1章
型推論

C++98 に型の推論規則は一種類しかありませんでした[1]。関数テンプレートです。C++11 ではこの規則に若干の変更を加え、auto による推論規則、decltype による推論規則の 2 つを追加しました。さらに C++14 では auto と decltype の適用可能場面を拡張しました。型を推論できる場面が増えれば、暴虐にも等しい、自明もしくは冗長な型を 1 文字ずつ正確に入力する労苦からプログラマを解放できます。ソースコードの 1 箇所で型を変更すれば、型推論のおかげで他の箇所にも自動的に伝搬され、C++ ソフトウェアの適用性も高まります。しかしながら、ソースコードを読み下す際の難易度が上がってしまうという問題もあります。これはコンパイラが推論した型が期待されるほど明白ではない場合があるためです。

型がどのように推論されるかを完全に把握しなければ、現代の C++ での効率的なプログラミングは不可能です。型が推論される場面は多岐に渡り、関数テンプレート呼び出しや、auto を記述した箇所はほぼそうです。decltype の式もそうですし、C++14 には謎めいた暗号のような decltype(auto) もあります。

本章では型推論を解説します。すべての C++ 開発者にとって必須の知識です。テンプレートの型がどのように推論されるか、auto はこれをどのように利用しているか、また decltype はどのように処理されるかについてです。さらに、コンパイラを利用した、推論結果を可視化する方法についても解説します。この方法を用いれば、コンパイラが期待通りの型を推論したことを確認できます。

項目 1：テンプレートの型推論を理解する

システムが複雑で、ユーザがその内部動作に関する知識を持ち合わせていなくても、システムの動作には満足できます。この事実はシステム設計の多くを物語っており、この点では C++ の型推論は素晴らしい成果を挙げています。数百万ものプログラマがテンプレート関数に実引数を渡し、その結果に充分満足しています。テンプレート関数の使用する型がどのように推論されるかを、ほと

[1] 訳者注：導出、推定、推測などの訳語がありますが、本訳では型の「推論」とします。

んどのプログラマが満足に説明できないにも関わらずです。

読者も満足に説明できないならば、著者から良いニュースと悪いニュースをお知らせしましょう。良いニュースとは、テンプレートの型推論が、現代の C++ の最も面白い機能である auto の基礎になっているという点です。C++98 のテンプレートの型推論に満足していれば、C++11 の auto の型推論についても満足いくでしょう。悪いニュースとは、テンプレートの型推論規則を auto に適用すると、テンプレートに比べ分かりにくい結果になることがあるという点です。このため、auto の基礎となるテンプレートの型推論を隅々まで把握しておくことが重要です。本項目では押さえておくべき重要な点を解説します。

関数テンプレートを疑似コードでごく簡潔に表現すれば、次のようになります。

```
template<typename T>
void f(ParamType param);
```

呼び出し箇所は次のようになります。

```
f(expr);                    // call f with some expression
                                                f に式を与え呼び出す
```

上記コードのコンパイル時に、コンパイラは *expr* から 2 つの型を推論します。1 つは T、もう 1 つは *ParamType* です。この 2 つは一致しないことが多く、*ParamType* は多くの場合修飾されています。すなわち、const 修飾子や参照修飾子が付加されています。例として、テンプレートを次のように宣言した場合を考えてみましょう。

```
template<typename T>
void f(const T& param);     // ParamType is const T&
                                              ParamType は const T&
```

また、呼び出し箇所は次のようなものだとします。

```
int x = 0;

f(x);                       // call f with an int
                                              f に int を与え呼び出す
```

上例で T は int と推論されますが、*ParamType* は const int& と推論されます。

T に推論したのと同じ型が、関数の実引数の型にも推論されると期待するのはごく自然なことでしょう。すなわち、*expr* の型が T となる推論結果です。上例でも x は int であり、T は int と推論されます。しかし、常にこのような結果になる訳ではありません。T に推論する型は *expr* の型だけから決定される訳ではなく、*ParamType* からも影響を受けます。次に挙げる 3 通りの場合があります。

- ケース 1：*ParamType* が参照もしくはポインタだが、ユニヴァーサル参照ではない（ユニヴァーサル参照については**項目 24** で述べる。ここでは左辺値参照とも右辺値参照とも違うものが存在するとだけ認識しておけばよい）[2]。
- ケース 2：*ParamType* がユニヴァーサル参照である。
- ケース 3：*ParamType* がポインタでも参照でもない。

検証すべき型推論には上記 3 つの場合があります。それぞれの場合につき、次に挙げるテンプレートの一般形とその呼び出しを考えて行きましょう。

```
template<typename T>
void f(ParamType param);

f(expr);                // deduce T and ParamType from expr
                                    expr から T と ParamType を推論
```

ケース 1：*ParamType* が参照もしくはポインタだが、ユニヴァーサル参照ではない

ParamType が参照型またはポインタ型で、かつユニヴァーサル参照ではない場合が最も単純です。この場合、型は次のように推論されます。

1. *expr* が参照型ならば、参照性（参照動作部分）を無視する。
2. *expr* の型を *ParamType* とパターンマッチングし、T を決定する。

