Template <Programming>

― テンプレートとは何か ―

高橋 晶(Akira Takahashi)
id:faith_and_brave / @cpp_akira

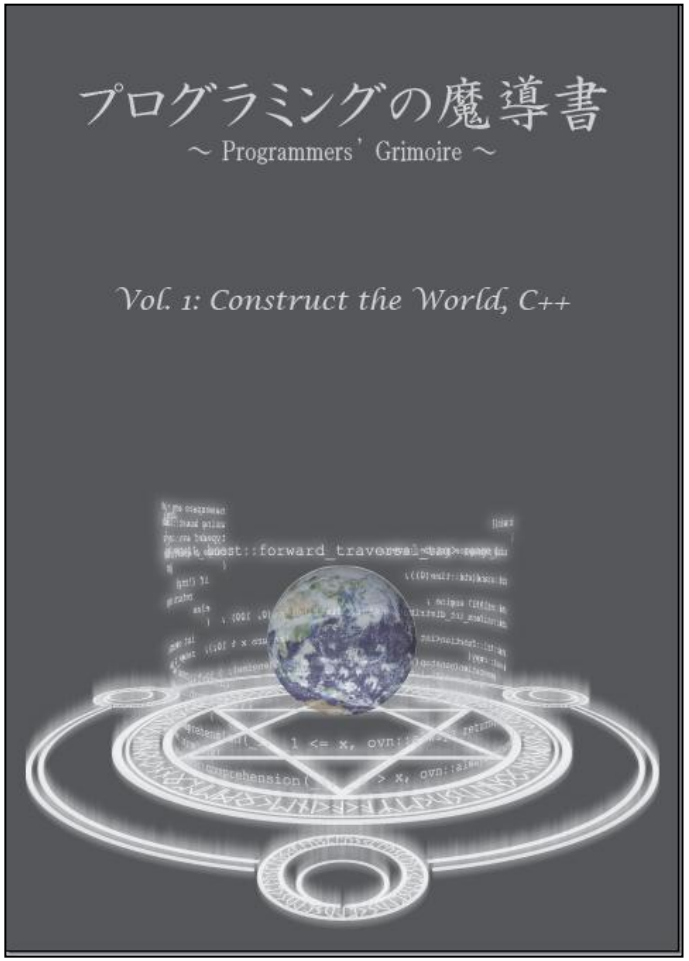
2012/11/23 Effective C++読書会 vol.11 大阪～第7章特別編～

自己紹介

- Boost.勉強会 東京の主催者
- boostjp/cpprefjpサイトを作ってます
- 著書
  - 『C++テンプレートテクニック』
  - 『プログラミングの魔導書 Vol.1 ～Construct the World, C++～』

自己紹介


C++ Template Technique
簡潔で再利用しやすいコードのためのC++活用術
C++テンプレートテクニック
επιστημη+高橋 晶著
SoftBank Creative


プログラミングの魔導書
～ Programmers' Grimoire ～
Vol. 1: Construct the World, C++

本日のお題

1. テンプレート入門
2. どんな場面で使用するか
3. テンプレート技法
4. 未来

Chapter 01

テンプレート入門

Introduction to Templates

テンプレートとは何か

- 日本語では「雛形」
- 型のパラメータ化によってコードの共通化を行う
- 基本的な目的:
  - 任意のデータ型のオブジェクトを格納できるコンテナ
  - データ型に依存しないアルゴリズム

任意の型のオブジェクトを格納できるコンテナ

```
template <class T>
class List {
  T* data_;
  size_t size_;
public:
  void add(const T& x) {
    T* tmp = new T[size_ + 1];
    for (size_t i = 0; i < size_; ++i) { tmp[i] = data_[i]; }
    delete[] data_;

    data_ = tmp;
    data_[size_++] = x;
  }

  T& operator[](size_t i) { return data_[i]; }
  size_t size() const { return size_; }
};
```

任意の型のオブジェクトを格納できるコンテナ

```
List<int> ls;
ls.add(3);
ls.add(1);
ls.add(4);

for (size_t i = 0; i < ls.size(); ++i) {
  cout << ls[i] << endl;
}
```

```
List<string> ls;
ls.add("abc");
ls.add("hello");
ls.add("goodbye");

for (size_t i = 0; i < ls.size(); ++i) {
  cout << ls[i] << endl;
}
```

内部の型が異なるだけで
使い方は同じ

データ型に依存しないアルゴリズム

以下のmin()関数は、operator<持つあらゆる型に適用可能な関数

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

```
int    x = min(1, 2);      // Tはintに置き換えられる
double d = min(1.0, 2.0); // Tはdoubleに置き換えられる
char   c = min('1', '1'); // Tはcharに置き換えられる
```

クラステンプレート

クラス内で扱う型をパラメータ化する

```
template <class T>
class X {
  // T型が内部で持っている型を取得する
  using value_type = typename T::value_type;

  void f(const T& x) {
    x.member(); // メンバ変数／関数を使用する
    T temp = T(); // パラメータ型のオブジェクトを構築する
  }
};
```

```
X<Y> object;
object.f(y_object);
```

