解 説

まもなく訪れるC++0x

高橋 晶 （株）ロングゲート

C++0x とは
what is C++0x

Bjarne Stroustrup によって設計されたプログラミング言語 C++ は，1998 年に ISO 標準として制定され，2003 年に小さな改訂が行われた．現在のバージョンは C++03 と呼ばれている．そして現在策定中の規格は，200x 年中の策定完了を目指していたため C++0x と呼ばれている．だが，現在はすでに 2011 年である．そう，2009 年中の策定は終了できなかったのである．C++1x に改名しようという話も一時期あったが，C++0x という名称で書かれた多くの文献や記事があったため，混乱を避けるために C++ 次期バージョンの仮名称は C++0x のままとなっている．策定作業は 2011 年か 2012 年に完了する見込みである．おそらく C++11 か C++12 という名称で呼ばれることになるだろう．

新たな言語機能とライブラリ
language feature, and library

では，C++0x で導入される新たな機能のいくつかを紹介していこう．

⇨範囲 for 文と初期化子リスト

C++03 において，コンテナを走査するためには，非常に冗長なコードを記述しなければならなかった．たとえば以下は，コンテナのすべての要素を標準出力に出力するコードである：

```
std::vector<int> v;
v.push_back(3);
v.push_back(1);
v.push_back(4);

for (std::vector<int>::const_iterator
     it = v.begin(), last = v.end();
     it != last; ++it) {
  std::cout << *it << std::endl;
}
```

これは非常に冗長であるとともに，「++it」を「it++」と書いてしまっただけで非効率なコードになり得るという問題もあった．C++0x では範囲 for 文 (Range-based for statement) と呼ばれる新たな for 文が導入される．以下はこれを利用して先のコードを直した例である：

```
const std::vector<int> v = {3, 1, 4};
for (int x : v) {
  std::cout << x << std::endl;
}
```

書き直す前よりも非常にシンプルで，間違いも犯しにくくなった．for 文以外にも変わったことがある．コンテナの初期化を行っている部分だ．このように C++0x では，組込み配列のような初期化がユーザ定義型でも定義可能となるため，より直感的な記述が可能となる．

⇨右辺値参照とムーブセマンティクス

C++03 では，大きなオブジェクトを返す関数

を定義する場合に一時オブジェクトのコストが高かったため，コストを抑えるために小手先の手法に頼らなければならないことがあり，それがコストのかからない抽象化を難しくしていた．そのような問題に対応するため，C++0x では右辺値参照 (Rvalue References) という機能を採用することによって，現在ある「コピー (copy)」のセマンティクスに加えて「移動 (move)」という新たなセマンティクスを導入する．たとえば，右辺値参照の導入に伴い，std::vector に以下のようなコンストラクタが追加される：

```
vector(vector&&)
```

この「&&」記号が，右辺値参照と呼ばれるものを表現する記号である．これは，右辺値と呼ばれる名前のない値からオブジェクトの所有権を他のオブジェクトに移動するために使われる．大量の要素を持つ std::vector を関数の戻り値として返すケースを考えよう：

```
std::vector<int> get_primes() {
  std::vector<int> result;
  … 素数列を計算 …
  return result;
}

const std::vector<int> primes =
                              get_primes();
```

get_primes() 関数で返した素数列を受け取った場合，C++03 では素数列のすべての値を primes 変数にコピーし，その後 get_primes() で返された一時オブジェクトが破棄されていた．C++0x の場合，このコードでは get_primes() で返された一時オブジェクトからのコピーは作成されず，一時オブジェクトの内部で保持しているデータが primes 変数に移動される．

ムーブセマンティクスという考え方で重要なのは，「一時オブジェクトをコピーするのは無駄だ」ということである．コピーとは，2 つの等価なオブジェクトを作り出すことであるが，一時オブジェクトは式の評価が終わった時点で破棄されるデータであるため，コピーして自分を破棄する，という行為は無駄なコストである．すぐになくなるということは，そのオブジェクト自身は破壊してもその後に問題にはならないデータだということである．そのため，一時オブジェクトであることを判定し，内部のメモリや，メモリを解放する責任などをそのまま移してしまえばいいのである．右辺値参照は，その後すぐに消えてしまう破棄してかまわない一時オブジェクトかどうかを判定する手段と，それを参照する機能を提供するものである．

これが右辺値参照によるムーブセマンティクスであり，一時オブジェクトが問題となっていた多くの場面でパフォーマンスを改善する．右辺値参照の詳細な使用法や技術的な詳細はここではくわしく説明しないが，全体的に右辺値参照対応の入る新たな標準ライブラリをベースにしてプログラムを作成すれば，右辺値参照のコンストラクタと代入演算子が自動的に生成されるため，多くのユーザが何も気にすることなく右辺値参照の恩恵を受けることができる．