例えば、次のテンプレートがあるとします。

```
template<typename T>
void f(T& param);       // param is a reference        param は参照
```

また次の変数宣言もあるとします。

```
int x = 27;             // x is an int
const int cx = x;       // cx is a const int
const int& rx = x;      // rx is a reference to x as a const int
                            x は int、cx は const int、rx は const int と
                            しての x の参照
```

param と T に推論される型は次のようになります。

[2] 訳者注：universal reference、ユニヴァーサルレファレンス。国内では決まった訳語はないようで、原語表記も多く見受けられますが、本訳ではこのように表記します。その後の C++ 標準仕様で、forwarding reference と名付けられました。

```
f(x);                   // T is int, param's type is int&
                                    T は int、param の型は int&
f(cx);                  // T is const int,
                        // param's type is const int&
                                T は const int、param の型は const int&
f(rx);                  // T is const int,
                        // param's type is const int&
                                T は const int、param の型は const int&
```

上例で 2 番目と 3 番目の呼び出しに注目してください。cx と rx には const を指定しており、T は const int と推論されています。その結果、仮引数の型は const int& となります。この点は呼び出し側にとって大きな意味を持ちます。参照仮引数に const オブジェクトを渡せば、オブジェクトが変更されないことを期待できるのです。実際 const 参照仮引数となり、このため T& を仮引数にとるテンプレートへ const オブジェクトを渡しても安全であり、またオブジェクトの const 性が T に推論される型の一部となります。

3 番目の呼び出しでは、rx が参照型であるにも関わらず、T は参照型と推論されない点に注目してください。これは、型の推論では rx が備える参照性が無視されるためです。

上例はすべて左辺値参照仮引数ですが、型推論の動作は右辺値参照仮引数の場合でもまったく同じです。もちろん、右辺値参照仮引数へ渡せるのは右辺値実引数だけですが、この点は型推論とは関係ありません。

正誤表より補足

（訂正）上段落はまるごと削除します。先に挙げたテンプレート f の仮引数の型を右辺値参照に変更すると（すなわち、T&&）、ユニヴァーサル参照になり、後述するケース 2 の規則が適用されます。

仮に f の仮引数を T& から const T& へ変更しても、状況はやや変化しますが、それほど大きな違いはありません。cx と rx の const 性は維持されます。しかし、param は const 参照であると想定しているため、T の一部として const を推論する必要がなくなります。

```
template<typename T>
void f(const T& param);  // param is now a ref-to-const
                                            param は const 参照
int x = 27;              // as before
const int cx = x;        // as before              先の例と変わらず
const int& rx = x;       // as before

f(x);                    // T is int, param's type is const int&
                                    T は int、param の型は const int&
f(cx);                   // T is int, param's type is const int&
```

```
f(rx);                  // T is int, param's type is const int&
```

先の例と同様に、型推論では rx の参照性は無視されます。

仮に param が参照ではなくポインタだったとしても（または const を指すポインタ）、基本的には同様に推論されます。

```
template<typename T>
void f(T* param);       // param is now a pointer
                                                  param は pointer
int x = 27;             // as before
const int *px = &x;     // px is a ptr to x as a const int
                           先の例と変わらず。px は const int としての x
                           を指す
f(&x);                  // T is int, param's type is int*
                                          T は int、param の型は int*
f(px);                  // T is const int,
                        // param's type is const int*
                                T は const int、param の型は const int*
```

C++ の型推論規則は、参照やポインタの仮引数に対してはごく自然に動作するため、改めて言葉で説明されても退屈で居眠りをしてしまったかも知れません。すべて自明で分かりきったことです！型推論システムは期待される通りに動作します。

ケース 2：*ParamType* がユニヴァーサル参照である

ユニヴァーサル参照を仮引数にとるテンプレートの場合はずっと分かりにくくなります。この種の仮引数は右辺値参照のように宣言されますが（T を仮引数にとる関数テンプレートでは、ユニヴァーサル参照に宣言する型が T&&）、左辺値の実引数が渡された場合の動作が変化します。詳細は**項目 24** で解説し、ここでは概要を簡単に述べます。

- *expr* が左辺値ならば、T も *ParamType* も左辺値参照と推論される。これは 2 つの意味で特殊である。まず、テンプレートの型推論で、T を参照として推論するのはこの場合だけである。もう 1 つは、*ParamType* の宣言には右辺値参照という形態をとりながら、推論される型は左辺値参照となる点である。
- *expr* が右辺値の場合は、「通常の」規則が適用される（ケース 1）。

例を挙げます。

```
template<typename T>
void f(T&& param);      // param is now a universal reference
                                                 param はユニヴァーサル参照
```

```
int x = 27;             // as before
const int cx = x;       // as before              先の例と変わらず
const int& rx = x;      // as before

f(x);                   // x is lvalue, so T is int&,
                        // param's type is also int&
                          x は左辺値、よって T は int&、param の型も int&
f(cx);                  // cx is lvalue, so T is const int&,
                        // param's type is also const int&
                           cx は左辺値、よって T は const int& param の
                           型も const int&
f(rx);                  // rx is lvalue, so T is const int&,
                        // param's type is also const int&
                           rx は左辺値、よって T は const int& param の
                           型も const int&
f(27);                  // 27 is rvalue, so T is int,
                        // param's type is therefore int&&
                           27 は右辺値、よって T は int、ゆえに param の
                           型は int&&
```

