

C++0x コンセプト

高橋晶(アキラ)

ブログ:「Faith and Brave – C++で遊ぼう」
http://d.hatena.ne.jp/faith_and_brave/


Agenda

一部:基礎

- 概要
- 動機
- コンセプトの定義
- テンプレートパラメータの制約
- コンセプトマップ

二部:発展

- 関連型と関連要件
- 暗黙のコンセプト定義と、明示的なコンセプト定義
- 要件指定のシンタックスシュガー
- Axiom
- コンセプトの階層化
- コンセプトに基づいたオーバーロード
- 範囲for文
- 次期標準ライブラリのコンセプト



一部：基礎

背景

主要な機能



概要

- コンセプトは、テンプレートをさらに強力に、さらに使いやすくするための型システム
- テンプレートパラメータに対する制約
- テンプレートライブラリを作るのがラクになる
(というよりは、より明示的になる、かな)

動機1 - エラーメッセージの問題

テンプレートインスタンス化時の
エラーメッセージを改善する

```
list<int> ls;  
sort(ls.begin(), ls.end()); // エラー!
```

↓VC9が出力するエラーメッセージ



main.cpp

error C2784: 'reverse_iterator<_Ranlt>::difference_type std::operator -(const std::reverse_iterator<_Ranlt> &,const std::reverse_iterator<_Ranlt2> &)' : テンプレート 引数を 'const std::reverse_iterator<_Ranlt> &' に対して 'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした

```
with  
[  
  _Ty=int,  
  _Secure_validation=true  
]
```

C:\Program Files\Microsoft Visual Studio 9.0\VC\include\xutility(2238) : 'std::operator -' の宣言を確認してください。

main.cpp(9) : コンパイルされたクラスの テンプレートのインスタンス化 'void std::sort<std::list<_Ty>::_Iterator<_Secure_validation>>(_Ranlt,_Ranlt)' の参照を確認してください

```
with  
[  
  _Ty=int,  
  _Secure_validation=true,  
  _Ranlt=std::list<int>::_Iterator<true>  
]
```

error C2784: 'reverse_iterator<_Ranlt>::difference_type std::operator -(const std::reverse_iterator<_Ranlt> &,const std::reverse_iterator<_Ranlt2> &)' : テンプレート 引数を 'const std::reverse_iterator<_Ranlt> &' に対して 'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした

```
with  
[  
  _Ty=int,  
  _Secure_validation=true  
]
```

'std::operator -' の宣言を確認してください。

error C2784: 'reverse_iterator<_Ranlt>::difference_type std::operator -(const std::reverse_iterator<_Ranlt> &,const std::reverse_iterator<_Ranlt2> &)' : テンプレート 引数を 'const std::reverse_iterator<_Ranlt> &' に対して 'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした

```
with  
[  
  _Ty=int,  
  _Secure_validation=true  
]
```

'std::operator -' の宣言を確認してください。

error C2784: 'reverse_iterator<_Ranlt>::difference_type std::operator -(const std::reverse_iterator<_Ranlt> &,const std::reverse_iterator<_Ranlt2> &)' : テンプレート 引数を 'const std::reverse_iterator<_Ranlt> &' に対して 'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした

```
with  
[  
  _Ty=int,  
  _Secure_validation=true  
]
```

'std::operator -' の宣言を確認してください。

error C2784: 'reverse_iterator<_Ranlt>::difference_type std::operator -(const std::reverse_iterator<_Ranlt> &,const std::reverse_iterator<_Ranlt2> &)' : テンプレート 引数を 'const std::reverse_iterator<_Ranlt> &' に対して 'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした

```
with  
[  
  _Ty=int,  
  _Secure_validation=true  
]
```

'std::operator -' の宣言を確認してください。

error C2784: 'reverse_iterator<_Ranlt>::difference_type std::operator -(const std::reverse_iterator<_Ranlt> &,const std::reverse_iterator<_Ranlt2> &)' : テンプレート 引数を 'const std::reverse_iterator<_Ranlt> &' に対して 'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした

```
with  
[  
  _Ty=int,  
  _Secure_validation=true  
]
```

