C++0x

言語の未来を語る

高橋 晶(アキラ)

Blog: Faith and Brave – C++で遊ぼう

Agenda

- What's C++0x
- Angle Bracket
- Initializer List
- auto
- decltype
- Delegating Constructor
- Extending sizeof
- Raw String Literal
- Defaulted and Deleted Functions
- Member Initializers
- nullptr
- constexpr
- Template Aliases
- static_assert
- Variadic Templates
- Concept
- Range-base for loop
- RValue Reference・Move Semantics
- Lambda Expression
- New Function Daclarator Syntax

What's C++0x (C++0xってなに？)

- C++の次バージョン(現在はC++03)
- 「0x」とは200x年の意味で、C++09を目指している
  (もしかしたら C++10 になっちゃうかも)
- エキスパートよりも初心者のための言語拡張を行う

Angle Bracket(山カッコ)

- C++03では以下のコードはコンパイルエラーになる

```
vector<basic_string<char>> v;
                          // エラー！ operator>> が使われている
```

- 連続した山カッコがシフト演算子と判断されてしまう
- C++0xではこの問題が解決される

Initializer List(初期化子リスト)

- ユーザー定義クラスで配列のような初期化を可能にする
- Bjarne氏曰く「何で今までできなかったのかわからない」
- std::initializer_listクラスを使用する

```
// 初期化
vector<int> v = {3, 1, 4};

// 戻り値
vector<int> factory() { return {3, 1, 4}; }

// 引数
void foo(const vector<int>& v) {}
foo({3, 1, 4});
```

auto

- 型推論
- autoキーワードの意味が変更になる(めずらしい)
- autoで宣言された変数の型は右辺から導出される

```
vector<int> v;
vector<int>::iterator it = v.begin(); // C++03
auto                  it = v.begin(); // C++0x
```

const/volatile修飾や、参照(&)・ポインタ(*)等を付加することもできる(const auto&, auto*, etc...)

decltype

- 計算結果の型を求めることができる
- 他の機能で型を推論に使われたり、インスタンスからメンバの型を取得したり…

```
int x = 3;
decltype(x) y = 0;    // int y = 0;
decltype(x*) z = &x; // int* z = &x;


int foo() { return 0; }
decltype(foo()) x = foo(); // int x = foo();
```

Delegating Constructor

- 移譲コンストラクタ
- コンストラクタ内で、他のコンストラクタを呼べる

```
class widget {
  int x_, y_, z_;

  widget(int x, int y, int z)
    : x_(x), y_(y), z_(z) {}
public:
  widget()
    : widget(0, 0, 0) {} // widget(int, int, int)
};
```

Extending sizeof(拡張sizeof)

- インスタンス作成せずにメンバ変数を sizeof できるようになる

```
struct hoge {
  int id;
};

sizeof(hoge::id); // 今まではできなかった
```

Raw String Literal

- Rプレフィックスを付加した文字列は
  エスケープシーケンスを無視するようになる

```
// C++03
string  path  = “C:¥¥abc”;
string  data  = “Akira¥¥nTakahashi”;
wstring wpath = L”C:¥¥abc”;

// C++0x
string  path  = R”C:¥abc”;
string  data  = R”Akira¥nTakahashi”;
wstring wpath = LR”C:¥abc”;
```

Defaulted and Deleted Functions

- コンパイラによって自動生成される関数の制御ができるようになる

```
struct hoge {
    hoge() = default;
    // コンパイラが自動生成するデフォルトコンストラクタを使う

    // コピー禁止
    hoge(const hoge&) = delete;
    hoge& operator=(const hoge&) = delete;
};
```

Member Initializers(メンバ初期化子)

- メンバ変数の宣言時に初期化できるようになる

```
struct hoge {
  int    age = 23;
  string name(“Akira”);
};
```

- コンストラクタでの初期化子リストが優先される

nullptr

- ヌルポインタを表すキーワード nullptr を追加
- nullptrが導入されれば、0とNULLを非推奨とすることができる

```
int* p = nullptr;                    // C++0x
int* p = static_cast<int*>(0); // C++03
int* p = NULL;                       // C...(void*)0
```

constexpr

- 定数式
- コンパイル時に実行される式を作成することができる

```
constexpr int size(int x) {
  return x * 2;
}

int ar[size(3)]; // OK...配列のサイズは6
```

いろいろと制限があるらしい(再帰ができない等...)