上例がなぜこのように動作するかについては、**項目 24** で述べます。重要なのは、ユニヴァーサル参照の仮引数に対する型推論規則は、左辺値参照や右辺値参照の仮引数の場合とは異なるという点です。特に、型推論が左辺値実引数と右辺値実引数を区別する点は重要であり、ユニヴァーサル参照に限った特殊な規則です。

ケース 3：*ParamType* がポインタでも参照でもない

ParamType がポインタでも参照でもなければ、値渡しとなります。

```
template<typename T>
void f(T param);        // param is now passed by value
                                                  param は値渡しされる
```

この場合の `param` は渡したもののコピー、すなわちまったく別のオブジェクトとなります。この、`param` が新規オブジェクトになるという点は、*expr* から `T` を推論する動作に大きく影響します。

1. これまでと同様に、*expr* の型が参照であれば、参照性（参照動作部分）は無視される。
2. 参照性を無視した *expr* が `const` であれば、これも無視する。`volatile` であれば、同様にこれも無視する（`volatile` オブジェクトは滅多に使用されない。使用されるのは一般にデバイスドライバを実装する場合に限られる。詳細は**項目 40** を参照）。

実際には次のようになります。

```
int x = 27;           // as before                    先の例と同様
const int cx = x;     // as before
const int& rx = x;    // as before

f(x);                 // T's and param's types are both int
                                          T と param の型はいずれも int
f(cx);                // T's and param's types are again both int
                                       T と param の型はいずれもやはり int
f(rx);                // T's and param's types are still both int
                                       T と param の型はいずれも変わらず int
```

cx および rx の値が const の場合でも、param は const とならない点に注意してください。param は cx や rx の**コピー**、すなわち cx と rx とはまったく別のオブジェクトですから、納得がいくでしょう。cx と rx が変更不可である点は param には影響しません。これが *expr* の const 性（および volatile 性、もしあれば）が、param の型を推論する際に無視される理由です。*expr* が変更不可であっても、このコピーの変更を禁止することにはならないのです。

const（および volatile）が値渡しの場合にのみ無視される点は重要ですので、よく覚えておいてください。これまで見てきたように、仮引数が const を指すポインタ／参照の場合は、*expr* の const 性は型を推論しても失われません。では次にこんな例を考えてみましょう。*expr* が const オブジェクトを指す const なポインタであり、*expr* を param に値を渡した場合です。

```
template<typename T>
void f(T param);          // param is still passed by value
                                          param は変わらず値渡しされる
const char* const ptr =   // ptr is const pointer to const object
  "Fun with pointers";        ptr は const オブジェクトを指す const なポイ
                              ンタ
f(ptr);                   // pass arg of type const char * const
                                       const char * const 型の実引数を渡す
```

上例で、アステリスクの右にある const は ptr が const であることを意味します。ptr は他のアドレスを指すことも、ヌル（ナル）ポインタになることもありません（アステリスクの左にある const は ptr が指すもの、上例では文字列、が const であることを表し、文字列の変更が禁止される）。ptr を f に渡すと、ptr を構成する全ビットが param にコピーされます。**ポインタ自身（ptr）を値渡しする動作です**。仮引数を値渡しする際の型推論規則により、ptr の const 性は無視され、param に推論される型は const char* となります。すなわち、const な文字列を指す変更可能なポインタです。型を推論しても、ptr が指すオブジェクトの const 性は維持されますが、ptr をコピーし新たなポインタ param を作成する時点で、ptr 自身の const 性は失われます。

配列実引数

ここまででテンプレートの型推論についてはそのほとんどに対応できますが、あと少しだけ覚えておくことがあります。配列型とポインタ型は交換可能と言われることもありますが、両者は異なる型であるという点です。この目眩ましのような状態の元凶は、配列は、多くの場面で、その先頭要素を指すポインタに**成り下がる**（decay）という動作です。この動作から、次のようなコードがコンパイル可能になります。

```
const char name[] = "J. P. Briggs";  // name's type is
                                     // const char[13]
                                          name の型は const char[13]
const char * ptrToName = name;       // array decays to pointer
                                          配列がポインタに成り下がる
```

上例で、const char* のポインタ ptrToName は const char[13] である name により初期化されます。const char* と const char[13] は同じ型ではありませんが、配列からポインタへ変換する規則により、コンパイル可能なのです。

では、仮引数を値渡しするテンプレートに配列を渡すのはどうでしょうか？ どんなことが起こると思いますか？

```
template<typename T>
void f(T param);      // template with by-value parameter
                                        仮引数を値渡しするテンプレート
f(name);              // what types are deduced for T and param?
                                  T と param にはどんな型が推論されるか？
```

まず、関数の仮引数として配列なぞはあり得ないという事実から確認しましょう。もちろん、文法的には問題ありません。

```
void myFunc(int param[]);
```

しかし、配列として宣言してもポインタの宣言として扱われます。つまり上例の myFunc は次のようにも宣言可能です。

```
void myFunc(int* param);          // same function as above
                                                  上例と同じ関数
```