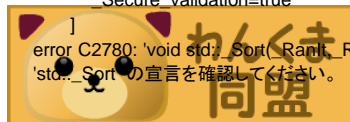
'std::operator -' の宣言を確認してください。

を確認してください。



わんくま同盟 東京勉強会 #33

```
error C2784: '_Base1::difference_type std::operator -(const std::_Revrantlt<_Ranlt,_Base> &,const std::_Revrantlt<_Ranlt2,_Base2> &)' : テンプレート 引数を 'const std::_Revrantlt<_Ranlt,_Base> &' に対して
'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした
with
[
  _Ty=int,
  _Secure_validation=true
]
'std::operator -' の宣言を確認してください。
error C2784: '_Base1::difference_type std::operator -(const std::_Revrantlt<_Ranlt,_Base> &,const std::_Revrantlt<_Ranlt2,_Base2> &)' : テンプレート 引数を 'const std::_Revrantlt<_Ranlt,_Base> &' に対して
'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした
with
[
  _Ty=int,
  _Secure_validation=true
]
'std::operator -' の宣言を確認してください。
error C2784: '_Base1::difference_type std::operator -(const std::_Revrantlt<_Ranlt,_Base> &,const std::_Revrantlt<_Ranlt2,_Base2> &)' : テンプレート 引数を 'const std::_Revrantlt<_Ranlt,_Base> &' に対して
'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした
with
[
  _Ty=int,
  _Secure_validation=true
]
'std::operator -' の宣言を確認してください。
error C2784: '_Base1::difference_type std::operator -(const std::_Revrantlt<_Ranlt,_Base> &,const std::_Revrantlt<_Ranlt2,_Base2> &)' : テンプレート 引数を 'const std::_Revrantlt<_Ranlt,_Base> &' に対して
'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした
with
[
  _Ty=int,
  _Secure_validation=true
]
'std::operator -' の宣言を確認してください。
error C2784: '_Base1::difference_type std::operator -(const std::_Revrantlt<_Ranlt,_Base> &,const std::_Revrantlt<_Ranlt2,_Base2> &)' : テンプレート 引数を 'const std::_Revrantlt<_Ranlt,_Base> &' に対して
'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした
with
[
  _Ty=int,
  _Secure_validation=true
]
'std::operator -' の宣言を確認してください。
error C2676: 二項演算子 '!' : 'std::list<_Ty>::_Iterator<_Secure_validation>' は、この演算子または定義済の演算子に適切な型への変換の定義を行いません。(新しい動作; ヘルプを参照)
with
[
  _Ty=int,
  _Secure_validation=true
]
error C2780: 'void std::Sort<_Ranlt,_Diff_Pr>' : 4 引数が必要で 3 が設定されます。
'std::Sort' の宣言を確認してください。
```



動機1 - エラーメッセージの問題

本当は以下のようなエラーメッセージを
出力してほしい

エラー！ list<int>::iteratorはoperator<を持ってないので
ランダムアクセスイテレータの要件を満たしません。

動機2 – テンプレートをより強かに

- C++03までは、知る人ぞ知る
テンプレートの手法が数多くあった
(SFINAE, Traits, タグディスパッチ, etc...)
- コンセプトは、これらの手法を言語で総括的にサポートするという目的がある

コンセプトの定義

テンプレートパラメータの型に対する 制約を定義する

```
auto concept LessThanComparable<class T> {  
    bool operator<(const T&, const T&);  
}
```

LessThanComparableは

「Tはoperator<によって比較できなければならない」
という制約

テンプレートパラメータの制約

テンプレート定義の際にコンセプトを指定することによって、テンプレートパラメータを制約することができる

```
template <LessThanComparable T>
const T& min(const T& a, const T& b)
{
    return a < b ? a : b;
}
```

typename T / class Tの代わりに
LessThanComparable Tと書く

テンプレートパラメータの制約

- `requires`で制約することもできる

```
template <class T>
    requires LessThanComparable<T>
const T& min(const T& a, const T& b)
{
    return a < b ? a : b;
}
```

テンプレートパラメータの制約

- requiresは&&で複数のコンセプトを指定できる

```
template <class T>
    requires LessThanComparable<T> &&
             CopyConstructible<T>
const T& min(const T& a, const T& b)
{
    return a < b ? a : b;
}
```

テンプレートパラメータの制約

このような制約を定義／指定することで
テンプレートパラメータが何を満たさなければならないかが
明示的になるのでよりわかりやすいエラーメッセージを出力できるようになる。

```
list<int> ls;  
sort(ls.begin(), ls.end()); // エラー！
```

