンプレート実引数など、型変換が必要な定数式の文脈で使われる。

ここまで解説して、始めて定数式の定義ができるようになる。

定数式(constant expression)とはコア定数式のうち、glvalueのstaticストレージか、関数か、prvalueコア定数式である。

prvalueコア定数式の場合、その値のオブジェクトとサブオブジェクトが、以下の条件をすべて満たす場合のみ、定数式になる。

- リファレンス型の非staticデータメンバーはすべて、staticストレージ上のオブジェクトか、関数を参照していること
- オブジェクトかサブオブジェクトがポインター型の場合、その値は、staticストレージ上のオブジェクトへのアドレスか、staticストレージ上のオブジェクトを超えたアドレスか、関数のアドレスか、nullポインターであること

C++規格は、浮動小数点数の計算精度を規定していないため、C++の実装でコンパイル時と実行時で、浮動小数点数の計算結果が変わったとしても、その実装は規格準拠である。

```
bool f()
{
    char array[ 1 + int( 1 + 0.2 - 0.1 - 0.1 ) ] ; // コンパ
    イル時評価される
```

```

int size = 1 + int( 1 + 0.2 - 0.1 - 0.1 ) ; // 実行時評価
される可能性がある

return sizeof( array ) == size ;
}

```

このような関数fの戻り値がtrueかfalseか、C++規格は規定することができない。

リテラルクラス型の式が、整数定数式を必要とする文脈で用いられた場合、その式は文脈上、整数型かスコープなしenum型に暗黙に変換される。その場合に使われる変換関数は、constexprでなければならない。

6 文(Statements)

6.1 ラベル文(Labeled statement)

識別子 : 文
 case 定数式 : 文
 default : 文

文にはラベルを付けることができる。ラベルとは、その文を指す識別子である。文にラベルを付けるための文を、ラベル文(label statement)という。ラベル文には、必ず後続する文が存在しなければならない。

```

void f()
{
    label :
// エラー、ラベルに続く文がない
} // ブロックの終わり

void g()
{
// OK、ラベルに続く文がある
    label_1 : /* 式文 */ ;
    label_2 : { /* 複合文 */ } ;
}

```

識別子ラベル(identifier label)は、識別子を宣言する。この識別子は、goto文でしか使えない。

```
int main()
{
label_1 : ;
label_2 : ;
label_3 : ;

    goto label_2 ;
}
```

ラベルの識別子のスコープは、宣言された関数内である。ラベルを再宣言することはできない。ラベルの識別子は、宣言する前にgoto文で使うことができる。

```
// ラベルのスコープは、宣言された関数内
void f() { label_f : ; }
void g()
{
    goto label_f ; // エラー、この関数のスコープのラベルではない
}

// ラベルを再宣言することはできない
void h()
{
    label_h : ; // label_hの宣言
    label_h : ; // エラー、ラベルの再宣言はできない
}

// ラベルの識別子は、宣言する前にgoto文で使うことができる
void i()
{
// 識別子label_iは、この時点では、まだ宣言されていないが、使うことができる
    goto label_i ;
label_i ;
}
```

ラベルの識別子は、独自の名前空間を持つので、他の識別子と混同されることはない。

```
int main()
{
```

```

identifier : ; // ラベルの識別子
int identifier ; // 変数の識別子

goto identifier ; // ラベルの識別子が使われる
identifier = 0 ; // 変数の識別子が使われる
}

```

caseラベルとdefaultラベルは、switch文の中でしか使うことができない。

```

int main()
{
    switch(0)
    {
        case 0 : ;
        default : ;
    }
}

```

6.2 式文(Expression statement)

式_{opt} ;

式文(expression statement)とは、式を書く事のできる文である。文の多くは、この式文に該当する。式文は、セミコロン(;)を終端記号として用いる。式文は、書かれている式を評価する。

```

int main()
{
    0 ; // 式は0
    1 + 1 ; // 式は1 + 1

    // これは式文ではなく、return文
    return 0 ;
}

```

式文は、式を省略することもできる。式を省略した式文を、null文という。

```
/* 式を省略 */ ; // null文

;;;; // null文が四つ
;;;;;; // null文が八つ
```

null文は、評価すべき式がないので、何もしない文である。null文はたとえば、ブロックの終りにラベル文を書きたい場合や、for文やwhile文のようなループを、単に回したい場合などに、使うことができる。

```
int main()
{
    // 単にループを回すだけのfor文
    for ( int i = 0 ; i != 10 ; ++i ) ;

    label : ; // ラベル文には、後続する文が必要。
}
```

6.3 複合文、ブロック(Compound statement or block)

{ ひとつ以上の文_{opt} }

複合文、またはブロックという文は、文をひとつしか書けない場所に、複数の文を書くことができる文である。

```
void f( bool b )
{
    if ( b )
        /*ここにはひとつの文しか書けない*/ ;

    if ( b )
    {
        // いくらでも好きなだけ文を書くことができる。
    }
}
```

複合文は、ブロックスコープを定義する。

```
int main()
{ // ブロックスコープ
    { // 新たなブロックスコープ
    }
}
```

6.4 選択文(Selection statements)

選択文は、複数あるフローのうち、どれかひとつを選ぶ文のことである。

もし、選択文の中の文が、複合文ではなかった場合、その文を複合文で囲んだ場合と同じになる。

```
void f( bool b )
{
    if ( b )
        int x ;

    x = 0 ; // エラー、xは宣言されていない
}
```

このコードは、以下のコードと同等であるため、if文の次の式文で、xという名前を見つからない。

```
void f( bool b )
{
    if ( b )
    { int x ; }

    x = 0 ; // エラー、xは宣言されていない
}
```

条件について

条件： 式 宣言

条件には、式か宣言を書くことができる。条件は、if文やswitch文だけではなく、while文などでも使われる。

```
void f()
{
    if ( true ) ;
    if ( int x = 1 ) ;
}
```

条件に宣言を書くことができる理由は、コードを単純にするためである。

```
// 何か処理をして結果を返す関数
int do_something() ;

int main()
{
    int result = do_something() ;
    if ( result )
    {
        // 処理
    }
}
```

条件には宣言を書く事ができるため、以下のように書くことができる。

```
if ( int result = do_something() )
```

6.4.1 if文 (The if statement)

```
if ( 条件 ) 文
if ( 条件 ) 文 else 文
```

`if`文は、条件の値によって、実行すべき文を変える。

条件が`true`と評価された場合、一つ目の文が実行される。条件が`false`と評価された場合、`else`に続く二つ目の文が有るのならば、二つ目の文が実行される。

```
int main()
{
    if ( true ) // 一つ目の文が実行される
        /*一つ目の文*/ ;

    if ( false ) // 一つ目の文は実行されない
        /*一つ目の文*/ ;

    if ( true ) // 一つ目の文が実行される
        /*一つ目の文*/ ;
    else
        /*二つ目の文*/ ;

    if ( false ) // 二つ目の文が実行される
        /*一つ目の文*/ ;
    else
        /*二つ目の文*/ ;
}
```

`else`は、近い方の`if`文に対応する。

```
int main()
{
    if ( false ) // #1
        if ( true ) ; // #2

    else { } // #2のif文に対応するelse
}
```

インデントに騙されてはいけない。インデントを正しく対応させると、以下のようになる。

```
int main()
{
    if ( false ) // #1
        if ( true ) ; // #2
    else ; // #2のif文に対応するelse
```

}

このため、`else`のある`if`文の中に、さらに`if`文をネストさせたい場合は、内側の`if`文にも、`else`が必要である。

```
int main()
{
    if ( false ) // #1
        if ( true ) ; // #2
        else ; // #2のif文に対応するelse
    else ; // #1のif文に対応するelse
}
```

あるいは、ブロック文を使うという手もある。

```
int main()
{
    if ( false ) // #1
    { if ( true ) ;

        else ; // #1のif文に対応するelse
    }
```

6.4.2 switch文 (The switch statement)

switch(条件) 文

switch文は、条件の値によって、実行する文を選択する。

条件は、整数型か`enum`型、もしくは非`explicit`な変換関数を持つクラス型でなければならない。条件がクラス型の場合、整数型か`enum`型に型変換される。

```
struct C
{
    operator int(){ return 0 ; }
```

```

} ;

int main()
{
    switch(1) ; // OK
    C c ;
    switch(c) ; // OK、C::operator int()が呼ばれる

    switch(1.0) ; // エラー、浮動小数点数型は指定できない

    switch( static_cast<int>(1.0) ) ; // OK
}

```

switch文の中の文には、通常、複合文を指定する。複合文の中には、caseラベル文やdefaultラベル文を書く。

```

switch(1)
{
    case 1 :
        /* 処理 */ ;
    break ;

    case 2 :
        /* 処理 */ ;
    break ;

    default :
        /* 処理 */ ;
}

```

caseラベル文に指定する式は、整数の定数式でなければならない。また、同じswitch内で、caseラベル文の値が重複してはならない。

defaultラベル文は、switch文の中の文に、ひとつだけ書くことができる。

switch文が実行されると、まず条件が評価される。結果の値が、switch文の中にあるcaseラベルに対して、ひとつづつ比較される。もし、値が等しいcaseラベル文が見つかった場合、そのラベル文に実行が移る。

```

void f( int const value )
{
    switch( value )

```

```
{  
    case 1 :  
        std::cout << "Good morning." << std::endl ;  
    break ;  
    case 2 :  
        std::cout << "Good afternoon." << std::endl ;  
    break ;  
    case 3 :  
        std::cout << "Good evening." << std::endl ;  
    break ;  
}  
}  
  
int main()  
{  
    f( 1 ) ; // Good morning.  
    f( 2 ) ; // Good afternoon.  
    f( 3 ) ; // Good evening.  
}
```

条件と値の等しいcaseラベルが見つからない場合で、defaultラベルがある場合、defaultラベルに実行が移る。

```
void f( bool const value )  
{  
    switch( value )  
    {  
        case true :  
            std::cout << "true" << std::endl ;  
        break ;  
  
        default :  
            std::cout << "false" << std::endl ;  
        break ;  
    }  
}  
  
int main()  
{  
    f( true ) ; // true  
    f( false ) ; // false  
}
```

条件と値の等しいcaseラベルが見つからず、defaultラベルもない場合、switch内の文は実行されない。

```
int main()
{
    // switch内の文は実行されない
    switch( 0 )
    {
        case 999 :
            std::cout << "hello" << std::endl ;
            break ;
        case 123456 :
            std::cout << "hello" << std::endl ;
            break ;
    }
}
```

caseラベルとdefaultラベル自体には、文の実行を変更する機能はない。

```
void f( int const value )
{
    switch( value )
    {
        case 1 :
            std::cout << "one" << std::endl ;
        default :
            std::cout << "default" << std::endl ;
        case 2 :
            std::cout << "two" << std::endl ;
    }
}
```

この場合、valueの値が1の場合、case 1のラベル文に続く文も、すべて実行されてしまう。また、valueの値が1でも2でもない場合、defaultラベル文に続くcase 2のラベル文も、実行されてしまう。このため、switch内の実行を切り上げたい時点で、6.5.5.1 [break文](#)を書かなければならない。break文を書き忘れたことによる、意図しない文の実行は、よくあることなので、注意が必要である。なお、このことは、逆に利用することもできる。

```
void f( int const value )
{
```

```
switch( value )
{
    case 3 :
    case 5 :
    case 7 :
        /* 何らかの処理 */ ;
}
}
```

この例では、valueの値が3, 5, 7のいずれかの場合に、何らかの処理が実行される。

6.5 繰り返し文 (Iteration statements)

繰り返し文 (Iteration statements) は、ループを書くための文である。

繰り返し文の中の文は、暗黙的に、ブロックスコープを定義する。このブロックスコープは、文の実行のループ一回ごとに、出入りする。例えば、

```
while ( true )
    int i ;
```

という文は、以下のように書いたものとみなされる。

```
while ( true )
{ int i ; }
```

従って、繰り返し文の中の変数は、ループが回されるごとに、生成、破棄されることになる。

```
struct C
{
    C(){ std::cout << "constructed." << std::endl ; }
    ~C(){ std::cout << "destructed." << std::endl ; }
} ;

int main()
{
    while ( true )
```

```
{ // 生成、破棄を繰り返す  
    C c;  
}  
}
```

6.5.1 while文 (The while statement)

while (条件) 文

while文は、条件の結果がfalseになるまで、文を繰り返し実行する。条件は、文の実行前に、繰り返し評価される。

```
int main()  
{  
    // 一度も繰り返さない  
    while ( false )  
    {  
        std::cout << "hello" << std::endl ;  
    }  
  
    // 無限ループ  
    while ( true )  
    {  
        std::cout << "hello" << std::endl ;  
    }  
  
    // iが10になるまで繰り返す  
    int i = 0 ;  
    while ( i != 10 )  
    {  
        ++i ;  
    }  
}
```

条件が宣言である場合、変数のスコープは、while文の宣言された場所から、while文の最後までである。条件の中で宣言された変数は、文の実行が繰り返されるたびに、生成、破棄される。

```
while ( T t = x ) 文
```

という文は、

```
label:  
{  
    T t = x;  
    if (t)  
    {  
        文  
        goto label;  
    }  
}
```

と書くのに等しい。

while文の条件の中で宣言された変数は、ループの繰り返しのたびに、破棄されてから再び生成される。

```
#include <iostream>  
  
class nonzero  
{  
private :  
    int value ;  
  
public :  
    nonzero( int i )  
        : value(i)  
    { std::cout << "constructed" << std::endl ; }  
    ~nonzero()  
    { std::cout << "destructed" << std::endl ; }  
  
    operator bool() { return value != 0 ; }  
};  
  
int main()  
{  
    int i = 3 ;
```

```
while ( nonzero n = i )
{
    --i ;
}
}
```

6.5.2 do文(The do statement)

do 文 while (式) ;

do文の式は、boolに変換される。boolに変換できない場合、エラーとなる。

do文は、式の結果がfalseになるまで、文が繰り返し実行される。ただし、式の評価は、文の実行の後に行われる。

```
int main()
{
    // 一度だけ文を実行
    do {
        std::cout << "hello" << std::endl ;
    } while ( false ) ;

    // 無限ループ
    do {
        std::cout << "hello" << std::endl ;
    } while ( true ) ;
}
```

6.5.3 for文(The for statement)

for (for初期化文 条件_{opt} ; 式_{opt}) 文

for初期化文:

式文

宣言

for文は、for初期化文で、ループ前の初期化を書き、条件で、ループを実行するかどうかの判定を行い、文が実行されたあとに、そのつど式が評価される。

for文の実行では、まず、for初期化文が実行される。for初期化文は、式文か、変数の宣言を行うことができる。変数のスコープは、for文の最後までである。次に、文の実行の前に、条件が評価され、falseとなるまで文が繰り返し実行される。文の実行の後に、式が評価される。

```
for ( for初期化文 条件 ; 式 ) 文
```

は、以下のコードと同等である。

```
{  
    for初期化文  
    while ( 条件 )  
    {  
        文  
        式 ;  
    }  
}
```

ただし、文の中でcontinue文を使ったとしても、式は評価されるという違いがある。

for文は、while文でよく書かれるコードを書きやすくした構文である。例えば、while文を10回実行したい場合、

```
int main()  
{  
    // カウンター用の変数の宣言  
    int i = 0 ;  
  
    while ( i != 10 )  
    {  
        // 処理  
        ++i ;  
    }  
}
```

このようなコードを書く。for文は、このようなコードを、一度に書けるようにしたものである。

```
int main()
{
    for ( int i = 0 ; i != 10 ; ++i )
    {
        // 処理
    }
}
```

for文の条件と式は、省略することができる。条件を省略した場合、trueとみなされる。

```
int main()
{
    // 条件を省略、for ( ; true ; ) と同じ
    for ( ; ; );
}
```

6.5.4 range-based for文 (The range-based for statement)

ここでは、range-based forの言語機能を説明している。ライブラリとしてのレンジや、ユーザー定義のクラスでレンジをサポートする方法については、ライブラリの[レンジ](#)を参照。

6.5.4.1 range-based forの基本

```
for ( for-range-宣言 : for-range-初期化子 ) 文
```

range-based forは、レンジをサポートしている配列、初期化リスト、クラスの各要素に対して、それぞれ文を実行するための文である。

range-based forは、forに続けて、括弧を書く。括弧の中には、変数の宣言と、レンジとを、:で区切る。

```
int main()
{
    int a[] = { 1, 2, 3 } ;
    for ( int i : a ) ; // 各要素をint型のコピーで受ける
    for ( int & ref : a ) ; // 各要素をリファレンスで受ける
    for ( auto i : a ) ; // auto指定子を使った例
}
```

このようにして宣言した変数は、range-based for文の中で使うことができる。range-based for文は、変数をレンジの各要素で初期化する。

```
int main()
{
    int a[] = { 1, 2, 3 } ;
    for ( auto i : a )
    {
        i ;
    }
}
```

この例では、ループは3回実行され、変数iの値は、それぞれ、1, 2, 3となる。

ループを使ってコードを書く場合、配列やコンテナーの各要素に対して、それぞれ何らかの処理をするという事が多い。

```
#include <iostream>

int main()
{
    int a[5] = { 1, 2, 3, 4, 5 } ;
    for (
        int * iter = &a ; // 各要素を表す変数の宣言
        iter != &a + 5 ; // 終了条件の判定
        ++iter // 次の要素の参照
    )
    {
        // 各要素に対する処理
        std::cout << *iter << std::endl ;
    }
}
```

しかし、このようなループを正しく書くのは、至難の業である。なぜならば、人間は間違いを犯すからである。しかし、このようなループは、誰が書いても、概ね似たようなコードになる。range-based forを使えば、このような冗長なコードを省くことができる。

```
int main()
{
    int a[5] = { 1, 2, 3, 4, 5 } ;
    for ( auto i : a )
    {
        std::cout << i << std::endl ;
    }
}
```

range-based forは、極めて簡単に使うことができる。for-range宣言で、各要素を得るために変数を宣言する。for-range初期化子で、レンジをサポートした式を書く。文で、各要素に対する処理を書く。

```
int main()
{
    int a[5] = { 1, 2, 3, 4, 5 } ;
    for ( int & i : a )
    {
        i *= 2 ; // 二倍する
    }
}
```

この例では、配列aの各要素は、二倍される。配列の要素を書き換えるために、変数は参照で受けている。

range-based forには、配列の他にも、初期化リストや、レンジをサポートしたクラスを書くことができる。STLのコンテナーは、レンジをサポートしている。配列以外にrange-based forを適用する場合、<iterator>の#includeが必要である。

```
#include <iterator>

int main()
{
    // 配列
    int a[] = { 1, 2, 3 } ;
    for ( auto i : a )
    { std::cout << i << std::endl ; }
```

```
// 初期化リスト
for ( auto i : { 1, 2, 3 } )
{ std::cout << i << std::endl ; }

// クラス
std::vector<int> v = { 1, 2, 3 } ;
for ( auto i : v )
{ std::cout << i << std::endl ; }
}
```

6.5.4.2 range-based forの詳細

range-based forは、本来、コンセプトという言語機能と共に提供される予定であった。しかし、コンセプトは紆余曲折を経た結果、C++11では却下された。そのため、現行のrange-based forは、コンセプトではなく、ADLによる実装をされている。

以下のrange-based for文があるとする。

```
for ( for-range-宣言 : for-range-初期化子 ) 文
```

このrange-based for文は、以下のように変換される。

for-range-初期化子が式の場合、括弧でくくる。これは、コンマ式が渡されたときに、正しく式を評価するためである。

```
for ( auto i : a, b, c, d ) ;
// 括弧でくくる
for ( auto i : (a, b, c, d) ) ;
```

for-range-初期化子が初期化リストの場合、なにもしない。

```
{
    // 式の結果をlvalueかrvalueのリファレンスで束縛
    auto && __range = for-range-初期化子 ;
    for (
        auto __begin = begin式, // 先頭のイテレーター
        __end = end式 ; // 終端のイテレーター
```

```

    __begin != __end ; // 終了条件
    ++__begin ) // イテレーターのインクリメント
{
    for-range-宣言 = *__begin; // 要素を得る
    文
}
}

```

ここでの、`_range`、`_begin`、`_end`という変数は、説明のための仮の名前である。実際の range-based for文の中では、このような変数名は存在しない。

`_range`とは、for-range-初期化子の式の結果を保持するためのリファレンスである。`auto`指定子とrvalueリファレンスの宣言子が使われていることにより、式のlvalue、rvalue、CV修飾子をいかんを問わずに、結果をリファレンスとして束縛できる。

begin式とend式は、先頭と終端へのイテレーターを得るための式である。

for-range-初期化子の型が、配列の場合、begin式は「`_range`」となり、end式は、「`_range + 配列の要素数`」となる。

```

int x [10] ;
for ( auto i : x )
{
    // 処理
}

```

上記のrange-based for文は、以下のように変換される。

```

int x [10] ;
{
    auto && __range = ( x ) ;
    for (
        auto __begin = __range,
        __end = __range + 10 ;
        __begin != __end ;
        ++__begin )
    {
        auto i = *__begin;
        // 処理
    }
}

```

型が配列以外の場合、begin式は「begin(_range)」に、end式は「end(_range)」に変換される。

```
std::vector<int> v ;
for ( auto i : v )
{
    // 処理
}
```

```
std::vector<int> v ;
{
    // 式の結果をlvalueかrvalueのリファレンスで束縛
    auto && __range = ( v ) ;
    for (
        auto __begin = begin(__range),
        __end = end(__range) ;
        __begin != __end ;
        ++__begin )
    {
        auto i = *__begin;
        // 処理
    }
}
```

ここでbegin(_range)とend(_range)は、関数呼び出しである。ただし、この名前の解決には、通常の名前探索のルールは用いられない。begin/endの名前探索には、関連名前空間に特別にstdを加えた、ADLによってのみ名前探索される。通常のunqualified名前探索は用いられない。3.4.3 ADLの詳細については、詳しい説明を別に設けてあるので、そちらを参照。

6.5.5 ジャンプ文 (Jump statements)

ジャンプ文は、実行する文を無条件で変更するための文である。

6.5.5.1 break文 (The break statement)

```
break ;
```

break文は、繰り返し文かswitch文の中で使うことができる。break文は、最も内側の繰り返し文かswitch文から、抜け出す機能を持つ。もし繰り返し文かswitch文に続く、次の文があれば、実行はその文に移る。break文は、ループを途中で抜けたい場合に使うことができる。

```
int main()
{
    while ( true )
    {
        break ;
    }

    do
    {
        break ;
    } while ( true ) ;

    for ( ; ; )
    {
        break ;
    }

    switch(0)
    {
        default :
            break ;
    }
}
```

break文によって抜ける繰り返し文かswitch文とは、break文が書かれている場所からみて、最も内側の文である。

```
int main()
{
    while ( true ) // 外側
        while ( true ) // 内側
        {
            break ;
        }
}
```

break文が使われている内側の文からは抜けるが、外側の文から抜けることはできない。

6.5.5.2 continue文(The continue statement)

```
continue ;
```

continue文は、繰り返し文の中で使うことができる。continue文を実行すると、そのループの実行を中止する。

while文やdo文の場合、条件が評価され、その結果次第で、次のループが再び始まる。for文の場合は、ループの最後に必ず行われる式が、もしあれば評価され、条件が評価され、その結果次第で、次のループが再び始まる。

```
int main()
{
    while ( true )
    {
        continue ;
    }

    do
    {
        continue ;
    } while ( true ) ;

    for ( int i = 0 ; true ; ++i )
    {
        continue ; // for文の式である++iが評価される。
    }
}
```

continue文に対する繰り返し文とは、continue文が書かれている場所からみて、最も内側の繰り返し文のループである。

```
int main()
```

```
{
    while ( true ) // 外側
        while ( true ) // 内側
    {
        continue ;
    }

}
```

この例では、`continue`文は、内側の`while`文のループを中止する。ただし、`continue`文は`break`文とは違い、繰り返し文から抜け出すわけではないので、内側の`while`文の実行が続く。

6.5.5.3 return文 (The return statement)

```
return 式opt ;
return 初期化リスト；
```

`return`文は、関数の呼び出し元に実行を戻す文である。

```
int f()
{
    return 0 ; // OK
    return ; // エラー、戻り値がない
}
```

`return`文の式は、関数の呼び出し元に、戻り値として返される。式は関数の戻り値の型に暗黙的に変換される。変換できない場合はエラーとなる。

戻り値を返さない関数の場合、`return`文の式は省略できる。戻り値を返さない関数とは、戻り値の型が`void`型の関数、コンストラクター、デストラクターである。

```
struct C
{
    C() { return ; }
    ~C() { return ; }
} ;
```

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

戻り値を返さない関数の場合は、return文で戻り値を返してはならない。

```
void f()
{
    return ; // OK
    return 0 ; // エラー、関数fは戻り値を返さない
}
```

ただし、return文の式がvoidと評価される場合は、戻り値を返していることにはならない。

```
void f() { }
void g()
{
    // 関数fの呼び出しの結果は、void
    return f() ;
}
```

関数の本体の最後は、値を返さないreturn文が書かれたことになる。

```
void f()
{
// 値を返さないreturn文が書かれた場合と同じ
}
```

値を返す関数で、return文が省略された場合の挙動は未定義である。ただし、main関数だけは、特別に0が返されたものとみなされる。

```
// 値を返す関数
int f( bool b )
{
    if ( b )
        { return 0 ; }
// bがfalseの場合の挙動は未定義
}
```

```
int main()
{
// return 0 ;が書かれた場合と同じ
}
```

return文には、初期化リストを書くことができる。

```
std::initializer_list<int> f()
{
    return { 1, 2, 3 } ;
}

struct List
{
    List( std::initializer_list<int> ) { }
} ;

List g()
{
    return { 1, 2, 3 } ;
}
```

return文は、関数の戻り値の為に、一時オブジェクトを生成するかもしれない。一時オブジェクトを生成する場合、値はコピーかムーブをしなければならないが、return文では、コピーかムーブかの選択のために、式をrvalueとみなす可能性もある。式をrvalueとみなすということは、lvalueであっても、暗黙的にムーブされる可能性があることを意味する。これは例えば、「return文を実行して関数の呼び出し元に戻った場合、関数のローカル変数は破棄されるためムーブしてもかまわない」という状況で、コピーではなく、ムーブを選択できるようにするためにある。

```
// コピーとムーブが可能なクラス
struct C
{
    C() = default ; // デフォルトコンストラクター
    C( C const & ) = default ; // コピーコンストラクター
    C( C && ) = default ; // ムーブコンストラクター
} ;

C f()
{
    C c ;
```

```
// 一時オブジェクトが生成される場合、コピーかムーブが行われる。
return c ;
// なぜならば、ローカル変数はreturn文の実行後、破棄されるので、ムーブしても構わないからである。
}
```

また、上記のコードで、一時オブジェクトが生成されない場合もある。これはインライン展開やフロー解析などによる最適化の結果、コピーもムーブも行わなくてもよいと判断できる場合、そのような最適化を許可するためである。

6.5.5.4 goto文 (The goto statement)

```
goto 識別子 ;
```

goto文は、関数内のラベル文に無条件で実行を移すための文である。同じ関数内であれば、どこにでもジャンプできる。

```
int main()
{
label : ; // labelという名前のラベル文

    goto label ;
}
```

宣言文の前にジャンプする、あるいは、宣言文を飛び越すことについては、宣言文の項目で詳しく解説している。

6.6 宣言文 (Declaration statement)

```
ブロック宣言 ;
```

宣言文は、あるブロックの中に、新しい識別子を導入するための文である。ブロック宣言や、その他の宣言についての詳細は、宣言、宣言子、クラスを参照。

```

int main()
{
    int a ; // int型の識別子aという変数の宣言

    void f(void) ; // void (void)型の識別子fという関数の宣言

}

```

自動ストレージの有効期間を持つ変数は、宣言文が実行されるたびに、初期化される。また、宣言されているブロックから抜ける際に、破棄される。

```

struct Object
{
    Object()
    { std::cout << "constructed." << std::endl ; }
    ~Object()
    { std::cout << "destructed." << std::endl ; }
} ;

int main()
{
    {
        Object object ; // 生成
    } // ブロックスコープから抜ける際に破棄される
}

```

ジャンプ文を使えば、宣言文の後から前に実行を移すことが可能である。その場合、宣言文によって生成されたオブジェクトは破棄され、宣言文の実行と共に、再び生成、初期化される。

```

struct Object
{
    Object()
    { std::cout << "constructed." << std::endl ; }
    ~Object()
    { std::cout << "destructed." << std::endl ; }
} ;

int main()
{

```

```

label :
    Object object ; // 変数objectが生成、初期化される

    goto label ; // 変数objectは破棄される
}

```

この例では、Objectクラスの変数objectは、gotoで宣言文の前にジャンプするたびに、破棄されることになる。

goto文やswitch文などのジャンプ文を使えば、自動変数の宣言文を実行せずに、通り越すコードが書ける。

```

// goto文の例
void f()
{
    // labelという名前のラベル文にジャンプする
    goto label ;

    int value ; // 自動変数の宣言文

label : ;
// valueの宣言文を、実行せずに通り越してしまった。
}

// switch文の例
void g( int value )
{
    switch ( value )
    {
        int value ; // 変数の宣言文

        // 宣言文を飛び越えてしまっている。
        case 0 : break ;
        case 1 : break ;
        default : break ;
    }
}

```

このようなコードは、ほぼすべての場合、エラーとなるので、書くべきではない。では、変数の宣言文を通り越してもエラーとならない場合は何か。これは、相当の制限を受ける。まず、変数の型は、スカラー型か、trivialなデフォルトコンストラクターとtrivialなデストラクターを持つクラス型でなければならない。また、そのような型にCV修飾子を加えた型と、配列型でもよい。その上で、初期化子が存在していないなければならない。

```

struct POD { } ;
// trivialではないコンストラクターを持つクラス
struct Object { Object() {} } ;

int main()
{
    // 変数の宣言文を飛び越えるgoto文
    goto label ;

    // エラー
    // 変数の型はスカラー型だが、初期化子がある。
    int value = 0;

    int scalar ; // OK

    // エラー
    // 変数のクラス型がtrivialではないコンストラクターを持っている
    Object object ;

    POD pod ; // OK

label : ;
}

```

すべてのstatic変数とthread_local変数は、他のあらゆる初期化に先立って、ゼロ初期化される。

```

int main()
{
    goto label ;
    static int value ; // static変数は必ずゼロ初期化される
label :
    // この場合、valueは0であることが保証されている
    if ( value == 0 ) ;
}

```

ブロックスコープ内のstatic変数とthread_local変数は、定数初期化による早期の初期化が行われない場合、宣言に始めて処理が到達した際に、初期化される。

```
// 定数初期化できない型
struct X
{
    int member ;
    X( int value ) : member( value ) { }
} ;

void f()
{
    // xのゼロ初期化はすでに行われている
    static X x(123) ;
    // この時点で、xの初期化は完了している。
}
```

ブロックスコープ内のstatic変数とthread_local変数が定数初期化されている場合、実装は早期に初期化を行なってもかまわない。ただし、行われるという保証はない。

```
// 定数初期化できる型
struct X
{
    int member ;
    constexpr X( int value ) : member(value) { }
} ;

// 定数初期化できない型
struct Y
{
    int member ;
    Y( int value ) : member(value) { }
} ;

int g()
{
    goto label ; // 宣言文を飛び越してしまっている。

    // constexpr指定子が使われていないことに注意
    // xはstatic変数であり、constexprコンストラクターを使ってい
    るため、定数初期化である
    static X x( 123 ) ;
    // constexprコンストラクターを使っていないため、定数初期化で
    はない
    static Y y( 123 ) ;
```

```

label :
    // xは初期化されているかもしれないし、初期化されていないかもし
    // れない
    // yは初期化されていない
    // 両方とも、ゼロ初期化は保証されている
}

```

この例では、関数gのstaticローカル変数xとyの宣言文には、処理が到達しない。そのため、xとyが初期化されている保証はない。ただし、xは定数初期化なので、実装によっては、早期初期化されている可能性がある。ゼロ初期化だけは常に保証されている。

static変数とthread_local変数は、宣言文の実行のたびに初期化されることはない。

```

int f( int x )
{
    // 一回だけ初期化される
    // 定数初期化なので、いつ初期化されるかは定められていない
    // ただし、ゼロ初期化はすでに行われている
    static int value = 1 ;
    // この時点で、初期化は完了していることが保証されている

    int temp = value ;
    value = x ;

    return temp ;
}

int main()
{
    f(2) ; // 1
    f(3) ; // 2
    f(4) ; // 3
}

```

もし、static変数とthread_local変数の初期化が、例外がthrowされたことにより終了した場合は、初期化は未完了だとみなされる。そのような場合、次に宣言文を実行した際に、再び初期化が試みられる。

```

int flag = 0 ;

struct X

```

```

{
    X()
    {
        if ( flag++ == 0 )
            throw 0 ;
    }
}

void f()
{
    static X x ;
}

int main()
{
    try
    {
        f() ; // 関数fのstaticローカル変数xの初期化は未完了
    }
    catch ( ... ) { }
    f() ; // 関数fのstaticローカル変数xの初期化完了
}

```

もし、static変数とthread_local変数の宣言文の初期化が再帰した場合、挙動は未定義である。

```

int f( int i )
{
    static int s = f(2*i); // エラー、初期化が再帰している
    return i+1;
}

```

この例では、static変数sの初期化が終わらなければ、関数fはreturn文を実行できない。しかし、sの初期化は、再帰している。この場合、挙動は未定義である。

6.7 曖昧解決 (Ambiguity resolution)

関数形式のキャストを用いた式文と、宣言文とは、文法が曖昧になる場合がある。そ

の場合、宣言文だと解釈される。

```
int main()
{
    int x = 0 ;
// 不必要な括弧がついた宣言文？ それともキャスト？
    int(x) ;
}
```

この場合、`int(x)` ;という文は、キャストを含む式文ではなく、宣言文になる。したがって、上記の例は、変数xの再定義となるので、エラーである。

7 宣言 (Declarations)

宣言 (declaration) とは、名前がどのように解釈されるかを指定するための文法である。

宣言には、ブロック宣言 (block-declaration)、関数定義 (function-definition)、テンプレート宣言 (template-declaration)、明示的インスタンス化 (explicit-instantiation)、明示的特殊化 (explicit-specialization)、リンクージ指定 (linkage-specification)、名前空間定義 (namespace-definition)、空宣言 (empty-declaration)、アトリビュート宣言 (attribute-declaration) がある。

ブロック宣言には、単純宣言 (simple-declaration)、名前空間エイリアス定義 (namespace-alias-definition)、using宣言 (using-declaration)、usingディレクティブ (using-directive)、static_assert宣言 (static_assert-declaration)、エイリアス宣言 (alias-declaration)、opaque-enum宣言 (opaque-enum-declaration) がある。

7.1 単純宣言 (simple-declaration)

単純宣言 (simple-declaration) は、大きく分けて三つに分割できる。

アトリビュート指定子 指定子 宣言子 ;

変数や関数の宣言などは、この単純宣言で書かれることになる。

単純宣言のアトリビュート指定子は、宣言子のエンティティに属する。詳しくは、7.7 アトリビュートを参照。

指定子というのは、`int` や `class C`、`typedef`などを指す。指定子は複数指定できる。

宣言子は、変数や関数、型などを、ひとつ宣言する。宣言子も複数指定できる。

```
// int型の変数xの宣言
int // 指定子
x // 宣言子
;

// int const * const型の変数pの宣言
const int // 指定子
* const p // 宣言子
;

// typeという名前のint型を宣言。
typedef int // 指定子
type // 宣言子
;
```

宣言子を複数指定できることには、注意が必要である。例えば、ひとつの宣言文で、複数の変数を宣言することもできる。

```
// int型で、それぞれa, b, c, dという名前の変数を宣言
// 宣言子は4個
int a, b, c, d ;
```

これは、比較的分かりやすい。しかし、ポインターや配列、関数などという型は、宣言子で指定するので、ひとつの宣言文で、複数の宣言子を使うということは、非常に読みにくいコードを書く事もできてしまうのである。

```
int * a, b, c[5], (*d)(void) ;
```

この文は非常に分かりにくい。この文を細かく区切って解説すると、以下のようになる。

```
int // 指定子
* a, // int *型の変数
b, // int型の変数
c[5], // int [5]型の変数
(*d)(void) // int(*)(void)型の変数、引数を取らずint型の戻り値を
返す関数ポインター
; 
```

ひとつの宣言文で複数の宣言子を書くことは避けるべきである。

7.2 static_assert宣言 (static_assert-declaration)

```
static_assert ( 定数式 , 文字列リテラル ) ;
```

static_assert宣言は、条件付きのコンパイルエラーを引き起こすための宣言である。static_assertの定数式はboolに変換される。結果がtrueならば、何もしない。結果がfalseならば、コンパイルエラーを引き起こす。いわば、コンパイル時のassertとして働くのである。結果がfalseの場合、C++のコンパイラは文字列リテラルをエラーメッセージとして、何らかの方法で表示する。

```
static_assert( true, "" ) ; // コンパイルが通る
static_assert( false, "" ) ; // コンパイルエラー

// コンパイルエラー
// 何らかの方法で、helloと表示される。
static_assert( false, "hello" ) ;
```

具体的な利用例としては、今、int型のサイズが4バイトであることを前提としたコードを書きたいとする。このコードは当然ながらポータビリティがない。そこで、int型のサイズが4バイトではない環境では、コンパイルエラーになってほしい。これは、以下のように書ける。

```
static_assert( sizeof(int) == 4, "sizeof(int) must be 4" ) ;
```

sizeof(int)が4ではない環境のC++のコンパイラでは、このコードはコンパイルエラーになる。また、文字列リテラルが、何らかの方法で表示される。

また別の例では、以下のような関数があるとする。

```
// 仕様 : Derived型はBase型から派生されていること
template < typename Base, typename Derived >
void f( Base base, Derived derived )
{
// 処理
}
```

この関数では、Derivedという型は、Baseという型から派生されていることを前提とした処理を行う。そこで、もしユーザーがうっかり、そのような要求を満たさない型を渡した場合、エラーになって欲しい。これは、以下のように書ける。

```
#include <type_traits>

template < typename Base, typename Derived >
void f( Base base, Derived derived )
{
    static_assert(
        !std::is_same<Base, Derived>::value // 同じ型でなければtrue
        && std::is_base_of<Base, Derived>::value // DerivedがBaseから派生されていればtrue
        , "Derived must derive Base." ) ;

    // 処理
}

struct Base { } ;
struct Derived : Base { } ;

int main()
{
    Base b ; Derived d ;

    f(b, d) ; // OK
    f(b, b) ; // エラー
}
```

このように、テンプレート引数の型が、あらかじめ定められた要求を満たしていない場合、`static_assert`を使ってコンパイルエラーにすることもできる。

`static_assert`の文字列リテラルには、2.3 [基本ソース文字セット](#)を使うことができる。C++の実装は、基本ソース文字セット以外の文字を、エラーメッセージとして表示する義務がない。我々日本人としては、日本語を使いたいところだが、すべてのコンパイラに日本語の文字コードのサポートを義務づけるのが現実的ではない。そのため規格では、現実的に最低限保証できる文字しかサポートを義務づけていない。もちろん、コンパイラが`static_assert`の日本語表示をサポートするのは自由である。しかし、サポートする義務がない以上、`static_assert`の文字列リテラルに基本ソース文字セット以外の文字を使うのは、ポータビリティ上の問題がある。

```
// 文字列リテラルが表示されるかどうかは実装依存
static_assert( sizeof(int) == 4, u"このコードはint型のサイズは4" );
```

であることを前提にしている") ;

7.3 指定子(Specifiers)

指定子には、ストレージクラス指定子、関数指定子、`typedef`指定子、`friend`指定子、`constexpr`指定子、型指定子がある。

指定子は、組み合わせて使うことができる場合もある。例えば、`typedef`指定子と型指定子は、組み合わせて使うことができる。その際、指定子の順番には、意味が無い。以下の2行のコードは、全く同じ意味である。

```
// int型の別名typeを宣言  
// typedefはtypedef指定子、intは型指定子、typeは宣言子  
typedef int type ;  
int typedef type ;
```

もちろん、指定子と宣言子は違うので、以下はエラーである。

```
// エラー、*は宣言子。宣言子の後に指定子を書く事はできない  
int * typedef type ;
```

7.3.1 ストレージクラス指定子(Storage class specifiers)

ストレージクラス指定子には、`register`、`static`、`thread_local`、`extern`、`mutable`がある。

ひとつの宣言の中には、ひとつのストレージクラス指定子しか書く事はできない。つまり、ストレージクラス指定子同士は、組み合わせて使う事はできない。ただし、`thread_local`だけは、`static`や`extern`と併用できる。

7.3.1.1 `register`指定子

`register`指定子を使ってはならない。`register`は、変数への最適化のヒントを示す目的で導入された。これは、まだハードウェアが十分に高速でないので、賢いコンパイラを実装できなかった当時としては、意味のある機能であった。しかし、現在では、ハードウェアの性能の向上により、コンパイラはより複雑で高機能な実装ができるようになり、`register`は単に無視されるものとなってしまった。

`register`は歴史的理由により存在する。この機能は、現在では互換性のためだけに残

されている機能であり、使用を推奨されていない。また、将来的には廃止されるだろう。

7.3.1.2 `thread_local`指定子

`thread_local`指定子のある変数は、スレッドストレージの有効期間を持つ。すなわち、`thread_local`が指定された変数は、スレッドごとに別のオブジェクトを持つことになる。

`thread_local`指定子は、名前空間スコープかブロックスコープの中の変数と、`static`データメンバーに対して適用することができる。ブロックスコープの変数に`thread_local`が指定された場合は、たとえ`static`指定子が書かれていなくても、暗黙的に`static`と指定されることになる。

正しい例

```
// グローバル名前空間のスコープ
thread_local int global_variable ;

// 名前の付いている名前空間のスコープ
namespace perfect_cpp
{
    thread_local int variable ;
}

// ブロックスコープ
void f()
{
    // 以下の3行は、すべてthread_localかつstaticな変数である。
    thread_local int a ;
    thread_local static int b ;
    static thread_local int c ;
}

struct C
{
    // 以下の2行は、すべてthread_localなstaticデータメンバーである。
    static thread_local int a ;
    thread_local static int b ;
} ;
```

`thread_local`指定子は、`static`データメンバーにしか指定できないということには、注意を要する。データメンバーが`static`データメンバーとなるには、`static`指定子がなければな

らない。ブロックスコープ内の変数とは違い、暗黙のうちにstaticが指定されたことにはならない。

```
struct C
{
    // エラー、thread_localは非staticデータメンバーには適用できない。
    thread_local int a ;
} ;
```

thread_localが指定された変数に対する、同じ宣言は、すべてthread_local指定されなければならない。

```
// 翻訳単位 1
thread_local int value ;

// 翻訳単位 2
extern thread_local int value ;

// 翻訳単位 2
extern int value ; // エラー、thread_localが指定されていない
```

7.3.1.3 static指定子

static指定子には、変数をstatic変数にするという機能と、名前を内部リンクージにするという機能がある。static指定子は、変数と関数と無名unionに指定することができる。ただし、ブロックスコープ内の関数宣言と、関数の仮引数に指定することはできない。

```
struct C
{
    // staticデータメンバー
    static int data ;
    // staticメンバー関数
    static void f() {}
} ;

int main()
{
```

```
// 変数、static変数になる
static int x ;

// 無名union、static変数になる
static union { int i ; float f ; } ;
}
```

static指定子が指定された変数は、[静的ストレージの有効期間を持つ](#)。ただし、`thread_local`指定子も指定されている場合は、[スレッドストレージの有効期間を持つ](#)。

クラスのメンバーに対するstatic指定子については、9.8 staticメンバーを参照。

static指定子とリンクエージの関係については、3.5 プログラムとリンクエージを参照。

名前空間スコープにおける、リンクエージ指定目的でのstaticの使用は、無名名前空間で代用した方がよい。この機能は、C++11で非推奨にされるはずだったが、直前で見直された。理由は、既存のコードを考えると、この機能を将来的に廃止することはできないからである。

```
// グローバル名前空間スコープ
// 内部リンクエージの指定
static int x ;
static void f() {}

// 無名名前空間を使う
namespace
{
    int x ;
    void f() {}
}
```

7.3.1.4 [extern指定子](#)

extern指定子には、名前のリンクエージを外部リンクエージにするという機能と、名前の定義をしないという機能がある。extern指定子は、変数と関数に適用できる。ただし、クラスのメンバーと関数の仮引数には指定できない。

```
// 変数
extern int i ;
// 関数
```

```
extern void f() ;
```

extern指定子と、宣言と定義の関係については、3.1 [宣言と定義](#)を参照。

extern指定子とリンクエージの関係については、3.5 [プログラムとリンクエージ](#)を参照。

テンプレートの14.6.2 [明示的なインスタンス化](#)と、7.6 [リンクエージ指定](#)は、externキーワードを使うが、指定子ではない。

7.3.1.5 mutable指定子

mutable指定子は、constでもstaticでもないクラスのデータメンバーに適用することができる。mutable指定子の機能は、クラスのオブジェクトへのconst指定子を、無視できることである。これにより、constなメンバー関数から、データメンバーを変更することができる。

```
class C
{
private:
    mutable int value ;

public :
    void f() const
    {
        // 変更できる
        value = 0 ;
    }
} ;

int main()
{
    C c ;
    c.f() ;
}
```

mutableの機能について詳しくは、7.3.6.1 [CV修飾子](#)も参照。

7.3.2 関数指定子 (Function specifiers)

関数指定子(Function specifier)には、`inline`、`virtual`、`explicit`がある。

7.3.2.1 `inline`指定子

`inline`指定子が書かれた関数宣言は、インライン関数(inline function)を宣言する。
`inline`指定子は、この関数をインライン展開することが望ましいと示すための機能である。ただし、インライン関数だからといって、必ずしもインライン展開されるわけではない。インライン関数ではなくても、インライン展開されることもある。あくまで最適化のヒントに過ぎない。

```
// インライン関数
inline void f() { }
```

クラス定義の中の関数定義は、`inline`指定子がなくても、自動的にinline関数になる。

```
struct C
{
    // 関数定義、インライン関数である
    void f() {}
    // 関数の宣言、インライン関数である
    inline void g() ;
    // 関数の宣言、インライン関数ではない
    void h() ;
};

// 関数C::gの定義
inline void C::g() { }

// 関数C::hの定義
void C::h() { }
```

インライン指定子は、関数のリンクエージに何の影響も与えない。インライン関数のリンクエージは、通常の関数と同じである。すなわち、`static`指定子があれば内部リンクエージ持つ。そうでなければ外部リンクエージを持つ。

```
// 外部リンクエージを持つ
inline void f() {}
// 内部リンクエージを持つ
inline static void g() {}
```

ただし、インライン関数は、外部リンクエージを持っていたとしても、通常の関数とは異なる扱いを受ける。これは、インライン展開の実装を容易にするための制約である。インライン関数は、その関数を使用するすべての翻訳単位で、「定義」されていなければならない。インライン関数の定義は、すべての翻訳単位で、まったく同一でなければならない。

```
// 翻訳単位 1
// translation_unit_1.cpp

// 外部リンクエージを持つインライン関数の定義
inline void f() {}
inline void g() {}
```

```
// 翻訳単位 2
// translation_unit_2.cpp

// 宣言だけ
inline void f() ;

// 定義
inline void g() {}

// 関数の宣言
int main()
{
    // エラー
    // この翻訳単位に関数fの定義がない
    f() ;

    // OK、定義もある
    g() ;
}
```

これは、テンプレートと同じような制限となっている。そのため、外部リンクエージを持つインライン関数は、通常、ヘッダーファイルに書き、必要な翻訳単位で#includeする。まったく同一ということに関して、詳しくは、3.2 ODR(One definition rule)を参照。

ただし、翻訳単位に定義があればいいので、呼び出す場所では、宣言だけだとしても、問題はない。

```
// 宣言
```

```
inline void f() ;  
  
int main()  
{  
    // すでに名前fは宣言されていて、この翻訳単位に定義がある  
    f() ; // OK  
}  
  
// 定義  
inline void f() {}
```

7.3.2.2 [virtual指定子](#)

virtual指定子は、クラスの非staticメンバー関数に指定することができる。詳しくは、10.3 [virtual関数](#)を参照。

7.3.2.3 [explicit指定子](#)

explicit指定子は、クラス定義内のコンストラクターと変換関数に指定することができる。詳しくは、12.3.1 [コンストラクターによる型変換](#)と、12.3.2 [変換関数\(Conversion functions\)](#)を参照。

7.3.3 [typedef指定子\(The typedef specifier\)](#)

typedef指定子は、型の別名を宣言するための指定子である。この別名のことを、typedef名(typedef-name)という。typedef名は、型と同じように扱われる。

```
typedef int integer ;  
// これ以降、typedef名integerは、int型とおなじように使える。  
  
integer main()  
{  
    integer x = 0 ;  
    integer y = x ;
```

}

typedef名は、エイリアス宣言(alias-declaration)で宣言することもできる。

```
using 識別子 = 型名 ;
```

```
using integer = int ;
```

エイリアス宣言では、usingキーワードに続く識別子が、typedef名となる。typedef指定子によって宣言されたtypedef名と、エイリアス宣言によって宣言されたtypedef名は、全く同じ意味を持つ。そのため、本書で「typedef名」と記述されている場合、それはtypedef指定子による宣言であろうと、エイリアス宣言による宣言であろうと、等しく適用される。一方、「typedef指定子」と記述されている場合、エイリアス宣言には当てはまらない。

エイリアス宣言の文法は、typedefより分かりやすい。例えば、関数ポインターの別名を宣言したいとする。

```
// 同じ意味  
typedef void (*type)(void) ;  
using type = void (*)(void) ;
```

typedef指定子は、指定子であるので、単純宣言と同じ文法で名前を宣言しなければならない。using宣言は、名前を先に書き、その後に、純粋な型名を書くことができる。

エイリアス宣言とテンプレートについては、14.4.9 テンプレートエイリアスを参照。

typedef指定子は、クラス以外の同じスコープ内で、同じ型のtypedef名を再宣言することができます。

```
typedef int I ;  
typedef int I ; // OK、同じ型の再宣言  
typedef short I ; // エラー、型が違う  
  
void f()  
{  
    typedef short I ; // OK、別のスコープなので別の宣言  
}  
  
struct Class_Scope
```

```
{  
    typedef int type ;  
    typedef int type ; // エラー、クラススコープ内では、同じ型で  
    // も再宣言できない  
};
```

typedef名とconstの関係は、一見して分かりにくい。

```
typedef int * type ;  
  
// aの型はint const *  
const int * a ;  
// bの型は、int * const  
const type b ;
```

これは、指定子と宣言子との違いによる。

```
const int // 指定子  
* a // 宣言子  
;  
  
const type // 指定子、typeの型は int *  
b // 宣言子  
;
```

変数aの場合、const intへのポインター型となる。変数bの場合、const type型となる。typeの型は、int *なので、int *へのconst型となる。そのため、違う型となる。

7.3.4 friend指定子(The friend specifier)

friend指定子については、11.3 [friend](#)を参照。

7.3.5 constexpr指定子(The constexpr specifier)

constexpr指定子は、constexprの制約を満たした、変数の定義、関数と関数テンプレート

トの宣言、staticデータメンバーの宣言に対して指定できる。

```
constexpr int value = 0 ;
```

constexpr指定子を使って定義され、定数式で初期化された変数は、定数式として使うことができる。

```
void f()
{
    constexpr std::size_t size = 10 ;
    int a[size] ;
}
```

constexpr指定子を使う変数の型は、リテラル型でなければならない。

```
struct literal
{
    int a ;
} ;

struct non_literal
{
    non_literal() { }
} ;

int main()
{
    constexpr literal a{} ; // OK
    constexpr non_literal b{} ; // エラー
}
```

コンストラクター以外の関数にconstexpr指定子を記述すると、その関数は、constexpr関数(constexpr function)となる。コンストラクターにconstexpr指定子を記述すると、そのコンストラクターは、constexprコンストラクター(constexpr constructor)となる。constexpr関数とconstexprコンストラクターは暗黙にinlineになる。

constexpr関数の定義は、以下の条件を満たさなければならない。

- virtual関数でないこと
- 戻り値の型がリテラル型であること
- 仮引数の型がリテラル型であること

- 関数の本体は、= deleteか、= defaultか、複合文であること。
複合文として使える文は、以下のものだけである。
 - null文
 - static_assert宣言
 - typedef宣言とエイリアス宣言で、クラスやenumを定義しないもの
 - using宣言
 - usingディレクトィブ
 - return文ひとつ
- 戻り値を初期化する際のコンストラクター呼び出しや、暗黙の型変換は、すべて定数式でなければならない。

以下は合法なconstexpr関数の例である。

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

constexpr int g( int x, int y )
{
    return x + y + f() ;
}

constexpr int h( unsigned int n )
{
    return n == 0 ? 0 : h( n - 1 ) ;
}
```

以下は、constexpr関数の制約を満たさない誤ったコードの例である。

```
// エラー、使えない文の使用
constexpr int f( )
{
    constexpr int x = 0 ;
    return x ;
}

// エラー、使えない文の使用
constexpr int g( bool b )
{
    if ( b )
        return 1 ;
```

```

    else
        return 2 ;
}

// エラー、return文がふたつ
constexpr int h()
{
    return 0 ;
    return 0 ;
}

// エラー、戻り値の型がリテラル型ではない
struct S{ S(){ } } ;
constexpr S i()
{
    return S() ;
}

```

C++11のconstexpr関数の制約はとても厳しい。C++14では、この制約は大幅に緩和される。

constexprコンストラクターの定義は、仮引数の型がリテラルでなければならない。関数の本体は、= deleteか、= defaultか、複合文でなければならない。複合文は以下の制約を満たさなければならない。

- クラスはvirtual基本クラスを持たないこと
- 関数の本体は関数tryブロックではないこと
- 関数の本体の複合文は、以下のいずれかしか含まないこと
 - null文
 - static_assert宣言
 - typedef宣言とエイリアス宣言で、クラスやenumを定義しないもの
 - using宣言
 - usingディレクトィブ
- クラスの非staticデータメンバーと、基本クラスのサブオブジェクトは、すべて初期化されること
- 非staticデータメンバーと基本クラスのサブオブジェクトの初期化に関わるコンストラクターは、constexprコンストラクターであること
- 非staticデータメンバーに指定された初期化句は定数式であること

```

struct S
{
    int member = 0 ; // 定数式であること

    constexpr S() { }
}

```

```
} ;
```

- コンストラクターの実引数を仮引数の型に変換する際の型変換は、定数式であること

`constexpr`コンストラクターは、ユーザー定義の初期化を記述したリテラル型のクラスを書くことができる。

```
struct point
{
    int x ;
    int y ;

    constexpr S( int x, int y )
        : x(x), y(y)
    { }
};
```

このようなリテラル型のクラスは、`constexpr`指定子を使った変数で使える。

```
constexpr S s( 10, 10 ) ;
```

どのような実引数(無引数関数も含む)を与えても、`constexpr`が定数式にならない場合、エラーとなる。

```
// OK、定数式になる実引数がある
constexpr int f( bool b )
{
    return b ? throw 0 : 0 ;
}

// エラー、絶対に定数式にならない。
constexpr int g( )
{
    throw ;
}
```

`constexpr`関数テンプレートや、クラステンプレートの`constexpr`メンバー関数のインスタンス化された特殊化が、`constexpr`関数の制約を満たさない場合、そのような関数やコンストラクターは、`constexpr`関数、`constexpr`コンストラクターとはみなされない。

```

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

struct non_literal { non_literal(){ } } ;

int main()
{
    f( 0 ) ; // OK、constexpr関数

    non_literal n ;
    f( n ) ; // OK、ただし通常の関数
}

```

constexpr関数を呼び出した結果の値は、同等だがconstexprではない関数を呼び出した結果の値と等しくなる。

コンストラクターを除く非staticメンバー関数にconstexpr指定子を使うと、そのメンバー関数はconst修飾される。

```

struct S
{
    constexpr int f() ; // constになる
} ;

```

これは、以下のように書くのと同等である。

```
constexpr int f() const ;
```

constexpr指定子は、これ以外には関数の型に影響を与えない。

constexpr非staticメンバー関数を持つクラスは、リテラル型でなければならない。

```

// OK、リテラル型
struct literal
{
    constexpr int f() { return 0 ; }
}

```

```

} ;

// エラー、リテラル型ではない
struct non_literal
{
    non_literal() { }
    constexpr int f() { return 0 ; }
} ;

```

constexpr指定子が変数定義に使われた場合、変数はconstになる。変数の型はリテラル型でなければならず、また初期化されなければならない。

```

constexpr int a = 0 ;

// エラー、初期化されていない
constexpr int b ;

struct non_literal { non_literal() { } } ;
// エラー、リテラル型ではない
constexpr non_literal c{ } ;

```

初期化にコンストラクター呼び出しが行われる場合、コンストラクター呼び出しは定数式でなければならない。

7.3.6 型指定子(Type specifiers)

型指定子(type specifier)には、**クラス指定子**、**enum指定子**、**単純型指定子(simple-type-specifier)**、**複雑型指定子(elaborated-type-specifier)**、**typename指定子**、**CV修飾子(cv-qualifier)**がある。

クラス指定子は9 [クラス](#)で、enum指定子は7.4 [enumの宣言](#)で、typename指定子は、テンプレートの14.5 [名前解決](#)を参照。

型指定子は、一部を除いて、宣言の中にひとつだけ書くことができる。組み合わせることのできる型指定子は、以下の通りである。

constは、**const**以外の型指定子と組み合わせることができる。**volatile**は、**volatile**以外の型指定子と組み合わせることができる。

signedと**unsigned**は、**char**, **short**, **int**, **long**を後に書くことができる。

`short`と`long`は、`int`を後に書くことができる。

`long`は、`double`を後に書くことができる。`long`は、`long`を後に書くことができる。

```
long double a = 0.01 ;
long long int b = 0 ;
```

7.3.6.1 CV修飾子 (The cv-qualifiers)

CV修飾子 (cv-qualifier) は、指定子の他に、ポインターの宣言子にも使うことができる。CV修飾子には、`const`と`volatile`がある。この二つのキーワードの頭文字をとって、CV修飾子と呼ばれている。CV修飾子付きの変数は、必ず初期化子が必要である。CV修飾子がオブジェクトに与える影響については、基本事項の3.9.3 [CV修飾子](#)を参照。

```
const int a = 0 ;
volatile int b = 0 ;
const volatile int c = 0 ;

const int d ; // エラー、初期化子がない
```

指定子の始めに述べたように、指定子の順番に意味はないので、`const int`と`int const`は、同じ意味となる。

CV修飾子付きの型へのポインターやリファレンスは、必ずしも、CV修飾子付きのオブジェクトを参照する必要はない。ただし、CV修飾子が付いているように振舞う。

```
int main()
{
    int x = 0 ; // 非constなオブジェクト
    x = 0 ; // 変更できる

    int const & ref = x ; // 参照する
    ref = 0 ; // エラー、変更できない
}
```

7.3.6.2 単純型指定子 (Simple type specifiers)

単純型指定子には、基本型、クラス名、enum名、typedef名、auto指定子、decltype指定子を使うことができる。

基本型については、3.9.1 基本型を参照。

注意すべきこととしては、signedやunsignedは、単体で使われると、int型だとみなされる。

```
// signed int  
signed a = 0 ;  
// unsigned int  
unsigned b = 0 ;
```

また、shortやlongやlong longは、それぞれintが省略されたものとみなされる。

```
// short int  
short a = 0 ;  
// long int  
long b = 0 ;  
// long long int  
long long c = 0 ;
```

7.3.6.3 複雑型指定子(Elaborated type specifiers)

複雑型指定子の複雑(Elaborated)というのは、あまりふさわしい訳語ではないが、本書では便宜上、elaboratedに対し、複雑という訳語を使用する。class、struct、union、enumなどのキーワードを使った型指定子を指す。

```
struct StructName { } ;  
class ClassName { } ;  
union UnionName { } ;  
enum struct EnumName { value } ;  
  
int main()  
{  
    {  
        // 複雑型指定子  
        struct StructName a ;
```

```

    class ClassName b ;
    union UnionName c ;
    enum EnumName d = EnumName::value ;
}

{
    // 単純型指定子
    StructName a ;
    ClassName b ;
    UnionName c ;
    EnumName d = EnumName::value ;
}
}

```

識別子に対するキーワードは、enumにはenumキーワードを、unionにはunionキーワードを、クラスにはstructキーワードかclassキーワードを、すでに行われた宣言と一致して使わなければならない。

```

class Name { } ;
struct Name a ; // OK、structキーワードでもよい

enum Name b ; // エラー、キーワードが不一致
union Name c ; // エラー、キーワードが不一致

```

7.3.6.4 auto指定子 (auto specifier)

ここでは、変数の宣言に対するauto指定子について説明する。関数の宣言に対するauto指定子については、宣言子の8.3.5 関数を参照。また、5.6.4 new式にも、似たような機能がある。

変数を宣言する際、型指定子にautoキーワードを書くと、変数の型が、初期化子の式から推定される。

```

auto a = 0 ; // int
auto b = 0L ; // long
auto c = 0.0 ; // double
auto d = 0.0L ; // long double

```

もちろん、単なるリテラルだけにはとどまらない。およそ初期化子に書ける式ならば、何でも使うことができる。

```
int f() { return 0 ; }

bool g(int){ return true ; }
char g(double){ return 'a' ; }

int main()
{
    auto a = f() ; // int

    // もちろん、オーバーロード解決もされる
    auto b = g(0) ; // bool
    auto c = g(0.0) ; // char

    auto d = &f ; // int (*)(void)
}
```

auto指定子は、冗長な変数の型の指定を省くためにある。というのも、初期化子の型は、コンパイル時に決定できるので、わざわざ変数の型を指定するのは、冗長だからだ。また、変数の型を指定するのが、非常に面倒な場合もある。

```
#include <vector>
#include <string>

int main()
{
    std::vector< std::string > v ;
    // 型名が長くて面倒
    std::vector< std::string >::iterator iter1 = v.begin() ;

    // 簡潔に書ける
    auto iter2 = v.begin() ;
}
```

この場合では、`std::vector< std::string >::iterator`型の変数を宣言している。auto指定子を使わないと、非常に冗長になってしまう。

```
template < typename T1, typename T2 > struct Add { } ;
template < typename T1, typename T2 > struct Sub { } ;
```

```

template < typename T1, typename T2 >
Add< T1, T2 > operator + ( T1, T2 )
{
    typedef Add< T1, T2 > type ;
    return type() ;
}

template < typename T1, typename T2 >
Sub< T1, T2 > operator - ( T1, T2 )
{
    typedef Sub< T1, T2 > type ;
    return type() ;
}

struct A { } ;
struct B { } ;

int main()
{
    A a ; B b ;
    auto result = a + b - b + (b - a) ;
}

```

この場合、resultの型を明示的に書こうとすると、以下のようになる。これはとてもではないが、まともに書く事はできない。

```
Add< Sub< Add< A, B>, B>, Sub< B, A> > result = a + b - b +
(b - a) ;
```

auto指定子による変数の宣言では、変数の型は、関数のテンプレート実引数の推定と同じ方法で推定される。

```
auto u = expr ;
```

という式があったとすると、変数uの型は、

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

このような関数を、`f(expr)`と呼び出した場合の、テンプレート仮引数`U`と同じ型となる。

ただし、`auto`指定子では、初期化子が初期化リストであっても、型を推定できるという違いがある。

```
// std::initializer_list<int>
auto a = { 1, 2, 3 } ;
// std::initializer_list<double>
auto b = { 1.0, 2.0, 3.0 } ;

// エラー、型を推定できない
auto c = { 1, 2.0 } ;
auto d = { } ;

// OK、明示的なキャスト
auto e = std::initializer_list<int>{ 1, 2.0 } ;
auto f = std::initializer_list<int>{ } ;
```

テンプレートの実引数推定と同じ方法で型を推定するために、配列型は配列の要素へのポインターに、関数型は関数ポインター型になってしまう。これには注意が必要である。

```
void f() { }

int main()
{
    int a[1] ;
    auto t1 = a ; // int *
    auto t2 = f ; // int (*)(void)
}
```

`auto`指定子は、他の指定子や、`CV`修飾子、宣言子と組み合わせることもできる。

```
int const expr = 0 ; // exprの型はint const
auto a = expr ; // int
auto const b = expr ; // int const
auto const & c = expr ; // int const &
auto const * d = &expr ; // int const *

static auto e = expr ; // static指定子付きのint型の変数
```

この際の型の決定も、関数のテンプレート実引数の推定と同じルールになる。

宣言子と初期化子の型が合わない場合は、エラーとなる。

```
auto & x = 0 ; // エラー
```

この例では、xの型は、リファレンス型であるが、初期化子の型は、リファレンス型ではない。そのため、エラーとなる。

宣言子がrvalueリファレンスの場合、注意を要する。auto指定子の型は、テンプレート実引数の推定と同じ方法で決定されるので、lvalueリファレンスになることもある。

```
int main()
{
    int x = 0 ;

    int && r1 = x ; // エラー、rvalueリファレンスをlvalueで初期化できない

    auto && r2 = x ; // OK、ただし、r2の型はint &
    auto && r3 = std::move(x) ; // OK、r3の型はint &&
}
```

これは、テンプレート実引数の推定と同じである。

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

int main()
{
    int x = 0 ;

    f( x ) ; // Uはint &
    f( std::move(x) ) ; // Uはint &&
}
```

auto指定子で変数を宣言する場合は、必ず初期化子がなければならない。また、宣言しようとしている変数名が、初期化子の中で使われていてはならない。

```
auto a ; // エラー、初期化子がない
```

```
auto b = b ; // エラー、初期化子の中で宣言しようとしている変数名  
が使われている
```

初期化子が要素の型Uの初期化リストの場合、autoの型はstd::initializer_list<U>になる。

```
// std::initializer_list<int>  
auto a = { 1, 2, 3 } ;
```

宣言子が関数の場合、auto指定子を使った宣言は関数になる。

```
void f() { }  
auto (*p)() -> void = &f ;
```

auto指定子を使って、ひとつの宣言文で複数の変数を宣言することは可能である。その場合、変数の型は、それぞれの宣言子と初期化子から推定される型になる。

```
auto a = 0, & b = a, * c = &a ;
```

この例では、aの型はint、bの型はint &、cの型はint *となる。ただし一般に、コードの可読性の問題から、ひとつの宣言文で複数の変数を宣言するのは、避けたほうがよい。

ただし、複数の変数を宣言する場合、autoによって推定される型は、必ず同じでなければならない。

```
int x = 0 ;  
auto a = x, * b = &x ; // OK、autoに対して推定される型は同じ  
auto c = 0, d = 0.0 ; // エラー、型が同じではない
```

従来の、変数が自動ストレージの有効期間を持つということを明示的に指定する意味でのauto指定子は、廃止された。C++11では、autoキーワードを昔の文法で使用した場合、エラーとなる。

```
auto int x = 0 ; // エラー、C++11には存在しない昔の機能
```

7.3.6.5 decltype指定子 (decltype specifier)

`decltype (式)`

`decltype`の型は、オペランドの式の型になる。`decltype`指定子のオペランドの式は、未評価式である。

```
int main()
{
    decltype( 0 ) x ; // xの型はint
    decltype( x ) y ; // yの型はint

    int const c = 0 ;
    decltype( c ) z = 0 ; // zの型はint const
}
```

`decltype`指定子の型は、以下のような順序で、条件の合うところで、上から優先的に決定される。

`decltype(e)`に対して、

- もし、eが括弧式で囲まれていない、名前かクラスのメンバーアクセスであれば、`decltype`の型は、名前eの型になる。

```
int x ;
// decltype(x)の型はint
decltype(x) t1 ;

class C
{
public :
    int value ;
} ;
C c ;
// decltype(c)の型は、class C
decltype(c) t2 ;
// decltype(c.value)の型は、int
decltype(c.value) t3 ;
```

もし、eが関数呼び出しかオーバーロード演算子の呼び出しであれば、`decltype`の

型は、関数の戻り値の型になる。この際、括弧式は無視される。eという名前が見つからない場合や、eの名前がオーバーロード関数のセットであった場合、エラーとなる。

`decltype`のオペランドは未評価式なので、`sizeof`などと同じく、関数が実際に呼ばれることはない。

```
int f() ;

// decltype( f() ) の型は、int
decltype( f() ) t1 ;
// decltype( (f()) ) の型は、int
decltype( (f()) ) t2 ;

// エラー、fooという名前は見つからない
decltype( foo ) t3 ;

// エラー、オーバーロード関数のセット
void f(int) ;
void f(short) ;

decltype(f) * ptr ;
```

2. もし、eが`xvalue`であれば、eの型をTとした場合、`decltype(e)`の型は、`T &&`となる。

```
int x ;
// typeの型はint &&
using type = decltype( static_cast< int && >(x) ) ;
```

3. もし、eが`lvalue`であれば、eの型をTとした場合、`decltype(e)`の型は、`T &`となる。

```
int x ;
// decltype( (x) ) の型は、int &
using type = decltype( (x) ) ;
```

条件が上から優先されるということに注意。eが括弧で囲まれていない場合は、すでに1の条件に当てはまるので、この条件には当てはまらない。この条件はeが括弧で囲まれている場合である。

4. 上記以外の場合、`decltype`の型は、eの型となる。

```
// decltype(0)の型は、int
decltype(0) t1 ;
// decltype("hello")の型は、char const [6]
decltype("hello") t2 = "hello" ;
```

eが`value`で、しかも括弧式で囲まれている場合は、リファレンス型になるということには、注意を要する。

```
int x = 0 ;

decltype( x ) t1 = x ; // t1の型はint
decltype( (x) ) t2 = x ; // t2の型はint &
```

`decltype`は、他の型指定子や宣言子と併用できる。

```
int x ;
decltype( x ) * ptr ; // int *
decltype( x ) const & const_ref = x ; // int const &
```

`decltype`は、ネストされた名前の指定子として使用できる。

```
struct C
{
    typedef int type ;
} ;

int main()
{
    C c ;
    decltype(c)::type x = 0 ; // int
}
```

`decltype`は、基本クラスの指定子、メンバー初期化子として使用できる。

```
struct Base { } ;
Base base ;
```

```
struct Derived
    : decltype(base) // decltypeを基本クラスとして指定
{
    Derived()
        : decltype(base) () // メンバー初期化子
    { }
} ;
```

`decltype`は、疑似デストラクタ名として使用できる。

```
struct C { } ;

int main()
{
    C c ;
    c.~decltype(c)(); // 疑似デストラクターの呼び出し
}
```