仮引数の配列とポインタの等価性は、C++ の土台である C 言語を根とし、成長した枝葉のようなもので、ここから配列とポインタは同じものであるという幻想が醸し出されています。

配列仮引数の宣言は、ポインタ仮引数として扱われるため、テンプレート関数へ値渡しされた配列の型はポインタ型と推論されます。このことは、テンプレート f を呼び出すと、その型仮引数 T は const char* と推論されることを意味します。

```
f(name);          // name is array, but T deduced as const char*
                                  name は配列だが、T は const char* と推論される
```

ここで変化球の登場です。関数は仮引数を真の配列とは宣言できないけれど、配列の**参照としては宣言できる**のです！ 次のようにテンプレート f の実引数を参照に変更してみます。

```
template<typename T>
void f(T& param);      // template with by-reference parameter
                                        参照渡しの仮引数を持つテンプレート
```

そして配列を渡してみます。

```
f(name);               // pass array to f              f へ配列を渡す
```

すると、T に推論される型は配列の型になります！ この型は配列の要素数も含んでおり、上例では const char [13] です。また、f の仮引数の型は（配列の参照）、const char (&)[13] となります。そう、この構文には毒もあるのです。この点を押さえておくと、こんなことまで意識するごく一部の人々と議論する際に役立つこともあるでしょう。

面白いことに、配列の参照を宣言できるようになると、配列の要素数を推論するテンプレートを記述できます。

```
// return size of an array as a compile-time constant. (The
// array parameter has no name, because we care only about
// the number of elements it contains.)
    配列の要素数をコンパイル時定数として返す（要素数のみを考慮するため、仮引数
    の配列に名前はない）
template<typename T, std::size_t N>                 // see info
constexpr std::size_t arraySize(T (&)[N]) noexcept  // below on
{                                                   // constexpr
  return N;                                         // and
}                                                   // noexcept
                                                constexpr と noexcept
                                                については下記を参照
```

項目 15 でも述べますが、上例の関数を constexpr と宣言することで、その戻り値をコンパイル時に使用できます。これにより、例えば、配列の宣言時に要素数を明示しなくとも、波括弧を用いた初期化子から要素数を算出できるようになります。

```
int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };       // keyVals has
                                                  // 7 elements
                                                     keyVals の要素
                                                     数は 7
int mappedVals[arraySize(keyVals)];               // so does
                                                  // mappedVals
                                                  mappedVals も同じ
```

もちろん、現代の C++ 開発者ならば、組み込み配列よりも std::array の方が当然好みでしょう。

```
std::array<int, arraySize(keyVals)> mappedVals;  // mappedVals'
                                                 // size is 7
                                                     mappedVals の
                                                     要素数は 7
```

arraySize は noexcept と宣言しているため、コンパイラにはより良いコードを生成する機会が生まれます。詳細は**項目 14** で述べます。

関数実引数

C++ でポインタに成り下がるのは配列だけではありません。関数型も関数ポインタに成り下がり、先に述べた配列の型推論に関することはすべて、関数の型推論および関数ポインタへの成り下がりにも適用されます。

```
void someFunc(int, double);    // someFunc is a function;
                               // type is void(int, double)
                                someFunc は関数、型は void(int, double)
template<typename T>
void f1(T param);              // in f1, param passed by value
                                                  f1 の param は値渡し
template<typename T>
void f2(T& param);             // in f2, param passed by ref
                                                  f2 の param は参照渡し
f1(someFunc);                  // param deduced as ptr-to-func;
                               // type is void (*)(int, double)
                                  param は関数を指すポインタと推論、型は
                                  void (*)(int, double)
f2(someFunc);                  // param deduced as ref-to-func;
                               // type is void (&)(int, double)
                                          param は関数の参照と推論、型は
                                          void (&)(int, double)
```

現実に何らかの差異が生まれることはまずありませんが、配列がポインタに成り下がる点を覚えるならば、関数がポインタに成り下がることも知っておくと良いでしょう。

さて、ここまでで auto に関するテンプレートの型推論規則を学びました。初めに述べたように型推論規則はきわめて直観的です。ほとんどの場合は直観的に理解できます。特別な注意が必要なのは泥水をかき回すようなユニヴァーサル参照の型を推論する際の左辺値です。また、配列と関数がポインタに成り下がる規則がさらにかき回して濁りが増します。コンパイラの胸ぐらをつかみ、「お前が推論する型を吐け！」と怒鳴りつけたくなることもあるかも知れません。そんな時は**項目 4** を読んでください。コンパイラがそう動作するよう上手くおだてるための項目です。

重要ポイント

- テンプレートの型推論時には、参照実引数は参照とは扱われない。すなわち参照性は無視される。
- ユニヴァーサル参照仮引数の型を推論する際には、左辺値実引数を特別扱いする。
- 値渡しの仮引数の型を推論する際には、const および／または volatile 実引数は非 const、非 volatile と扱われる。
- 参照を初期化するものでなければ、配列または関数実引数はテンプレートの型推論時にポインタに成り下がる。

