C++14

変数テンプレート

高橋 晶(Akira Takahashi)
faithandbrave@longgate.co.jp
2013/10/26(土) C++14規格レビュー勉強会

はじめに

- この発表は、C++14のコア言語に導入される予定の
  「変数テンプレート(Variable Templates)」に関するレビュー資料です。

  - 提案文書：
    http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3651.pdf

  - 日本語訳：
    http://dl.dropboxusercontent.com/u/1682460/translation/C++14/n3651_variable_templates.html

概要

- この提案の狙いは、パラメータ化された定数の定義と使用をシンプルにすることである。
- これは、constexpr変数テンプレート(Variable Templates)の宣言によって許可する。
- その結果として、よりシンプルなプログラミングルールを覚えやすくする。これによって、現在知られている回避策を、より予測可能な慣習と意味論で置き換える。

変数テンプレートの使い方

- 数学の定数であるπを、浮動小数点数型の精度を指定して直接的に表現したい。

```
template <typename T>
constexpr T pi = T(3.14159265358979323385);
```

変数テンプレートの使い方

- そしてこれをジェネリックな関数内で使用したい。たとえば、与えられた半径から円の面積を計算する。

```
template <typename T>
T area_of_circle_with_radius(T r)
{
  return pi<T> * r * r;
}
```

変数テンプレートの使い方

- 変数テンプレートの型は、組み込み型に制限されない。ユーザー定義型も使用可能だ。
- たとえば、ここにパウリ行列(Pauli matrices)の基本的な定義がある(これは量子力学で使用する)。

```
template <typename T>
constexpr pauli<T> sigma1 = { { 0, 1 }, { 1, 0 } };

template <typename T>
constexpr pauli<T> sigma2 = { { 0, -1i }, { 1i, 0 } };

template <typename T>
constexpr pauli<T> sigma3 = { { 1, 0 }, { -1, 0 } };
```

- pauli<T>は2x2行列のcomplex<T>型として定義される。

これまでの回避策

- 変数テンプレート宣言がないC++11までは、以下の2つの回避策がとられていた。
  - クラステンプレートのconstexpr静的データメンバ
  - constexpr関数テンプレートによって返される結果値

回避策1. constexpr静的データメンバ

- 標準のstd::numeric_limitsクラスで、この回避策がとられている。

```
template <typename T>
struct numeric_limits {
  static constexpr bool is_modulo = ...;
};

// ...

template <typename T>
constexpr bool numeric_limits<T>::is_modulo;
```

回避策1. constexpr静的データメンバ

静的データメンバの主な問題：

- それらは「重複する」宣言を要求する： その定数がodr-usedである場合、クラステンプレートの内側に一度、クラステンプレートの外側に一度、「本物の(real)」定義を提供する。

- プログラマは、同じ宣言を2回宣言する必要性に怒り、混乱する。
対照的に、「普通の(ordinary)」定数宣言では重複した宣言は必要ない。

つまり何を言ってるのか？

回避策1. constexpr静的データメンバ

```
template <class T>
struct X {
  static constexpr bool is_modulo = false;
};
/*
template <class T>
constexpr bool X<T>::is_modulo;
*/
int main()
{
  constexpr const bool& x = X<int>::is_modulo; // リンクエラー！
}
```

- 宣言しかされていない段階で、is_moduloを「**使用**」している。
- この段階では、is_moduloに実体がないため、ポインタや参照はとれない。
- 上記コメントアウトを外して、is_moduloを定義しなければならない。

回避策2. constexpr関数テンプレート

- この回避策をとっているものには、`std::numeric_limits`の静的メンバ関数や、`boost::math::constants::pi<T>()`関数などがある。
- こちらの回避策では、静的データメンバが持つ「重複宣言」問題は起こらない。
- この回避策の問題は、データの使用方法(const参照なのか非const参照なのか、もしくはコピーなのか)を、**提供側が事前に選択しなければならない**、ということ。
- もしコピーを返す方法が一つだけ用意されていたとして、組み込み型なら大した問題にはならないが、行列や多倍長演算型の場合に致命的になる。

解決策

constexpr変数テンプレートを使いましょう

具体的な規格の変更内容

- もともと、構文的にはあらゆる宣言にテンプレートを付けることが可能になっている。
- 規格への変更内容は、変数宣言をテンプレートの許可リストに加えるだけ。

特殊化について

- 変数テンプレートは、特殊化、および部分特殊化をサポートする。

所感

- 元々浅い理解だったときは、constexpr関数で十分だと考えていたが、odr-usedと、使用方法のユーザー側選択の説明で、納得が行った。
- 懸念事項としては、変数テンプレートは可変個変数とかができてしまうはずなので、
  「テンプレートがいくつインスタンス化されたかの個数と、インスタンス化された型リストを取得したい」
  という要望が出てきそうで怖いと感じた。
  - → でもそれは静的データメンバも同じだった。
- 現状の機能と動機については、全く問題ないと考える。

可変個変数

- テンプレートパラメータごとに異なるインスタンスを持つので、同じ変数名でも別な型と値を持てる。(Clang 3.5になる予定のtrunkで確認)

```
template<int x>
constexpr char y = 3;

template<>
constexpr double y<42> = 2.5;

int main()
{
  constexpr char c = y<17>;

  constexpr double d = y<42>;

  static_assert(c == 3, "");
  static_assert(d == 2.5, "");
}
```

http://stackoverflow.com/questions/19108345/c1y-c14-variable-template-specialization

標準ライブラリへの採用展望

- C++14での、Type Traitsのエイリアステンプレート版追加にともない、同提案者のVicente J. Botet Escribaさんが、値を返すメタ関数の変数テンプレート版を提案しようとしている(C++14には間に合わないだろう)。

- RFC: TypeTraits Variables - std-proposals
https://groups.google.com/a/isocpp.org/forum/?hl=en&fromgroups#!topic/std-proposals/QOYLLJjH98k

```
template<typename T>
constexpr bool is_reference_c = is_reference<T>::value;
```

参考文献

- N3552 Walter Brown. Introducing Object Aliases.
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3552.pdf
- N3615 Gabriel Dos Reis. Constexpr Variable Templates.
  http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3615.pdf