エラー！ list<int>::iteratorはRandomAccessIteratorの要件を満たしません。
不足しているもの : operator<

コンセプトマップ

既存の型がどのようにコンセプトの要件を満たすのかを定義する

以下のようなコンセプトがあった場合

```
auto concept RandomAccessIterator<class X> {  
    typename value_type = X::value_type;  
}
```

RandomAccessIteratorコンセプトは

「**Xはvalue_typeという型を持っていなければならない**」という要件になるので



コンセプトマップ

vector::iteratorのようなvalue_typeを持つ型は
RandomAccessIteratorの要件を満たす

```
template <RandomAccessIterator Iter>  
void sort(Iter first, Iter last);
```

```
std::vector<int> v;  
sort(v.begin(), v.end()); // OK
```

ポインタもランダムアクセスイテレータとして振る舞えなくてはならないが、
ポインタはvalue_typeという型を持っていないので
RandomAccessIteratorの要件を満たすことはできない。

```
int ar[3];  
sort(ar, ar + 3); // エラー！
```



コンセプトマップ

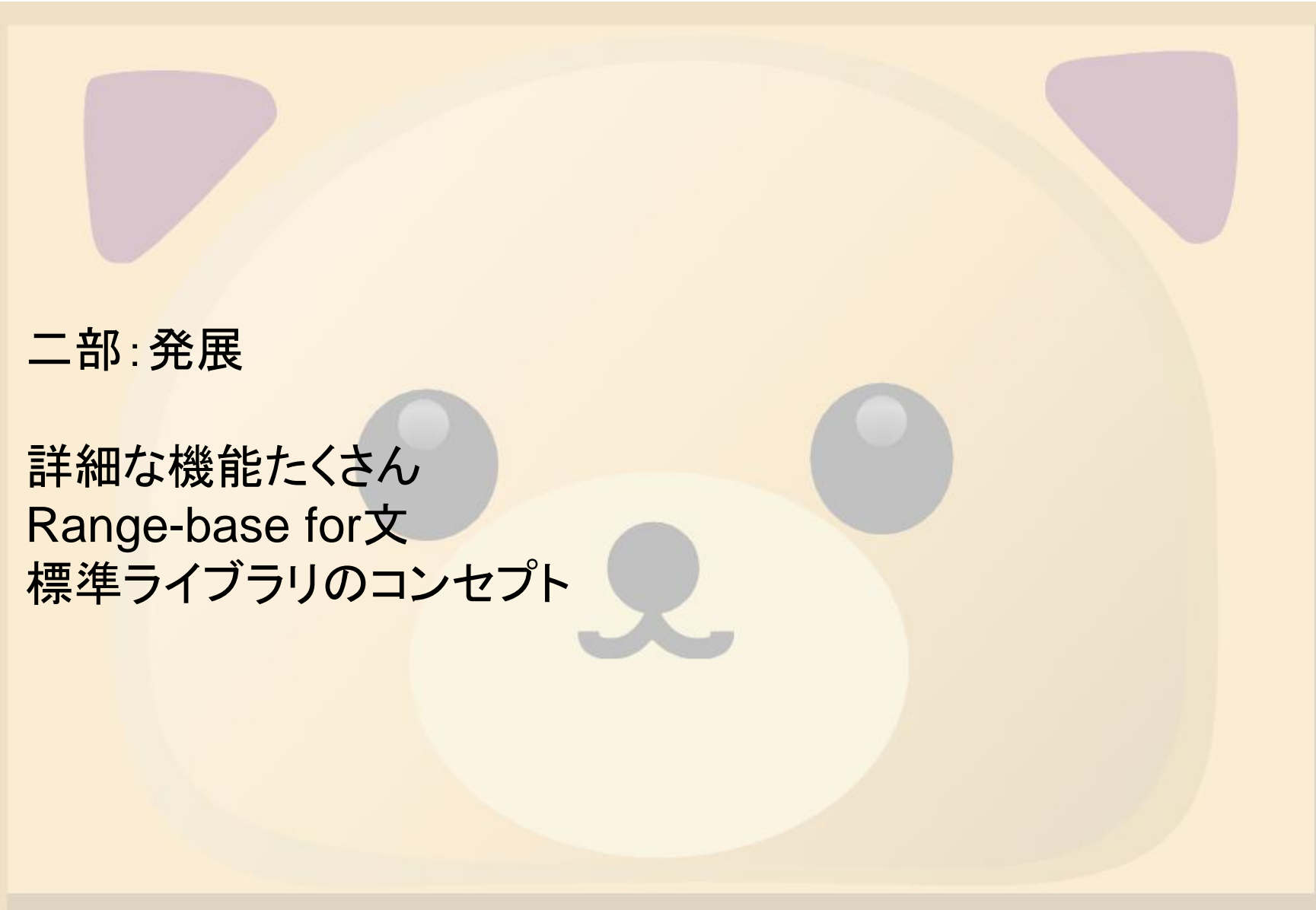
そこで、コンセプトマップという機能を使用することで
RandomAccessIteratorコンセプトをポインタで特殊化する。

```
template <class T>
concept_map RandomAccessIterator<T*> { // ポインタだったら
    typename value_type = T; // T型をvalue_typeとする
}
```