項目 2：auto の型推論を理解する

項目 1 のテンプレートの型推論を読んでいれば、auto の型推論についてもすでにほぼすべてを把握していることになります。やや奇異にも見える一点だけを除き、auto の型推論はテンプレートの型推論と**同一です**。しかしそんなことが可能でしょうか？ テンプレートの型推論ではテンプレート、関数、仮引数が対象ですが、auto ではそんなものは対象としません。

対象としないのはその通りですが、問題にはなりません。テンプレートの型推論と auto の型推論は直接的に対応しており、機械的に字面を置き換えるだけのことです。

項目 1 では、次の関数テンプレートの一般形を例にテンプレートの型推論を解説しました。

```
template<typename T>
void f(ParamType param);
```

また、呼び出し箇所の一般形も用いました。

```
f(expr);                    // call f with some expression
                                                f に式を与え呼び出す
```

f を呼び出す箇所では、コンパイラが *expr* から T と *ParamType* の型を推論します。

auto を用いた変数宣言では、auto がテンプレートの T の役割を、また変数の型指定子が *ParamType* の役割をそれぞれ果たします。言葉で説明するよりも例を挙げた方が分かり易いでしょう。

```
auto x = 27;
```

上例の x の型指定子は auto のみです。

```
const auto cx = x;
```

上例の型指定子は const auto です。

```
const auto& rx = x;
```

上例の型指定子は const auto& です。x、cx、rx の型を推論する際、コンパイラはそれぞれの宣言につきテンプレートが存在し、さらにそのテンプレートに初期化式を与えるコードも存在するものとして処理します。

```
template<typename T>                // conceptual template for
void func_for_x(T param);           // deducing x's type
                                        x の型を推論するための概念上の
                                        テンプレート
func_for_x(27);                     // conceptual call: param's
                                    // deduced type is x's type
                                        概念上の呼び出し：param に推論
                                        した型は x の型
template<typename T>                // conceptual template for
void func_for_cx(const T param);    // deducing cx's type
                                                        以下同様
func_for_cx(x);                     // conceptual call: param's
                                    // deduced type is cx's type

template<typename T>                // conceptual template for
void func_for_rx(const T& param);   // deducing rx's type

func_for_rx(x);                     // conceptual call: param's
                                    // deduced type is rx's type
```

すでに触れましたが、auto の型推論は一点を除き（後述）、テンプレートの型推論と同じです。

項目 1 では *ParamType* の性質、関数テンプレート一般の param の型指定子を基に、テンプレートの型推論を 3 種類に分け解説しました。auto を用いた変数宣言では、型指定子が *ParamType* に相当し、やはり 3 種類に分けられます。

- ケース 1：型指定子が参照もしくはポインタだが、ユニヴァーサル参照ではない

- ケース 2：型指定子がユニヴァーサル参照である
- ケース 3：型指定子がポインタでも参照でもない

ケース 1、ケース 3 についてはすでに例を挙げました。

```
auto x = 27;          // case 3 (x is neither ptr nor reference)
                                ケース 3（x はポインタでも参照でもない）
const auto cx = x;    // case 3 (cx isn't either)
                                      ケース 3（cx もどちらでもない）
const auto& rx = x;   // case 1 (rx is a non-universal ref.)
                              ケース 1（rx はユニヴァーサル参照ではない）
```

ケース 2 の場合も期待通り動作します。

```
auto&& uref1 = x;     // x is int and lvalue,
                      // so uref1's type is int&
                            x は int かつ左辺値のため、uref1 の型は int&
auto&& uref2 = cx;    // cx is const int and lvalue,
                      // so uref2's type is const int&
                          cx は const int かつ左辺値のため、uref2 の型は
                          const int&
auto&& uref3 = 27;    // 27 is int and rvalue,
                      // so uref3's type is int&&
                           27 は int かつ右辺値のため、uref3 の型は int&&
```

項目 1 の最後では、配列および関数名が非参照型指定子のポインタに成り下がることを述べましたが、同様のことが auto の型推論でも発生します。

```
const char name[] =           // name's type is const char[13]
  "R. N. Briggs";                       name の型は const char[13]

auto arr1 = name;             // arr1's type is const char*
                                        arr1 の型は const char*
auto& arr2 = name;            // arr2's type is
                              // const char (&)[13]
                                      arr2 の型は const char (&)[13]
void someFunc(int, double);   // someFunc is a function;
                              // type is void(int, double)
                              someFunc は関数、型は void(int, double)
auto func1 = someFunc;        // func1's type is
                              // void (*)(int, double)
                                  func1 の型は void (*)(int, double)
auto& func2 = someFunc;       // func2's type is
                              // void (&)(int, double)
                                  func2 の型は void (&)(int, double)
```

上例が示す通り、auto の型推論はテンプレートの場合と同様に動作します。いわば一枚のコインの裏表です。

しかし異なる動作を示す場面が 1 つだけあります。初期値を 27 とする int を宣言する場合を考えてみましょう。C++98 では 2 通りの記述があります。

```
int x1 = 27;
int x2(27);
```

C++11 では初期化の統一記法（uniform initialization）を採用したため、次の記述も可能です。

```
int x3 = { 27 };
int x4{ 27 };
```