Template Aliases

- テンプレートを使用して型の別名を付けることができるようになる(いわゆる template typedef)

```
template <class T>
using Vec = vector<T>;

Vec<int> v;
```

- 型を返すメタ関数が書きやすくなる
  (もしくはいらなくなる)

static_assert

- コンパイル時アサート
- 静的な条件と、エラーメッセージを指定する

```
template <class T>
void foo(T value) {
  static_assert(is_integral<T>::value,
                "not integral");
}

foo(3);     // OK
foo(3.14); // エラー！ not integral
```

Variadic Templates

- 可変引数テンプレート
- 基本的に再帰を使って処理する

```
template <class T>
void print(T arg) {
  cout << arg << endl;
}
```

```
template <class Head, class... Tail>
void print(Head head, Tail... tail) {
  cout << head << endl;
  print(tail...);
}

print(1, 2, 3, 4, 5, 6, 7);
```

ちなみに、sizeof...(Args); とすると、型の数を取得できる

Concept

- 型に対する制約
- テンプレートをより簡単にし、より強力にするもの

コンセプトには以下の3つの機能がある

- Concept definitions
- Requirements clauses
- Concept maps

Concept definitions

- 型に対する要求を定義する
  「型Tはxxメンバを持っていなければならない」
  「型Tはyy型を持っていなければならない」

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

このコンセプトは、「型T は operator< を持っていなければならない」という要求

Requirements clauses

- 型Tに必要な要求(コンセプト)を指定する

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

これにより
「型`T`は`LessThanComparable`の要求を満たさなければならない」
といった制約を行うことができる

ここまででうれしいこと

- 人間が読めるコンパイルエラーを出力してくれる

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

「第1引数はRandomAccessIteratorの要求を満たしません」
といったエラーメッセージを出力してくれるだろう

Concept maps その1

- 既存の型がどのようにコンセプトの要求を満たすかを定義する

以下のようなコンセプトとそれを要求する関数があった場合

```
concept RandomAccessIterator<class T> {
  typename value_type;
  typename difference_type;
}

template <RandomAccessIterator Iterator>
void sort(Iterator first, Iterator last) {
  ...
}
```

Concept maps その2

以下のような結果になる

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

int ar[] = {3, 1, 4};
sort(ar, ar + 3);
    // エラー！ポインタはRandomAccessIteratorの要求を満たしません
```

ポインタはランダムアクセスイテレータとして使用できなければならない

Concept maps その3

- そこで、concept_mapを使用して RandomAccessIteratorコンセプトを特殊化する

```
template <class T>
concept_map RandomAccessIterator<T*> {
  typedef T          value_type;
  typedef ptrdiff_t  difference_type;
}
```

これにより、ポインタを`RandomAccessIterator`として使える

Concept mapsがあるとできること

- Containerコンセプトを使用することにより
  配列をコンテナとして使用することができる

```
template <Container Cont>
void sort(Cont& c) {
  sort(c.begin(), c.end());
}

std::vector<int> v;
sort(v); // OK

int ar[] = {3, 1, 4};
sort(ar); // OK
```

書ききれないコンセプト

- Range-baseのアルゴリズムは(まだ？)提供されない
- requiresはカンマ区切りで複数指定可能
- late_checkブロックで従来の型チェックも可能
- コンセプトでのオーバーロードも可能
- Scoped Concept Mapsとか・・・
- Axiomとか・・・

- コンセプトは超強力！！！

Range-base for loop

- コレクションを走査するfor文
- std::Rangeコンセプトを使用している

```
vector<int> v;
for (const int& item : v) {
  cout << item << endl;
}
```

初期化子リストを渡すことも可能

```
for (int item : {3, 1, 4}) {
    ...
}
```

右辺値参照・Move Semantics

- 右辺値(一時オブジェクト)はどうせ消えちゃうのだから
  内部のメモリを移動しても問題ないでしょ。というもの
- 移動された一時オブジェクトは破壊される
- 右辺値参照には&&演算子を使用する

```
template <class T>
class vector {
public:
  vector(const vector&); // Copy Constructor
  vector(vector&&);      // Move Constructor
};
```

左辺値と右辺値

- 左辺値とは
  - 名前が付いたオブジェクトを指す

- 右辺値とは
  - 名前がないオブジェクトを指す
  - 関数の戻り値は、参照やポインタでない限り右辺値(一時オブジェクト)である

右辺値参照を使ってみる

右辺値参照は、使う側はとくに気にする必要がない(ことが多い)