これを用意しておけば、ポインタはRandomAccessIteratorの
要件を満たすことができる。

```
int ar[3];
sort(ar, ar + 3); // OK
```





二部: 発展

詳細な機能たくさん
Range-base for文
標準ライブラリのコンセプト

関連型

以下のようなコンセプトを書いた場合、
result_typeの型はテンプレートと同様の
あらゆる型を受け取れる型になる。
(プレースホルダだと考えていいかと)

```
auto concept Function<class F> {  
    typename result_type; // 関連型。=で制約していない  
    result_type operator()(F&);  
}
```

これを関連型 (Associated Type) という。



関連型

以下のような関数オブジェクトがあった場合に

```
struct functor {  
    int operator() () const;  
};
```

functorをFunctionコンセプトに通すと

```
template <Function F>  
F::result_type foo(F f)  
{  
    return foo();  
}
```

```
int result = foo(functor());
```

F::result_typeの型は、functor::operator()の戻り値の型(int)になる。



関連型

ちなみに、コンセプト内で

```
typename result_type;
```

と書いてあるので、`result_type`が型であることが明示的になるので

```
typename F::result_type
```

のように、依存名(dependent-name)に対するtypenameを書く必要がなく、

```
F::result_type
```

と書ける。

関連要件

requiresはコンセプト内でも書ける
これを関連要件 (Associated Requirements) という。

```
auto concept Predicate<class F, class R, class... Args> {  
    requires Convertible<R, bool>;  
}
```

述語 (Predicate) の戻り値の型はboolに変換可能な型でなければならない。

関連要件

関連要件は、関連型に対しても指定できる

```
auto concept Function<class F> {  
    typename result_type;  
    requires Returnable<result_type>;  
  
    result_type operator() (F&);  
}
```

関数(Function)の戻り値の型は、
return文で返せないといけない。(配列はNG、void
はOK)

暗黙のコンセプト定義と 明示的なコンセプト定義

コンセプトの定義は2つの書き方がある。
先頭にautoを付けるものと付けないもの。

```
auto concept Fooable<class X>  
{ typename value_type = X::value_type; }  
  
concept Fooable<class X>  
{ typename value_type; }
```

autoを付けないコンセプトは、必ずコンセプトマップしなければならないコンセプト。
明示的なコンセプト (explicit concept) という。

autoを付けるコンセプトは、デフォルトのコンセプトマップが自動生成されるコンセプト。
暗黙のコンセプト (implicit concept) という。



要件指定のシンタックスシュガー

まず、以下のようなコンセプトがあった場合

```
auto concept Fooable<class X> {}
```

要件指定するとき以下のように書くと

```
template <Fooable T>  
void foo(T t);
```

FooableコンセプトのパラメータXとしてTが渡される。

要件指定のシンタックスシュガー

複数パラメータを受け取るコンセプトは
どう書けばいいのだろうか？

```
auto concept Fooable<class X, class Y> {}
```

// NG : Tを宣言する前に使ってはいけない

```
template <Fooable<T, int> T>  
void foo(T t);
```

// OK : でもrequiresを書くのはめんどくさい

```
template <class T>  
    requires Fooable<T, int>  
void foo(T t);
```