記述は 4 通りでも結果は 1 つしかありません。値を 27 とする int です。

しかし、**項目 5** で述べるように、auto は型を明示した変数宣言より利点が多く、上例の変数宣言の int は auto に置き換えるのが良いでしょう。単純に字面を置き換えれば次のようになります。

```
auto x1 = 27;
auto x2(27);
auto x3 = { 27 };
auto x4{ 27 };
```

上例の宣言はすべて問題なくコンパイルできますが、すべてが同じ意味にはなりません。先頭の 2 つの文は同じ意味です。完全に同じです。27 という値を持つ int 型の変数を宣言します。しかし、次の 2 つの文は 27 という値の単一要素を持つ std::initializer_list<int> 型の変数を宣言しています！

```
auto x1 = 27;            // type is int, value is 27
                                          型は int、値は 27
auto x2(27);             // ditto                    同上

auto x3 = { 27 };        // type is std::initializer_list<int>,
                         // value is { 27 }
                          型は std::initializer_list<int> 値は { 27 }
auto x4{ 27 };           // ditto                    同上
```

正誤表より補足

2014 年 11 月に C++17 ドラフト（N3922）に変更が加えられ、次の 2 つは異なる意味になりました。

```
auto x3 = { 27 };
auto x4{ 27 };
```

x4 の型は int になり、std::initializer_list<int> にはなりません。すでに一部のコンパイラは対応を始めています。

これは auto の型推論規則の特例です。auto と宣言した変数の初期化子を波括弧で囲むと、推論される型は std::initializer_list になります。この型を推論できない場合は（波括弧で囲んだ初期化子の値が異なる型であるなどの理由により）、コンパイルできません。

```
auto x5 = { 1, 2, 3.0 };  // error! can't deduce T for
                          // std::initializer_list<T>
                              エラー！ std::initializer_list<T> の T を
                              推論できない
```

コメントに記したように、上例の型推論は失敗します。ここで 2 種類の型推論が実行されている点に注意してください。1 つは auto の使用から来るもので、x5 の型を推論するものです。x5 の初期化子を波括弧で囲んでいるため、x5 は必ず std::initializer_list と推論されます。しかし std::initializer_list はテンプレートです。ある型 T に対する std::initializer_list<T> とインスタンス化されるため（instantiation、実体化、具現化）、T の型も推論しなければなりません。この 2 番目の型推論、テンプレートの型推論が発生することにより型推論全体が失敗します。上例では、波括弧で囲んだ初期化子の型が単一ではないことが原因です。

auto の型推論とテンプレートの型推論で唯一異なる点は、この波括弧で囲んだ初期化子への対応です。auto で宣言する変数に波括弧で囲んだ初期化子を用いると、型の推論は std::initializer_list のインスタンス化となりますが、対応するテンプレートに同じ初期化子を与えても型を推論できず失敗します。

```
auto x = { 11, 23, 9 };   // x's type is
                          // std::initializer_list<int>
                                  x の型は std::initializer_list<int>
template<typename T>      // template with parameter
void f(T param);          // declaration equivalent to
                          // x's declaration
                              仮引数を持つテンプレートの宣言は x の宣言と
                              同じ
f({ 11, 23, 9 });         // error! can't deduce type for T
                                        エラー！ T の型を推論できない
```

しかし、テンプレートの方で param を未知の T に対する std::initializer_list<T> と指定すると、T は次のように推論されます。

```
template<typename T>
void f(std::initializer_list<T> initList);
```

```
f({ 11, 23, 9 });          // T deduced as int, and initList's
                           // type is std::initializer_list<int>
                                   T は int と、initList の型は std::
                                   initializer_list<int> と推論される
```

最終的に auto とテンプレートの型推論の唯一かつ実質的な差異は、auto が波括弧で囲んだ初期化子が std::initializer_list を表すと**想定する**のに対し、テンプレートの型推論は想定しないという点です。

auto の型推論では波括弧で囲んだ初期化子を特別扱いし、テンプレートではそうしないのはなぜだろうかと思う読者もいるかも知れません。著者も同感です。納得のいく説明を見つけられません。しかし規則は規則であり、auto により変数を宣言し、波括弧で囲んだ初期化子を用いるならば、推論される型は常に std::initializer_list になると覚えておかなければなりません。この動作は、特に初期値を波括弧で囲む初期化の統一記法を採用する場合に、当然のこととして肝に銘じておく必要があります。従来から C++11 に多いプログラミング上の誤りに、別の型を宣言するつもりで、std::initializer_list の変数を宣言してしまうというものがあります。初期化子を波括弧で囲むのはどうしてもそうしなければならない場合に限るとしている開発者もいますが、その理由の 1 つがこの落とし穴です（どうしてもそうしなければならない場合については**項目 7** で述べる）。

C++11 についてはこれ以上述べることはありませんが、C++14 では話はまだ続きます。C++14 は関数の戻り型を推論するための auto （**項目 3** を参照）、およびラムダの仮引数宣言での auto の使用を認めています。しかし、これらの auto は**テンプレートの型推論**であり、auto の型推論ではありません。そのため、波括弧で囲んだ初期化子を返し、戻り型を auto とした関数はコンパイルできません。

```
auto createInitList()
{
  return { 1, 2, 3 };          // error: can't deduce type
}                              // for { 1, 2, 3 }
                                  エラー：{ 1, 2, 3 } の型を推論できない
```