7.4 enumの宣言 (Enumeration declarations)

`enum`指定子は、名前付きの定数と型を宣言、定義する。`enum`(*Enumeration*)は、歴史的に列挙型と呼ばれている。`enum`型の名前は、`enum`名といい、`enum`が宣言する定数のことを、列挙子(*enumerator*)と呼ぶ。

```
// Eはenum名、valueは列挙子
enum E { value = 0 } ;
```

本書では、`enum`の機能を四種類に大別して解説する。`unscoped enum`、`scoped enum`、`enum`基底(`enum-base`)、`enum`宣言(`opaque-enum-declaration`)である。

7.4.1 unscoped enum

`enum`指定子:
`enum` 識別子_{opt} `enum`基底_{opt} { 列挙子リスト }

`enum`というキーワードだけで宣言する`enum`のことを、`unscoped enum`という。`unscoped enum`は、弱い型付けの`enum`を宣言、定義する。`enum`の定義は、それぞれ別の型を持つ。列挙子リストとは、コンマで区切られた識別子である。各列挙子には、`=`に続けて定数を指定することで、値を指定できる。これを`enum`の初期化子という。ただし、列挙子自体はオブジェクトではない。`enum`は先頭の列挙子に初期化子がない場合、値は0になる。先頭以外の列挙子に初期化子がない場合、そのひとつ前の列挙子の値に、1を加算した値になる。

```
// a = 0, b = 1, c = 2
enum E { a, b, c } ;

// a = 3, b = 4, c = 5
enum E { a = 3, b = 4, c = 5 } ;

// a = -5, b = -4, c = -3
enum E { a = -5, b, c } ;

// a = 0, b = 1, c = 0, d = 1
enum E { a, b, c = 0, d } ;
```

宣言した列挙子は、次の列挙子から使うことができる。

```
// a = 0, b = 0, c = 5, d = 3
enum E { a, b = a, c = a + 5, d = c - 2 } ;
```

`enum`名と`unscoped enum`の列挙子は、`enum`指定子があるスコープで宣言される。

```
enum GlobalEnum { a, b } ;

GlobalEnum e1 = a ;
GlobalEnum e2 = b ;

int main()
{
    enum LocalEnum { c, d } ;
    LocalEnum e1 = c ;
    LocalEnum e2 = d ;
}
```

`unscoped enum`によって宣言された列挙子は、整数のプロモーションによって、暗黙的に整数型に変換できる。整数型は、明示的なキャストによって、`enum`型に変換できる。

整数型の値が、enum型の表現できる範囲を超えていた場合の挙動は、未定義である。

```
enum E { value = 123 } ;  
  
void f()  
{  
    int x = value ; // enum Eからintへの暗黙の型変換  
  
    E e1 = 123 ; // エラー、intからenum Eへの暗黙の型変換はできない  
    E e2 = static_cast<E>(123) ; // intからenum Eへの明示的なキャスト  
}  
  
// コマンドの種類を表す定数  
enum Command { copy, cut, paste } ;  
  
// コマンドを処理する関数  
void process_command( Command id )  
{  
    switch( id )  
    {  
        case copy :  
            // 処理  
            break ;  
        case cut :  
            // 処理  
            break ;  
        case paste :  
            // 処理  
            break ;  
  
        default :  
            // エラー処理  
            break ;  
    }  
}
```

クラスのスコープ内で宣言された列挙子の名前は、クラスのメンバーアクセス演算子 (::, ., ->) を使うことによって、参照することができる。

```

struct C
{
    enum { value } ;
    // クラススコープのなかでは名前のまま参照できる
    void f() { value ; }
} ;

int main()
{
    C::value ; // ::による参照
    C c ;
    c.value ; // .による参照
    C * p = &c ;
    p->value ; // ->による参照
}

```

unscoped enumは、識別子を省略することができる。

```

// 識別子あり
enum E { a } ;

// 識別子なし
enum { b } ;

```

7.4.2 scoped enum

scoped enumは、強い型付けをするenumである。

```

enum struct 識別子 enum基底opt { 列挙子リスト } ;
enum class 識別子 enum基底opt { 列挙子リスト } ;

```

scoped enumは、enum structかenum classという連続した二つのキーワードによって宣言する。enum structとenum classは、全く同じ意味である。どちらを使ってもよい。enumには、クラスのようなアクセス指定はない。scoped enumの識別子は省略できない。列挙子リストの文法は、unscoped enumと変わらない。

```
enum struct scoped_enum_1 { a, b, c } ;
```

```
enum class scoped_enum_2 { a, b, c } ;
```

scoped enumは、非常に強い型付けを持っている。列挙子は、scoped enumが宣言されているスコープに導入されることはない。かならず、enum名に::演算子をつけて参照しなければならない。

```
enum struct E { value } ;

value ; // エラー、scoped enumの列挙子は、このように参照できない
E::value ; // OK
```

このため、同じスコープ内で、同じ名前の列挙子を持つ、複数のscoped enumを宣言することもできる。

```
void f()
{
    // scoped enumの場合
    enum struct Foo { value } ;
    enum struct Bar { value } ; // OK

    Foo::value ; // enum struct Fooのvalue
    Bar::value ; // enum struct Barのvalue
}

void g()
{
    // unscoped enumの場合
    enum Foo { value } ;
    enum Bar { value } ; // エラー、すでにvalueは宣言されている。
}
```

scoped enumの列挙子は、暗黙的に整数型に変換することはできない。明示的にキャストすることはできる。整数型からenumへの変換は、unscoped enumと変わらない。つまり、明示的なキャストが必要である。

```
enum struct E { value = 123 } ;

int x = E::value ; // エラー、scoped enumの列挙子は暗黙的に変換できない
```

```
int y = static_cast<int>( E::value ) ; // OK、明示的なキャスト
E e = static_cast<E>( 123 ) ; // OK、unscoped enumと同じ
```

ただし、switch文の中のcaseラベルや、非型テンプレート実引数では、scoped enumも使うことができる。

```
enum struct E { value } ;
template < int N > struct C { } ;

void f( E e )
{
    // switch文のcaseラベル
    switch( e )
    { case E::value : ; }

    // 非型テンプレート実引数
    C< E::value > c ;
}
```

これが許されている理由は、scoped enumの内部的な値は使わないものの、強い型付けがされた一種のユニークな識別子として、scoped enumを使えるようにするためにある。

scoped enumは、強い型付けをするenumである。scoped enumは、列挙子の内部的な値は使わないが、単に名前付きの状態を表すことができる変数が欲しい場合、また、たとえ内部的な値を使うにしても、強い型付けによって、些細なバグを未然に防ぎたい場合などに使うことができる。

7.4.3 enum基底(enum-base)

enum基底：
：型指定子

enum型は、内部的には单なる整数型である。この内部的な整数型のことを、内部型(underlying type)という。enum基底(enum-base)は、この内部型を指定するための文法である。enum基底の型は、基本クラスの指定とよく似た文法で指定することができる。enum基底の型指定子は、整数型でなければならない。

enum基底が指定されたenum型の内部型は、enum基底の型指定子の型になる。

```
enum E : int { } ; // 内部型はint

enum struct Foo : int { } ; // 内部型はint
enum struct Bar : unsigned int { } ; // 内部型はunsigned int

enum Error : float { } ; // エラー、enum基底が整数型ではない
```

enum基底が省略された場合、scoped enumの内部型は、intになる。

```
enum struct E { } ; // scoped enumのデフォルトの内部型はint
```

scoped enumで、int型の範囲を超える値の列挙子を使いたい場合は、明示的にenum基底を指定しなければならない。

unscoped enumのenum基底が省略された場合、内部型は明確に定められない。

```
enum E { } ; // 内部型は定められない。
```

enumの内部型が定められていない場合、内部型は、enumの列挙子をすべて表現できる整数型になる。その際、どの整数型が使われるかは、実装依存である。どの整数型でも、すべての列挙子を表現できない場合は、エラーとなる。

7.4.4 enum宣言 (opaque-enum-declaration)

enum宣言は、正式には、opaque-enum-declarationという。これは、定義をしないenumの宣言である。関数や変数が、定義せずに宣言できるように、enumも、定義せずに宣言することができる。

```
enum 識別子 enum基底 ;
enum struct 識別子 enum基底opt ;
enum class 識別子 enum基底opt ;
```

unscoped enumの場合は、必ず定義と一致するenum基底を指定しなければならない。scoped enumの場合は、enum基底を省略した場合、内部型はデフォルトのintになる。ただし、安全のためにには、enum宣言と対応するenumの定義には、enum基底を明示的に書いておいたほうがよい。

```

enum struct Foo : unsigned int ; // 内部型はunsigned int
enum class Bar ; // enum基底が省略された場合、内部型はint

enum E1 : int ; // 内部型はint

enum E2 ; // エラー、unscoped enumの宣言では、enum基底を省略してはならない。

```

enum宣言によって、宣言のみされたenum名は、通常のenumと同じように使用できる。ただし、列挙子を使うことはできない。なぜならば、列挙子は宣言されていないからだ。

列挙子を使うことができないenum宣言に、何の意味があるのか。enum宣言が導入された背景には、ある種の利用方法では、すべての翻訳単位に列挙子が必要ないことがあるのだ。

無駄な定義を省く

```

// 翻訳単位 1

enum ID : int { /* 自動的に生成される多数の列挙子 */ } ;

```

```

// 翻訳単位 2

enum ID : int ; // enum宣言
void f( ID id ) // IDを引数にとる関数
{
    int x = id ;
    id = static_cast<ID>(0) ;
}

```

翻訳単位 1で定義されているenumの列挙子は、外部のツールによって、自動的に生成されるものだとしよう。この定義は、かなり頻繁に更新される。もし、翻訳単位 2では、enumの内部的な値が使われ、列挙子という名前付きの定数には、それほど意味が無い場合、この自動的に生成される多数の列挙子は、無駄である。なぜならば、enumが生成されるたびに、たとえ翻訳単位 2のソースコードに変更がなく、再コンパイルが必要ない場合でも、わざわざコンパイルしなおさなければならないからだ。

なぜenumの定義が必要かというと、完全な定義がなければ、enumの内部型を決定できないからである。C++11では、enum基底によって、明示的に内部型を指定できる。これにより、enumを定義せず宣言することができるようになった。

型安全なデータの秘匿

以下のクラスを考える。

```
// クラスCのヘッダーファイル
// C.h
enum struct ID { /* 自動的に生成される多数の列挙子 */ } ;

class C
{
public :
    // 外部に公開するメンバー

private :
    ID id ;
} ;
```

さて、このクラスを、複数の翻訳単位で使いたいとする。このクラスには、データメンバーとしてenum型の変数があるが、これは外部に公開しない。クラスの中の実装の都合上のデータメンバーである。enumの列挙子は、外部ツールで自動的に生成されるものとする。

すると、このヘッダーファイルを#includeしているソースファイルは、enumが自動的に生成されるたびに、再コンパイルしなければならない。しかし、このクラスを使うにあたって、enumの定義は必要ないはずである。この場合にも、enum宣言が役に立つ。

```
// クラスCのヘッダーファイル
// C.h

enum struct ID : int ;

class C { /* メンバー */ } ;
```

```
// クラスCのソースコード
// C.cpp

enum struct ID : int { /* 自動的に生成される多数の列挙子 */ } ;

// メンバーの実装
```

このようにしておけば、enumの定義が変更されても、クラスのヘッダーファイルを #includeして、クラスを使うだけのソースコードまで、再コンパイルする必要はなくなる。

7.5 名前空間(Namespace)

名前空間とは、宣言の集まりに名前を付ける機能である。名前空間の名前は、::演算子によって、宣言を参照するために使うことができる。名前空間は、複数定義することができる。グローバルスコープも、一種の名前空間スコープとして扱われる。

7.5.1 名前空間の定義(Namespace definition)

```
inline_opt namespace 識別子 { 名前空間の本体 }
```

名前空間は、別の名前空間スコープの中か、グローバルスコープに書くことができる。名前空間の本体には、あらゆる宣言を、いくつでも書くことができる。これには、名前空間自身も含まれる。

```
// グローバルスコープ

namespace NS
{ // NSという名前の名前空間のスコープ
    // 宣言の例
    int value = 0 ;
    void f() { }
    class C { } ;
    typedef int type ;
} // NSのスコープ、ここまで

int main()
{
    // NS名前空間の中の名前を使う
    NS::value = 0 ;
    NS::f() ;
    NS::C c ;
    NS::type t ;

    value ; // エラー、名前が見つからない
}
```

名前空間は、名前の衝突を防ぐために使うことができる。

```
// グローバルスコープ
int value ;
int value ; // エラー、valueという名前はすでに宣言されている

// OK、名前空間Aのvalue
namespace A { int value ; }
// OK、名前空間Bのvalue
namespace B { int value ; }

int main()
{
    value ; // グローバルスコープのvalue
    A::value ; // 名前空間Aのvalue
    B::value ; // 名前空間Bのvalue
}
```

グローバル変数として、valueのような、分かりやすくて誰でも使いたがる名前を使うのは、問題が多い。しかし、名前付きの名前空間スコープの中であれば、名前の衝突を恐れずに、気軽に短くてわかりやすい名前を使うことができる。

名前空間は、何度も定義することができる。

```
// 最初の定義
namespace NS { int x ; }

// 二度目の定義
namespace NS { int y ; }
```

名前空間はネストできる。

```
namespace A { namespace B { namespace C { int value ; } } }

int main()
{
    A::B::C::value ; // Aの中の、Bの中の、Cの中のvalue
}
```

7.5.1.1 inline名前空間

inlineキーワードの書かれた名前空間の定義は、inline名前空間である。inline名前空間スコープの中で宣言された名前は、inline名前空間の外側の名前空間のスコープの中でも使うことができる。

```
inline namespace NS { int value ; }

namespace Outer
{
    inline namespace Inner
    {
        int value ;
    }
}

int main()
{
    value ; // NS::value
NS::value ;

Outer::value ; // Outer::Inner::value
Outer::Inner::value ;
}
```

7.5.1.2 無名名前空間(Unnamed namespaces)

無名名前空間(unnamed namespace)は、名前をつけない名前空間の宣言である。

```
namespace { }
```

無名名前空間は、通常の名前つき名前空間とは違い、その名前空間スコープ内のエンティティを、内部リンクージにするのに使われる。

```
namespace
{
```

```
int x ; // 内部リンクエイリアス
}
```

7.5.1.3 名前空間のメンバーの定義(Namespace member definitions)

ある名前空間スコープの中で宣言された名前を、その名前空間のメンバーと呼ぶ。名前空間のメンバーは、名前空間の外側で定義することができる。

```
namespace NS
{
    void f() ; // 関数fの宣言
    namespace Inner
    {
        void g() ; // 関数gの宣言
        void h() ; // 関数hの宣言
    }
    void Inner::g() { } // 関数gの定義
}

void NS::f() { } // 関数fの定義
void NS::Inner::h() { } // 関数hの定義
```

ただし、名前空間の外側で定義されているからといって、名前空間の外側のスコープにも、名前が導入されるわけではない。あくまで、名前空間の外側でも、定義ができるだけである。

7.5.2 名前空間エイリアス(Namespace alias)

```
namespace 識別子 = 名前空間の名前 ;
```

名前空間エイリアス(Namespace alias)とは、名前空間の名前の別名を定義する機能である。

名前空間は、名前の衝突を防いでくれる。しかし、名前空間の名前自体が衝突してし

まうこともある。それを防ぐためには、名前空間の名前には、十分にユニークな名前をつけなければならない。しかし、衝突しない名前をつけようすると、どうしても、短い名前をつけることはできなくなってしまう。

```
namespace Perfect_cpp
{
    int x ;
}

int main()
{
    Perfect_cpp::x = 0 ;
}
```

この例では、十分にユニークな名前、Perfect_cppを使っている。このため、xという名前の変数名でも、衝突を恐れず使うことができる。しかし、このPerfect_cppは長い上に、大文字とアンダースコアを使っており、タイプしづらい。そこで、名前空間エイリアスを使うと、別名を付けることができる。

```
namespace Perfect_cpp { int x ; }

int main()
{
    namespace p = Perfect_cpp ;
    p::x ; // Perfect_cpp::x
}
```

ネストされた名前にも、短い別名をつけることができる。

```
namespace Perfect_cpp { namespace Library { int x ; } }

int main()
{
    namespace pl = Perfect_cpp::Library ;
    pl::x ; // Perfect_cpp::Library::x
}
```

別名の別名を定義することもできる。

```
namespace Long_name_is_Looooong { }
```

```
namespace long_name = Long_name_is_Looooong ;
namespace ln = long_name ;
```

同じ宣言領域で、別名と名前空間の名前が衝突してはならない。

```
namespace A { } namespace B { }

// エラー、同じ宣言領域では、別名と名前空間の名前が衝突してはならない
namespace A = B ;

int main()
{
    // OK、別の宣言領域なら可
    namespace A = B ;
}
```

7.5.3 using宣言 (The using declaration)

```
using 識別子 ;
```

using宣言は、宣言が書かれている宣言領域に、指定した名前を導入する。これにより、明示的に名前空間名と::演算子を使わなくても、その宣言領域で、名前が使えるようになる。

using宣言を、名前空間のメンバーに使う場合、using宣言が書かれているスコープで、::演算子による明示的なスコープの指定なしに、その名前を使うことができる。

```
namespace NS { int name ; }

int main()
{
    name = 0 ; // エラー、名前nameは宣言されていない

    using NS::name ;
    name = 0 ; // NS::nameと解釈される
    NS::name = 0 ; // 明示的なスコープの指定
```

```
}
```

```
// ブロックスコープ外でもusing宣言は使える
using NS::name ;
```

using宣言は、テンプレート識別子を指定することはできない。テンプレート名は指定できる。

```
namespace NS
{
    template < typename T > class C { } ;

int main()
{
    using NS::C ; // OK、テンプレート名は指定できる
    using NS::C<int> ; // エラー、テンプレート識別子は指定できない
}
```

using宣言は、名前空間の名前を指定することはできない。

```
namespace NS { }
using NS ; // エラー
```

using宣言は、scoped enumの列挙子を指定することはできない。

```
namespace NS
{
    enum struct E { scoped } ;
    enum { unscoped } ;

int main()
{
    using NS::unscoped ; // OK、unscoped enumの列挙子
    using NS::E::scoped ; // エラー、scoped enumの列挙子は指定できない
}
```

using宣言は、その名の通り、宣言である。したがって、通常通り、外側のスコープの名前を隠すこともできる。7.5.4 usingディレクティブとは、違いがある。

```
int name ; // グローバルスコープのname
namespace NS { int name ; } // NS名前空間のname

int main()
{
    // ここではまだ、名前が隠されてはいない
    name = 0 ; // ::nameを意味する

    using NS::name ; // このusing宣言は::nameを隠す
    name = 0 ; // NS::nameを意味する
}
```

using宣言は、宣言された時点で、すでに宣言されている名前を、スコープに導入する。宣言場所から見えない名前は、導入されない。

```
namespace NS { void f( int ) { } }
// void NS::f(int)をグローバルスコープに導入する
// void NS::f(double)は、この時点では宣言されていないので、導入されない
using NS::f ;
namespace NS { void f( double ) { } }

int main()
{
    // この時点で、unqualified名fとして名前探索されるのは
    // void NS::f(int)のみ
    f( 1.23 ) ; // NS::f(int)を呼ぶ。

    using NS::f ; // void NS::f(double) をmain関数のブロックスコープに導入する
    f( 1.23 ) ; // オーバーロード解決により、NS::f(double)を呼ぶ
}
```

ただし、テンプレートの部分的特殊化は、プライマリークラステンプレートを経由して探すので、たとえusing宣言の後に宣言されていたとしても、発見される。

```
namespace NS
{
```

```

template < typename T >
class C { } ;

}

using NS::C ;

namespace NS
{
    template < typename T >
    class C<T * > { } ; // ポインター型への部分的特殊化
}

int main()
{
    C<int * > c ; // 部分的特殊化が使われる。
}

```

using宣言は、12.9 [コンストラクターの継承](#)に使うこともできる。詳しくは、該当の項目を参照。ここでは、クラスのメンバー宣言としてusing宣言を使う際の、文法上の注意事項だけを説明する。

クラスのメンバー宣言としてusing宣言を使う場合、基本クラスのメンバーネームを指定しなければならない。名前空間のメンバーは指定できない。using宣言は、基本クラスのメンバーネームを、派生クラスのスコープに導入する。

```

namespace NS { int value ; }

class Base
{
public :
    void f() { }
} ;

class Derived : private Base
{
public :
    using Base::f ; // OK、基本クラスのメンバーネーム
    using NS::value ; // エラー、基本クラスのメンバーではない
} ;

```

クラスのメンバー宣言としてのusing宣言は、基本クラスのメンバーの名前を、クラスのメンバーの名前探索で発見させることができる。

```

struct Base { void f( int ) { } } ;
struct Derived1 : Base
{
    void f( double ) { }
} ;
struct Derived2 : Base
{
    using Base::f ;
    void f( double ) { }
} ;

int main()
{
    Derived1 d1 ;
    d1.f( 0 ) ; // Derived::f(double)を呼ぶ
    Derived2 d2 ;
    d2.f( 0 ) ; // Base::f(int)を呼ぶ
}

```

Derived1::fは、Base::fを隠してしまうので、Base::fはオーバーロード解決の候補関数に上がることはない。Derived2では、using宣言を使って、Base::fをDerived2のスコープに導入しているので、オーバーロード解決の候補関数として考慮される。

また、using宣言は、基本クラスのメンバーのアクセス指定を変更する目的でも使える。

```

class Base
{
    int value ;
public :
    int get() const { return value ; }
    void set( int value ) { this->value = value ; }
} ;

// Baseからprivateで派生
class Derived : private Base
{
public : // Base::getのみpublicにする
    using Base::get ;
} ;

```

この例では、DerivedはBaseからprivate派生している。ただし、Base::getだけは、publicにしたい。そのような場合に、using宣言が使える。

using宣言でクラスのメンバーネームを指定する場合、クラスのメンバー宣言でなければならない。クラスのメンバー宣言以外の場所で、using宣言にクラスのメンバーネームを指定してはならない。

```
struct C
{
    int x ;
    static int value ;
} ;

int C::value ;

using C::x ; // エラー、これはクラスのメンバー宣言ではない
using C::value ; // エラー、同上
```

7.5.4 usingディレクティブ(Using directive)

```
using namespace 名前空間名 ;
```

usingディレクティブ(using directive)は、その記述以降のスコープにおける非修飾名前探索に、指定された名前空間内のメンバーを追加するための指示文である。usingディレクティブを使うと、指定された名前空間内のメンバーを、::演算子を用いないで使用できる。

```
namespace NS
{
    int a ;
    void f() { }
    class C { } ;
}

int main()
{
    using namespace NS ;

    a = 0 ; // NS::a
    f() ; // NS::f
    C c ; // NS::C
```

}

usingディレクティブを使えば、指定された名前空間内のすべてのメンバーを、明示的演算子を使わずにアクセスできるようになる。

usingディレクティブは、名前空間スコープとブロックスコープ内で使用することができる。クラススコープ内では使用できない。

```
namespace A { typedef int type ; }

void f()
{
    // ブロックスコープ内
    using namespace A ;
    type x ; // A::type
}

namespace B
{
    // 名前空間スコープ
    using namespace A ;
    type x ; // A::type
}

other_namespace::type g1 ; // A::type

// 名前空間スコープ（グローバルスコープ）
using namespace A ;
type g2 ; // A::type

class C
{
    using namespace A ; // エラー、クラススコープ内では使用でき
    // ない
} ;
```

グローバルスコープにusingディレクティブを記述するのは推奨できない。特に、ヘッダーファイルのグローバルスコープにusingディレクティブを記述すると、非常に問題が多い。名前空間の本来の目的は、名前の衝突を防ぐためである。usingディレクティブは、名前空間という仕組みに穴を開けるような機能だからだ。

しかし、usingディレクティブは必要である。たとえば、非常に長い名前空間名や、深くネストした名前空間内の多数のメンバーを使う場合、いちいち::演算子で明示的にスコープを指定したり、using宣言でひとつひとつ宣言していくのは、非常に面倒である。ある

ブロックスコープで、名前が衝突しないということが保証できるならば、usingディレクティブを使っても構わない。

```
namespace really_long_name { namespace yet_another_long_name
{
    int a ; int b ; int c ; int d ;
} }

void f()
{
    // このスコープでは、a, b, c, dという名前は衝突しないと保証できる
    using namespace really_long_name::yet_another_long_name ;
    a = 0 ; b = 0 ; c = 0 ; d = 0 ;
}
```

usingディレクティブは、宣言ではない。usingディレクティブは、非修飾名前探索に、名前を探すべき名前空間を、特別に追加するという機能を持っている。したがって、usingディレクティブは、名前を隠さない。以下の例はエラーである。7.5.3 [using宣言](#)と比較すると、違いがある。

```
int name ; // グローバルスコープのname
namespace NS { int name ; } // NS名前空間のname

int main()
{
    // ここではまだ、名前が隠されてはいない
    name = 0 ; // ::nameを意味する

    using namespace NS ; // 名前探索にNS名前空間内のメンバーを追加
    name = 0 ; // エラー、::nameとNS::nameとで、どちらを使うべきか曖昧
}
```

usingディレクティブは、非修飾名前探索にしか影響を与えない。ADLには影響を与えない。

```
namespace NS
{
    struct S { } ;
```

```

namespace inner
{
    void f(S) { }
    void g(S) { }
}

using inner::f ; // inner::fをNS名前空間に導入する
using namespace inner ; // 非修飾名前探索に影響をおぼす
} ;

int main()
{
    NS::S s ;
    f(s) ; // OK
    g(s) ; // エラー、usingディレクティブはADLには影響しない
}

```

usingディレクティブで探索できるようになった名前は、オーバーロード関数の候補にもなる。

```

void f( int ) { }
namespace NS { void f( double ) { } }

int main()
{
    // この時点では、NS::fは名前探索で発見されない
    f( 1.23 ) ; // ::f(int)

    using namespace NS ; // NS名前空間のメンバーがunqualified名
    前探索で発見されるようになる
    f( 1.23 ) ; // オーバーロード解決により、NS::f( double )
}

```

usingディレクティブは、unqualified名前探索のルールを変更するという、非常に特殊な機能である。usingディレクティブは、確実に名前が衝突しないブロックスコープ内で使うか、あるいは、オーバーロード解決をさせてるので、同じ関数名を複数、意図的に名前探索で発見させる場合にのみ、使うべきである。

7.6 リンケージ指定 (Linkage specifications)

関数型、外部リンクエージを持つ関数名、外部リンクエージを持つ変数名には、言語リンクエージ(language linkage)という概念がある。リンクエージ指定(Linkage specification)は、言語リンクエージを指定するための文法である。リンクエージ指定と、7.3.1 [ストレージクラス指定子](#)のextern指定子とは、別物である。

注意、実装依存の話：言語リンクエージは、C++と他のプログラミング言語との間での、関数名や変数名の相互利用のための機能である。異なる言語間で名前を相互利用するには、共通の仕組みが必要である。これには、たとえば名前マングリングを始めとして、レジスターの使い方、引数のスタックへの積み方などの様々な要素がある。しかし、これらはいずれも本書の範疇を超えるので解説しない。

```
extern 文字列リテラル { 宣言リスト }
extern 文字列リテラル 宣言
```

標準では、C++言語リンクエージと、C言語リンクエージを定めている。C++の場合、文字列リテラルは“C++”となり、C言語の場合、文字列リテラルは“C”となる。何も指定しない場合、デフォルトでC++言語リンクエージとなる。異なるリンクエージ指定がされた名前は、たとえその他のシグネチャーがすべて同じであつたとしても、別の型として認識される。その他の文字列がどのような扱いを受けるかは、実装依存である。

```
// 関数型へのC言語リンクエージの指定
extern "C" typedef void function_type() ;
// 関数名へのC言語リンクエージの指定
extern "C" int f() ;
// 変数名へのC言語リンクエージの指定
extern "C" int value ;
```

[]を使う方のリンクエージ指定子は、複数の宣言に対して、一括して言語リンクエージを指定するための文法である。

```
// 関数名f, g, hは、すべてC言語リンクエージを持つ
extern "C"
{
    void f() ;
    void g() ;
    void h() ;
}

// C_functions.hというヘッダーファイルで宣言されているすべての関
// 数型、関数名、変数名は、C言語リンクエージを持つ
extern "C"
{
    #include "C_functions.h"
```

}

リンクエージ指定をしない場合、デフォルトでC++言語リンクエージだとみなされる。通常、C++言語リンクエージを指定する必要はない。

```
// デフォルトのC++言語リンクエージ  
void g() ;  
// 明示的な指定  
extern "C++" void g() ;
```

リンクエージ指定はネストすることができる。その場合、一番内側のリンクエージ指定が使われる。言語リンクエージは、スコープをつくれない。

```
extern "C"  
{  
    void f() ; // C言語リンクエージ  
    extern "C++"  
    {  
        void g() ; // C++言語リンクエージ  
    }  
}
```

リンクエージ指定は、名前空間スコープの中でのみ、使うことができる。ブロックスコープ内などでは使えない。

C言語リンクエージは、クラスのメンバーに適用しても無視される。

```
extern "C"  
{  
    class C  
    {  
        void f() ; // C++言語リンクエージ  
    } ;  
}
```

C言語リンクエージを持つ同名の関数が、複数あってはならない。これにより、C言語リンクエージを持つ関数は、オーバーロードできない。

```
extern "C"
```

```
{
// エラー、C言語リンクエージを持つ同名の関数が複数ある
void f(int) ;
void f(double) ;
}

// OK、互いに異なる言語リンクエージを持つ
void g() ;
extern "C" void g() ;
```

このルールは、たとえ関数が名前付きの名前空間の中で宣言されていても、同様である。

```
namespace A
{
    extern "C" void f() ;
}

namespace B
{
    extern "C" void f() ; // エラー、A::fとB::fは同じ関数を参照
    // する
    void f() ; // OK、C++言語リンクエージを持つ
}
```

このように、たとえ名前空間が違ったとしても、C言語リンクエージを持つ関数は、名前が衝突してはならない。これは、名前空間という仕組みが存在しないC言語からでも使えるようにするためにの仕様である。ただし、C言語リンクエージを持つ関数を、C++側から、名前空間の中で宣言して、通常通り使うことはできる。

```
namespace NS
{
    extern "C" void f() { }
}

int main()
{
    NS::f() ; // OK
}
```

これにより、C言語で書かれた関数を、何らかの名前空間の中にいれて、C++側から使うこともできる。

```
// ヘッダーファイル、C_functions.hは、C言語で書かれているものとする
namespace obsolete_C_lib
{
    extern "C"
    {
        #include "C_functions.h"
    }
}
```

7.7 アトリビュート(Attribute)

NOTE: アトリビュートの解説は、C++14となる予定の現時点での最新ドラフト、N3797を参考にしている。

アトリビュート(attribute)は、属性とも呼ばれ、ソースコードに追加的な情報を指定するための文法である。

アトリビュートの文法は、以下のようになる。

[[アトリビュートリスト]]

アトリビュートは、文法上、実に様々な箇所に書くことができる。アトリビュートリストには、アトリビュート用のトークンを指定することができる。このトークンは、アトリビュートの中だけで通用する識別子であり、予約語ではない。

アトリビュートは、特定のC++実装の独自機能を、独自の文法を使わずに表現するために追加された。また、C++規格でも、アトリビュート用の機能をいくつか定めている。

7.7.1 アライメント指定子(alignment specifier)

アライン(align)やアライメント(alignment)とは、ある型のオブジェクトを構築するストレージのアドレスに、制約を指定する機能である。

アライメント指定子は、通常の[[]]のようなアトリビュートの文法を使わない。alignasというキーワードが与えられている。当初、アライメントを指定する機能は、アトリビュートの文法を使う予定だったが、アライメント指定のような基本的な機能には、独自のキーワードが与えられるべきだという合意がなされたため、独自のキーワードが与えられ

た。指定子と名前はついているものの、アライメント指定子が現れることがある箇所の文法は、アトリビュートの文法に基づいている。

アライメント指定子は、変数とクラスのデータメンバーに指定できる。書ける場所は、宣言の前がわかりやすい。

```
// OK
alignas(16) char buf[16] ;

struct S
{
// OK
    alignas(16) char buf[16] ;
};
```

ただし、ビットフィールド、関数の仮引数、例外宣言、registerストレージクラス指定子つきで宣言された変数には指定できない。

```
struct S
{
// エラー、ビットフィールド
    alignas(4) unsigned int data:16 ;
};

// エラー、関数の仮引数
void f( alignas(4) int x ) ;

void g()
{
    try { }
// エラー、例外宣言
    catch ( alignas(4) int x ) { }
}
```

アライメント指定子は、クラスの宣言や定義に適用することもできる。書ける場所は、クラスキーの後、クラス名の前である。

```
struct alignas(16) S ;
struct alignas(16) S { } ;
```

同様に、アライメント指定子は、enumの宣言や定義にも適用することができる。

```
enum struct alignas(16) E1 : int
{ value } ;
enum alignas(16) E2
{ value } ;
```

アライメント指定子にエリプシス(...)を適用したものは、パック展開である。

```
template < typename ... Types >
struct alignas( Types ... ) // パック展開
S { } ;
```

アライメント指定子には、二種類の形がある。alignas()の括弧の中身が、コンマで区切られた代入式という形と、コンマで区切られた型という形だ。

```
// 代入式
alignas( 4, 8, 16 ) char b1[128] ;
// 型
alignas( short, int, long ) char b2[128] ;
```

アライメント指定子がalignas(代入式)の形の場合、

- 代入式は整数定数式でなければならない
- 定数式が拡張アライメントであり、実装がそのアライメントをその文脈でサポートしている場合は、宣言されたエンティティのアライメントは、指定されたアライメントになる
- 定数式が拡張アライメントであり、実装がそのアライメントをその文脈でサポートしていない場合、エラーとなる
- 定数式がゼロだと評価された場合、アライメント指定子の効果はなくなる

これ以外の場合、エラーとなる。

```
// OK
constexpr std::size_t f() { return 4 ; }
alignas( f() ) char b1[128] ;

// エラー
int g() { return 4 ; }
alignas( g() ) char b2[128] ;

// 実装がサポートしている場合OK
```

```
// 実装がサポートしていなければエラー
alignas( alignof(std::max_align_t) * 2 ) char b3[128] ;

// OK、ただしアライメント指定の効果なし
alignas( 0 ) char b4[128] ;
```

アライメント指定子が、alignas(型名)の場合、alignas(alignof(形名))と同等の効果になる。

```
// alignas( alignof(int) )と同等
alignas( int ) int x ;
```

ひとつのエンティティに複数のアライメントが指定された場合、最も厳格なアライメント要求になる。

```
// もし、実装が16のアライメント要求をサポートしている場合
// アライメント要求は16
alignas( 4, 8, 16 ) char b[128] ;
```

アライメント指定子が、宣言されるエンティティのアライメント要求より緩いアライメントを指定すると、エラーとなる。

```
// Sの最低のアライメント要求は8
struct alignas(8) S { } ;

// エラー、Sの最低のアライメント要求より緩い
alignas(4) S s ;
```

あるエンティティの定義である宣言にアライメント指定子がある場合、同じエンティティの定義ではない宣言は、同等のアライメントを指定するか、アライメント指定子なしでなければならない。

```
struct alignas(8) S { } ;

struct S ; // OK
struct alignas(8) S ; // OK

struct alignas(4) S ; // エラー、同等ではない
```

```
struct alignas(16) S ; // エラー、同等ではない
```

もし、エンティティの宣言のどれかひとつにでもアライメント指定子があった場合、そのエンティティの定義となる宣言にも、同等のアライメントを指定しなければならない。

```
struct S { } ; // エラー、アライメント指定つきの宣言がある
struct alignas(4) S ;
```

7.7.2 noreturnアトリビュート(Noreturn attribute)

アトリビュートトークン、`noreturn`は、関数が`return`しないことを指定する。このアトリビュートは、関数宣言の宣言子`id`に適用できる。

```
// この関数は決してreturnしない
[[ noreturn ]] void f()
{
// なぜならば、必ずthrowするからだ
    throw 0 ;
}
```

`noreturn`は、宣言中に一度しか現れてはならない。`noreturn`を指定する関数は、その最初に現れる宣言にこのアトリビュートを適用しなければならない。

```
void f() ; // エラー、最初の宣言だがnoreturnがない
[[ noreturn ]] void f() ;
```

もし、ある関数が、ある翻訳単位では`noreturn`が指定され、別の翻訳単位では指定されていない場合、エラーとなる。

`noreturn`を指定した関数から`return`した場合、挙動は未定義である。例外によって抜けるのは問題がない。

8 宣言子(Declarators)

7 宣言は、指定子と宣言子に分けることができる。宣言子は、変数、関数、型を宣言す

る。また、指定子によって指定された型を変更することもある。

```
// 変数の例
int // 指定子、int型
x ; // 宣言子、xという名前のint型の変数

// 関数の例
int // 指定子
f(void) ; // 宣言子、fという名前のint (void)型の関数の宣言

// 型の例
typedef int // 指定子
type ; // 宣言子、typeという名前のint型の別名の宣言

// 指定子の型を変更する例
int // 指定子
* p ; // 宣言子（ポインター型の変数）
```

宣言子をコンマで区切ることによって、ひとつの宣言の中で、複数の宣言子を書くことができる。ただし、ひとつの宣言の中で複数の宣言子を書くコードは、非常に分かりにくくなるので、やめたほうがよい。

```
int          // 指定子
a,          // 宣言子、aという名前のint型の変数
* b,        // 宣言子、bという名前のint *型の変数
c(void) ;   // 宣言子、cという名前のint (void)型の関数
```

8.1 型名 (Type names)

キャスト、sizeof、alignof、new、typeidでは、型の名前を指定する。これには、型名が用いられる。型名とは、変数や関数を宣言する場合と同じ指定子と宣言子だが、名前を省略したものである。

```
int value ;           // int型の変数
int                 // 型名
int array[10] ;      // 配列型の変数
int [10]            // 型名
int (*p)(void) ;    // 関数ポインター型の変数
int (*)() ;         // 型名
```

ただし、あまりにも複雑な型名は、`typedef`やエイリアス宣言で`typedef`名を定義した方が分かりやすい。

8.2 曖昧解決 (Ambiguity resolution)

宣言文は、文法上、曖昧になる場合がある。これは、通常気にする必要はない。ただし、まれにこの曖昧性に引っかかり、不可解なコンパイルエラーを引き起こす可能性がある。

仮引数名に無駄な括弧のある関数の宣言と、初期化式に関数形式のキャストを用いた変数の宣言は曖昧である。その場合、`=`が使われていると、初期化であるとみなされる。また、仮引数の周りの不必要的括弧は取り除かれる。その結果、変数の宣言のつもりで書いたコードが、関数の宣言とみなされてしまうことがある。

```
struct S { S(int) { } } ;

int main()
{
    double d = 0.0 ;

    S w( int(d) ); // S(int)型の関数の宣言、dは仮引数名、無駄な
                    括弧
    S x( int() ); // S(int)型の関数の宣言、仮引数名の省略、無駄
                    な括弧

    S y( static_cast<int>(d) ); // S型の変数yの宣言
    S z = int(d); // S型の変数zの宣言
}
```

一般に、この曖昧解決の結果、変数を宣言したいのに、関数の宣言とみなされてしまい、コンパイルエラーになることがある。この文法上の曖昧性を避けるためには、`static_cast`などの他のキャストを使うか、`=`を使った初期化に書き換える必要がある。

関数形式のキャストと型名は、曖昧になることがある。例えば、`int()`というコードは、関数形式のキャストにも、`int(void)`型の関数の型名にも解釈できる。この場合、型名が期待される場所では、常に型名として解釈される。

```
template < int N > struct C { } ;

int main()
{
    int x = int(); // OK、intは関数形式のキャスト
```

```
C<int()> ; // エラー、int()は型名
C<int(0)> ; // OK

sizeof(int()) ; // エラー、int()は型名
sizeof(int(0)) ; // OK
}
```

8.3 宣言子の意味 (Meaning of declarators)

7.1 単純宣言は、指定子と宣言子で構成される。

指定子 宣言子 ;

宣言子は、必ず、識別子を持たなければならない。この識別子を、規格上では、宣言識別子(declarator-id)と名付けている。この識別子は、宣言される名前となる。

```
int name1 ; // name1はint型の変数の名前
int * name2 ; // name2 はint *型の変数の名前
int name3(void) ; // name3 はint (void)型の関数の名前
```

static, thread_local, extern, register, mutable, friend, inline, virtual, typedefといった指定子は、この宣言子の識別子に適用される。

宣言子の識別子の意味は、指定子と宣言子の組み合わせによって決定される。

```
// 識別子aはint型の変数
int      // 指定子
a ;      // 宣言子、
// 識別子bはint型のtypedef名
typedef int // 指定子
b ;      // 宣言子
```

変数、関数、型は、すべてこの指定子と宣言子の組み合わせによって宣言される。宣言子は、識別子の他にも、様々な文法があり、それによって、指定子の型を変更する。これには、ポインター、リファレンス、メンバーへのポインター、配列、関数、デフォルト実引数がある。

8.3.1 ポインター(Pointers)

宣言子が以下のように記述されている場合、ポインターを意味する。

* CV修飾子_{opt} 識別子

`int * a ; // intへのポインター型`

*に続くCV修飾子は、ポインター型に対するCV修飾子として解釈される。

`int * const`は、指定子`int`へのポインターに対する`const`である。`int`に対する`const`ではない。`const int *`と`int const *`はどちらも同じ型である。`const int`も`int const`も指定子だからだ。

```
typedef const int * type1 ;
typedef int const * type2 ;
typedef int * const type3 ;
```

このように宣言されている場合、`type1`と`type2`は同じ型である。`type3`は、`type1`や`type2`とは別の型である。

宣言子の中に、ポインターは複数書くことができる。

```
int obj ;
int * p = &obj ;
int * * pp = &p ;
int * * * ppp = &pp ;

int const * * a ; // int const *へのポインター
int * const * b ; // int * constへのポインター
int * * const c = nullptr ; // int *へのconstなポインター
```

`T **`という型は、`T *`へのポインターということになる。

リファレンスへのポインターは存在しない。

`int & * ptr_to_ref ; // エラー、リファレンスへのポインターは存在しない`

ビットフィールドのアドレスを取得することは禁止されているので、ビットフィールドへのポインターも存在しない。

関数へのポインターや、配列へのポインターは存在する。ただし、記述がやや難しい。コードの可読性を上げるために、これらの型や変数を宣言するには、typedefやエイリアス宣言、autoなどを使うという手もある。

```
void func( void ) { } // 型はvoid (void)

void g()
{
    void (*ptr_func)( void ) = &func ;
    ptr_func() ; // 関数呼び出し

    int array[5] ; // 型はint [5]
    int (*ptr_array)[5] = &array ;
}
```

8.3.2 リファレンス (References)

宣言子が以下のように記述されている場合、リファレンスを意味する。

& 識別子
 && 識別子

&の場合、lvalueリファレンスとなり、&&の場合、rvalueリファレンスとなる。lvalueリファレンスとrvalueリファレンスは、ほとんど同じ働きをする。単にリファレンスといった場合、lvalueリファレンスとrvalueリファレンスの両方を指す。

```
void f( int obj )
{
    int & lvalue_reference = obj ;
    int && rvalue_reference = static_cast< int && >( obj ) ;

    lvalue_reference = 0 ; // objに0を代入
    rvalue_reference = 0 ; // objに0を代入
}
```

リファレンスは、CV修飾できない。

```
void f( int obj )
{
    int const & a = obj ; // OK、int constへのリファレンス
    int & const b = obj ; // エラー、リファレンスへのCV修飾はできない
}
```

ただし、typedefやテンプレート実引数にCV修飾子が使われた場合は、単に無視される。

```
// typedefの例
void f( void )
{
    int const & a = 3 ; // OK、int constへのリファレンス
    typedef int & type ;
    typedef const type type2 ; // type2の型はint &
    const type b = 3 ; // エラー、bの型はint &
}

// テンプレート実引数の例
template < typename T >
struct S
{
    typedef const T type ;
} ;

void g()
{
    typedef S< int & >::type type ; // type はint &
}
```

typedefの例では、typeというtypedef名にconst修飾子が使われているが、これは単に無視される。したがって、bの型は、int &である。int & constとはならないし、int const &ともならない。

テンプレートの例では、テンプレート仮引数Tの実際の型が決定されるのは、テンプレート実引数が渡されて、インスタンス化されたときである。

void型へのリファレンスは存在しない。

```
typedef void & type ; // エラー
```

リファレンスへのリファレンス、リファレンスの配列、リファレンスへのポインターは存在しない。

```
typedef int & & type1 ; // エラー、リファレンスへのリファレンス
typedef int & type2[5] ; // エラー、リファレンスの配列
typedef int & * type3 ; // エラー、リファレンスへのポインター
—
```

その他の型へのリファレンスは存在する。例えば、配列へのリファレンスや、関数へのリファレンスは存在する。ただし、ポインターと同じく、記述がやや難しい。

```
void func( void ) { } // 型はvoid (void)

void g()
{
    void (&ref_func)( void ) = func ;
    ref_func() ; // 関数呼び出し

    int array[5] ; // 型はint [5]
    int (&ref_array)[5] = array ;
}
```

リファレンスの宣言には、8.5 初期化子が必要である。

```
void f()
{
    int obj ;

    int & ref1 ; // エラー、初期化子がない
    int & ref2 = obj ; // OK
}
```

ただし、宣言がextern指定子を含む場合、クラスのメンバーの宣言である場合、関数の仮引数や戻り値の型の宣言である場合は、初期化子は必要ない。

```

int obj ;

// クラスの例
struct S
{
    int & ref ;
    S() : ref(obj) { }
};

// extern指定子の例
// このリファレンスは单なる宣言。
// 実体はどこか別の場所で定義されている
extern int & external_ref ;

// 関数の仮引数と戻り値の型の例
int & f( int & ref ) { return ref ; }

```

リファレンスは必ず、有効なオブジェクトか関数を参照していなければならない。nullリファレンスというものはあり得ない。なぜならば、nullリファレンスを作る方法というのは、nullポインターを参照することである。nullポインターを参照することは、それ自体が未定義動作となるので、規格の上では、合法にnullリファレンスを作ることはできない。

```

void f()
{
    int * ptr = nullptr ;
    int & ref = *ptr ; // エラー、nullポインターを参照している
}

```

リファレンスのリファレンスは存在しないということはすでに述べた。ただし、見かけ上、リファレンスが重なるという場合が存在する。

typedef、テンプレート仮引数の型、decltype指定子が、Tへのリファレンス型であるとする。リファレンスというのは单なるリファレンスであり、lvalueリファレンスとrvalueリファレンスの両方を含む。その場合、この型に対するlvalueリファレンスを宣言した場合、Tへのlvalueリファレンスになる。一方、この型に対するrvalueリファレンスを宣言した場合、Tへのリファレンス型になる。

これはどういうことかというと、すでにリファレンス型であるtypedef、テンプレート仮引数、decltype指定子に対して、さらにリファレンスの宣言子を付け加えるという意味である。もし、この型に対して、lvalueリファレンスを宣言しようとした場合、元のリファレンス型の如何に関わらず、lvalueリファレンスとなる。rvalueリファレンスを宣言しようとした場合、元のリファレンス型になる。

```
int main()
```

```
{
    typedef int & lvalue_ref ; // lvalueリファレンス
    typedef int && rvalue_ref ; // rvalueリファレンス

    // lvalueリファレンスを宣言しようとした場合
    // 元のリファレンス型が、lvalueリファレンスでもrvalueリファレンスでも、lvalueリファレンスになる
    typedef lvalue_ref & type1 ; // int &
    typedef rvalue_ref & type2 ; // int &

    // rvalueリファレンスを宣言しようとした場合
    // 元のリファレンス型になる
    typedef lvalue_ref && type3 ; // int &
    typedef rvalue_ref && type4 ; // int &&
}
```

換言すれば、lvalueリファレンスは優先され、rvalueリファレンスは無視されるということである。

リファレンスが、内部的にストレージを確保するかどうかは規定されていない。したがって、memcpyなどをつかい、リファレンスを他のストレージの上にコピーすることはできない。

8.3.3 メンバーへのポインター (Pointers to members)

宣言子が以下のように記述されている場合、メンバーへのポインターを意味する。

::_{opt} ネストされたクラス名 * CV修飾子_{opt} 識別子

```
struct S
{
    void func(void) { }
    int value ;
} ;

void ( S:: * ptr_func )( void ) = &S::func ;
int S:: * ptr_value = &S::value ;
```

メンバーへのポインターは、クラスのstaticなメンバーを参照することはできない。また、リファレンス型のメンバーを参照することもできない。

```
struct S
{
    static void func(void) { }
    static int value ;
    int & ref ;
} ;
int S::value ;

void ( S:: * p1 )( void ) = &S::func ; // エラー
void ( *p2 )(void) = &S::func ; // OK
int S:: * p3 = &S::value ; // エラー
```

メンバーへのポインターは、ポインターとは異なる型である。staticメンバー関数へのポインターは、メンバー関数ポインターではなく、通常の関数ポインターである。

8.3.4 配列(Arrays)

宣言子が以下のように記述されている場合、配列を意味する。

識別子 [定数式_{opt}]

```
int a[5] ; // 要素数5のint型の配列
float b[123] ; // 要素数123のfloat型の配列
```

型指定子と、配列以前の宣言子を合わせた型を、配列の要素型(element type)という。要素型は、void以外の基本型、ポインター型、メンバーへのポインター型、クラス、enum型、配列型でなければならない。要素型には、リファレンス型、void型、関数型、10.4 抽象クラスは使えない。

int a[5] ; // void以外の基本型

int * b[5] ; // ポインター型

```

struct S { int value ; } ;
int S::*c[5] ; // メンバーへのポインター型

S d[5] ; // クラス

enum struct E{ value } ;
E e[5] ; // enum型

```

配列に対する配列を作ることができる。これを、多次元配列(multidimensional array)という。

```
int a[3][5][7] ;
```

ここでは、aは $3 \times 5 \times 7$ の配列である。詳しく言うと、aは要素数3の配列である。その各要素は、要素数5の配列である。その各要素は、要素数7のint型の配列である。

配列の定数式は、整数の定数式でなければならない。また、その値は、0より大きくなければならない。

```

int a[0] ; // エラー
int b[-1] ; // エラー

```

定数式は、配列の要素数を表す。今、定数式の値がNであるとすると、配列の要素数はN個であり、0からN-1までの数字を持って表される。その配列のオブジェクトは、連続したストレージ上に、N個の要素型のサブオブジェクトを持っていることになる。

```

int main()
{
    constexpr int N = 5 ;
    int a[N] ; // 要素数5の配列
    a[0] = 0 ; // 最初要素
    a[4] = 0 ; // 最後の要素
}

```

配列の定数式が省略された場合、要素数の不明な配列となる。これは不完全なオブジェクト型である。多次元配列の場合は、最初の配列の定数式のみ省略できる。

定数式の省略された配列は、不完全なオブジェクト型なので、不完全なオブジェクト型の使用が許可されている場所で使うことができる。

```
typedef int a[] ;
typedef int b[][5][7] ; // 最初の定数式のみ省略可
```

また、関数の仮引数の型として使うことができる。

```
void f( int parameter[] ) { }
```

ただし、関数の仮引数の場合、型は、要素型への配列ではなく、要素型へのポインターに置き換えられる。詳しくは、宣言子の8.3.5 [関数](#)を参照。

宣言に初期化子がある場合、配列の定数式を省略できる。この場合、要素数は、初期化子から決定される。

```
int a[] = { 1, 2, 3 } ; // 型はint [3]
```

詳しくは、初期化子の8.5.4 [アグリゲート](#)を参照。

8.3.5 関数(Functions)

関数の宣言方法

関数の宣言子には、文法が二つある。指定子がautoではない場合、以下の文法となる。

```
識別子 ( 仮引数リスト ) CV修飾子opt リファレンス修飾子opt 例外  
指定opt
```

この場合、指定子と、指定子に続く識別子以前の宣言子が、戻り値の型になる。

```
int f( int ) ; // int型の引数を取り、int型の戻り値を返す関数
int * f( int, int ) ; // int型の引数とint型の引数を取り、
int *型の戻り値を返す関数
```

この例では指定子のint型は、戻り値の型を意味する。

指定子がautoの場合、以下の文法で関数を宣言できる。

識別子（仮引数リスト）CV修飾子_{opt} リファレンス修飾子_{opt} 例外
指定_{opt} -> 戻り値の型

この場合、戻り値の型は、指定子ではなく、宣言子の中に記述される。

```
auto f( int ) -> int ; // int型の引数を取り、int型の戻り値を
                          // 戻す関数
auto f( int, int ) -> int * ; // int型の引数とint型の引数を
                                // 取り、int *型の戻り値を返す関数
```

違いは、戻り値の型を指定子で指定するか、宣言子の最後で指定するかである。
同じ名前と型の関数は、どちらの文法で宣言されたとしても、同じ関数となる。

```
// 関数fの宣言
int f( int ) ;
// 同じ関数fの再宣言
auto f( int ) -> int ;
```

指定子にautoを書く、新しい関数の宣言子では、戻り値の型を後置できる。この新しい関数の宣言子によって、戻り値の型名の記述に、仮引数名を使うことができる。例えば、今、二つの引数に、operator *を適用する関数を考える。

```
template < typename T1, typename T2 >
??? multiply( T1 t1, T2 t2 )
{
    return t1 * t2 ;
}
```

ここで、???という部分で、戻り値の型を指定したい。ところが、T1とT2に対してoperator *を適用した結果の型は、テンプレートをインスタンス化するまで分からぬ。

```
struct Mass { } ;
struct Acceleration { } ;
struct Force { } ;
// ニュートンの運動方程式、F=ma
```

```

Force operator *( Mass, Acceleration ) { return Force() ; }

int main()
{
    Mass m ; Acceleration a ;
    Force f = multiply( m, a ) ;
}

```

この例では、Massクラス型とAccelerationクラス型同士をかけ合わせると、結果はForceクラス型となる。すると、関数multiplyの戻り値の型は、式の結果の型でなければならない。一体どうするか。これには、decltypeが使える。decltypeは、式の結果の型を得る指定子である。

```

int main()
{
    Mass m ; Acceleration a ;
    typedef decltype( m * a ) type ; // Force
}

```

ところが問題は、従来の関数の文法では、戻り値を記述する場所では、まだ仮引数名が宣言されていないということである。

```

template < typename T1, typename T2 >
decltype( t1 * t2 ) // エラー、t1とt2は宣言されていない
multiply( T1 t1, T2 t2 )
{
    return t1 * t2 ;
}

```

これを解決するには、やや不自然なメタプログラミングの手法を用いるか、引数を後置できる文法を用いるしかない。新しい関数宣言の文法を用いれば、以下のように書ける。

```

template < typename T1, typename T2 >
auto multiply( T1 t1, T2 t2 ) -> decltype( t1 * t2 )
{
    return t1 * t2 ;
}

```

仮引数リスト

仮引数リストは、コンマで区切られた0個以上の仮引数の宣言である。仮引数リストは、関数を呼び出す際に、実引数の型や数を指定する。

```
void f( int i, float f, double d, char const * c ) ;
```

仮引数リストが空である場合、引数を取らないことを意味する。仮引数が(void)の場合は特別なケースで、仮引数リストが空であることと同義である。

```
// 引数を取らない関数
void f( ) ;
// 同じ意味
void f( void ) ;
```

仮引数の名前は省略できる。8.4 [関数の定義](#)に、仮引数の名前が書かれている場合、その名前が仮引数を表す。

```
// OK、関数fの宣言、仮引数の名前がない
void f( int ) ;
// OK、同じ関数fの宣言、仮引数に名前がある
void f( int x ) ;

// OK、同じ関数fの定義
void f( int x )
{
    x ; // 仮引数を表す
}
```

関数の定義で、仮引数に名前が与えられていない場合、関数の本体から、実引数を使えない。ただし、実引数としては、渡されている。

```
void f( int ) { /* 実引数を使えない */ }

int main()
{
    f( 0 ) ; // 実引数を渡すことには変わりない。
}
```

仮引数の名前は、関数の型には影響を及ぼさない。以下はすべて、同じ関数である。型も同じである。オーバーロードではない。

```
// 型はvoid (int, int)
void f( int foo, int bar ) ;
// 同じ関数fの宣言
void f( int bar, int foo ) ;

// 同じ関数fの定義
void f( int hoge, int /*名前の省略*/ ) { }
```

関数の型

関数の型として意味のあるものは、戻り値の型、仮引数の型リスト、リファレンス修飾子、CV修飾子である。関数の型は、以下のように決定される。

まず、仮引数リストの中のそれぞれの仮引数に対して、指定子と宣言子から、型を決定する。

```
// 仮引数リスト : int
void f( int ) ;
// 仮引数リスト : int *, int *
void f( int *, int * ) ;
```

このようにして得られた仮引数リストの型に対して、以下の変換を行う。

「T型への配列型」は、「T型へのポインター型」に置き換えられる。「関数型」は、「関数ポインター型」に置き換えられる。

```
// void ( int * )
void f( int [3] ) ;
// void ( int * )
void g( int [] ) ;

// void ( void (*)(int) )
void h( void ( int ) ) ;
```

この変換は、関数の本体の中でも有効である。

```

void f( int array[5] )
{
// arrayの型は int *である。int [5]ではない
}

void g( void func( int ) )
{
// funcの型は、void (*)(int)である。void (int)ではない
}

```

このようにして得られた仮引数の型のリストに対し、さらに変換が行われる。これ以降の変換は、関数の型を決定するための変換であり、関数本体の中の引数の型には影響しない。

トップレベルのCV修飾子を消す。

```

// void (int)
void f( const volatile int ) ;
// void (int const *)
void g( int const * const ) ;

// void (int)
void h( const int x )
{
// 関数の本体では、xの型はconst int
}

```

関数の本体の中では、引数の型には、トップレベルのCV修飾子も含まれる。しかし、関数の型としては、仮引数に指定されたトップレベルのCV指定子は影響しない。

仮引数の名前がdecltypeに使われている場合、型は、配列からポインター、関数から関数ポインターへの変換が行われた後の型となる。トップレベルのCV修飾子は残る。

```

// void ( int *, int * )
void f( int a[10], decltype(a) b) ;
// void ( int, int const * ) ;
void g( int const a, decltype(a) * b) ;

```

仮引数の型に影響を与える7.3.1 [ストレージクラス指定子](#)を消す。

```
// void (int)
void f( register int ) ;
```

ただし、現行のC++には、仮引数の型に影響を与えるストレージクラス指定子は、registerだけである。registerの使用は推奨されていない。

上記の変換の結果を、仮引数の型リスト(parameter-type-list)という。

関数が非staticなメンバー関数の場合、CV修飾子があるかどうかが、型として考慮される。

```
struct S
{
    // void S::f(void)
    void f() ;
    // void S::f(void) const
    void f() const ;
} ;
```

関数が非staticなメンバー関数の場合、リファレンス修飾子があるかどうかが、型として考慮される。

```
struct S
{
    // void S::f(void) &
    void f() & ;
    // void S::f(void) &&
    void f() && ;

    // リファレンス修飾子が省略された関数
    void g() ;
} ;
```

メンバー関数に対するCV修飾子とリファレンス修飾子については、9.7.1 非static メンバー関数を参照。リファレンス修飾子については、13.3.1 オーバーロード解決の候補関数と実引数リストも参照。

この他の記述、仮引数名やデフォルト実引数や例外指定は、関数の型には影響しない。

```
// void f(int)
void f( int param = 0 ) noexcept ;
```

関数の戻り値の型には、関数や配列を使うことはできない。

```
// エラー
auto f() -> void (void);
auto f() -> int [5] ;
```

ただし、関数や配列へのポインターやリファレンスは、戻り値の型として使うことができる。

```
// OK、ポインター
auto f() -> void (*)(void) ;
auto f() -> int (*)[5] ;
// OK、リファレンス
auto f() -> void (&)(void) ;
auto f() -> int (&)[5] ;
```

同じ名前で、型の違う関数を、同じスコープ内で複数宣言して使うことができる。これを、関数のオーバーロードという。詳しくは13 [オーバーロード](#)項目を参照。

8.3.6 デフォルト実引数(Default arguments)

仮引数に対して、=に続けて式が書かれていた場合、その式は、デフォルト実引数として用いられる。デフォルト実引数は、関数呼び出しの際に、実引数が省略された場合、かわりに引数として渡される。

```
void f( int x = 123, int y = 456 ) { }

int main()
{
    f( ) ; // f( 123, 456 )と同じ
    f( 0 ) ; // f( 0, 456 )と同じ
    f( 1, 2 ) ; // デフォルト実引数を使用しない
}
```

式は、単にリテラルでなくてもよい。

```
int f() { return 0 ; }
void g( int x = f() ) { }
```

関数の呼び出しの際に、前の実引数を省略して、との実引数を指定することはできない。

```
void f( int x = 123, int y = 456 ) { }
int main()
{
    f( , 0 ) ; // エラー
}
```

デフォルト実引数は、関数のパラメーターパックに指定することはできない。

```
template < typename ... Types >
void f( Types .. args = 0 ) ; // エラー
```

デフォルト実引数は、後から付け加えることも出来る。ただし、再宣言してはいけない。たとえ同じ式であったとしても、再宣言はできない。

```
void f( int ) ;
// OK、デフォルト実引数を付け加える
void f( int x = 0 ) ;
// エラー、デフォルト実引数の再宣言
void f( int x = 0 ) ;
// 定義
void f( int x ) { }
```

可読性のためには、デフォルト実引数は、関数の最初の宣言に記述すべきである。

```
void f( int = 0 ) ; // 宣言
void f( int ) { } // 定義
```

デフォルト実引数が使われる場合、式は、関数呼び出しの際に、毎回評価される。評価の順序は規定されていない。

8.4 関数の定義 (Function definitions)

関数の定義:

関数の宣言 関数の本体

関数の本体:

コンストラクター初期化子_{opt} 複合文
 関数tryブロック
 = default ;
 = delete ;

関数の定義とは、関数の宣言に続けて、関数の本体の書かれている関数宣言である。

```
void f() ; // 宣言
void f() {} // 定義

void // 指定子
g() // 宣言子
{} // 関数の本体
```

関数は、名前空間スコープか、クラススコープの中でのみ、定義できる。

```
// グローバル名前空間スコープ
void f() {}
// クラススコープ
struct S { void f() {} } ;

void g()
{
    void h(){} ; // エラー、関数のブロックスコープの中では定義で
    きない
}
```

コンストラクター初期化子は、クラスのコンストラクターで用いる。詳しくは、クラスの12.1 [コンストラクター](#)と、クラスの12.6 [初期化](#)を参照。

関数の定義には、複合文の他に、関数tryブロックを使うこともできる。

```
void f()
try
{
// 関数の本体
}
catch ( ... )
{
// 例外ハンドラー
}
```

関数tryブロックについて詳しくは、15 [例外](#)を参照。

関数の本体では、`_func_`(ダブルアンダースコアであることに注意)という名前の変数が、以下のようにあらかじめ定義されている。

```
static const char __func__[] = "関数名" ;
```

「関数名」とは、実装依存の文字列である。C++規格では、この実装依存の文字列の意味は、何も規定されていない。この機能はC言語から取り入れられたものである。C99規格では、関数本体の属する関数の名前を表す、実装依存の文字列とされている。いずれにせよ、具体的な文字列については、何も規定されていない。

8.4.1 default定義 (Explicitly-defaulted functions)

```
関数の宣言 = default ;
```

関数の宣言に続けて、`= default ;`と書く関数の定義を、明示的にデフォルト化された関数(Explicitly-defaulted functions)という。本書では、default定義と呼ぶ。

明示的にデフォルト化された関数は、12 [特別なメンバー関数](#)でなければならない。また、暗黙的に定義された場合と同等の型でなければならない。デフォルト実引数と例外指定は使えない。

明示的なデフォルト化は、暗黙の定義と同等の定義を、明示的に定義するための機能である。

```
struct S
{
    S() = default ; // default定義
    S(int){ }
} ;
```

この例では、もし、明示的なデフォルト化が書かれていない場合、Sのデフォルトコンストラクターは、暗黙的に定義されない。

明示的にデフォルト化された関数と、暗黙的に定義される関数とをあわせて、デフォルト化された関数(defaulted function)という。

8.4.2 delete定義(Deleted definitions)

関数の宣言 = `delete ;`

関数の宣言に続けて、`= delete ;`と書く関数の定義を、削除された定義(Deleted definitions)という。また、本書では、分かりやすさのため、`delete`定義と呼ぶ。

削除された関数定義は、宣言として存在しているが、定義のない関数である。宣言としては存在しているので、名前解決やオーバーロード解決、テンプレートのインスタンス化の際には、通常通り考慮される。

ただし、削除された関数定義を、宣言以外の方法で参照した場合、エラーとなる。参照というのは、明示的、暗黙的に関数を呼び出すことや、関数へのポインター、メンバーポインター、リファレンスを得ることなどである。また、たとえ未評価式の中であっても、参照した場合エラーとなる。

```
// 削除された関数の定義
void f() = delete ;

void f() ; // OK、宣言はできる

int main()
{
    f() ; // エラー、削除された関数の呼び出し
    &f ; // エラー、削除された関数のポインターを得ようとしている
    void (& ref )(void) = f ; // エラー、削除された関数のリファレンスを得ようとしている
    typedef decltype(f) type ; // エラー、未評価式の中で、削除された関数を参照している
}
```

削除された関数の定義は、関数の最初の宣言でなければならない。

```
void f() ; // 削除された定義ではない
void f() = delete ; // エラー、最初の宣言でなければならない

void g() = delete ; // OK、最初の宣言
void g() ; // OK、再宣言
```

ただし、関数テンプレートの特殊化の場合、最初の特殊化の宣言となる。

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

// 特殊化
template < >
void f<int>(int) = delete ;

int main()
{
    f(0) ; // エラー
    f(0.0) ; // OK
}
```

関数がオーバーロードされている場合、オーバーロード解決によって、削除された関数定義が参照される場合のみ、エラーとなる。

```
void f( int ) {} // 削除されていない関数
void f( double ) = delete ; // 削除された関数定義

int main()
{
    f( 0 ) ; // OK、void f(int)は削除されていない
    f( 0.0 ) ; // エラー、削除された関数の参照
}
```

関数のオーバーロード、関数テンプレート、削除された定義を組み合わせると、非常に面白い事ができる。

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

```
// 特殊化でdoubleでインスタンス化された場合の定義を削除
template < >
void f<double>(double) = delete ;

void call_f()
{
// doubleでインスタンス化した場合、エラーになる
    f( 0 ) ; // OK
    f( true ) ; // OK
    f( 0.0 ) ; // エラー
}

// あらゆるインスタンス化を削除
template < typename T >
void g( T ) = delete ;

// 削除されていない定義
template < >
void g< double >( double ) { }

void call_g()
{
// double以外でインスタンス化した場合、エラーになる
    g( 0 ) ; // エラー
    g( true ) ; // エラー
    g( 0.0 ) ; // OK
}

// 非テンプレートな関数
void h( int ) { }

// 関数テンプレートの定義を削除
template < typename T >
void h( T ) = delete ;

void call_h()
{
// intへの標準型変換を禁止
    h( 0 ) ; // OK、非テンプレートな関数を呼び出す
    h( true ) ; // エラー、関数テンプレート
    h( 0.0 ) ; // エラー、関数テンプレート
}

void i( int ) = delete ;
```

```

void i( double ) { }

void call_i()
{
// intからdoubleへの標準変換をエラーにする
    i( 0 ) ; // エラー
    i( true ) ; // OK
    i( 0.0 ) ; // OK
}

```

このように、削除された定義を使うことで、意図しない標準型変換やインスタンス化を阻害できる。

削除された関数定義の具体的な使い方は、実に様々な例が考えられる。ここでは、その一部を挙げる。

クラスのコンストラクターを制御する。

```

struct Boolean
{
    Boolean( ) = delete ;
    Boolean( bool ) { }

    template < typename T >
    Boolean( T ) = delete ;
} ;

int main()
{
    Boolean a = true ; // OK
    Boolean b = 123 ; // エラー
    Boolean c = &a ; // エラー
}

```

Booleanクラスは、必ず、ひとつのbool型の引数で初期化しなければならない。このクラスの初期化の際に、bool以外の型を渡すと、テンプレートのインスタンス化とオーバーロード解決により、関数テンプレート版のコンストラクターが優先される。しかし、定義は削除されているため、エラーとなる。結果的に、暗黙の型変換を禁止しているのと同じ意味となる。そのため、意図しない数値やポインターでの初期化という、つまらないバグを防げる。

クラスのオブジェクトをnewで生成することを禁止する。

```

struct Do_not_new
{
    void *operator new(std::size_t) = delete;
    void *operator new[](std::size_t) = delete;
} ;

int main()
{
    Do_not_new a ; // OK
    Do_not_new * ptr = new Do_not_new ; // エラー
    Do_not_new * array_ptr = new Do_not_new[10] ; // エラー
}

```

何らかの理由で、あるクラスのオブジェクトを、newで生成してほしくないとする。削除された定義を使えば、あるクラスに対して、newを禁止できる。

クラスのコピーを禁止する。

```

struct move_only
{
    move_only() = default ;
    ~move_only() = default ;

    move_only( const move_only & ) = delete ;
    move_only( move_only && ) = default ;
    move_only & operator = ( const move_only & ) = delete ;
    move_only & operator = ( move_only && ) = default ;
} ;

int main()
{
    move_only m ;
    move_only n ;

    n = m ; // エラー、コピーは禁止されている
    n = std::move(m) ; // OK、ムーブはできる
}

```

クラスmove_onlyは、ムーブができるが、コピーはできないクラスになる。

8.5 初期化子(Initializers)

宣言子の宣言する変数に対して、初期値を指定することができる。この初期値を指定するための文法を、初期化子(Initializer)という。この初期化子の項目で解説している初期化は、宣言文以外にも、関数の仮引数を実引数で初期化することや、関数の戻り値の初期化、new式やクラスのメンバー初期化などにも適用される。

初期化子の文法と意味について解説する前に、まず基本的な三つの初期化について解説しなければならない。ゼロ初期化、デフォルト初期化、値初期化である。

8.5.1 ゼロ初期化(zero-initialize)

ゼロ初期化(zero-initialize)とは、T型のオブジェクトやリファレンスに対して、Tがスカラー型の場合、整数の定数、0を、T型に変換して初期化する。

