# C++テンプレートメタプログラミングについて

ソフトウエア研究会

2009/7/7

池田公平

# C++テンプレートができた理由

- 1980年代の初期のC++にはテンプレートはなかった。
- smalltalkのような汎用のコンテナクラスをC++で実現するために、さまざまな手法が試された。
- クラスの継承、#defineマクロで似たような機能は実現できたが、使いやすいものは作れなかった。
- そこで、C++にテンプレートというマクロに似た新しい機能を追加することにした。

# C++テンプレートの特徴

- C++の構文に馴染んだ文法
- #defineのようなプリプロセッサによる処理ではなく、コンパイラが型を推測しながら実装を決定できる仕組みを取り入れた
- 関数、クラス、クラス関数それぞれにテンプレートを使用できる
- テンプレートの特化機能により、型特有の細かな処理を書くことが可能
- テンプレートの引数にテンプレートを用いることができ、再帰的な定義が可能

# #defineの限界1

- #defineは文字列置換を行う強力なマクロ機能を有するが、C++においてはいくつかの欠点をもつ。

- 欠点1 ー　マクロなので展開された後のコードを考慮しなければならない。

例： 大きいほうの値を返すマクロ
```
#define max(a,b) ((a > b) ? a : b)

int x = 1;
int y = 2;
printf (” max value is %d”, max(x++,y++));
```

左の例では、“３”が出力され、xは3に、yは4になる。
なぜならば、#define max はマクロなので、printf文は
以下のように展開される。

```
printf( “max value is %d”, ((x++ > y++) ? x++ : y++));
```

- 欠点２ー　文字列置換してしまうので、ネームスペースの領域を超えてすべての関数名、変数名、クラス名に影響を与えてしまう。

例: たとえば、printをTRACEに置換するような#defineが定義されているヘッダをインクルードした場合、ネームスペースにかかわらず、“print”という名の関数、クラス、変数を使ったクラスばエラーになる。
(実際に、MFCクラスライブラリでは、SentMessage, PostMessageなどよくつかわれる単語が#defineされているため、非常に使いづらい)
```
#define print TRACE
```

# #defineの限界2

- 欠点３ ー　型のチェックは行われないので、C++の特徴である厳密な型チェック機能を使えない。

```
例: 欠点1の#define maxマクロで、const char*は比較できない。
#define max(a,b) ((a > b) ? a: b)
...
printf(”%s”, max(”min”,”max”));  //<- どうなるか不定
```

- 欠点４　ー　引数を文字として扱うので、スペースを含む型などの展開に限界がある。

```
例: 与えられた型をメンバに持つクラスを作るマクロ
#define ImmClass(type)  class ImmClass##_type { public: type m_value; }
...
ImmClass(int) a;             // OK   class ImmClass_int; が定義される
ImmClass(unsigned char*) c; // NG!  class ImmClass_unsigned char*; は文法エラー
```

# C++テンプレートの特殊機能

- C++テンプレートは、単なるマクロの拡張にとどまらず、言語機能として将来性を見据えて様々な拡張が施された。
- この、様々な拡張によって、当初は想定してなかったテンプレートメタプログラミングという新しい言語スタイルが生み出された。
- C++　のテンプレートメタプログラミングは、独立したコンピューター言語として認定されている。

# C++テンプレートの特殊機能

- 機能１　パラメータの特殊化

```
Template < typename T>
void member (T& val) {
    printf(“%d”, val);
};
Template <>
Void member(char* val) {
  printf(“%s”, val);
}
```

# C++テンプレートの特殊機能

```
template <typename T>
struct  isCharPointer {
          enum { value = 0 }
};
tempalte  <>
struct isCharPointer <char*> {
          enum {value = 1}
};

printf( “%d”, isCharPointer<int >::value );   /// “0”と表示される。
printf( “%d”, isCharPointer <char*>::value); // “1”と表示される。
```

# C++テンプレートの特殊機能

- 特殊機能２　パラメータとして整数を指定できる

```
tempalte <int N>
void* alloc() {
  static char buf[N];
   reutrn &buf;
};

template <typename T, int N>
int countOf(T(&)[N] ) {
  return N;
}
char buf[256];
printf( “%d”, countOf(buf)); // 256と表示される。
memset(buf, 0, 256); // よろしくない
memset(buf,0,countOf(buf)); // bufのサイズが変わっても安全
```

```
typedef char A10[10];
void hoge(A10& a) {
}
void hoge(char (&a)[10])
```