⇨マクロを不要にする拡張

C 言語から引き継いだ #define によるマクロ定義は，C++03 においてもなお多くの場面で使われていた．マクロは単純な置き換えであるため，マクロ内でバグが起きた場合に原因の特定を難しくした．マクロは，型安全な可変引数，定数式，デバッグ時にのみ動作するコード，独自構文 (DSEL) の実装といった大きく 4 つのケースで使われている．C++0x では，これらのうちの 2 つにおいてマクロを不要にする拡張を導入する．

可変引数テンプレート

C 言語においても可変引数と呼ばれる機能は存在し，printf などで使われていたが，C のそれは型安全ではなく，%d などのプレースホルダをユーザ定義することもできなかった．C++0x では，可変引数テンプレート (Variadic Templates) と呼ばれる機能を導入し，あらゆる型で振る舞えるテンプレートでの可変引数を提供することで利便性と型安全性をユーザにもたらす．以下は，可変引数テンプレートに

よる関数オブジェクトの定義例である：

```
template <class F> struct Function {
  F f;
  Function(F f_) : f(f_) {}
  template <class... Args>
  void operator()(Args... args)
    { f(args...); }
};

void foo(int x, double d)
  { std::cout << x << ',' << d; }

Function<void(*)(int, double)> f = foo;
f(1, 3.14); // OK
f(1, 3.14, "abc"); // エラー！
                   // 引数の数が異なる
```

これは，メンバ変数として保持している関数ポインタあるいは関数オブジェクトに，関数呼び出し演算子で渡された引数を転送するコードである．関数呼び出し演算子の定義に可変引数テンプレートが使われており（template <class... Args> の部分），間違った型の値を渡した場合や，引数の数が異なっていた場合にはコンパイル時にエラーになる．C++03 ではこのようなコードを書こうとした場合，任意の数の引数を転送できるようにするためにマクロによってコードを自動生成していた．C++0x では，可変引数テンプレートによってこのような自動生成が必要なくなる．

constexpr

C++03 では，関数適用の結果が定数式にはなり得ない，という問題があったため，マクロが使われていた．const 定義されたリテラルはコンパイル時定数として扱われるのに，その値に対して関数適用した結果はコンパイル時定数にはなり得ないのである．

```
// 実行時に評価される関数
int abs(int x) { return x < 0 ? -x : x; }

const int a = -12;
// OK．定数値なのでコンパイル時検証可能
static_assert(a == -12, "not equal");

const int b = abs(a);
// エラー！コンパイル時定数ではない
static_assert(b == 12,
          "abs result not equal");
```

ここで使用している static_assert も C++0x の拡張の 1 つで，コンパイル時条件でのアサートを行うものだ．コンパイル時条件しか渡すことのできない static_assert によって，上記コードの abs() を適用した結果が定数式ではないということが分かる．こういった関数をコンパイル時に評価できないという理由から，C++03 ではマクロが使われ，ときにはメタ関数と呼ばれる関数もどきを定義せざるを得なかった．C++0x ではこのような定数式の問題を解決するため，constexpr という定数式を定義するためのキーワードが導入される．先ほどの abs() 関数を constexpr で定義してみよう：

```
constexpr int abs(int x)
  { return x < 0 ? -x : x; }

const int a = -12;
const int b = abs(a);

// OK.定数式として扱える.
static_assert(b == 12, "not equal");
```

constexpr 修飾した関数は，コンパイル時に評価可能な場合はコンパイル時に評価されてその結果は定数式として扱え，コンパイル時に評価できないような式の場合は，従来通りの実行時に評価される関数となる．つまり，constexpr で関数を定義しておけば，実行時に有用な関数をコンパイル時にも扱うことができるのである．

constexpr は数値計算，検証，文字列操作等，多くの計算をコンパイル時に行い，実行時のコストを軽減させる可能性を秘めており，マクロの問題を解決する以上の価値がある．

ほかの残されたマクロの用途である「デバッグ時にのみ動作するコード」「独自構文の実装」は，マクロの本質的な利用方法であるため，現状では問題視されていないが，独自構文の定義は抽象構文木（AST）レベルでカスタマイズするという提案は今後出てくるかもしれない．