```
static int x ; // 0で初期化される
static float f ; // 0がfloat型に変換されて初期化される
static int * ptr ; // 0がnullポインターに変換されて初期化される
```

Tがunionではないクラス型の場合、非staticなデータメンバーと基本クラスのサブオブジェクトが、それぞれゼロ初期化される。また、アライメント調整などのための、オブジェクト内のパディングも、ゼロビットで初期化される。

```
struct Base { int x ; } ;
struct Derived : Base
{
    int y ;
} ;

// 非staticなデータメンバー、基本クラスのサブオブジェクトが、それぞれゼロ初期化される
static Derived d ;
```

この例では、Derivedのデータメンバーであるyと、基本クラスであるBaseのオブジェクトがゼロ初期化される。Baseをゼロ初期化するということは、Baseのデータメンバーであるxもゼロ初期化される。

Tがunion型の場合、オブジェクトの最初の、非staticな名前のつけられているデータメンバーが、ゼロ初期化される。また、アライメント調整などのための、オブジェクト内のパディングも、ゼロビットで初期化される。

```
union U
{
    int x ;
    double d ;
} ;
```

このunionのオブジェクトをゼロ初期化した場合、U::xがゼロ初期化される。

Tが配列型の場合、各要素がそれぞれゼロ初期化される。

Tがリファレンス型の場合、初期化は行われない。

8.5.2 デフォルト初期化(default-initialize)

デフォルト初期化(default-initialize)とは、T型のオブジェクトに対して、

Tがクラス型の場合、Tのデフォルトコンストラクターが呼ばれる。デフォルトコンストラクターにアクセス出来ない場合は、エラーである。

```
class A
{
public :
    A() { }
} ;

class B
{
private :
    B() { }
} ;

int main()
{
    A a ; // デフォルトコンストラクターが呼ばれる
    B b ; // エラー、デフォルトコンストラクターにアクセスできない
}
```

Tが配列型の場合、各要素がそれぞれデフォルト初期化される。

上記以外の場合、初期化は行われない。

```
int main()
{
    int x ; // 初期化は行われない
}
```

`const`修飾された型をデフォルト初期化する場合、型はユーザー定義コンストラクターを持つクラス型でなければならない。

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

int main()
{
    int const a ; // エラー、intはユーザー定義コンストラクターを持つクラス型ではない
    X const b ; // エラー、Xはユーザー定義コンストラクターを持つクラス型ではない

    Y const c ; // OK
}
```

リファレンス型をデフォルト初期化しようとした場合、エラーになる。

8.5.3 値初期化(value-initialize)

値初期化(value-initialize)とは、T型のオブジェクトに対して、

Tがクラス型で、ユーザー提供のコンストラクターを持つ場合、Tのデフォルトコンストラクターが呼ばれる。デフォルトコンストラクターにアクセス出来ない場合は、エラーである。

```
struct S
{
    S() { }
    int x ;
} ;
```

クラスSのオブジェクトを値初期化した場合、Sのデフォルトコンストラクターが呼ばれ

る。xの値は不定である。

Tが、unionではないクラス型で、ユーザー提供のコンストラクターを持たない場合、オブジェクトはゼロ初期化される。もし、暗黙的に定義されたコンストラクターが、トリビアルではない場合、コンストラクターが呼ばれる。

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

struct B
{
    // ユーザー提供のコンストラクターがない
    // 暗黙に定義されたコンストラクターはトリビアルではない
    A a ;
    int x ;
} ;
```

クラスBのオブジェクトを値初期化した場合、Aのデフォルトコンストラクターが呼ばれる。また、xはゼロ初期化される。

Tが配列型の場合、各要素がそれぞれ値初期化される。

上記以外の場合、オブジェクトはゼロ初期化される。

リファレンス型をゼロ初期化しようとした場合、エラーとなる。

初期化の文法と方法

staticストレージの期間を持つオブジェクトは、プログラムの開始時に、必ずゼロ初期化される。その後、必要であれば、初期化される。

```
struct S
{
    S() : x(1) { }
    int x ;
} ;

S s ; // staticストレージの期間を持つオブジェクト
```

ここでは、Sのデフォルトコンストラクターが実行される前に、データメンバーのxはゼロ初期化されている。

初期化子が空の括弧、()、であるとき、オブジェクトは値初期化される。ただし、通常の宣言文では、初期化子として空の括弧を書く事はできない。なぜならば、空の括弧は、関数の宣言であるとみなされるからだ。

```
int main()
{
    int x(); // int (void)型の関数xの宣言
}
```

初期化子としての空の括弧は、5.6.4 new、5.5.3 関数形式の明示的型変換、12.6.2 基本クラスとデータメンバーの初期化子で使うことができる。

```
struct S
{
    S()
    : x() // メンバー初期化子
    { }
    int x ;
}

int main()
{
    new int(); // new、初期化子として空の括弧
    int(); // 関数形式のキャスト
}
```

初期化子が指定されていない場合、オブジェクトはデフォルト初期化される。

```
struct S { } ;

int main()
{
    S s; // デフォルト初期化される
}
```

デフォルト初期化では、すでに説明したように、クラス以外の型は、初期化が行われない。初期化が行われないオブジェクトの値は、不定である。

```
int main()
```

```
{
    int i ; // 値は不定
    double d ; // 値は不定
}
```

ただし、staticやthreadストレージの有効期間を持つオブジェクトは、ゼロ初期化されることが保証されている。

```
// グローバル名前空間
int x ; // staticストレージ、ゼロ初期化される
thread_local int y ; // threadストレージ、ゼロ初期化される
```

以下のような文法の初期化子を、コピー初期化(copy-initialization)という。

```
T x = a ;
```

これに加えて、関数の実引数を渡す、関数のreturn文、例外のthrow文、例外を受ける、アグリゲートのメンバーの初期化も、コピー初期化という。この「コピー」という言葉は、コピー構造体やコピー代入演算子とは関係がない。コピー初期化でも、コピーではなく、ムーブされることもある。

以下のような文法の初期化子を、直接初期化(direct-initialization)という。

```
T x(a) ;
T x{a} ;
```

これに加えて、5.6.4 [new](#)、5.5.9 [static_cast](#)、5.5.3 [関数形式のキャスト](#)、12.6.2 [基本クラスとデータメンバーの初期化子](#)も、直接初期化という。

初期化子の意味は、以下のように決定される。オブジェクトの型を、目的の型(destination type)とし、初期化子の型を、元の型(source type)とする。元の型は、初期化子が、ひとつの初期化リストか、括弧で囲まれた式リストの場合は、存在しない。

```
// 目的の型はT
// 元の型はint
T x = 0 ;

// 目的の型はT
// 元の型は存在しない（ひとつの初期化リスト）
T x = { } ;
```

```

T x({ }) ;
T x{ } ;

// 目的の型はT
// 元の型は存在しない（括弧で囲まれた式リスト）
T x(1, 2, 3)

```

初期化子が、ひとつの初期化リストの場合、8.5.7 リスト初期化される。

```

// リスト初期化される
T x = { } ;
T x({ }) ;
T x{ } ;

```

目的の型がリファレンスの場合、8.5.6 リファレンスを参照。

目的の型が、char、signed char、unsigned char、char16_t、char32_t、wchar_tの配列で、初期化子が文字列リテラルの場合、8.5.5 文字配列を参照。

初期化子が空の()の場合、オブジェクトは値初期化される。

ただし、通常の変数の宣言では、空の()を書く事はできない。なぜならば、空の括弧は、関数の宣言とみなされるからだ。

```

// int (void)型の関数xの宣言
// int x(void) ; と同じ
int x() ;

```

空の()を書く事ができる初期化子には、5.5.3 関数形式のキャスト、5.6.4 new、12.6.2 メンバー初期化がある。

```

// 関数形式のキャスト
int x = int() ;
// new
int * p = new int() ;

// メンバー初期化
class C
{
    int member ;
    C() : member()
}

```

```
{ }
};
```

`{}`や、空の初期化リストを含む(`{}`)は、文法上曖昧とならないので、宣言文でも書ける。詳しくは8.5.7 [リスト初期化](#)で解説するが、この場合も、オブジェクトは値初期化される。

```
int x{} ; // int型の変数xの宣言と定義と初期化子
```

それ以外の場合で、目的の型が配列型の場合、エラーとなる。これは、初期化子がある場合で、上記のいずれにも該当しない場合を指す。

```
int a[5] = 0 ; // エラー、int [5]型は、int型で初期化できない
```

目的の型がクラス型で、初期化子が直接初期化である場合、最も最適なコンストラクターが、オーバーロード解決によって選ばれる。

```
struct Elem { } ;

struct S
{
    S( int ) { }
    S( Elem ) { }
};

int main()
{
    S s1( 0 ) ; // S::S(int)
    S s2( 0.0 ) ; // S::S(int)、標準型変換による

    Elem elem ;
    S s3( elem ) ; // S::S(Elem)
}
```

コンストラクターが適用できなかったり、曖昧である場合は、エラーとなる。

```
struct S
{
    S( long ) { }
```

```

        S( long long ) { }

} ;

struct Elem { } ;

int main()
{
    S s1( 0 ) ; // エラー、コンストラクターが曖昧
    Elem elem ;
    S s2( elem ) ; // エラー、適切なコンストラクターが見つからな
    い
}

```

目的の型がクラス型で、初期化子がコピー初期化で、初期化子の型が目的の型か、その派生クラス型である場合、直接初期化と同じ方法で初期化される。初期化子の型が目的の型か、その派生クラス型でない場合は、後述。

```

struct Base { } ;
struct Derived : Base { } ;

int main()
{
    Base b1 = Base() ;
    Base b2 = Derived() ;
}

```

目的の型がクラス型で、初期化子がコピー初期化の場合(ただし上記を除く)、ユーザ一定義の型変換が試みられ、最適な候補が、オーバーロード解決によって選ばれる。

```

struct Elem { } ;
struct Integer
{
    operator int () const { return 0 ; }
} ;

struct S
{
    S( int ) { }
    S( Elem ) { }
} ;

int main()

```

```

{
    S s1 = 0 ; // コンストラクターによる変換、S::S(int)
    S s2 = 0.0 ; // 標準型変換の結果、S::S(int)
    S s3 = Elem() ; // コンストラクターによる変換、S::S(Elem)
    S s4 = Integer() ; // Integer::operator int()が呼ばれ、次
    [S::S(int)]
}

```

型変換できなかったり、型変換が曖昧である場合は、エラーとなる。

```

struct A { } ;
struct B { } ;

struct C
{
    C( long ) { }
    C( long long ) { }
} ;

int main()
{
    A a ;
    B b = a ; // エラー、変換関数が見つからない
    C c = 0 ; // エラー、変換関数が曖昧
}

```

それ以外の場合、つまり、目的の型がクラス型ではない場合で、初期化子の型がクラスであった場合、目的の型に型変換が試みられ、最適な候補がオーバーロード解決によって選択される。型変換ができない場合や、曖昧な場合は、エラーとなる。

```

struct S
{
    operator int() { return 0 ; }
} ;

int main()
{
    S s ;
    int x = s ; // OK、S::operator int()が呼ばれる
}

```

それ以外の場合、つまり、目的の型も初期化子の型も、クラスではない場合、標準型変換が試みられる。ユーザー定義の変換関数は考慮されない。型変換ができない場合、エラーとなる。

```
int main()
{
    int i = 0 ;
    float f = i ; // OK、整数から浮動小数点数へ型変換される
    int * p = f ; // エラー、floatはint *に変換できない
}
```

8.5.4 アグリゲート(Aggregates)

アグリゲート(aggregate)とは、配列か、いくつかの制約を満たしたクラスである。クラスの場合、ユーザー定義のコンストラクター、非staticデータメンバーの初期化子、privateおよびprotectedな非staticデータメンバー、基本クラス、virtual関数が存在しないものだけを、アグリゲートという。

以下は、アグリゲートの例である。

```
struct Aggregate
{
    // 非staticデータメンバーはすべてpublic
    int x ;
    float y ;

    // 非virtualなメンバー関数
    void f() { }

    // staticデータメンバーはpublicでなくてもよい
    private :
        static int data ;
} ;
int Aggregate::data ;

int a[10] ;
Aggregate b ;
Aggregate c[10] ;
```

以下はアグリゲートの条件を満たさないクラスの例である。

```

struct Base { } ;

struct NonAggregate
    : Base // 基本クラス
{
// ユーザー定義のコンストラクター
    NonAggregate() { }
// 非staticデータメンバーの初期化子
    int x = 0 ;
// privateおよびprotectedな非staticデータメンバー
private :
    int y ;
protected :
    int z ;
// virtual関数
    virtual void f() { }
} ;

```

配列は必ずアグリゲートである。たとえアグリゲートではないクラス型であっても、そのクラスの配列型は、アグリゲートとなる。

```

// アグリゲートではないクラス
struct NonAggregate
{
    NonAggregate() { }
} ;

NonAggregate a[3] ; // アグリゲート

```

アグリゲートが初期化リストで初期化される場合、初期化リストの対応する順番の要素が、それぞれアグリゲートのメンバーの初期化に用いられる。メンバーはコピー初期化される。

```

// 配列の例
int a[3] = { 1, 2, 3 } ;
// 各要素の初期化、a[0] = 1, a[1] = 2, a[2] = 3

// クラスの例
struct S
{

```

```

        int x ; int y ;
        double d ;
    } ;

S s = { 1, 2, 3.0 } ;
// 各メンバーの初期化は、s.x = 1, s.y = 2, s.d = 3.0

// クラスの配列の例
S sa[3] =
{
    { 1, 2, 3.0 },
    s,
}
; // アグリゲートではないクラスの配列の例

class C
{
    int value ;
public :
    C(int value) : value(value) { }
}
;

C obj( 3 ) ; // クラスのオブジェクト

// 配列はアグリゲート
C ca[3] = { 1, 2, obj } ;
// コピー初期化を適用した結果、C::C(int)が呼び出される
// ca[0]は1、ca[1]は2、ca[2]はobjで、それぞれ初期化される

```

初期化の際に、縮小変換が必要な場合、エラーである。

```

int a( 0.0 ) ; // OK
int b{ 0.0 } ; // エラー、縮小変換が必要

struct S { int x ; } ;
S s = { 0.0 } ; // エラー、縮小変換が必要

```

初期化リストが、内部に初期化リストを含む場合、アグリゲートの対応するメンバーは、初期化リストによって初期化される。

```
struct Inner { int x ; int y ; } ;
```

```

struct Outer
{
    int x ;
    Inner obj ;
} ;

Outer a = { 1, { 1, 2 } } ;
// a.x = 1, a.obj = { 1, 2 }
// a.obj.x = 1, a.obj.y = 2

```

要素数不定の配列のアグリゲートが、初期化リストで初期化される場合、配列の要素数は、初期化リストの要素数になる。

```

int a[] = { 1 } ; // int [1]
int b[] = { 1, 2, 3 } ; // int [3]
int c[] = { 1, 2, 3, 4, 5 } ; // int [5]

```

staticデータメンバーと無名ビットフィールドは、アグリゲートのリスト初期化の際には、メンバーとして考慮されない。つまり、初期化リストの要素の順番などにも影響しない。

```

struct S
{
    int x ;
    static int static_data_member ; // staticデータメンバー
    int y ;
    int : 8 ; // 無名ビットフィールド
    int z ;
} ;

int S::static_data_member ;

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

```

この例では、`static_data_member`と、`y`と`z`との間にある無名ビットフィールドは、リスト初期化の際には、メンバーとして考慮されない。

もし、アグリゲートのメンバーよりも、初期化リストの要素の方が多い場合は、エラーとなる。

```
int a[3] = { 1, 2, 3, 4 } ; // エラー
```

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

S s = { 1, 2, 3, 4 } ; // エラー
```

もし、アグリゲートのメンバーよりも、初期化リストの要素の方が少ない場合、明示的に初期化されていないアグリゲートのメンバーは、すべて値初期化される。値初期化では、アグリゲートの条件を満たす型はすべて、ゼロ初期化される。したがって、単にゼロで初期化されると考えても差し支えない。

```
int a[5] = { 1, 2, 3 } ;
// a[0] = 1, a[1] = 2, a[2] = 3
// a[4]とa[5]は、値初期化される

struct S { int x ; int y ; } ;

S s { 1 } ;
// s.x = 1
// s.yは値初期化される
```

空の初期化リストでは、値初期化される。

```
// メンバーはすべて値初期化される
int a[5] = { } ;
struct S
{
    int x ; int y ;
} ;

S s = { } ;
```

既存のコードでは、アグリゲートのすべてのメンバーをゼロ初期化するために、{0}という初期化リストが使われていることがある。

```
int x[100] = {0} ; // すべてゼロで初期化
```

これは、C++ではなく、C言語に由来するコードである。C言語では、空の初期化リストを

書く事がない。そのため、Cプログラマは、`{0}`と書くのである。多くのC++プログラマは、C言語もよく知っているので、既存のコードでは、慣習的に`{0}`が使われている。しかし、C++では、`{0}`と書く必要はない。`{}`で十分である。

アグリゲートが、内部に別のアグリゲートを持っている場合、初期化リストは、その内部のアグリゲートを無視することはできない。

```
struct SubAggregate { } ;

struct Aggregate
{
    int m1 ;
    SubAggregate s1 ;
    int m2 ;
    SubAggregate s2 ;
    int m3 ;
    SubAggregate s3 ;
} ;

SubAggregate s ;

Aggregate a =
{
    1,
    { }, // 空の初期化リスト
    2,
    s, // オブジェクトの変数
    3,
    SubAggregate() // 関数形式のキャスト
} ;
```

このように、内部のアグリゲートに続くメンバーを初期化したい場合は、たとえ空の初期化リストでも、必ず書かなければならない。もちろん、その内部のアグリゲートをコピー初期化できる型の値ならば、なんでも使える。

アグリゲート内のリファレンスのメンバーが、明示的に初期化されなかった場合、エラーとなる。

```
struct S
{
    int & ref ; // リファレンスのメンバー
} ;

int x ;
```

```
S s1 = { x } ; // OK、リファレンスを初期化している
S s2 = { } ; // エラー、リファレンスが初期化されていない
```

多次元配列は、初期化リストのネストによって初期化することができる。

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

int b[3][3][3] =
{
    { { 1, 2, 3 }, { 1, 2, 3 }, { 1, 2, 3 } },
    { { 1, 2, 3 }, { 1, 2, 3 }, { 1, 2, 3 } },
    { { 1, 2, 3 }, { 1, 2, 3 }, { 1, 2, 3 } }
};
```

8.5.5 文字配列(Character arrays)

`char`(`signed char`と`unsigned char`も含む), `wchar_t`, `char16_t`, `char32_t`の配列は、それぞれ、対応する文字列リテラルで初期化できる。

```
char str[6] = "hello" ;
char utf8_str[6] = u8"hello" ;
wchar_t wide_str[6] = L"hello" ;
char16_t utf16_str[6] = u"hello" ;
char32_t utf32_str[6] = U"hello" ;
```

文字列リテラルは、null文字を含むということに注意しなければならない。文字列リテラル、“hello”の型は、`char const [6]`である。

文字配列の要素数が指定されていない場合、初期化子の文字列リテラルの要素数になる。

```
// 要素数は6
char str[] = "hello" ;
```

配列の要素数より、文字列リテラルの要素数の方が多い場合、エラーとなる。

```
char str[5] = "hello" ; // エラー
```

配列の要素数より、文字列リテラルの要素数の方が少ない場合、明示的に初期化されない要素は、ゼロ初期化される。

```
// 以下の二行は同等
char str[10] = "hello" ;
char str[10] = { 'h', 'e', 'l', 'l', 'o', '\0', 0, 0, 0, 0 } ;
```

8.5.6 リファレンス(References)

`T &`を、「`T`型への`lvalue`リファレンス」という。`T &&`を、「`T`型への`rvalue`リファレンス」という。これらを二つまとめて、「`T`型へのリファレンス」という。`lvalue`リファレンスは、`lvalue`へのリファレンスであり、`lvalue`で初期化する。`rvalue`リファレンスは、`rvalue`へのリファレンスであり、`rvalue`で初期化する。そのため、このような名前になっている。

`T`型へのリファレンスは、`T`型のオブジェクトか関数、あるいは、`T`型に変換可能なオブジェクトで初期化できる。

```
struct Integer
{
    int object ;
    operator int & () { return object ; }
} ;

void f(void) { }

int main()
{
    // オブジェクトによる初期化
    int int_object ;
    int & ref_int = int_object ;

    // 関数による初期化の例
    void ( &ref_function )(void) = f ;

    // int型に変換可能なオブジェクトによる初期化
    Integer integer_object ;
```

```
int & ref_integer = integer_object ;
}
```

リファレンスは、必ず初期化されなければならない。リファレンスの参照先を変更することはできない。リファレンスは、参照先のオブジェクトと同じように振舞う。

```
int main()
{
    int x ;
    int & ref = x ;
    ref = 0 ; // x = 0 と同じ
    int y = 0 ;
    ref = y ; // x = y と同じ、参照先は変わらない
}
```

関数の仮引数、関数の戻り値の型、クラス定義の中のクラスメンバー宣言、extern指定子が明示的に使われている宣言では、リファレンスの初期化子を省略できる。

```
// 関数の仮引数
void f( int & ) ;

// 関数の戻り値の型
int & g() ;
auto g() -> int & ;

// クラス定義のクラスメンバーの宣言
struct S { int & ref ; } ;

// extern指定子 (refは別の場所で定義されている)
extern int & ref ;
```

もちろん、これらのリファレンスも、使うときには、初期化されていなければならない。

リファレンスの具体的な初期化について説明する前に、リファレンス互換(reference-compatible)を説明しなければならない。ある型、T1とT2があるとする。もし、T1が、T2と同じ型か、T2の基本クラス型であり、T1のCV修飾子が、T2のCV修飾子と同等か、それ以上の場合、T1はT2に対してリファレンス互換である。

一般に、T1がT2に対してリファレンス互換である場合、T1へのリファレンスは、T2へのリファレンスで初期化できる。

```

// AとBとは、お互いにリファレンス互換ではない
struct A { } ;
struct B { } ;

A a ;
B & r1 = a ; // エラー、リファレンス互換ではない

// BaseはDerivedに対して、リファレンス互換である
// DerivedはBaseに対して、リファレンス互換ではない
// 基本クラスは派生クラスに対してリファレンス互換であるが、
// 派生クラスは基本クラスに対してリファレンス互換ではない
struct Base { } ;
struct Derived : Base { } ;

Base base ;
Derived derived ;

Base & r2 = derived ; // OK
Derived & r3 = base ; // エラー

// Non_const_intはConst_intに対して、リファレンス互換ではない
// Const_intはNon_const_intに対して、リファレンス互換である
// CV修飾子が同じか、それ以上である場合、リファレンス互換である
// CV修飾子が少ない場合、リファレンス互換ではない
typedef int Non_const_int ;
typedef int const Const_int ;

Non_const_int nci ;
Const_int ci ;

Non_const_int & r4 = ci ; // エラー
Const_int & r5 = nci ; // OK

```

T型へのlvalueリファレンスは、リファレンス互換なlvalueで初期化できる。

```

struct Base { } ;
struct Derived : Base { } ;

int main()
{
    int object ;
    int & r1 = object ;

```

```
Base base ;
Base & r2 = base ;
// 派生クラスのlvalueでも初期化できる
Derived derived ;
Base & r3 = derived ;

int const const_object = 0 ;
// OK
int const & r5 = const_object ;
int const volatile & r6 = const_object;
}
```

基本的に、lvalueリファレンスはlvalueでしか初期化できない。xvalueやprvalueで初期化することはできない。また、CV修飾子を取り除くことはできない。以下はエラーの例である。

```
#include <utility>

struct A { } ;
struct B { } ;

struct Base { } ;
struct Derived : Base { } ;

int main()
{
// エラー、BはAに対してリファレンス互換ではない
    A a ;
    B & b = a ;

    int & r1 = 0 ; // エラー、prvalueでは初期化できない
    int x ;
    int & r2 = std::move( x ) ; // エラー、xvalueでは初期化できない

    Base base ;
    Derived & r3 = base ; // エラー、派生クラスは基本クラスに対してリファレンス互換ではない

    int const ci = 0 ;
    int & r4 = ci ; // エラー、CV修飾子が少ないので、リファレンス互換ではない
}
```

T型へのlvalueリファレンスは、リファレンス互換な型のlvalueに暗黙的に変換できるクラス型のオブジェクトで初期化できる。

```
// int &に暗黙的に変換できるクラス
struct Integer
{
    int object ;
    operator int & () { return object ; }
} ;

int main()
{
    Integer object ;
    int & ref = object ; // OK、暗黙的にリファレンス互換な
lvalueに変換できる
} ;
```

volatileではない、const T型へのlvalueリファレンスは、rvalueでも初期化できる。rvalueとは、prvalueとxvalueである。

```
#include <utility>

int f() { return 0 ; }

int main()
{
    // OK、prvalueで初期化できる
    int const & r1 = f() ;
    // OK、xvalueで初期化できる
    int object ;
    int const & r2 = std::move( object ) ;

    // エラー、たとえconstでも、volatileであってはならない
    int volatile const & r3 = f() ;
}
```

volatileではない、const T型へのlvalueリファレンスが、rvalueでも初期化できるというのは、非常に異質なルールである。これは、C++にまだrvalueリファレンスがなかった時代に、リファレンスでrvalueをも参照する必要があったために導入されたルールである。

T型へのrvalueリファレンスは、リファレンス互換なrvalueで初期化できる。rvalueとは、

xvalueとprvalueのことである。

```
#include <utility>

int f() { return 0 ; }

int main()
{
    // OK、prvalueで初期化できる
    int && prvalue_ref = f() ;

    // OK、xvalueで初期化できる
    int object ;
    int && xvalue_ref = std::move( object ) ;
}
```

rvalueリファレンスは、必ずrvalueで初期化しなければならない。lvalueでは初期化できない。

```
int object ;
int && ref = object ; // エラー、lvalueでは初期化できない
```

rvalueリファレンス自体はlvalueであるということに、注意しなければならない。もし、rvalueリファレンスをrvalueリファレンスで初期化したいのならば、明示的にrvalueにしなければならない。

```
#include <utility>

int f() { return 0 ; }

int main()
{
    int && rvalue_ref = f() ;
    int && r1 = rvalue_ref ; // エラー、rvalue_ref自体はlvalueである。
    int && r2 = std::move( rvalue_ref ) ; // OK、xvalueでの初期化
}
```

T型へのrvalueリファレンスは、リファレンス互換な型のrvalueに暗黙的に変換できる

ラス型のオブジェクトで初期化できる。

```
#include <utility>

// int &&に暗黙的に変換できるクラス
struct Integer
{
    int object ;
    operator int && () { return std::move(object) ; }
} ;

int main()
{
    Integer object ;
    int && ref = object ; // OK、暗黙的にリファレンス互換な
    rvalueに変換できる
}
```

rvalueリファレンスの初期化子が、リテラルの場合、一時オブジェクトが生成され、参照される。

```
int main()
{
    int && ref = 0 ; // 一時オブジェクトが生成される
}
```

8.5.7 リスト初期化(List-initialization)

リスト初期化(List-initialization)とは、ひとつの{}で囲まれた初期化子を使ってオブジェクトやリファレンスを初期化することである。このような初期化子を、初期化リスト(Initializer list)という。初期化リストの中の、コンマで区切られた式のことを、初期化リストの要素という。

リスト初期化は、直接初期化でも、コピー初期化でも使える。

```
T x( {} ) ; // 直接初期化
T x{ } ; // 直接初期化
T x = { } ; // コピー初期化
```

直接初期化のリスト初期化を、直接リスト初期化(direct-list-initialization)といい、
コピー初期化のリスト初期化を、kopied-list-initialization(copy-list-initialization)という。

初期化リストの使える場所

初期化リストは、以下の場所で使うことができる。

変数定義の初期化子

```
T x( { } ) ;  
T x{ } ;  
T x = { } ;
```

new式の初期化子

```
new T{ } ;
```

return文

```
#include <initializer_list>  
  
auto f() -> std::initializer_list<int>  
{  
    return { 1, 2, 3 } ;  
}
```

関数の実引数

```
#include <initializer_list>  
  
void f( std::initializer_list<int> ) { }  
  
int main()  
{  
    f( { 1, 2, 3 } ) ;  
}
```

5.5.1 添字

```
#include <initializer_list>

struct S
{
    void operator [] ( std::initializer_list<int> ) { }
} ;

int main()
{
    S s ;
    s[ { 1, 2, 3 } ] ;
}
```

コンストラクター呼び出しの実引数

```
#include <initializer_list>

struct S
{
    S( std::initializer_list<int> ) { }
} ;

int main()
{
    S s1( { 1, 2, 3 } ) ;
    S s2{ 1, 2, 3 } ;
    S({ 1, 2, 3 }) ; // 関数形式のキャスト
    S{ 1, 2, 3 } ; // 関数形式のキャスト
}
```

非staticデータメンバーの初期化子

```
struct S
{
    int m1[3] = { 1, 2, 3 } ;
    int m2[3]( { 1, 2, 3 } ) ;
    int m3[3]{ 1, 2, 3 } ;
} ;
```

メンバー初期化子

```
struct S
{
    int data[3] ;
    // 以下二行は同じ意味
    S() : data{ 1, 2, 3 } { }
    S() : data( { 1, 2, 3 } ) { }
};
```

代入演算子の右側

```
#include <initializer_list>

struct S
{
    S & operator = ( std::initializer_list<int> ) { }
};

int main()
{
    S s ;
    s = { 1, 2, 3 } ;
}
```

初期化リストによる初期化の詳細

初期化リストによる初期化の詳細について説明する前に、縮小変換と、初期化リストコンストラクターを説明する。

縮小変換(narrowing conversion)

縮小変換(narrowing conversion)とは、暗黙の型変換のうち、変換先の型では、変換元の値を表現できない可能性のある変換のことをいう。具体的には、四種類の変換がある。

浮動小数点数型から整数型への変換。

```
// 変換の一例
int x = double(0.0) ; // 縮小変換、doubleからint
```

浮動小数点数型の間の変換のうち、long doubleからdoubleかfloatへの変換、doubleからfloatへの変換。

```
// 縮小変換の例
long double ld = 0.01 ;
double d = ld ; // 縮小変換、long doubleからdouble
float f = ld ; // 縮小変換、long doubleからfloat
f = d ; // 縮小変換、doubleからfloat
```

整数型、もしくはunscoped enum型から、浮動小数点数型への変換。

```
// 縮小変換の例
int i = 0 ;
double d = i ; // 縮小変換、intからdouble
enum { e } ;
d = e ; // 縮小変換、unscoped enumからdouble
```

ある整数型、もしくはunscoped enum型から、別の整数型かunscoped enum型への変換において、変換先の型が、変換元の型の値を、すべて表現できない場合。

```
// short型はint型の値をすべて表現できないとする
int i = 0 ;
short s = i ; // 縮小変換
```

ただし、最初の浮動小数点数型から整数型の変換を除く、三つの変換（浮動小数点間の変換、整数から浮動小数点数への変換、整数型間の変換）には、ひとつ例外がある。もし、変換元が定数式で、その値が変換先の型で表現可能な場合、縮小変換とはみなされない。

```
const double cd = 0.0 ; // cdは定数式
float f = cd ; // 縮小変換ではない
```

```
const int ci = 0 ; // ciは定数式
double d = ci ; // 縮小変換ではない
short s = ci ; // 縮小変換ではない
```

これは、ソースコード中に定数式を書いた場合の、煩わしいエラーを防ぐための、例外的なルールである。

この場合、浮動小数点数では、変換元の定数式の値が、変換元の型では、正確に表現できなくてもよい。これは、浮動小数点数の特性に基づくものである。

初期化リストでは、縮小変換は禁止されている。

```
int main()
{
    int a = 0.0 ; // OK
    int b = { 0.0 } ; // エラー、縮小変換
    int c{ 0.0 } ; // エラー、縮小変換
    int d( { 0.0 } ) ; // エラー、縮小変換

    // OK、明示的な型変換
    int d{ static_cast<int>(0.0) } ;
}
```

初期化リストコンストラクター(initializer-list constructor)

ある型をTとして、ひとつのstd::initializer_list<T>を仮引数に取るコンストラクターか、あるいは、最初の仮引数がstd::initializer_list<T>であり、続く仮引数すべてに、デフォルト実引数が指定されている場合、そのコンストラクターを、初期化リストコンストラクター(initializer-list constructor)という。

初期化リストコンストラクターの仮引数の型は、ある型Tに対する、std::initializer_list<T>か、そのリファレンスでなければならない。

```
#include<initializer_list>

struct S
{
    // 初期化リストコンストラクター
    S( std::initializer_list<int> list ) ;
```

```
// リファレンスでもよい  
S( std::initializer_list<int> & list ) ;  
// CV修飾子付きの型に対するリファレンスでもよい  
S( std::initializer_list<int> const & list ) ;  
  
// これも初期化リストコンストラクター  
// デフォルト実引数のため  
S( std::initializer_list<int> list, int value = 0  
) ;  
  
// これらは初期化リストコンストラクターではない  
S( int value = 0, std::initializer_list<int>,  
short value ) ;  
S( std::initializer_list<int>, short value ) ;  
} ;
```

初期化リストコンストラクターは、リスト初期化の際に、他のコンストラクタより優先して考慮される。

リスト初期化の方法

リスト初期化は、以下のような優先順位で初期化される。先に書いてある条件に一致した場合、その初期化が選ばれ、その条件に対する初期化が行われる。後の条件は、先の条件に一致しなかった場合にのみ、考慮される。最後の条件にも当てはまらない場合は、エラーとなる。

T型のオブジェクト、あるいはT型へのリファレンスに対して——

初期化リストに要素がなく、Tはデフォルトコンストラクターを持つクラス型の場合

オブジェクトは値初期化される。

```
// デフォルトコンストラクターを持つクラス  
struct A { } ;  
  
int main()  
{  
    // すべて、値初期化される
```

```
A a1 = { } ;
A a2{ } ;
A a3( { } ) ;
}
```

Tがアグリゲートの場合

8.5.4 アグリゲートとして初期化される。

```
struct Aggregate
{
    int x ;
    double d ;
    char str[10] ;
} ;
// アグリゲートとして初期化
Aggregate a = { 123, 3.14, "hello" } ;
```

縮小変換が必要な場合、エラーとなる。

```
// エラー、縮小変換が必要。
int a[1] = { 1.0 } ;
```

Tがstd::initializer_list<E>の場合

initializer_listのオブジェクトが構築される。この時、initializer_listの各要素は、初期化リストの各要素によって、初期化される。縮小変換が必要な場合は、エラーになる。

```
// 空のinitializer_list
std::initializer_list<int> a = { } ;
// 要素数3のinitializer_list
std::initializer_list<int> b = { 1, 2, 3 } ;
```

```
// std::stringを要素に持つinitializer_list
std::initializer_list< std::string > c = { "hello",
"C++", "world" } ;

// エラー、縮小変換が必要
std::initializer_list<int> d = { 0.0 } ;
```

Tがクラス型の場合

適切なコンストラクターが選ばれる。縮小変換が必要な場合は、エラーとなる。まず、初期化リストコンストラクターが優先して選ばれる。

```
struct S
{
    S( std::initializer_list<int> ) { }
    S( std::initializer_list<double> ) { }
} ;

int main()
{
    // S::S( std::initializer_list<int> )
    S s1 = { 1, 2, 3 } ;
    S s2{ 1, 2, 3 } ;
    S s3( { 1, 2, 3 } ) ;

    // S::S( std::initializer_list<double> )
    S s4 = { 1.0, 2.0, 3.0 } ;
    S s5{ 1.0, 2.0, 3.0 } ;
    S s6( { 1.0, 2.0, 3.0 } ) ;
}
```

Tが初期化リストコンストラクターを持たない場合、初期化リストの要素が、実引数リストとみなされ、通常のコンストラクターが、オーバーロード解決によって選ばれる。縮小変換が必要な場合はエラーとなる。

```
struct S
{
```

```

        S( int, int ) { }
        S( int, double ) { }
    } ;

int main()
{
    // S::S( int, int )
    S s1 = { 1, 2 } ;
    // S::S( int, double )
    S s2 = { 1, 2.0 } ;

    // エラー、S::S( int, double )が選ばれるが、縮小変換
    // が必要
    S s3 = { 1.0, 2.0 } ;
}

```

リスト初期化の優先順位に注意すること。初期化リストが空の場合は、すでに挙げた一番最初のリスト初期化の条件が選ばれ、値初期化されるので、ここでの条件による初期化が行われることはない。初期化リストコンストラクターは、通常のコンストラクタより、常に優先される。

```

struct S
{
    S( std::initializer_list<int> ) { }
    S( double ) { }
} ;

int main()
{
    // OK、値初期化される
    S s1 = { } ;
    // エラー、S( std::initializer_list<int> )が選ばれ
    // る
    // しかし、縮小変換が必要
    S s2 = { 0.0 } ;
}

```

この例では、s1はリスト初期化の最初の条件である、空の初期化リストを満たすので、値初期化される。s2では、初期化リストコンストラクターが優先される。S::S(double)が考慮されることはない。しかし、この場合、縮小変換が必要なので、エラーとなる。

Tがリファレンス型の場合

正確には、Tがクラス型へのリファレンスか、リファレンス型で、初期化リストが空の場合。

```
// 条件に一致する例
class C { } ;

C && r1 = { } ; // OK、Tがクラス型へのリファレンス
int && r2 = { } ; // OK、リファレンス型で初期化子が空

// これは条件に一致しない。後の条件を参照
int && r3 = { 0 } ;
```

リファレンスされている型に対するprvalueの一時オブジェクトが生成され、初期化リストでリスト初期化される。リファレンスは、その一時オブジェクトを参照する。

```
struct A
{
    int x ;
} ;

struct B
{
    B( std::initializer_list<int> ) { }
} ;

int main()
{
    A && r1 = { } ;
    A && r2 = { 1 } ;
    B && r3 = { 1, 2, 3 } ;
    int && ref = { } ;
}
```

prvalueの一時オブジェクトが生成されることに注意しなければならない。
prvalueを参照できるリファレンスは、rvalueリファレンスか、constかつ非volatileなlvalueリファレンスだけである。

```
// 以下はOK
int && r1 = { } ; // OK、rvalueリファレンス
int const & r2 = { } ; // OK、constかつ非volatileな
lvalueリファレンス

// 以下はエラー
int & r3 = { } ; // エラー、非constなlvalueリファレンス
int const volatile & r4 = { } ; // エラー、constではあるが、volatileもある
```

初期化リストの要素がひとつの場合

オブジェクトは、初期化リストの要素で初期化される。縮小変換が必要な場合はエラーとなる。

```
int a{ 0 } ;
int b( { 0 } ) ;
int c = { 0 } ;

// 通常のリファレンスの初期化と同じ
// リファレンスの初期化は、8.5.6 リファレンスを参照。
int && ref = { 0 } ;

// エラー、縮小変換が必要
int d{ 0.0 } ;
```

初期化リストに要素がない場合

オブジェクトは値初期化される。この条件に当てはまるのは、ポインターがある。クラスは、すでに先の条件に一致しているので、この条件には当てはまらない。

```
// pは値初期化される。つまり、nullポインターとなる。
int * p{ } ;
```

std::initializer_listの実装

std::initializer_listがどのように実装されるかは、規格では具体的に規定されていない。ただし、いくつかの保証はある。

std::initializer_list<E>型のオブジェクトは、{}で囲まれた初期化リストによって生成される。このとき、Eの配列が生成され、初期化リストによって初期化される。たとえば、以下のようなコードがあるとき、

```
std::initializer_list<int> list = { 1, 2, 3 } ;
```

以下のように、ユーザー側からは見えない配列が生成される。

```
int __array[3] = { 1, 2, 3 } ;
// 実装依存のstd::initializer_list<int>の初期化の実装例
// 配列の先頭要素へのポインターと、最後からひとつ後ろのポ
// インターを格納する
std::initializer_list<int> list( array, array + 3 ) ;
```

実際には、std::initializer_listには、このようなコンストラクターはない。あくまで参考のための、実装の一例である。

初期化リストにより生成される配列の寿命は、std::initializer_listのオブジェクトの寿命と同じである。

配列は、staticストレージか自動ストレージ上に構築される。動的ストレージの確保が起こることはない。というのも、動的にストレージを確保しなければならない技術的な理由はないからだ。

9 クラス (Classes)

クラスとは、ユーザーが定義できる型である。クラスは、クラス指定子 (class-specifier) によって定義する。クラス指定子は指定子なので、宣言文の中で使うことができる。したがって、クラスの定義とは、宣言文である。

クラス指定子:

```
class-key 識別子opt finalopt 基本クラス指定opt { メンバー指定
opt }

class-key:
  class
  struct
  union
```

```
class A { } ;
struct B { } ;
union C { } ;
```

// クラスの定義は宣言文なので、変数やポインターなどを同時に宣言できる。

```
class C { } obj, *ptr ;
```

クラスの定義は、宣言文である。したがって、終端には必ず、セミコロンを書かなければならない。

```
struct A { } ; // 宣言文の終端にはセミコロンが必要
struct B { } // エラー、セミコロンがない
```

クラス指定子は、識別子をクラス名として、宣言されたスコープに導入する。クラス指定子の識別子は省略することができる。その場合、無名クラスの定義となる。

```
// 無名クラス
class { } a ;
```

クラス指定子のキーワードには、class、struct、unionがある。このうち、classとstructは、デフォルトのアクセス指定の違い以外は、全く同じである。詳しくは、11 メンバーのアクセス指定を参照。unionキーワードによって定義されたクラスは、unionとなる。詳しくは、9.9 unionを参照。

クラス名は、現れた場所の直後から、スコープに導入される。

```
// クラス名は直後から使える
struct A { } * ptr = static_cast<A *>( nullptr ) ;
```

また、クラス名は、そのクラスのスコープ内にも導入される。

```
struct A
{
    // クラス名はクラスのスコープ内にも導入される
    typedef A type ;
};
```

クラスは、'}' 記号が現れた場所以降、完全に定義されたものとみなされる。たとえ、メンバー関数が定義されていなくても、クラス自体は、その場所で定義されている。

```
class A ; // クラスAは不完全型
class A { void f(void) ; } ; // クラスAは定義された
```

この例では、A::f(void)は定義されていないが、クラスAは、}が現れた場所以降、すでに定義されている。

クラスの宣言で、クラス名の後、基本クラスの前にfinalを記述できる。

```
struct base_class { } ;
struct class_name final : base_class
{ } ;
```

finalが指定されたクラスが基本クラス指定子に現れた場合、エラーとなる。つまり、finalが指定されたクラスから派生することはできない。

```
class non_final_class { } ;
// OK、finalの指定されていないクラスは基本クラスに指定できる
class ok : non_final_class { } ;

class final_class final { } ;
// エラー、finalの指定されているクラスは基本クラスに指定できない
class error : final_class { } ;
```

完全型のクラスとそのメンバーのオブジェクトは、ゼロではないサイズを持つ。つまり、sizeof演算子は、少なくとも、1以上を返す。

```
struct Subobject { } ;
```

```

struct Object
{
    Subobject sub ;
} ;

int main()
{
    Object obj ;
    sizeof( obj ) ; // 値は実装依存だが、少なくとも1以上
    sizeof( obj.sub ) ; // 同上
}

```

ただし、基本クラスのサブオブジェクトは、内部的にはサイズを持たないかもしれない。これは最適化のために許されている。

```

struct Base { } ;
struct Derived : Base { } ;

```

この場合、`sizeof(Base)`と`sizeof(Derived)`は、同じ値を返すかもしれない。

9.1 トリビアルにコピー可能なクラス (trivially copyable class)

トリビアルにコピー可能なクラス (trivially copyable class) とは、クラスのオブジェクトを構成するバイト列の値を、そのままコピーできるクラスのことである。詳しくは、3.9 [型](#) を参照。

あるクラスが trivially copyable class となるには、以下の条件をすべて満たさなければならない。

非trivialな、コピー構造子、ムーブ構造子、コピー代入演算子、ムーブ代入演算子を持たないこと。trivialなデストラクターを持っていること。

つまり、これらの特別なメンバー関数を、ユーザー定義してはならない。また、`virtual` 関数や `virtual` 基本クラスも持つことはできない。`trivial` の詳しい定義については、12.8 [クラスオブジェクトのコピーとムーブ](#) を参照。

ここで、「持たない」ということは、`delete` 定義によって削除しても構わないということである。

```

struct A
{
    A( A const & ) = delete ;
    A( A && ) = delete ;
    A & operator = ( A const & ) = delete ;
}

```

```

A & operator = ( A && ) = delete ;

// trivialなデストラクターは、「持って」いなければならない

int x ;
} ;

```

また、アクセス指定子が異なるメンバーを持っていても構わない。

上記のクラスAは、コピーもムーブもできないクラスであるが、trivially copyable classである。

9.2 トリビアルクラス (trivial class)

トリビアルクラス (trivial class) とは、trivially copyable classの条件に加えて、非トリビアルデフォルトコンストラクターを持たないクラスである。

9.3 標準レイアウトクラス (standard-layout class)

標準レイアウトクラス (standard-layout class) の詳しい説明については、9.6 クラスのメンバーを参照。

あるクラスが標準レイアウトクラスとなるためには、以下の条件をすべて満たさなければならない。

標準レイアウトクラスではない非staticメンバーをもたない。また、そのようなクラスへの配列やリファレンスも持たない。

virtual関数とvirtual基本クラスを持たない。

非staticデータメンバーは、すべて同じアクセス指定子である。

```

// 標準レイアウトクラス
struct S
{
    // すべて同じアクセス指定子
    int x ;
    int y ;
    int z ;

    // staticデータメンバーは、別のアクセス指定子でもよい

```

```
private :
    static int data ;
} ;
int S::data ;
```

標準レイアウトクラスではないクラスを基本クラスに持たない。

基本クラスに非staticデータメンバーがある場合は、クラスは非staticデータメンバーを持たない。クラスが非staticデータメンバーを持つ場合は、基本クラスは非staticデータメンバーを持たない。つまり、クラスとその基本クラス(複数可)の集合の中で、どれかひとつのクラスだけが、非staticなデータメンバーを持つことが許される。

```
struct Empty { } ;
struct Non_empty { int member ; } ;

// 以下は標準レイアウトクラス
struct A
// 基本クラスに非staticデータメンバーがある場合
    : Non_empty
{
// 非staticデータメンバーを持たない
} ;

struct B
// 基本クラスは非staticデータメンバーを持たない
    : Empty
{
// 非staticデータメンバーを持つ場合
    int data ;
} ;

// 以下は標準レイアウトクラスではない
// 基本クラスもクラスCも非staticデータメンバーを持っている
struct C
    : Non_empty
{
    int data ;
} ;
```

最初の非staticデータメンバーと、基本クラスとで、同じ型を使わない。

```
// 標準レイアウトクラスではない例
```

```

struct A { } ;
struct B
// 基本クラス
: A
{
    A a ; // 最初の非staticデータメンバー
    int data ;
} ;

// 最初の非staticデータメンバーでなければよい
struct C : A
{
    int data ;
    A a ;
} ;

```

この制限は、基本クラスとデータメンバーとの間で、アドレスが重複するのを防ぐためである。

```

struct A { } ;
struct B : A { A a ; } ;

B obj ;

A * p1 = &obj ; // 基本クラスのサブオブジェクトへのアドレス
A * p2 = &obj.a ; // データメンバーへのアドレス

// p1 != p2が保証される

```

このような場合、もし、クラスBが標準レイアウトクラスであれば、基本クラスのサブオブジェクトへのアドレスと、データメンバーのサブオブジェクトへのアドレスが重複してしまう。つまり、p1とp2が同じ値になってしまふ。異なるアドレスを得られるためには、このようなクラスを標準レイアウトクラスにすることはできない。

標準レイアウトクラスのうち、**struct**と**class**のキーワードで定義されるクラスを、特に標準レイアウト**struct**と言う。**union**キーワードで定義されるクラスを、特に標準レイアウト**union**という。

9.4 POD構造体(POD struct)

PODとは、Plain Old Dataの略である。これは、C言語の構造体に相当するクラスであ

る。C++11では、クラスがPODの条件を満たした際に保証される動作を、*trivially copyable class*と、標準レイアウトクラスの二つの動作に細分化した。そのため、C++11では、特にPODにこだわる必要はない。

クラスがPODとなるためには、*trivial class*と標準レイアウトクラスの条件を満たし、さらに、PODではないクラスを非staticデータメンバーに持たないという条件が必要になる。

9.5 クラス名 (Class names)

クラス宣言は、クラス名をスコープに導入する。クラス名は外側のスコープの名前を隠す。

もし、クラス名が宣言されたスコープ内で、同名の変数、関数、enumが宣言された場合は、どちらの宣言もスコープに存在することになる。その場合、クラス名を使うには、7.3.6.3 複雑型指定子を使う以外に方法がなくなる。

```
void name() { }
class name { } ;

name n ; // エラー、nameは関数名
class name n ; // OK、複雑型指定子
```

9.6 クラスのメンバー (Class members)

クラスのメンバー指定には、宣言文、関数の定義、using宣言、static_assert宣言、テンプレート宣言、エイリアス宣言を書くことができる。

```
struct Base { int value ; } ;

struct S : Base
{
    int data_member ; // 宣言文
    void member_function() { } // 関数の定義
    using Base::value ; // using宣言
    static_assert( true, "this must not be an error." ) ; // static_assert宣言
    template < typename T > struct Inner { } ; // テンプレート宣言
    using type = int ; // エイリアス宣言
```

```
}
```

このうち、クラスのメンバーとなるのは、データメンバーとメンバー関数、ネストされた型名、列挙子である。

```
struct S
{
    int x ; // データメンバー
    void f() { } ; // メンバー関数
    using type = int ; // ネストされた型名
    enum { id } ; // 列挙子
};
```

データメンバーは、俗にメンバー変数とも呼ばれている。クラス定義内で、変数の宣言文を書くと、データメンバーとなる。

クラスのメンバーを、クラスの定義内で複数回宣言することはできない。ただし、クラス内のクラスとenumに関しては、前方宣言することができる。

```
class Outer
{
    void f() ;
    void f() ; // エラー、複数回の宣言

    class Inner ; // クラス内クラスの宣言
    class Inner { } ; // OK、クラス内クラスの定義

    enum struct E : int ; // クラス内enumの宣言
    enum struct E : int { id } ; // OK、クラス内enumの定義
};
```

クラスは、}で閉じた所をもって、完全に定義されたとみなされる。たとえ、メンバー関数が定義されていなくても、クラス自体は完全に定義された型となる。

メンバーはコンストラクターで初期化できる。詳しくは12.1 コンストラクターを参照。メンバーは初期化子で初期化できる。詳しくは、9.8.2 staticデータメンバーと、12.6.2 基本クラスとデータメンバーの初期化を参照。

```
struct S
{
    S() : x(0) { } // コンストラクター
```

```

int x = 0 ; // 初期化子
static int data ;
} ;

int S::data = 0 ; // 初期化子

```

メンバーは、`extern`や`register`指定子と共に宣言することはできない。メンバーを`thread_local`指定子と共に宣言する場合は、`static`指定子も指定しなければならない。

```

struct S
{
    extern int a ; // エラー
    register int b ; // エラー
    thread_local int c ; // エラー

    // OK、staticと共に宣言している
    static thread_local int d ;
} ;

thread_local int S::d ; // 定義

```

基本的に、クラス名と同じ名前のメンバーを持つことはできない。これには、一部の例外が存在するが、本書では解説しない。

```

struct name
{
    static int name ;          // エラー
    void name() ;             // エラー
    static void name() ;       // エラー
    using name = int ;        // エラー
    enum { name } ;           // エラー
    union { int name } ;      // エラー
} ;

```

`union`ではないクラスにおいて、同じアクセス指定下にある非`static`データメンバーは、クラスのオブジェクト上で、宣言された順番に確保される。つまり、先に宣言されたデータメンバーは、後に宣言されたデータメンバーよりも、上位のアドレスに配置される。ただし、実装は必要なパディングを差し挟むかもしれないで、後のデータメンバーが、先のデータメンバーの直後に配置されるという保証はない。

```
struct S
```

```

{
public :
// 同じアクセス指定下にある非staticデータメンバー
    int x ;
    int y ;
} ;

int main()
{
    S s ;
    int * p1 = &s.x ;
    int * p2 = &s.y ;

    // この式は、必ずtrueとなる
    p1 < p2 ;
    // この式がtrueとなる保証はない
    p1 + 1 == p2 ;
}

```

アクセス指定子が異なる非staticデータメンバーの配置に関しては、未規定である。

標準レイアウトstructのオブジェクトへのポインターは、`reinterpret_cast`によって変換された場合、クラスの最初のメンバーへのポインターに変換できる。また、その逆も可能である。

```

struct S { int x ; } ;

int main()
{
    S s ;
    S * p1 = &s ;
    int * p2 = &s.x ;

    // 以下の2式は、trueとなることが保証されている
    p1 == reinterpret_cast< S * >( p2 ) ;
    p2 == reinterpret_cast< int * >( p1 ) ;
}

```

9.6.1 レイアウト互換(layout-compatible)

レイアウト互換(layout-compatible)という概念がある。まず、同じ型は、レイアウト互換である。

もし、二つの標準レイアウトstructが、同じ数の非staticデータメンバーを持ち、対応する非staticデータメンバーが、それぞれレイアウト互換であったならば、そのクラスは、お互いにレイアウト互換structである。

もし、二つの標準レイアウトunionが、同じ数の非staticデータメンバーを持ち、対応する非staticデータメンバーが、それぞれレイアウト互換であったならば、そのクラスは、お互いにレイアウト互換unionである。

二つの標準レイアウトstructは、レイアウト互換structのメンバーが続く限り、オブジェクト上で共通の表現をしていると保証される。

```
// A、Bは、2番目のメンバーまで、お互いにレイアウト互換
struct A
{
    int x ;
    int y ;
    float z ;
} ;

struct B
{
    int x ;
    int y ;
    double z ;
} ;

int main()
{
    A a ;

    B * ptr = reinterpret_cast< B * >( &a ) ;
    // OK、aのオブジェクトの、対応するレイアウト互換なメンバーが変
更される
    ptr->x = 1 ;
    ptr->y = 2 ;

    // エラー、3番目のメンバーは、レイアウト互換ではない
    ptr->z = 0.0 ;
}
```

ただし、メンバーがビットフィールドの場合、お互いに同じビット数でなければならない。

```
// AとBはお互いにレイアウト互換
```

```
struct A
{
    int x:8 ;
} ;
struct B
{
    int x:8 ;
} ;
```

標準レイアウトunionが、お互いにレイアウト互換な複数の標準レイアウトstructを持つとき、先頭から共通のメンバーについては、一方を変更して、他方で使うこともできる。

```
struct A
{
    int x ;
    int y ;
    float z ;
} ;

struct B
{
    int x ;
    int y ;
    double z ;
} ;

union U
{
    A a ;
    B b ;
} ;

int main()
{
    U u ;
    u.a.x = 1 ;
    u.a.y = 2 ;

    // 以下の2式はtrueとなることが保証されている
    u.b.x == 1 ;
    u.b.y == 2 ;

    // 3番目のメンバーは、レイアウト互換ではない
```

}

9.7 メンバー関数(Member functions)

クラスの定義内で宣言される関数を、クラスのメンバー関数(member function)という。メンバー関数はstatic指定子と共に宣言することができる。その場合、関数はstaticメンバー関数となる。staticメンバー関数ではないメンバー関数のことを、非staticメンバー関数という。ただし、friend指定子と共に宣言された関数は、メンバー関数ではない。

```
struct S
{
    void non_static_member_function() ; // 非staticメンバー関
数
    static void static_member_function() ; // staticメンバー関
数

    friend void friend_function() ; // これはメンバー関数ではな
い
} ;
```

メンバー関数は、クラス定義の内側でも外側でも定義することができる。クラス定義の内側で定義されたメンバー関数を、inlineメンバー関数という。これは、暗黙的にinline関数となる。クラス定義の外側でメンバー関数を定義する場合、クラスと同じ名前空間スコープ内で定義しなければならない。

```
struct S
{
    // inlineメンバー関数
    void inline_member_function()
    { /*定義*/ }

    void member_function() ; // メンバー関数の宣言
} ;

// クラスSと同じ名前空間スコープ
void S::member_function()
{ /*定義*/ }
```

クラス定義の外側でinlineメンバー関数の定義をすることもできる。それには、関数宣言

か関数定義で、`inline`指定子を使えばよい。

```
struct S
{
    inline void f() ;
    void g() ;
} ;

void S::f(){}
inline void S::g() { }
```

クラス定義の外側でメンバー関数を定義するためには、メンバー関数の名前を、`::`演算子によって、クラス名で修飾しなければならない。

```
struct S
{
    void member() ;
} ;

void S::member(){ }
```

9.12 ローカルクラスでは、メンバー関数をクラス定義の外側で宣言する方法はない。

9.7.1 非staticメンバー関数(Nonstatic member functions)

非staticメンバー関数は、そのメンバー関数が属するクラスや、そのクラスから派生されたクラスのオブジェクトに対して、5.5.5 クラスメンバーアクセス演算子を使うことで、呼び出すことができる。同じクラスや、そのクラスから派生されたクラスのメンバー関数からは、他のメンバー関数を、通常の関数の呼び出しと同じように呼ぶことができる。

```
struct Object
{
    void f(){}
    void g()
    {
        // 関数呼び出し
        f();
    }
};
```

```
int main()
{
    Object object ;
    // クラスのメンバーアクセス演算子と、関数呼び出し
    object.f() ;
}
```

非staticメンバー関数は、CV修飾子と共に宣言することができる。

```
struct S
{
    void none() ;
    void c() const ; // constメンバーエンターフィー
    void v() volatile ; // volatileメンバーエンターフィー
    void cv() const volatile ; // const volatileメンバーエンターフィー
} ;
```

このCV修飾子は、thisポインターの型を変える。また、メンバー関数の型も、CV修飾子に影響される。

非staticメンバー関数は、リファレンス修飾子と共に宣言することができる。リファレンス修飾子は、オーバーロード解決の際の、暗黙の仮引数の型に影響を与える。詳しくは、13.3.1 [候補関数と実引数リスト](#)を参照。

非staticメンバー関数は、virtualやピュアvirtualの文法で宣言することができる。詳しくは、10.3 [virtual関数](#)や、10.4 [抽象クラス](#)を参照。

9.7.2 thisポインター(The this pointer)

非staticメンバー関数内では、thisというキーワードが、クラスのオブジェクトへのprvalueのポインターを表す。このクラスのオブジェクトは、メンバー関数を呼び出した際のクラスのオブジェクトである。

```
struct Object
{
    void f()
    {
        this ;
    }
} ;
```

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

    a.f() ; // thisは&a
    b.f() ; // thisは&b
    c.f() ; // thisは&c
}
```

class Xのメンバー関数におけるthisの型は、X *である。もし、constメンバー関数の場合、X const *となり、volatileメンバー関数の場合、X volatile *となり、const volatileメンバー関数の場合は、X const volatile *となる。

```
class X
{
    // thisの型はX *
    void f() { this ; }
    // thisの型はX const *
    void f() const { this ; }
    // thisの型はX volatile *
    void f() volatile { this ; }
    // thisの型はX const volatile *
    void f() const volatile { this ; }
};
```

constメンバー関数では、メンバー関数に渡されるクラスのオブジェクトがconstであるため、thisの型も、constなクラスへのポインター型となり、非staticデータメンバを変更することができない。

```
class X
{
    int value ;

    void f() const
    {
        this->value = 0 ; // エラー
    }
};
```

これは、thisの型を考えてみると分かりやすい。

```
class X
{
    int value ;

    void f() const
    {
        X const * ptr = this ;
        ptr->value = 0 ; // エラー
    }
};
```

thisの型が、constなクラスに対するポインターなので、データメンバを変更することはできない。

同様にして、volatileの場合も、非staticデータメンバーはvolatile指定されたものとみなされる。volatileの具体的な機能については、実装に依存する。

CV修飾されたメンバ関数は、同等か、より少なくCV修飾されたクラスのオブジェクトに対して呼び出すことができる。

```
struct X
{
    X() { }
    void f() const { } ;
};

int main()
{
    X none ;
    none.f() ; // OK、より少なくCV修飾されている
    X const c ;
    c.f() ; // OK、同じCV修飾

    X const volatile cv ;
    cv.f() ; // エラー、CV修飾子を取り除くことはできない
}
```

これは、非staticメンバ関数から、他の非staticメンバ関数を、クラスメンバーアクセスなしで呼び出す際にも当てはまる。その場合、クラスのオブジェクトとは、thisである。

```

struct X
{
    void c() const // constメンバー関数
    {
        // thisの型はX const *
        nc(); // エラー、CV修飾子を取り除くことはできない
    } ;

    void nc() // 非constメンバー関数
    {
        // thisの型はX *
        c(); // OK、CV修飾子を付け加えることはできる
    }
} ;

```

コンストラクターとデストラクターは、CV修飾子と共に宣言することができない。ただし、これらの特別なメンバー関数は、constなクラスのオブジェクトの生成、破棄の際にも、呼び出される。

9.8 staticメンバー(Static members)

クラスのデータメンバーやメンバー関数は、クラス定義内で、static指定子と共に宣言することが出来る。そのように宣言されたメンバーを、クラスのstaticメンバーという。

```

struct S
{
    static int data_member ;
    static void member_function() ;
} ;
int S::data_member ;

```

クラスXのstaticメンバーは、::演算子を用いて、X::sのように参照することで、使うことができる。staticメンバーは、クラスのオブジェクトがなくても参照できるので、クラスのメンバーアクセス演算子を使う必要はない。しかし、クラスのメンバーアクセス演算子を使っても参照できる。

```

struct X
{
    static int s ;
}

```

```

} ;

int X::s ;

int main()
{
    // ::演算子による参照
    X::s ;

    // クラスのメンバーアクセス演算子でも参照することはできる
    X object ;
    object.s ;
}

```

クラスのメンバー関数内では、非修飾名を使った場合、クラスのstaticメンバー、enum名、ネストされた型が名前探索される。

```

struct X
{
    void f()
    {
        s ; // X::s
        value ; // X::value
        type obj ; // X::type
    }

    static int s ;
    enum { value } ;
    typedef int type ;
} ;
int X::s ;

```

staticメンバーにも、通常通りのアクセス指定が適用される。

9.8.1 staticメンバー関数 (Static member functions)

staticメンバー関数は、thisポインターを持たない。virtual指定子を使えない。CV修飾子を使えない。名前と仮引数の同じ、staticメンバー関数と非staticメンバー関数は、両立できない。

```
struct S
```

```
{
    // エラー、同じ名前と仮引数のstatic関数と非static関数が存在する
    void same() ;
    static void same() ;
};
```

その他は、通常のメンバー関数と同じである。

9.8.2 staticデータメンバー(Static data members)

staticデータメンバーは、クラスのサブオブジェクトには含まれない。staticデータメンバーは、クラスのスコープでアクセス可能なstatic変数だと考えてもよい。クラスのすべてのオブジェクトは、ひとつのstaticデータメンバーのオブジェクトを共有する。ただし、static thread_localなデータメンバーのオブジェクトは、スレッドにつきひとつ存在する。

```
struct S
{
    static int member ;
} ;
int S::member ;

int main()
{
    S a, b, c, d, e ;
    // すべて同じオブジェクトを参照する
    a.member ; b.member ; c.member ; e.member ;
}
```

クラス定義内のstaticデータメンバー宣言は、定義ではない。staticデータメンバーの定義は、クラスの定義を含む名前空間スコープの中に書かなければならない。その際、名前に::演算子を用いて、クラス名を指定する必要がある。staticデータメンバーの定義には、初期化子を使うことができる。初期化子は必須ではない。

```
struct S
{
    static int member ;
};
```

```
// クラス名 :: メンバー名
int S::member = 0 ;
```

staticデータメンバーが、constなリテラル型の場合、クラス定義内の宣言に、初期化子を書くことができる。この場合、初期化子は定数式でなければならない。staticデータメンバー自体も、定数式になる。初期化子を書かなければ、通常通り、クラス定義の外、同じ名前空間スコープ内で、定義を書かなければならない。この場合は、定数式にはならない。

```
struct S
{
    static const int constant_expression = 123 ; // 定数式
    static const int non_constant_expression ; // 宣言、定数式
    // ではない
} ;
const int S::non_constant_expression = 123 ; // 定義
```

リテラル型のstaticデータメンバーは、constexpr指定子をつけて宣言することもできる。この場合、初期化子を書かなければならない。初期化子は、定数式でなければならない。このように定義されたstaticデータメンバーは、定数式になる。

```
struct S
{
    static constexpr int constant_expression = 123 ; // 定数
    // 式
} ;
```

名前空間スコープのクラスのstaticデータメンバーは、外部リンクエージを持つ。ローカルクラスのstaticデータメンバーは、リンクエージを持たない。

staticデータメンバーは、非ローカル変数と同じように、初期化、破棄される。詳しくは、3.6.2 非ローカル変数の初期化、3.6.3 終了を参照。

staticデータメンバーには、mutable指定子は使えない。

9.9 union(Unions)

unionというクラスは、クラスキーにunionキーワードを用いて宣言する。unionでは、非staticデータメンバーは、どれかひとつのみが有効である。これは、unionのオブジェクト内では、非staticデータメンバーのストレージは、共有されているからである。unionのサイズは、非staticデータメンバーのうち、最も大きな型を格納するのに十分なサイズとな

る。

```
union U
{
    int i ;
    short s ;
    double d ;
} ;

int main()
{
    U u ;
    u.i = 0 ; // U::iが有効
    u.s = 0 ; // U::sが有効、U::iは有効ではなくなる

}
```

この例では、unionのサイズは、int, short, doubleのうちの、最もサイズが大きな型を格納するのに十分なだけのサイズである。データメンバーであるi, s, dは、同じストレージを共有している。

unionと、標準レイアウトクラスについては、9 [クラス](#)を参照。

unionは、通常のクラスに比べて、いくらか制限を受ける。unionはvirtual関数を持つことができない。unionは、基本クラスを持つことができない。unionは基本クラスになれない。unionは、リファレンス型の非staticデータメンバーを持つことができない。unionの非staticデータメンバーのうち、初期化子を持てるのは、ひとつだけである。

// エラーの例

```
// エラー、virtual関数を持ってない
union U1 { virtual void f() { } } ;
// エラー、基本クラスを持ってない
struct Base { } ;
union U : Base { } ;
// エラー、基本クラスになれない
union Union_base { } ;
union Derived : Union_base { } ;
// エラー、リファレンス型の非staticデータメンバーを持ってない
union U2 { int & ref ; } ;
// エラー、非staticデータメンバーで、初期化子持てるのは、ひとつだけ
union U3
```

```
{
    int x = 0 ;
    int y = 0 ; // エラー、複数の初期化子、どちらか一つならエラー
    // ではない
};
```

その他は、通常のクラスと変わることがない。unionはメンバー関数を持つ。メンバー関数には、コンストラクタやデストラクター、演算子のオーバーロードも含まれる。アクセス指定も使える。staticデータメンバーならば、リファレンス型でも構わない。ネストされた型も使える。

unionの非staticデータメンバーが、非trivialなコンストラクター、コピーコンストラクター、ムーブコンストラクター、コピー代入演算子、ムーブ代入演算子、デストラクターを持っている場合、unionの対応するメンバーが、暗黙的にdeleteされる。そのため、これらのメンバーを使う場合には、union側で、ユーザー定義しなければならない。

```
union U
{
    std::string str ;
    std::vector<int> vec ;
};

int main()
{
    U u ; // エラー、コンストラクターとデストラクターがdelete定義
    // されている。
}
```

この例では、strやvecは、非trivialなコンストラクタやデストラクタなどを持っているので、union側でも、それらを定義しなければならない。また、この例の場合、unionのコンストラクタやデストラクターは何もしないので、このunionを実際に使う場合には、placement newや、明示的なデストラクター呼び出しが必要になる。

```
union U
{
    std::string str ;
    std::vector<int> vec ;

    U() { }
    ~U() { }
};

int main()
```

```
{
    U u ;

    new ( &u.str ) std::string( "hello" ) ;
    u.str.~basic_string() ;

    new ( &u.vec ) std::vector<int> { 1, 2, 3 } ;
    u.vec.~vector() ;
}
```

もちろん、unionのコンストラクタやデストラクターで、どれかのデータメンバーの初期化、破棄をすることは可能である。しかし、どのデータメンバーが有効なのかということを、union内で把握するのは難しい。

9.9.1 無名union(anonymous union)

以下のような形式のunionの宣言を、無名union(anonymous union)という。

```
union { メンバー指定opt } ;
```

無名unionは、無名の型のunionの、無名のオブジェクトを生成する。無名unionのメンバー指定は、非staticデータメンバーだけでなければならない。無名unionのメンバーの名前は、宣言されているスコープの他の名前と衝突してはならない。無名unionでは、staticデータメンバーやメンバー関数、ネストされた型などは使えない。また、privateやprotectedアクセス指定も使えない。

```
int main()
{
    union { int i ; short s ; } ;
    // iかsのどちらかひとつだけが有効
    i = 0 ;
    s = 0 ;
}
```

これは、以下のようないいコードと同じであると考えることもできる。

```
int main()
{
    union Anonymous { int i ; short s ; } unnamed ;
    unnamed.i = 0 ;
```

```
        unnamed.s = 0 ;
}
```

名前空間スコープで宣言される無名unionには、必ずstatic指定子をつけなければならぬ。

```
// グローバル名前空間スコープ
static union { int x ; int y ; } ;
```

名前空間スコープで宣言される無名unionは、staticストレージの有効期間と、内部リンクを持つ。

ブロックスコープで宣言される無名unionは、ブロックスコープ内で許されているすべてのストレージ上に構築できる。

```
int main()
{
    // 自動ストレージ
    union { int a } ;
    // staticストレージ
    static union { int b } ;
    // thread_localストレージ
    thread_local union { int c } ;
}
```

クラススコープで宣言される無名unionには、ストレージ指定子を付けることはできない。

```
struct S
{
    union
    {
        int x ;
    } ;
};
```

オブジェクトやポインターを宣言している、クラス名の省略されたunionは、無名unionではない。

```
// クラス名の省略されたunion
// 無名unionではない
union { int x ; } obj, * ptr ;
```

9.9.2 共用メンバー(variant member)

union、もしくは無名unionを直接のメンバーを持つクラスを、unionのようなクラス(union-like class)という。unionのようなクラスには、共用メンバー(variant member)という概念が存在する。unionの共用メンバーは、unionの非staticデータメンバーである。無名unionを直接のメンバーを持つクラスの場合、無名unionの非staticデータメンバーである。

```
// xとyは共用メンバー
union U { int x ; int y ; }

// xとyは共用メンバー
struct S
{
    union { int x ; int y ; } ;
};
```