※T()というコンストラクタ呼び出し構文を許可するために、intやcharのような組み込み型にもコンストラクタ構文が使えるようになっている

関数テンプレート

関数内で扱う型をパラメータ化する

```
template <class T>
void f(const T& x) {
  …同じく型Tのメンバや型を使用できる…
}
```

```
X x;
f(x); // オブジェクトxの型XでTが置き換えられる

Y y;
f<Y>(y); // 明示的な型指定も可能
```

クラステンプレートと違い、関数テンプレートの型は引数として
渡されたオブジェクトから推論される

可変引数テンプレート

任意個数のテンプレートパラメータを扱う機能(C++11)

```
void printValues() {}

template <class X, class... XS>
void printValues(X x, XS... xs) { // 先頭とそれ以外に分ける
  std::cout << x << std::endl;    // 先頭要素を出力
  printValues(xs...);             // 残りを出力
}
```

```
printValues(1, 'a', "hello");
```

テンプレートの特殊化

特殊化には、完全特殊化と部分特殊化の2種類がある

1. 完全特殊化は、汎用的なものとは別に、特定の型に対する特殊バージョンを定義する
2. 部分特殊化は、特定のパターンに一致する型に対する特殊バージョンを定義する

```
template <class T> // 汎用的な処理
struct X { void f() {} };

template <>
struct X<int> { void f() {} }; // intに対する完全特殊化

template <class T>
struct X<T*> { void f() {} }; // ポインタ型に対する部分特殊化
```

完全特殊化はクラステンプレートと関数テンプレートで使用でき、
部分特殊化はクラステンプレートで使用できる

デフォルトテンプレート引数

クラステンプレートと関数テンプレートには、それぞれデフォルトの型を指定できる(関数テンプレートはC++11から)

```
template <class T = void>
class X;
```

```
template <class Option = nothing>
void f();
```

非型テンプレートパラメータ

テンプレートパラメータには、型だけでなく値も指定できる。

指定できるのは、整数型(int, char, enum, ...)、および外部リンケージを持つオブジェクトへのポインタと参照。

```
template <size_t N>
class X {
  static constexpr int size = N;
  int ar[size];
}
```

```
X<3> x;
for (size_t i = 0; i < x.size; ++i)
  x.ar[i] ⋯;
```

エイリアステンプレート

テンプレートを使用して型に別名を付ける(C++11から)

typedef templateとも言える

```
// アロケータだけ先に設定しておく
template <class T>
using vec = std::vector<T, stack_allocator<T>>;
```

```
vec<int> v;
```

※特殊化はできない

Chapter 02

どんな場面で使用するか

Use Situations

テンプレートをどんな場面で使用するか

ここでは、テンプレートを使用するにあたっての指針を示す

テンプレートの適用場面は、大きく3つに分類できる:

1. 同じ意味論とインタフェースを持つ値を一様に扱う
2. クラス・関数の内部操作をコンパイル時に切り替える
3. 実行時エラーを阻止する

同じ意味論とインタフェース持つ値を一様に扱いたい場面で使用する

同じ意味を持つ操作に共通インタフェースを持たせて、同じように扱う

```
class Title { void update(); void draw(); };
class Quest { void update(); void draw(); };
```

```
template <class Scene>
void updateGame(Scene& scene) {
  scene.update();
  scene.draw();
}
```

クラス・関数の内部操作をコンパイル時に切り替える

内部のアルゴリズムが違う以外は同じインタフェースで使える、という場合に使用する。内部戦略のパラメータ化。

```
struct NonePolicy    { static void print(int x) {} };
struct PrintPolicy   { static void print(int x) { /*標準出力へ*/ } };
struct LoggingPolicy { static void print(int x) { /*ファイルへ*/ } };

template <class OutputPolicy>
class X {
  int value_ = 0;
public:
  void f(int add) {
    value_ += add;
    OutputPolicy::print(value);
  }
};
```

実行時エラーを阻止する

場面によってやり方は様々だが、static assertのようなものと組み合わせることで、コンパイル時に正当性チェックを行う。
ここでは、strong typedefのためにテンプレートを使用する例を示す。

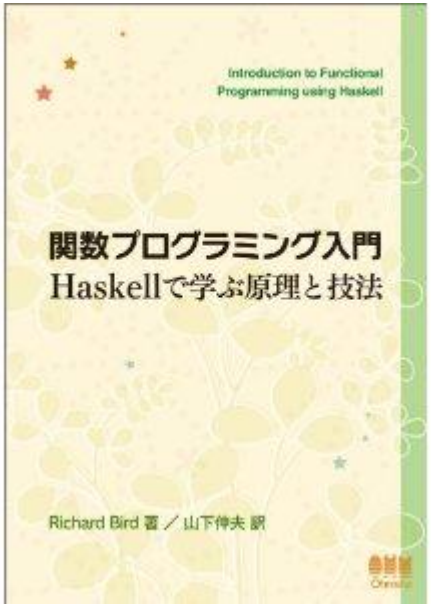
```
// タグ付き浮動小数点数型(タグが異なれば型が異なるので変換できなくなる)
template <class FloatingPoint, class Tag>
class tagged_real {
  FloatingPoint value_;
public:
  tagged_real(FloatingPoint value = 0) : value_(value) {}
  FloatingPoint& get() { return value_; }
};
```

```
struct degree {}; // 空クラス
struct radian {};
tagged_real<float, degree> deg(90.0f);
tagged_real<float, radian> rad = deg; // コンパイルエラー！型が違う
```

https://github.com/faithandbrave/Shand/blob/master/libs/strong_typedef/tagged_real_example.cpp