C++14 のラムダ式で仮引数の型指定に auto を用いた場合も同様です。

```
std::vector<int> v;
…

auto resetV =
  [&v](const auto& newValue) { v = newValue; };      // C++14

…
```

```
resetV({ 1, 2, 3 });          // error! can't deduce type
                              // for { 1, 2, 3 }
                                エラー！{ 1, 2, 3 } の型を推論できない
```

重要ポイント

- auto の型推論は通常はテンプレートのそれと同様だが、auto では波括弧で囲んだ初期化子を std::initializer_list と想定する点が異なる。
- 関数の戻り型やラムダ式の仮引数での auto はテンプレートの型推論と同じ動作となり、auto の型推論とは異なる。

項目 3：decltype を理解する

decltype はまったくおかしな代物です。名前や式を与えるとその型を教えてくれます。通常は予想通りの型を教えてくれますが、頭をかきむしったり、確認のために仕様やオンライン上の Q&A サイトを探しまくらなければならないような結果を返すこともあります。

まずは一般的な例から始めましょう。すんなり受け入れられるありふれた例です。テンプレート、auto の型推論の動作とは対照的に（**項目 1** および**項目 2** を参照）、decltype は、通常は、与えられた名前や式の正確な型をそのまま返します。

```
const int i = 0;              // decltype(i) is const int
                                                decltype(i) は const int
bool f(const Widget& w);      // decltype(w) is const Widget&
                              // decltype(f) is bool(const Widget&)
                                            decltype(w) は const Widget&
                                        decltype(f) は bool(const Widget&)
struct Point {
  int x, y;                   // decltype(Point::x) is int
};                            // decltype(Point::y) is int
                                          decltype(Point::x) は int
                                          decltype(Point::y) は int
Widget w;                     // decltype(w) is Widget
                                             decltype(w) は Widget
if (f(w)) …                   // decltype(f(w)) is bool
                                           decltype(f(w)) は bool
template<typename T>          // simplified version of std::vector
class vector {                                std::vector の簡易バージョン
public:
  …
```

```
  T& operator[](std::size_t index);
  …
};

vector<int> v;              // decltype(v) is vector<int>
…                                          decltype(v) は vector<int>
if (v[0] == 0) …            // decltype(v[0]) is int&
                                           decltype(v[0]) は int&
```

ご覧の通りです。驚くようなことは何もありません[3]。

C++11 での decltype の主な用途は、恐らく、戻り型が仮引数の型により決定される関数テンプレートの宣言でしょう。例えば、角括弧（「[]」）によるインデックス演算に対応したコンテナと、インデックス値をとる関数を開発するとします。関数はインデックス演算の結果を返す前にユーザ認証も行います。この場合の関数の戻り型はインデックス演算が返すのと同じ型になるべきです。

要素の型を T とするコンテナの operator[] は、通常 T& を返します。例えば std::deque がそうですし、std::vector もまず間違いなくそうです。しかし、std::vector<bool> では、operator[] は bool& を返さず、まったく別のオブジェクトを返します。この動作、理由については**項目 6** で別途解説します。ここで重要なのはコンテナの operator[] が返す型はコンテナにより決定される点です。

decltype を用いるとこの表現が容易になります。ここで開発するテンプレートの最初のバージョンを挙げましょう。戻り型を求める decltype の使用法を示すものです。まだ改善が必要なバージョンですが、おって解説します[4]。

```
template<typename Container, typename Index>    // works, but
auto authAndAccess(Container& c, Index i)       // requires
  -> decltype(c[i])                             // refinement
{                                                    動作するが、改
  authenticateUser();                                善が必要
  return c[i];
}
```

関数名の前に記述した auto は型推論に関与せず、C++11 の**戻り型の後置**（trailing return type）構文を表します。すなわち、仮引数の並びに続け、関数の戻り型を宣言します（「->」の後）。戻り型の後置には、関数の仮引数をその戻り型の指定に使用できる利点があります。上例の authAndAccess

[3] 正誤表より補足： 上例の最終行で、v が const vector の場合、decltype(v[0]) は const int& になる点に注意してください。std::vector::operator[] の const バージョンにオーバロード解決され、このバージョンの戻り型は const 参照であるためです。

[4] 正誤表より補足： この例ではインデックスに数値を使用するコンテナのみを対象としており、インデックスに任意の型を使用する std::map、std::unordered_map は除外されます。この点が値渡しする根拠にもなっています。

では c と i を用い戻り型を指定していますが、従来用いられて来た、関数名の前に戻り型を記述する記法では、c と i を使用できません。まだ宣言されていないのですから。

このように宣言すると、authAndAccess は、対象のコンテナの operator[] が返す型がどんなものでもそのまま返します。まさに期待通りです[5]。

C++11 は単文のラムダの戻り型を推論することを認めており、C++14 ではこれを拡張し、すべてのラムダ、すべての関数の戻り型を推論できます。複文も可能です。authAndAccess の場合では、C++14 を用いれば戻り型の後置を省略でき、冒頭の auto のみで済みます。この形態の宣言を用いると、auto による型推論が**実行されます**。特に、コンパイラが関数の戻り型をその実装から推論する点は大きな意味を持ちます。

```
template<typename Container, typename Index>     // C++14;
auto authAndAccess(Container& c, Index i)        // not quite
{                                                // correct
  authenticateUser();                                   誤っている
  return c[i];                    // return type deduced from c[i]
}                                                  戻り型を c[i] から推論
```