要件指定のシンタックスシュガー

複数パラメータを受け取るコンセプトの
要件指定には、autoを指定するシンタックスシュガーが用意されている

```
auto concept Fooable<class X, class Y> {}
```

```
template <Fooable<auto, int> T>  
void foo(T t);
```

この場合、autoはTに置き換えられ、以下と同じ意味になる。

```
template <class T>  
    requires Fooable<T, int>  
void foo(T t);
```



要件指定のシンタックスシュガー

autoによるシンタックスシュガーの使用例:

標準で提供される予定のCallableコンセプトはその名の通り「**関数呼び出し可能**」という制約を持つコンセプトである。

```
auto concept Callable<typename F, typename... Args> {  
    typename result_type;  
    result_type operator()(F&&, Args...);  
}
```

Callableコンセプトは、以下のパラメータを受け取る

関数ポインタ／関数オブジェクト : F

引数の型 : ...Args



要件指定のシンタックスシュガー

Callableコンセプトは使用頻度が高くなる(と思う)ので、
毎回以下のように書くのはめんどくさい。

```
template <class F>
    requires Callable<F, int>
F::result_type call(F f)
{
    return f();
}
```

autoによるシンタックスシュガーを使用すればすっきり書ける。

```
template <Callable<auto, int> F>
F::result_type call(F f)
{
    return f();
}
```

Axiom

コンセプトの定義では、`axiom`という機能を使用することにより、型のセマンティクスを定義することもできる。

たとえば、以下のコンセプトは「`operator==`で比較可能」という制約を持っているが、それと同時に「`operator==`は、`a == b`、`b == c`が成り立つなら`a == c`が成り立つ」という公理を定義している。

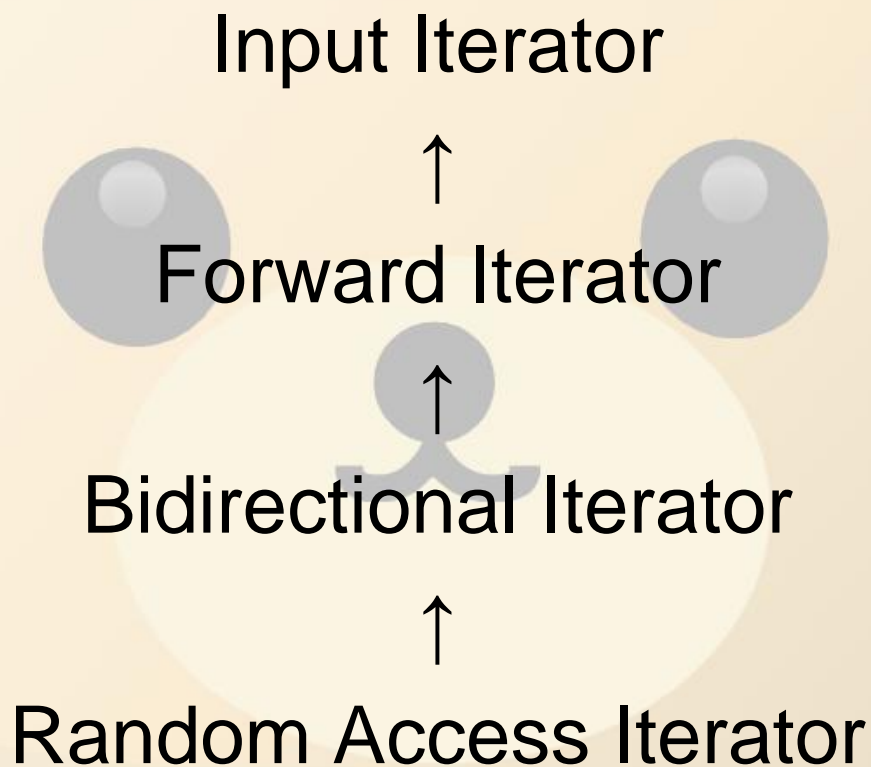
```
auto concept EqualComparable<class T> {  
    bool operator==(const T&, const T&);  
  
    axiom Transitivity(T a, T b, T c)  
    {  
        if (a == b && b == c)  
            a == c;  
    }  
}
```