9.10 ビットフィールド(Bit-fields)

ビットフィールドは、以下のようなメンバー宣言子の文法で宣言できる。

識別子_{opt} : 定数式

定数式は、0よりも大きい整数でなければならない。

```
struct S
{
    int // 型指定子
    x : 8 ; // ビットフィールドの宣言子
};
```

ビットフィールドの定数式は、データメンバーのサイズをビット数で指定する。ビットフィールドに関しては、ほとんどの挙動が実装依存である。特に、ビットフィールドがクラスオブジェクトのストレージ上でどのように表現されるのかということや、アライメントなどは、すべて実装依存である。また、実装は、ビットフィールドのメンバー同士を詰めて表現することが許されている。

```
struct S
{
    char x : 1 ;
    char y : 1 ;
};
```

ここで、`sizeof(S)`は、2以上になるとは限らない。例えば、`sizeof(S)`が1を返す実装もあり得る。

ビットフィールドの定数式は、オブジェクトのビット数を上回ることができる。その場合、上回ったビット数は、パディングとして確保されるが、オブジェクトの内部表現として使われることはない。

```
struct S
{
    int x : 1000 ;
};
```

ここで、`sizeof(S)`は、少なくとも1000ビット以上になる値を返す（規格では、1バイトあたりのビット数は定められていない）。ただし、`X::s`は、本来のint型以上の範囲の値を保持することはできない。`int型`のオブジェクトのビット数を上回った分は、単にパディングとして確保されているに過ぎない。

ビットフィールドの宣言子で、識別子を省略した場合、無名ビットフィールドとなる。無名ビットフィールドはクラスのメンバーではなく、初期化もされない。ただし、実装依存の方法で、クラスのオブジェクト内に存在する。一般的な実装では、無名ビットフィールドは、オブジェクトのレイアウトを調整するためのパディングとして用いられる。

```
struct S
{
    int x : 4 ;
    char : 3 ; // 無名ビットフィールド
    int y : 1 ;
};
```

あるコンパイラーでは、このような無名ビットフィールドにより、`S::x`と`S::y`の間に、3ビット

分のパディングを挿入することができる。ただし、すでに述べたように、ビットフィールドの内部表現とアライメントは実装依存なので、これはすべてのコンパイラーに当てはまるわけではない。使用しているコンパイラーが、ビットフィールドをどのように実装しているかは、コンパイラー独自のマニュアルを参照すべきである。

無名ビットフィールドでは、特別に、定数式に0を指定することができる。

```
struct S
{
    int x : 4 ;
    char : 0 ; // 無名ビットフィールド
    int y : 4 ;
};
```

これは、無名ビットフィールドの次のビットフィールドのアライメントを、アロケーション単位の境界に配置させるための指定である。上記の構造体は、ある環境では、S::xとS::yが同一のアロケーション単位に配置されるかもしれないが、無名ビットフィールドを使うことで、X::yを別のアロケーション単位に配置できる。

ビットフィールドはstaticメンバーにはできない。ビットフィールドの型は、整数型かenum型でなければならない。符号が指定されていない整数型のビットフィールドの符号は実装依存である。

```
struct S
{
    static int error : 8 ; // エラー、ビットフィールドはstatic
メンバーにはできない
    int impl : 8 ; // 符号は実装依存
    signed s : 8 ; // 符号はsigned
    unsigned u : 8 ; // 符号はunsigned
};
```

bool型のビットフィールドは、ビット数に関わらず、bool型の値を表現できる。

```
struct S
{
    bool a : 1 ;
    bool b : 2 ;
    bool c : 3 ;
};

int main()
```

```
{
    S s ;
    s.a = true ;
    s.b = false ;
    s.c = true ;
}
```

ビットフィールドのアドレスを得ることはできない。つまり、&演算子をビットフィールドに適用することはできない。

```
struct S
{
    int x : 8 ;
} ;

int main()
{
    S s ;
    &s.x ; // エラー
}
```

リファレンスは、ビットフィールドを参照することはできない。ただし、`const`な`lvalue`リファレンスの初期化子が、`lvalue`のビットフィールドの場合は、一時オブジェクトが生成され、そのオブジェクトを参照する。

```
struct S
{
    int x : 8 ;
} ;

int main()
{
    S s ;
    // エラー
    int & ref = s.x ;
    // OK
    // ただし、crefが参照するのは、生成された一時オブジェクトである
    // s.xではない
    int const & cref = s.x ;
}
```

9.11 クラス宣言のネスト(Nested class declarations)

クラスは、他のクラスの内側で宣言することができる。これを、ネストされたクラス(nested class)という。

```
class Outer
{
    class Inner { } ; // ネストされたクラス
} ;

int main()
{
    Outer::Inner object ;
}
```

ネストされたクラスのスコープは、外側のクラスのスコープに従う。これは、名前探索の際も、外側のクラスのスコープが影響するということである。

```
int x ; // グローバル変数

struct Outer
{
    int x ; // Outer::x
    struct Inner
    {
        void f()
        {
            sizeof(x) ; // OK、sizeofのオペランドは未評価式。
Outer::xのサイズを返す
            x = 0 ; // エラー、Outer::xはOuterの非staticメンバ
—
            ::x = 0 ; // OK、グローバル変数
        }
    } ;
}
```

関数Inner::fの中で、xという名前を使うと、Outer::xが見つかる。これは、クラスInnerが、クラスOuterのスコープ内にあるためである。しかし、非staticデータメンバーであるOuter::xを使うためには、Outerのオブジェクトが必要なので、ここではエラーとなる。sizeofのオペランドは未評価式なので、問題はない。ただし、ここでxは、Outer::xである。グローバル変数のxではない。

ネストされたクラスのメンバー関数やstaticデータメンバーは、通常のクラス通り、クラス定義の外側、同じ名前空間内で定義することができる。

```
// グローバル名前空間
struct Outer
{
    struct Inner
    {
        static int x ;
        void f() ;
    } ;
} ;

// 同じ名前空間内
int Outer::Inner::x = 0 ;
void Outer::Inner::f() { }
```

また通常通り、クラスの宣言だけをして、定義を後で書くこともできる。

```
struct Outer
{
    class Inner ; // 宣言
} ;

class Outer::Inner { } ; // 定義
```

9.12 ローカルクラス宣言 (Local class declarations)

関数定義の中で、クラスを定義することができる。これをローカルクラス (local class) という。

```
int main()
{ // 関数定義
    class Local { } ; // ローカルクラス
    Local object ; // ローカルクラスのオブジェクト
}
```

ローカルクラスのスコープは、クラス定義の外側のスコープである。また、名前探索は、ローカルクラスが定義されている関数と同じとなる。

ローカルクラスは、定義されている関数内の自動変数を使うことは出来ない。typedef名やstatic変数などは使える。

```
int main()
{
    int x ; // ローカル変数

    typedef int type ;
    static int y ;

    class Local
    {
        void f()
        {
            x = 0 ; // エラー

            // typedef名やstatic変数などは使える
            type val ; // OK
            y = 0 ; // OK

            // OK、sizeofのオペランドは未評価式
            sizeof(x) ;
        }
    } ;
}
```

ローカルクラスは、通常のクラスより制限が多い。ローカルクラスをテンプレート宣言することはできない。メンバーテンプレートを持つこともできない。ローカルクラスのメンバー関数は、クラス定義内で定義されなければならない。ローカルクラスの外側でメンバー関数を定義する方法はない。staticデータメンバーを持つことはできない。

```
int main()
{
    // エラー、ローカルクラスはテンプレート宣言できない
    template < typename T >
    class Local
    {
        // エラー、ローカルクラスはメンバーテンプレートを持てない。
        template < typename U > void f() { }
```

```
// OK、ただし、ローカルクラスの外側でメンバー関数を定義する方法はない
void f() ;
// エラー、ローカルクラスはstaticデータメンバーを持つことはできない。
static int x ;
}
}
```

9.13 型名のネスト(Nested type names)

クラス内の型名を、ネストされた型名(nested type name)という。ネストされた型名を、クラスの外側で使うには、クラス名による修飾が必要である。

```
struct X
{
    typedef int I ;
    class Inner { } ;
    I member ;
} ;

X::I object ;
```

10 派生クラス(Derived classes)

クラスには、基本クラス指定によって基本クラスを指定することができる。基本クラス指定は、以下のような文法である。

基本句:
 : 基本指定子リスト

基本指定子リスト:
 基本指定子 ...^{opt}
 基本指定子リスト, 基本指定子 ...^{opt}

基本指定子:

アトリビュート指定子_{opt} 基本型指定子

アトリビュート指定子_{opt} virtual アクセス指定子_{opt} 基本型指定子

アトリビュート指定子_{opt} アクセス指定子 virtual_{opt} 基本型指定子

基本型指定子:

クラスもしくはdecltype

アクセス指定子:

private

protected

public

基本クラス指定子に指定されたクラスのことを、基本クラス(base class)という。また、基本クラスを指定したクラスを、基本クラスに対する、派生クラス(derived class)という。クラスの基本クラス指定に指定されているクラスを、クラスの直接の基本クラス(direct base class)という。基本クラス指定には指定されていないものの、直接の基本クラスを通じて基本クラスとなっているクラスを、クラスの間接の基本クラス(indirect base class)という。単に基本クラスという場合、直接の基本クラスと間接の基本クラスの両方を意味する。

他のプログラミング言語の中には、基本クラスのことをスーパークラスと呼び、派生クラスのことをサブクラスと名付けている言語もある。C++では、そのような名称は用いない。これは、スーパーとサブでは意味が分かりにくく、他ならぬBjarne Stroustrup自身が考えたためである。そのため、C++では、スーパーのかわりに基本(base)、サブのかわりに派生(derived)という言葉を用いることになった。

基本クラスは、基本指定子に記述する。これは、

```
struct Base { } ;
struct Derived1 : Base { } ;
struct Derived2 : Derived1 { } ;
```

ここでは、Derived1の基本クラスはBaseである。Derived2の基本クラスはDerived1とBaseである。Derived2の直接の基本クラスはDerived1、間接の基本クラスはBaseである。

派生(derived)と継承(inherited)という言葉には、規格上、明確な違いがある。

派生という言葉は、派生クラスと基本クラスの関係を記述するために用いられる。あるクラスが基本クラスを持つ場合、「あるクラスは、基本クラスから、派生される(A class is derived from its base class)」という。「DerivedクラスはBaseクラスから派生されている(The Derived class is derived from the Base class.)」といえば、以下のようなコードを意味する。

```
struct Base { } ;
struct Derived : Base { } ;
```

この場合、DerivedクラスはBaseクラスから派生されている、という。クラスの派生関係に、継承という言葉を使うのは誤りである。

ただし、継承されている基本クラス(inherited base class)という言い方をすることはある。これは、対象が基本クラスで、これに対して派生という言葉を用いると、派生クラスという意味になってしまうからだ。

継承という言葉は、クラスのメンバーに対して用いられる。「基本クラスのメンバーは、派生クラスに継承される(The Base class's member is inherited by the derived class.)」という。例えば、「Baseクラスのメンバー関数fは、Derivedクラスに、継承されている(The Base class's member function f is inherited by the Derived class.)」といえば、以下のようなコードを意味する。

```
struct Base { void f() ; } ;
// DerivedはBase::fを継承
struct Derived : Base { } ;
```

クラスのメンバーに対して、派生という言葉は使うのは誤りである。

基本クラス指定子に...が使われた場合、パック展開とみなされる。

```
template < typename ... Types >
struct X : Types ... { } ;
```

アクセス指定については、11 [メンバーのアクセス指定](#)を参照。

10.1 複数の基本クラス(Multiple base classes)

基本クラスは、複数指定することができる。これを、複数の基本クラスという。複数の基本クラスを指定することを、俗に、多重継承(Multiple Inheritance)ということがあるが、これは、C++の規格上、正しい用語ではない。継承は、基本クラスのメンバーを派生クラスも受け継ぐことを意味する用語であって、クラスの派生関係を表すのに使う言葉ではないからだ。

ただし、歴史的に言えば、Multiple Inheritanceという言葉を最初に使ったのは、他ならぬBjarne Stroustrupご本人である。当時、Stroustrup氏が複数の基本クラスの設計をしていました時に使った言葉が、多重継承であった。ちなみに、多重継承が初めて使われたコードは、Jerry Schwarzによって書かれたiostreamである。

複数の基本クラスは、コンマで区切ることによって指定する。

```
struct A { } ; struct B { } ; struct C { } ;
struct D
    : A, B, C
{ } ;
```

この例では、Dは、A、B、Cという3個の基本クラスを持っている。

同じクラスを複数、直接の基本クラスとして指定することは出来ない。間接の基本クラスとしては指定できる。

```
struct Base { } ;

struct Derived
    : Base, Base // エラー、直接の基本クラス
{ } ;

struct Derived1 : Base { } ;
struct Derived2 : Base { } ;
struct Derived3
    : Derived1, Derived2 // OK、間接の基本クラス
{ } ;
```

この場合、Derived3は、Baseクラスのサブオブジェクトを、2個持つことになる。

基本クラスに、virtualが指定されていない場合、非virtual基本クラス(non-virtual base class)となる。非virtual基本クラスには、それぞれ独立したサブオブジェクトが割り当たされる。

同じクラスが複数、非virtual基本クラスとして存在することは、基本クラスのメンバーの名前に対するオブジェクトが容易に曖昧になる。このとき、派生クラスから基本クラスのメンバーを使うには、名前を正しく修飾しなければならない。

```
struct Base { int member ; } ;
struct Derived1 : Base { } ;
struct Derived2 : Base { } ;

// Derived3には、2個のBaseサブオブジェクトが存在する
struct Derived3 : Derived1, Derived2
{
    void f()
    {
        member ; // エラー、曖昧
        Base::member ; // エラー、曖昧
```

```

        Derived1::member ; // OK
        Derived2::member ; // OK
    }
}

int main()
{
    Derived3 x ;
    x.member ; // エラー、曖昧
    x.Derived1::member ; // OK
    x.Derived2::member ; // OK
}

```

ただし、staticメンバーの名前は、曖昧にならない。これは、staticメンバーの利用には、クラスのオブジェクトは必要ないからである。

```

struct Base
{
    static void static_member() { }
    static int static_data_member ;
};

int Base::static_data_member = 0 ;

struct Derived1 : Base { } ;
struct Derived2 : Base { } ;

struct Derived3 : Derived1, Derived2
{
    void f()
    {
        static_member() ; // OK
        static_data_member ; // OK
    }
};

```

直接、間接の両方の基本クラスに、同じクラスを持つことは可能である。ただし、そのような派生クラスは、基本クラスの非staticメンバーを使うことができない。なぜなら、基本クラスの名前自体の曖昧性を解決する方法がないからだ。

```

struct Base
{
    int member() ; // 非staticメンバー
}

```

```

    static void static_member() { } // staticメンバー
} ;
struct Derived1 : Base { } ;

// Baseという名前自体が曖昧になる
struct Derived2 : Base, Derived1
{
    void f()
    {
        // Baseの非staticメンバーを使う方法はない

        static_member(); // OK、staticメンバーは使える
    }
} ;

```

このため、直接、間接の両方で同じクラスを基本クラスに持つ派生クラスの利用は、かなり制限される。

基本クラスに、`virtual`が指定されている場合、`virtual`基本クラス(`virtual base class`)という。`virtual`基本クラスには、ひとつしかオブジェクトが割り当てられない。`virtual`基本クラスのオブジェクトは、派生クラスで共有される。

```

struct L { } ;
struct A : virtual L { } ;
struct B : virtual L { } ;
struct C : A, B { } ;

```

この例で、Cクラスには、Lのサブオブジェクトは1個存在する。これは、A、Bで共有される。

`virtual`基本クラスでは、サブオブジェクトが共有されているため、`virtual`基本クラスのメンバーは、曖昧にならない。

```

struct Base { int member ; } ;
struct Derived1 : virtual Base { } ;
struct Derived2 : virtual Base { } ;
struct Derived3 : Derived1, Derived2
{
    void f()
    {
        member ; // OK
    }
} ;

```

非virtual基本クラスとvirtual基本クラスは、両方持つことができる。

```
struct B { } ;
struct X : virtual B { } ;
struct Y : virtual B { } ;
struct Z : B { } ;
struct A : X, Y, Z { } ;
```

この例では、Aクラスには、Bのサブオブジェクトは、2個存在する。X、Yで共有されるサブオブジェクトと、Zのサブオブジェクトである。

10.2 メンバーの名前探索 (Member name lookup)

メンバーの名前探索は、すこし難しい。派生クラスのメンバーネームは、基本クラスのメンバーネームを隠すということだ。あるメンバーネームを名前探索する際に、派生クラスで名前が見つかった場合、その時点で名前探索は終了する。基本クラスのメンバーを探すことはない。

```
struct Base
{
    void f( int ) { }
} ;

struct Derived : Base
{
    void f( double ) { }
} ;

int main()
{
    Derived object;
    object.f( 0 ) ; // Derived::f( double )が呼ばれる
}
```

ここで、Derivedクラスには、二つのfという名前のメンバーが存在する。Derived::fとBase::fである。もし、名前探索によって両方の名前が発見された場合、オーバーロード解決によって、Base::f(int)が選ばれるはずである。しかし、実際には、Derived::f(double)が選ばれる。これは、Derivedクラスに、fという名前のメンバーが存在するので、その時点で名前探索が終了するからである。Baseのメンバーネームは発見されない。名前が発見

されない以上、オーバーロード解決によって選ばれることもない。

これは、名前探索に対するルールなので、型は関係がない。

```
// fという名前のint型のデータメンバー
struct Base { int f ; } ;
// fという名前のvoid (void)型のメンバー関数
struct Derived : Base { void f( ) { } } ;

int main()
{
    Derived object;
    object.f = 0 ; // エラー、メンバー関数Derived::fに0を代入す
    ることはできない
    object.Base::f = 0 ; // OK、明示的な修飾
}
```

したがって、基本クラスと同じ名前のメンバーを派生クラスで使う際には、注意が必要である。

名前探索という仕組みを考えずに、この挙動を考えた場合、これは、派生クラスのメンバー名が、基本クラスのメンバー名を、隠していると考えることもできる。もし、基本クラスのメンバー名を隠したくない場合、7.5.3 [using宣言](#)を使うことができる。[using宣言](#)を使うと、基本クラスのメンバー名を、派生クラスのスコープに導入することができる。

```
struct Base
{
    void f( int ) { }
} ;

struct Derived : Base
{
    using Base::f ; // using宣言
    void f( double ) { }
} ;

int main()
{
    Derived object;
    object.f( 0 ) ; // Base::f( int )が呼ばれる
}
```

名前探索で、派生クラスのメンバーが見つからない場合は、直接の基本クラスのメン

バーから、名前が探される。

```
struct Base { int member ; } ;
struct Derived : Base
{
    void f()
    {
        member ; // Base::member
    }
} ;
```

メンバー名を探す基本クラスは、直接の基本クラスだけである。間接の基本クラスのメンバーは、直接の基本クラスを通じて、探される。

```
struct A { int member ; } ;
struct B : A { } ;
struct C : B
{
    void f()
    {
        member ; // A::member
    }
} ;
```

この例では、C::fでmemberという名前のメンバーを使っている。Cクラスにはmemberという名前のメンバーが見つからないので、名前探索はBクラスに移る。クラスは、基本クラスのメンバー名を継承している。そのため、Bクラスの基本クラスのAクラスのメンバー名は、Bクラスのスコープからも発見することができる。

直接の基本クラスが複数ある場合、それぞれの直接の基本クラスから、名前が探される。この際、複数のクラスから同じ名前が発見され、名前の意味が違う場合、名前探索は無効となる。

```
struct Base1 { void member( int ) { } } ;
struct Base2 { void member( double ) { } } ;
struct Derived : Base1, Base2 // 複数の直接の基本クラス
{
    void f()
    {
        member( 0 ) ; // エラー、名前探索が無効
        Base1::member( 0 ) ; // OK
    }
}
```

```
}
```

これは、memberという名前に対し、複数の直接の基本クラスで、複数の同じ名前が見つかり、しかも意味が違っているので、名前検索が無効となる。その結果、memberという名前が見つからず、エラーとなる。

もし、この例で、Derivedから、明示的な修飾をせずに、両方の基本クラスのメンバー関数を呼び出したい場合、7.5.3 [using宣言](#)が使える。

```
struct Base1 { void member( int ) { } } ;
struct Base2 { void member( double ) { } } ;
struct Derived : Base1, Base2
{
    // 基本クラスのメンバーネームをDerivedスコープで宣言する
    using Base1::member ;
    using Base2::member ;

    void f()
    {
        member( 0 ) ; // OK、オーバーロード解決により、
Base1::member(int)が呼ばれる
    }
}
```

この例は、複数の直接の基本クラスがある場合の制限である。複数の間接の基本クラスでは、名前探索が失敗することはない。ただし、名前探索の結果として、複数の名前が発見され、曖昧になることはある。

10.3 virtual関数 (Virtual functions)

本書のサンプルコードは、解説する文法のための最小限のコードであり、virtual関数を持つクラスがvirtualデストラクターを持たないことがある。これは現実ではほとんどの場合、不適切である。

メンバー関数にvirtual指定子を指定すると、virtual関数となる。virtual関数を宣言しているクラス、あるいはvirtual関数を継承しているクラスは、ポリモーフィッククラス (polymorphic class) となる。

```
struct Base
{
    virtual void f() { } // virtual関数
```

```
} ;
struct Derived : Base { } ;
```

BaseとDerivedは、ポリモーフィッククラスである。

クラスがポリモーフィックであるかどうかということは、dynamic_castやtypeidを使う際に、重要である。

基本クラスのvirtual関数は、派生クラスのメンバーに、同じ名前、同じ仮引数リスト、同じCV修飾子、同じリファレンス修飾子という条件を満たすメンバー関数があった場合、オーバーライドされる。この時、派生クラスのメンバー関数は、virtual指定子がなくても、自動的にvirtual関数になる。

```
struct A { virtual void f() {} } ;
struct B : A { } ; // オーバーライドしない
struct C : A
{
    void f() {} // オーバーライド
} ;
struct D : C
{
    void f(int) {} // オーバーライドしない
    void f() const {} // オーバーライドしない
} ;

// リファレンス修飾子が違う例
struct Base { virtual void f() & {} } ;
struct Derived : Base { void f() && {} } ;
```

もちろん、virtualをつけてもよい。

```
struct Base { virtual f() {} } ;
struct Derived : Base { virtual f() {} } ; // オーバーライド
```

派生クラスで、最後にオーバーライドしたvirtual関数を、ファイナルオーバーライダー（final overrider）と呼ぶ。あるクラスのオブジェクトに対して、virtual関数を呼び出す際は、オブジェクトの実行時の型によって、最後にオーバーライドしたvirtual関数が呼び出される。これは、基本クラスのポインターやリファレンスを経由してオブジェクトを使った場合でも、同様である。通常のメンバー関数は、virtual関数とは違い、実行時の型チェックを行わない。オブジェクトを指しているリファレンスやポインターの型によって、決定される。

```
// virtual関数と非virtual関数の違いの例
struct A
{
    virtual void virtual_function() { }
    void function() { }
} ;
struct B : A
{
    virtual void virtual_function() { }
    void function() { }
} ;
struct C : B
{
    virtual void virtual_function() { }
    void function() { }
} ;

void call( A & ref )
{
    ref.virtual_function();
    ref.function();
}

int main()
{
    A a ; B b ; C c ;

    call( a ) ; // A::virtual_function, A::functionが呼び出される
    call( b ) ; // B::virtual_function, A::functionが呼び出される
    call( c ) ; // C::virtual_function, A::functionが呼び出される
}
```

Aは、virtual functionとfunctionという名前のvirtual関数を持っており、Aから派生しているB、Bから派生しているCは、オーバーライドしている。call関数の仮引数refは、オブジェクトの型が、実際に何であるかは、実行時にしか分からない。virtual関数であるvirtual_functionは、オブジェクトの型に合わせて正しく呼び出されるが、virtual関数ではないfunctionは、Aのメンバーが呼び出される。

virt指定子(virt-specifier)は、finalかoverrideで、virtual関数の宣言子の後、pure指定子の前に記述できる。

```
// virt指定子の文法の例示のための記述
virtual f() final override = 0 ;
```

finalが指定されたvirtual関数を持つクラスから派生したクラスが、同virtual関数をオーバーライドした場合はエラーになる。

```
struct base
{
    virtual void f() { }
} ;

struct derived
{
    virtual void f() final { }
} ;

struct ok : derived
{
// OK
} ;

struct error : derived
{
    // エラー、final指定されているderived::fをオーバーライド
    virtual void f() { }
} ;
```

virtual関数にfinalを指定すると、それ以上のオーバーライドを禁止できる。

overrideが指定されたvirtual関数が、基本クラスのメンバー関数をオーバーライドしていない場合、エラーとなる。

```
struct base
{
    virtual void virtual_function() { }
} ;

struct ok : base
{
    // OK、ok::virtual_functionはbase::virtual_functionをオーバーライドしている
    virtual void virtual_function() override { }
```

```

} ;

struct typo : base
{
    // OK、typo::virtual_functionはbase::virtual_functionとは別
    // のvirtual関数
    virtual void virtual_function() { }
} ;

struct error : base
{
    // エラー、error::virtual_functionはオーバーライドしていない
    virtual void virtual_function() override { }
} ;

```

これにより、タイプミスによる些細な間違いをコンパイル時に検出できる。

オーバーライドであることに注意。以下のコードはエラーである。

```

struct base
{
    void f() { } // 非virtual関数
} ;

struct error : base
{
    // エラー、オーバーライドしていない
    virtual void f() override { }
} ;

```

finalとoverrideを両方指定することもできる。

virtual関数をオーバーライドする関数は、戻り値の型が同じでなくても構わない。ただし、何でもいいというわけではない。戻り値の型は、まったく同じ型か、相互変換可能(covariant)でなければならない。covariantは、以下のような条件をお互いに満たした型のことである。

今、関数D::fが、関数B::fをオーバーライドしているとする。

```

// D::f、B::fの例
struct B { virtual 戻り値の型 f() ; } ;
struct D : B { virtual 戻り値の型 f() ; } ;

```

その場合、戻り値の型は、以下の条件を満たさなければならない。

お互にクラスへのポインター、もしくは、お互にクラスへのlvalueリファレンス、もしくは、お互にクラスへのrvalueリファレンスであること。

片方がポインターで片方がリファレンスの場合や、片方がlvalueリファレンスで片方がrvalueリファレンスの場合は、不適である。もちろん、ポインターでもリファレンスでもない型は不適である。また、クラスでもない型へのポインターやリファレンスも不適である。

```
// ポインター
struct B { virtual B * f() ; } ;
struct D : B { virtual D * f() ; } ;
// lvalueリファレンス
struct B { virtual B & f() ; } ;
struct D : B { virtual D & f() ; } ;
// rvalueリファレンス
struct B { virtual B && f() ; } ;
struct D : B { virtual D && f() ; } ;
```

B::fの戻り値の型のクラスは、D::fの戻り値の型のクラスと同じか、曖昧がなくアクセスできる基本クラスでなければならぬ。

オーバーライドしている関数が、基本クラスを戻り値に使ってたり、そもそもクラスの派生関係にない場合は、不適である。private派生していて、派生クラスからはアクセスできない場合や、基本クラスのサブオブジェクトが複数あって曖昧な場合はエラーとなる。

```
struct Base { } ; // 基本クラス
struct Derived : Base { } ; // 派生クラス
struct Other { } ; // BaseやDerivedとは派生関係にないクラス

// クラスが同じ
struct B { virtual Base & f() ; } ;
struct D : B { virtual Base & f() ; } ;

// B::fのクラスはD::fのクラスの基本クラス
struct B { virtual Base & f() ; } ;
struct D : B { virtual Derived & f() ; } ;

// エラー
```

```

struct B { virtual Derived & f() ; } ;
struct D : B { virtual Base & f() ; } ;

// エラー
struct B { virtual Base & f() ; } ;
struct D : B { virtual Other & f() ; } ;

```

両方のポインターは同じCV修飾子を持たなければならない。D::fの戻り値の型のクラスは、B::fの戻り値の型のクラスと同じCV修飾子を持つか、あるいは少ないCV修飾子を持たなければならない。

補足: ポインターに対するCV修飾子とは、T cv1 * cv2という型がある場合、cv2である。クラスに対するCV修飾子は、cv1である。

```

// int *に対するCV修飾子
int * const
// intに対するCV修飾子
const int *
int const *

```

// 両方のポインターは同じCV修飾子を持たなければならない例

```

// ポインターのCV修飾子はconst
struct B { virtual B * const f() ; } ;
// OK
struct D : B { virtual D * const f() ; } ;

// エラーのDクラスの例、ポインターのCV修飾子が一致していない
struct D : B { virtual D * f() ; } ;
struct D : B { virtual D * volatile const f() ; } ;
struct D : B { virtual D * const volatile f() ; } ;

```

// D::fの戻り値の型のクラスは、B::fの戻り値の型のクラスと同じCV修飾子を持つか、
// あるいは少ないCV修飾子を持たなければならない例

// B::fの戻り値の型のクラスのCV修飾子はconst

```

struct B { virtual B const & f() ; } ;
// 問題ないDクラスの例、CV修飾子が同じか少ない
struct D : B { virtual D const & f() ; } ;
struct D : B { virtual D & f() ; } ;

// エラーのDクラスの例、CV修飾子が多い
struct D : B { virtual D volatile & f() ; } ;
struct D : B { virtual D const volatile & f() ; } ;

```

明示的な修飾を用いた場合は、virtual関数呼び出しが阻害される。これは、オーバーライドしたvirtual関数から、オーバーライドされたvirtual関数を呼び出すのに使える。

```

struct Base { virtual void f() { } } ;
struct Derived : Base
{
    virtual void f()
    {
        f() ; // Derived::fの呼び出し
        Base::f() ; // 明示的なBase::fの呼び出し
    }
} ;

```

virtual関数とdelete定義は併用できる。ただし、delete定義のvirtual関数を、非delete定義のvirtual関数でオーバーライドすることはできない。非delete定義のvirtual関数を、delete定義のvirtual関数でオーバーライドすることはできない。

```

// OK、delete定義のvirtual関数を、delete定義のvirtual関数でオーバーライドしている
struct Base { virtual void f() = delete ; } ;
struct Derived : Base { virtual void f() = delete ; } ;

// エラー、非delete定義ではないvirtual関数を、delete定義のvirtual関数でオーバーライドしている
struct Base { virtual void f() { } } ;
struct Derived : Base { virtual void f() = delete ; } ;

// エラー、delete定義のvirtual関数を、非delete定義のvirtual関数でオーバーライドしている
struct Base { virtual void f() = delete ; } ;
struct Derived : Base { virtual void f() { } } ;

```

10.4 アブストラクトクラス (Abstract classes)

ピュア指定子:

= 0

アブストラクトクラス (abstract class) は、抽象的な概念としてのクラスを実現する機能である。これは、例えば図形を表すクラスである、CircleやSquareなどといったクラスの基本クラスであるShapeや、動物を表すDogやCatなどといったクラスの基本クラスであるAnimalなど、異なるクラスに対する共通のインターフェースを提供する目的に使える。

```
struct Shape
{
    // 図形描画用の関数
    // Shapeクラスは抽象的な概念であり、具体的な描画方法を持たない
    // 単に共通のインターフェースとして提供される
    virtual void draw() = 0 ;
};

struct Circle : Shape
{
    virtual void draw() { /* 円を描画 */ }
};

struct Square : Shape
{
    virtual void draw() { /* 正方形を描画 */ }
};

void f( Shape * ptr )
{
    ptr->draw() ; // 実行時の型に応じて図形を描画する
}
```

ここでは、Shapeクラスというのは、具体的に描画する方法を持たない。そもそも、Shapeクラス自体のオブジェクトを使うことは想定されていない。このように、そのクラス自体は抽象的な概念であり、実体を持たない場合、ピュアvirtual関数を使うことで、共通のインターフェースとすることができる。

他の言語では、この機能を明確にクラスから分離して、「インターフェース」という名前の機能にしているものもある。C++では、抽象クラスも、制限はあるものの、クラスの一種である。

少なくともひとつのピュアvirtual関数を持つクラスは、アブストラクトクラスとなる。ピュアvirtual関数は、virtual関数の宣言に、ピュア指定子を書くことで宣言できる。

ピュア指定子:

= 0

```
struct abstract_class
{
    virtual void f() = 0 ;
} ;
```

ピュアvirtual関数は、呼ばれない限り、定義する必要はない。

```
struct Base
{
    virtual void f() = 0 ;
    virtual void g() = 0 ;
} ;

struct Derived
{
    virtual void g() { }
} ;

void call_g( Base & base )
{
    base.g() ;
}

int main()
{
    Derived d ;
    call_g( d ) ;
}
```

一つの関数宣言にピュア指定子と定義を両方書くことはできない。

```
struct X
```

```
{
    // エラー
    virtual void f() = 0 { } ;
}
```

ただし、複数の関数宣言を使えば、ひとつの関数にピュア指定子と定義を両方与えることができる。

```
struct X
{
    // OK、ピュア指定子を与える関数宣言
    virtual void f() = 0 ;
}

// OK、定義を与える関数宣言
void X::f() { }
```

この仕様は、オブジェクトの破棄の際に何らかの処理を行いたい抽象クラスに使うことができる。デストラクターを純粋仮想関数かつ定義付きの関数とすることができる。

```
class Base
{
    int * ptr ;
public :
    Base( int value )
        : ptr( new int(value) )
    {
    }
    virtual ~Base() = 0 ;
}

Base::~Base()
{
    delete ptr ;
}
```

ただし、デストラクターの呼び出しは、通常のメンバー関数とは異なっているので、注意が必要である。以下のコードを考える。

```
struct Base
{
```

```
virtual void f() = 0 ;
virtual ~Base() = 0 ;
} ;

void Base::f() { }
Base::~Base
{
// Derivedはすでに破棄されている。

    f() ; // エラー、Base::fのvirtual関数呼び出しの挙動は未定義
}

struct Derived : Base
{
    virtual void f() { }
    virtual ~Base() { }
} ;

int main()
{
    Derived d ;
}
```

Baseのデストラクター呼び出しは問題がない。なぜならば、基本クラスのデストラクターは、あたかも明示的に直接呼び出されたかのように振る舞うからだ。`virtual`関数呼び出しではない。

ただし、デストラクターの中で未修飾名のfを呼び出すと、これはvirtual関数呼び出しになる。オブジェクトの構築中、破棄中にvirtual関数を呼び出した場合、オブジェクトの型が、あたかも最終的な派生クラスの型とみなされる。そのため、ここではBase::fがvirtual関数呼び出しがれる。

Base::fは、定義が与えられてはいるものの、依然としてピュアvirtual関数であることに変わりはない。ピュアvirtual関数をvirtual関数呼び出した場合の挙動は未定義である。そのため、上記のコードは、以下のように、明示的な修飾名で呼び出し、通常の関数呼び出しにしなければ、規格上、動作が保証されない。

```
Base::~Base
{
    Base::f() ; // OK、修飾名は通常の関数呼び出し
}
```

この、`=0`という文法は、初期化子や代入式とは、何の関係もない。ただ、C++の文法上、メンバー関数の宣言の中の、`=0`というトークン列を、特別な意味を持つものとして

扱っているだけである。ピュア指定子を記述する位置は、*virt-specifier*の後である。

```
struct Base { virtual void f() { } }
struct abstract_class : Base
{
    virtual void f() override = 0 ; // virt-specifierの後
} ;
```

アブストラクトクラスは、他のクラスの基本クラスとして使うことしかできない。アブストラクトクラスのオブジェクトは、派生クラスのサブオブジェクトとしてのみ、存在することができる。

```
struct abstract_class
{
    virtual void f() = 0 ;
} ;

struct Derived : abstract_class
{
    void f() { }
} ;
```

アブストラクトクラスのオブジェクトを、直接作ることはできない。これには、変数や関数の仮引数、new式などが該当する。

```
struct abstract_class
{
    virtual void f() = 0 ;
} ;

// エラー、abstract_classのオブジェクトは作れない
void f( abstract_class param )
{
    abstract_class obj ; // エラー
    new abstract_class ; // エラー
}
```

アブストラクトクラスへのポインターやリファレンスは使える。

```

struct abstract_class
{
    virtual void f() = 0 ;
} ;

// OK、ポインターとリファレンスはよい
void f( abstract_class *, abstract_class & ) ;

```

ピュアvirtual関数を継承していく、ファイナルオーバーライダーがピュアvirtual関数である場合も、アブストラクトクラスとなる。これは例えば、アブストラクトクラスから派生されているクラスが、ピュアvirtual関数をオーバーライドしていなかった場合などが、該当する。

```

struct Base { virtual void f() = 0 ; } ;
struct Derived : Base { } ;

```

この場合、Derivedも、Baseと同じく、アブストラクトクラスになる。

派生クラスによって、ピュアvirtual関数ではないvirtual関数をオーバーライドして、ピュアvirtual関数にすることができる。その場合、派生クラスはアブストラクトクラスとなる。

```

struct Base { virtual void f() { } } ;
struct Derived : Base { virtual void f() = 0 ; } ;

int main()
{
    Base b ; // OK
    Derived d ; // エラー
}

```

この例では、Baseはアブストラクトクラスではない。Derivedはアブストラクトクラスである。

構築中、または破棄中のアブストラクトクラスのコンストラクタやデストラクターの中で、ピュアvirtual関数を呼び出した場合の挙動は、未定義である。

```

struct Base
{
    virtual void f() = 0 ;

    // この関数を、Baseのコンストラクタやデストラクターから呼ぶ

```

```
とエラー
void g()
{ f() ; }

// コンストラクター
Base() // エラー、未定義の挙動
{ f() ; }

// デストラクター
~Base() // エラー、未定義の挙動
{ f() ; }
} ;

struct Derived : Base
{
    virtual void f() { }

    // Derivedはアブストラクトクラスではないので、問題はない
    Derived() { f() ; }
    ~Derived() { f() ; }
} ;
```

11 メンバーのアクセス指定 (Member access control)

クラスのメンバーは、private、protected、publicのいずれかのアクセス指定を持つ。

privateが指定されたメンバーは、同じクラスのメンバーとfriendから使うことができる。

protectedが指定されたメンバーは、同じクラスと、そのクラスから派生されているクラスのメンバーとfriendから使うことができる。

publicでは、メンバーはどこからでも制限なく使える。

```
class Base
{
private :
    int private_member ;
protected :
    int protected_member ;
public :
    int public_member ;
```

```

void f()
{
    // 同じクラスのメンバー
    private_member ; // OK
    protected_member ; // OK
    public_member ; // OK
}
} ;

void f()
{ // クラス外
    Base base ;
    base.private_member ; // エラー
    base.protected_member ; // エラー
    base.public_member ; // OK
}

class Derived : public Base
{
    void f()
    {
        // 派生クラスのメンバー
        private_member ; // エラー
        protected_member ; // OK
        public_member ; // OK
    }
} ;

```

classキーワードで定義されたクラスのメンバーは、デフォルトでprivateになる。structキーワードで定義されたクラスのメンバーは、デフォルトでpublicになる。

```

// nameはprivate
class C { int name ; } ;
// nameはpublic
struct S { int name ; } ;

```

classとstructキーワードの違いは、デフォルトのアクセス指定子が異なるだけである。アクセス指定子を明示的に記述すると、classとstructキーワードの違いはなくなる。

アクセス指定は、メンバーの種類を問わず、名前に対して一律に適用される。アクセス指定は、名前探索に影響をあたえることはない。たとえアクセス指定によって使えない名前であっても、名前は発見される。名前が発見されること自体はエラーではない。ア

アクセス指定によって使えない名前を使おうとするとエラーになる。例えば、名前が関数のオーバーロードのセットであった場合、オーバーロード解決された結果の名前に対し、アクセス指定が適用される。

```
class X
{
private :
    void f( int ) { }
public :
    void f( double ) { }
} ;

int main()
{
    X x ;
    // エラー、privateメンバーにはアクセス出来ない
    // オーバーロード解決の結果はX::f( int )
    x.f( 0 ) ;
    // OK、X::f( double )が呼ばれる
    x.f( 0.0 ) ;
}
```

この例では、Xのメンバーfに対して、f(0)という関数呼び出しの式を適用している。アクセス指定は名前探索に影響をあたえることはないので、関数オーバーロードのセットとして、X::f(int)と、X::f(double)という名前が発見される。そして、オーバーロード解決によって、X::f(int)が最適な関数として選ばれる。アクセス指定のチェックは、オーバーロード解決の後に行われる。この場合、X::f(int)はprivateメンバーなので、Xのメンバーでもfriend関数でもないmain関数から呼び出すことはできない。

11.1 アクセス指定子 (Access specifiers)

クラスのメンバーのアクセス指定は、ラベルにアクセス指定子 (Access specifiers) を記述することで指定する。

アクセス指定子 : メンバー指定_{opt}

アクセス指定 :

- private
- protected
- public

アクセス指定子とは、private、protected、publicのいずれかである。アクセス指定子が現れた場所から、次のアクセス指定子か、クラス定義の終了までの間のメンバーが、

アクセス指定子の影響を受ける。

```
class X
{
    int a ; // デフォルトのprivate
public :
    int b ; // public
    int c ; // public
protected :
    int d ; // protected
private :
    int e ; // private
} ;
```

アクセス指定子には、順番や使用可能な回数の制限はない。好きな順番で、何度も指定できる。

```
class X
{
public :
public :
protected :
public :
public :
private :
} ;
```

11.2 基本クラスと、基本クラスのメンバーへのアクセス指定 (Accessibility of base classes and base class members)

あるクラスを、別のクラスの基本クラスとするとき、いずれかのアクセス指定子を指定する。

```
class Base { } ;

class Derived_by_public : public Base { } ; // public派生
class Derived_by_protected : protected Base { } ; //
protected派生
```

```
class Derived_by_private : private Base { } ; // private派生
```

アクセス指定子がpublicの場合、基本クラスのpublicメンバーは、派生クラスのpublicメンバーとしてアクセス可能になり、基本クラスのprotectedメンバーは、派生クラスのprotectedメンバーとしてアクセス可能になる。

```
class Base
{
public :
    int public_member ;
protected :
    int protected_member ;
} ;

class Derived : public Base
{
    void f()
    {
        public_member ; // OK
        protected_member ; // OK
    }
} ;

int main()
{
    Derived d ;
    d.public_member ; // OK
}
```

アクセス指定子がprotectedの場合、基本クラスのpublicとprotectedメンバーは、派生クラスのprotectedメンバーとしてアクセス可能になる。

```
class Base
{
public :
    int public_member ;
protected :
    int protected_member ;
} ;

class Derived : protected Base
{
```

```
void f()
{
    public_member ; // OK、ただしprotectedメンバー
    protected_member ; // OK
}
} ;

int main()
{
    Derived d ;
    d.public_member ; // エラー、Derivedからは、protectedメンバー
    である
}
```

アクセス指定子がprivateの場合、基本クラスのpublicとprotectedメンバーは、派生クラスのprivateメンバーとしてアクセス可能になる。

```
class Base
{
public :
    int public_member ;
protected :
    int protected_member ;
} ;

class Derived : private Base
{
    void f()
    {
        public_member ; // OK、ただし、privateメンバー
        protected_member ; // OK、ただし、privateメンバー
    }
} ;

class Derived2 : public Derived
{
    void f()
    {
        public_member ; // エラー、基本クラスのprivateメンバー
        にはアクセスできない
        protected_member ; // エラー、基本クラスのprivateメンバー
        にはアクセスできない
    }
}
```

```

} ;

int main()
{
    Derived d ;
    d.public_member ; // エラー、Derivedからは、privateメンバー
である
}

```

基本クラスにアクセス指定子を指定しなかった場合、structキーワードで宣言されたクラスは、デフォルトでpublicに、classキーワードで宣言されたクラスは、デフォルトでprivateになる。

```

struct Base { } ;

// デフォルトのpublic派生
struct D1 : Base { } ;
// デフォルトのprivate派生
class D2 : Base { } ;

```

どのアクセス指定子を指定しても、基本クラスのprivateメンバーを派生クラスから使うことはできない。クラスAからprivate派生したクラスBから派生しているクラスCでは、クラスAのメンバーは使えないのも、この理由による。

```

// classキーワードで宣言されたクラスのメンバーはデフォルトで
private
class Base { int private_member ; } ;

class Derived : public Base
{
// どのアクセス指定を用いても、基本クラスのprivate_memberは使えな
い
} ;

struct A { int public_member ; } ;
class B : private A { } ;
class C : public B
{
// クラスBは、クラスAからprivate派生しているため、ここでは
A::public_memberは使えない。
} ;

```

クラス名自体も、クラススコープ内の名前として扱われる。クラスAからprivate派生したクラスBから派生しているクラスCでは、クラスAのクラス名自体がprivateメンバーになってしまう。

```
// グローバル名前空間のスコープ
struct A { } ;
class B : private A { } ;
class C : public B
{
    void f()
    {
        A a1 ; // エラー、名前Aは、基本クラスのprivateメンバーの
A
        ::A a2 ; // OK、名前::Aは、グローバル名前空間スコープ内
のA
    }
} ;
```

この例では、クラスCのスコープ内で、非修飾名Aに対して、クラス名Aが発見されてしまうので、エラーになる。クラスCの中でクラスAを使いたい場合、明示的な修飾が必要である。

アクセス指定子は、staticメンバーにも適用される。publicなstaticメンバーを持つクラスを、protectedやprivateで派生すると、基本クラスからはアクセスできるが、派生クラスを介してアクセスできなくなってしまうこともある。

```
// グローバル名前空間のスコープ
struct A { static int data ; } ;
int A::data ;

class B : private A { } ;
class C : public B
{
    void f()
    {
        data ; // エラー
        ::A::data ; // OK
    }
} ;
```

クラスCからは、名前dataは、基本クラスAのメンバーdataとして発見されるので、アクセスできない。しかし、クラスA自体は、名前空間に存在するので、明示的な修飾を使えば、アクセスできる。

protectedの場合、friendではないクラス外部の関数からアクセスできなくなる。

```
struct A { static int data ; }
int A::data ;

class B : protected A { } ;

int main()
{
    B::data ; // エラー
    A::data ; // OK
}
```

ここでは、B::dataとA::dataは、どちらも同じオブジェクトを指しているが、アクセス指定の違いにより、B::dataという修飾名では、クラスBのfriendではないmain関数からアクセスすることができない。

基本クラスにアクセス可能である場合、派生クラスへのポインター型から、基本クラスへのポインター型に型変換できる。

```
class A { } ;
class B : public A { } ;
class C : protected A
{
    void f()
    {
        static_cast< A * >( this ) ; // OK、アクセス可能
    }
} ;

int main()
{
    B b ;
    static_cast< A * >( &b ) ; // OK、アクセス可能
    C c ;
    static_cast< A * >( &c ) ; // エラー、main関数からは、
protectedメンバーにアクセスできない
}
```

11.3 friend(Friends)

クラスはfriendを宣言することができる。friendを宣言するには、friend指定子を使う。クラスのfriendとして宣言できるものは、関数かクラスである。クラスのfriendは、クラスのprivateとprotectedメンバーにアクセスできる。

```
class X
{
private :
    typedef int type ; // privateメンバー
    friend void f() ; // friend関数
    friend class Y ; // friendクラス
} ;

void f()
{
    X::type a ; // OK、関数void f(void)はXのfriend
}

class Y
{
    X::type member ; // OK、クラスYはXのfriend
    void f()
    {
        X::type member ; // OK、クラスYはXのfriend
    }
} ;
```

friendクラスの宣言は、friend指定子に続けて、7.3.6.3 複雑型指定子、7.3.6.2 単純型指定子、typename指定子(14.5 名前解決を参照)のいずれかを宣言しなければならない。

複雑型指定子は、最も分かりやすい。

```
class X
{
    friend class Y ;
    friend struct Z ;
} ;
```

複雑型指定子を使う場合、クラスをあらかじめ宣言しておく必要はない。名前がクラスであることが、その時点で宣言されるからだ。

単純型指定子に名前を使う場合は、それより以前に、クラスを宣言しておく必要がある。

```
class Y ; // Yをクラスとして宣言

class X
{
    friend Y ; // OK、Yはクラスである
    friend Z ; // エラー、名前Zは見つからない

    friend class A ; // OK、Aはクラスとして、ここで宣言されている
} ;
```

あらかじめ名前が宣言されていない場合は、エラーとなる。

単純型指定子にテンプレート名を使うこともできる。

```
template < typename T >
class X
{
    friend T ; // OK
} ;
```

typename指定子を指定する場合は、以下のようになる。

```
template < typename T >
class X
{
    friend typename T::type ;
} ;
```

T::typeは、依存名を型として使っているので、typenameが必要である。

もし、型指定子がクラス型ではない場合、単に無視される。これは、テンプレートコードを書くときに便利である。

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

```

    friend T ;
} ;

X<int> x ; // OK、friend宣言は無視される

template < typename T >
class Y
{
    friend typename T::type ;
} ;

struct Z { typedef int type ; } ;

Y<Z> y ; // OK、friend宣言は無視される

```

無視されるのは、あくまで、型指定子がクラス型ではなかった場合である。すでに説明したように、単純型指定子で、名前が見つからなかった場合は、エラーになる。

friend関数の宣言は、通常通りの関数の宣言の文法に、friend指定子を記述する。前方宣言は必須ではない。friend関数には、7.3.1 [ストレージクラス指定子](#)を記述することはできない。

```

class X
{
    friend void f() ;
    friend int g( int, int, int ) ;
    friend X operator + ( X const &, X const & ) ;
} ;

```

friend関数として宣言された関数がオーバーロードされていた場合でも、friend関数として宣言したシグネチャの関数しか、friendにはならない。

```

void f( int ) ;
void f( double ) ;

class X
{
    friend void f( int ) ;
} ;

```

この例では、void f(int)のみが、Xのfriend関数になる。void f(double)は、friend関数にはならない。

他のクラスのメンバー関数も、friend関数として宣言できる。メンバー関数には、コンストラクタやデストラクターも含まれる。

```
class X ; // 名前Xをクラス型として宣言

class Y
{
public :
    void f( ) ; // メンバー関数
    Y & operator = ( X const & ) ; // 代入演算子
    Y() ; // コンストラクター
    ~Y() ; // デストラクター
} ;

class X
{
    // 以下4行は、すべて正しいfriend宣言
    friend void Y::f( ) ;
    friend Y & Y::operator = ( X const & ) ;
    friend Y::Y() ;
    friend Y::~Y() ;
} ;
```

friend宣言自体には、アクセス指定は適用されない。ただし、friend宣言の中でアクセスできない名前を使うことはできない。

```
class Y
{
private :
    void f( ) ; // privateメンバー
} ;

class X
{
    // エラー、Yのprivateメンバーにはアクセス出来ない
    // friend宣言の中の名前の使用には、アクセス指定が影響する
    friend void Y::f() ;

    // アクセス指定は、friend宣言自体に影響を及ぼさない
    // 以下3行のfriend宣言に、アクセス指定は何の意味もなさない
private :
    friend void f() ;
```

```

protected :
    friend void g() ;
public :
    friend void h() ;
} ;

```

`Y::f`は`private`メンバーなので、`X`からはアクセスできない。`X`の`friend`宣言は、関数`f`, `g`, `h`を、`X`の`friend`として宣言しているが、この宣言に、`X`のアクセス指定は何の効果も与えない。

`friend`宣言は、実は関数を定義することができる。

```

class X
{
    friend void f() { } // 関数の定義
} ;

```

`friend`宣言で定義された関数は、クラスが定義されている名前空間スコープの関数になる。クラスのメンバー関数にはならない。ただし、`friend`宣言で定義された関数は、ADLを使わなければ、呼び出すことはできない。非修飾名前探索や、修飾名前探索で、関数名を参照する方法はない。

```

// グローバル名前空間のスコープ

class X
{
    // fはメンバー関数ではない
    // クラスXの定義されているグローバル名前空間のスコープ内の関数
    friend void f( X ) { }
    // gはメンバー関数ではない
    // gを呼び出す方法は存在しない
    friend void g() { }
} ;

int main()
{
    X x ;
    f(x) ; // OK、ADLによる名前探索

    (f)(x) ; // エラー、括弧がADLを阻害する。ADLが働くないので名
    前fが見つからない
    ::f(x) ; // エラー、名前fが見つからない
}

```

```
    g() ; // エラー、名前gが見つからない  
}
```

このように、通常の名前探索では関数名が見つからないという問題があるため、`friend`宣言内での関数定義は、行うべきではない。

`friend`によって宣言された関数は、前方宣言されていない場合、外部リンクエージを持つ。前方宣言されている場合、リンクエージは前方宣言に従う。

```
inline void g() ; // 前方宣言、関数gは内部リンクエージを持つ  
  
class X  
{  
    friend void f() ; // 関数fは外部リンクエージを持つ  
    friend void g() ; // 関数gは内部リンクエージを持つ  
} ;  
  
// 定義  
void f() { } // 外部リンクエージ  
inline void g() { }
```

`friend`宣言は、派生されることはない。また、あるクラスの`friend`の`friend`は、あるクラスの`friend`ではない。つまり、友達の友達は、友達ではない。

```
class A  
{  
private :  
    typedef int type ;  
    friend class B ;  
} ;  
  
class B  
{  
    // OK、BはAのfriend  
    typedef A::type type ;  
  
    friend class C ;  
} ;  
  
class C  
{  
    // エラー、BはAのfriendである。CはBのfriendである。
```

```

// Cは、Aからみて、friendのfriendにあたる。
// しかし、CはAのfriendではない。
typedef A::type type ;
} ;

class D : public B
{
    // エラー、DはBから派生している。BはAのfriendである。
    // しかし、DはAのfriendではない
    typedef A::type type ;
} ;

```

ローカルクラスの中でfriend宣言で、非修飾名を使った場合、名前探索において、ローカルクラスの定義されている関数外のスコープは考慮されない。friend関数を宣言する場合、対象の関数はfriend宣言に先立って宣言されていなければならない。friendクラスを宣言する場合、クラス名はローカルクラスの名前であると解釈される。

```

class A ; // ::A
void B() ; // ::B

void f()
{
    // 関数の前方宣言は関数内でも可能
    void C( void ) ; // 定義は別の場所

    class Y ; // ローカルクラスYの宣言

    // ローカルクラスXの定義
    class X
    {
        friend class A ; // OK、ただし、::Aではなく、ローカルクラスのA
        friend class ::A ; // OK、::A
        friend class Y ; // OK、ただしローカルクラスY

        friend void B() ; // エラー、Bは宣言されていない。::Bは考慮されない
        friend void C() ; // OK、関数内の前方宣言により名前を発見
    } ;
}

```

friend宣言とテンプレートの組み合わせについては、14.4.4 テンプレート宣言のfriendを参照。

11.4 protectedメンバーアクセス(Protected member access)

protectedメンバーは、friendか、メンバーの宣言されたクラスか、クラスから派生しているクラスからアクセスすることができる。

```
void g() ;  
  
class Base  
{  
    friend void g() ;  
  
protected :  
    int member ;  
} ;  
  
void f()  
{  
    Base base ;  
    base.member ; // エラー、protectedメンバー  
}  
  
void g()  
{  
    Base base ;  
    base.member ; // OK、gはBaseのfriend  
}  
  
class Derived : protected Base  
{  
    void f()  
    {  
        member ; // OK、DerivedはBaseから派生している  
    }  
} ;
```

11.5 virtual関数へのアクセス (Access to virtual functions)

virtual関数へのアクセスは、virtual関数の宣言によって決定される。virtual関数のオーバーライドには影響されない。

```
class Base
{
public :
    virtual void f() { }
} ;

class Derived : public Base
{
private :
    void f() { } // Base::fをオーバーライド
} ;

int main()
{
    Derived d ;
    d.f() ; // エラー、Derived::fはprivateメンバー

    Base & ref = d ;
    ref.f() ; // OK、Derived::fを呼ぶ
}
```

Derived::fはprivateメンバーなので、関数mainから呼び出すことはできない。しかし、Base::fはpublicメンバーである。Base::fはvirtual関数なので、呼び出す関数は、実行時のオブジェクトの型によって決定される。この時、オーバーライドしたvirtual関数のアクセス指定は、考慮されない。Base::fのアクセス指定のみが考慮される。この例では、関数mainから、Derived::fを直接呼び出すことはできないが、Baseへのリファレンスやポインターを経由すれば、呼び出すことができる。

virtual関数呼び出しのアクセスチェックは、呼び出す際の式の型によって、静的に決定される。基本クラスでpublicメンバーとして宣言されているvirtual関数を、派生クラスでprotectedやprivateにしても、基本クラス経由で呼び出すことができる。

11.6 複数のアクセス (Multiple access)

多重派生によって、基本クラスのメンバーに対して、複数のアクセスパスが形成されている場合、アクセス可能なパスを経由してアクセスが許可される。

```
class Base
{
public :
    void f() { }
} ;

class D1 : private virtual Base { } ;
class D2 : public virtual Base { } ;

class Derived : public D1, public D2
{
    void f()
    {
        Base::f() ; // OK、D2を経由してアクセスする
    }
} ;
```

D1はBaseをprivate派生しているので、DerivedからD1経由では、Baseにアクセスできない。しかし、D2経由でアクセスできる。

11.7 ネストされたクラス (Nested classes)

ネストされたクラスも、クラスのメンバーであるので、他のメンバーとアクセス権限を持つ。

```
class Outer
{
private :
    typedef int type ; // privateメンバー

    class Inner
    {
        Outer::type data ; // OK、InnerはOuterのメンバー
    } ;
} ;
```

OuterにネストされたクラスInnerは、Outerのメンバーなので、Outerのprivateメンバーにアクセスすることができる。

ただし、ネストされたクラスをメンバーとして持つクラスは、ネストされたクラスに対して、特別なアクセス権限は持たない。

```
class Outer
{
    class Inner
    {
        private :
            typedef int type ; // privateメンバー
    } ;

    void f()
    {
        Inner::type x ; // エラー、Inner::typeはprivateメンバ
    }
}
```

この例では、Outerは、Innerのprivateメンバーにはアクセスできない。

12 特別なメンバー関数(Special member functions)

クラスのメンバー関数の中でも、特別な扱いを受けるメンバー関数が存在する。デフォルトコンストラクター、コピーコンストラクター、コピーダイアクリティカル演算子、ムーブコンストラクター、ムーブ代入演算子、デストラクターは、特別なメンバー関数(special member functions)である。

これらのメンバー関数を明示的に定義しない場合、暗黙のメンバー関数が生成される。暗黙に生成された特別なメンバー関数は、明示的に使用することもできる。

```
#include <utility>

struct X { } ;

int main()
{
    X x1 ; // デフォルトコンストラクター
    X x2( x1 ) ; // コピーコンストラクター
    X x3( std::move(x1) ) ; // ムーブコンストラクター

    x2 = x1 ; // コピーダイアクリティカル演算子
```

```

x2 = std::move( x1 ) ; // ムーブ代入演算子

x2.operator=( x1 ) ; // メンバー関数の明示的な使用、x2 = x1
                     と同等
}

```

クラスXは、特別なメンバー関数を一切定義していない。しかし、特別なメンバー関数は、暗黙のうちに生成されるので、クラスXのオブジェクトを初期化、破棄できるし、コピーやムーブもできる。

ユーザー定義されていない特別なメンバー関数は、条件次第で、暗黙にdelete定義されることもある。そのようなクラスの特別なメンバー関数を使いたい場合は、ユーザー定義しなければならない。

```

struct X
{
    int & member ;
// デフォルトコンストラクターは暗黙にdelete定義される
// X() = delete ; が暗黙に宣言される
} ;

int data ; // グローバル変数

struct Y
{
    int & member ;
    Y() : member(data) { }
} ;

int main()
{
    X x ; // エラー、デフォルトコンストラクターはdelete定義されている
    Y y ; // OK、ユーザー定義のデフォルトコンストラクターがある
}

```

特別なメンバー関数を明示的に宣言、定義することで、クラスをどのように生成、破棄、コピー、ムーブするかを記述できる。また、これらの操作を、明示的に禁止することもできる。

```

struct X
{

```

```
// コピー、ムーブコンストラクターをdelete定義
X( X const & ) = delete ;
X( X && ) = delete ;

// コピー、ムーブ代入演算子をdelete定義
X & operator = ( X const & ) = delete ;
X & operator = ( X && ) = delete ;
} ;
```

この例では、コピーもムーブもできないクラスを定義している。

特別なメンバー関数も、アクセス指定の影響を受ける。

```
class X
{
public :
    X( int ) { }
private :
    X( double ) { }
} ;

int main()
{
    X a( 0 ) ; // OK、X::X(int)はpublicメンバー
    X b( 0.0 ) ; // エラー、X::X(double)はprivateメンバー
}
```

この例では、X::X(int)はpublicメンバーなので、main関数からアクセスできるが、X::X(double)は、privateメンバーなので、main関数からアクセスできない。その結果、変数bの定義はエラーとなる。

12.1 コンストラクター(Constructor)

コンストラクターは名前を持たない。コンストラクターの宣言には、特別な文法が用いられる。

関数指定子もしくはconstexpr クラス名 仮引数リスト

```
struct X
{
```

```
X() ; // コンストラクターの宣言
} ;

X::X() { } // コンストラクターの定義
```

コンストラクターは、クラス型のオブジェクトを初期化するのに用いられる。

コンストラクターのクラス名に、`typedef`名を用いることはできない。

コンストラクターに、`virtual`指定子、`static`指定子を指定することはできない。要約すれば、コンストラクターに使用可能な指定子は、`inline`、`explicit`、`constexpr`である。コンストラクターをCV修飾することはできない。ただし、コンストラクターは、CV修飾されたオブジェクトの初期化に対しても、呼び出される。オブジェクトのCV修飾子は、構築中のオブジェクトには適用されないからである。コンストラクターをリファレンス修飾することはできない。

```
struct X {
    virtual X() ; // エラー、コンストラクターにvirtual指定子は使えない
    static X() ; // エラー、コンストラクターにstatic指定子は使えない
    X() const ; // エラー、コンストラクターはCV修飾できない
    X() & ; // エラー、コンストラクターはリファレンス修飾できない
} ;
```

デフォルトコンストラクター

実引数なしで呼べるコンストラクターを、デフォルトコンストラクター(`default constructor`)という。これには、仮引数を取らないコンストラクターの他に、仮引数にすべてデフォルト実引数が指定されているコンストラクターも含まれる。

以下はすべて、`X`に対するデフォルトコンストラクターの宣言である。

```
X() ;
X( int = 0 ) ;
X( int = 0, int = 0 ) ;
```

もし、ユーザー定義のコンストラクターが存在しない場合、暗黙のデフォルトコンストラクターが、デフォルト化されて宣言される。

```
struct X
{
```

```
// デフォルトコンストラクターの定義なし
// 暗黙に、X() = default ; が宣言される
} ;

struct Y
{
    Y(int) ; // ユーザー定義のコンストラクター
// 暗黙のデフォルトコンストラクターはdefault化されない
} ;
```

ただし、以下のいずれかの条件をみたすクラスの場合、暗黙のデフォルトコンストラクターは`delete`定義される。

9.9.2 [unionのようなクラス](#)で、共用メンバーが非トリビアルデフォルトコンストラクターを持つ場合

```
// 非トリビアルデフォルトコンストラクターを持つクラス
struct NonTrivial
{
    NonTrivial() { } // 非トリビアルデフォルトコンストラクター
} ;

// 非トリビアルデフォルトコンストラクターを持つメンバーのあるunion
union X
{
// 暗黙のデフォルトコンストラクターはdelete定義される
    NonTrivial nt ;
} ;

// そのような無名unionを直接のメンバーに持つクラス
struct Y
{
// 暗黙のデフォルトコンストラクターはdelete定義される
    union { NonTrivial nt ; } ;
} ;
```

初期化子のないリファレンス型の非staticデータメンバーを持つクラスの場合

```
int OBJECT ; // グローバル変数

struct X
{
```

```
// 初期化子のないリファレンス型の非staticデータメンバー
    int & ref ;
} ;

struct Y
{
// 初期化子がある
    int & ref = OBJECT ;
} ;

struct Z
{
    int & ref ;
// ユーザー定義のコンストラクターがある
    Z() : ref( OBJECT ) { }
} ;

int main()
{
    X x ; // エラー、デフォルトコンストラクターがdelete定義される
    Y y ; // OK
    Z z ; // OK
}
```

クラスXのデフォルトコンストラクターは暗黙にdelete定義される。Y::refには初期化子がある。Zにはユーザー定義のコンストラクターがある。

unionのメンバーではない、const修飾された型があるいはその配列型の、非staticデータメンバーが、ユーザー定義デフォルトコンストラクターを持たず、初期化子もない場合。

```
// ユーザー定義デフォルトコンストラクターを持たないクラス
struct NoUserDefined { } ;

struct X
{
// 初期化子がない
    NoUserDefined const member ;
    NoUserDefined const array[1] ;
} ;

struct Y
{
```

```

// 初期化子がある
    NoUserDefined const member = NoUserDefined() ;
} ;

struct Z
{
    NoUserDefined const member ;
// memberに対する初期化子がないので不可
// Z() : member() { } なら可
    Z() { }
} ;

int main()
{
    X x ; // エラー
    Y y ; // OK
    Z z ; // エラー
}

```

気をつける点としては、Zのユーザー定義デフォルトコンストラクターが、Z(){}という形の場合、Z::memberに対する初期化子がないので、エラーになる。Z() : member() {}という形の場合は、初期化子があるので、エラーにはならない。

9.9.2 [unionのようなクラス](#)で、共用メンバーがconst修飾されている場合。これには、const修飾されている型への配列型も含む。

```

union X
{ // すべてconst修飾されている
    int const a ;
    int const b ;
    int const c[1] ;
} ;

struct Y
{ // すべてconst修飾されている無名unionをメンバーに持つ
    X x ;
} ;

union Z
{
    int a ; // const修飾されていない
    int const b ;
}

```

```

        int const c[1] ;
} ;

int main()
{
    X x ; // エラー
    Y y ; // エラー
    Z z ; // OK、Y::aはconst修飾されていない。
}

```

unionのすべての非staticデータメンバーがconst修飾されている場合のみ、デフォルトコンストラクターが暗黙にdelete定義される。ひとつでもconst修飾されていない非staticデータメンバーがある場合、この条件には当てはまらない。

直接の基本クラス、仮想基本クラス、初期化子のない非staticデータメンバーの型が、デフォルトコンストラクターが使えない型である場合。これには、配列型も含まれる。

```

// デフォルトコンストラクターが使えないクラスの例
struct X
{
    X() = delete ;
} ;

// 直接の基本クラス
struct A : X { } ;
// 仮想基本クラス
struct B : virtual X { } ;
// 初期化子のない非staticデータメンバー
struct C
{
    X a ;
    X b[1] ;
} ;

```

クラスA、B、Cは、いずれもデフォルトコンストラクターが暗黙にdelete定義される。

ある型のデフォルトコンストラクターが使えない場合というのは、以下の通りである。

デフォルトコンストラクターがない場合。

```

// ユーザー定義コンストラクターがある場合、暗黙のデフォルトコンストラクターは定義されない。

```

```
struct NoDefaultConstructor { NoDefaultConstructor(int) ; } ;
struct X : NoDefaultConstructor { } ;
```

デフォルトコンストラクターのオーバーロード解決の結果が曖昧になる場合。

```
struct Ambiguous
{
    Ambiguous( int = 0 ) { }
    Ambiguous( double = 0.0 ) { }
} ;

struct X : Ambiguous { } ;

int main()
{
    Ambiguous a ; // エラー、デフォルトコンストラクターのオーバーロード解決が曖昧
    X b ; // エラー、デフォルトコンストラクターは暗黙にdelete定義されている
}
```

デフォルトコンストラクターがdelete定義されている場合

```
struct X
{
    X() = delete ; // デフォルトコンストラクターのdelete定義
} ;
```

クラスのデフォルト化されたデフォルトコンストラクターから、ある型のデフォルトコンストラクターにアクセス出来ない場合。

```
class B1
{
private :
    B1() = default ;
} ;

class B2
{
private :
```

```

B2() { }

} ;

class D1 : public B1 { } ;
class D2 : public B2 { } ;

int main()
{
    D1 a ; // エラー、デフォルトコンストラクターは暗黙にdelete定義されている
    D2 b ; // エラー、デフォルトコンストラクターは暗黙にdelete定義されている
}

```

クラスB1、B2のデフォルトコンストラクターは、`private`メンバーなので、`friend`ではないクラスD1、D2からはアクセスできない。そのため、デフォルトコンストラクターは暗黙に`delete`定義される。

これらの条件に当てはまらない場合、デフォルトコンストラクターは暗黙に`default`化される。

デフォルトコンストラクターがトリビアル(trivial)となるためには、以下の条件をすべて満たさなければならない。

デフォルトコンストラクターが`ユーザー定義`も`delete定義`もされていない。クラスは`virtual`関数と`virtual`基本クラスを持たない。クラスの非`static`データメンバーは、初期化子を持たない。クラスの直接の基本クラスは、トリビアルデフォルトコンストラクターを持つ。クラスの非`static`データメンバーは、トリビアルデフォルトコンストラクターを持つ。

デフォルトコンストラクターは、使われたときに、暗黙に`default`化される。デフォルトコンストラクターが`delete`定義されずに、`default`化された場合、暗黙に定義される。この暗黙のデフォルトコンストラクターは、コンストラクター初期化子を書かずコンストラクターの本体を空にした、`ユーザー定義`のデフォルトコンストラクターと同等である。

```

struct X
{
    // 暗黙のデフォルトコンストラクターは、以下のコードと同じ
    // X() {}
};

struct Y { } ;

int main()
{
    X x ; // 暗黙のデフォルトコンストラクターを使う
}

```

クラスXはデフォルトコンストラクターが使われているので、暗黙にdefault化される。クラスYのデフォルトコンストラクターは使われていないので、定義されない。

もし、暗黙のデフォルトコンストラクターが定義されていて、同等のユーザー定義のデフォルトコンストラクターを書いた場合にエラーとなる場合は、プログラムもエラーとなる。

```
struct X { int & ref ; } ;
struct Y { int & ref ; } ;

int main()
{
    X x ; // エラー
    int obj = 0 ;
    Y y = { obj } ; // OK
}
```

クラスXはデフォルトコンストラクターを使っているので、暗黙のデフォルトコンストラクターがdefault化される。しかし、リファレンスの非staticデータメンバーを持つことにより、同等のユーザー定義のデフォルトコンストラクターがエラーになるので、エラーとなる。一方、クラスYでは、デフォルトコンストラクターが使われていない。