いつでもテンプレート

- テンプレートは小難しい機能でライブラリを書く人だけ知っていればいい、というものではない
- テンプレートはとても広く一般的に使うもので、関数やクラスを書く際にいつでも設計の選択肢に入る
- 入門書の1章で紹介される程度には基本的:




```
template <class T>
T square(T x) { return x * x; }
```

2乗を計算する基本な関数。テンプレートにしておけば、intでもdoubleでも、*演算子を持つあらゆるクラスで使用できる

Chapter 03

テンプレート技法

Template Techniques

テンプレート技法について

- 限られたルールの中で限界を超えるための必殺技がいくつかある
- 紹介しきれないので、キーワードと概要の紹介にとどめる
- 詳細は、以下の文献を参照:
  - 『C++テンプレート完全ガイド(David Vandevoode)』
  - 『C++テンプレートメタプログラミング(Dave Abrahams』
  - 『C++テンプレートテクニック(επιστημη、高橋晶)』
  - More C++ Idioms
  - Boostのソースコード
  - id:Cryolite, id:DigitalGhost, id:iorateのブログ

テンプレート技法各種

- SFINAE(Substitution Failure Is Not An Error)
  - テンプレートの置き換え失敗を、エラーではなくオーバーロードの候補から外す言語機能。コンパイル時に任意の条件でオーバーロードするのに使用する。「スフィネェ」と発音する。
- テンプレートメタプログラミング
  - テンプレートの各種機能を駆使して、コンパイル時のC++プログラムに関するメタ情報を操作する技法
- タグディスパッチ
  - タグと呼ばれる空クラスを駆使するオーバーロード技法
- 型消去(Type Erasure)
  - 一時的に型情報を消去してあらゆる型を包含し、元の型に関する情報を保持しておいて安全に復元して操作する技法

テンプレート技法各種

- 式テンプレート(Expression Template)
  - 式をその場では評価せずに式情報をテンプレートに保持しておき、演算が本当に必要になったときに評価する最適化技法
- CRTP(Curiously Recursive Template Pattern)
  - 自身の型を基本クラスのテンプレート引数として渡し、基本クラス内で派生クラスの情報を使用する技法
- ポリシーに基づく設計(Policy-based Design)
  - 単なる型ではなく、処理をテンプレートとして渡し、コンパイル時に戦略を切り替える設計手法

Chapter 05

未来

The Future

C++11の次

- C++11では、テンプレート関係で以下の機能が追加された:
  - エイリアステンプレート
  - 可変引数テンプレート
  - 関数テンプレートのデフォルトテンプレート引数
- 次期バージョンC++1yは、まだ何も決まっていない状態ではあるが、考えられている機能をいくつか紹介する

非型テンプレートパラメータの制限緩和

constexprオブジェクトをテンプレートパラメータで受け取れるようにしよう、というもの

```
struct C {
  constexpr C(int v) : v(v) { }
  int v;
};

template <C c>
struct X {
  int ar[c.v];
};
```

N3413 - Allowing arbitrary literal types for non-type template parameters
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3413.html

コンパイル時if文

static if宣言／文。テンプレートの特殊化やオーバーロードではなくif文で分岐する。

```
template <int n>
struct factorial {
  static if (n <= 1) {
    static constexpr value = 1;
  }
  else {
    static constexpr value = factorial<n - 1>::value * n;
  }
};
```

N3329 Proposal: static if declaration
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3329.pdf

2つしかなかった！

- せっかくなので関連情報。
- コンセプトはおそらくC++1yではなくC++22。Clangで実験的に実装されている。
- ほか、関数の戻り値型の推論強化など：

```
auto f(); // 宣言時点では、戻り値型は不明
auto f() { return 42; } // 戻り値型はint
```

N3386 Return type deduction for normal functions
http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2012/n3386.html
GCC 4.8から-std=c++1yオプションを付けると使用できる

おわりに

- テンプレートは、共通コードをまとめるのに使用でき、重複コードを限りなく減らせます
- テンプレートの特殊化やオーバーロード技法を使用することにより、特定の条件で最適化をかけられます
- コンパイル時にプログラムを検証することで、実行をより安全に行えるようにできます
- テンプレートは多くの場面で適用できます。特定の型でのみ振る舞えればいいのか、汎用的なのかを常に考え、設計の選択肢を広げましょう。