ただし、`axiom`はコンパイル時に検証することは困難であるため「コンパイラでこの情報を最適化や単体テストに使ってもいいよ」という処理系依存の動作となる。



コンセプトの階層化

イテレータは以下のような階層構造を持っている。



コンセプトの階層化

コンセプトはこのような階層構造を直接表現できる。

```
concept InputIterator<class X> {}  
concept ForwardIterator<class X> : InputIterator<X> {}  
concept BidirectionalIterator<class X> : ForwardIterator<X> {}  
concept RandomAccessIterator<class X> : BidirectionalIterator<X> {}
```

これは「**コンセプトの継承**」と見なすことができる。
コンセプトの階層化 (Concept Refinement) は
関連要件 (Associated Requirements) と違い、
コンセプトマップにも影響を与える。

コンセプトの階層化と関連要件は、継承と包含の関係と同様に
“is_a”関係、“has-a”関係を表す。



コンセプトに基づいたオーバーロード

C++03ではイテレータの分類によるオーバーロードを実現するためにタグディスパッチを使用していた。

1. 分類用のタグを用意する

```
struct input_iterator_tag {};  
struct output_iterator_tag {};  
struct forward_iterator_tag : input_iterator_tag{};  
struct bidirectional_iterator_tag : forward_iterator_tag{};  
struct random_access_iterator_tag : bidirectional_iterator_tag{};
```



コンセプトに基づいたオーバーロード

2. イテレータのクラスにタグを埋め込む

```
class vector::iterator(仮) {  
    typedef random_access_iterator_tag iterator_category;  
};  
  
class list::iterator(仮) {  
    typedef bidirectional_iterator_tag iterator_category;  
};
```



コンセプトに基づいたオーバーロード

3. タグを返すメタ関数を用意する

```
template <class Iterator>
struct iterator_traits {
    ...
    typedef typename Iterator::iterator_category iterator_category;
};

template <class T>
struct iterator_traits<T*> {
    ...
    typedef random_access_iterator_tag iterator_category;
};
// ポインタはランダムアクセスイテレータとして扱う
```

コンセプトに基づいたオーバーロード

4. タグでオーバーロードする

```
template <class InputIter, class Distance>
void _advance(InputIter first, InputIter last, Distance n, input_iterator_tag);

template <class ForwardIter, class Distance>
void _advance(ForwardIter first, ForwardIter last, Distance n, forward_iterator_tag);

template <class BidIter, class Distance>
void _advance(BidIter first, BidIter last, Distance n, bidirectional_iterator_tag);

template <class RAIter, class Distance>
void _advance(RAIter first, RAIter last, Distance n, random_access_iterator_tag);

template <class Iterator class Distance>
void advance(Iterator first, Iterator last, Distance n)
{
    _advance(first, last, typename iterator_traits<Iterator>::iterator_category());
}
```



コンセプトに基づいたオーバーロード

タグディスパッチによるオーバーロードはとてもめんどくさかった。
コンセプトを使えばこれだけで済む。

```
template <InputIterator Iter, class Distance>  
void advance(Iter first, Iter last, Distance n);
```

```
template <ForwardIterator Iter, class Distance>  
void advance(Iter first, Iter last, Distance n);
```

```
template <BidirectionalIterator Iter, class Distance>  
void advance(Iter first, Iter last, Distance n);
```

```
template <RandomAccessIterator Iter, class Distance>  
void advance(Iter first, Iter last, Distance n);
```



コンセプトに基づいたオーバーロード

型特性によるオーバーロードは

C++03ではSFINAE(Substitution Failure Is Not An Error)を使用していた

// 整数型

```
template <class T>  
void foo(T t, typename enable_if<is_integral<T> >::type* = 0);
```

// 整数型以外

```
template <class T>  
void foo(T t, typename disable_if<is_integral<T> >::type* = 0);
```



コンセプトに基づいたオーバーロード

コンセプトを使えばこれだけで済む

```
template <class T>  
    requires IntegralType<T> // 整数型  
void foo(T t);
```

```
template <class T>  
    requires !IntegralType<T> // 整数型以外  
void foo(T t);
```



範囲for文

C++0xではコンテナ／配列をループするための新たなfor文が提供される。

```
vector<int> v = {1, 2, 3};  
for (int i : v) {  
    cout << i << endl;  
}
```

```
int ar[] = {1, 2, 3};  
for (int i : ar) {  
    cout << i << endl;  
}
```