⇨新たなライブラリ

非常に多くのライブラリ拡張があるため，ここですべてを紹介することはできないが，Boost C++ Libraries で得られた経験をもとに導入される実績あるライブラリとして，スマートポインタ，乱数生成，正規表現などが取り入れられ，さらに大規模ファイルを扱うための規定変更や，コンテナやアルゴリズムのような，既存ライブラリに対するさらなる強化が予定されている．ここでは，特に有用で，かつ筆者が広く使われてほしいと願うスマートポインタを紹介しよう．

C 言語から受け継いだポインタという機能は，主に動的なリソース確保に使われていたが，プログラマの人的ミスによる解放忘れが起きやすく，「メモリリーク」または「リソースリーク」と呼ばれる追いにくいバグの元凶になっていた．スマートポインタとは，確保したリソースを自動的に解放してくれるポインタである．C++0x 標準で提供予定のスマートポインタは shared_ptr と unique_ptr の 2 つで，これらを要件に合わせて使い分けることになる．まず，shared_ptr は，所有権を共有するスマートポインタである．ポインタを複数個所から参照し，誰も参照しなくなったタイミングで適切に解放してほしいときに使用する．

```
void foo() {
  std::shared_ptr<X> p(new X);
                          // pが所有
  {
    std::shared_ptr<X> q = p;
                      // pとqが所有
    q->bar();
  } // qが所有権を放棄.
    // pが所有しているので
    // ここではまだ解放されない.
} // pが所有権を放棄.
  // 誰も所有していないので解放される.
```

次に unique_ptr は，たった一人しか所有できないことが保証されたスマートポインタである．コピー操作が禁止されているため，ローカルでメモリ確保を行い即時解放する場合などで安全に使用することができる：

	VC++ 2010	GCC 4.6	ICC 12.0
範囲 for 文	×	○	×
初期化子リスト	×	○	×
右辺値参照	○	○	○
可変引数テンプレート	×	○	×
Constexpr	×	○	×
static_assert	○	○	○

コンパイラ正式名称：
VC++2010 : Microsoft Visual C++ 2010
GCC 4.6 : GNU C++ Compiler 4.6
ICC 12.0 : Intel C++ Compiler 12.0

表 -1　コンパイラの C++0x 対応状況

```
void foo() {
  std::unique_ptr<X> p(new X);
//std::unique_ptr<X> q = p;
            // エラー！コピーできない
  p->bar();
} // ここで解放される.
```

C++0x では，レガシーコードとの連携を除き，生のポインタを使うことはもはやないだろう．

コンパイラの C++0x 対応状況 compiler status

C++0x は，すでに多くの主要コンパイラが対応を始めている．本稿で紹介した機能について，主要コンパイラの対応状況は**表 -1** の通りである．

C++0x 策定への期待 hopes for C++0x

C++0x は，今回紹介した範囲の機能だけでも非常に有用で，C++03 から移行する価値は十分にある．特に C++ での開発現場は，日本だけかもしれないがサードパーティライブラリを嫌う傾向にある．そのような現状では，標準ライブラリの強化によってスマートポインタが手に入ることは，多くの開発現場において救済になり得るだろうし，マルチスレッド機能を使用できるあらゆる環境で同じスレッド

ライブラリを使える，というのは移植性や学習においてもプログラマの負担を軽減することにつながる．また，言語機能の多くはヒューマンエラーを防止することにもつながるため，小規模開発から大規模開発まで，非常に高い効果を発揮することになるだろう．C++0x の規格が早く策定され，筆者を含む多くのユーザに良い影響をもたらすことを切に願う．

参考文献

1) C++ Standards Committee Papers
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/
（C++ 言語および標準ライブラリの提案書や規格の草案がここで参照できる）
2) Range-Based For Loop Wording (Without Concepts)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2930.html
（範囲 for 文の最新の提案書．動機，構文などを参照することができる）
3) Proposed Wording for Variadic Templates (Revision 2)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2242.pdf
（可変引数テンプレートの最新の提案書．型安全な `printf`，型リストの操作例などが記載されている）
4) Generalized Constant Expressions—Revision 5
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2235.pdf
（`constexpr` の最新の提案書．基本的な動機や，概念を学ぶことができる）
5) A Brief Introduction to Rvalue References
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html
（右辺値参照の入門記事．右辺値参照の基本的な使い方，ムーブセマンティクスの概念などを学ぶことができる）
6) C++0x Compiler Support
http://wiki.apache.org/stdcxx/C++0xCompilerSupport
（主なコンパイラの C++0x サポート状況を確認することができる）

（2011 年 4 月 29 日受付）

高橋　晶　faithandbrave@gmail.com

1985 年 3 月 1 日生まれ．C++ 標準化委員会エキスパートメンバ．著書『C++ テンプレートテクニック』（ソフトバンククリエイティブ，2009 年）．