同等のユーザー定義のデフォルトコンストラクターがconstexprコンストラクターの要求を満たす場合、暗黙のデフォルトコンストラクターも、constexprコンストラクターになる。

デフォルトコンストラクターは、初期化子なしで定義されたオブジェクトや、関数形式の明示的なキャストに対して呼ばれる。

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

X x ; // デフォルトコンストラクターが呼ばれる

int main()
{
    X x ; // デフォルトコンストラクターが呼ばれる
    new X ; // デフォルトコンストラクターが呼ばれる
    X() ; // デフォルトコンストラクターが呼ばれる
}
```

クラスのオブジェクトをコピー、ムーブする際には、コピー、ムーブコンストラクターがそれぞれ使われる。詳しくは、12.8 [クラスオブジェクトのコピーとムーブ](#)を参照。

基本クラスと非staticデータメンバーのコンストラクターが呼ばれる順番や、実引数の渡し方については、12.6.2 [基本クラスとデータメンバーの初期化](#)を参照。

その他のコンストラクターについては、12.3.1 型変換コンストラクターを参照。

コンストラクターに戻り値の型を指定することはできない。コンストラクターの本体の中でreturn文を使う場合は、値を指定してはならない。コンストラクターのアドレスを取得することはできない。

constなオブジェクトの構築中に、コンストラクターのthisを、直接、間接的に経由しないglvalueによってオブジェクト、またはそのサブオブジェクトにアクセスした場合、値は未規定である。この制限には、通常、まず遭遇することはない。例えば、以下のようなコードが問題になる。

```
struct C ;

void f( C * ) ;

struct C
{
    int value ;
    C() : c(123)
    { // オブジェクトは、まだ構築中
        f( this ) ; // オブジェクトの構築中に呼び出す
    }
} ;

const C cobj ; // staticストレージ上のconstなオブジェクト

void f( C* cptra )
{
    cptra->value ; // OK、値は123。cptraはコンストラクターのthis
由来
    cobj.value ; // 値は未規定
}
```

オブジェクトは、コンストラクターを実行し終わった時点で、構築済みとなる。コンストラクターを実行中ということは、まだオブジェクトは構築中ということである。cobjは、デフォルトコンストラクターを呼び出す。したがって、関数fは、cobjの構築中に呼び出されるということになる。cptraの値は、コンストラクターのthisによって得られたアドレスである。したがって、cptraの値である、Cのオブジェクトへのアドレスを参照して、cobjにアクセスすることはできる。関数fは、クラスCのコンストラクターから呼び出されている。関数fの中からcobjを直接参照することは、cobjの構築中に、コンストラクターのthisによらずにアクセスするということである。この場合、値は未規定となる。この例では、123であるとは保証されない。

この条件に当てはまるようなコードは、現実には極めて珍しい。

12.2 一時オブジェクト(Temporary objects)

一時オブジェクト(temporary object)は、様々な場面で、自動的に生成、破棄される。例えば、prvalueをリファレンスに束縛する、prvalueを返す、prvalueを生成する型変換、例外のthrow、ハンドラーでキャッチ、初期化などである。例外における一時オブジェクトの寿命は、15 [例外](#)を参照。

```
struct X { } ;

X f()
{
    return X() ; // prvalueを生成する型変換
}

int main()
{
    int && ref = 0 ; // prvalueをリファレンスに束縛する
    f() ; // prvalueを返す
}
```

実装は一時オブジェクトの生成を省略できる。例えば、以下のコードについて考える。

```
struct X
{
    X( int ) ; // コンストラクター
    X( X const & ) ; // コピーコンストラクター
    X & operator = ( X const & ) ; // コピーデリケタ
    ~X() ; // デストラクター
} ;

struct Y
{
    Y( int ) ; // コンストラクター
    Y( Y && ) ; // ムーブコンストラクター
    ~Y() ; // デストラクター
};

X f( X ) ;
Y g( Y ) ;

int main()
{
```

```

X a = f( X(2) ) ; // #1
Y b = g( Y(3) ) ; // #2
X c(1) ;
c = f(c) ; // #3
}

```

#1について考える。ある実装では、X(2)という式で一時オブジェクトがひとつ作られ、関数の実引数として渡す際に、別の一時オブジェクトがひとつ作られてコピーされるかもしれない。関数の戻り値も、別の一時オブジェクトがひとつ作られて、変数aにコピーされるかもしれない。別の実装では、X(2)という式による一時オブジェクトは、関数の実引数の一時オブジェクト上に直接構築されるので、一時オブジェクトを省略できるかもしれない。関数の戻り値も、変数aのオブジェクト上に直接構築されるので、一時オブジェクトを省略できるかもしれない。

#2も、コピーがムーブに、変数aがbに変わっただけで、同じことが言える。

#3では、変数cは、すでに構築されたオブジェクトなので、関数の戻り値の一時オブジェクトを、変数cのオブジェクトの上に、直接構築することはできない。ここでは一時オブジェクトが構築され、変数cにコピーされる。

一時オブジェクトの構築が省略されたとしても、もし一時オブジェクトを作成していればエラーになるようなコードは、エラーになる。たとえば、コンストラクターやデストラクターにアクセスできない場合だ。

一般に、ある式において一時オブジェクトがいくつ構築されるかということは、規格では定義されていない。実装次第である。一時オブジェクトが構築されなければ、コンストラクターやデストラクターも呼ばれない。

逆に、一時オブジェクトが構築される場合、非トリビアルなコンストラクターは必ず呼び出されるし、破棄するときには、非トリビアルなデストラクターは必ず呼び出される。

一時オブジェクトの破棄は原則として、その一時オブジェクトを構築することになった式を含む1.6 完全式の評価の最後の段階として、実行される。言いかえれば、一時オブジェクトの寿命は、完全式が評価され終わるまでということもできる。

```

struct X
{
    X operator + ( X const & ) { return X() ; }
} ;

int main()
{
    X a = 0 ;
    X b = a + a + a ;
}

```

この例で、bの初期化子の中の式において構築された一時オブジェクトがもしあれば、その寿命は、ソースコード上で言えば、セミコロンまでとなる。

ただし、この原則に従わない場合が、ふたつ存在する。

ひとつは、配列の要素を初期化する際に、デフォルトコンストラクターがデフォルト実引数を持っていた場合、ある要素のデフォルトコンストラクター実行における一時オブジェクトの破棄は、次の要素の初期化の前に行われる。破棄に伴うあらゆるサイドエフェクトは、次の要素の初期化の以前にシーケンス(sequenced before)される。

```
struct Fat { char [1000] ; } ;

struct X
{
    X( Fat f = Fat() ) { } // デフォルトコンストラクター
} ;

int main()
{
    X a[1000] ;
}
```

この例では、a[0]からa[999]までの1000個のX型の配列の要素に対し、デフォルトコンストラクターが呼び出される。もし、a[0]がa[1]の前に初期化された場合、a[0]のデフォルトコンストラクター呼び出しによって構築されたFat型の一時オブジェクトは、a[1]を初期化するときには、すでに破棄されている。配列のすべての要素を初期化し終わるまで、1000個のFat型の一時オブジェクトが保持されることはない。

もうひとつは、一時オブジェクトをリファレンスに束縛した場合、一時オブジェクトの寿命は、リファレンスの寿命まで延長される。一時オブジェクトは、rvalueリファレンスか、constなlvalueリファレンスで束縛できる。

```
struct X { } ;

int main()
{
    {
        X const & lvalue_reference = X() ;
        X && rvalue_reference = X() ;
        // 一時オブジェクトの寿命はここまで
    }
}
```

ただし、このリファレンス束縛の寿命の延長には、いくつかの例外が存在する。

コンストラクター初期化子によってリファレンスのメンバーに束縛された一時オブジェクトの寿命は、コンストラクター呼び出しが終了するまでである。

```
struct Member { } ;

struct X
{
    Member const & ref ;
    X( Member const & ref ) : ref(ref)
    {
        // refは妥当なオブジェクトを参照している
    }
} ;

int main()
{
    X x = Member() ;
    // 一時オブジェクトが破棄される
    // x.refは無効なオブジェクトを参照している
}
```

もし、クラスXのコンストラクターの実引数が、一時オブジェクトへのリファレンスだった場合、一時オブジェクトの寿命は、コンストラクター呼び出しが終了するまでである。そのため、初期化の終わった変数xのメンバーrefは、無効なオブジェクトを参照することになる。

仮引数のリファレンスに束縛された一時オブジェクトの寿命は、関数呼び出しを含む式の完全式の評価が終了するまでである。

```
struct X { } ;

void f( X const & ref )
{
    // refは妥当なオブジェクトを参照している
}

int main()
{
    f( X() ) ;
    // 一時オブジェクトが破棄される
}
```

この例で、X()を含む完全式というのは、関数fに対する関数呼び出し式のオペランドである。したがって、完全式はX()となる。しかし、この解釈に従うと、関数の本体では、refは無効なオブジェクトを参照することになってしまう。そのため、仮引数のリファレンス束縛に対しては、関数呼び出しを含む完全式になる。この場合、f(X())である。そのため、X()によって構築された一時オブジェクトは、関数fの本体の中でも妥当である。

関数のreturn文によって構築された、関数の戻り値のリファレンスに束縛された一時オブジェクトの寿命は、延長されない。一時オブジェクトは、return文を含む完全式の終了をもって、破棄される。

```
struct X { } ;

X const & f( X const & ref )
{
    return X() ;
}

int main()
{
    X const & ref = f( X() ) ;
    // 一時オブジェクトは破棄される
    // refは無効なオブジェクトを参照している
}
```

このように、関数の戻り値としてのリファレンスに束縛されても、一時オブジェクトの寿命は延長されない。これには注意が必要である。

new初期化子の中でリファレンス束縛された一時オブジェクトの寿命は、new初期化子を含む完全式の終わりまでである。

```
struct X { int const & ref ; } ;

int main()
{
    X * ptr = new X{ 0 } ;
    // 一時オブジェクトが破棄される
    // ptr->refは無効なオブジェクトを参照している
}
```

リファレンス束縛により、寿命の延長を受けていない一時オブジェクトの破棄の順番は、構築の逆順に行われる。後に構築された一時オブジェクトの方が、先に構築された一時オブジェクトより、先に破棄される。

もし、リファレンス束縛を受けた、複数の一時オブジェクトが、同じ場所で破棄される場

合、破棄の順番は、構築の逆順に行われる。

```
struct X { X( int ) { } } ;
void f( X const &, X const & ) { }

int main()
{
    f( X(1), X(2) ) ; // #1
}
```

今、#1のX(1)、X(2)という式に対して、それぞれ一時オブジェクトが構築されたとする。関数の実引数の評価順序は規定されていないので、どちらが先に構築されるかは、規格の定めるところではない。しかし、仮にX(1)の一時オブジェクトが、X(2)に先んじて構築された場合、オブジェクトの破棄は、X(2)が先になる。

12.3 型変換(Conversions)

クラスの型変換を実現するには、方法がふたつある。コンストラクターと変換関数(conversion function)だ。このふたつを合わせて、ユーザー定義型変換(user-defined conversions)という。ユーザー定義型変換は、暗黙の型変換、初期化、明示的な型変換に用いられる。

ひとつの値に対して、ユーザー定義型変換は1回しか適用されない。

```
struct X { } ;
struct Y
{
    Y( int ){ } // int型からY型へ
    operator X() { return X(); } // Y型からX型へ
} ;

int main()
{
    Y a = 0 ; // OK、int型からY型への型変換
    X b = Y(0) ; // OK、Y型からX型への型変換
    X c = 0 ; // エラー
}
```

ユーザー定義型変換によって、int型からY型に変換することはできる。また、Y型からX型に変換することはできる。ただし、int型から、暗黙にX型に変換することはできない。なぜならば、それにはユーザー定義型変換を、2回適用しなければならないからだ。

ユーザ一定義型変換は、曖昧にならない場合のみ、暗黙に使われる。派生クラスの型変換関数は、基本クラスの型変換関数を隠さない。ただし、同じ型に対する型変換関数の場合を除く。複数の型変換関数がある場合、関数のオーバーロード解決と同じ方法で、最適な関数が解決される。

```
struct Base
{
    operator int() { return 0 ; }
} ;
struct Derived : Base
{
    // Base::operator intを隠さない
    operator char() { return char(0) ; }
} ;

int main()
{
    Derived obj ;
    int a = obj ; // OK
    char b = obj ; // OK
    bool c = obj ; // エラー、曖昧
}
```

12.3.1 コンストラクターによる型変換(Conversion by constructor)

`explicit`指定子を使わずに宣言されているコンストラクターは、仮引数からクラスへの型変換の方法を指定するメンバー関数である。このような関数を、型変換コンストラクター(*converting constructor*)という。

```
struct X
{
    X( int ) {} // int型からの型変換を提供
    X( double ) {} // double型からの型変換を提供
    X( int, int, int ) ; // 3個のint型からの型変換を提供
} ;

int main()
{
    X a = 0 ; // int型からの型変換
    X b(0) ; // int型からの型変換
```

```
X c = 0.0 ; // double型からの型変換
X d( 1, 2, 3 ) ;
}
```

型変換コンストラクターは、仮引数からクラス型への変換方法を指定する。仮引数は、複数でもよい。

explicit指定子のあるコンストラクターは、explicitのないコンストラクターとほぼ同じである。ただし、explicitコンストラクターは、8.5 [直接初期化](#)か、キャストが明示的に使われたときにしか、使われない。

```
struct X
{
    explicit X( int ) {} // explicitコンストラクター
} ;

void f( X ) { }

int main()
{
    X a = 0 ; // エラー
    X b(0) ; // OK、直接初期化
    X c = X(0) ; // OK、明示的なキャスト
    X d = static_cast<X>(0) ; // OK、明示的なキャスト

    f( 0 ) ; // エラー
    f( static_cast<X>(0) ) ; // OK
}
```

デフォルトコンストラクター、コピーコンストラクター、ムーブコンストラクターにも、explicitを指定できる。これらの関数も、explicit指定子を指定しない場合、暗黙に使われる。

```
struct X
{
    X() { }
    explicit X( X const & ) {} // explicitコピーコンストラクタ
}

int main()
{
```

```
X a ;
X b = a ; // エラー、コピー初期化（代入式とは違うことに注意）
X c( a ) ; // OK、直接初期化
}
```

デフォルトコンストラクターに対するexplicit指定子の有無は、以下のコード例のような違いをもたらす。

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

struct Y
{
    explicit Y() { }
} ;

int main( )
{
    X x( {} ) ; // OK、非explicitデフォルトコンストラクター
    Y y( {} ) ; // エラー、explicitデフォルトコンストラクター
}
```

実用上、気になるほどの違いはない。

12.3.2 型変換関数(Conversion functions)

以下のような文法で宣言されるメンバー関数を、型変換関数(Conversion function)という。

```
explicitopt operator 型識別子 ( )
```

型変換関数は、仮引数を取らず、戻り値の型を指定しない。型変換関数は、メンバーであるクラス型から、型識別子の型への型変換を提供する。

```

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

int main()
{
    X x ;
    int i = x ; // 0
}

```

この例では、クラスXは、暗黙にint型の0に変換できるクラスとなる。

型変換関数の型は、「仮引数を取らず、型識別子の型を返す、メンバー関数」になる。

```

struct X
{
    operator int() const { return 0 ; }
} ;

int main()
{
    // 型はint (X::*)(void) const
    int (X::*ptr)(void) const = &X::operator int ; // ポイン
    ターを得る
    X x ;
    (x.*ptr)() ; //呼び出す
}

```

型識別子が、自分自身のクラス型(リファレンスも含む)、自分自身の基本クラス型(リファレンスも含む)、void型、またこれらの型にCV修飾子を付けた型の場合、型変換関数が使われることはない。これらの型変換には、標準型変換が用いられる。型変換関数は使われない。これらの型変換関数を宣言することはエラーではないが、使われることはない。

```

struct Base{ } ;

struct Derived : Base
{
    // これらの型変換関数は、使われることはない
    operator Derived () ; // 自分自身のクラス型
    operator Derived & () ; // 自分自身のクラスへのリファレンス
}

```

型

```

operator Derived const & () ; // 自分自身のクラスへのconst
リファレンス型
operator Base () ; // 基本クラス型
operator void () ; // void型
} ;

int main()
{
    Derived d ;
    Base b = d ; // 標準型変換が使われる。型変換関数は使われない
}

```

型変換関数にexplicit指定子が指定されている場合、型変換関数は、直接初期化や明示的なキャストが使われなければ、呼び出されない。

```

struct A { } ;
struct B { } ;

struct X
{
    operator A() { return A() ; }
    explicit operator B() { return B() ; }

} ;

int main()
{
    X x ;

    A a1 = x ; // OK
    A a2(x) ; // OK
    A a3 = A(x) ; // OK

    B b1 = x ; // エラー、コピー初期化
    B b2(x) ; // OK、直接初期化
    B b3 = B(x) ; // OK、明示的なキャスト
}

```

型変換関数の型識別子を、関数型と配列型にすることはできない。

```
struct X
{
    operator void (void) () ; // エラー、関数型
    operator int[1] () ; // エラー、配列型
} ;
```

その他の型には、特に制限はない。

```
struct Y { } ;

struct X
{
    using pointer_type = void (*) (void) ;
    using reference_type = int (&)[1] ;

    operator pointer_type () ; // 関数ポインター型
    operator reference_type () ; // 配列への参照型

    operator Y() ; // 他のクラス型
} ;
```

型変換関数は継承される。

```
struct Base
{
    operator int() { return 0 ; }
} ;

struct Derived : Base { } ;

int main()
{
    Derived d ;
    int i = d ; // Base::operator intを呼び出す
}
```

型変換関数はvirtual関数にできる。

```
struct Base
```

```
{
    // ピュアvirtual関数
    virtual operator int() = 0 ;
} ;

struct Derived : Base
{
    // オーバーライド
    virtual operator int() { return 0 ; }
} ;
```

型変換関数はstatic関数にはできない。

```
struct X
{
    static operator int() ; // エラー
} ;
```

12.4 デストラクター(Destructors)

以下のような文法の宣言を、デストラクター(destructor)という。

関数指定子_{opt} ~ クラス名 ()

デストラクターの宣言は、~(チルダ)に続いて、クラス名、空の引数リストを指定する。関数指定子には、`inline`と`virtual`を指定できる。クラス名の代わりに、`typedef`名を使用することはできない。

```
struct X
{
    ~X() ; // デストラクターの宣言
} ;
```

デストラクターはクラス型のオブジェクトを破棄する際に使われる。デストラクターには、仮引数や戻り値の型を指定することはできない。デストラクターのアドレスを得ることはできない。デストラクターはstaticメンバーにはなれない。デストラクターは、CV修飾、リファレンス修飾できない。ただし、デストラクターは、CV修飾されたクラス型のオブジェ

クトに対しても呼び出される。

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

int main()
{
{
    X x ;
    // オブジェクト破棄、デストラクターが呼ばれる
}

X * ptr = new X ;
delete ptr ; // オブジェクト破棄、デストラクターが呼ばれる
}
```

デストラクターの宣言に例外指定がない場合は、暗黙のデストラクターと同等の例外指定が、暗黙に指定される。詳しくは、[例外指定](#)を参照。

クラスにユーザー宣言されたデストラクターがない場合、デストラクターは暗黙にdefault化されて宣言される。暗黙に宣言されたデストラクターは、クラスのinline publicメンバーである。

暗黙のデストラクターは、以下のいずれかの条件を満たしたとき、`delete`定義される。

9.9.2 [unionのようなクラス](#)で、共用メンバーが、非トリビアルデストラクターを持つ場合。

```
struct Trivial { } ;
struct NonTrivial { ~NonTrivial() { } } ;

// デストラクターは暗黙にdefault化される
union A1 { Trivial member ; } ;
struct A2 { union { Trivial member ; } ; } ;

// デストラクターは暗黙にdelete定義される
union B1 { NonTrivial member ; } ;
struct B2 { union { NonTrivial member ; } ; } ;

int main()
{
// OK、暗黙のデストラクターを使う
}
```

```

    A1 a1 ; A2 a2 ;
// エラー、暗黙のデストラクターはdelete定義されている
    B1 b1 ; B2 b2 ;
}

```

クラスの非staticデータメンバーのデストラクターがdelete定義されているか、デフォルトデストラクターからアクセス出来ない場合。

```

struct deleted_destructor
{
    ~deleted_destructor() = delete ;
} ;

struct inaccessible_destructor
{
private :
    ~inaccessible_destructor() ;
    friend struct Y ;
} ;

// クラスのデストラクターは暗黙にdelete定義される
struct X
{
    deleted_destructor m1 ; // デストラクターがdelete定義されている
    inaccessible_destructor m2 ; // デストラクターにアクセス出来ない
} ;

struct Y
{
    inaccessible_destructor m ; // friendなので、デストラクターにアクセス可能
} ;

```

直接の基本クラス、もしくは、virtual基基本クラスのデストラクターがdelete定義されているか、デフォルトデストラクターからアクセス出来ない場合。

```

struct Base
{
    ~Base() = delete ;
}

```

```
} ;  
  
struct D1 : Base  
{  
// D1のデストラクターはdelete定義される  
} ;
```

間接の基本クラスのデストラクターは、影響しない。

```
struct Base  
{  
private :  
    ~Base() { }  
    friend struct D1 ;  
} ;  
  
struct D1 : Base  
{  
// friend宣言により、Baseのデストラクターにアクセスできる  
} ;  
  
struct D2 : D1  
{  
// D1のデストラクターにアクセスできる  
} ;  
  
int main()  
{  
    D1 d1 ; // OK  
    D2 d2 ; // OK  
}
```

ただし、virtual基本クラスには、直接と間接の違いはないので、影響する。

```
struct Base  
{  
private :  
    ~Base() { }  
    friend struct D1 ;  
} ;
```

```

struct D1 : virtual Base
{
// friend宣言により、Baseのデストラクターにアクセスできる
} ;

struct D2 : D1
{
// virtual基本クラスのBaseのデストラクターにアクセスできない
// デストラクターは暗黙にdelete定義される
} ;

int main()
{
    D1 d1 ; // OK、暗黙のデストラクターを使う
    D2 d2 ; // エラー、デストラクターはdelete定義されている
}

```

D2からD1のデストラクターにアクセスすることはできるが、D2からvirtual基本クラスであるBaseのデストラクターにアクセス出来ないため、D2のデストラクターは暗黙にdelete定義される。

デストラクターがトリビアルとなるためには、ユーザー提供もdelete定義もされておらず、以下の条件をすべて満たす必要がある。

- デストラクターはvirtualではない。
- 直接の基本クラスのデストラクターは、すべてトリビアルである。
- 非staticデータメンバーのデストラクターは、すべてトリビアルである。

注意すべきこととしては、直接の基本クラスのデストラクターがトリビアルとなるためには、直接の基本クラスの直接の基本クラスのデストラクターもトリビアルでなければならぬ。つまり、最終的には、間接の基本クラスのデストラクターも、すべてトリビアルでなければならない。

delete定義されていない暗黙のデストラクターは、使われたときに、定義される。もしくは、明示的にdefault化されたときにも定義される。

デストラクターの呼び出しは、コンストラクター呼び出しの逆順に行われる。コンストラクター呼び出しの順番については、12.6.2 [基本クラスとデータメンバーの初期化](#)を参照。

クラスのデストラクターの本体の実行を終え、本体内の自動変数を破棄する。9.9.2 [共用メンバ](#)を除くクラスの直接のメンバーに対して、デストラクターを呼び出す。クラスの直接の基本クラスのデストラクターを呼び出す。クラスが、最上位の派生クラスならば、virtual基本クラスのデストラクターを呼び出す。

配列の要素に対するデストラクターも、コンストラクターの逆順に呼ばれる。

デストラクターは、virtual関数やピュアvirtual関数にすることができる。基本クラスのデ

ストラクターがvirtual関数である場合、派生クラスのデストラクターもvirtualになる。基本クラスのデストラクターがピュアvirtual関数の場合、派生クラスのオブジェクトを構築するためには、デストラクターを定義しなければならない。これらは、通常のvirtual関数と変わらない。

```
struct Base
{
    virtual ~Base() { } // デストラクターはvirtual関数
} ;

struct Derived : Base
{
    ~Derived() { } // デストラクターはvirtual関数
} ;

struct Abstract_base
{
    virtual ~Abstract_base() = 0 ; // デストラクターはピュア
//virtual関数
} ;
```

デストラクターをvirtual関数にする目的は、オブジェクトに動的に構築、破棄する際に、型情報を管理しなくてもいいという点にある。

```
#include <iostream>

struct B1
{
    virtual ~B1() { } // virtual関数
} ;

struct D1 : B1
{
    ~D1() { std::cout << "D1 destructor" << std::endl ; }
} ;

struct B2
{
    ~B2() {} // 非virtual関数
} ;

struct D2 : B2
```

```

{
    ~D2() { std::cout << "D2 destructor" << std::endl ; }
} ;

int main()
{
    B1 * b1_ptr = new D1 ;
    delete b1_ptr ; // 派生クラスのデストラクターが呼ばれる

    B2 * b2_ptr = new D2 ;
    delete b2_ptr ; // 派生クラスのデストラクターが呼ばれない
}

```

delete式に渡しているのは、基本クラスへのポインターである。そのため、非virtualなデストラクターでは、派生クラスのデストラクターが呼び出されない。デストラクターをvirtual関数にしておけば、このような場合にも、派生クラスのデストラクターが正しく呼び出される。

デストラクターが暗黙に呼ばれる条件は、以下の通りである。

- staticストレージ上のオブジェクトに対しては、プログラムの終了時に呼ばれる。
- threadストレージ上のオブジェクトに対しては、スレッドの終了時に呼ばれる。
- 自動ストレージ上のオブジェクトに対しては、オブジェクトを構築したブロックを抜けたときに呼ばれる。
- 一時オブジェクトに対しては、寿命が尽きたときに呼ばれる。
- new式で構築されたオブジェクトに対しては、delete式で破棄されるときに呼ばれる
- その他、例外として投げられたオブジェクトのキャッチに関連して呼ばれることがある

クラス型、もしくはクラスの配列型のオブジェクトが宣言された箇所で、クラスのデストラクタにアクセス出来ない場合は、エラーとなる。

```

class X
{
private :
    ~X() { } // privateメンバー
} ;

int main()
{
    X x ; // エラー、デストラクターにアクセスできない。
}

```

クラスがvirtualデストラクターを持つ場合、クラスには対応する解放関数が使える状態でなければならない。解放関数は、まずクラスのスコープ内で探され、見つからない場合は、グローバルスコープで探される。解放関数が見つからないか、曖昧か、delete定義されている場合、エラーとなる。これは、たとえプログラム中でdelete式を使わなくてもエラーとなる。

```

struct X
{
    virtual ~X() { } // OK、グローバルスコープのoperator delete
    // が発見される
} ;

struct Y
{
    virtual ~Y() { } // エラー、operator deleteはdelete定義さ
    // れている。
    void operator delete( void * ptr ) = delete ;
} ;

struct B1
{
    void operator delete( void * ptr ) ;
    virtual ~B1() { }
} ;

struct B2
{
    void operator delete( void * ptr ) ;
} ;

struct Derived : B1, B2
{
// エラー、曖昧
// 暗黙のデストラクターはvirtual関数
} ;

```

この規格の意図は、動的な型のオブジェクトは、常にdelete式が適用できることを保証するためである。

デストラクターは、明示的に呼び出すことができる。デストラクターを明示的に呼び出すには、メンバーアクセス演算子を使い、~に続いて、クラス型に対応する型名か、decltype指定子を使う。

```

// このコードは、あくまで明示的なデストラクター呼び出しを説明する
ための例である
// 関数fを呼び出すと、Xのデストラクターは4回呼び出されることにな
り、挙動は未定義である
struct X { } ;

void f()
{
    X x ;
    x.~X() ; // デストラクターの明示的な呼び出し（型名）
    x.~decltype(x)() ; // デストラクタの明示的な呼び出し
    (decltype指定子)
    x.X::~X() ; // 修飾名付き

    // ブロックを抜けた際に、デストラクターが暗黙に呼び出される
}

```

たとえ、自動ストレージ上のオブジェクトに対してデストラクターを明示的に呼び出したとしても、ブロックを抜けた際に、デストラクターは暗黙的に呼び出される。デストラクターを呼び出した後のオブジェクトに対して、再びデストラクターを呼び出した場合、挙動は未定義なので、上記のコードの挙動も、未定義である。

一般に、自動ストレージ上のオブジェクトに対して明示的にデストラクターを呼び出した後、そのオブジェクトに対して、通常ならば暗黙にデストラクターが呼び出される状況になっている場合、挙動は未定義である。

デストラクターの明示的な呼び出しは、まず使う必要はない。デストラクターが暗黙に呼び出されることがない場合としては、placement newによる、ユーザー指定のストレージ上へのオブジェクトの構築が挙げられる。

```

#include <new>

struct X
{
    ~X() { /*実装*/ }
} ;

int main()
{
    void * ptr = ::operator new( sizeof(X) ) ; // ストレージを
    確保
    X * x_ptr = new(ptr) X ; // ptrの指すストレージ上にX型のオ
    ブジェクトを構築
    x_ptr->~X() ; // デストラクターの明示的な呼び出し
    ::operator delete( ptr ) ; // ストレージの解放
}

```

}

スカラー型に対しても、デストラクターを明示的に呼び出すことができる。これによつて、テンプレートのコードにおいて、型が組み込み型であるかどうかを気にしなくてすむ。

```
int main()
{
    typedef int I ;
    I i ;
    i.I() ; // OK、なにもしない
}
```

デストラクターを呼び出した後のオブジェクトに対して、再びデストラクターを呼び出した場合の挙動は未定義である。

12.5 フリーストア (Free store)

クラスに対する確保関数 (operator new) と解放関数 (operator delete) は、メンバーメンバーメンバー関数としてオーバーロードすることができる。確保関数と解放関数の具体的な実装方法については、[動的メモリー管理](#)を参照。

クラスのメンバーメンバーメンバー関数としての確保関数、解放関数は、staticメンバーメンバーメンバー関数である。たとえstatic指定子が明示的に使われていなくても、staticメンバーメンバーメンバー関数となる。

```
#include <cstddef>

struct X
{
    // 確保関数
    void * operator new ( std::size_t size )
    { return ::operator new( size ) ; }
    // 配列
    void * operator new[] ( std::size_t size )
    { return ::operator new( size ) ; }
    // placement form
    void * operator new ( std::size_t size, int, int, int )
    { return ::operator new( size ) ; }

    // 解放関数
}
```

```

void operator delete( void * ptr )
{ ::operator delete ( ptr ) ; }
// 配列
void operator delete[] ( void * ptr )
{ ::operator delete ( ptr ) ; }

// placement form
void operator delete( void * ptr, int, int, int )
{ ::operator delete ( ptr ) ; }

} ;

```

解放関数に例外指定がない場合、noexcept(true)が指定されたものとみなされる。

12.6 初期化(Initialization)

この項目では、クラスのオブジェクトの初期化について取り扱う。特に、明示的に初期化子が指定されているオブジェクトと、クラスの基本クラスとメンバーのサブオブジェクトの初期化方法を解説する。

クラスのオブジェクトに初期化子が指定されていない場合の初期化方法は、8.5 [初期化子](#)を参照。

クラスオブジェクトの配列の要素を初期化する際には、コンストラクターは、添字の順番に呼ばれる。デストラクターはコンストラクターの逆順に呼ばれる。

```

#include <iostream>

class X
{
private :
    int value ;
public :
    X( int value ) : value(value) { std::cout << value ; }
    ~X() { std::cout << value ; }
} ;

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

```

X型の配列の要素は、a[0], a[1], a[2]の順番に構築され、a[2], a[1], a[0]の順番に破棄される。したがって、出力は、123321となる。

12.6.1 明示的な初期化(Explicit initialization)

クラスのオブジェクトの初期化子には括弧に囲まれた式リストを使うことができる。この場合、適切な仮引数リストのコンストラクターによって初期化される。

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

int main()
{
    X x1( 1 ) ; // X::X(int)
    X x2( 1, 2 ) ; // X::X(int,int)
    X x3( 1, 2, 3 ) ; // X::X(int,int,int)
}
```

詳しくは、8.5 [初期化子](#)の直接初期化を参照。

また、=(イコール)記号に続いて値を指定することで、初期化することもできる。

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

int main()
{
    X x = 0 ; // X::X(int)
}
```

詳しくは、8.5 [初期化子](#)のコピー初期化を参照。

クラスのオブジェクトは、初期化リストで初期化することができる。

```

struct X
{
    X( int ) { }
} ;

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

```

詳しくは、8.5.7 [リスト初期化](#)を参照。

12.6.2 基本クラスとデータメンバーの初期化 (Initializing bases and members)

基本クラスは、コンストラクター初期化子により初期化できる。データメンバーは、コンストラクター初期化子か、メンバーの宣言に続く初期化子によって、初期化できる。また、コンストラクターはデリゲート(Delegate)できる。

コンストラクター初期化子

クラスのコンストラクターの定義で、直接の基本クラス、virtual基本クラス、非staticデータメンバーを初期化できる。文法は、以下のようになる。

コンストラクター初期化子:

: メンバー初期化子, メンバー初期化子 ...opt

メンバー初期化子:

メンバー初期化識別子 (式リスト)

メンバー初期化識別子 初期化リスト

メンバー初期化識別子:

クラス名

decltype

識別子

struct Base

```

{
    Base( int ) { }
} ;

struct Derived : Base
{
    int member1 ;
    int member2 ;

    Derived()-
        // コンストラクター初期化子
        : Base( 0 ), // 基本クラス
          member1( 0 ), // メンバー
          member2{ 0 } // メンバー（初期化リスト）
    { }
} ;

```

非修飾のメンバー初期化識別子は、まずコンストラクターのクラスのスコープ内で名前探索され、見つからなかった場合は、基本クラスのスコープから探される。そのため、基本クラスの名前と、クラスの非staticデータメンバーの名前が衝突した場合、必ずメンバーの名前が使われる。その場合、基本クラスを指定するには、修飾名を用いなければならない。

```

struct Base { } ;

struct Derived : Base
{
    int Base ;
    Derived() :
        Base(), // Derivedの非staticデータメンバー
        Derived::Base() // 基本クラス
    { }
} ;

```

メンバー初期化識別子として使える名前は、直接の基本クラスと、virtual基本クラス、コンストラクターのクラスの非staticデータメンバーである。

```

struct A { } ;
struct B { } ;
struct C : B, virtual A { } ;
struct D : C
{

```

```
D() :
    C(), // OK
    A() // OK
    // BはDの直接の基本クラスではないので使えない
    { }
} ;
```

メンバー初期化子識別子には、基本クラスの型を指し示すtypedef名やdecltype指定子を使うこともできる。

```
struct A { } ;
typedef A type ;
struct B { } ;
B b ;

struct C : A, B
{
    C() : type(), decltype(b)()
    { }
} ;
```

複数の9.9.2 共用メンバーのうちの、ひとつだけを、メンバー初期化子で初期化することができる。

```
union U
{
    int a ; int b ;
    U() : a(0) { } // OK、ひとつだけ
} ;

struct S
{
    union { int a ; int b ; } ;
    S() : a(0) { } // OK ひとつだけ
} ;

union Error
{
    int a ; int b ;
    Error() : a(0), b(0) // エラー、複数の指定
    { }
```

```
} ;
```

同じunionのメンバーである共用メンバーは、オブジェクト上のストレージを共有しているので、複数初期化することはできない。

メンバー初期化子に、同じメンバー名、あるいは基本クラス名を、複数指定することはできない。

```
struct Base { } ;
struct Derived : Base
{
    int member ;
    Derived()
    : member(), member(), // エラー、同じメンバー名
      Base(), Base() // エラー、同じ基本クラス名
    { }
} ;
```

コンストラクターのデリゲートについては、[12.6.2.1 コンストラクターのデリゲート](#)を参照。

メンバー初期化が明示的に指定されておらず、アブストラクトクラスのvirtual基本クラスでもない非staticデータメンバーと基本クラスは、次のように初期化される。

非staticデータメンバーに初期化子が指定されている場合、[8.5 初期化子](#)の方法に従って初期化される。

```
struct S
{
    int member = 123 ;
    S() /*メンバー初期化子によるmemberの指定なし*/ { }
} ;
```

この例では、memberは、123で初期化される。

共用メンバーの場合、初期化されない。

```
union U
{
    int member ;
    U() /*メンバー初期化子による共用メンバー指定なし*/ { }
```

```

} ;

struct S
{
    union { int member ; } ;
    S() /*メンバー初期化子による共用メンバーの指定なし*/ { }
} ;

union initialize
{
    int member ;
    initialize() : member(0) { } // memberを0で初期化
} ;

```

共用メンバーには、明示的な初期化が必要である。

それ以外の場合、8.5.2 [デフォルト初期化](#)される。

```

struct X { X() { } } ;

struct S
{
    int m1 ;
    X m2 ;
    S() { }
} ;

```

この例では、int型の非staticデータメンバーm1の初期化処理は8.5.2 [デフォルト初期化](#)で定義されているように、何もしない。X型m2は、X型のデフォルトコンストラクターによって初期化される。

同じunionに属する非staticな共用メンバーは、ひとつしか初期化できない。

```

union U
{
    int m1 ; int m2 ;
    U() : m1(0), m2(0) { } // エラー
} ;

struct X
{
    union { int m1 ; int m2 ; } ;

```

```

    X() : m1(0), m2(0) { } // エラー
} ;

struct Y
{
    union { int m1 ; } ;
    union { int m2 ; } ;
    Y() : m1(0), m2(0) { } // OK、違うunionの共用メンバー
} ;

```

Yの例は、違うunionの共用メンバーなので、問題のないコードである。

クラスのコンストラクターの実行が終了した時点で、初期化も明示的な値の設定もされていないメンバーの値は、不定である。

```

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

```

X x1 ; // staticストレージ上に構築されたオブジェクトは、ゼロ初期化されるので、x1.memberの値は0

```

int main()
{
    X x2 ; // x2.memberの値は不定
}

```

非staticデータメンバーの宣言に初期化子があり、メンバー初期化子も指定されている場合、メンバー初期化子が優先される。この場合、メンバー宣言の初期化子は無視される。

```

struct X
{
    int member = 1;
    X() { } // memberは1で初期化される
    X( int arg ) : member( arg ) { } // memberはargで初期化される
} ;

```

```
int main()
{
    X x1 ; // x1.memberの値は1
    X x2(2) ; // x2.memberの値は2
}
```

デリゲートしていないコンストラクターにおける初期化は、以下のように行われる。

まず始めに、クラスが最も派生した型である場合、virtual基本クラスが初期化される。

```
struct V { } ;

struct B : virtual V { } ;
struct C : B { } ;
struct D : C { } ;

int main()
{
    D d ; // DのコンストラクターでVが初期化される
    C c ; // CのコンストラクターでVが初期化される。
    B b ; // BのコンストラクターでVが初期化される。
}
```

virtual基本クラスは、最も派生したクラスで初期化されるということには、注意が必要である。例えば、以下のような場合、

```
struct V
{
    int member ;
    V( int arg ) : member( arg ) { }
} ;

struct B : virtual V
{
    B() : V(1) { }
} ;
struct C : B { } ;

int main()
{
    C c ; // エラー、Vのデフォルトコンストラクターは暗黙にdelete
           // 定義されている。
}
```

}

V1はCで初期化されるので、Bによる初期化は、無視されてしまう。Cのメンバー初期化子には、V1は記述されていないので、V1はデフォルト初期化される。V2のデフォルトコンストラクターは暗黙にdelete定義されているので、エラーとなる。

複数のvirtual基本クラスを持つ場合、初期化の順番は、深度優先(depth-first)かつ、左から右(left-to-right)となる。「深度」とは、基本クラスに行くほど深くなる。「左から右」とは、基本クラス指定子リストに現れるvirtual基本クラスの順番である。

```
struct V1 { } ; struct V2 { } ; struct V3 { } ;

struct B : virtual V1, virtual V2 { } ;
struct C : B, virtual V3 { } ;

C c ; // V1, V2, V3の順番で初期化される
```

virtual基本クラスの初期化が終わった後で、直接の基本クラスが、基本クラス指定子リストに現れる順番で初期化される。メンバー初期化子は、初期化の順番に影響しない。

```
struct B1 { } ; struct B2 { } ;
struct D : B1, B2
{
    D() : B2(), B1() { }
} ;

D d ; // B1, B2の順番に初期化される
```

メンバー初期化子は、初期化の順番に何の影響も与えないことに注意しなければならない。これは、副作用が初期化に影響をあたえるような場合、問題になる。

```
int i ;

struct B1 { B1(int) { } } ;
struct B2 { B2(int) { } } ;

struct D1 : B1, B2
{
    D1() : B2(++i), B1(++i) { }
} ;
```

```

struct D2 : B2, B1
{
    D2() : B2(++i), B1(++i) { }
} ;

int main()
{
    i = 0 ;
    D1 d1 ; // B1(1)、B2(2)で初期化される
    D2 d2 ; // B2(1)、B1(2)で初期化される
}

```

メンバー初期化子の順番は、基本クラスの初期化順序に影響しない。そのため、ある基本クラスの初期化における副作用が、次の基本クラスの初期化に影響をあたえるようなコードでは、基本クラスの記述の順番を変えるだけで、初期化の結果が異なってしまう。一般に、直接の基本クラスの初期化の順番が保証されていることを前提にしたコードを書くべきではない。

直接の基本クラスの初期化が終わった後で、クラス定義内の非staticデータメンバーが、宣言されている順番で初期化される。メンバー初期化子は、初期化の順番に影響しない。

```

struct X
{
    int m1 ;
    int m2 ;
    X() : m2(0), m1(0) { }
} ;

X x ; // m1, m2の順番で初期化される

```

直接の基本クラスの場合と同じく、メンバー初期化子は初期化の順番に影響しないということに注意しなければならない。非staticデータメンバーの初期化の順番は、クラス定義の中でメンバーが宣言されている順番である。したがって、あるメンバーの初期化の副作用が、次のメンバーの初期化に影響をあたえるようなコードでは、メンバーの宣言の順番を変えただけで、初期化処理が異なってしまう。具体的な問題例は、直接の基本クラスの場合と同じである。一般に、非staticデータメンバーの初期化の順番が保証されていることを前提にしたコードを書くべきではない。

最後に、コンストラクターの本体が実行される。

```

struct V { } ;
struct B { } ;

```

```
struct M { } ;

struct D : B, virtual V
{
    M m ;
    D() { /* コンストラクターの本体 */ }
} ;
```

D d ; // V, B, m, コンストラクターの本体の順番で初期化される

メンバー初期化子における名前は、コンストラクターの本体で評価される。

```
int ;

struct X
{
    int x ;
    int y ;
    X() : x(0), y(x) { }
} ;
```

メンバー初期化子では、thisを使うことができる。ただし、thisの参照先はまだ構築途中である場合もあるので、注意が必要である。

非staticメンバー関数は、virtual関数を含めて、構築中のオブジェクトであっても呼び出すことができる。また、構築途中のオブジェクトを、5.5.8 typeid演算子や5.5.7 Dynamic cast(Dynamic cast)のオペランドに渡すこともできる。

ただし、コンストラクター初期化子において、まだすべての基本クラスの初期化が終わっていない時点で、この種の操作を行った場合、結果は未定義である。これは、間接的に操作が行われる場合も含む。

```
struct A { A(int) { } } ;
struct B : A
{
    int f() { return 0 ; }
    B() : A( f() ) // 結果は未定義
    { }
} ;

// 間接的に操作が行われる例
struct C : A
{
```

```

static int call_f( C * ptr ) { return ptr->f() ; }
int f() { return 0 ; }
C() : A( call_f( this ) ) // 未定義
{
}
} ;

```

構築中のオブジェクトに対してvirtual関数を呼び出したり、typeidやdynamic_castを使った場合の挙動は、12.7 [生成と破棄](#)を参照。

メンバー初期化子では、パック展開できる。

```

template < typename Bases >
struct X : Bases...
{
    X() : Bases()...
    {
    }
} ;

```

12.6.2.1 コンストラクターのデリゲート

メンバー初期化識別子に、クラス型を指定することによって、別のコンストラクターに初期化処理を委譲することができる。これを、コンストラクターのデリゲート(delegate)という。

```

struct X
{
    int member ;

    X( int value ) : member( value )
    { /* 初期化処理 */ }

    X( double d ) : X(123) // コンストラクターのデリゲート
    {
        // 追加の処理
    }
} ;

```

この例では、コンストラクターX::X(double)は、初期化処理をX::X(int)にデリゲートしている。

別のコンストラクターにデリゲートしているコンストラクターのことを、デリゲートコンストラクター(delegating constructor)といい、デリゲート先のコンストラクターのことを、ターゲットコンストラクター(target constructor)という。またオブジェクトの初期化のために最初に呼び出されたコンストラクターのことを、最初のコンストラクター(principal constructor)という。

```
struct X
{
    X() : X( 0 ) { }
    X( int ) : X( 0.0 ) { }
    X( double ) { }
} ;

X x ; // 初期化
```

上に例における、Xのオブジェクトxの初期化では、最初のコンストラクターとして、X::X()が選ばれる。これは、デリゲートコンストラクターであり、ターゲットコンストラクターであるX::X(int)にデリゲートする。X::X(int)もデリゲートコンストラクターであり、ターゲットコンストラクターのX::X(double)にデリゲートする。

デリゲートコンストラクターは、他のメンバー初期化識別子を指定してはならない。

```
struct Base { } ;

struct X : Base
{
    int member ;
    X() : X( 0 ),
          Base(), member(0) // エラー、デリゲートコンストラクター
    // は他の識別子を指定できない
    {
    }
    X( int ) { }
} ;
```

ターゲットコンストラクターは、オーバーロード解決により選ばれる。

```
struct X
{
    X() : X( 0 ) { } // X::X(int)を呼ぶ
    X( int ) : X( 0.0 ) { } // X::X(double)を呼ぶ
    X( double ) { }
} ;
```

ターゲットコンストラクターが処理を返した後に、デリゲートコンストラクターの本体が実行される。

```
struct X
{
    int member ;
    X() : X( 0 )
    { /* 処理2 */ }
    X( int value ) : member( value )
    { /* 処理1 */ }
} ;

X x ;
```

この例では、オブジェクトxの初期化の際、最初のコンストラクターとしてX::X()が選ばれる。これはデリゲートコンストラクターである。ターゲットコンストラクターは、X::X(int)となる。ターゲットコンストラクターは、通常のコンストラクターと同じように基本クラスやメンバーの初期化を終えた後、コンストラクターの本体を実行し(処理1)、処理を返す。ターゲットコンストラクターが処理を返したので、最初のコンストラクターの本体が実行される(処理2)。

デリゲートコンストラクターが、直接的にせよ、間接的にせよ、自分自身にデリゲートを行った場合は、エラーとなる。

```
struct X
{
    X() : X() {} // エラー、直接的な自分自身へのデリゲート
} ;

struct Y
{
    Y() : Y(0) {} // エラー、間接的な自分自身へのデリゲート
    Y(int) : Y(0.0) {} // エラー、間接的な自分自身へのデリゲート
    Y(double) : Y() {} // エラー、間接的な自分自身へのデリゲート
} ;
```

クラスYは、間接的に、自分自身へのデリゲートを行う例である。デリゲートのネスト、つまり他のデリゲートコンストラクターへのデリゲートは可能である。ただし、間接的であっても、自分自身へのデリゲートは認められない。

12.7 生成と破棄(Construction and destruction)

クラスのオブジェクトが構築される前、破棄された後、あるいは構築や破棄の最中に
は、いくつか気を付けなければならないことがある。

非トリビアルコンストラクターを持つクラスのオブジェクトのコンストラクターの実行が始
まる前に、非staticメンバーや基本クラスにアクセスした場合、挙動は未定義である。

```
struct X
{
    X() { } // 非トリビアルコンストラクター
    int member ;
} ;

int main()
{
    X * ptr = static_cast<X *>( operator new( sizeof(X) ) )
; // 初期化されていないストレージ
    ptr->member ; // 未定義
    &ptr->member ; // 未定義、ポインターを得ることもできない
    new(ptr) X ; // 初期化
    ptr->member ; // OK
    &ptr->member ; // OK
    operator delete( ptr ) ;
}
```

非トリビアルデストラクターを持つクラスのオブジェクトのデストラクターの実行が終わっ
た後に、非staticメンバーや基本クラスにアクセスした場合、挙動は未定義である。

```
struct X
{
    ~X() { } // 非トリビアルデストラクター
    int member ;
} ;

int main()
{
    X * ptr = static_cast<X *>( operator new( sizeof(X) ) )
; // 初期化されていないストレージ
    new(ptr) X ; // 初期化
```

```

ptr->~X() ; // デストラクターの実行

ptr->member ; // 未定義
operator delete( ptr ) ;
}

```

クラスへのポインターを、基本クラスへのポインターに型変換する際には、クラスとそのすべての基本クラスのコンストラクターの実行が始まっていなければならない。また、デストラクターの実行が完了していかなければならない。そうでない場合の挙動は未定義である。これは、トリビアルクラスにも当てはまる。

```

struct X { } ;
struct Y : X { } ;

int main()
{
    Y * y_ptr = static_cast<Y *>( operator new( sizeof(Y) ) )
; // 初期化されていないストレージ

    X * x_ptr = y_ptr ; // 未定義
    new (y_ptr) Y ; // 初期化
    x_ptr = y_ptr ; // OK
    y_ptr->~Y() ; // デストラクターの実行
    x_ptr = y_ptr ; // 未定義

    operator delete( y_ptr ) ;
}

```

オブジェクトの構築や破棄の途中で、メンバー関数を呼び出すことはできる。ただし、virtual関数をコンストラクターやデストラクターの中で呼び出す際には、注意が必要である。virtual関数をコンストラクターやデストラクターの中、あるいはその中から呼び出された関数内で呼び出すと、そのコンストラクターあるいはデストラクターのクラスの型にとってのファイナルオーバーライダーが用いられ、派生クラスは考慮されない。これは、基本クラスのコンストラクターの実行中には、派生クラスはまだ完全に初期化されていないからである。

```

struct A
{
    virtual void f() { }
    virtual void g() { }
    // A::f、A::gを呼び出す
    A() { f() ; g() ; }
    virtual ~A() { f() ; g() ; }
}

```

```

} ;

struct B : A
{
    virtual void f() { }
    // B::f, A::gを呼び出す
    B() { f() ; g() ; }
    virtual ~B() { f() ; g() ; }
} ;

struct C : B
{
    virtual void g() { }
    // B::f, C::gを呼び出す
    C() { f() ; g() ; }
    virtual ~C() { f() ; g() ; }
} ;

```

この例で、たとえクラスCのオブジェクトを構築したとしても、基本クラスAのコンストラクターの中ではA::f, A::gが呼ばれることになる。

オブジェクトの構築や破棄の途中で、typeid演算子を使うことはできる。 typeid演算子がコンストラクターやデストラクターの中、あるいはその中から呼び出された関数内で使われていて、 typeid演算子のオペランドが、そのクラスの構築中のオブジェクトである場合、 typeidはコンストラクターやデストラクターのクラス型情報を表す std::type_info オブジェクトを返す。これは、基本クラスの構築中は、まだ派生クラスは構築し終わっていないからである。

```

struct A
{
    A()
    {
        typeid( *this ) == typeid( A ) ; // true
    }
    virtual ~A()
    {
        typeid( *this ) == typeid( A ) ; // true
    }
} ;

struct B : A { } ;

```

この例では、たとえBのオブジェクトが構築されたとしても、 typeidはA型を表す std::type_info オブジェクトを返す。

オブジェクトの構築や破棄の途中で、`dynamic_cast`を使うことはできる。`dynamic_cast`がコンストラクターやデストラクターの中、あるいはその中から呼び出された関数内で使われていて、オペランドがそのクラスの構築中のオブジェクトである場合、コンストラクターやデストラクターの属するクラスが、最終的に派生された型であるとみなされる。これは、基本クラスの構築中は、まだ派生クラスは構築し終わっていないからである。

```
struct A
{
    A() ;
    virtual ~A() ;
} ;

struct B : A { } ;

A::A()
{
    B * ptr = dynamic_cast<B *>( this ) ; // 常にnullポインタ
}

A::~A()
{
    B * ptr = dynamic_cast<B *>( this ) ; // 常にnullポインタ
}
```

たとえ、Aから派生されたB型のオブジェクトであっても、Aのコンストラクター、デストラクターの中では、A型が最終的な派生クラスであるとみなされる。

12.8 クラスのコピーとムーブ

クラスのオブジェクトは、初期化と代入によって、コピーもしくはムーブされる。コピーやムーブを行うためのコンストラクターと代入演算子を、それぞれ特別に、コピー・コンストラクター、ムーブ・コンストラクター、コピー・代入演算子、ムーブ・代入演算子と呼ぶ。

コピー・コンストラクター(`copy constructor`)とは、あるクラスXにおいて、非テンプレートなコンストラクターで、一つ目の仮引数の型が、`X &`、`const X &`、`volatile X &`、`const volatile X &`のいずれかであり、二つ目以降の仮引数は存在しないか、すべてデフォルト実引数があるものをいう。

```
struct X
{
```

```
// コピーコンストラクター
X( X & ) ;
X( X const & ) ;
X( X volatile & ) ;
X( X const volatile & ) ;
X( X const &, int x = 0, int y = 0 ) ; // 二つ目以降の仮引数にデフォルト実引数がある

// コピーコンストラクターではない
X( ) ;
X( int ) ;
template < typename T >
X( T ) ; // テンプレートコンストラクターはコピーコンストラクターではない
X( X const &, short ) ; // 二つ目以降の仮引数にデフォルト実引数がない
} ;
```

ムーブコンストラクター(move constructor)とは、あるクラスXにおいて、非テンプレートなコンストラクターで、一つ目の仮引数の型が、`X &&`、`const X &&`、`volatile X &&`、`const volatile X &&`のいずれかであり、二つ目以降の仮引数は存在しないか、すべてデフォルト実引数があるものをいう。

```
struct X
{
    X( X && ) ;
    X( X const && ) ;
    X( X volatile && ) ;
    X( X const volatile && ) ;
    X( X const &&, int x = 0 ) ;
}
```

クラスXのコンストラクターの一つ目の仮引数の型がXで、二つ目以降の仮引数が存在しないか、すべてデフォルト実引数が指定されている場合は、エラーとなる。

```
struct X
{
    X( X ) ; // エラー
}
```

また、テンプレートコンストラクターのインスタンス化の結果が、このようなシグネチャに

なる場合、そのテンプレートはインスタンス化されない。

```
struct X
{
    template < typename T >
    X( T ) { } // X<X>というインスタンス化は起こらない。
};

int main()
{
    X a( 0 ) ; // テンプレートコンストラクターが使われる。
    X b( a ) ; // 暗黙のコピーコンストラクターが使われる
}
```

あるクラスにおいて、コピーコンストラクターが明示的に宣言されていない場合、コピーコンストラクターは暗黙的に宣言される。もし、クラスにユーザー定義のムーブコンストラクター、ムーブ代入演算子、コピー代入演算子、デストラクターが存在する場合、コピーコンストラクターは暗黙的にdelete定義される。そうでない場合は、default定義される。

```
struct A
{
    // コピーコンストラクターは暗黙的にdefault定義される
    // A( A const & ) = default ;
};

struct B
{
    B( B && ) ; // ユーザー定義ムーブコンストラクター
    B & operator = ( B && ) ; // ユーザー定義ムーブ代入演算子
    B & operator = ( B & ) ; // ユーザー定義コピー代入演算子
    ~B() ; // ユーザー定義デストラクター

    // コピーコンストラクターは暗黙的にdelete定義される
    // B( B const & ) = delete ;
};
```

C++98/03では、暗黙のコピーコンストラクターはユーザー定義のコピー代入演算子、ユーザー定義のムーブ代入演算子、ユーザー定義デストラクターがある場合でも、暗黙的に宣言された。C++11では、この挙動は非推奨になった。将来的に取り除かれる予定だ。

```

struct S
{
    ~S() { }
} ;

int main()
{
    S s1 ;
    // OK: C++98, C++03まで
    // 非推奨: C++11以降
    S s2( s1 ) ;

}

```

理由は、そのようなユーザー定義の特別なメンバー関数がある場合は、大抵、暗黙のデフォルトのコピーコンストラクターの挙動は、容易にプログラミング上の誤りを引き起こすためである。

ユーザー定義のコピー代入演算子、ユーザー定義のムーブ代入演算子、ユーザー定義デストラクターがある場合の暗黙のコピーコンストラクターの宣言は、非推奨であり、将来の規格では取り除かれる。そのため、このような非推奨に依存したコードを書いてはならない。

クラスXの暗黙のコピーコンストラクターのシグネチャは、通常、

```
X::X( const X & )
```

となる。ただし、直接の基本クラスやvirtual基本クラス、非staticデータメンバーがconst修飾されていない仮引数のコンストラクターを持つ場合、

```
X::X( X & )
```

となる。

```

struct A
{
    A() = default ;
    A( A const & ) { }
} ;

```

```

struct B : A
{
    // 暗黙のコピーコンストラクターのシグネチャ
    // B( B const & ) = default ;
} ;

struct C
{
    C() = default ;
    C( C & ) { }
} ;

struct D : C
{
    // 暗黙のコピーコンストラクターのシグネチャ
    // D( D & ) = default ;
} ;

```

あるクラスにおいて、ムーブコンストラクターが明示的に宣言されていない場合、ムーブコンストラクターは暗黙的に宣言される。もし、あるクラスがユーザー定義の、コピーコンストラクター、コピー代入演算子、ムーブ代入演算子、デストラクターを持たず、またムーブコンストラクターがdelete定義されていない場合、ムーブコンストラクターはdefault定義される。

ユーザー定義のムーブ代入演算子が存在する場合、ムーブコンストラクターは暗黙的にdefault定義されない。これは、デフォルトのムーブコンストラクターの挙動と、ユーザー定義のムーブ代入演算子の挙動が異なる可能性があるため、安全のためにdefault定義されないのである。

```

struct X
{
    // ムーブコンストラクターはdefault定義されない

    // ユーザー定義のムーブ代入演算子
    X & operator = ( X && obj )
    {
        // ユーザー定義のムーブを実装
    }
} ;

```

そのため、自前実装のムーブ構築とムーブ代入を行いたい場合、ムーブコンストラクターとムーブ代入演算子を両方ユーザー定義する必要がある。

ムーブコンストラクターが、暗黙にも明示的にも宣言されていない場合、ムーブコンスト

ラクターを呼び出す式は、代わりにコピーコンストラクターを呼び出す。

```
struct X
{
    X() = default;
    // ユーザー定義のコピーコンストラクター
    X( X const & ) { }
    // ムーブコンストラクターは宣言されない
};

int main()
{
    X a;
    // コピーコンストラクターを呼び出す
    X b( static_cast< X && >( a ) );
}
```

クラスXの暗黙のムーブコンストラクターのシグネチャは、以下の通りである。

```
X::X( X && )
```

あるクラスが以下の条件を満たした時、対応するコピー／ムーブコンストラクターは delete定義される。つまり、以下の条件でコピーができない場合はコピーコンストラクターが、ムーブができないときはムーブコンストラクターが、それぞれ個別にdelete定義される。

- クラスがunionのようなクラスで、共用メンバーがそれぞれ非トリビアルなコピー／ムーブ・コンストラクターを持つ場合

```
struct NonTrivial
{
    NonTrivial( NonTrivial const & ) { }
    NonTrivial( NonTrivial && ) { }
};

struct X
{
    X() { }
    union { NonTrivial n; } ;
    // コピーコンストラクターは、nが非トリビアルなコピーコン
```

```
ストラクターを持つためにdelete定義される
// ムーブコンストラクターは、nが非トリビアルなムーブコン
ストラクターを持つためにdelete定義される。
} ;
```

- 非staticデータメンバーが、オーバーロード解決の結果、コピー/ムーブできない場合

オーバーロード解決の結果というのは、複数の候補があつて曖昧である場合、選ばれた最適関数がdeleted定義されている場合、アクセス指定によりクラスのデフォルトコンストラクターからは利用できない場合だ。

```
// コピーできない
struct uncopyable
{
    uncopyable( uncopyable const & ) = delete ;
} ;

// コピーコンストラクターがdelete定義される
struct S
{
    uncopyable member ;
} ;
```

アクセス指定によりクラスのデフォルトコンストラクターからは利用できない場合の例

```
struct private_copy
{
private :
    private_copy( private_copy const & ) = delete ;
} ;

// コピーコンストラクターがdelete定義される
struct S
{
    // memberのコピーコンストラクターは
    // クラスSのデフォルトコンストラクターからアクセスできな
    // i
    private_copy member ;
} ;
```

- 直接、あるいはvirtualな基本クラスが、オーバーロード解決の結果、コピー／ムーブできない場合
- 直接、あるいはvirtualな基本クラス、もしくは非staticデータメンバーの、デストラクターが、delete定義であるか、あるいはアクセス指定により、デフォルトコンストラクターからアクセスできない場合

```
// デストラクターがprivateメンバーのクラス
struct M
{
private :
    ~M() { }
} ;

// コピーとムーブコンストラクターがdelete定義される
struct S
{
    M m ;
} ;
```

- 非staticデータメンバーがrvalueリファレンス型である場合、コピーコンストラクターがdelete定義される。

```
// コピーコンストラクターがdelete定義される
struct S
{
    static int data ;

    // rvalueリファレンス型の非staticデータメンバー
    int && rref ;
    S() : rref( static_cast< int && >(data) ) { }

    int S::data ;

    int main()
    {
        S s1 ;
        S s2 = s1 ; // エラー、コピーコンストラクターがdelete定義されている
    }
}
```

デフォルト定義されたムーブコンストラクターがdeleted定義されている場合、オーバーロード解決では無視される。

コピー/ムーブコンストラクターがトリビアル(trivial)となるには、ユーザー定義されず、仮引数リストが暗黙に宣言された場合の仮引数リストと同じで、以下の条件をすべて満たす必要がある。

- クラスはvirtual関数とvirtual基本クラスを持たないこと
- volatile修飾された非staticデータメンバーを持たないこと
- 直接の基本クラスのサブオブジェクトをコピー/ムーブするのに使われるコンストラクターがトリビアルであること
- クラス型か、クラスの配列型の非staticデータメンバーをコピー/ムーブするのに使われるコンストラクターがトリビアルであること

これらの条件をすべて満たさない限り、コピー/ムーブコンストラクターは、非トリビアル(non-trivial)である。

コピー/ムーブコンストラクターが、default化されていて、定義もdeleted定義もされていない場合、ODRの文脈で使われるか、最初の宣言で明示的にdefault化された場合、暗黙に定義される(implicitly defined)。もし、暗黙に定義されたコンストラクターがconstexprコンストラクターの制約を満たすのならば、暗黙に定義されたコンストラクターは、constexprコンストラクターになる。

コピー/ムーブコンストラクターが暗黙に定義されるためには、直接の基本クラス、virtual基本クラス、非staticデータメンバーの、ユーザー提供されていないコピー/ムーブコンストラクターは、すべて暗黙に定義されていなければならない。ユーザー提供されている場合は、暗黙に定義されていなくてもよい。

```
struct Base
{
    // コピーコンストラクターは暗黙に定義されていない
    Base( const Base & ) = delete ;
} ;

struct Derived : Base
{
    // コピーコンストラクターは暗黙にdefault化される
} ;

void f()
{
    Derived d1 ;
    Derived d2 = d1 ; // エラー、コピーコンストラクターは暗黙に
                      // 定義されていない
}
```

unionではないクラスの暗黙に定義されたコピー／ムーブコンストラクターは、基本クラスとメンバーに対し、メンバーごとのコピー／ムーブを行う。初期化の順番は、ユーザー定義されるコンストラクターの場合と同じである。

union型の暗黙に定義されたコピー／ムーブコンストラクターは、union型のオブジェクトの内部表現をコピーする。

ユーザー宣言されたクラスXのコピー／ムーブ代入演算子、X::operator =は、クラスXの非static、非テンプレートのメンバー関数で、仮引数を一つだけ取り、その型はX, X &, const X &, volatile X & const volatile X &でなければならない。

```
struct X
{
    X & operator = ( X & ) ;
    X & operator = ( X const & ) ;
    X & operator = ( X volatile & ) ;
    X & operator = ( X const volatile & ) ;
};
```

ユーザー宣言されるコピー／ムーブ代入演算子の戻り値の型は制限されていない。

```
struct X
{
    // OK
    void operator = ( X const & ) { }

};

void f()
{
    X x1 ;
    X x2 ;
    x1 = x2 ; // OK、結果の型はvoid
}
```

コンストラクターと同じく、代入演算子でも、代入演算子のテンプレートは、コピー／ムーブ代入演算子ではない。ただし、コンストラクターと同じように、テンプレートの特殊化がコピー／ムーブ代入演算子と同じ仮引数になった場合は、オーバーロード解決の候補となり、非テンプレートのコピー／ムーブ代入演算子より優先して選択されることもある。このため、代入演算子のテンプレートは、コピー風／ムーブ風の代入演算子と非公式に呼ばれている。

```
struct X
{
```

```

X() { }

X & operator = ( X const & ) ; // #1
template < typename T >
X & operator = ( T & ) ; // #2
} ;

void f()
{
    X x1 ;
    X x2 ;
    x1 = x2 ; // #2が呼ばれる
}

```

もし、クラス定義の中でコピー代入演算子が明示的に宣言されていない場合、コピー代入演算子は暗黙的に宣言される。

```

struct X
{
// コピー代入演算子が暗黙的に宣言される
} ;

```

もし、クラス定義で、ムーブコンストラクター、ムーブ代入演算子、コピーコンストラクター、デストラクターがユーザー宣言されていた場合、暗黙に宣言されたコピー代入演算子は、`delete`定義される。

```

struct A
{
// コピー代入演算子はdelete定義される
    A( A && ) ;
} ;

struct B
{
// コピー代入演算子はdelete定義される
    B & operator = ( B && ) ;
} ;

```

C++11では、コピーコンストラクター、デストラクターがユーザー宣言されていた場合、コピー代入演算子は`default`化される。この挙動は非推奨であり、将来的には取り除かれる。このような非推奨の機能に頼ったコードを書いてはならない。

```
// このようなコードを書いてはならない

struct X
{
// C++11では、コピー代入演算子は暗黙にdefault化される
// この機能は非推奨であり、使ってはならない

    X( X const & ) ;
    ~X() ;
} ;
```

暗黙に宣言される、クラスXのコピー代入演算子は、以下の条件、

- クラスXの直接の基本クラスBが、仮引数の型としてconst B &, const volatile B &, Bのいずれかであるコピー代入演算子を持つ

```
struct B1 { B1 & operator = ( const B & ) ; }
struct B2 { B2 & operator = ( const volatile B & ) ; }
struct B3 { B3 & operator = ( B ) ; }

struct X : B1, B2, B3 { } ;
```

- Xの非staticデータメンバー、もしくは配列がすべて、型をMとおくと、仮引数の型がconst M &, const volatile M &, Mであるコピー代入演算子を持つ

をすべて満たす場合、

```
X & X::operator = ( const X & )
```

の形を取る。

上の条件を満たさない場合、暗黙に宣言されるクラスXのコピー代入演算子の形は、

```
X & X::operator = ( X & )
```

となる。

クラスXのユーザー宣言されるムーブ代入演算子、X::operator =は、非static、非テンプレートのメンバー関数で、仮引数を一つだけ取り、その型は、X &&, const X &&, volatile X &&, const volatile X &&でなければならない。

あるクラスXの定義でムーブ代入演算子が明示的に宣言されていない場合は、以下の条件をすべて満たした時、暗黙的にムーブ代入演算子が宣言される。

- クラスXはユーザー宣言されたコピー構造体を持たない
- クラスXはユーザー宣言されたムーブ構造体を持たない
- クラスXはユーザー宣言されたコピー代入演算子を持たない
- クラスXはユーザー宣言されたデストラクターを持たない
- ムーブ代入演算子は暗黙にdelete定義されていない

暗黙に宣言されたクラスXのムーブ代入演算子は、以下の形を取る。

```
X & X::operator = ( X && )
```

暗黙に宣言されたクラスXのコピー/ムーブ代入演算子は、X &型の戻り値を返す。戻り値として返されるのは代入演算子が呼ばれたクラスのオブジェクトへのリファレンスである。暗黙に宣言されたコピー/ムーブ代入演算子は、クラスのinline publicメンバーとなる。

デフォルト化されたクラスXのコピー/ムーブ代入演算子は、クラスXが以下のいずれかのメンバーを持つ時、それぞれ対応するコピー/ムーブ代入演算子が、delete定義される。

- クラスXがunion風クラスで、その共用メンバーに、非トリビアルなコピー/ムーブ代入演算子があるとき

9.9 [union風クラス](#)とは、unionか無名unionを含むクラスである。

```
// 非トリビアルなコピー代入演算子を持つ型
struct S
{
    S & operator = ( S const & ) { }
} ;

union U1 { S s ; } ;
struct U2
{
    union { S s ; } ;
} ;

int main()
{
    U1 a ;
    U1 b ;
    a = b ; // エラー、コピー代入演算子がdelete定義されてい
    る
```

```

U2 c ;
U2 d ;
c = d ; // エラー、コピー代入演算子がdelete定義されてい
る
}

```

- const修飾された非クラスの、型か配列型を、非staticデータメンバーとして持つ場合