```
vector<int> factory()
{
  vector<int> v;
  ...
  return v;
}

vector<int> v = factory(); // Move Contructor
```

右辺値参照を使えば大きいオブジェクトを気軽に戻り値にできる

右辺値参照のための標準関数

- <utility>にstd::moveとstd::forwardという2つの関数が用意される

```
namespace std {
  template <class T>
  struct identity {
    typedef T type;
  };

  template <class T>
  inline typename remove_reference<T>::type&& move(T&& x)
    { return x; }

  template <class T>
  inline T&& forward(typename identity<T>::type&& x)
    { return x; }
}
```

・std::move は左辺値を右辺値に変換する関数
・std::forward は右辺値を安全に転送する関数

基本的な
Move ConstructorとMove Assinmentの書き方

```
class Derived : public Base {
    std::vector<int> vec;
    std::string      name;
public:
    // move constructor
    Derived(Derived&& x)
        : Base(std::forward<Base>(x)),
          vec(std::move(x.vec)),
          name(std::move(x.name)){}

    // move assignment operator
    Derived& operator=(Derived&& x) {
        Base::operator=(std::forward<Base>(x));
        vec  = std::move(x.vec);
        name = std::move(x.name);
        return *this;
    }
};
```

Lambda Expression(ラムダ式)

- 匿名関数オブジェクト

```
vector<int> v;
find_if(v.begin(), v.end(),
            [](int x) { return x == 3; });
```

以下はラムダ式によって生成される匿名関数オブジェクト(簡易的)

```
struct F {
  bool operator()(int x) {
    return x == 3;
  }
};
```

環境のキャプチャ

- デフォルトのキャプチャ方法

```
vector<int> v;
int sum = 0;
int rate = 2;
for_each(v.begin(), v.end(), [&](int x) { sum += x * rate; });
```

[&] : デフォルトのキャプチャ方法は参照

[=] : デフォルトのキャプチャ方法はコピー

[] : 環境なし

- キャプチャリスト(個別指定)

```
for_each(v.begin(), v.end(), [=, &sum] { sum += x * rate; });
```

デフォルトのキャプチャ方法はコピー、sumは参照でキャプチャ

戻り値の型

- ラムダ式の戻り値の型は明示的に指定することもできる

```
[](int x) -> bool { return x == 3; }
```

- 省略した場合は、return文をdecltypeした型が戻り値の型

以下の2つのラムダ式は同じ意味である

```
[](int x) -> decltype(x == 3) { return x == 3; }
[](int x)                     { return x == 3; }
```

return 文を省略した場合、戻り値の型は void になる

書ききれなかったラムダ

- メンバ関数内でのラムダ式はそのクラスのfriendと見なされ、
  privateメンバにアクセスできる
- 参照環境のみを持つラムダ式はstd::reference_closureを継承したクラスになる
  (キャプチャした変数をメンバに持たずにスコープをのぞき見る)
- 参照環境のみを持つラムダ式はスコープから外れるとその実行は未定義
  (つまり、参照ラムダを戻り値にしてはいけない)
- const/volatile修飾も可能... []() const {}
- 例外指定も可能... []() throw {}

新たな関数宣言構文(戻り値の型を後置)

- 現在

```
vector<double> foo();
```

- 提案1:戻り値の型を後置する関数宣言構文

```
auto foo() -> vector<double>;
```

- 提案2:(ラムダ式と)統一された関数宣言構文

```
[]foo() -> vector<double>;
```

戻り値の型を後置できると、引数をdecltypeした型を戻り値の型にすることができる
ラムダ式と同様に戻り値の型を推論することも検討中

書ききれなかった言語仕様

- char16_t/char32_t
- Alignment
- union の制限解除
- Strongly Typed Enums
- explicit conversion
- 継承コンストラクタ
- ユーザー定義リテラル
- 60進数リテラル(?)
- Nested Exception
- Uniformed initialization
- etc...

まとめ

- C++0x には、プログラミングをより簡単にするための拡張が数多くあります
- 紹介しきれませんでしたが、標準ライブラリもかなり強力になっています
- C++0x で遊ぼう！！

参考サイト

- C++ Standards Committee Papers

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/

- Wikipedia(英語)

http://en.wikipedia.org/wiki/C%2B%2B0x

- Faith and Brave – C++で遊ぼう

http://d.hatena.ne.jp/faith_and_brave/

- Cry's Diary

http://d.hatena.ne.jp/Cryolite/

- ntnekの日記

http://d.hatena.ne.jp/ntnek/

- かそくそうち

http://d.hatena.ne.jp/y-hamigaki/