C++
テンプレートメタプログラミング

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/

はじめに

Q.テンプレートメタプログラミングってなんぞ？

A.テンプレートのインスタンス化を利用して
　コンパイル時に評価されるプログラムを書こうぜ！
　っていうパラダイム

メタ関数

- コンパイル時に評価される関数

```
template <class T> // Tがパラメータ
struct identity {
  typedef T type; // typeが戻り値
};

identity<int>::type i; // int i;
```

テンプレートパラメータを関数のパラメータ、
入れ子型(nested-type)や
クラス内定数(static const T)を関数の戻り値を見なす。

特殊化で型特性の判別と条件分岐

テンプレートの特殊化を使って、
型がどんな特性を持ってるのかを判別する

以下はTがvoidかどうかを判別するメタ関数

```
template <class T>
struct is_void {        // void以外だったらfalseを返す
  static const bool value = false;
};

template <>
struct is_void<void> { // voidだったらtrueを返す
  static const bool value = true;
};

bool a = is_void<int>::value;  // bool a = false;
bool b = is_void<void>::value; // bool b = true;
```

部分特殊化で型特性の判別

- 部分特殊化使った場合。
  パターンマッチみたいなもん。

```
template <class T>
struct is_pointer {      // ポインタ以外はfalseを返す
  static const bool value = false;
};

template <class T>
struct is_pointer<T*> { // ポインタならtrueを返す
  static const bool value = true;
};

bool a = is_pointer<int>::value;  // bool a = false;
bool b = is_pointer<int*>::value; // bool b = true;
```

型を修飾する

- Tを受け取ってT*を返すメタ関数

```
template <class T>
struct add_pointer {
  typedef T* type;
};

add_pointer<int>::type p;
    // int* p;

add_pointer<add_pointer<int>::type>::type pp;
    // int** pp;
```

再帰テンプレート

- メタ関数がメタ関数自身を呼ぶことによって再帰によるループを表現する

```
template <class T, int N>
struct add_pointer {
  typedef typename add_pointer<T*, N-1>::type type;
};

template <class T>
struct add_pointer<T, 0> { // 再帰の終了条件
  typedef T type;
};

add_pointer<int, 5> p; // int***** p;
```

応用例1 : コンパイル時if文(型の選択)

テンプレートパラメータで条件式をbool値で受け取って
パラメータがtrueの場合の型、falseの場合の型を選択する

```
template <bool Cond, class Then, class Else>
struct if_c;

template <class Then, class Else>
struct if_c<true, Then, Else> {
  typedef Then type;
};

template <class Then, class Else>
struct if_c<false, Then, Else> {
  typedef Else type;
};

if_c<true, int, char>::type
→ int

if_c<false, int, char>::type
→ char
```

応用例2：
コンテナ/配列からイテレータ/ポインタの型を取り出す

```
template <class Range>
struct range_iterator {          // 配列以外だったらRange::iterator型を返す
  typedef typename Range::iterator type;
};

template <class T, int N>
struct range_iterator<T[N]> { // 配列だったらT*型を返す
  typedef T* type;
};

template <class Range>
void foo(Range& r)
{
  typedef typename range_iterator<Range>::type Iterator;
}

vector<int> v;
int ar[3];

foo(v);  // Iteratorの型はvector<int>::iteratorになる
foo(ar); // Iteratorの型はint*になる
```

応用例3:
型のシグニチャから部分的に型を抜き出す

```
template <class Signature>
struct argument_of;

template <class R, class Arg>
struct argument_of<R(Arg)> { // 型がR(Arg)の形になってたら
  typedef R   result_type;   // 戻り値の型を取り出す
  typedef Arg argument_type; // 引数の型を取り出す
};

typedef argument_of<int(double)>::result_type   result;   // int
typedef argument_of<int(double)>::argument_type argument; // double
```

boost::result_ofで関数オブジェクトの戻り値の型を取得するときに使える

チューリング完全

特殊化によって条件分岐を表現し、
再帰テンプレートによってループを表現できる

これらのことから、C++テンプレートは
ほぼ(※)チューリング完全だと言われてるみたい。
つまり、コンパイル時に全てのアルゴリズムを解くことができる。

※再帰的に入れ子にされたテンプレートの
　インスタンス化は17回までは保証されてる。

『C++ Templates are Turing Complete』
http://ubiety.uwaterloo.ca/~tveldhui/papers/2003/turing.pdf