範囲for文

範囲for文は、std::Rangeコンセプトを満たす
型のみ適用可能

```
concept Range<class T> {  
    InputIterator iterator;  
    iterator begin(T&);  
    iterator end(T&);  
}
```

範囲for文

標準では配列、コンテナ、initializer_list等での
コンセプトマップが提供される。

```
// 配列
template <class T, size_t N>
concept_map Range< T[N] > {
    typedef T* iterator;
    iterator begin(T(&a) [N]) { return a; }
    iterator end(T(&a) [N])    { return a + N; }
};
```



範囲for文

```
// コンテナ (vector, list, map, etc...)
template <Container C>
concept_map Range<C>
{
    typedef C::iterator iterator;
    iterator begin(C& c) { return Container<C>::begin(c); }
    iterator end(C& c)  { return Container<C>::end(c); }
}
```

ユーザー定義のコンテナも、`std::Range`を
コンセプトマップすれば範囲for文に適用可能。



標準ライブラリで提供される予定のコンセプト

// 型変換

```
auto concept IdentityOf<typename T>;
```

```
auto concept RvalueOf<typename T>;
```

// true

```
concept True<bool> {}
```

```
concept_map True<>true> {}
```

// 型特性

```
concept LvalueReference<typename T> { }
```

```
template <typename T> concept_map LvalueReference<T&> { }
```

```
concept RvalueReference<typename T> { }
```

```
template <typename T> concept_map RvalueReference<T&&> { }
```



標準ライブラリで提供される予定のコンセプト

// 演算子

```
auto concept HasPlus<typename T, typename U>;  
auto concept HasMinus<typename T, typename U>;  
auto concept HasMultiply<typename T, typename U>;  
auto concept HasDivide<typename T, typename U>;  
...
```

// 述語

```
auto concept Predicate<typename F, typename... Args>;
```

// 比較

```
auto concept LessThanComparable<typename T>;  
auto concept EqualityComparable<typename T>;  
auto concept StrictWeakOrder<typename F, typename T>;  
auto concept EquivalenceRelation<typename F, typename T>;
```



標準ライブラリで提供される予定のコンセプト

// コンストラクタとデストラクタ

```
auto concept HasConstructor<typename T, typename... Args>;  
auto concept Constructible<typename T, typename... Args>;  
auto concept HasDestructor<typename T>;  
auto concept HasVirtualDestructor<typename T>;
```

// コピーとムーブ

```
auto concept MoveConstructible<typename T>;  
auto concept CopyConstructible<typename T>;  
auto concept MoveAssignable<typename T>;  
auto concept CopyAssignable<typename T>;
```



標準ライブラリで提供される予定のコンセプト

// C++言語がサポートする型特性

```
concept Returnable<typename T>;
```

```
concept PointeeType<typename T>;
```

```
concept ClassType<typename T>;
```

```
concept PolymorphicClass<typename T>;
```

```
concept IntegralType<typename T>;
```

```
concept FloatingPointType<typename T>;
```

```
concept SameType<typename T, typename U>;
```

```
concept DerivedFrom<typename Derived, typename Base>;
```

...

紹介しきれなかったものを簡単に紹介

- late_check
テンプレートのインスタンス化時までコンセプトのチェックを遅らせる
- 名前付き要件
requiresに名前を付ける
- スコープ付きコンセプトマップ
別な名前空間でコンセプトマップを定義する
- ルックアップ
コンセプトにもADLあるよ。
- 否定要件
requiresを指定するときに頭に「!」を付けると「コンセプトの要件を満たさなかった場合」という要件にできる

参考

- N1758 Concepts for C++0x

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1758.pdf>

- N2773 Proposed Wording for Concepts(Revision 9)

<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2773.pdf>

- An Introduction to Concepts in C++0x

<http://tinyurl.com/c38f7o>

- Concepts: Extending C++ Templates for Generic Programming

<http://www.generic-programming.org/languages/conceptcpp/papers/accu07.ppt>



まとめ

- テンプレートパラメータの要件が明確になる
- コンセプトマップによってサードパーティ製のライブラリでも標準コンセプトの要件を満たすことができる
- コンセプトによるオーバーロードは超強力！