```

struct S
{
    const int member ;
    const int array[10] ;
} ;

```

- リファレンス型の非staticデータメンバーを持つ場合
- オーバーロード解決の結果、コピー／ムーブ代入できないクラス型を非staticデータメンバーとして持つ場合。

コピー／ムーブ代入できないというのは、オーバーロード解決の結果が曖昧であるか、最適候補がdeleted定義であるか、アクセス指定により、クラスXのデフォルト代入演算子からアクセスできない場合をいう。以下同じ

- 直接、あるいはvirtualな基本クラスが、オーバーロード解決の結果、コピー／ムーブ代入できない場合。

デフォルト化されたムーブ代入演算子がdelete定義された場合、オーバーロード解決では無視される。

コピー／ムーブ代入演算子は、明示的に宣言されなかった場合、必ず暗黙的に宣言される。宣言の方法は様々で、delete定義になることもあるが、宣言されることはされる。これは、クラスの基本クラスの代入演算子は、必ず隠されることだ。using宣言を使って基本クラスの代入演算子をクラススコープに導入しても、そのクラスで宣言された代入演算子が優先されるため、やはり隠される。

あるクラスXのコピー／ムーブ代入演算子がトリビアルであるためには、ユーザー提供されず、仮引数リストが、暗黙に宣言された場合の仮引数リストと同じで、さらに以下の条件をすべて満たさなければならない。

- クラスXはvirtual関数を持たず、virtual基本クラスも持たない
- クラスXはvolatile修飾された型の非staticデータメンバーを持たない
- 直接の基本クラスのサブオブジェクトで使われるコピー／ムーブの代入演算子がトリビアル
- クラスXのクラス型とクラスの配列型の非staticデータメンバーの、コピー／ムーブに使われる代入演算子がトリビアル

この条件を満たさない場合、コピー／ムーブ代入演算子は非トリビアルとなる。

あるクラスXの、default化されているが、`delete`定義されていないコピー/ムーブコンストラクターは、ODRの文脈で使われた場合、もしくは、最初の宣言で明示的にdefault化された場合、暗黙に定義される。

暗黙に定義されたコピー/ムーブ代入演算子は、以下の条件をすべて満たした場合、`constexpr`関数になる。

- クラスXはリテラル型
- 直接の基本クラスのサブオブジェクトで、コピー/ムーブのために使われる代入演算子が、`constexpr`関数
- クラスXの、クラス型、もしくはクラスの配列型の非staticデータメンバーの、コピー/ムーブのために使われる代入演算子が、`constexpr`関数

あるクラスのデフォルト化されたコピー/ムーブ代入演算子が暗黙に定義されるには、クラスの直接の基本クラスと、非staticデータメンバーのコピー/ムーブ代入演算子のうち、ユーザー提供されていないものは、すべて暗黙に定義されなければならない。

`union`以外のクラスで、暗黙に定義されたコピー/ムーブ代入演算子は、クラスのサブオブジェクトに対してメンバーごとのコピー/ムーブを行う。クラスの直接の基本クラスがまず代入される。代入の順序は、基本クラス指定のリストで宣言された順番である。

```
struct A { } ;
struct B { } ;

// 代入されるときは、A, Bの順番
struct C : A, B { } ;

// 代入されるときは、B, Aの順番
struct D : B, A { } ;
```

その次に、クラスの非staticデータメンバーが、クラス定義で宣言された順番で代入される。サブオブジェクトがクラス型の場合は`operator =`を呼び出し、基本型の場合は組み込みの代入演算子を使い。配列型の場合は、要素ごとに代入される。

暗黙に定義されたコピーデ入演算子が、複数回派生されているvirtual基本クラスのサブオブジェクトのコピーデ入演算子を、派生された回数だけ呼び出すかどうかは、未規定である。

```
struct V
{
    V & operator = ( V const & ) ; // #1
} ;
struct A : virtual V { } ;
struct B : virtual V { } ;
struct C : A, B { } ; // 暗黙のコピーデ入演算子がdefault定義される
```

```

int main()
{
    C c1 ;
    C c2 ;
    c2 = c1 ; // #1が一度呼ばれるか、二度呼ばれるかは、未規定
}

```

クラスCには、VはA,B二つのクラスのvirtual基本クラスになっているため、クラスCには、Vのサブオブジェクトはひとつしかない。この場合、Vのコピー代入演算子が、一回呼ばれるとは限らない。ただし二回呼ばれるとも限らない。規格上、この場合に、暗黙に定義されたコピー代入演算子は、Vのコピー代入演算子を呼び出す回数は、一回でも二回でも良い。

なお、ムーブ代入演算子は、virtual基本クラスがある場合、暗黙にdelete定義されるので、この未規定はない。

もちろん、暗黙の定義ではなく、明示的に定義した場合は、明示的に書いただけの回数呼び出す。

unionの暗黙に定義されたコピー代入演算子は、オブジェクトの内部表現をコピーする。

コピー／ムーブのコンストラクターか代入演算子がODRの文脈で使われていて、アクセス指定のためにアクセスできない場合、エラーとなる。

条件次第で、C++の実装はオブジェクトのコピー／ムーブ構築を省略することができる。たとえ、そのオブジェクトのコンストラクタやデストラクターが、副作用を持っていたとしても、遠慮なく省略される。これをコピー省略(copy elision)という。規格上の用語は「コピー省略」だが、コピーだけではなく、ムーブも省略される可能性がある。

コピー省略は、以下の場合に許されている。

- クラスを返す関数のreturn文のオペランドの式が、関数とcatch句の仮引数を除く非volatileの自動オブジェクトの名前であり、関数の戻り値の型と自動オブジェクトの型が、ともにCV非修飾の型である場合、関数内の自動オブジェクトのコピー／ムーブが省略され、関数の戻り値のオブジェクトとして直接に構築することが許されている

```

struct S
{
    S() { }
    S( S const & ) { }
    ~S() { }
} ;

S f()
{

```

```

    S s ;
    return s ; // コピー省略が許されている
}

```

- throw式のオペランドが、関数とcatch句の仮引数を除く非volatileの自動オブジェクトの名前であり、その自動オブジェクトのスコープが、最も内側のブロックの外に出ない場合、例外オブジェクトに対するコピー/ムーブが省略され、例外オブジェクト上に直接構築することが許されている。

```

struct S
{
    S() { }
    S( S const & ) { }
    ~S() { }
} ;

int main()
{
    try
    {
        S s ; // 最も内側のブロックスコープの外に出ない
        throw s ; // コピー省略が許されている
    } catch ( S & s ) { }
}

```

- リファレンスで束縛されていないクラスの一時オブジェクトが、同じCV非修飾の型に、コピー/ムーブされた場合、コピー/ムーブは省略され、コピー/ムーブ先のオブジェクトに、一時オブジェクトを直接構築することが許されている。

```

struct S
{
    S() { }
    S( S const & ) { }
    ~S() { }
} ;

S f()
{
    return S() ; // コピー省略が許されている
}

```

```
int main()
{
    S s = f() ; // コピー省略が許されている
}
```

- 例外ハンドラーの例外宣言が、CV修飾子以外は同じ型のオブジェクトを、例外オブジェクトとして宣言している場合、コンストラクターとデストラクター呼び出し以外にプログラムの意味が変わらなければ、コピー／ムーブは省略され、例外宣言の仮引数は、例外オブジェクトを参照することが許されている。

```
struct S
{
    S() { }
    S( S const & ) { }
    ~S() { }
} ;

int main()
{
    try
    {
        throw S() ;
    }
    // コピー省略が許されている
    catch( S s ) { }
}
```

「コンストラクターとデストラクター呼び出し以外にプログラムの意味が変わらない」というのは、たとえば例外オブジェクトを変更するような場合だ。

```
catch( S s )
{
    s = S() ; // 例外オブジェクトを変更
}
```

例外宣言のリファレンスではない仮引数の指示するオブジェクトに変更を加えても、元の例外オブジェクトを変更しないと規定されているので、この場合は、コピー省略はできない。

コピー省略の条件は、組み合わさった場合でも、コピー省略されることが許されている。

コピー省略できる条件がそろい、コピーされるオブジェクトがlvalueの場合、オーバーロード解決が二回行われることがある。一回目のオーバーロード解決は、オブジェクトをrvalueとして、コピーするコンストラクターを探す。一回目のオーバーロード解決が失敗するか、選択されたコンストラクターの第一引数がrvalueリファレンスではない場合、オブジェクトをlvalueとして、二回目のオーバーロード解決が行われる。

一回目のオーバーロード解決で、選択されたコンストラクターの第一引数がrvalueリファレンスではない場合というのは、たとえばconst修飾されたlvalueリファレンスが該当する。

この二回のオーバーロード解決は、たとえコピー省略をしないとしても、必ず行われる。この規定の目的は、コピー省略が行われなかつたならば呼び出されるコンストラクターの、アクセス指定を調べるためにある。

12.9 コンストラクター継承 (Inheriting constructors)

using宣言を使って派生クラスから基本クラスのコンストラクターを指定することで、基本クラスのコンストラクターを明示的に継承できる。これにより、機械的な手書きのコードを省くことができる。

```
class Base
{
private :
    int member ;
public :
    Base( int value ) : member(value) { }
} ;

class Derived : Base
{
public :
    // Base::Base(int)を継承
    using Base::Base ;
} ;

int main()
{
    Derived d(0) ; // 継承コンストラクターを使う
}
```

using宣言は通常通り、アクセス指定の影響を受けることに注意すること。派生クラスによって継承された基本クラスのコンストラクターは、同じ仮引数をとり、引数をそのままメンバー初期化子で基本クラスに渡し、関数本体は空であるコードを手書きした場合と同じように動く。

```

struct Base { Base(int, double) { } } ;
struct Derived : Base
{
    // Baseクラスのコンストラクターの継承
    using Base::Base ;
    // 以下のコードを手書きした場合と同等
    // Derived( int p1, double p2 )
    // : Base( p1, p2 )
    // { }
} ;

```

このような手書きのコンストラクターを、実際に使うとエラーとなる場合、継承コンストラクターの使用もエラーとなる。

派生クラスで同じシグネチャーのコンストラクターをユーザー定義した場合、そのコンストラクターの継承は起こらない。

```

struct Base
{
    Base( int ) { }
    Base( double ) { }
} ;
struct Derived : Base
{
    using Base::Base ;

    Derived( int value ) : Base(value)
    {
        // 処理
    }
}

```

この場合、Base::Base(double)は継承されるが、Base::Base(int)は継承されない。クラス Derivedでユーザー定義されたコンストラクターが使用される。

継承コンストラクターの詳細はすこし難しい。まず、継承される基本クラスのコンストラクターが、継承コンストラクターの候補(candidate set of inherited constructors)として列挙される。この際、コンストラクターにデフォルト実引数がある場合は、デフォルト実引数を省略したシグネチャーの関数も追加される。たとえば、以下のようなクラスの場合、

```

struct A
{
    A( int i ) { }
} ;

struct B
{
    B( int p1 = 1, int p2 = 2 ) { }
} ;

```

クラスAの継承コンストラクターの候補は、以下の通り。

```

A( int )
A( A const & )
A( A && )

```

クラスBの継承コンストラクターの候補は、以下の通り。

```

B( ) // デフォルト実引数の省略形
B( int = 1 ) // デフォルト実引数の省略形
B( int = 1, int = 2 )
B( B const & )
B( B && )

```

さて、この継承コンストラクターの候補から、引数を取らないコンストラクター（デフォルトコンストラクター）、引数をひとつだけ取るコピー／ムーブコンストラクターを除くコンストラクターが継承コンストラクターとなる。これらのコンストラクターは、派生クラス側で暗黙に宣言されるものだからだ。ただし、派生クラスで同じシグネチャーのコンストラクターがユーザー定義されている場合は、継承されない。

いくつか例を示す。

```

struct Base
{
    Base( int ) { }
} ;

struct Derived : Base
{
    using Base::Base ;
} ;

```

この場合、クラスDerivedのコンストラクターは、以下のようになる。

```
Derived( ) // 繙承コンストラクターではない。使うとエラーになる
Derived( int ) // クラスBから継承されたコンストラクター
Derived( Derived const & ) // 繙承コンストラクターではない
Derived( Derived && ) // 繙承コンストラクターではない
```

デフォルトコンストラクタやコピー/ムーブコンストラクターは継承されないので、通常通りの挙動になる。この場合、クラスBaseのデフォルトコンストラクターは暗黙に宣言されていないし、ユーザー定義もされていないため、使うとエラーになる。

```
struct A
{
    A( int ) { }
} ;

struct B
{
    B( int ) { }
} ;

struct C : A, B
{
    using A::A ;
    using B::B ;
// エラー、宣言の重複
} ;

struct D : A, B
{
    using A::A ;
    using B::B ;
    D( int x ) : A(x), B(x) { } // OK、ユーザー定義を優先
}
```

クラスCでは、C(int)を重複して宣言してしまうので、エラーとなる。クラスDでは、ユーザー定義があるために、コンストラクターの継承は起こらない。もっとも、この場合、using宣言を使ってコンストラクターを継承する意味がない。

13 オーバーロード(Overloading)

関数と関数テンプレートは、異なる宣言であれば、同一スコープ内でも、同じ名前を使ふことができる。これを、オーバーロード(overload)という。オーバーロードされた名前が関数呼び出して使われた場合、オーバーロード解決(overload resolution)が行われ、最も最適な宣言が選ばれる。

```
void name( int ) ;
void name( double ) ;

int main()
{
    name( 0 ) ; // name(int)
    name( 0.0 ) ; // name(double)
}
```

これにより、引数が違うだけで本質的には同じ関数群に、それぞれ別名を付けなくてもよくなる。

13.1 オーバーロード可能な宣言(Overloadable declarations)

シグネチャが異なっていれば、どのような関数、あるいは関数テンプレートでもオーバーロードできるわけではない。以下は、オーバーロードでは考慮されないシグネチャ上の違いである。

- 戻り値の型

```
int f( int ) { return 0 ; }
double f( int ) { return 0.0 ; } // エラー、オーバーロード
できない
```

- メンバー関数とメンバー関数テンプレートにおいて、staticと非staticの違い

```
struct Foo
{
    void f() ;
    static void f() ; // エラー
} ;
```

- メンバー関数とメンバー関数テンプレートにおいて、リファレンス修飾子の有無が混在している場合

メンバー関数の暗黙のオブジェクト仮引数のリファレンスによるオーバーロードを行いたい場合は、lvalueリファレンスでも、リファレンス修飾子を省略することはできない。

```
struct Foo
{
    void f() ; // リファレンス修飾子の省略、暗黙にlvalueリフ
アレンス
    void f() && ; // エラー、他の宣言でリファレンス修飾子が
省略されている

    void g() & ; // OK
    void g() && ; // OK
};
```

- 仮引数の型が、同じ型を指す異なるtypedef名の場合

```
using Int = int ;

void f( int ) ;
void f( Int ) ; // 再宣言
```

typedef名は単なる別名であって、異なる型ではないので、シグネチャはおなじになる。

- 仮引数の型の違いが、*か[]である場合

8.3.5 関数の型で説明したように、仮引数のポインターと配列のシグネチャは同じである。ただし、2つ目以降の配列は考慮されるので注意。

```
void f( int * ) ;
void f( int [] ) ; // 再宣言、void f(int *)と同じ
void f( int [2] ) ; // 再宣言、void f(int *)と同じ

void f( int [][]2 ) ; // オーバーロード、シグネチャはvoid
f(int(*)[2])
```

- 仮引数が関数型か、同じ関数型へのポインターである場合

8.3.5 関数の型で説明したように、仮引数としての関数型は同じ関数型へのポインター型に変換される。

```
void f( void(*)() ) ;
void f( void () ) ; // 再宣言
void f( void g() ) ; // 再宣言
```

これらはオーバーロードではない。

- 仮引数のトップレベルのCV修飾子の有無

8.3.5 [関数の型](#)で説明したように、仮引数のトップレベルのCV修飾子は無視される。トップレベル以外のCV修飾子は別の型とみなされるので、オーバーロードとなる。

```
void f( int * ) ;
void f( int * const ) ; // 再宣言
void f( int * volatile ) ; // 再宣言
void f( int * const volatile ) ; // 再宣言

void f( int const * ) ; // オーバーロード
void f( int volatile * ) ; // オーバーロード
void f( int const volatile * ) ; // オーバーロード
```

- デフォルト実引数の違い

デフォルト実引数の違いは、オーバーロードとはみなされない。

```
void f( int, int ) ;
void f( int, int = 0 ) ; // 再宣言
void f( int = 0, int = 0 ) ; // 再宣言
```

13.2 オーバーロードのその他の注意事項

オーバーロード解決は、名前解決によって複数の宣言が列挙される場合に行われる。内側のスコープによって名前が隠されている場合は、オーバーロード解決は行われない。

たとえば、派生クラスで基本クラスのメンバー関数名と同名のものがある場合、そのメンバー関数は基本クラスのメンバー関数の名前を隠す。

```

struct Base
{
    void f( int ) { }
} ;

struct Derived : Base
{
    void f( double ) { } // Base::f(int)を隠す
} ;

int main()
{
    Derived d ;
    d.f( 0 ) ; // Derived::f(double)が呼ばれる
}

```

似たような例に、関数のローカル宣言がある。

```

void f( int ) { }
void f( double ) { }

int main()
{
    f( 0 ) ; // f(int)を呼び出す
    void f( double ) ; // f(int)を隠す
    f( 0 ) ; // f(double)を呼び出す
}

```

オーバーロードされたメンバー関数は、それぞれ別々のアクセス指定を持つことができる。アクセス指定は名前解決には影響ないので、オーバーロード解決は行われる。

```

class X
{
private :
    void f( int ) { }
public :
    void f( double ) { }

} ;

```

```
int main()
{
    X x ;
    x.f( 0 ) ; // エラー、X::f(int)はprivateメンバー
}
```

この例では、オーバーロード解決によって、X::f(int)が選ばれるが、これはprivateメンバーなので、Xのfriendではないmain関数からは呼び出せない。よってエラーになる。

13.3 オーバーロード解決(Overload resolution)

オーバーロードされた関数を呼び出す際に、実引数から判断して、最もふさわしい関数が選ばれる。これを、オーバーロード解決(Overload resolution)と呼ぶ。オーバーロード解決のルールは非常に複雑である。単純に実引数と仮引数の型が一致するだけならまだ話は簡単だ。

```
void f( int ) { }
void f( double ) { }

int main()
{
    f( 0 ) ; // f(int)が呼ばれる
    f( 0.0 ) ; // f(double)が呼ばれる
}
```

この結果には、疑問はない。実引数と仮引数の型が一致しているからだ。しかし、もし、実引数の型と仮引数の型が一致していないが、暗黙の型変換によって仮引数の型に変換可能な場合、問題は非常にややこしくなる。

```
void f( int ) { }
void f( double ) { }

int main()
{
    short a = 0 ;
    f( a ) ; // f(int)を呼ぶ

    float b = 0.0f ;
    f( b ) ; // f(double)を呼ぶ
}
```

この結果も、妥当なものである。shortは整数型なので、doubleよりはintを優先して欲しい。floatは、浮動小数点数型なので、doubleを優先して欲しい。

では、以下のような場合はどうだろうか。

```
void f( int ) { }
void f( long long ) { }
int main()
{
    long a = 0l ;
    f( a ) ; // 暫昧

    short b = 0 ;
    f( b ) ; // f(int)を呼び出す
}
```

この結果は、少し意外だ。比べるべき型は、intとlong long intである。long型を渡すと曖昧になる。しかし、short型を渡すと、なんとint型が選ばれる。こちらは曖昧にならない。これは、short型からint型への型変換に4.6 整数のプロモーションが使われているためである。

では、ユーザー定義の型変換が関係する場合はどうだろうか。

```
void f( int ) { }

class X
{
public :
    X() = default ;
    X( double ) { } // ユーザー定義の型変換
} ;

void f( X ) { }

int main()
{
    f( 0.0 ) ; // f(int)を呼ぶ
}
```

この場合、ユーザー定義の型変換より、言語側に組み込まれた、標準型変換を優先している。

では、引数が複数ある場合はどうなるのか。関数テンプレートの場合はどうなるのか。疑問は尽きない。オーバーロード解決のルールは非常に複雑である。これは、できるだけオーバーロード解決の挙動を、人間にとて自然にし、詳細を知らない問題がないように設計した結果である。その代償として、オーバーロード解決の詳細は非常に複雑になり、実装にも手間がかかるようになった。

オーバーロード解決の手順を、簡潔にまとめると、以下のようになる。

1. 名前探索によって見つかる同名の関数をすべて、候補関数(Candidate functions)として列挙する
2. 候補関数から、実際に呼び出すことが可能な関数を、適切関数(Viable functions)に絞る
3. 実引数から仮引数への暗黙の型変換を考慮して、最適な関数(Best viable function)を決定する

例えば、以下のようなオーバーロード解決の場合、

```

void f() { }
void f( int ) { }
void f( int, int ) { }
void f( double ) { }

void g( int ) { }

int main()
{
    f( 0 ) ; // オーバーロード解決が必要
}

```

候補関数には、f(), f(int), f(int,int), f(double)が列挙される。適切関数には、f(int), f(double)が選ばれる。これを比較すると、f(int)が型一致で最適関数となる。

本書におけるオーバーロード解決の解説は、細部をかなり省略している。

13.3.1 候補関数(Candidate functions)

候補関数(Candidate functions)は、正確に言えば、候補関数群とでも訳されるべきであろう。候補関数とは、その名前の通り、オーバーロード解決の際に呼び出しの優先順位を考慮される関数のことである。候補関数に選ばれなければ、呼び出されることはない。ある名前にに対してオーバーロード解決が必要な場合に、まず最初に行われるのが、候補関数の列挙である。候補関数は、通常通りに名前探索をおこなって見つけた関数すべてである。これには、実際には呼び出すことのできない関数も含む。オーバーロード解決の際に考慮するのは、この候補関数だけである。その他の関数は考慮しない。

```

void f() { }
void f( int ) { }
void g() { }

int main()
{
    f( 0 ) ; // 候補関数の列挙が必要
}

```

ここでの候補関数とは、f()とf(int)である。f()は、実際に呼び出すことができないが、候補関数として列挙される。この場合、g()は候補関数ではない。

オーバーロード解決の際に使われる名前探索は、通常の名前探索と何ら変わりないとすることに注意しなければならない。例えば、名前が隠されている場合は、発見されない。

```

void f( int ) { }
void f( double ) { }

int main()
{
    f( 0 ) ; // #1 f(int)
    void f( double ) ; // 再宣言、f(int)を隠す
    f( 0 ) ; // #2 f(double)
}

```

#1では、f(int)が名前探索で見つかるので、オーバーロード解決によって、f(int)が最適関数に選ばれる。#2では、f(int)は隠されているので、名前探索では見つからない。そのため、f(int)は候補関数にはならない。結果として、f(double)が最適関数に選ばれる。

関数のローカル宣言はまず使われないが、派生クラスのメンバー関数の宣言によって、基本クラスのメンバー関数が隠されることはある。

```

struct Base
{
    void f( int ) { }
    void f( long ) { }
} ;

struct Derived : Base
{
    void f( double ) { } // Baseクラスの名前fを隠す
    void g()
}

```

```

    {
        f( 0 ) ; // Derived::f(double)
    }
} ;

```

この例では、Derived::f(double)が、Baseのメンバー関数fを隠してしまうので、候補関数にはDerived::f(double)しか列挙されない。

候補関数がメンバー関数である場合、コード上には現れない仮引数として、クラスのオブジェクトを取る。これを、暗黙のオブジェクト仮引数(implicit object parameter)と呼ぶ。これは、オーバーロード解決の際に考慮される。暗黙のオブジェクト仮引数は、オーバーロード解決においては、関数の第一引数だとみなされる。暗黙のオブジェクト仮引数の型は、まず、クラスの型XにCV修飾子がつき、さらに、

リファレンス修飾子がない場合、あるいは、リファレンス修飾子が&の場合、X(場合によってCV修飾子)へのlvalueリファレンス。

```

struct X
{
    // コメントは暗黙のオブジェクト仮引数の型
    void f() & ; // X &
    void f() const & ; // X const &
    void f() volatile & ; // X volatile &
    void f() const volatile & ; // X const volatile &

    void g() ; // X &
} ;

```

リファレンス修飾子が&&の場合、X(場合によってCV修飾子)へのrvalueリファレンス。

```

struct X
{
    // コメントは暗黙のオブジェクト仮引数の型
    void f() && ; // X &&
    void f() const && ; // X const &&
    void f() volatile && ; // X volatile &&
    void f() const volatile && ; // X const volatile &&
} ;

```

となる。例えば、以下のようにオーバーロード解決に影響する。

```

struct X
{
    void f() & ; // #1 暗黙のオブジェクト仮引数の型は、X &
    void f() const & ; // #2 暗黙のオブジェクト仮引数の型は、X
const &
    void f() && ; // #3 暗黙のオブジェクト仮引数の型は、X &&
} ;

int main()
{
    X x ;
    x.f() ; // #1
    X const cx ;
    cx.f() ; // #2
    static_cast<X &&>(x).f() ; // #3
}

```

候補関数には、メンバー関数と非メンバー関数の両方を含むことがある。

```

struct X
{
    X operator + ( int ) const
    { return X() ; }
} ;

X operator + ( X const &, double )
{ return X() ; }

int main()
{
    X x ;
    x + 0 ; // X::operator+(int)
    x + 0.0 ; // operator+(X const &, double)
}

```

この場合、候補関数には、メンバー関数であるX::operator +と、非メンバー関数であるoperator+の両方が含まれる。候補関数に列挙されるので、当然、オーバーロード解決で最適関数が決定される。

テンプレートの実引数推定は、名前解決の際に行われる。そのため、候補関数として関数テンプレートのインスタンスが列挙された時点で、テンプレート実引数は決定されている。

オーバーロード解決が行われる文脈には、いくつか種類がある。それによって、候補関数の選び方も違ってくる。

13.3.1.1 関数呼び出しの文法(Function call syntax)

最も分かりやすい関数呼び出しは、関数呼び出しの文法によるものだろう。しかし、一口に関数呼び出しの文法といつても、微妙に違いがある。単なる関数名に対する関数呼び出し式の適用もあれば、クラスのオブジェクトに`.や->`を使った式に対する関数呼び出し、つまりメンバ関数の呼び出しや、クラスのオブジェクトに対する関数呼び出し式、つまりoperator ()のオーバーロードを呼び出すものがある。

```
struct X
{
    void f( int ) { }
    void f( double ) { }

    void operator () ( int ) { }
    void operator () ( double ) { }
} ;

int main()
{
    X x ;
    x.f( 0 ) ; // オーバーロード解決が必要
    x( 0 ) ; // オーバーロード解決が必要
}
```

オーバーロード解決は、関数へのポインター やリファレンスを経由した間接的な呼び出しの際には、行われない。

```
void f( int ) { }
void f( double ) { }

int main()
{
    void (* p)( int ) = &f ;
    p( 0.0 ) ; // f(int)
}
```

13.3.1.2 式中の演算子(Operators in expressions)

この項は、オーバーロードされた演算子を候補関数として見つける際の詳細である。演算子のオーバーロードの宣言方法については、13.7 オーバーロードされた演算子を参照。

演算子を使った場合にも、オーバーロード解決が必要になる。ただし、演算子にオーバーロード解決が行われる場合、オペランドにクラスやenumが関わっていなければならぬ。オペランドが基本型だけであれば、組み込みの演算子が使われる。

```
// エラー、オペランドがすべて基本型
int operator + (int, int) { return 0 ; }
```

演算子のオーバーロードは、メンバー関数としてオーバーロードする方法と、非メンバー関数としてオーバーロードする方法がある。すでに述べたように、候補関数には、どちらも列挙される。

演算子のオーバーロード関数は、演算子を仮に@と置くと、以下の表のように呼ばれる。

種類	式	メンバー関数として呼び出す場合	非メンバー関数として呼び出す場合
単項前置	@a	(a).operator@()	operator@(a)
単項後置	a@	(a).operator@(0)	operator@(a, 0)
二項	a@b	(a).operator@(b)	operator@(a, b)
代入	a=b	(a).operator=(b)	
添字	a[b]	(a).operator[](b)	
クラスメンバーアクセス	a->	(a).operator->()	

代入、添字、クラスメンバーアクセスの演算子は、メンバー関数として宣言しなければならないので、非メンバー関数は存在しない。

13.3.1.3 コンストラクターによる初期化(Initialization by constructor)

クラスのオブジェクトの直接初期化の場合、そのクラスからコンストラクターが候補関数

として列挙され、オーバーロード解決が行われる。

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

int main()
{
    X a( 0 ) ; // オーバーロード解決が行われる
    X b( 0.0 ) ; // オーバーロード解決が行われる
}
```

13.3.1.4 ユーザー定義型変換によるクラスのコピー初期化 (Copy-initialization of class by user-defined conversion)

クラスのコピー初期化におけるユーザー定義型変換には、オーバーロード解決が行われる。ユーザー定義型変換には、変換コンストラクターと変換関数がある。これは、両方とも、候補関数として列挙される。

```
struct Destination ;
extern Destination obj ;

struct Source
{
    operator Destination &() { return obj ; }
} ;

struct Destination
{
    Destination() { }
    Destination( Source const & ) { }
} ;

Destination obj ;

int main()
{
```

```

Source s ;
Destination d ;
d = s ; // オーバーロード解決、Source::operator
Destination &()
{
    Source const cs ;
    d = cs ; // オーバーロード解決、Destination::Destination(
    Source const & )
}

```

この例では、変換コンストラクターと変換関数の両方が候補関数として列挙される。この例で、もし変換コンストラクターの仮引数が、Source &ならば、オーバーロード解決は曖昧になる。

ただし、`explicit`変換コンストラクターと`explicit`変換関数は、直接初期化か、明示的なキャストが使われた際にしか候補関数にならない。

```

struct X
{
    X() { }
    explicit X( int ) { }
    explicit operator int() { return 0 ; }

} ;

int main()
{
    X x ;
    int a( x ) ; // OK
    int b = x ; // エラー

    X c( 0 ) ; // OK
    X d = 0 ; // エラー
}

```

この場合の実引数リストには、初期化式が使われる。変換コンストラクターの場合は、第一仮引数と比較され、変換関数の場合は、クラスの隠しオブジェクト仮引数と比較される。

```

// 変換コンストラクターの例
struct A { } ;

struct X

```

```
{
    // 候補関数
    X( A & ) { }
    X( A const & ) { }
} ;

int main()
{
    A a ;
    X x1 = a ; // オーバーロード解決、A::A(A&)
    A const ca ;
    X x2 = ca ; // オーバーロード解決、A::A(A const &)
}
```

この例では、実引数としてaやcaが使われ、クラスXの変換コンストラクターの第一仮引数と比較される。

```
// 変換関数の例
struct A { } ;

struct X
{
    // 候補関数
    operator A() & { return A() ; }
    operator A() const & { return A() ; }
    operator A() && { return A() ; }

} ;

int main()
{
    X x ;
    // オーバーロード解決、X::operator A() &
    // 実引数はlvalueのX
    A a1 = x ;
    X const cx ;
    // オーバーロード解決、X::operator A() const &
    // 実引数はconstなlvalue
    A a2 = cx ;
    // オーバーロード解決、X::operator A() &&
    // 実引数はxvalue
    A a3 = static_cast<X &&>(x) ;
}
```

この例では、クラスXのオブジェクトが実引数として、変換関数のクラスの隠しオブジェクト仮引数として比較される。たとえば、`A a1 = x ;` の場合、実引数は非constなlvalueなので、オーバーロード解決により、`X::operator A() &`が選ばれる。

その他の変換コンストラクターと変換関数に対しても、オーバーロード解決で比較する実引数と仮引数はこれに同じ。

13.3.1.5 変換関数によるクラスではないオブジェクトの初期化 (Initialization by conversion function)

クラスではないオブジェクトを、クラスのオブジェクトの初期化式で初期化する際、クラスの変換関数が候補関数として列挙され、オーバーロード解決が行われる。実引数リストには、初期化式がひとつの実引数として渡される。

```
struct X
{
    operator int() { return 0 ; }
    operator long() { return 0L ; }
    operator double() { return 0.0 ; }
} ;

int main()
{
    X x ;
    int i = x ; // オーバーロード解決が行われる
}
```

この例では、候補関数に、`X::operator int`、`X::operator long`、`X::operator double`が列挙され、オーバーロード解決によって`X::operator int`が選ばれる。

13.3.1.6 変換関数によるリファレンスの初期化(Initialization by conversion function for direct reference binding)

リファレンスを初期化するとき、初期化式に変換関数を適用して、その結果を束縛できる。このとき、クラスの変換関数が候補関数として列挙され、オーバーロード解決が行われる。

```

struct X
{
    operator int() { return 0 ; }
    operator short() { return 0 ; }
} ;

int main()
{
    X x ;
    int && ref = x ; // オーバーロード解決、X::operator int()
}

```

13.3.1.7 リスト初期化による初期化(Initialization by list-initialization)

8.5.4 アグリゲートではないクラスがリスト初期化によって初期化されるとき、オーバーロード解決によってコンストラクターが選択される。

この際の候補関数の列挙は、二段階に分かれている。

まず一段階に、クラスの初期化リストコンストラクターが候補関数として列挙され、オーバーロード解決が行われる。実引数リストには、初期化リストが唯一の実引数として、`std::initializer_list<T>`の形で、与えられる。

```

struct X
{
    // 初期化リストコンストラクター
    X( std::initializer_list<int> ) { }
    X( std::initializer_list<double> ) { }

    // その他のコンストラクター
    X( int, int, int ) { }
    X( double, double, double ) { }
} ;

int main()
{
    X a = { 1, 2, 3 } ; // オーバーロード解決、X::X(
    std::initializer_list<int> )
}

```

```
X b = { 1.0, 2.0, 3.0 } ; // オーバーロード解決、X::X(
std::initializer_list<double> )
}
```

この場合、候補関数には、初期化リストコンストラクターしか列挙されない。

もし、一段階目の名前解決で、13.4 適切な初期化リストコンストラクターが見つからなかった場合、二段階の候補関数として、再びオーバーロード解決が行われる。今度は、クラスのすべてのコンストラクターが候補関数として列挙される。実引数は、初期化リストの中の要素が、それぞれ別の実引数として渡される。

```
struct X
{
    // 適切な初期化リストコンストラクターなし

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

int main()
{
    X a = { 1, 2, 3 } ; // オーバーロード解決、X::X( int, int,
int )
    X b = { 1.0, 2.0, 3.0 } ; // オーバーロード解決、X::X(
double, double, double )
    X c = { 1, 2.0, 3 } ; // オーバーロード解決、X::X( int,
double, int )
}
```

「適切」という用語に注意すること。もし、8.5.7 縮小変換が必要となれば、適切関数かどうかを判定する前にエラーとなる。

```
struct X
{
    X( std::initializer_list<int> ) { }
    X( double, double, double ) { }
} ;

int main()
{
    X b = { 1.0, 2.0, 3.0 } ; // エラー、縮小変換が必要
```

}

デフォルトコンストラクターを持つクラスに空の初期化リストが渡された場合、一段階目のオーバーロード解決は行われず、デフォルトコンストラクターが呼ばれる。

```
struct X
{
    X( ) { }
    template < typename T >
    X( std::initializer_list<T> ) { }
} ;

int main()
{
    X x = { } ; // デフォルトコンストラクターが呼ばれる
}
```

コピーリスト初期化では、`explicit`コンストラクターが選ばれた場合、エラーとなる。

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

int main()
{
    X a = { 0 } ; // エラー、コピーリスト初期化でexplicitコンストラクター
    X b{ 0 } ; // OK、直接初期化
}
```

13.4 適切関数(Viable functions)

候補関数は、単に名前探索の結果であり、実際には呼び出すことができない関数も含まれている。このため、候補関数を列挙した後、呼び出すことが出来る関数、すなわち適切関数(Viable functions)を列挙する。

適切関数とは、与えられた実引数で、実際に呼び出すことが出来る関数である。これには、大きく二つの要素がある。仮引数の数と型である。

適切関数となるためにはまず、与えられた実引数の個数に対して、仮引数の個数が対応していなければならない。そのための条件は、以下のいずれかを満たしていればよい。

- 実引数の個数と、候補関数の仮引数の個数が一致する関数

これは簡単だ。実引数と同じ個数だけの仮引数があればよい。可変長テンプレートのインスタンス化による関数もこのうちに入る。

```
void f( int, int ) { }

int main()
{
    f( 0, 0 ) ; // OK
    f( 0 ) ; // エラー
}
```

- 候補関数の仮引数の個数が、実引数の個数より少ないが、仮引数リストにエリipsis(...)がある場合。

これは、C言語でお馴染みのことだ。可変長テンプレートは、このうちには入らない。

```
void f( int, ... ) ;

int main()
{
    f( 0 ) ; // 適切関数
    f( 0, 1 ) ; // 適切関数
    f( 0, 1, 2, 3, 4, 5 ) ; // 適切関数
}
```

- 候補関数の仮引数の個数は、実引数より多いが、実引数より多い仮引数にはすべて、デフォルト実引数が指定されていること。

```
void f( int, int = 0, int = 0, int = 0, int = 0, int = 0 )
;

int main()
{
```

```
f( 0 ) ; // 適切関数
f( 0, 1 ) ; // 適切関数
f( 0, 1, 2, 3, 4, 5 ) ; // 適切関数
}
```

さらに、対応する実引数から仮引数に対して、後述する暗黙の型変換により、妥当な変換が存在しなければならない。

```
void f( int ) { }

int main()
{
    f( 0 ) ; // OK、完全一致
    f( 0L ) ; // OK、整数変換
    f( 0.0 ) ; // OK、整数と浮動小数点数間の変換
    f( &f ) ; // エラー
}
```

適切関数であるからといって、実際に呼び出せるとは限らない。たとえば、宣言されているが未定義であったり、アクセス指定による制限を受けたり、あるいはその他実装依存の理由など、現実には呼び出すことができない理由は多数存在する。

13.5 最適関数(Best viable function)

適切関数が複数ある場合、定められた方法で関数を比較することによって、ひとつの最も適切(best viable)な関数を選択する。この関数を最適関数と呼ぶ。オーバーロード解決の結果は、この最適関数となる。もし、最も適切な関数をひとつに決定できない場合、オーバーロード解決は曖昧であり、エラーとなる。

最適関数の決定は、主に、後述する暗黙の型変換の優先順位によって決定される。

まず大前提として、ある関数が、別の関数よりも、より適切であると判断されるには、ある関数のすべて仮引数に対する実引数からの暗黙の型変換の優先順位が劣っておらず、かつ、ひとつ以上の優れている型変換が存在しなければならない。

```
void f( int, double ) { } // #1
void f( long, int ) { } // #2

int main()
{
    f( 0 , 0 ) ; // エラー、オーバーロード解決が曖昧
```

}

この例では、どの関数も、仮引数への型変換の優先順位が、他の関数より劣っている。したがってオーバーロード解決は曖昧となる。一見すると、#2の方が、どちらも整数型であるので、よりよい候補なのではないかと思うかもしれない。しかし、#1の第一仮引数の型はintなので、longよりも優れている。一方、第二引数では、#2の方が優れている。このため、曖昧となる。最適関数となるためには、全ての仮引数の型が、他の候補より劣っていてはならないのだ。

ユーザー定義型変換による初期化の場合、ユーザー定義型変換の結果の型から、目的の型へ、標準型変換により変換する際、より優先順位の高いものが選ばれる。

```
struct X
{
    operator int() ;
    operator double() ;

} ;

void f()
{
    X x ;
    int i = x ; // operator intが最適関数
    float f = x ; // エラー、曖昧
}
```

一見すると、doubleからfloatへの変換は、intからの変換より優先順位が高いのではないかと思うかもしれないが、後述する標準型変換の優先順位のルールにより、同じ優先順位なので、曖昧となる。

非テンプレート関数と関数テンプレートの特殊化では、非テンプレート関数が優先される。

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

int main()
{
    f( 0 ) ; // 非テンプレート関数を優先
}
```

もちろん、これは大前提の、すべての仮引数に対し劣った型変換がないということが成

り立つまでの話である。

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

int main()
{
    f( 0 ) ; // 関数テンプレートの特殊化f<int>を優先
}
```

この場合は、テンプレートの特殊化である仮引数int型の方が、実引数int型に対して、より優れた型変換なので、優先される。

テンプレートの実引数推定のルールは複雑なので、一見して、非テンプレート関数が優先されると思われるコードで、関数テンプレートの実体化の方が優先される場合がある。

```
// #1
// 非テンプレート関数
void f( int const & ) ;

// #2
// 関数テンプレート
template < typename T >
void f( T && ) ;

int main()
{
    int x = 0 ; // xは非constなlvalue
    f( x ) ; // #2を呼ぶ
}
```

これは、#2の実体化の結果が、`f<int &>(int &)`になるからだ。`x`は非constなlvalueであるので、非constなlvalueリファレンス型の仮引数と取る#2の方が優先される。

ふたつの関数が両方ともテンプレートの特殊化の場合、14.4.8 半順序によって、より特
殊化されていると判断される方が、優先される。

```
template < typename T > void f( T ) ; // #1
template < typename T > void f( T * ) ; // #2
```

```
int main()
{
    int * ptr = nullptr ;
    f( ptr ) ; // 半順序により#2を優先
}
```

#1と#2の特殊化による仮引数の型は、どちらも `int *` であるが、#2のテンプレートの特殊化の方が、半順序のルールによって、より特殊化されているとみなされるため、#2が優先される。

13.5.1 暗黙の型変換の順序(Implicit conversion sequences)

暗黙の型変換には、いくつかの種類と、多数の例外ルールがあり、それぞれ優先順位を比較することができる。残念ながら、この詳細は非常に冗長であり、本書では概略の説明に留める。

まず、暗黙の型変換には、大別して三種類ある。[4 標準型変換](#)、[12.3 ユーザー定義型変換](#)、エリプシス変換である。優先順位もこの並びである。標準型変換が一番優先され、次にユーザー定義型変換、最後にエリプシス変換となる。

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

void f( long ) ; // #1
void f( X ) ; // #2

void g( X ) ; // #3
void g( ... ) ; // #4

int main()
{
    f( 0 ) ; // #1、標準型変換がユーザー定義型変換に優先される
    g( 0 ) ; // #3、ユーザー定義型変換がエリプシス変換に優先され
    る
}
```

さらに、標準型変換とユーザー定義変換同士の間での優先順位がある。

エリプシスに基づき型以外を渡して呼び出した場合の挙動は未定義だが、オーバーロード解決には影響しない。

13.5.1.1 標準型変換(Standard conversion sequences)

オーバーロード解決における標準型変換の間の優先順位は、非常に複雑で、単に、ランクA>ランクBのような単純な比較ができない。ここでは、とくに問題になりそうな部分のみ取り上げる。

まず、型変換の必要のない、完全一致が最も優先される。

```
void f( int ) ;
void f( double ) ;

int main()
{
    f( 0 ) ; // f(int)
    f( 0.0 ) ; // f(double)
}
```

この完全一致には、4.1 [lvalueからrvalueへの型変換](#)、4.2 [配列からポインターへの型変換](#)、4.3 [関数からポインターへの型変換](#)が含まれる。

```
void f( int ) ;

int main()
{
    int x = 0 ;
    f( x ) ; // lvalueからrvalueへの変換
}
```

配列や関数からポインターへの変換は、完全一致とみなされることに注意。

```
void g( ) ;

void f( void (*)() ) ; // ポインター
void f( void (&)() ) ; // リファレンス

int main()
{
    f( g ) ; // エラー、オーバーロード解決が曖昧、候補関数はすべて完全一致
    f( &g ) ; // OK、f( void (*)() )
```

}

完全一致は、ポインター や リファレンス に 4.4 CV 修飾子 を 付け 加える 型 変換 より 優先 さ れる。

```
void f( int & ) ; // #1
void f( int const & ) ; // #2

int main()
{
    int x = 0 ;
    f( x ) ; // #1、完全一致
}
```

整数と浮動小数点数のプロモーションは、その他の整数と浮動小数点数への変換より優先される。

```
void f( int ) ;
void f( long ) ;

int main()
{
    short x = 0 ;
    f( x ) ; // f(int)、プロモーション
}
```

13.6 オーバーロード関数のアドレス(Address of overloaded function)

ある関数の名前に対して、複数の候補関数がある場合でも、名前から関数のアドレスを取得できる。どの候補関数を選ぶかは、文脈が期待する型の完全一致で決定される。初期化や代入、関数呼び出しの実引数や明示的なキャストの他に、関数の戻り値も、文脈により決定される。

```
void f( int ) ;
void f( long ) ;

void g( void ( * )(int) ) ;
```

```

void h()
{
    // 初期化
    void (*p)(int) = &f ; // void f(int)のアドレス
    // 代入
    p = &f ; // void f(int)のアドレス
    // 関数呼び出しの実引数
    g( &f ) ;
    // 明示的なキャスト
    static_cast<void (*)(int)>(&f) ; // void f(int)のアドレス
}

// 関数の戻り値
auto i() -> void (*) (int)
{
    return &f ; // void f(int)のアドレス
}

```

これらの文脈では、ある具体的な完全一致の型を期待しているので、オーバーロードされた関数名から、適切な関数を決定できる。

完全一致の型ではない場合や、型を決定できない場合はエラーである。

```

void f( int ) ;
void f( long ) ;

template < typename T >
void g( T ) { }

int main()
{
    g( &f ) ; // エラー
}

```

13.7 オーバーロード演算子(Overloaded operators)

特別な識別子を使っている関数宣言は、演算子関数(operator function)として認識される。この識別子は以下のようになる。

operator 演算子

オーバーロード可能な演算子は以下の通りである。

```
new      delete   new[]    delete[]
+       -       *       /       %       ^       &       |       ~
!       =       <       >       +=      -=      *=      /=      %= 
^=      &=      |=      <<      >>     >>=     <<=     ==      != 
<=      >=      &&      ||      ++      --      ,       ->*   ->
( ) [ ]
```

以下の演算子は、単項、二項の両方でオーバーロードできる。

```
+       -       *       &
```

以下の演算子は、関数呼び出しと添え字である。

```
( ) [ ]
```

以下の演算子は、オーバーロードできない。

```
.       .*      ::      ?:
```

new, new[], delete, delete[]については、[確保関数](#)と[3.7.2 解放関数](#)も参照。

演算子関数は、非staticメンバー関数か、非メンバー関数でなければならない。非staticメンバー関数の場合、暗黙のオブジェクト仮引数が、第一オペランドになる。これが*thisである。非メンバー関数の場合、仮引数のひとつは、クラスか、クラスへのリファレンス、enumかenumへのリファレンスでなければならない。

```
struct X
{
    // 非staticメンバー関数による演算子関数
    X operator +( ) const ; // 暗黙のオブジェクト仮引数 X const
    &
    X operator +( int ) const ; // 暗黙のオブジェクト仮引数 X
```

```
const &
} ;

// 非メンバー関数による演算子関数
X operator -( X const & ) ;
X operator -( X const &, int ) ;
X operator -( int, X const & ) ;
```

以下の例はエラーである。

```
// エラー、組み込みの演算子をオーバーロードできない
int operator +( int, int ) ;

struct X { } ;
// エラー、組み込みの演算子をオーバーロードできない
X operator + ( X * ) ;
```

ただし、代入演算子や添字演算子のように、非staticメンバー関数として実装しなければならない例外的な演算子もある。

演算子関数は、必ず元の演算子と同じ数の仮引数を取らなければならない。

```
struct X { } ;

X operator / ( X & ) ; // エラー、仮引数が少ない
X operator / ( X &, X &, X & ) ; // エラー、仮引数が多い
```

ただし、これも関数呼び出し演算子のように、例外的な演算子がある。

演算子関数は、組み込みの演算子と同じ挙動を守らなくてもよい。例えば、戻り値の型は自由であるし、オーバーロードされた演算子関数が、基本型にその単項演算子を適用した場合に期待される挙動をしなくてもかまわない。例えば、オーバーロードした演算子関数では、“`++a`”、と、“`a += 1`”というふたつの式を評価した際の挙動や結果が同じにならなくてもよい。また、組み込み演算子ならば非`const`な`lvalue`を渡す演算子で、`const`な`lvalue`や`rvalue`を受け取っても構わない。

```
struct X { } ;

void operator + ( X & ) ; // OK、戻り値の型は自由
void operator ++ ( X const & ) ; // OK、constなlvalueリファレンスでもよい
```

演算子関数は、通常通り演算子を使うことによって呼び出すことができる。その際、演算子の優先順位は、組み込みの演算子と変わらない。また、識別子を指定することによって、通常の関数呼び出し式の文法で、明示的に呼び出すこともできる。

```
struct X
{
    X operator +( X const & ) const ;
    X operator *( X const & ) const ;
} ;

int main()
{
    X a ; X b ; X c ;
    a + b ; // 演算子を使うことによる呼び出し
    a + b * c ; // 優先順位は、(a + (b * c))

    a.operator +(b) ; // 明示的な関数呼び出し
}
```

代入演算子=や、単項演算子の&や、カンマ演算子は、オーバーロードしなくてもすべての型に対してあらかじめ定義された挙動がある。この挙動はオーバーロードして変えることもできる。

13.7.1 単項演算子(Unary operators)

オーバーロード可能な単項演算子は、以下の通りである。

```
+ - * & ~ !
```

ここでは、*&は単項演算子であることに注意。13.7.2 [二項演算子](#)の項も参照。

インクリメント演算子とデクリメント演算子については、13.7.7 [インクリメントとデクリメント](#)を参照。

単項演算子は、演算子を@とおくと、@xという式は、非staticメンバー関数の場合、x.operator @()、非メンバー関数の場合、operator @(x)として呼び出される。単項演算子では、非staticメンバー関数と非メンバー関数は、機能的に違いはない。

```
struct X
{
```

```

    void operator + () ;
} ;

void operator -( X & ) ;

int main()
{
    X x ;
    +x ; // x.operator + ()
    -x ; // operator - (x)
}

```

非staticメンバー関数の場合、明示的に仮引数をとらない。暗黙のオブジェクトが仮引数として渡される。

```

struct X
{
    void operator + () & ;
    void operator + () const & ;
    void operator + () volatile & ;
    void operator + () const volatile & ;

    void operator + () && ;
    void operator + () const && ;
    void operator + () volatile && ;
    void operator + () const volatile && ;
} ;

int main()
{
    X x ;
    +x ; // void operator + () &
    +static_cast<X &&>(x) ; // void operator + () &&

    X const cx ;
    +x ; // void operator + () const &
}

```

同様のコードを、非メンバー関数として書くと、以下のようになる。

```
struct X { } ;
```

```

void operator + ( X & ) ;
void operator + ( X const & ) ;
void operator + ( X volatile & ) ;
void operator + ( X const volatile & ) ;

void operator + ( X && ) ;
void operator + ( X const && ) ;
void operator + ( X volatile && ) ;
void operator + ( X const volatile && ) ;

int main()
{
    X x ;
    +x ; // void operator + ( X & )
    +static_cast<X &&>(x) ; // void operator + ( X && )

    X const cx ;
    +x ; // void operator + ( X const & )
}

```

また、非メンバー関数の場合は、クラス型を引数に取ることができる。

```

struct X { } ;
void operator + ( X ) ;

```

operator &には、注意を要する。これは、組み込みの演算子、すなわち、オペランドのアドレスを得る演算子として、すべての型にあらかじめ定義されている。

```

// operator &のオーバーロードなし
struct X { } ;

int main()
{
    X x ;
    X * ptr = &x ; // 組み込みのoperator &の呼び出し
}

```

この演算子をオーバーロードすると、組み込みのoperator &が働かなくなる。

```

struct X
{
    X * operator &() { return nullptr ; }
} ;

int main()
{
    X x ;
    X * ptr = &x ; // 常にnullポインターになる。
}

```

もちろん、戻り値の型は自由だから、なにか別のことをさせるのも可能だ。

```

class int_wrapper
{
private :
    int obj ;
public :
    int * operator &() { return &obj ; }
} ;

int main()
{
    int_wrapper wrap ;
    int * ptr = &wrap ;
}

```

ただし、クラスのユーザーが、オブジェクトのアドレスを得たい場合、組み込みの演算子を呼び出すのは簡単ではない。そのため、標準ライブラリヘッダー<memory>には、`std::addressof`という関数テンプレートが定義されている。これを使えば、`operator &`がオーバーロードされているクラスでも、クラスのオブジェクトのアドレスを得ることができる。

```

struct X
{
    void operator &() { }
} ;

int main()
{
    X x ;
    X * p1 = &x ; // エラー、operator &の戻り値の型はvoid
}

```

```
X * ptr = std::addressof(x) ; // OK
}
```

13.7.2 二項演算子(Binary operators)

オーバーロード可能な二項演算子は以下の通りである。

+	-	*	/	%	^	&		~
!	<	>	+=	-=	*=	/=	%=	
^=	&=	=	<<	>>	>>=	<<=	==	!=
<=	>=	&&		,				

代入演算子は特別な扱いを受ける。詳しくは、13.7.3 [代入演算子](#)を参照。複合代入演算子は、二項演算子に含まれる。

二項演算子は、演算子を@とおくと、x@yという式に対して、非staticメンバー関数の場合、x.operator @(y)、非メンバー関数の場合、operator @(x,y)のように呼び出される。

```
struct X
{
    void operator + (int) const ;
} ;

void operator - ( X const &, int ) ;

int main()
{
    X x ;
    x + 1 ; // x.operator +(1)
    x - 1 ; // operator -(x, 1)
}
```

非staticメンバー関数の場合、第一オペランドが暗黙のオブジェクト仮引数に、第二オペランドが実引数に渡される。

```
struct X
{
```

```
void operator + (int) & ;
void operator + (int) const & ;
void operator + (int) volatile & ;
void operator + (int) const volatile & ;

void operator + (int) && ;
void operator + (int) const && ;
void operator + (int) volatile && ;
void operator + (int) const volatile && ;
} ;

int main()
{
    X x ;
    x + 1 ; // X::operator + (int) &
    static_cast<X &&>(x) + 1 ; // X::operator + (int) &&
    X const cx ;
    cx + 1 ; // X::operator + (int) const &
}
```

同様のコードを、非メンバー関数で書くと以下のようになる。

```
struct X { } ;

void operator + ( X &, int) ;
void operator + ( X const &, int) ;
void operator + ( X volatile &, int) ;
void operator + ( X const volatile &, int) ;

void operator + ( X &&, int) ;
void operator + ( X const &&, int) ;
void operator + ( X volatile &&, int) ;
void operator + ( X const volatile &&, int) ;

int main()
{
    X x ;
    x + 1 ; // operator + ( X &, int)
    static_cast<X &&>(x) + 1 ; // operator + ( X &&, int)
    X const cx ;
    cx + 1 ; // operator + ( X const &, int)
}
```

非メンバー関数の場合は、クラス型を仮引数に取ることができる。

```
struct X { } ;
void operator + ( X, int ) ;
```

第二オペランドにクラスやenum型、あるいはそのリファレンス型を取りたい場合は、非メンバー関数しか使えない。

```
struct X { } ;

void operator + ( int, X & ) ;

int main()
{
    X x ;
    1 + x ;
}
```

メンバー関数によるオーバーロードでは、必ず第一オペランドのメンバーとして演算子関数が呼ばれるので、これはできない。

カンマ演算子、operator ,には、あらかじめ定義された組み込みの演算子が存在する。オーバロードにより、この挙動を変えることもできる。ただし、operator ,の挙動を変えるのは、ユーザーを混乱させるので、慎むべきである。もし、単に任意個の引数を取りたいというのであれば、可変長テンプレートや初期化リストなどの便利な機能が他にもある。

13.7.3 代入(Assignment)

代入演算子のオーバーロードは、仮引数をひとつとる非staticメンバー関数として実装する。非メンバー関数として実装することはできない。複合代入演算子は、代入演算子ではなく、二項演算子である。

```
struct X
{
    // コピーアイ代入演算子
    X & operator = ( X const & ) ;
    // ムーブ代入演算子
    X & operator = ( X && ) ;
```

```

// intからの代入演算子
X & operator = ( int ) ;
} ;

// エラー、非メンバー関数として宣言することはできない
X & operator = ( X &, double ) ;

// OK、複合代入演算子は二項演算子
X & operator += ( X &, double ) ;

```

もちろん、戻り値の型は自由である。ただし、慣例として、暗黙に定義される代入演算子は、`*this`を返すようになっている。詳しくは、12.8 [クラスオブジェクトのコピーとムーブ](#)を参照。

13.7.4 関数呼び出し(Function call)

関数呼び出し演算子の識別子は、`operator ()`である。関数呼び出し演算子のオーバーロードは、任意個の仮引数を持つ非staticメンバー関数として宣言する。非メンバー関数として宣言することはできない。デフォルト実引数も使うことができる。

関数呼び出し演算子は、`x(arg1, ...)`とおくと、`x.operator()(arg1, ...)`のように呼び出される。

```

struct X
{
    void operator () ( ) ;
    void operator () ( int ) ;
    void operator () ( int, int, int = 0 ) ;
} ;

int main()
{
    X x ;
    x() ; // x.operator () ( )
    x( 0 ) ; // x.operator () ( 0 )
    x( 1, 2 ) ; // x.operator() ( 1, 2 )
}

```

13.7.5 添字(Subscripting)

添字演算子の識別子は、operator []である。添字演算子のオーバーロードは、ひとつ
の仮引数を持つ非staticメンバー関数として宣言する。非メンバー関数として宣言する
ことはできない。

添字演算子は、x[y]とおくと、x.operator [] (y)のように呼び出される。

```
struct X
{
    void operator [] ( int ) ;
} ;

int main()
{
    X x ;
    x[1] ; // x.operator [] (1)
}
```

添字演算子に複数の実引数を渡すことはできない。ただし、初期化リストならば渡すこ
とができる。

```
struct X
{
    void operator [] ( std::initializer_list<int> list ) ;
} ;

int main()
{
    X x ;
    x[ { 1, 2, 3 } ] ;
}
```

13.7.6 クラスメンバーアクセス(Class member access)

クラスメンバーアクセス演算子の識別子は、operator ->である。クラスメンバーアクセ
ス演算子は仮引数を取らない非staticメンバー関数として宣言する。非メンバー関数に
することはできない。クラスメンバーアクセス演算子は、後述するように、少し変わった
特徴がある。

クラスメンバーアクセス演算子は、`x->m`とおくと、`(x.operator->())->m`のように呼び出される。つまり、もし、`x.operator->()`の戻り値の型がクラスへのポインターであれば、そのまま組み込みのクラスメンバーアクセス演算子が使われる。それ以外の場合は、戻り値に対してクラスメンバーアクセス演算子を適用しているために、さらに戻り値のクラスメンバーアクセス演算子が、もし存在すれば、呼び出される。

```

struct A
{
    int member ;
} ;

struct B
{
    A a ;
    A * operator ->() { return &a ; }
} ;

struct C
{
    B b ;
    B & operator ->() { return b ; }
} ;

int main()
{
    B b ;
    b->member ; // (b.operator ->())->member

    C c ;
    // (c.operator ->())->member
    // すなわちこの場合、以下のように展開される。
    // ((c.operator ->()).operator ->())->member
    c->member ;
}

```

クラスBは、

クラスCの`operator ->`が`B &`型を返していることに注目。`b->member`の`B`にクラスメンバーアクセス演算子である`->`が使われるため、クラスBのクラスメンバーアクセス演算子が呼ばれる。

クラスメンバーアクセス演算子の評価の結果に対するクラスメンバーアクセス演算子の呼び出しは、際限なく行われる。このループを断ち切るには、最終的にクラスへのポインターを返し、組み込みのクラスメンバーアクセス演算子を使わなければならない。

もちろん、これは演算子として使用した場合であって、明示的に関数を呼び出す場合には、通常通り、その関数だけが呼ばれる。もちろん、戻り値の型をvoid型にすることもできる。

```
struct X
{
    void operator ->() { }
} ;

int main()
{
    X x ;
    x.operator ->() ; // OK
}
```

13.7.7 インクリメントとデクリメント(Increment and decrement)

インクリメント演算子の識別子はoperator ++、デクリメント演算子の識別子はoperator --である。インクリメント演算子とデクリメントの演算子は非staticメンバー関数と、非メンバー関数の両方で宣言できる。インクリメント演算子とデクリメント演算子は、識別子の違いを除けば、同じように動く。ここでのサンプルコードは、インクリメント演算子の識別子を使う。

インクリメントとデクリメントには、前置と後置の違いがある。

```
++a ; // 前置
a++ ; // 後置
```

前置演算子は、非staticメンバー関数の場合、仮引数を取らない。非メンバー関数の場合は、ひとつの仮引数を取る。

前置演算子は、`++x`という式に対して、非staticメンバー関数の場合、`x.operator ++ ()`、非メンバー関数の場合、`operator ++(x)`のように呼び出される。

```
struct X
{ // 非staticメンバー関数の例
    void operator ++ () ;
} ;
```

```

struct Y { } ;
// 非メンバー関数の例
void operator ++ ( Y & ) ;

int main()
{
    X x ;
    ++x ; // x.operator ++()

    Y y ;
    ++y ; // operator ++(y)
}

```

後置演算子は、非staticメンバー関数の場合、int型の引数を取る。非メンバー関数の場合は、二つの仮引数を取る。第二仮引数の型はintでなければならない。int型の仮引数は、単に前置と後置を別の宣言にするためのタグであり、それ以上の意味はない。式としてインクリメントとデクリメントを使うと、実引数には0が渡される。

後置演算子は、`x++`という式に対して、非staticメンバー関数の場合、`x.operator ++(0)`、非メンバー関数の場合、`operator ++ (x, 0)`のように呼び出される。

```

struct X
{ // 非staticメンバー関数の例
    void operator ++ (int) ;
}

struct Y { } ;
// 非メンバー関数の例
void operator ++ ( Y & , int ) ;

int main()
{
    X x ;
    x++ ; // x.operator ++( 0 )

    Y y ;
    y++ ; // operator ++( y, 0 )
}

```

intをタグとして使うこの仕様はすこし汚いが、例外的な文法を使わなくてもよいという利点があるので採用された。もし明示的に呼び出した場合は、int型の仮引数に対し、0以

外の実引数を与えることもできる。

13.7.8 確保関数と解放関数(allocation function and deallocation function)

ここでは確保関数と3.7.2 解放関数のオーバーロードについて解説している。

注意: 本来、これはコア言語ではなくライブラリで規定されていることなので、本書の範疇ではないのだが、ここでは読者の便宜のため、宣言方法と、デフォルトの挙動のリファレンス実装を提示する。また、サンプルコードは分割して掲載しているが、確保関数と解放関数はそれぞれ関係しており、すべて一つのソースファイルに含まれることを想定している。そのため、ヘッダーファイルのincludeは最初のサンプルコードにしか書いていない。

確保関数の識別子はoperator newである。解放関数の識別子はoperator deleteである。この関数は、動的ストレージの確保と解放を行う。確保関数と解放関数が行うのは、生の動的ストレージの確保と解放である。よく誤解があるが、コンストラクタやデストラクターの呼び出しの責任は持たない。

確保関数と解放関数のオーバーロードは、グローバル名前空間か、クラスのメンバー関数として宣言する。グローバル名前空間以外の名前空間で宣言するとエラーとなる。確保関数と解放関数がユーザー定義されない場合、実装によってデフォルトの挙動を行う確保関数と解放関数が自動的に定義される。

```
// グローバル名前空間
void* operator new(std::size_t size) ; // OK

namespace NS
{
void* operator new(std::size_t size); // エラー、グローバル名前
空間ではない
}

struct X
{
    void* operator new(std::size_t size) ; // OK
};
```

グローバル名前空間の宣言は、デフォルトの確保関数と解放関数の生成を妨げる。クラスのメンバー関数は、そのクラスと派生クラスの確保と解放に使われる。

確保関数には、効果(effect)と必須の挙動(required behavior)とデフォルトの挙動(default behavior)が規定されている。解放関数には、効果とデフォルトの挙動が規定されている。効果とは、その関数がどのように使われるのかという規定である。必

須の挙動とは、たとえユーザー定義の関数であっても必ず守らなければならない挙動のことである。デフォルトの挙動とは、関数がユーザー定義されていない場合、実装によって用意される定義の挙動である。

C++11ではスレッドの概念が入ったので、確保関数と解放関数は、データ競合を引き起こしてはならない。この保証は、ユーザー定義の確保関数と解放関数にも要求される。

C++11ではアライメントの概念が入ったので、確保関数の確保するストレージは、要求されたサイズ以下の大きさのオブジェクトを配置できるよう、適切にアラインされなければならない。

単数形の確保関数

```
void* operator new(std::size_t size) ;
```

効果

この確保関数は、new式からsizeバイトのストレージを確保するために呼ばれる。

必須の挙動

適切にアラインされたストレージを指示するnullポインターではない値を返す。もししくは、std::bad_exceptionがthrowされる。

デフォルトの挙動

- ループを実行する。ループの中で、まず要求されたストレージの確保を試みる。ストレージ確保の方法は実装依存である。
- ストレージの確保が成功したならば、ストレージへのポインターを返す。ストレージの確保が成功しなかった場合で、現在のnew_handlerがnullポインターの場合、std::bad_allocをthrowする。
- 現在のnew_handlerがnullポインター以外の場合、現在のnew_handlerを呼び出す。呼び出しが返ったならば、ループを続行する。
- ループはストレージの確保が成功するか、new_handlerの呼び出しが返らなくなるまで、続けられる。

```
#include <cstddef>
#include <cstdlib>
#include <new>

void* operator new( std::size_t size )
{
    // std::mallocに実引数0を渡した場合の挙動は定義されていない
    if ( size == 0 ) { size = 1 ; }
```

```

while ( true ) // ループを実行する
{
    // ループの中で、要求されたストレージの確保を試みる
    void * ptr = std::malloc( size ) ;

    // ストレージの確保が成功したならば
    if ( ptr != nullptr )
    { // ストレージへのポインターを返す
        return ptr ;
    }

    // ストレージの確保が成功しなかった場合
    std::new_handler handler = std::get_new_handler()
;
    if ( handler == nullptr ) // 現在のnew_handlerが
nullptrポインターの場合
    { // std::bad_allocをthrowする。
        throw std::bad_alloc() ;
    } else // 現在のnew_handlerがnullptrではない
場合
    { // 現在のnew_handlerを呼び出す
        handler( ) ;
    }
    // ループを続行する
}
}

```

nothrow版の单数形の確保関数

```

void * operator new( std::size_t size, const std::nothrow_t &
) noexcept ;

```

効果

前項の確保関数と同じ。ただし、エラー報告としてstd::bad_allocをthrowする代わりに、nullptrポインターを返す。

必須の挙動

適切にアラインされたストレージへのポインターを返すか、nullptrポインターを返す。この関数の返すポインター、nothrowではない確保関数を呼び出してストレージを確保した場合と同じポインターを返す。

デフォルトの挙動

`operator new(size)`を呼び出す。呼び出しが通常通り返れば、その戻り値を返す。それ以外の場合、`null`ポインターを返す。

```
void* operator new( std::size_t size, const
std::nothrow_t & ) noexcept
{
    try
    { // operator new(size)を呼び出す
        // 呼び出しが通常通り返れば、その戻り値を返す
        return operator new( size );
    }
    catch ( ... )
    { // それ以外の場合、nullポインターを返す。
        return nullptr;
    }
}
```

単数形の解放関数

```
void operator delete( void * ptr ) noexcept ;
```

効果

この解放関数は、`ptr`の値を無効にするため、`delete`式から呼ばれる。`ptr`の値は、`null`ポインターか、`operator new(std::size_t)`もしくは `operator new(std::size_t,const std::nothrow_t&)`によって返された値で、まだ`operator delete(void*)`に渡していないものである。

デフォルトの挙動

`ptr`の値が`null`ポインターであれば、なにもしない。それ以外の場合、先の`operator new`の呼び出しで確保されたストレージを解放する。

```
void operator delete( void* ptr ) noexcept
{
    std::free( ptr );
}
```

nothrow版の单数形の解放関数

```
void operator delete( void * ptr, const std::nothrow_t & )
noexcept ;
```

効果

nothrow版のnew式によって呼び出されたコンストラクターが例外を投げた時に、ストレージを解放するために呼ばれる。delete式では呼ばれない。

```
struct X
{
    X() { throw 0 ; }
} ;

int main()
{
    new(std::nothrow) X ; // operator delete( void *,
std::nothrow_t &)が呼ばれる
}
```

デフォルトの挙動

operator delete(ptr)を呼び出す。

```
void operator delete( void* ptr, const std::nothrow_t & )
noexcept
{
    operator delete( ptr ) ;
}
```

配列形の確保関数

```
void * operator new[]( std::size_t size ) ;
```

効果

この確保関数は配列形のnew式からsizeバイトのストレージを確保するために呼ばれる。

必須の挙動

単数形の確保関数と同じ。

デフォルトの挙動

operator new(size)を返す。

```
void * operator new[]( std::size_t size )
{
    return operator new( size ) ;
}
```

nothrow版の配列形の確保関数

```
void * operator new[]( std::size_t size, const std::nothrow_t & ) noexcept ;
```

効果

この確保関数は、nothrow版のnew式から呼ばれる。エラー報告として、std::bad_allocをthrowする代わりに、nullポインターを返す。

必須の挙動

適切にアラインされたストレージへのポインターを返すか、nullポインターを返す。

デフォルトの挙動

operator new[](size)を呼び出す。呼び出しが通常通り返れば、その結果を返す。それ以外の場合は、nullポインターを返す。

```
void * operator new[]( std::size_t size, const std::nothrow_t & ) noexcept
{
    try
    {
        return operator new[]( size ) ;
    }
```

```

    catch ( ... )
    {
        return nullptr ;
    }
}

```

配列型の解放関数

```
void operator delete[]( void * ptr ) noexcept ;
```

効果

この解放関数は、配列型のdelete式から、ptrの値を無効にするために呼ばれる。

デフォルトの挙動

operator delete(ptr)を呼ぶ。

```

void operator delete[]( void * ptr ) noexcept
{
    operator delete( ptr ) ;
}

```

nothrow版の配列型の解放関数

```
void operator delete[]( void * ptr, const std::nothrow_t & )
noexcept ;
```

効果

nothrow版の配列型のnew式によって呼び出されたコンストラクターが例外を投げた時に、ストレージを解放するために呼ばれる。配列型のdelete式では呼ばれない。

デフォルトの挙動

operator delete[](ptr)を呼び出す。

```
void operator delete[]( void * ptr, const std::nothrow_t
& ) noexcept
{
    operator delete[]( ptr ) ;
}
```

13.7.9 ユーザー定義リテラル

以下の形のオーバーロード演算子は、ユーザー定義リテラル演算子のオーバーロードである。

operator "" 識別子

""と識別子の間には、必ずひとつ以上の空白文字を入れなければならない。また、識別子の先頭文字は、必ずアンダースコアひとつから始まらなければならない。ただし、通常の識別子では、アンダースコアから始まる名前は予約されているので注意すること。これは、ユーザー定義リテラル演算子のみの特別な条件である。

```
// OK
void operator "" /* 空白文字が必要 */ _x( unsigned long long
int ) ;

// エラー、""と_yの間に空白文字がない
void operator ""_y( unsigned long long int ) ;

// エラー、識別子がアンダースコアから始まっていない
void operator "" z( unsigned long long int ) ;

// エラー、""の間に空白文字がある
void operator " " _z( unsigned long long int ) ;
```

リテラル演算子の仮引数リストは、以下のいずれかでなければならない。

```
const char*
unsigned long long int
long double
```

```
char
wchar_t
char16_t
char32_t
const char*, std::size_t
const wchar_t*, std::size_t
const char16_t*, std::size_t
const char32_t*, std::size_t
```

上記以外の仮引数リストを指定すると、エラーとなる。

リテラル演算子テンプレートは、仮引数リストが空で、テンプレート仮引数は、char型の非型テンプレート仮引数の仮引数パックでなければならない。

```
template < char ... Chars >
void operator "" _x () { }
```

これ以外のテンプレート仮引数を取るリテラル演算子テンプレートはエラーとなる。

リテラル演算子は、Cリンクエージを持つことができない。

```
// エラー
extern "C" void operator "" _x( unsigned long long int ) { }

// OK
extern "C++" void operator "" _x( unsigned long long int ) { }
```

リテラル演算子は、名前空間スコープで宣言しなければならない。つまり、クラススコープで宣言することはできない。ただし、friend関数になることはできる。

```
// グローバル名前空間スコープ
void operator "" _x( unsigned long long int ) { }

namespace ns {
// ns名前空間スコープ
void operator "" _x( unsigned long long int ) { }
}

class X
{
```

```
// OK、friend宣言できる
friend void operator "" _x( unsigned long long int ) ;

// エラー、クラススコープでは宣言できない
static void operator "" _y( unsigned long long int ) ;
}
```

ただし、名前空間スコープで宣言したリテラル演算子を、ユーザー定義リテラルとして使うには、using宣言かusingディレクティブが必要となる。

```
namespace ns {
void operator "" _x( unsigned long long int ) { }

int main( )
{
    1_x ; // エラー、operator "" _xは見つからない

    {
        using namespace ns ;
        1_x ; // OK
    }

    {
        using ns::operator "" _x ;
        1_x ; // OK
    }
}
```

これ以外は、通常の関数と何ら変りない。例えば、明示的に呼び出すこともできるし、その際には通常のオーバーロード解決に従う。inlineやconstexpr関数として宣言することもできる。内部リンクエージでも外部リンクエージのどちらでも持てる。アドレスも取得できる。等々。

14 テンプレート(Templates)

テンプレートとは、コンパイル時に型や値を引数として渡す機能のことである。

template < テンプレート仮引数リスト > 宣言

テンプレートを指定できる宣言は、クラス、関数、エイリアス宣言である。それぞれ、クラステンプレート、関数テンプレート、エイリアステンプレートと呼ばれる。

```
// 関数テンプレート
template < typename T >
void f( ) ;

// クラステンプレート
template < typename T >
struct X ;

// エイリアステンプレート
template < typename T >
using type = T ;
```

テンプレート仮引数リストのテンプレート仮引数名は省略できる。

```
template < typename >
struct X { } ;
```

俗に、クラス/関数のテンプレートの代わりに、テンプレートのクラス/関数などと呼ばれることがある。ただし、テンプレートコードには実体がないので、テンプレートのクラスというよりは、クラスのテンプレートと言ったほうが正確である。

テンプレートには実体がない。テンプレートは具体的な型実引数を与えられて、実体化(インスタンス化、Instantiation)する。テンプレートの異なる実体は、それぞれ別の型を持つ。

```
// クラステンプレートの宣言
template < typename T >
struct X ;

// クラステンプレートの定義
template < typename T >
struct X { } ;

// クラステンプレートの実体化
X<int> x ;
```

テンプレートを宣言できるのは、名前空間スコープかクラススコープの中だけである。例えば、関数のブロックスコープでは宣言できない。

```
void f()
{
    // エラー
    template < typename T >
    struct X { } ;
}
```

14.1 テンプレート仮引数/実引数(Template parameters/arguments)

テンプレート仮引数(template parameters)は、テンプレート側で記述する引数である。テンプレート実引数は、テンプレートに与える引数である。本来、テンプレート仮引数とテンプレート実引数は明確に別の機能であるが、本書では分かりやすさを重視して、同時に説明する。

テンプレート仮引数と実引数には、型、非型、テンプレートがある。ここでは、テンプレート仮引数の宣言方法とテンプレート実引数の渡し方を説明する。

14.1.1 型テンプレート仮引数/実引数

型を引数に取るテンプレート仮引数は、classまたはtypenameというキーワードに続いて、テンプレート仮引数名を記述する。classとtypenameには、意味上の違いはない。

```
template < typename T >
class X ;

template < class T >
class Y ;

// 複数の引数を取る場合は、,で区切る
template < typename T, typename U >
class Z ;
```

テンプレート実引数は、テンプレート名に続いて、<>で実引数を囲んで渡す。

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

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

```
int main( )
{
    // テンプレート実引数 int
    f<int>() ;
    // テンプレート実引数 int
    X<int> x ;
}
```

テンプレート仮引数は、テンプレートコードの中で、あたかも型や値であるかのように使うことができる。

```
template < typename T >
struct X
{
    // T型のデータメンバーmemberの宣言
    T member ;
} ;
```

Tの具体的な型は、テンプレート実引数が与えられたときに、すなわち、テンプレートが実体化したときに決定される。

14.1.2 非型テンプレート仮引数/実引数

非型テンプレート仮引数(non-type template parameters)は、型以外、つまり値を引数に取る。非型テンプレート仮引数は、class/typenameと記述する代わりに、型を記述する。非型テンプレート仮引数に使える型は以下の通り。

- 整数型とenum型

```
// 整数型の一例
template < int I, unsigned int UI, unsigned long long int
ULLI >
struct A { } ;

// enum型の例
enum struct E : int { value = 0 } ;

template < E value >
```

```

struct B { } ;

int main( )
{
    // 非型テンプレート実引数
    A< 0, 0u, 0ull > a ;
    B< E::value > b ;
}

```

- オブジェクトへのポインターと、関数へのポインター

```

// オブジェクトへのポインター
template < int * P >
struct A { } ;

// 関数へのポインター
using func_ptr_type = void (*)();

template < func_ptr_type FUN >
struct B { } ;

void f() { }
static int global = 0 ;

int main( )
{
    // 非型テンプレート実引数
    A< &global > a ;
    B< &f > b ;
}

```

- オブジェクトへのlvalueリファレンスと関数へのlvalueリファレンス

```

// オブジェクトへのlvalueリファレンス
template < int & P >
struct A { } ;

// 関数型のtypedef名
using func_type = void () ;
// 関数へのlvalueリファレンス

```

```

template < func_type & FUN >
struct B { } ;

void f() { }
static int global = 0 ;

int main( )
{
    // 非型テンプレート実引数
    A< global > a ;
    B< f > b ;
}

```

- メンバーへのポインター

```

struct X
{
    int member ;
} ;

// メンバーへのポインター
template < int X::* P >
struct Y { } ;

int main( )
{
    Y< &X::member > y ;
}

```

- std::nullptr_t

```

template < std::nullptr_t N >
struct X { } ;

X< nullptr > x ;

```

非型かつ非リファレンスのテンプレート仮引数はprvalueであり、いかなる方法をもっても代入などの値の変更をすることはできない。アドレスを取得することはできない。リファレンスに束縛される場合には、一時オブジェクトが使われる。

```

template < int I >
void f()
{
// 値の変更はできない
    I = 0 ; // エラー
    ++I ; // エラー

// アドレスの取得はできない
    int * p = &I ; // エラー

// リファレンスの束縛には一時オブジェクトがつかわれる
    int const & ref = I ;
}

```

非型テンプレート仮引数は、浮動小数点数型、クラス型、void型として宣言することはできない。

```

// エラー
template < double d >
struct S1 ;

struct X { } ;

// エラー
template < X x >
struct S2 ;

// エラー
template < void v >
struct S3 ;

```

非型テンプレート仮引数の型が、「T型への配列」や、「T型を返す関数」である場合、それぞれ、「T型へのポインター」、「T型を返す関数へのポインター」と、型が変換される。

```

// int * a
template < int a[5] >
struct S1 ;

// int (*func)()
template < int func() >
struct S2 ;

```

非型テンプレート実引数は、厳しい制約を受ける。

整数型とenum型の非型テンプレート仮引数に対するテンプレート実引数は、以下のとおりである。

- テンプレート仮引数の型に変換できる定数式

```
template < int I >
struct S { } ;

int main()
{
    S< 0 > s1 ;
    S< 1 + 1 > s2 ;

    constexpr int x = 0 ;
    S< x > s3 ;
}
```

- 非型テンプレート仮引数の名前

```
template < int I >
struct A { } ;

template < int I >
struct B
{
    A<I> a ;
} ;
```

- 定数式の、静的ストレージ上のオブジェクトへのアドレスと関数で、リンクエージを持つもの。

```
int x = 0 ;

template < int * ptr >
struct A { } ;
```

```

int f() { return 0 ; }

template < int (*func)() >
struct B { } ;

int main()
{
    A< &x > a ; // OK
    B< &f > b ; // OK

    static int no_linkage = 0 ; // リンケージを持たない
    A< no_linkage > a2 ; // エラー
}

```

これは、実際にはもっと複雑な条件だが、本書では省略する。

- nullポインター、nullメンバーの値であると評価される定数式
- メンバーへのポインター

文字列リテラルをテンプレート実引数として渡すことはできない。配列の要素へのアドレスを渡すこともできない。

14.1.3 テンプレートテンプレート仮引数/実引数

テンプレート仮引数は、テンプレートを実引数に取ることができる。これをテンプレートテンプレート仮引数、テンプレートテンプレート実引数と呼ぶ。

```

template < typename T >
struct A { } ;

template <
    template < typename T >
    // ここはclassキーワードを使わなければならない
    class U
>
struct B
{
    // Uはテンプレートとして使える
    U<int> u ;
} ;

```

```

int main()
{
    B< A > b ; // エラー

    B< int > e1 ; // エラー
    B< 0 > e2 ; // エラー
}

```

テンプレートテンプレート仮引数は、テンプレートを受け取る。テンプレートを受け取るテンプレートなので、「テンプレートテンプレート仮引数」となる。この宣言は、テンプレート仮引数自体が、さらにtemplateキーワードを使う文法になる。テンプレート仮引数の名前には、文法上の制約により、classキーワードを使わなければならない。

```

// OK
template < template < typename > class T >
struct X { } ;

// エラー
template < template < typename > typename T >
struct Y { } ;

```

テンプレートテンプレート仮引数に対するテンプレートテンプレート実引数は、クラステンプレートか、エイリアステンプレートでなければならない。

```

// クラステンプレート
template < typename T >
struct A { } ;

// エイリアステンプレート
template < typename T >
using B = T ;

template < template < typename T > class U >
struct C
{
    U<int> u ;
} ;

int main()
{
    C< A > a ; // クラステンプレート
    C< B > b ; // エイリアステンプレート
}

```

}

14.1.4 デフォルトテンプレート実引数

テンプレート仮引数には、デフォルトテンプレート実引数を指定することができる。デフォルトテンプレート実引数は、=に続けて実引数を記述する。

```
template < typename T = int, int I = 0 >
struct X { } ;

int main()
{
    X<> a ; // X< int, 0 >
    X< double > b ; // X< double, 0 >
    X< short, 1 > c ; // X< short, 1 >
}
```

デフォルトテンプレート実引数は、可変引数テンプレートを除く、すべての種類のテンプレート仮引数(型、非型、テンプレート)に指定できる。

```
// 型テンプレート
template < typename T = int >
struct X { } ;

// 非型テンプレート
template < int I = 123 >
struct Y { } ;

// テンプレートテンプレート
template < template < typename > class TEMP = X >
struct Z { } ;

// エラー、可変引数テンプレートには指定できない
template < typename ... T >
struct Error { } ;
```

デフォルトテンプレート実引数は、クラステンプレートのメンバーのクラス外部での定義

に指定することはできない。これは説明が難しい。この場合のテンプレート仮引数には、クラスのテンプレートと、クラスのメンバーのテンプレートがあるが、このどちらにも、デフォルトテンプレート実引数を指定することはできない。

```
// クラステンプレートの定義
template < typename T >
struct X
{
    // メンバー関数テンプレートの宣言
    template < typename U >
    void f() ;
};

// クラス外部での定義
template < typename T = int > // エラー（クラスXのテンプレート）
template < typename U = int > // エラー（クラスXのメンバーfのテンプレート）
void X<T>::f()
{ }
```

この場合、デフォルトテンプレート実引数を指定したい場合は、それぞれ、クラステンプレートの定義や、メンバーの宣言に指定しなければならない。

デフォルトテンプレート実引数は、`friend`クラステンプレートのメンバー宣言に指定することはできない。

```
template < typename T >
struct X
{
    // エラー
    template < typename U = int >
    friend class X ;
};
```

デフォルトテンプレート実引数が、`friend`関数テンプレート宣言に指定された場合、その`friend`関数テンプレート宣言は、定義でなければならない。

```
template < typename T >
struct X
{
```

```
// エラー、宣言
template < typename U = int >
friend void f() ;

// OK、定義
template < typename U = int >
friend void g() { }
} ;
```

これは、`friend`関数の宣言は、定義となることができるためである。詳しくは、11.3 [friend](#)を参照。

デフォルトテンプレート実引数が指定されているテンプレート仮引数に続くテンプレート仮引数には、デフォルトテンプレート実引数が指定されていなければならない。

```
// エラー、後続のテンプレート仮引数にテンプレート実引数が指定され
てない
template < typename T = int, typename U >
struct X ;
```

あるいは、後続のテンプレート仮引数は、可変引数テンプレート仮引数でなければならない。

```
// OK
template < typename T = int , typename ... Types >
struct X { } ;
```

同じ宣言のテンプレート仮引数には、二度以上デフォルトテンプレート実引数を指定してはならない。

```
// OK
template < typename T = int> struct A ;
template < typename T > struct A { } ;

// OK
template < typename T > struct B ;
template < typename T = int > struct B { } ;

// エラー
template < typename T = int > struct C ;
template < tvbename T = int > struct C { } ;
```

デフォルトテンプレート実引数に与える式に含まれる>には注意が必要である。ネストされていない>は、テンプレート宣言の終了とみなされる。

```
// 文法エラー
template < int i = 1 > 2 >
struct X { } ;

// OK
template < int i = ( 1 > 2 ) >
struct Y { } ;
```

テンプレートテンプレート仮引数内のテンプレート仮引数にも、デフォルトテンプレート実引数を指定することができる。

```
template < template < typename TT = int > class T >
struct X
{
    T<> a ; // T< int >
} ;
```

14.1.5 可変テンプレート仮引数

識別子の前に...を記述したテンプレート仮引数は、仮引数パックの宣言となる。これは、可変引数テンプレート(Variadic Templates)のための仮引数の宣言である。詳しくは、14.4.3 [可変引数テンプレート](#)を参照。

```
template < typename ... Types >
struct X { } ;
```

14.2 テンプレート特殊化の名前(Names of template specializations)

特殊化されたテンプレートは、テンプレートID(template-id)によって参照できる。これは

普通の名前とは違い、特殊化したテンプレート実引数を指定する。テンプレートidとは、テンプレート名に続けて、<を記述してテンプレート実引数を記述し、>で閉じることによって記述できる。

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

X テンプレート名

X<int> テンプレートXをintに特殊化したテンプレートID

14.3 型の同一性(Type equivalence)

二つのテンプレートidが同じクラスや関数であるためには、以下の条件を満たさなければならない。

- テンプレート名、演算子関数ID(オーバーロード演算子のテンプレートの場合)、リテラル演算子ID(オーバーロードリテラル演算子のテンプレートの場合)が同じ。
- 対応するテンプレート実引数の型が等しい。
- 対応する非型テンプレート実引数の値が同じ。

```
#include <type_traits>

template < int I >
class X { } ;

int main()
{
    std::cout << std::is_same< X<0>, X<0> >::value ; // true
    std::cout << std::is_same< X<0>, X<1> >::value ; // false
    std::cout << std::is_same< X<2>, X< 1 + 1 > >::value
; // true
}
```

- 対応する非型テンプレート実引数が、ポインター、メンバーへのポインター、リファレンスの場合、同じ外部オブジェクトを指し示していかなければならない。ポインターとメンバーへのポインターの場合、nullポインターでもよい。

```
#include <type_traits>

template < int * P >
class X { } ;

int a ;
int b ;

int main()
{
    std::cout << std::is_same< X<&a>, X<&a> >::value ; // true
    std::cout << std::is_same< X<&a>, X<&b> >::value ; // false
    std::cout << std::is_same< X<nullptr>, X<nullptr> >::value ; // true
}
```

- 対応するテンプレートテンプレート実引数が同じ。

```
#include <type_traits>

template < template < typename > class T >
class X { } ;

template < typename T >
class Y { } ;
template < typename T >
class Z { } ;

int main()
{
    std::cout << std::is_same< X<Y>, X<Y> >::value ; // true
    std::cout << std::is_same< X<Y>, X<Z> >::value ; // false
}
```

14.4 テンプレート宣言(Template declarations)

14.4.1 クラステンプレート(Class templates)

クラステンプレートはテンプレート仮引数に続けてクラス宣言を書くことで宣言できる。

```
template < typename T, std::size_t SIZE >
class Array
{
private :
    T buf[SIZE] ;
public :

    Array() : buf{}
    { }

    T & operator []( std::size_t i )
    { return buf[i] ; }

    T const & operator []( std::size_t i ) const
    { return buf[i] ; }

} ;

int main( )
{
    Array< int, 10 > a ;
    a[3] = 100 ;
}
```

テンプレート仮引数は、型や値やテンプレートとして使うことができ、テンプレートが実体化されたときにテンプレート実引数によって置き換えられる。

クラステンプレートのメンバー関数、メンバークラス、メンバーenum、staticデータメンバー、メンバーテンプレートを、メンバーが属するクラステンプレート定義の外部で定義する場合は、メンバー定義には、メンバーが属するひとつ外側のクラステンプレートのテンプレート仮引数を記述し、さらにメンバーが属するひとつ外側のクラステンプレートのテンプレート名に続けてテンプレート実引数リストを、同じ順番で記述しなければならない。

```
template < typename T >
class Outer
```

```
{  
    // クラス定義内部でのメンバーの宣言  
    void member_function() ;  
    class member_class ;  
    enum struct member_enum : int ;  
    static int static_data_member ;  
  
    template < typename U >  
    class member_template ;  
};  
  
// クラス定義外部でのメンバーの定義  
  
// メンバー関数  
template < typename T >  
void Outer<T>::member_function() { }  
  
// メンバークラス  
template < typename T >  
class Outer<T>::member_class { } ;  
  
// メンバーenum  
template < typename T >  
enum struct Outer<T>::member_enum : int  
{ value = 1 } ;  
  
// staticデータメンバー  
template < typename T >  
int Outer<T>::static_data_member ;  
  
// メンバーテンプレート  
// クラス以外のテンプレートも同じ方法で記述する  
template < typename T > // Outerのテンプレート仮引数  
template < typename U > // member_templateのテンプレート仮引数  
class Outer<T>::member_template { } ;
```

これは、一見すると、恐ろしいほど難しそうに見えるが、クラスのメンバーをクラス定義の外部で宣言する文法に、テンプレート仮引数が加わっただけだ。ただし、メンバーテンプレートの定義方法だけは、少し分かりにくい。クラステンプレートのメンバーテンプレートには、メンバーテンプレートのテンプレート仮引数と、メンバーテンプレートが属するクラステンプレートのテンプレート仮引数の、両方が必要だからだ。

もちろん、メンバーはいくらでもネストできるので、宣言はもっと複雑になることもある。

```

template < typename T >
class Outer
{
    template < typename U >
    class Inner
    {
        template < typename V >
        class Deep ;
    } ;
} ;

template < typename T > // Outerのテンプレート仮引数
template < typename U > // Innerのテンプレート仮引数
template < typename V > // Deepのテンプレート仮引数
class Outer<T>::Inner<U>::Deep { } ;

```

14.4.2 メンバーテンプレート(Member Templates)

テンプレートは、クラス定義の内部で宣言することができる。そのようなテンプレートを、メンバーテンプレート(Member Templates)と呼ぶ。

```

class Outer
{
    // メンバーテンプレート
    template < typename T >
    class Inner { } ;
} ;

```

メンバーテンプレートは、クラス定義の内部でも外部でも定義できる。メンバーをクラス定義の外で定義する方法については、14.4.1 クラステンプレート(Class templates)を参照。

メンバーテンプレートには、いくつかの制限や例外的なルールが存在する。

ローカルクラスはメンバーテンプレートを持つことができない。

```

void f()
{
    // エラー、

```

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

デストラクターはメンバーテンプレートとして宣言できない。

```
class X
{
    // エラー
    template < typename T >
    ~X() { }
};
```

メンバー関数テンプレートはvirtual関数にはできない。

```
class X
{
    // エラー
    template < typename T >
    virtual void f() ;
};
```

また、メンバー関数テンプレートの特殊化は基本クラスのvirtual関数をオーバーライドすることはない。

```
struct Base
{
    virtual void f( int ) ;
};

struct Derived : Base
{
    template < typename T >
    void f( T ) ; // Base::fをオーバーライドしない
};
```

14.4.3 可変引数テンプレート(Variadic Templates)

可変引数テンプレート(Variadic templates)は、0個以上のテンプレート実引数や関数実引数を取るテンプレートのことである。

テンプレート仮引数の宣言で、識別子の前に...が記述されているとき、これをテンプレート仮引数パック(Template parameter pack)と呼ぶ。

```
// 型テンプレート仮引数パック
template < typename ... Type_pack >
class X { } ;

// 非型テンプレート仮引数パック
template < int ... Int_pack >
class Y { } ;

// テンプレートテンプレート仮引数パック
template < template < typename > class ... Template_pack >
class Z { } ;

// テンプレート実引数として与えるためのテンプレート
template < typename T >
class Arg { } ;

int main( )
{
    X< > x1 ;
    X< int > x2 ;
    X< int, int, double, float > x3 ;

    Y< > y1 ;
    Y< 0 > y2 ;
    Y< 1, 2, 3, 4, 5 > y3 ;

    Z< > z1 ;
    Z< Arg > z2 ;
    Z< Arg, Arg, Arg > z3 ;
}
```

上記の例のように、テンプレート仮引数パックは、0個以上の任意の数のテンプレート実引数を取る。

非型テンプレート仮引数として、オブジェクトや関数へのポインターやリファレンスなども、可変引数テンプレートにできる。ただし、記述方法が少し分かりにくい。そのため、直

接書くよりもtypedef名を使う方が読みやすくなる。

```
// やや読みにくい宣言
template < void ( * ... func_pack )() >
class X { } ;

using type = void (*)() ;
// 読みやすい宣言
template < type ... func_pack >
class Y { } ;
```

可変引数テンプレートをテンプレート実引数に取るテンプレートテンプレート仮引数は、以下のように書く。

```
// 可変引数テンプレートを取るテンプレート
template <
    template < typename ... Type > class Template
>
class X { } ;

// テンプレート実引数に渡すテンプレート
template < typename ... Type_pack >
class Arg { } ;

int main( )
{
    X< Arg > x ;
}
```

もちろん、このクラステンプレートXを、さらに可変引数テンプレートにすることもできる。

```
// 可変引数テンプレートを取る可変引数テンプレート
template <
    template < typename ... Type_pack > class ...
Template_pack
>
class X { } ;

// テンプレート実引数に渡すテンプレート
template < typename ... Type_pack >
```

```
class Arg { } ;

int main( )
{
    X< Arg, Arg, Arg > x ;
}
```

関数仮引数パック(function parameter pack)は、テンプレート仮引数パックを使って、0個以上の関数の実引数を得る関数仮引数である。宣言方法は、関数の仮引数の識別子の前に...を記述する。

```
// 任意の型の0個以上の実引数をとる関数
template < typename ... template_parameter_pack >
void f( template_parameter_pack ... function_parameter_pack )
{ }

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

テンプレート仮引数パックと関数仮引数パックをあわせて、仮引数パック(Parameter pack)という。

仮引数パックは、そのままでは使えない。仮引数パックを使うには、展開しなければならない。これをパック展開(Pack expansion)という。パック展開は、パターンと...を組み合わせて記述する。テンプレートの実体化の際に、0個以上の実引数が、パターンに合わせて展開される。パターンの範囲は文脈により異なるが、見た目上は、仮引数パックを含む文字列を繰り返しているように見えるよう設計されている。

```
template < typename ... pack >
struct type_list_impl { } ;

template < typename ... pack >
struct type_list
{
    using type = type_list_impl<
        pack // パターン
        ...
    > ;
} ;
```

```

int main()
{
    // type_list_impl<>
    type_list<>::type t1 ;
    // type_list_impl<int, int>
    type_list<int, int>::type t2 ;
}

```

この例では、`type_list`の仮引数パックである`pack`をパック展開して、`type_list_impl`のテンプレート実引数に渡している。“`pack...`”というのが、パック展開である。`pack`がパターンだ。この場合は、そのまま展開している。

関数仮引数パックの場合も同様である。

```

template < typename ... Types >
void f_impl( Types ... pack ) { }

template < typename ... Types >
void f( Types ... pack )
{
    f_impl( pack... ) ;
}

int main()
{
    // f_impl( )を呼ぶ
    f( ) ;
    // f_impl( 1, 2, 3 )を呼ぶ
    f( 1, 2, 3 ) ;
}

```

ここでも、“`pack...`”がパック展開で、`pack`がパターンになっている。

パターンの記述は、パック展開の文脈に依存する。パック展開中の仮引数パックに対するパターンには、その文脈で許される記述をすることができ、そのパターンによって展開される。

```

template < typename T >
struct wrap { } ;

```

```

template < typename ... pack >
struct type_list_impl { } ;

template < typename ... pack >
struct type_list
{
    using pointers = type_list_impl<
        pack *...
    > ;

    using references = type_list_impl<
        pack &...
    > ;

    using wraps = type_list_impl<
        wrap<pack>...
    > ;
} ;

int main()
{
    // type_list_impl< char *, short *, int * >
    type_list< char, short, int >::pointers t1 ;

    // type_list_impl< char &, short &, int & >
    type_list< char, short, int >::references t2 ;

    // type_list_impl< wrap<char>, wrap<short>, wrap<int> >
    type_list< char, short, int>::wraps t3 ;
}

```

packがテンプレート仮引数パックであるならば、pack * ...はそれぞれの実引数に*を加えたパターンとして展開される。wrap<pack>は、それぞれの実引数をクラステンプレートwrapの実引数に渡すパターンとして展開される。

関数仮引数パックの場合も同様。

```

template < typename ... param_pack >
void f( param_pack ... ) { } // 仮引数名の省略

template < typename T >
T identity( T value )
{

```

```

        return value ;
}

template < typename ... param_pack >
void g( param_pack ... pack )
{
    f( pack ... ) ; // そのまま関数fに渡す
    f( (pack + 1)... ) ; // +1して関数fに渡す
    f( identity(pack)... ) ; // identity()の評価の結果を関数f
に渡す

}

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

```

パック展開のパターンの中には、未展開のパラメータパック名がひとつは存在しなければならない。

```

template < typename ... Types >
struct seq { } ;

template < typename ... Types >
struct S
{
    using t1 = seq< Types ... > ; // OK
    using t2 = seq< Types ... ... > ; // エラー、未展開のパラメータパックがない
} ;

```

未展開のパラメータパックをそのまま使うことはできない。

```

template < typename ... Types >
struct seq { } ;

template < typename ... Types >

```

```
struct S
{
    using t1 = seq< Types > ; // エラー、未展開のパラメーターパック
};
```

もし、パック展開のパターンに、複数のパラメーターパックがある場合は、それぞれのパラメーターパックの数が等しくなければならない。

```
template < int ... >
struct int_seq { } ;

template < typename ... Types >
void f( Types ... ) { }

template <
    int ... Is,
    typename ... Types
>
void g( int_seq< Is... >, Types ... args )
{
    // パラメーターパックargsとIsがひとつのパターンに存在
    f( args(Is)... );
}

int h( int x ) { return x ; }

int main()
{
    // OK、パラメーターパックargsとIsの個数が一致
    g( int_seq< 1, 2, 3 >(), &h, &h, &h ) ;
    // f( h(1), h(2), h(3) ) と展開される

    // エラー、パラメーターパックの個数が不一致
    g( int_seq< 1, 2 >(), &h ) ;
}
```

パック展開できる文脈は限られているが、一般に、型や式をコンマで区切って記述する文脈に書くことができる。

まず、特別なパック展開が三種類ある。ただし、規格上は区別されてない。この特別なパック展開は、パック展開でありながら、仮引数パックでもあるという特徴を持つ。

- 関数仮引数パックの宣言

関数仮引数パックの宣言は、...以外がそのままパック展開のパターンにもなる。そのため、これは仮引数パックかつパック展開となる。

```
template < typename ... param_pack >
void f( param_pack const & ... pack ) ;
```

この例では、"param_pack const &"がパターンとなる。

関数仮引数パックは、宣言 자체がパターンとなっているので、実引数の型がパターンに一致しない場合、実引数推定が失敗する。これを利用して、型を部分的に限定することができる。

```
// ポインター型しか実引数に指定できない関数
template < typename ... Types >
void f( Types * ... pack )
{ }

int main( )
{
    int * p = nullptr ;
    f( p ) ; // OK
    f( 123 ) ; // エラー
}
```

同様に、メンバーへのポインターや、関数へのポインターに限定することもできる。

```
// 実引数を取らず、任意の型を戻り値に返す関数へのポインターを
// 実引数に取る関数
template < typename ... ReturnType >
void f( ReturnType (* ... pack)( ) )
{ }

void g( ) { }
int h( ) { }
double i( ) { }

int main( )
{
    f( &g, &h, &i ) ;
```

}

パターンは...を除いた部分なので、この場合のパターンは、“`ReturnType (*)()`”となる。つまり関数へのポインター型になっている。

もっと複雑な条件で型を限定したい場合は、テンプレートメタプログラミングの技法を使うことができる。その詳細は本書の範疇を超えるので、ここには書かない。

- テンプレート仮引数パックがパック展開となる場合

これには二種類ある。ひとつは、非型テンプレート仮引数パックの仮引数宣言に、先に宣言されたテンプレート仮引数が使われる場合。パターンは、...を除いた部分となる。

```
template < typename T, /* パターンここから */ T * /* パター
ンここまで */... Types >
struct X { } ;

int main( )
{
    X< int, nullptr > x ;
}
```

この例では、“`T *`”がパターンとなっている。関数仮引数パックと同じように、実引数の型を制限できる。

もうひとつは、テンプレートテンプレート仮引数パックの中で、先に宣言されたテンプレート仮引数パックが使われる場合。パターンは...を除いた部分となる。

```
template< typename ... Types >
struct Outer
{
    template< template< Types > class ... pack >
    struct Inner ;
};
```

これは少し分かりにくい。仮引数パック`Types`のパック展開が、クラステンプレート`Inner`のテンプレート仮引数で行われている。パターンは、“`template< Types >`
`class`”だ。

今、`Outer< char, short, int >`のようにテンプレート実引数が渡されたとすると、`Inner`のパック展開の結果を擬似的に記述すると、以下のようになる。

```

template < char, short, int >
struct Outer
{
    template <
        template < char > class pack_1,
        template < short > class pack_2,
        template < int > class pack_3
    >
    struct Inner ;
} ;

```

つまり、テンプレートInnerのテンプレート仮引数は、テンプレートテンプレート仮引数がパターンで、仮引数パックであるTypesに対してパターンを適用してパック展開されることになる。

これはあくまで解説のための擬似的なコードである。可変引数テンプレートは、パックとパック展開という形で使うので、手で書いたようなテンプレートに展開されるわけではない。

その他のパック展開だけの文脈は以下の通り。

- 初期化リストの中

関数呼び出し式の中の式リストも、初期化リストである。パターンは初期化子の式となる。

```

struct X
{
    template < typename ... pack >
    X( pack ... ) { }
} ;

template < typename ... Types >
void f( Types ... pack ) { }

template < typename ... Types >
void g( Types ... pack )
{
    X x( pack ... );
    f( pack... );

    f( (pack + 1)... ); // パターンはpack+1
}

```

- 基本クラス指定リストの中

これにより、仮引数パックに対するすべての実引数の型を基本クラスに指定することができる。パターンは基本クラス指定子。

```
template < typename ... pack >
struct X : pack ...
{ } ;

struct A { } ;
struct B { } ;

int main( )
{
    X< > x1 ; // 基本クラスなし
    X< A > x2 ; // 基本クラスA
    X< B > x3 ; // 基本クラスB
    X< A, B > x4 ; // 基本クラスAとB

    X< int > x5 ; // エラー、intは基本クラスにできない
}
```

複雑なパターンの例。

```
template < typename T >
struct wrap { } ;

template < typename ... pack >
struct X : wrap< pack > ...
{ } ;

int main( )
{
    // Xの基本クラスはwrap<int>とwrap<double>
    X< int, double > x ;
}
```

この例では、テンプレート実引数はwrap<T>に包まれて基本クラスに指定される。そのため、直接intから派生するのではなく、テンプレートwrapの特殊化から派生することになる。

- メンバー初期化リスト

これは、初期化リストとほぼおなじだ。パターンはメンバー初期化子の式になる。

```
struct X
{
    template < typename ... pack >
    X( pack ... ) { }
} ;

struct Y
{
    X x ;
    template < typename ... pack >
    Y( pack ... args )
        : x( args... )
    { }
} ;

int main( )
{
    Y y( 1, 2, 3 ) ;
}
```

- テンプレート実引数リスト

パターンはテンプレート実引数。

```
template < typename ... pack >
struct X { } ;

template < typename T >
struct wrap { } ;

template < typename ... pack >
struct Y
{
    // パターンはpack
    X< pack ... > x1 ;
    // パターンはwrap<pack>
    X< wrap<pack> ... > x2 ;
} ;
```

- アトリビュートリスト

パターンはアトリビュート

例えば、以下のような型を引数に取る実装依存のアトリビュートトークンがあったとする。

```
[[ token( 型 ) ]]
```

この場合、可変長テンプレートで、パラメーターパックとして受け取った型をすべて渡したい場合、以下のように書ける。

```
template < typename ... pack >
struct X
{
    [[ token( pack )... ]] ;
} ;

void f()
{
// [[ token( int ) ]]
    X< int > a ;
// [[ token( int ), token( int ) ]]
    X< int, int > b ;
// [[ token( char ), token( short ), token( int ), token(
long ) ]]
    X< char, short, int, long > c ;
}
```

- アライメント指定子

パターンは...を除いたアライメント指定子

```
template < typename ... pack >
struct X
{
    alignas( pack ... ) int data_member ;
} ;
```

たとえば、`X<int, short, double>`と実引数を与えると、この特殊化されたクラスXの

データメンバ—`data_member`のアライメント指定は、`alignas(int, short, double)`と記述したものと等しくなる。

- キャプチャーリスト
パターンはキャプチャー

```
template < typename ... Types >
void f_impl( Types ... ) { }

template < typename ... Types >
void f( Types ... pack )
{
    // キャプチャーのパターンは&pack
    [ &pack ... ]{ f_impl( pack... ) ; }() ;
}
```

この例では、関数仮引数パックをパック展開して、ラムダ式でリファレンスキャプチャーしている。

- `sizeof...`式

`sizeof...`式も、パック展開の一種である。ただし、パターンは識別子なので、特に特別なことはできない。ただ、仮引数パックの数を返すだけだ。

```
template < typename ... Types >
int f( Types ... args )
{
    sizeof...(Types) ; // 評価はテンプレート仮引数パックの数
    sizeof...(args) ; // 評価は関数仮引数パックの数

    return sizeof...(Types) ;
}

int main( )
{
    f() ; // 評価は0
    f( 1, 2, 3 ) ; // 評価は3
}
```

14.4.3.1 可変引数テンプレートの使い方

本書はコア言語の文法と機能を解説するものであって、使い方を紹介するものではない。ただし、可変引数テンプレートでは、最低限の使い方を解説する。

仮引数テンプレートは、仮引数パックによって、0個以上の実引数を取ることができる。ただし、仮引数パックはそのままでは使えない。仮引数パックは、仮引数ではないからだ。仮引数パックを使うには、パック展開しなければならない。では、一体どうやって、任意個の実引数に対応した汎用的なコードを書くのか。それには、二つ方法がある。固定長の仮引数を取るものに渡すか再帰だ。

固定長の仮引数を取るものに渡すというのは、単にパック展開して渡せばよい。

```
#include <iostream>

// 実引数ゼロ個の時は何もしない
void print_impl( ) { }

template < typename T1 >
void print_impl( T1 a1 )
{
    std::cout << a1 << std::endl ;
}

template < typename T1, typename T2 >
void print_impl( T1 a1, T2 a2 )
{
    std::cout << a1 << std::endl ;
    std::cout << a2 << std::endl ;
}

template < typename ... Types >
void print( Types ... pack )
{
    print_impl( pack ... ) ;
}

int main( )
{
    print( ) ; // なにもしない
    print( 1 ) ; // OK
    print( 1, 2 ) ; // OK
    print( 1, 2, 3 ) ; // エラー
}
```

問題は、コード例をみても分かるように、このコードは汎用的ではないという事だ。固定長の仮引数を持つ型数に渡していくために「固定長」が扱えない。これではせっかく

の可変引数の意味がない。

固定長の仮引数を取るものに渡すというのは、特定の条件で、特別な処理をしたい場合に使えるが、汎用的に使うことはできない。

任意の個数の実引数に対応した汎用的なコードを書くためには、再帰を使う。

```
// #1 再帰の終了条件
void print( ) { }

// #2 引数を一つづつ処理
template < typename T, typename ... Types >
void print( T head, Types ... tail )
{
    std::cout << head << std::endl ;
    print( tail ... ) ; // 再帰的実体化
}

int main( )
{
    print( ) ; // #1
    print( 1 ) ; // #2
    print( 1, 2 ) ; // #2
    print( 1, 2, 3 ) ; // #2
}
```

再帰と言っても、同じ関数を再帰的に呼び出すわけではない。仮引数パックで受ける実引数の数をひとつづつ減らしていき、新たなテンプレートを実体化させて呼び出している。たとえば、`print(1, 2, 3, 4, 5)`を呼び出した場合、以下のように実体化されて呼び出される。

```
print<int, int, int, int, int>( 1, 2, 3, 4, 5 )
print<int, int, int, int>( 2, 3, 4, 5 )
print<int, int, int>( 3, 4, 5 )
print<int, int>( 4, 5 )
print<int>( 5 )
print()
```

トリックは、実引数をすべて仮引数パックで受けるのではなく、一つを除いた残りを受け形にすることだ。これにより、実引数をひとつづつ減らしながら再帰的にテンプレートを実体化して呼び出すことができる。仮引数パックが空になった時に、コンパイル時再帰を正しく終了するために、仮引数を取らない非テンプレートな関数を用意している。

主ナ 仮引数パックを使わない関数テンプレートの実体化は 仮引数パックを使った関数

テンプレートの実体化より、オーバーロード解決で優先される。たとえば、二つ以上の任意個の実引数を取り、最小の値を返す関数テンプレートminは、以下のように実装できる。

```
// 終了条件
template < typename T >
T min( T a1, T a2 )
{
    return a1 < a2 ? a1 : a2 ;
}

template < typename T, typename ... Types >
T min( T head, Types ... tail )
{
    return min( head, min( tail ... ) ) ;
}
```

クラステンプレートのテンプレート仮引数パックの場合も、同様に、固定長へのパック展開か、再帰的な汎用コードが使える。

固定長へのパック展開の例は、以下の通り。

```
template < typename T1, typename T2, typename T3 >
struct type_list_impl
{ } ;

template < typename ... pack >
struct type_list
    : type_list_impl< pack ... >
{ } ;
```

明らかに、このコードは汎用的ではない。それに、クラステンプレートには、部分的特殊化があるので、このようなことはしなくても固定長への特殊化はできる。

```
// Primary Class Template
template < typename ... pack >
struct type_list ;

template < typename T1 >
struct type_list< T1 >
```

```
{ } ;  
  
template < typename T1, typename T2 >  
struct type_list< T1, T2 >  
{ } ;  
  
template < typename T1, typename T2, typename T3 >  
struct type_list< T1, T2, T3 >  
{ } ;
```

ただし、固定長の別の関数にパック展開する関数仮引数パックと同じように、固定長の部分的特殊化は、汎用的ではない。特定の条件に対する特殊な別の実装のためには適切であっても、汎用的なコードは書けない。

テンプレート仮引数パックを使うクラステンプレートを汎用的に書くには、再帰を使えばよい。再帰の方法として、再帰的に基本クラスから派生する方法と、再帰的にデータメンバーとして持つ方法がある。

```
// 再帰的に基本クラスから派生する方法  
// primary class template  
template < typename ... >  
struct base_class_trick ;  
  
// 部分的特殊化  
template < typename Head, typename ... Tail >  
struct base_class_trick< Head, Tail ... >  
    : base_class_trick< Tail ... >  
{ } ;  
  
// 終了条件  
template < >  
struct base_class_trick< >  
{ } ;  
  
// 再帰的にデータメンバーとして持つ方法  
// primary template  
template < typename ... >  
struct data_member_trick ;  
  
// 部分的特殊化  
template < typename Head, typename ... Tail >  
struct data_member_trick< Head, Tail ... >  
{  
    data_member_trick< Tail ... > tail ;
```

```

} ;

// 終了条件
template < >
struct data_member_trick< >
{ } ;

```

自分自身から派生したり、自分自身をデータメンバーに持っているわけではない。再帰のたびに、別の実体化を発動させてるので、派生したりデータメンバーに持っているのは、別のクラスである。再帰の終了には、部分的特殊化か、明示的特殊化を使う。

これを応用して、任意のテンプレート実引数の型と数をコンストラクターで受け取り、クラスのデータメンバーとして格納するクラスが書ける。

```

template < typename ... >
struct tuple ;

template < typename Head, typename ... Tail >
struct tuple< Head, Tail ... >
    : tuple< Tail ... >
{
    Head data ;
    tuple( Head const & head, Tail const & ... tail )
        : tuple< Tail ... >( tail ... ),
          data( head )
    { }
} ;

// 終了条件
template < >
struct tuple< >
{ } ;

int main()
{
    tuple< int, short, double, float > t( 12, 34, 5.6, 7.8f )
;
}

```

クラステンプレートtupleは、別の実体化から再帰的に派生する。実体化されたtupleは、それぞれ先頭のテンプレート実引数の型をデータメンバーとして持つ。コンストラクターは先頭の型の値と、仮引数パックからなる実引数を取り、先頭の値をデータメンバーに格納して、残りを基本クラスに投げる。これが再帰的に行われるため、すべての実引数を格納することができる。

ただし、tupleから値を取り出すのは、少し面倒だ。なぜならば、対応するクラスの型にキャストしなければならないからだ。

```
int main( )
{
    tuple< int, short, double, float > t( 12, 34, 5.6, 7.8f )
;
    double d = static_cast< tuple< double, float> & >(t).data
;
}
```

これも可変引数テンプレートを使って解決できる。何番目の値が欲しいか実引数として与えれば、その値を返してくれる関数テンプレートを書けばいい。インデックスは0から始まるとする。この関数テンプレートは、以下のような形になる。

```
template < std::size_t I, typename ... Types >
戻り値の型 get( tuple< Types ... > & t )
{
    return static_cast< 対応する値が格納されているクラス型 >
(t).data ;
}

int main( )
{
    tuple< int, short, double, float > t( 12, 34, 5.6, 7.8f )
;
    auto value = get<2>( t ) ; // 5.6
}
```

さて、戻り値の型はどうやって指定すればいいのか。それには、インデックスを指定すれば、その型を返してくれるメタ関数を書けばよい。

```
template < std::size_t I, typename T >
struct tuple_element ;

template < std::size_t I, typename Head, typename ... Types >
struct tuple_element< I, tuple< Head, Types ... > >
    : tuple_element< I - 1, tuple < Types ... > >
{
    static_assert( I < 1 + sizeof...( Types ), "index exceeds
the tuple length." ) ;
```

```

} ;

template < typename Head, typename ... Types >
struct tuple_element < 0, tuple< Head, Types ... > >
{
    using type = Head ;
} ;

int main( )
{
    using type = tuple< int, short, double, float > ;
    tuple_element< 2, type >::type d ; // double
}

```

tuple_elementは、0から始まるインデックスを数値として指定すると、対応する型を、ネストされた型名typeとして返すメタ関数だ。

次に、キャストすべき型を返してくれるメタ関数をつくる。

```

template < std::size_t, typename ... >
struct tuple_get_type ;

template < std::size_t I, typename Head, typename ... Tail >
struct tuple_get_type< I, Head, Tail ... >
    : tuple_get_type< I-1, Tail ... >
{ } ;

template < typename Head, typename ... Tail >
struct tuple_get_type< 0, Head, Tail ... >
{
    using type = tuple< Head, Tail ... > ;
}

int main()
{
    // typeはtuple< double, float >
    using type = tuple_get_type< 2, int, short, double, float
>::type ;
}

```

この二つのメタ関数を合わせると、関数テンプレートgetは、以下のように書ける。

```
template < std::size_t I, typename ... Types >
typename tuple_element< I, tuple< Types ... > >::type &
get( tuple< Types ... > & t )
{
    return static_cast< typename tuple_get_type<I, Types ...
>::type & >(t).data ;
}
```

まだまだ、面白い技法はたくさんあるのだが、本書はテンプレートメタプログラミングの解説書ではないので、ここで筆を止める。さらに深く調べたい者は、標準ライブラリのtupleやfunctionやbindから始めるといいだろう。もし十分な需要があれば、C++11によるテンプレートメタプログラミングの本も執筆するかもしれない。

14.4.4 friend

friend宣言はテンプレートとして宣言できる。また、friend宣言はテンプレートの特殊化を指定できる。

非テンプレートなfriend宣言で、クラステンプレートを指定するには、特殊化を指し示していなければならない。関数テンプレートは、特殊化か、あるいは実引数推定されるものを指し示さなければならない。

```
template < typename T >
class X ;

template < typename T >
void f( T ) ;

template < typename T >
class Y
{
    friend class X ; // エラー

    friend class X<int> ; // OK
    friend void f( int ) ; // OK

    friend class X<T> ; // OK
    friend void f( T ) ; // OK
    friend void f<double>(double) ; // OK
```

```
} ;
```

friendで指定されたテンプレートの特殊化のみのクラス、あるいはクラステンプレートのfriendとなる。その他の特殊化はfriendとはならない。

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

template < typename T >
class X
{
private :
    int data ;

    friend void f<int>( ) ;
} ;

template < typename T >
void f()
{
    Y y ;
    y.data = 0 ;
}

int main( )
{
    f<int>() ; // OK、f<int>はXのfriend
    f<double>() ; // エラー、f<double>はXのfriendではない
}
```

friendテンプレートは、クラステンプレートと関数テンプレートの全ての実体化に対して働く。

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

template < typename T >
struct Y { } ;
```

```
class X
{
private :
    int data ;

    // 関数テンプレートfに対するfriendテンプレート
    template < typename T >
    friend void f() ;
    // クラステンプレートYに対するfriendテンプレート
    template < typename T >
    friend class Y ;

} ;

template < typename T >
void f()
{
    X x ;
    x.data = 0 ;
}

int main( )
{
    f<int>() ; // OK
    f<double>() ; // OK
}
```

クラステンプレートのすべての特殊化のメンバー関数をfriendにする場合は、friendテンプレートを使う。

```
template < typename T >
struct X
{
    static void f() ;
} ;

class Y
{
private :
    int data ;
    // クラステンプレートXのすべての特殊化のメンバー関数fに対する
    friendテンプレート
```

```

template < typename T >
friend void X<T>::f() ;
} ;

template < typename T >
void X<T>::f()
{
    Y y ;
    y.data = 0 ;
}

int main( )
{
    X<int>::f() ; // OK
    X<double>::f() ; // OK
}

```

friendテンプレートは、ローカルクラスでは宣言できない。

friendテンプレートは、部分的特殊化できない。

```

template < typename T >
struct X { } ;

struct Y
{
    // エラー、部分的特殊化はできない
    template < typename T >
    friend struct X< T * > ;
}

```

14.4.5 クラステンプレートの部分的特殊化(Class template partial specializations)

クラステンプレートの部分的特殊化(Class template partial specializations)は、特殊化(specialization)という名前がついているが、テンプレートの実体化の結果生成される特殊化や、明示的特殊化とは異なる。これは部分的な特殊化であって、テンプレートである。

テンプレート宣言の際のテンプレート名が、識別子だけの基本となるクラステンプレートを、プライマリークラステンプレート(Primary class template)と呼ぶ。俗に、プライマリーテンプレートとも呼ばれる。

```
// プライマリークラステンプレート
template < typename T >
class Identifier ;
```

クラステンプレートの部分的特殊化は、この基準となるプライマリークラステンプレートの一部を特殊化するものである。その宣言方法は、先に宣言したテンプレートと同名で、テンプレート実引数を加えた形のテンプレートIDで宣言する。

```
// プライマリークラステンプレートX
template < typename T >
struct X { } ;

// Xの部分的特殊化
template < typename T >
struct X< T * >
{ } ;
```

プライマリークラステンプレートは、部分的特殊化よりも先に宣言されていなければならない。

部分的特殊化は、直接参照することはできない。部分的特殊化のあるテンプレートを使う際に、最もテンプレート実引数に対して特殊化されたテンプレートが選ばれて実体化される。

```
// #1
template < typename T >
struct X { } ;

// #2
template < typename T >
struct X< T * > { } ;

// #3
template < typename T >
struct X< T & > { } ;

int main( )
{
    X< int > x1 ; // #1
    X< int * > x2 ; // #2
```

```
X< int & > x3 ; // #3
}
```

この例では、テンプレート実引数int *は、T *が最も特殊化されているので、#2が選ばれる。

部分的特殊化は、名前通り部分的に特殊化していればいい。一部に具体的な型や値やテンプレートを与えることもできる。

```
// #1
template < typename T1, typename T2 >
struct X { } ;

// #2
template < typename T >
struct X< T, T > { } ;

// #3
template < typename T >
struct X< T, double > { } ;

int main( )
{
    X< int, int > x1 ; // #2
    X< int, double > x2 ; // #3

    // エラー、曖昧
    X< double, double > x3 ;
}
```

プライマリークラステンプレートを使うつもりがないのであれば、プライマリークラステンプレートは宣言するだけで、定義しなくてもよい。以下の例は、テンプレート実引数としてポインター型だけを受け取るテンプレートである。

```
// プライマリークラステンプレートの宣言
// 定義はしない
template < typename T >
struct RequirePointerType ;

// 部分的特殊化
template < typename T >
struct RequirePointerType< T * > { } ;
```

```

int main( )
{
    RequirePointerType< int * > x1 ; // OK
    RequirePointerType< int > x2 ; // エラー、定義がない
}

```

このテンプレートに、ポインター以外の型をテンプレート実引数として渡しても、定義がないために、エラーとなる。

部分的特殊化のテンプレート仮引数の数は、プライマリークラステンプレートには左右されない。ただ、部分的特殊化として、テンプレートIDに指定する仮引数の数が一致していればよい。

```

// #1
template < typename T >
struct X { } ;

// #2
template < typename T, template < typename > class Temp >
struct X< Temp<T> >
{ } ;

int main( )
{
    X< X<int> > x ;
}

```

この例では、`X<int>`には#1のテンプレートが使われ、`X< X<int> >`には、#2のテンプレートが使われる。

テンプレートIDに与えるテンプレート実引数の数が、プライマリークラステンプレートに一致していなければ、エラーとなる。

```

// #1
template < typename T1, typename T2 >
struct X { } ;

// エラー
template < typename T >
struct X< T > ;

```

```
// エラー
template < typename T1, typename T2, typename T3 >
struct X< T1, T2, T3 > ;
```

また、プライマリークラステンプレートのテンプレート仮引数の種類、すなわち、型テンプレート、非型テンプレート、テンプレートテンプレートに一致していなければならない。

```
// 型、非型(int型)、テンプレート
template < typename T, int I, template < typename > class
Temp >
struct X ;
```



```
// OK
template < typename T, template < typename T > class Temp >
struct X< T, 0, Temp<T> > { } ;
```



```
// エラー、テンプレート仮引数の種類が一致していない。
template < typename T >
struct X< 0, T, T > { } ;
```

テンプレートテンプレート仮引数の場合は、テンプレートテンプレート仮引数のテンプレート仮引数の数や種類にも対応していなければならない。

```
// プライマリークラステンプレート
template < template < typename > class Temp >
struct X { } ;
```



```
// エラー、テンプレートテンプレート仮引数のテンプレート仮引数の数
// が一致していない
template < template < typename, typename > class Temp >
struct X< Temp > ;
```



```
// エラー、テンプレートテンプレート仮引数のテンプレート仮引数の種
// 類が一致していない
template < template < int > class Temp >
struct X< Temp > ;
```

可変引数テンプレートの場合は、0個以上の任意の数に特殊化できる。

```
// プライマリークラステンプレート
```

```
template < typename ... >
struct X ;

template < typename T >
struct X< T > ;

template < typename T1, typename T2 >
struct X< T1, T2 > ;

template < typename T1, typename ... Rest >
struct X< T1, Rest ... > ;
```

部分的特殊化は、プライマリークラステンプレートが宣言された名前空間スコープやクラススコープの外側で宣言できる。

```
namespace NS
{
    template < typename T >
    struct Outer
    {
        template < typename U >
        struct Inner ;
    } ;

    template < typename T >
    template < typename U >
    struct NS::Outer< T >::Inner< U * > { } ;
}
```

部分的特殊化の宣言中のテンプレートIDの実引数には、いくつかの制限が存在する。非型実引数の式は、部分的特殊化のテンプレート仮引数の識別子のみである時以外は、テンプレート仮引数と関わってはならない。

```
template < int I, int J >
struct X ;

// OK、識別子のみ
template < int I >
```

```
struct X< I, I > ;

// エラー、部分的特殊化のテンプレート仮引数が関わる式
template < int I >
struct X< I+1, I+2 > ;
```

非型実引数の型は、部分的特殊化のテンプレート仮引数に依存してはならない。

```
template < typename T, T I >
struct X ;

// エラー
template < typename T >
struct X< T, 0 > ;

template < int I, int ( * Array_ptr)[I] >
struct Y ;

int array[5] ;

// エラー
template < int I >
struct Y< I, &array > ;
```

部分的特殊化の実引数リストは、プライマリークラステンプレートに暗黙的に生成される実引数リストと同一であってはならない。

```
template < typename T1, typename T2 >
struct X ;

template < typename T1, typename T2 >
struct X< T1, T2 > ;
```

部分的特殊化のテンプレート仮引数には、デフォルトテンプレート実引数は使えない。

部分的特殊化のテンプレート実引数には展開されていない仮引数パックがあつてはならない。テンプレート実引数がパック展開ならば、最後に記述されなければならない。

```
template < typename ... >
struct X ;
```

```
// エラー、展開されていない仮引数パック
template < typename T, typename ... Pack >
struct X< T, Pack > ;

// エラー、仮引数パックは最後に記述されなければならない
template < typename T, typename ... Pack >
struct X< Pack ..., T > ;

template < typename ... >
struct wrap { } ;

// OK
template < typename ... Pack, typename T >
struct X< wrap<Pack...>, T > ;
```

14.4.5.1 クラステンプレートの部分的特殊化の一一致度の比較 (Matching of class template partial specializations)

クラステンプレートの部分的特殊化は、直接参照することはできない。クラステンプレートを使った時、プライマリークラステンプレートや部分的特殊化が比較され、最もテンプレート実引数に対して特殊化されたテンプレートが選ばれる。

```
template < typename T >
struct X { } ; // #1

template < typename T >
struct X< T * > { } ; // #2

template < typename T >
struct X< T const * > { } ; // #3

int main( )
{
    X< int > x1 ; // #1
    X< int * > x2 ; // #2
    X< int const * > x3 ; // #3
}
```

最適なテンプレートは、テンプレート実引数が、部分的特殊化のテンプレート実引数に、いかに一致しているかを比較することにより選択される。

この比較は以下のように行われる。

- 一致する部分的特殊化が、ただひとつだけ発見された場合、その部分的特殊化が選ばれる。

```
// プライマリークラステンプレート
template < typename T1, typename T2 >
struct X { } ;

// 部分的特殊化
template < typename T1, typename T2 >
struct X< T2, T1 > { } ;

int main( )
{
    X< int, int > x ; // 部分的特殊化が実体化される
}
```

これは極端な例だが、この例では、部分的特殊化はプライマリーテンプレートと同一ではない。`X< int, int >`には、プライマリークラステンプレートと部分的特殊化の両方が一致するが、ただひとつの部分的特殊化が一致するために、部分的特殊化が選ばれる。

一致する部分的特殊化が一つでもある場合、プライマリークラステンプレートが使われることはない。

- 二つ以上の一致する部分的特殊化が発見された場合、14.4.5.2 半順序の規則により、最も特殊化されている部分的特殊化が選ばれる。もし、他のすべての部分的特殊化よりもさらに特殊化している部分的特殊化が見つからない場合、結果は曖昧となり、エラーとなる。

```
template < typename T >
struct X { } ; // #1

template < typename T >
struct X< T const > { } ; // #2

template < typename T >
struct X< T const volatile > { } ; // #3

int main( )
{
```

```
X< int const > x1 ; // #2
X< int const volatile > x2 ; // #3

X< int > x3 ; // #1
}
```

以下のような場合は、曖昧でエラーとなる。

```
template < typename T1, typename T2 >
struct X { } ;

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

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

int main( )
{
    X< int, int > x ; // エラー、曖昧
}
```

この例では、プライマリークラステンプレートと、二つの部分的特殊化の、どのテンプレートを使っても実体化できる。ただし、二つ以上の一一致する部分的特殊化があるために、プライマリークラステンプレートは使われない。二つの部分的特殊化は、どちらがより特殊化されているとも決定できないので、曖昧となる。

- 一致する部分的特殊化が発見されなかった場合、プライマリークラステンプレートが使われる。

部分的特殊化が一致するかどうかは、テンプレート実引数から、部分的特殊化のテンプレート実引数を導けるかどうかで判断される。

```
template < typename T1, typename T2 >
struct X { } ;

template < typename T >
struct X< T, T > { } ; // #1

template < typename T >
struct X< T, int > { } ; // #2
```

```

template < typename T1, typename T2 >
struct X< T1 *, T2 > {} ; // #3

int main( )
{
    X< int, int > x1 ; // #1, #2に一致
    X< short, int > x2 ; // #2に一致
    X< int *, int * > x3 ; // #1, #3に一致
    X< int const *, int const * > x4 ; // #1, #3に一致
    X< int *, int > x5 ; // #3に一致

    X< int, short > x6 ; // 一致する部分的特殊化なし
}

```

14.4.5.2 クラステンプレートの部分的特殊化の半順序(Partial ordering of class template specializations)

二つのクラステンプレートの部分的特殊化の間で、どちらがより特殊化されているかということを、半順序(partial ordering)という。クラステンプレートの部分的特殊化の半順序は、比較のために部分的特殊化を関数テンプレートに書き換えた上で、関数テンプレートの半順序に従って決定される。

部分的特殊化の比較のための関数テンプレートへの書き換えは、以下のように行われる。

書き換えた関数テンプレートは、元の部分的特殊化と同じテンプレート仮引数を持つ。この関数テンプレートはひとつの仮引数をとる。仮引数の型は、元の部分的特殊化のクラステンプレート名に、テンプレート実引数として、部分的特殊化と同じ記述したものである。

たとえば、以下のような部分的特殊化の場合は、

```

template < typename T1, typename T2, typename T3 >
struct X { } ;

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

```

比較用の関数テンプレートへの書き換えは、以下のようになる。

```
template < typename T >
void f( X< T, T, T > ) ;
```

以下の二つの部分的特殊化を比較する場合、

```
template < typename T1, typename T2, typename T3 >
struct X { } ;

// #1
template < typename T1, typename T2 >
struct X< T1, T1, T2 > { } ;

// #2
template < typename T >
struct X< T, T, T > { } ;
```

以下のように、関数テンプレートに書き換えられて、関数テンプレートの半順序により判断される。

```
// #1
template < typename T1, typename T2 >
void f( X< T1, T1, T2 > ) ;

// #2
template < typename T >
void f( X< T, T, T > ) ;
```

この例では、#2の方がより特殊化されている。

14.4.5.3 クラステンプレートの特殊化のメンバー

プライマリークラステンプレートと部分的特殊化の間は別物である。それぞれ異なるメンバーの宣言と定義を持つ。

```
template < typename T >
struct X
```

```

{
    void foo() ;
} ;

template < typename T >
struct X< T * >
{
    void bar() ;
} ;

int main()
{
    X< int > x1 ;
    x1.foo() ; // OK
    x1.bar() ; // エラー

    X< int * > x2 ;
    x2.foo() ; // エラー
    x2.bar() ; // OK
}

```

部分的特殊化のメンバーをクラススコープの外で定義する場合、部分的特殊化と同じテンプレート仮引数とテンプレート実引数を使わなければならない。

```

template < typename T >
struct X ;

template < typename T >
struct X< T * >
{
    // メンバーの宣言
    void bar() ;
} ;

// メンバーの定義
template < typename T >
void X< T * >::bar() { }

```

メンバーテンプレートも部分的特殊化できる。

```
template < typename T >
```

```

struct class_template
{
    // プライマリーメンバークラステンプレート
    template < typename U >
    struct member_template { } ;

    // 部分的特殊化
    template < typename U >
    struct member_template< U * > { } ;
};

```

14.4.6 関数テンプレート(Function templates)

関数テンプレートは、特定の型にとらわれない関数のテンプレートを記述できる。

```

template < typename T >
void f( T param ) { }

```

関数テンプレートは、クラステンプレートと同じように、テンプレート実引数を指定して実体化させ、呼び出すことができる。

```

template < typename T >
void f( T param ) { }

int main( )
{
    f<int>( 0 ) ;
    f<double>( 0.0 ) ;
}

```

関数テンプレートは、テンプレート実引数を指定せずに呼び出すことができる。この場合、関数の実引数から、テンプレート実引数が導かれる。これを、実引数推定(Argument Deduction)という。

```

template < typename T >
void f( T param ) { }

```

```
int main( )
{
    f( 0 ) ; // f<int>
    f( 0.0 ) ; // f<double>
}
```

実引数推定できない場合、エラーとなる。

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

int main( )
{
    f( 0 ) ; // エラー、実引数推定できない
}
```

より詳しくは、14.7.2 [テンプレートの実引数推定](#)を参照。

14.4.7 関数テンプレートのオーバーロード(Function Template Overloading)

関数テンプレートはオーバーロードできる。オーバーロードは、関数テンプレートと通常の関数の区別なく記述できる。

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

template < typename T >
void f( T * ) ;

void f( int ) ;
```

異なる複数の関数テンプレートが同じテンプレート実引数に対して実体化できる場合、それぞれ異なる実体を持つので、ODR違反とはならない。

たとえば、ある一つのプログラムを構成する二つのソースファイルがあり、それぞれ以下のように記述されていたとする。

```
// ソースファイル1
template < typename T >
void f( T ) { }

void g( int * p )
{
    f( p );
}
```

```
// ソースファイル2
template < typename T >
void f( T * ) { }

void h( int * p )
{
    f( p );
}
```

この場合、それぞれのテンプレートから、それぞれ実体化が行われ、異なる特殊化が使われる。ODR違反とはならない。

関数テンプレートのオーバーロードは、実体化された特殊化が、全く同じシグネチャであっても構わない。

```
template < typename T > void f( ) ;
template < int I > void f( ) ;
```

テンプレート仮引数が、関数テンプレートの仮引数リストや戻り値の型における式の中で参照された場合、その式は関数テンプレートのシグネチャの一部になる。これにより、式の違いによる異なる関数テンプレートを記述できる。

```
template < int I >
struct X { } ;

template < int I, int J >
X< I + J > f( X<I>, X<J> ) ; // #1の宣言

template < int K, int L >
X< K + L > f( X<K>, X<L> ) ; // #1の再宣言
```

```
template < int I, int J >
X< I - J > f( X<I>, X<J> ) ; // #2、これは#1とは異なる宣言
```

最初の二つの関数テンプレートは、同一の関数テンプレートである。しかし、#2は式が違うため、異なる関数テンプレートである。

この時、シグネチャの式を評価した結果が同じものを、「機能的に同一」という。シグネチャの式が同じものを「同一」という。機能的に同一だが、同一ではない二つの宣言がある場合、エラーとなる。

```
template < int I >
struct X { } ;

template < int I >
void f( X< I + 2 > ) ;

template < int I >
void f( X< I + 1 + 1 > ) ; // エラー、機能的に同一だが、シグネチャの式が同一ではない
```

ただし、規格上、実装はこの誤りを検出して報告する必要はない。したがって、このエラーはコンパイルで見つけることは期待できない。よく注意しなければならない。

14.4.8 関数テンプレートの部分的特殊化(Partial ordering of function templates)

関数テンプレートがオーバーロードされている場合、どの関数テンプレートの特殊化を使うべきなのか曖昧になる。

```
template < typename T >
void f( T ) { } // #1

template < typename T >
void f( T * ) { } // #2

int main()
{
    int * p = nullptr ;
    f( p ) ; // #2の特殊化が呼ばれる
```

```
void ( *fp )( int * ) = &f ; // #2の特殊化のアドレスを得る
}
```

以下の文脈の場合、半順序(partial ordering)によって、最も特殊化されているテンプレートを決定する。

- 関数テンプレートの特殊化を呼び出す際のオーバーロード解決
- 関数テンプレートの特殊化のアドレスを取得するとき
- プレイスマントoperator newに一致するプレイスマントoperator deleteを選択するとき
- friend関数宣言、明示的実体化、明示的特殊化が、ある関数テンプレートの特殊化を参照しているとき

```
template < typename T >
void f( T ) { } // #1

template < typename T >
void f( T * ) { } // #2

class X
{
    int data ;
    template < typename T >
    friend void f( T ) ; // #1をfriendに指定
} ;

// #2の明示的実体化
template void f( int * ) ;

// #1の明示的特殊化
template < >
void f<double>( double ) { }
```

半順序は、複数のテンプレートの特殊化から、どれがより特殊であるかを選ぶ。

14.4.9 エイリアステンプレート(Alias templates)

宣言部分がエイリアス宣言のテンプレート宣言を、エイリアステンプレートと呼ぶ。エイリアステンプレートは、複数の型名をテンプレート化することができる。いわば、`typedef`

のテンプレート版とも言える。

```
template < typename T >
struct wrap { } ;

template < typename T >
using Alias = wrap<T> ;

Alias<int> t1 ; // wrap<int>
Alias<double> t2 ; // wrap<double>
```

エイリアステンプレートによって宣言されたテンプレートIDは、エイリアスされた型の別名として使うことができる。エイリアステンプレートのテンプレートIDは、typedef名と同じく、型の別名であり、別の型ではない。Alias<int>は、wrap<int>と同一の型である。

エイリアステンプレートの利用例を挙げる。

```
template < typename T, typename U >
struct wrap
{
    using type = T ;
} ;

template < typename T >
using a1 = wrap< T, int > ;
using t1 = a1<int> ; // wrap< int, int >

template < typename T >
using a2 = wrap< T, T > ;
using t2 = a2< int > ; // wrap< int, int >

template < typename T >
using a3 = typename wrap< T, void >::type ;
using t3 = a3<int> ; // wrap< int, void>::type、すなわちint
```

エイリアステンプレートは、その利用方法はさておき、以下のような記述もできる。

```
template < typename T >
using a1 = T ; // a1<T>はTの別名

template < typename T >
```

```
using a2 = int ; // a2<T>はintの別名
```

エイリアステンプレートは、テンプレート実引数の一部のみ指定し、残りをテンプレート化することができる。その利用方法は、例えば、カスタムアロケーターを指定したコンテナテンプレートの別名を宣言できる。

```
class MyAlloc ;

template < typename T >
using MyVec = std::vector< T, MyAlloc > ;
```

従来のtypedef宣言では、これができるない。

エイリアステンプレートは、名前通りテンプレートであるので、名前空間スコープかクラススコープの内側でしか宣言できない。たとえば、関数のブロックスコープの内側では宣言できない。

```
void f()
{ // 関数のブロックスコープ
    template < typename T > using A = T ; // エラー
}
```

エイリアステンプレート宣言内のテンプレートIDは、宣言中のエイリアステンプレートを参照してはならない。つまり、宣言中に自分自身の特殊化を使ってはならないという事である。

```
template < typename T > struct A ;
template < typename T > using B = typename A<T>::U;
template < typename T > struct A
{
    typedef B<T> U;
} ;

// エラー、B<int>の実体化の際に、A<int>::Uとして、自分自身を使ってしまう。
B<int> b;
```

14.5 名前解決(Name Resolution)

テンプレート定義内の名前解決は非常に複雑である。これは、テンプレートはある場所で宣言され、別の場所で特殊な形に実体化されるからである。

テンプレート定義内では、三種類の名前がある。

- テンプレート自身の名前、テンプレートで宣言された名前
- テンプレート仮引数に依存する名前
- テンプレート定義のあるスコープから見える名前

「テンプレート自身の名前、テンプレートで宣言された名前」というのは、テンプレート名と、テンプレート仮引数名である。

「テンプレート仮引数に依存する名前」は、依存名(Dependent Name)と呼ばれている。

「テンプレート定義のあるスコープから見える名前」とは、テンプレート定義のあるスコープやその外側のスコープで、すでに宣言された名前のことだ。

テンプレート仮引数に依存する名前は、暗黙に型を意味しないものと解釈される。

```
template < typename T >
void f()
{
    int x = T::value ; // T::valueは型ではない
}
```

この場合、Tに与えられるテンプレート実引数には、例えば以下のようなものが想定されている。

```
struct X
{
    static constexpr int value = 0 ;
};
```

テンプレート宣言や定義で、テンプレート仮引数に依存する名前を型として使おうとしてもエラーとなる。なぜならば、すでに述べたように、暗黙に型を意味しないものと解釈されるからだ。

```
template < typename T >
void f()
{
    typedef T::type type ; // エラー、T::typeは型ではない
}
```

依存名を型であると解釈せらるには、明示的に、名前の直前に、`typename`キーワードを記述して修飾しなければならない。

```
template < typename T >
void f()
{
    typedef typename T::type type ; // OK、T::typeは型
}
```

ただし、メンバー初期化子と基本クラス指定子には、文法上型しか記述できないので、`typename`で修飾する必要はない。

```
template < typename T >
struct X : T::type // OK
{
    X() : T::type() // OK
    { }
} ;
```

また、メンバーテンプレートは、文脈により、テンプレートかどうかが曖昧になる。

```
// Tに渡す型の例
struct X
{
    template < typename T >
    void func() ;
} ;

template < typename T >
void f()
{
    T t ;
    t.func<int>(0) ; // エラー
}
```

このコードの意味は、`t.func`と`int`に、比較演算子である`<`を適用し、さらに比較演算子`<`と`>`に囲まれた`0`を適用するものである。メンバー関数テンプレートの特殊化を呼び出すものではない。`t.func`と`int`を`<`演算子で比較するのは、文法上認められていないので、このコードはエラーになる。

依存名に`.や->`、あるいは`::`を用い、メンバーテンプレートの特殊化を記述する場合は、

メンバーテンプレートは、`template`キーワードで修飾しなければならない。これは、メンバー名がテンプレートであると明示的に解釈させるためである。

```
// Tに渡す型の例
struct X
{
    template < typename T >
    struct MemberClass ;

    template < typename T >
    void MemberFunction() ;

} ;

template < typename T >
void f()
{
    typedef typename T:: template MemberClass<int> obj ;

    T t ;
    t. template MemberFunction<int>() ;

    T * p = &t ;
    p-> template MemberFunction<int>() ;
}
```

これはメンバーテンプレートの特殊化を使う場合であって、メンバー関数テンプレートを実引数推定させて使う場合には、`template`キーワードを記述する必要はない。

```
// Tに渡す型の例
struct X
{
    template < typename T >
    void MemberFunction( T ) ;

} ;

template < typename T >
void f()
{
    T t ;
    t.MemberFunction( 0 ) ; // OK
}
```

templateキーワードの指定は、実際には文法の曖昧性の問題であって、名前解決の問題ではないのだが、typenameキーワードの指定と似ているために、便宜上、本書では同時に説明することにした。

14.5.1 依存(Dependent)

依存(Dependent)とは、テンプレート仮引数に依存することである。テンプレート仮引数に依存するものは、名前と式とテンプレート実引数である。式とテンプレート実引数には、型依存式と値依存式が存在する。

なぜテンプレート仮引数に依存しているかどうかが問題になるのか。テンプレート仮引数というのは、具体的な内容が確定していない存在だからだ。テンプレートは、実体化されて初めて、その具体的な内容が確定する。

依存の詳細は煩雑になるので省略するが、簡略化していえば、テンプレート仮引数が関わる名前や式は、すべて依存している。

```
void f( int ) ;

template < typename T >
struct identity
{
    using type = T ;
} ;

template < typename T >
struct X
{
    int data ;

    void member()
    {
        T t1 ; // Tは依存名
        T::value ; // 依存している。値と解釈される
        typename T::type t2 ; // 依存している。型と解釈される

        f( 0 ) ; // 依存していない

        &X::data ; // 依存している
        this->data ; / 依存している

        typename identity<T>::type t ; // 依存している
    }
} ;
```

クラステンプレートの場合、クラス名やthisを介した式も、テンプレート仮引数に依存している。なぜならば、クラステンプレートの場合、クラス名自体がテンプレート仮引数に依存しているからだ。

依存していない名前や式を、非依存(Non-dependent)という。

14.5.2 非依存名の名前解決

非依存名は、テンプレートが定義されている場所で名前解決される。

```
void f( int ) ;

template < typename T >
void g()
{
    f( 0 ) ; // f(int)
}

void f( double ) ;

int main()
{
    g<int>() ;
}
```

このコードの解釈は驚くにあたらない。ただし、状況によっては、意図しないことが起こる。

```
// Derivedのテンプレート仮引数Baseが想定している型
struct Base
{
    void member() { }
} ;

template < typename Base >
struct Derived : Base
```

```
{
    void f()
    {
        member() ; // エラー、memberが見つからない
    }

} ;
```

この、テンプレートクラス、Derivedは、テンプレート仮引数を基本クラスに指定している。そして、基本クラスはmemberという名前のメンバー関数を持っていることを期待している。Derived::f内で使われている、memberという名前は、非依存名であり、しかも非修飾名なので、メンバー関数であるとは解釈されない。そのため、外側のスコープのmemberという名前を探すが、見つからぬためエラーになる。

このようなコードで、memberをメンバーとして扱いたい場合、memberを修飾して依存名にする必要がある。それには、三種類の方法がある。

```
template < typename Base >
struct Derived : Base
{
    void f()
    {
        Derived::member() ; // OK、クラス名は依存名
        this->member() ; // OK、thisは依存式
        Base::member() ; // OK、テンプレート仮引数は依存名
    }

} ;
```

14.5.3 依存名の名前解決

依存名の名前解決は、実体化場所(Point of Instantiation)が重要になる。

実体化場所とは、テンプレートが実体化された場所のことである。

```
template < typename T >
void f( T ) { } // テンプレートの定義

struct Foo { } ;
```

```
int main()
{
    Foo foo ;
    f( foo ) ; // 実体化場所
}
```

依存名の名前解決は、テンプレートの定義ではなく、実体化場所で行われるので、テンプレートの定義の時点では見えていない、Fooという名前も使うことができる。また、テンプレートの定義中で、テンプレート仮引数のメンバーネームやネストされた型名を参照しても、実体化の結果が一致しているならば、名前解決できる。

テンプレートの名前解決の理解を難しくしているのは、オーバーロード解決における、候補関数の見え方である。

非修飾名前解決と修飾名前解決を使った候補関数は、テンプレートの定義場所から見える名前のみに制限される。

```
void f( int ) { }

// テンプレートの定義
template < typename T >
void call( )
{
    f( 0.0 ) ; // 候補関数はf(int)のみ
}

void f( double ) { }

int main()
{
    call<void>() ; // 実体化場所
}
```

このように、候補関数の非修飾名前解決と修飾名前解決は、実体化場所で行われるもの、発見される名前は、テンプレートの定義場所から見える名前のみに限定されているため、上記の例では、もし、f(double)が候補関数に含まれていたならば、そちらが最適関数だが、候補関数として発見されないために、最適関数になることもない。

同様に、以下の例はエラーとなる。

```
// テンプレートの定義
template < typename T >
void call( )
{
```

```

    f( 0 ) ; // エラー、名前fが見つからない
}

void f( int ) { }

int main()
{
    call<void>() ; // 実体化場所
}

```

ただし、ADLの場合は、例外的に異なる。ADLが発動した場合は、テンプレートの実体化場所から見える候補関数が発見される。

```

// グローバル名前空間

// クラスFooの関連名前空間はグローバル名前空間
struct Foo { } ;

// テンプレートの定義
template < typename T >
void call_f( T t )
{
    f( t ) ;
}

// グローバル名前空間内の名前
void f( Foo ) { }

int main()
{
    Foo foo ;
    call_f( foo ) ; // OK、ADLが発動
}

```

この場合、関数call_f内で呼び出している非修飾名fは、非修飾名前解決では見つからないため、ADLが発動する。

これはADLが発動する場合のみの例外的なルールである。ADLが発動しない場合は、このような例外的な挙動にはならない。

```
struct Foo { } ;
```

```

void f( Foo const & ) { } // #1

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

void f( Foo & ) { } // #2

int main()
{
    Foo foo ;
    call_f( foo ) ; // ADLは発動しない。#1が呼ばれる
}

```

この場合では、非修飾名前解決により、#1が、名前fとして見つかるため、ADLは発動しない。ADLが発動しないので、#2が候補関数に選ばれることもない。もし、#2が候補関数に選ばれていたならば、オーバーロード解決により、#2は#1より最適な関数となるが、ADLが発動しない以上、#2は発見されず、したがって候補関数にもならない。

また、基本型には、関連名前空間が存在しないため、ADLは発動しない。

このように、テンプレート内の名前は、テンプレートの定義場所と、実体化場所で、二段階に分けて名前解決されるので、二段階名前解決(Two Phase Lookup)と呼ばれている。

14.6 テンプレートの実体化と特殊化(Template instantiation and specialization)

テンプレートは、テンプレート実引数を与えられて実体化して始めて利用可能になる。これをテンプレート実体化(template instantiation)という。実体化には、暗黙の実体化と明示的な実体化がある。実体化したテンプレートのことを、特殊化(specialization)という。特殊化は、明示的に行うこともできる。テンプレートの部分的特殊化は、名前が似ているが、いまだにテンプレートであって、実体化された特殊化ではない。

14.6.1 暗黙の実体化(Implicit instantiation)

明示的に実体化されず、明示的に特殊化されていないテンプレートは、オブジェクトの完全な型が必要な場合や、クラス型が完全であることがプログラムの意味に影響を与える文脈で参照された場合に、暗黙に実体化される。

クラステンプレートが暗黙に実体化されても、クラステンプレートのメンバーまで暗黙に実体化されるわけではない。

```

template < typename T >
struct X
{
    void f() ;
    void g() ;
} ;

int main()
{
    typedef X< int > type ; // X<int>の実体化は必要ない
    type a ; // X<int>の実体化が必要
    X< char > * b ; // X<char>の実体化は必要ない
    X< double > * p ; // X<double>の実体化は必要ない

    a.f() ; // X<int>::f()の実体化が必要
    b->f() ; // X<char>::f()の実体化が必要
}

```

`typedef`名やポインター型の宣言は、クラスの完全な型が必要な文脈ではないので、テンプレートの暗黙の実体化は起こらない。

`X<char>`や `X<double>`の実体化が必要ないのは、クラスへのポインターを参照しているだけなので、クラスの完全な型が必要な文脈ではないからである。また、`X<int>::g()`や `X<double>::g()`も、参照されていないので実体化はされない。

関数テンプレートも、定義が必要な文脈で参照されなければ、暗黙に実体化されることはない。

実体化の必要のないクラステンプレートのメンバーが暗黙的に実体化されないという挙動は、規格上保証されている。

テンプレートが暗黙に実体化される場合、暗黙的な実体化が必要ない場合、また例外的に暗黙的に実体化されるかどうかが未規定の場合の詳細な解説は、煩雑になるので省略する。

14.6.2 明示的実体化(Explicit instantiation)

クラス、関数、メンバーテンプレートの特殊化は、テンプレートから明示的に実体化できる。メンバー関数、メンバークラス、クラステンプレートのstaticデータメンバーは、クラステンプレートのメンバーの定義として、明示的に実体化できる。これを明示的実体化(Explicit instantiation)という。

明示的実体化を宣言する文法は、以下の通りである。

extern省略可 template 宣言

externキーワードは省略できる。externキーワードの有無に意味上の違いはない。C++03までの規格では、externキーワードを使った文法は、明示的実体化の宣言ではなく、内部リンクージの宣言になり、プログラムの意味が変わってしまうので、注意が必要である。本書はC++11の規格のみを取り扱う。

クラス、もしくはメンバークラスに対する明示的実体化の場合、宣言中のクラス名はテンプレート実引数を指定した形で指定する。

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

// X<int>の明示的実体化
extern template struct X< int > ;
```

externキーワードは省略できるので、上記の明示的実体化は、以下のように書くこともできる。

```
template struct X< int > ;
```

関数、もしくはメンバー関数に対する明示的実体化の場合、宣言中の関数名は、テンプレート実引数を指定しているか、引数リストからテンプレート実引数が推定できる形で指定する。

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

// f<int>の明示的実体化
extern template void f< int >( int ) ;

template < typename T >
void g( T ) { }

// g<int>の明示的実体化
extern template void g( int ) ;
// 以下と同等
// extern template void g< int >( int ) ;
```

クラスのメンバーに対する明示的実体化の場合は、メンバーの属するクラス名はテンプレート実引数を指定した形で指定する。

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

extern template void X<int>::f() ;
```

同じテンプレートとテンプレート実引数に対する明示的実体化は、プログラム中に一度しか現れてはならない。つまり、複数のソースファイルからなるプログラム全体でも、一度しか現れてはならない。規格上、実装はこの違反を検出できるよう規定されてはいないので、実装の出力するコンパイル時、実行時のエラーや警告のメッセージに頼ることは出来ない。

関数テンプレート、メンバー関数、クラステンプレートのstaticデータメンバーは、明示的実体化の前に宣言されなければならない。

```
// エラー、前方に宣言がない
extern template void f<int>() ;

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

クラステンプレート、クラステンプレートのメンバークラス、メンバークラステンプレートは、明示的実体化の前に定義されなければならない。

```
template < typename T >
struct X ; // 宣言

// エラー、クラステンプレートXは前方で定義されていない
extern template struct X<int> ;

template < typename T >
struct X { } ; // 定義
```

明示的実体化で、暗黙に宣言された特別なメンバー関数を指定した場合、エラーとなる。

```

template < typename T >
struct X { } ;

// エラー、暗黙に宣言されたコンストラクター
extern template X<int>::X() ;

template < typename T >
struct Y
{
    Y() { } // 明示的な宣言
} ;

// OK
extern template Y<int>::Y() ;

```

同じテンプレート実引数の明示的特殊化の宣言の後に明示的実体化の宣言が現れた場合、明示的実体化は無効となる。これはエラーではない。

```

template < typename T >
struct X { } ;

// X<int>に対する明示的特殊化
template < >
struct X<int> { } ;

// X<int>に対する明示的実体化。
// 無効、エラーではない
extern template struct X<int> ;

```

同じテンプレート実引数に対する明示的特殊化の前に明示的実体化が現れた場合はエラーである。

```

template < typename T >
struct S { } ;

// エラー、同じテンプレート実引数に対する明示的実体化
extern template struct S< int > ;

// エラー、明示的特殊化の前に同じ明示的実体化
template < >

```

```
struct S< int > { } ;
```

明示的実体化に指定する名前には、通常のアクセス指定は行われない。例えば、テンプレート実引数や関数宣言子に、アクセス指定にprivateな名前を指定することもできる。

```
template < typename T, typename C, T C::* mem_ptr >
struct temp { } ;

struct X
{
private :
    int private_member ;
} ;

// 明示的実体化
// OK、通常のアクセス指定は行われない
extern template struct temp< int, X, &X::private_member > ;

// 暗黙の実体化を伴う変数宣言
// エラー、privateメンバーの使用
temp< int, X, &X::private_member > t ;
```

明示的実体化を使えば、プログラム中のテンプレートを必要とするソースファイルすべてにトークン列が一致するテンプレートの完全な定義を持ち込む必要がなくなる。

```
// func.h
// 関数テンプレートfuncの宣言
template < typename T >
void func( T ) ;
```

```
// func.cpp
// 関数テンプレートfuncの定義
#include "func.h"

template < typename T >
void func( T ) { }

// プログラム中で使われる実体化を明示的に宣言
extern template void func( int ) ;
```

```
extern template void func( double ) ;
```

```
// main.cpp

// このソースファイルmain.cppには、
// 関数テンプレートfuncの宣言のみ導入
#include "func.h"

int main()
{
    func( 0 ) ; // OK、プログラム中で明示的実体化されている
    func( 0.0 ) ; // OK、プログラム中で明示的実体化されている

    func( 'a' ) ; // エラー、定義がないため、実体化できない。
}
```

C++におけるテンプレートは、トークン列が一致するコード片を、テンプレートの特殊化を必要とするプログラム中のソースファイルすべてに持ち込むことで、ODRを例外的に回避している。明示的実体化を使えば、テンプレートの宣言と定義を分離し、すべてのソースファイルに定義を持ち込む必要がなくなる。ただし、明示的に実体化したテンプレートとそのテンプレート実引数に限定される。

14.6.3 明示的特殊化(Explicit specialization)

テンプレートはあるテンプレート実引数について、元となるテンプレートとは別に、明示的に特殊化することができます。これを明示的特殊化(Explicit specialization)という。明示的特殊化を使うと、ある与えられたテンプレート実引数に対しては、汎用のテンプレートから実体化される特殊化は異なる特殊化を与えることができる。明示的特殊化と、部分的特殊化は、名前は似ているは全くの別物である。

明示的特殊化の文法は以下の通り。

```
template < > 宣言
```

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

// 明示的特殊化
```

```
template < >
bool f( int ) { return true ; }

int main()
{
    f( 0 ) ; // true
    f( 0.0 ) ; // false
    f( 'a' ) ; // false
}
```

この例では、関数テンプレートfにテンプレート実引数intを与えた場合だけ、元のテンプレート定義とは別の、明示的特殊化による定義を使用する。そのため、f<int>はtrueを返す。

明示的特殊化は、元のテンプレートの定義の影響を受けない。たとえば、関数テンプレートの場合は戻り値の型を異なるものにできるし、クラステンプレートの場合、クラスのメンバーを全く違ったものにすることもできる。

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

template < >
struct X<int>
{
    void g() ;
};
```

この例では、元のテンプレートの定義であるメンバー関数X::fがなく、全く別名のメンバー関数gを定義している。

明示的特殊化できるテンプレートは、以下の通り。

- 関数テンプレート

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

// 明示的特殊化
template < >
bool f( int ) { return true ; }
```

```
// 明示的なテンプレート実引数の指定によるもの
template < >
void f<short>( short ) { }
```

- クラステンプレート

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

```
// 明示的特殊化
template < >
struct X<int>
{
    int data ;
} ;
```

- クラステンプレートのメンバー関数

```
template < typename T >
struct X
{
    void f() { }
    void g() { }
    int data ;
} ;
```

```
// ひとつのメンバー関数のみを明示的特殊化
template < >
bool X<int>::f() { return true ; }
```

クラステンプレートのメンバー関数を個別に明示的特殊化することができる。この場合、クラステンプレートXにテンプレート実引数intを与えて実体化させた特殊化は、X::fのみ明示的特殊化の定義を使い、残りのメンバーはテンプレートから実体化された特殊化を使う。

- クラステンプレートのstaticデータメンバー

```
template < typename T >
struct X
```

```
{  
    static int data ; // 宣言  
} ;  
  
template < typename T >  
int X<T>::data ; // 定義  
  
// 明示的特殊化  
template < >  
int X<int>::data ;
```

クラステンプレートのstaticデータメンバーの明示的特殊化は、宣言の型を変えることはできない。

```
template < typename T >  
struct X  
{  
    static int d1 ;  
    static T d2 ;  
} ;  
  
// 汎用的な定義  
template < typename T >  
int X<T>::d1 = 0 ;  
  
template < typename T >  
T X<T>::d2 = {} ;  
  
template < >  
double X<int>::d1 ; // エラー、型が宣言と一致しない  
  
template < >  
double X<int>::d2 ; // エラー、型が宣言と一致しない
```

ただし、初期化式を変えることはできる。

```
template < typename T >  
struct X  
{  
    static int data ;  
} ;
```

```
template < typename T >
int X<T>::data = 0 ;

template < >
int X<int>::data = 1 ;

template < >
int X<double>::data = 2 ;
```

- クラステンプレートのメンバークラス

```
template < typename T >
struct Outer
{
    struct Inner { /* 定義 */ } ;
} ;

// 明示的特殊化
template < >
struct Outer<int>::Inner
{
    // 定義
} ;
```

- クラステンプレートのメンバenum

```
template < typename T >
struct X
{
    enum struct E { foo, bar } ;
} ;

// 明示的特殊化
template < >
enum struct X<int>::E
{
    hoge, moke
} ;
```

- クラス、あるいはクラステンプレートのメンバークラステンプレート

```
// クラス
struct Outer_class
{
    template < typename T >
    struct Inner_class_template { } ;
} ;

// 明示的特殊化
template < >
struct Outer_class::Inner_class_template<int>
{
// 定義
} ;

// クラステンプレート
template < typename T >
struct Outer_class_template
{
    template < typename U >
    struct Inner_class_template { } ;
} ;

// 明示的特殊化
template < > // Outer_class_templateの明示的特殊化
template < > // Inner_class_templateの明示的特殊化
struct
Outer_class_template<int>::Inner_class_template<int>
{
// 定義
} ;
```

クラス、クラステンプレートを問わず、メンバークラステンプレートの明示的特殊化ができる。

- クラス、あるいはクラステンプレートのメンバー関数テンプレート

```
// クラス
struct Outer_class
{
    template < typename T >
```

```
    void member_function_template() { }
}

// メンバー関数テンプレートの明示的特殊化
template < >
void Outer_class::member_function_template<int>()
{
// 定義
}

// クラステンプレート
template < typename T >
struct Outer_class_template
{
    template < typename U >
    void member_function_template() { }
};

// メンバー関数テンプレートの明示的特殊化
template < > // Outer_class_templateの明示的特殊化
template < > // member_function_templateの明示的特殊化
void
Outer_class_template<int>::member_function_template<int>
()
{
// 定義
}
```

テンプレートの明示的特殊化は、修飾名の場合、テンプレートの宣言されている名前空間の外側で宣言することもできる。

```
namespace ns {

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

// ns::fの明示的特殊化
template < >
void ns::f( int ) { }
```

関数テンプレートとクラステンプレートの場合、明示的特殊化の元となるテンプレートの宣言は、明示的特殊化の宣言より先行していなければならない。

```
// エラー、テンプレートの宣言が先行していない
template < >
void f( int ) ;

// テンプレートの宣言
template < typename T >
void f( T ) ;

// OK、テンプレートの宣言が先行している
template < >
void f( short ) ;
```

メンバーテンプレートに対する明示的特殊化の定義には、メンバーの属するクラスもしくはクラステンプレートの定義が先行していなければならない。

```
// クラスの宣言
struct Outer ;

// エラー、クラスの定義が先行していない
template < >
struct Outer::Inner<int> { } ;

// クラスの定義
struct Outer
{
    // メンバーテンプレート
    template < typename T >
    struct Inner { } ;
} ;

// OK、クラスの定義が先行している
template < >
struct Outer::Inner<short> { } ;
```

メンバー関数、メンバー関数テンプレート、メンバークラス、メンバーenum、メンバークラステンプレート、クラステンプレートのstaticデータメンバーは、暗黙に実体化されるクラスの特殊化に対しても、明示的に特殊化できる。

```

template < typename T >
struct X
{
    void f() { }
    void g() { }
} ;

// X<int>::fの明示的特殊化
template < >
void X<int>::f() { }

int main()
{
    X<int> x ; // X<int>を暗黙的に実体化
    x.f() ; // 明示的特殊化を使う
    x.g() ; // 暗黙に実体化された特殊化を使う
}

```

このように、一部のメンバーだけを明示的に特殊化できる。明示的に特殊化されなかつたメンバーが使われた場合は、クラステンプレートから暗黙の実体化による特殊化が使われる。

メンバーの明示的特殊化より、クラステンプレートの定義が先行していなければならない。

```

// エラー
template < >
void X<int>::f() { }

template < typename T >
struct X
{
    void f() { }
} ;

```

暗黙に宣言される特別なメンバー関数を明示的特殊化することはできない。

```

template < typename T >
struct X
{
    // デフォルトコンストラクターは暗黙に宣言される
}

```

```

} ;

// エラー
// 暗黙に宣言される特別なメンバー関数の明示的特殊化
template < >
X<int>::X() { }

template < typename T >
struct Y
{
    Y() { } // 特別なメンバー関数の明示的な宣言
} ;

// OK
// 暗黙に宣言されていない特別なメンバー関数の明示的特殊化
template < >
Y<int>::Y() { }

```

このように、特別なメンバー関数を明示的特殊化する場合には、クラステンプレートの定義内で、明示的に宣言する必要がある。

明示的特殊化されたクラステンプレートのメンバーは、元のクラステンプレートとは独立して存在する。そのため、元のクラステンプレートとは全く違うメンバーの宣言にすることができる。

```

template < typename T >
struct X
{
    void f() ;
} ;

// 明示的特殊化
template < >
struct X<int>
{ // 元のクラステンプレートとは違うメンバー
    void g() ;
} ;

```

明示的特殊化されたクラステンプレートの定義内のメンバー宣言は、通常のクラス定義のように記述する。つまり、`template < >`をつける必要はない。メンバーの定義をクラス定義の外に記述する場合も同じ。

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

// 明示的特殊化されたクラステンプレート
template < >
struct X<int>
{
    void f() ; // 宣言
} ;

// 明示的特殊化されたクラステンプレート定義のメンバーの定義
// template < >は必要ない
void X<int>::f() { }
```

ただし、明示的に特殊化されたメンバークラステンプレートのメンバーを定義するときには、`template < >`が必要である。メンバークラスではなく、メンバークラステンプレートであることに注意。

```
template < typename T >
struct Outer
{
    // メンバークラス
    struct Inner { } ;

    // メンバークラステンプレート
    template < typename U >
    struct Inner_temp { } ;
} ;

// 特殊化Outer<int>のメンバークラスの明示的特殊化
template < >
struct Outer<int>::Inner
{
    void f() ;
} ;

// メンバークラスのメンバーの定義
// template < >は必要ない
void Outer<int>::Inner::f() { }

// 特殊化Outer<int>のメンバークラステンプレートの明示的特殊化
template < >
template < typename U >
```

```

struct Outer<int>::Inner_temp
{
    void f() ;
} ;

// メンバークラステンプレートのメンバーの定義
// template < >が必要
template < >
template < typename U >
void Outer<int>::Inner_temp<U>::f() { }

```

テンプレート、メンバーテンプレート、クラステンプレートのメンバーが明示的に特殊化されている場合、暗黙の実体化が起こる前に、明示的特殊化が宣言されていなければならない。

```

template < typename T >
struct X
{ } ;

// テンプレートの特殊化X<int>の使用
// X<int>に対する暗黙の実体化が起こる。
X<int> i ;

// エラー、明示的特殊化の宣言より前に、特殊化の暗黙の実体化が起こっている。
template < >
struct X<int> { } ;

```

テンプレートの明示的特殊化の名前空間スコープは、テンプレートの名前空間スコープと同じ。

宣言されているが定義されていない明示的特殊化を指すテンプレート名は、不完全定義されたクラスと同様に使うことができる。

```

template < typename T >
struct X { } ;

// 明示的特殊化の宣言
// X<int>はまだ定義されていない
template < >
struct X<int> ;

```

```
X<int> * p ; // OK、不完全型へのポインター
X<int> obj ; // エラー、不完全型のオブジェクト
```

関数テンプレートの明示的特殊化の際のテンプレート名のテンプレート引数は、テンプレート実引数の型が、関数の実引数の型から推定できる場合は、省略することができる。

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

template < typename T >
void f( X<T> ) ;

// 関数テンプレートf<int>の明示的特殊化
// テンプレートの特殊化の型は実引数の型から推定可能
template < >
void f( X<int> ) { }
```

ある関数テンプレートと同じ名前で、関数テンプレートの特殊化と同じ型の関数であっても、その関数は、関数テンプレートの明示的特殊化ではない。

```
// 関数テンプレートf
template < typename T >
void f( T ) { }

// 関数テンプレートfの明示的特殊化f<int>
template < >
void f( int ) { }

// 関数f
// 関数テンプレートfの明示的特殊化ではない
void f( int ) { }
```

関数テンプレートの明示的特殊化は、宣言にinline指定子があるか、deleted定義されている場合のみ、inlineとなる。元の関数テンプレートのinline指定子の有無には影響されない。

```
// inline指定子のある関数テンプレート
template < typename T >
inline void f( T ) { }
```

```
// 非inline関数
// 元のテンプレートのinline指定子には影響されない
template < >
void f( int ) { }

// inline関数
template < >
inline void f( short ) { }
```

テンプレートのstaticデータメンバーの明示的特殊化の宣言は、初期化子を含む場合、定義となる。初期化子を含まない場合は宣言となる。デフォルト初期化が必要なstaticデータメンバーを定義する場合は、文法上の制約から、初期化リストを使う必要がある。

```
template < typename T >
struct X
{
    static int data ;
} ;

// 明示的特殊化の宣言、定義ではない
template < >
int X<int>::data ;

// エラー、メンバー関数int()の宣言
// 文法上の制約による
template < >
int X<int>::data () ;

// 明示的特殊化の定義
template < >
int X<int>::data { } ;
```

クラステンプレートのメンバーとメンバーテンプレートは、クラステンプレートで定義されていて、クラステンプレートが暗黙に実体化されていても、明示的特殊化できる。

```
template < typename T >
struct X
{
    void f() { }
    void g() { }
} ;
```

```
// 明示的特殊化
template < >
void X<int>::f() { }

int main()
{
    X<int> x ; // X<int>の暗黙の実体化
    x.f() ; // 明示的特殊化が使われる
    x.g() ; // 暗黙の実体化により生成された特殊化が使われる
}
```

これにより、メンバーやメンバーテンプレートの一部だけを明示的に特殊化することができる。

ネストしたクラステンプレートのメンバーやメンバーテンプレートを明示的特殊化する場合、ネストした数だけtemplate<>を記述する必要がある。

```
template < typename T1 >
struct Outer
{
    template < typename T2 >
    struct Inner
    {
        template < typename T3 >
        void f() { }
    } ;
} ;

template < > // Outer<int>の明示的特殊化
template < > // Outer<int>::Inner<int>の明示的特殊化
template < > // Outer<int>::Inner<int>::f<int>の明示的特殊化
void Outer<int>::Inner<int>::f<int>() { }
```

14.7 関数テンプレートの特殊化(Function template specializations)

関数テンプレートが実体化したものを、関数テンプレートの特殊化(function template specialization)という。

テンプレート実引数は、明示的に指定することもできるし、文脈から推定させることもできる。

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

int main()
{
    f( 0 ) ; // f<int>
    f<double>( 0.0 ) ; // f<double>
}
```

14.7.1 明示的なテンプレート実引数指定(explicit template argument specification)

関数テンプレートには、テンプレート実引数を明示的に指定することができる。その文法は、他のテンプレートと同じで、テンプレート名の後に< テンプレート実引数リスト >を指定する。

```
template < typename T1, typename T2, typename T3 >
void f() { }

int main()
{
    f< int, int, int >() ;
    f< short, int, long>() ;
}
```

関数テンプレートのテンプレート実引数リストは、以下の場合に明示的に指定することができます。

- 関数が呼ばれた場合
- 関数のアドレスを取得する場合、関数へのリファレンスを初期化する場合、メンバ一関数へのポインターを取得する場合

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

// 関数のアドレスを取得する場合
auto fp = &f<int> :
```

```
// 関数へのリファレンスを初期化する場合
auto & fr = f<int> ;

struct S
{
    template < typename T >
    void f() { }
};

// メンバー関数へのポインターを取得する場合
auto mfp = &S::f<int> ;
```

- 明示的特殊化
- 明示的実体化
- friend宣言

明示的にテンプレート実引数が指定されていない後続のテンプレート実引数が、推定できたり、デフォルトのテンプレート実引数から得られる場合は、省略できる。

```
template < typename T1, typename T2, typename T3 = int >
void f( T2 ) { }

int main()
{
    // 明示的な指定
    f< int, int, int >( 0 ) ;
    // OK、T3はデフォルトのテンプレート実引数から得る
    f< int, int >( 0 ) ;
    // OK、T2は実引数推定される
    f< int >( 0 ) ;
}
```

すべてのテンプレート実引数が、推定できるか、デフォルトのテンプレート実引数を与えられている場合、明示的なテンプレート実引数指定は空でもよい。

```
template < typename T1, typename T2 = int >
void f( T1 ) { }

int main()
{
```

```
f<>( 0 ) ; // OK
}
```

空のテンプレート実引数指定は、省略することもできる。

```
template < typename T1, typename T2 = int >
void f( T1 ) { }

int main()
{
    f( 0 ) ; // OK
}
```

空のテンプレート実引数指定は、関数テンプレートの特殊化を明示的に呼び出すのに使うことができる。

```
void f( int ) { } // #1
template < typename T > void f( T ) { } // #2

int main()
{
    f( 0 ) ; // #1
    f<>( 0 ) ; // #2
}
```

14.7.2 テンプレートの実引数推定(Template argument deduction)

関数テンプレートの特殊化が参照される際には、テンプレート仮引数にはすべて、対応するテンプレート実引数がなければならぬ。テンプレート実引数は、明示的に指定することもできるが、関数テンプレートの場合は、文脈から推定させることができる。これを、テンプレートの実引数推定(argument deduction)という。

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

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

通常の文字列リテラルの場合の型は、関数の仮引数に渡す際の変換の都合上、`char const *`となる。

実引数の推定方法は、できるだけ普通のプログラマーの常識に合わせるために、とても複雑になっている。

テンプレートの実引数推定は、関数呼び出し以外の文脈でも行われる。実引数推定が行われる文脈は以下の通り。

- 関数呼び出しの文脈

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

int main()
{
    f( 0 ) ; // 実引数推定、f<int>
    f( 0.0 ) ; // 実引数推定、f<double>
}
```

- 関数のアドレスを得る文脈

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

int main()
{
    // 実引数推定、f<int>
    void (*ptr)(int) = &f ;
    // 実引数推定、f<double>
    void (*ptr)(double) = &f ; //
}
```

- 変換関数の文脈

```

struct S
{
    template < typename T >
    operator T() { return T(); }
};

int main()
{
    S s ;
    // 実引数推定、S::operator int
    int a = s ;
    // 実引数推定、S::operator double
    double b = s ;
}

```

実は、変換関数テンプレートには、明示的にテンプレート実引数を指定する方法がないので、実引数推定が唯一の特殊化を参照する方法である。

14.7.3 半順序(partial ordering)

TODO: より詳細な解説には、時間が必要。

複数のテンプレートの特殊化のうち、どの特殊化が、テンプレート実引数に対して、最も特殊であるかを決定することを、半順序(partial ordering)という。

```

// #1
template < typename T >
struct S { } ;

// #2
template < typename T >
struct S< T * > { } ;

// #3
template < typename T >
struct < T const * > { } ;

```

```
int main()
{
    S<int> s1 ; // #1
    S<int *> s2 ; // #2
    S<int const *> s3 ; // #3
}
```

半順序のルールは複雑だが、簡単に説明すると、CV修飾子、ポインター型、リファレンス型、配列型、関数型、テンプレートなどの、様々な型のパターンを再帰的に一致させていき、もっとも特殊なものを決定する。

この詳細は、通常のプログラマーの常識に合うように、複雑に定義されている。

15 例外(Exception handling)

例外(Exception)は、実行を例外ハンドラーに移す機能である。例外はスレッドごとに存在する。実行を例外ハンドラーに移す際に、オブジェクトを渡すことができる。例外ハンドラーに実行を移すには、tryブロックの中か、tryブロックの中で呼ばれている関数の中でthrow式を使う。

tryブロック:

try 複合文 ハンドラーseq

関数tryブロック:

try コンストラクター初期化子_{opt} 複合文 ハンドラーseq

ハンドラーseq:

ハンドラー ハンドラーseq

ハンドラー:

catch (例外宣言) 複合文

throw式:

throw 代入式_{opt}

tryブロック文の文法は、キーワードtryに続いて複合文を書き、ひとつ以上のハンドラーを記述する。throw式の型はvoidである。throw式を実行するコードのことを、「例外を投げる(throw an exception)」コードといい、処理がハンドラーに移る。

```
int main()
```

```
{  
    try  
    {  
        throw 0 ; // int型のオブジェクトをthrowする  
    }  
    catch ( int i )  
    {  
        // int型のオブジェクトがthrowされた時に実行される  
    }  
    catch ( double d )  
    {  
        // double型のオブジェクトがthrowされた時に実行される  
    }  
    catch ( ... )  
    {  
        // 任意の型のオブジェクトがthrowされた時に実行される  
    }  
}
```

goto文やswitch文を使い、tryブロックやハンドラーの外側から内側に処理を移してはならない。tryブロック内やハンドラー内の移動はできる。

```
int main()  
{  
    // エラー、tryブロックの外側から内側に処理を移す  
    goto begin_try_block ;  
    // エラー、ハンドラーの外側から内側に処理を移す  
    goto begin_handler ;  
    try  
    {  
        begin_try_block: ;  
  
        // OK、tryブロック内の移動  
        goto end_try_block ;  
  
        end_try_block: ;  
    }  
    catch ( ... )  
    {  
        begin_handler: ;  
  
        // OK、ハンドラー内の移動  
        goto end_handler ;  
    }  
}
```

```
        end_handler: ;
    }
}
```

```
void f( int i )
{
    switch( i )
    {
        // OK
        case 0 : ;

        try
        {
            // エラー
            case 1 : ;
        }
        catch ( ... )
        {
            // エラー
            case 2 : ;
        }

        // OK
        case 4 : ;
    }
}
```

goto文、break文、return文、continue文を使って、tryブロックとハンドラーの内側から外側に抜けることができる。

```
void f()
{
    try
    {
        goto end_f ; // OK
    }
    catch ( ... )
    {
        return ; // OK
    }
}
```

```
end_f : ;
}
```

関数tryブロック(function-try-block)は、関数の本体に記述できる。

```
void f()
try
{
}

catch ( ... )
{
}

}
```

コンストラクターの関数の本体として記述する場合には、tryと複合文の間にコンストラクター初期化子を記述する。

```
struct X
{
    int m1 ;
    int m2 ;

    X()
    try
    : m1(0), m2(0) // コンストラクター初期化子
    { }
    catch ( ... ) { }
};
```

関数tryブロックがコンストラクターかデストラクターの本体に用いられた場合、複合文と、構築と破棄の際にクラスのサブオブジェクトが例外を投げた場合、ハンドラーに処理が移る。

コンストラクターに関数tryブロックを使う例

```
// 構築時の例外を投げるクラス
struct throw_at_construction
{
```

```
throw_at_construction()
{
    throw 0 ;
}
} ;

struct X
{
    // 構築時に例外を投げるデータメンバー
    throw_at_construction member ;

    X()
    try : member()
    { }
    catch ( ... ) { } // ハンドラーに処理が渡る
} ;

// 構築時に例外を投げる基本クラス
struct Y : throw_at_construction
{
    Y()
    try { }
    catch ( ... ) { } // ハンドラーに処理が渡る
} ;
```

デストラクターに関数tryブロックを使う例

```
// 破棄時に例外を投げるクラス
struct throw_at_destruction
{
    ~throw_at_destruction()
    {
        throw 0 ;
    }
} ;

struct X
{
    throw_at_destruction member ;

    ~X()
    try { }
    catch ( ... ) { }
```

```
} ;  
  
struct Y : throw_at_destruction  
{  
    ~Y()  
    try { }  
    catch ( ... ) { }  
};
```

15.1 例外を投げる(Throwing an exception)

例外を投げる(throwing an exception)とは、日本語では他にも、送出するとかスローするなどとも書かれている。

例外を投げると、処理はハンドラーに移る。例外を投げるときには、オブジェクトが渡される。オブジェクトの型によって、処理が渡されるハンドラーが決まる。

```
// int型  
throw 0 ;  
  
// const char *型  
throw "hello" ;  
  
struct X { } ;  
X x ;  
// X型  
throw x ;
```

例外が投げられると、型が一致する最も近い場所にあるハンドラーに処理が移る。「最も近い」というのは、最近に入って、まだ抜けていないtryブロックに対応するハンドラーである。

```
// 例外を投げる  
void f() { throw 0 ; }  
  
int main()  
{  
    try  
    {  
        try  
        {
```

```

        try { f() } // 関数fの中で例外を投げる
        catch ( ... ) { } // ここに処理が移る
    }
    catch ( ... ) { }
}
catch ( ... ) { }
}

```

throw式はオペランドから一時オブジェクトを初期化する。この一時オブジェクトを例外オブジェクト(exception object)という。例外オブジェクトの型を決定するには、throw式のオペランドの型からトップレベルのCV修飾子を取り除き、T型への配列型はTへのポインター型へ、T型を返す関数型は、T型を返す関数へのポインター型に変換する。

```

throw 0 ; // int

int const a = 0 ;
throw a ; // int

int const volatile * const volatile p = &a ;
throw p ; // int const volatile *

int b[5] ;
throw b ; // int *

int f( int ) ;
throw f ; // int (*)(int)

```

この一時オブジェクトはlvalueであり、型が適合するハンドラーの変数の初期化に使われる。

```

void f()
{
    try
    {
        throw 0 ; // 例外オブジェクトはint型のlvalue
    }
    catch ( int exception_object ) // 例外オブジェクトで初期化
    {
    }
}

```

例外オブジェクトの型が不完全型か不完全型へのポインター型である場合は、エラーとなる。

```
struct incomplete_type ;  
  
void f()  
{  
    // エラー、不完全型へのポインター型  
    throw static_cast<incomplete_type *>(nullptr) ;  
}
```

ただし、void型はその限りではない。

```
void f()  
{  
    // OK、void *  
    throw static_cast<void *>(nullptr) ;  
}
```

いくつかの制限を除けば、throw式のオペランドは、関数への実引数やreturn文のオペランドとほぼ同じ扱いになっている。

例外オブジェクトのメモリーは、未規定の方法で確保される。

例外オブジェクトの寿命の決定にはふたつの条件があり、どちらか遅い方に合わせて破棄される。

ひとつは例外を再び投げる以外の方法で、例外を捉えたハンドラーから抜け出すこと。

```
void f()  
{  
  
    try  
    {  
        throw 0 ;  
    }  
    catch ( ... )  
    {  
        // return文やgoto文などでハンドラーの複合文の外側に移動するか  
        // あるいはハンドラーの複合文を最後まで処理が到達すれば、例外  
        // オブジェクトは破棄される  
    }  
}
```

}

例外が再び投げられた場合は、例外オブジェクトの寿命は延長される。

```
void f() ; // 例外を投げるかもしれない関数

void g() {

    try { f() ; }
    catch ( ... )
    {
        throw ; // 例外を再び投げる
    }
}
```

この場合、例外オブジェクトは破棄されずに、例外処理が続行する。

もうひとつの条件は、例外オブジェクトを参照する最後のstd::exception_ptrが破棄された場合。これはライブラリの話になるので、本書ではstd::exception_ptrについては解説しない。

例外オブジェクトのストレージが解放される方法は未規定である。

例外オブジェクトの型がクラスである場合、クラスのコピーコンストラクターかムーブコンストラクターのどちらか片方と、デストラクターにアクセス可能でなければならない。

以下のようなクラスは、例外オブジェクトとして投げることができる。

```
// 例外オブジェクトとして投げられるクラス
// コピーコンストラクター、ムーブコンストラクター、デストラクター
// にアクセス可能
struct throwable1
{
    throwable1( throwable1 const & ) { }
    throwable1( throwable1 && ) { }
    ~throwable1() { }
} ;

// 例外オブジェクトとして投げられるクラス
// コピーコンストラクター、デストラクターにアクセス可能

struct throwable2
```

```

{
    throwable2( throwable2 const & ) { }
    throwable2( throwable2 && ) = delete ;
    ~throwable2() { }
} ;

// 例外オブジェクトとして投げられるクラス
// ムーブコンストラクター、デストラクターにアクセス可能
struct throwable3
{
    throwable3( throwable3 const & ) = delete ;
    throwable3( throwable3 && ) { }
    ~throwable3() { }
} ;

```

例外オブジェクトとして投げられるクラスの条件を満たすには、コピーコンストラクターとムーブコンストラクターは、どちらか片方だけアクセスできればよい。デストラクターには必ずアクセス可能でなければならない。

以下のようなクラスは投げることができない。

```

// 例外オブジェクトとして投げられないクラス
struct unthrowable
{
    // コピーコンストラクター、ムーブコンストラクター両方にアクセスできない
    unthrowable( unthrowable const & ) = delete ;
    unthrowable( unthrowable && ) = delete ;

    // デストラクターにアクセスできない
    ~unthrowable() = delete ;
} ;

```

たとえ、コピーやムーブが省略可能な文脈でも、コピーコンストラクターかムーブコンストラクターのどちらか片方にはアクセス可能という条件を満たしていなければ、クラスは例外オブジェクトとして投げることができない。

例外は、あるハンドラーに処理が移った段階で、とらえられた(キャッチされた)とみなされる。ただし、例外がとらえられたハンドラーから再び投げられた場合は、再びとらえられていない状態に戻る。

```

try
{

```

```

    throw 0 ;
}
catch ( ... )
{
    // 例外はとらえられた

    throw ; // 再びとらえられていない状態に戻る
}

```

例外オブジェクトとして投げられる初期化式の評価が完了した後から、例外がとらえられるまでの間に、別の例外が投げられた場合は、`std::terminate`が呼ばれる。

これが起こるよくある状況は、スタックアンワインディングの最中にデストラクターから例外が投げられることだ。

```

// デストラクターが例外を投げるクラス
struct C
{
    // デストラクターに明示的な例外指定がない場合、この文脈では暗黙にthrow()になるため
    // デストラクターの外に例外を投げるには例外指定が必要
    ~C() noexcept( false ) { throw 0 ; }

};

int main()
{
    try
    {
        C c ;
        throw 0 ;
        // C型のオブジェクトcが破棄される
        // 例外中に例外が投げられたため、std::terminateが呼ばれる
    }
    catch ( ... ) { }
}

```

一般的に、デストラクターから例外を投げるべきではない。

初期化式の評価が完了した後という点に注意。`throw`式のオペランドの初期化式の評価中の例外はこの条件に当てはまらない。

```

struct X
{
    X() { throw 0 ; }
} ;

int main( )
{
    try
    {
        // OK、初期化式の評価中の例外
        // 例外オブジェクトの型はint
        throw X() ;
    }
    catch ( X & exception ) { }
    catch ( int exception ) { } // このハンドラーでとらえられる
}

```

この例ではX型のオブジェクトを例外としてthrowする前に、初期化中にint型の例外が投げられたので、結果として投げられる例外オブジェクトの型はint型になる。

ただし、初期化式の評価が完了した後という点に注意。初期化完了の後に例外が投げられた場合は、`std::terminate`が呼ばれる。

// この例が`std::terminate`を呼ぶかどうかは、C++の実装次第である。

```

struct X
{
    X( X const & ) { throw 0 ; }
} ;

int main( )
{
    try
    {
        // 実装がコピーを省略しない場合、std::terminateが呼ばれる
        // コピーコンストラクターの実行は評価完了後
        throw X() ;
    }
    catch ( ... ) { }
}

```

この文脈では、賢いC++の実装ならば、コピーを省略できる。ただし、コピーが省略され

る保証はない。もし、例外オブジェクトを構築する際にコピーが行われたならば、それはthrow式のオペランドの初期化式の評価完了後なので、この条件に当てはまり、std::terminateが呼ばれる。

また、現行の規格の文面には誤りがあり、以下のコードではstd::terminateが呼ばれるよう解釈できてしまう。

```
// 例外によって抜け出す関数
void f() { throw 0; }

struct C
{
    ~C()
    {
        // 例外によって抜け出す関数を呼ぶ
        try { f(); }
        catch ( ... ) { }
    }
};

int main()
{
    try
    {
        C c;
        throw 0;
        // 例外がハンドラーにとらえられる前に、cのデストラクターが
        呼ばれる
    }
    catch ( ... ) { }
}
```

これは規格の誤りであり、本書執筆の時点で、修正が検討されている。

オペランドのないthrow式は、現在とらえられている例外を再び投げる(rethrow)。これは、再送出とかリスローなどとも呼ばれている。例外が再び有効になり、例外オブジェクトは破棄されずに再利用される。つまり、例外をふたたび投げる際に一時オブジェクトを新たに作ることはない。例外は再びとらえられているものとはみなされなくなり、std::uncaught_exception()の値も、またtrueになる。

```
int main()
{
    try
```

```
{  
    try  
    {  
        throw 0 ;  
    }  
    catch ( int e )  
    { // 例外をとらえる  
        throw ; // 一度捉えた例外を再び投げる  
    }  
}  
catch ( int e )  
{  
    // 再び投げられた例外をとらえる  
}  
}
```

例外がとらえられていない状態でオペランドのないthrow式を実行すると、std::terminateが呼ばれる。

```
int main()  
{  
    throw ; // std::terminateが呼ばれる  
}
```

15.2 コンストラクターとデストラクター(Constructors and destructors)

処理がthrow式からハンドラーに移るにあたって、tryブロックの中で構築された自動オブジェクトのデストラクターが呼び出される。自動オブジェクトの破棄は構築の逆順に行われる。

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

```

int main()
{
    try
    {
        X a ;
        X b ;
        X c ;
        // a, b, cの順に構築される

        throw 0 ;
    }
    // このハンドラーに処理が移る過程で、
    // c, b, aの順に破棄される
    catch ( int ) { }
}

```

オブジェクトの構築、破棄が、例外により中断された場合、完全に構築されたサブオブジェクトに対してデストラクターが実行される。オブジェクトが構築されたストレージの種類は問わない。

```

struct Base
{
    Base() { }
    ~Base() { }
} ;

// コンストラクターに実引数trueが渡された場合、例外を投げるクラス
struct Member
{
    Member( bool b )
    {
        if ( b )
            throw 0 ;
    }
    ~Member() { }
} ;

// Xのサブオブジェクトは、基本クラスBaseと、非staticデータメンバ
// a, b, c
struct X : Base
{
    Member a, b, c ;
}

```

```

X() : a(false), b(true), c(false)
{ }
// Base, aのデストラクターが実行される。
~X() { }

} ;

int main()
{
    try
    {
        X x ;
    }
    catch ( int ) { }
}

```

この例では、クラスXは、サブオブジェクトとして、Base型の基本クラスと、Member型の非staticデータメンバー、a, b, cを持つ。その初期化順序は、基本クラスBase, a, b, c, Xである。クラスMemberは、コンストラクターの実引数にtrueが渡された場合、例外を投げる。クラスXのコンストラクターは、bのコンストラクターにtrueを与えていた。その結果、クラスXのオブジェクトの構築は、例外によって中断される。

この時、デストラクターが実行されるのは、基本クラスBaseのオブジェクトと、Member型の非staticデータメンバーaのオブジェクトである。bは、コンストラクターを例外によって抜けだしたため、構築が完了していない。cは、まだコンストラクターが実行されていないため、構築が完了していない。そのため、b, cのオブジェクトに対してデストラクターは実行されない。

ただし、union風クラスのvariantメンバーには、デストラクターは呼び出されない。

```

struct Member
{
    Member() { }
    ~Member() { }
};

struct X
{
    union { Member m ; } ;
    X() { throw 0 ; } // mのデストラクターは実行されない
}

```

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

あるオブジェクトの非デリゲートコンストラクターの実行が完了し、その非デリゲートコンストラクターを呼び出したデリゲートコンストラクターが例外によって抜けだした場合、そのオブジェクトに対してデストラクターが呼ばれる。

```
struct X
{
    // 非デリゲートコンストラクター
    X( bool ) { }

    // デリゲートコンストラクター
    X() : X( true )
    {
        throw 0 ; // Xのデストラクターが呼ばれる
    }

    ~X() { }
};
```

これは、オブジェクトの構築完了は、非デリゲートコンストラクターの実行が完了した時点だからだ。

例外によって構築が中断されたオブジェクトがnew式によって構築された場合、使われた確保関数に対応する解放関数があれば、ストレージを解放するために自動的に呼ばれる。

```
struct X
{
    X() { throw 0 ; }
    ~X() { }

    // 確保関数
    void * operator new( std::size_t size ) noexcept
    {
        return std::malloc( size ) ;
    }

    // 上記確保関数に対応する解放関数
    void operator delete( void * ptr ) noexcept
    {
```

```

        std::free( ptr ) ;
    }
}

int main()
{
    try
    {
        new X ; // 対応する解放関数が呼ばれる
    }
    catch ( int ) { }
}

```

この例では、Xを構築するためにmallocで確保されたストレージは、正しくfreeで解放される。

throw式から処理を移すハンドラーまでのtryブロック内の自動ストレージ上のオブジェクトのデストラクターを自動的に呼ぶこの一連の過程は、スタックアンワインディング(stack unwinding)と呼ばれている。もし、スタックアンワインディング中に呼ばれたデストラクターが例外によって抜けだした場合、std::terminateが呼ばれる。

```

struct X
{
    X() { }
    ~X() noexcept(false)
    {
        throw 0 ;
    }
};

int main()
{
    try
    {
        X x ;
        throw 0 ; // std::terminateが呼ばれる
    }
    catch ( int ) { }
}

```

現行の文面を解釈すると、以下のコードもstd::terminateを呼ぶように解釈できるが、これは誤りであり、将来の規格改定で修正されるはずである。

```

struct Y
{
    Y() { }
    ~Y() noexcept(false) { throw 0 ; }
} ;

struct X
{
    X() { }
    ~X() noexcept(false)
    {
        try
        {
            // スタックアンワインディング中に呼ばれたデストラクターが
例外によって抜け出す
            // 現行の規格の文面解釈ではstd::terminateが呼ばれてしま
う
            Y y ;
        }
        catch ( int ) { }
    }
} ;

int main()
{
    try
    {
        X x ;
        throw 0 ;
    }
    catch ( int ) { }
}

```

一般に、デストラクターを例外によって抜け出すようなコードは書くべきではない。デストラクターはスタックアンワインディングのために呼ばれるかもしれないからだ。スタックアンワインディング中かどうかを調べる、`std::uncaught_exception`のような標準ライブラリもあるにはあるが、スタックアンワインディング中かどうかを調べる必要は、通常はない。

C++11からは、デストラクターはデフォルトで例外指定がつくようになり、ほとんどの場合、`noexcept(true)`と互換性のある例外指定になる変更がなされたのも、通常はデストラクターを例外で抜け出す必要がないし、またそうすべきではないからだ。

15.3 例外の捕捉(Handling an exception)

`throw`式によって投げられた例外は、`try`ブロックのハンドラーによって捕捉される。ハンドラーの文法は以下の通り。

`catch (例外宣言) 複合文`

```
int main()
{
    try
    {
        throw 0 ; // 例外オブジェクトの型はint
    }
    catch ( double d ) { }
    catch ( float f ) { }
    catch ( int i ) { } // このハンドラーに処理が移る
}
```

例外が投げられると、処理は、例外オブジェクトの型と適合(match)する例外宣言を持つハンドラーに移される。

ハンドラーの例外宣言は、不完全型、抽象クラス型、rvalueリファレンス型であってはならない。

```
struct incomplete ; // 不完全型

struct abstract
{
    void f() = 0 ;
} ;

int main()
{
    try { }
    catch ( incomplete x ) { } // エラー、不完全型
    catch ( abstract a ) { } // エラー、抽象クラス型
    catch ( abstract * a ) { } // OK、抽象クラスへのポインター
    catch ( abstract & a ) { } // OK、抽象クラスへのリファレン
    型
    斯型
```

```

    } catch ( int && rref) { } // エラー、rvalueリファレンス型
}

```

また、例外宣言の型は、不完全型へのポインターやリファレンスであってはならない。ただし、void *, const void *, volatile void *, const volatile void *は、不完全型へのポインター型だが、例外的に許可されている。

ハンドラーの例外宣言が「Tへの配列」の場合、「Tへのポインター」型に変換される。「Tを返す関数」型は、「Tを返す関数へのポインター」型に変換される。

```

catch ( int [5] ) // int *と同じ
catch ( int f( void ) ) // int (*f)(void)と同じ

```

あるハンドラーが、例外オブジェクトの型Eと適合する条件は以下の通り。

- ハンドラーの型が cv Tもしくは cv T &で、EとTが同じ型である場合。

cvは任意のCV修飾子(const, volatile)のこと、トップレベルのCV修飾子は無視される。

たとえば、例外オブジェクトの型がintの場合、以下のようなハンドラーが適合する。

```

catch ( int )
catch ( const int )
catch ( volatile int )
catch ( const volatile int )
catch ( int & )
catch ( const int & )
catch ( volatile int & )
catch ( const volatile int & )

```

- ハンドラーの型がcv Tかcv T &で、TはEの曖昧性のないpublicな基本クラスである場合

例えば、以下のような例が適合する。

```

struct Base { } ;
struct Derived : public Base { } ;

int main()
{

```

```

try
{
    Derived d ;
    throw d ; // 例外オブジェクトの型はDerived
}
catch ( Base & ) { } // 適合、BaseはDerivedの曖昧性の
ないpublicな基本クラス
}

```

以下のような例は適合しない。

```

struct Base { } ;
struct Ambiguous { } ;
struct Derived : private Base, public Ambiguous { } ;

struct Sub : public Derived, public Ambiguous { } ;

int main()
{
    try
    {
        Sub sub ;
        throw sub ; // 例外オブジェクトの型はSub
    }
    catch ( Base & ) { } // 適合しない、非public基本クラス
    catch ( Ambiguous & ) { } // 適合しない、曖昧
}

```

- ハンドラーの型がcv1 T* cv2で、Eがポインター型で、以下のいずれかの方法でハンドラーの型に変換可能な場合
 - 標準ポインター型変換で、privateやprotectedなポインターへの変換や、曖昧なクラスへの変換を伴わないもの

```

struct Base { } ;
struct Derived : public Base { } ;

int main()
{
    try
    {
        Derived d ;

```

```

        throw &d ; // 例外オブジェクトの型はDerived
    }
    catch ( Base * ) { } // 適合、BaseはDerivedの曖昧
    性のないpublicな基本クラス
}

```

- 修飾変換

```

int main()
{
    int i ;
    try
    {

        throw &i ;
    }
    catch ( const int * ) { }
}

```

- ハンドラーの型がポインターかメンバーへのポインターで、Eがstd::nullptr_tの場合

```

struct X
{
    int member ;
} ;

int main()
{
    try
    {
        throw nullptr ;
    }
    catch ( void * ) { } // 適合
    catch ( int * ) { } // 適合
    catch ( X * ) { } // 適合
    catch ( int X::* ) { } // 適合
}

```

nullptrの型であるstd::nullptr_t型の例外オブジェクトは、あらゆるポインター型、メンバーへのポインター型に適合する。

`throw`式のオペランドが定数式で0と評価される場合でも、ポインターやメンバーへのポインター型のハンドラーには適合しない。

```
int main()
{
    try
    {
        throw 0 ; // 例外オブジェクトの型はint
    }
    catch ( int * ) { } // 適合しない
}
```

`try`ブロックのハンドラーは、書かれている順番に比較される。

```
int main()
{
    try
    {
        throw 0 ; // 例外オブジェクトの型はint
    }
    catch ( int ) { } // 適合する。処理はこのハンドラーに移る
    catch ( const int ) { }
    catch ( int & ) { }
}
```

この例では、3つのハンドラーはどれも例外オブジェクトの型に適合するが、比較は書かれている順番に行われる。一番最初に適合したハンドラーに処理が移る。関数のオーバーロード解決のような、ハンドラー同士の型の適合の優劣の比較は行われない。

ハンドラーの例外宣言に...が使われた場合、そのハンドラーはどの例外にも適合する。

```
void f()
{
    try { }
    catch ( int ) { }
    catch ( double ) { }
    catch ( ... ) { } // どの例外にも適合する
}
```

...ハンドラーを使う場合は、`try`ブロックのハンドラーの最後に記述しなければならない。

```
void f()
{
    try { }
    catch ( ... ) { }
    catch ( int ) { } // エラー
}
```

tryブロックのハンドラーのうちに、適合するハンドラーが見つからない場合、同じスレッド内で、そのtryブロックのひとつ上のtryブロックが試みられる。

```
void f()
{
    try { throw 0 ; } // 例外オブジェクトの型はint
    catch ( double ) { } // 適合しない
}

void g()
{

    try
    {
        f() ;
    }
    catch ( int ) { } // 適合する
}

int main()
{
    try
    {
        g() ;
    }
    catch ( ... ) { }
}
```

catch句の仮引数の初期化が完了した時点で、ハンドラーはアクティブ(active)になったとみなされる。スタックはこの時点でアンワインドされている。例外を投げた結果、`std::terminate`や`std::unexpected`が呼ばれた場合、暗黙のハンドラーというものがアクティブになったものとみなされる。catch句から抜けだした場合、ハンドラーはアクティブではなくなる。

現在、アクティブなハンドラーが存在する場合、直前に投げられた例外を、現在捕捉さ

れている例外(currently handled exception)と呼ぶ。

適合するハンドラーが見つからない場合、`std::terminate`が呼ばれる。`std::terminate`が呼ばれる際、スタックがアンワインドされるかどうかは実装次第である。

コンストラクターとデストラクターの関数tryブロックのハンドラー内で、非staticデータメンバーかオブジェクトの基本クラスを参照した場合、挙動は未定義である。

```
struct S
{
    int member ;

    S()
    try
    {
        throw 0 ;
    }
    catch ( ... )
    {
        int x = member ; // 挙動は未定義
    }
} ;
```

コンストラクターの関数tryブロックのハンドラーに処理が移る前に、完全に構築された基本クラスと非staticメンバーのオブジェクトは、破棄される。

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

struct Derived : Base
{
    Derived()
    try
    {
        throw 0 ;
    }
    catch ( ... )
    {
        // 基本クラスBaseのオブジェクトはすでに破棄されている
    }
} ;
```

```
// 非staticデータメンバーのオブジェクトについても同様
}
}
```

オブジェクトの非デリゲートコンストラクターの実行が完了したあとに、デリゲートコンストラクターが例外を投げた場合は、オブジェクトのデストラクターが実行されたあとに、関数tryブロックのハンドラーに処理が移る。

```
struct S
{
    // 非デリゲートコンストラクター
    S() { }

    // デリゲートコンストラクター
    S( int ) try
        : S()
    { throw 0; }
    catch ( ... )
    {
        // デストラクターS::~Sはすでに実行されている
    }

    ~S() { }
};

int main()
{
    S s(0);
}
```

非デリゲートコンストラクターの実行完了をもって、オブジェクトは構築されている。デリゲートコンストラクターが例外を投げた場合の関数tryブロックのハンドラーに処理が移る前に、オブジェクトを破棄されなければならない。そのために、ハンドラーに処理が移る前にデストラクターが呼び出されることになる。

デストラクターの関数tryブロックのハンドラーに処理が移る前に、オブジェクトの基本クラスと非variantメンバーは破棄される。

```
struct Base
{
    Base() { }
```

```

    ~Base() { }
} ;

struct Derived : Base
{
    ~Derived() noexcept(false)
    try { throw 0 ; }
    catch ( ... )
    {
        // 基本クラスはすでに破棄されている
        // 非staticデータメンバーについても同様
    }
} ;

```

関数のコンストラクターの仮引数のスコープと寿命は、関数tryブロックのハンドラー内で延長される。

```

void f( int param )
try
{
    throw 0 ;
}
catch ( ... )
{
    int x = param ; // OK、延長される
}

```

静的ストレージ上のオブジェクトのデストラクターから投げられる例外が、main関数の関数tryブロックのハンドラーで捕捉されることはない。threadストレージ上のオブジェクトのデストラクターから投げられる例外が、スレッドの初期関数の関数tryブロックのハンドラーで捕捉されることはない。

コンストラクターの関数tryブロックのハンドラーの中にreturn文がある場合、エラーとなる。

```

struct S
{
    S()
    try { }
    catch ( ... )
    {
        return ; // エラー
    }
}

```

```
    }  
};
```

コンストラクターとデストラクターの関数tryブロックで、処理がハンドラーの終わりに達したときは、現在ハンドルされている例外が、再びthrowされる。

```
struct S  
{  
    S()  
    try  
    {  
        throw 0 ;  
    }  
    catch ( int )  
    {  
        // 例外が再びthrowされる  
    }  
};
```

コンストラクターとデストラクター以外の関数の関数tryブロック、処理がハンドラーの終わりに達したときは、関数からreturnする。このreturnは、オペランドなしのreturn文と同等になる。

```
void f()  
try  
{  
    throw 0 ;  
}  
catch ( int )  
{  
// return ;と同等  
}
```

もしこの場合に、関数が戻り値を返す関数の場合、挙動は未定義である。

```
int f()  
try  
{  
    throw 0 ;  
}
```

```
catch ( ... )
{
// 挙動は未定義
}
```

例外宣言が例外の型と名前を指定する場合、例外の型のオブジェクトがその名前で、例外オブジェクトからコピー初期化される。

```
int main()
{
    try
    {
        throw 123 ; // 例外オブジェクトの型はint、値は123
    }
    catch ( int e )
    {
        // eの型はint、値は123
    }
}
```

例外宣言が、例外の型のみで名前を指定していない場合、例外の型の一時オブジェクトが生成され、例外オブジェクトからコピー初期化される。

```
int main()
{
    try
    {
        throw 123 ;
    }
    catch ( int )
    {
        // int型の一時オブジェクトが生成され、例外オブジェクトからコピー初期化される
        // 名前がないので、参照する方法はない
    }
}
```

例外宣言の名前の指示するオブジェクト、あるいは無名の一時オブジェクトの寿命は、処理がハンドラーから抜けだして、ハンドラー内で初期化された一時オブジェクトが解放された後である。

```

struct S
{
    int * p ;
    S( int * p ) : p(p) { }
    ~S() { *p = 0 ; }
} ;

int main()
{
    try
    {
        throw 123 ;
    }
    catch ( int e )
    {
        S s( &e ) ;

        // sが破棄された後に、eが破棄される
    }
}

```

そのため、上のコードは問題なく動作する。なぜならば、eが破棄されるのはsよりも後だからだ。

ハンドラーの例外宣言が、非constな型のオブジェクトの場合、ハンドラー内でそのオブジェクトに対する変更は、throw式によって生成された一時的な例外オブジェクトには影響しない。

```

int main()
{
    try
    {
        try
        {
            throw 0 ;
        }
        catch ( int e )
        {
            ++e ; // 変更
            throw ; // 例外オブジェクトの再throw
        }
    }
}

```

```

catch ( int e )
{
    // eは0
}
}

```

ハンドラーの例外宣言が、非constな型へのリファレンス型のオブジェクトの場合、ハンドラー内でそのオブジェクトに対する変更は、throw式によって生成された一時的な例外オブジェクトを変更する。この副作用は、ハンドラー内で再throwされたときにも効果を持つ。

```

int main()
{
    try
    {
        try
        {
            throw 0 ;
        }
        catch ( int & e )
        {
            ++e ; // 変更
            throw ; // 例外オブジェクトの再throw
        }
    }
    catch ( int e )
    {
        // eは1
    }
}

```

15.4 例外指定(Exception specifications)

例外指定(Exception specification)とは、関数宣言で、関数が例外を投げるかどうかを指定する機能である。

関数宣言における例外指定の文法は、リファレンス修飾子の後、アトリビュートの前に記述する。

T D(仮引数宣言) cv修飾子 リファレンス修飾子 例外指定 アトリビュ

一ト指定子

例外指定:

```
noexcept( 定数式 )
noexcept
```

```
void f() noexcept ;

struct S
{
    void f() const & noexcept [[ ]] ;
};
```

例外指定は、関数宣言と定義のうち、関数型、関数へのポインター型、関数型へのリファレンス、メンバー関数へのポインター型に適用できる。また、関数へのポインター型が仮引数や戻り値の型に使われる場合も指定できる。

```
void f() noexcept ; // OK
void (*fp)() noexcept = &f ; // OK
void (&fr)() noexcept = f ; // OK

// OK、仮引数として
void g( void (*fp)() noexcept ) ;
// OK、戻り値の型として
auto h() -> void (*)() noexcept ;

struct S
{
    void f() noexcept ; // OK
};
```

typedef宣言とエイリアス宣言には使用できない。

```
typedef void (*func_ptr_type)() noexcept ; // エラー
using type = void (*)() noexcept ; // エラー
```

例外指定のない関数宣言は、例外を許可する関数である。

例外指定にnoexceptが指定された場合、その関数は例外を許可しないと指定したこと

になる。

例外指定に、`noexcept(定数式)`を指定し、定数式が`true`と評価される場合、その関数は例外を許可しないと指定したことになる。定数式が`false`と評価される場合、その関数は例外を許可する関数と指定したことになる。

```
void f1() ; // 例外を許可
void f2() noexcept ; // 例外を許可しない
void f3() noexcept( true ) ; // 例外を許可しない
void f4() noexcept( false ) ; // 例外を許可
```

`noexcept(定数式)`は、コンパイル時の条件に従って、関数の例外指定を変えることによく使える。

```
template < typename T >
constexpr bool is_nothrow()
{
    return std::is_fundamental<T>::value ;
}

// テンプレート仮引数が基本型なら例外を投げない実装ができる関数
template < typename T >
void f( T x ) noexcept( is_nothrow<T>() ) ;
```

この例では、関数`f`は、テンプレート仮引数が基本型の場合、例外を投げない実装ができるものとする。そこで、テンプレートのインスタンス化の際に、型を調べることによって、例外を許可するかどうかをコンパイル時に切り替えることができる。

もし、例外を許可しない関数が、例外の`throw`によって抜け出した場合、`std::terminate`が呼ばれる。

```
// 例外を許可する関数
void allow_exception()
{
    throw 0 ; // OK
}

// 例外を許可しない関数
void disallow_exception() noexcept
{
    try
    {
```

```

        throw 0 ; // OK、例外は関数の外に抜けない
    }
    catch ( int ) { }

    throw 0 ; // 実行時にstd::terminateが呼ばれる
}

```

例外を許可しないというのは、例外によって関数から抜け出すことを禁止するものであり、関数の中で例外を使うことを禁止するものではない。

例外を許可しない関数は、例外を投げる可能性があったとしても、違法ではない。C++実装は、そのようなコードを合法にするように明確に義務付けられている。

```

void f() noexcept
{
    throw 0 ; // OK、コンパイルが通る
    // 実行時にstd::terminateが呼ばれる
}

void g( bool b ) noexcept
{
    if ( b )
        throw 0 ; // OK、コンパイルが通る
    // 実行時にbがtrueの場合、std::terminateが呼ばれる
}

```

もちろん、そのような関数を呼び出して、結果として関数の外に例外が投げられた場合、std::terminateが呼ばれる。

この他に、C++11では非推奨(deprecated)扱いになっている機能に、動的例外指定(dynamic-exception-specification)がある。この機能は将来廃止されるので、詳しく解説しないが、概ね以下のような機能となっている。

```

// 例外を許可しない
void f() throw( ) ;

// int型のthrowを許可する
void g() throw( int ) ;

// int型とshort型のthrowを許可する
void h() throw( int, short ) ;

```

動的例外指定のある関数では、例外を関数の外にthrowすると、`std::unexpected`が呼ばれる。もし、許可した型の例外をthrowした場合は、そのままハンドラーの検索が行われるが、許可しない型をthrowした場合は、`std::terminate`が呼ばれるとされている。

少なくとも、当初のC++の設計はそうであったが、現実には、そのように実装するC++実装は出てこなかった。ほとんどの実装では、動的例外指定は、単に無視された。

その後、何も例外として許可する型を指定しない、`throw()`だけが、本来の設計とは違う意味で使われだした。関数が外に例外を投げない保証を記述するために使われだしたのだ。この、例外を関数の外に投げない保証というのは、とても便利だったので、C++11では専用の文法を与えられ、無例外指定として追加された。そして、動的例外指定は、現実に実装されていないことから、非推奨に変更された。将来的には取り除かれる予定だ。

クラスの暗黙に宣言される特別なメンバー関数は、この動的例外指定を暗黙に指定される。その型リストは、暗黙の実装が呼び出す関数が投げる可能性のある例外のみを持つ。

これは、基本クラスや非staticメンバーが、明示的に例外を許可するものでないかぎり、クラスの暗黙の特別なメンバーは、無例外指定されるということである。

```
class S
{
// 暗黙のコンストラクター、デストラクター、代入演算子は、
// 例外指定throw()が指定される
};
```

解放関数の宣言に、明示的な例外指定がない場合は、`noexcept(true)`が指定されたものとみなされる。

```
// 暗黙にnoexcept(true)が指定される
void operator delete( void * ) ;
```

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
<http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover

Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not

use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of

the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.