項目 2 では、戻り型を auto とした関数に対しては、コンパイラはテンプレートの型推論を実行すると述べました。上例の場合、この動作は問題になります。ほとんどのコンテナでは、要素の型が T ならば operator[] は T& を返しますが、**項目 1** で述べた通り、テンプレートの型推論では初期化式の参照性は無視されます。この点を次の例から考えてみましょう。

```
std::deque<int> d;
…
authAndAccess(d, 5) = 10;  // authenticate user, return d[5],
                           // then assign 10 to it;
                           // this won't compile!
                               ユーザ認証は d[5] を返し、そこに 10 を代入
                               する。コンパイルできない！
```

上例で、d[5] の型は int& ですが、auto による推論のため、authAndAccess の戻り型から参照性が失われ、戻り型は int になります。関数の戻り値が int ということは右辺値であり、上例は右辺値の int へ 10 を代入しようとすることになります。この代入は C++ では認められていないため、コンパイルできません。

authAndAccess を期待通りに動作させるには、戻り型に decltype を用いて推論させなければなりません。すなわち、authAndAccess は c[i] という式が返すのとまったく同じ型を返すことを明示するのです。C++ の守り人達は、型を推論する場面の一部については decltype を使用する必要が

[5] 正誤表より補足： 戻り型を推論する関数に return 文を複数記述する場合は、すべての return 文が一意な型を推論できなければなりません。

あるだろうと見越しており、C++14 からは decltype(auto) という指定子を導入しました。一見すると矛盾しているようにも見えますが（auto を decltype するって何？）、実際には隅々まで合点がいきます。この auto は型を推論することを表し、また、この decltype は推論の際に decltype の規則を適用することを表します。authAndAccess は次のように記述できます。

```
template<typename Container, typename Index>   // C++14; works,
decltype(auto)                                 // but still
authAndAccess(Container& c, Index i)           // requires
{                                              // refinement
  authenticateUser();
  return c[i];
}
```

C++14。動作するがまだ改善が必要

これでようやく authAndAccess は c[i] が返すのと同じ型を返すようになりました。c[i] が T& を返す一般的な場合は authAndAccess も T& を返し、c[i] がオブジェクトを返すというあまりない場合でも、authAndAccess は同様にオブジェクトを返します。

decltype(auto) の使用は関数の戻り型に限定されません。初期化式に decltype の型推論規則を適用したい変数宣言にも使用できます。

```
Widget w;

const Widget& cw = w;

auto myWidget1 = cw;                // auto type deduction:
                                    // myWidget1's type is Widget
```

auto による型推論：myWidget1 の型は Widget

```
decltype(auto) myWidget2 = cw;      // decltype type deduction:
                                    // myWidget2's type is
                                    // const Widget&
```

decltype による型推論：myWidget2 の型は const Widget&

しかし、障害が 2 つあることが分かっています。1 つは先に述べた authAndAccess の改善です。まだ解説していませんでしたが、ここで踏み込みます。

authAndAccess の C++14 バージョンを見返してください。

```
template<typename Container, typename Index>
decltype(auto) authAndAccess(Container& c, Index i);
```

上例のコンテナは非 const 左辺値参照として渡されます。コンテナ要素の参照を返した場合はコンテナを更新することが認められているためです。しかし、これではこの関数に右辺値のコンテナ

を渡せなくなってしまいます。右辺値は左辺値参照にはバインド[6]できません（const 左辺値参照は例外であり、この場合には当てはまらない）。

authAndAccess に右辺値のコンテナを渡すのは特殊でしょう。この点はその通りです。右辺値コンテナは一時オブジェクトであるため、通常は authAndAccess の呼び出しを含む文の終わりで破棄されます。すなわち、コンテナ内の要素を表す参照は（通常は authAndAccess が返すもの）、文の終わりで無効になってしまいます。それでも authAndAccess に一時オブジェクトを渡すことに意味がある場合もあります。authAndAccess を使用する開発者は、単に一時コンテナ内の要素のコピーが欲しいだけかも知れません。例を挙げましょう。

```
std::deque<std::string> makeStringDeque();   // factory function
                                                    factory 関数
// make copy of 5th element of deque returned
// from makeStringDeque
            makeStringDeque が返す deque の 5 番目の要素のコピーを作成する
auto s = authAndAccess(makeStringDeque(), 5);
```

このような場面にも対応するには、authAndAccess が左辺値も右辺値も受け付けるよう、その宣言を改善する必要があります。オーバロードも可能かも知れませんが（1 つが左辺値参照仮引数を、もう 1 つが右辺値参照仮引数をそれぞれ宣言する）、2 つの関数を保守しなければならなくなります。これを回避するには authAndAccess が**左辺値にも右辺値にも**バインド可能な参照仮引数を受け付けるよう宣言します。このユニヴァーサル参照の動作については**項目 24** で解説します。最終的に authAndAccess は次のように宣言できます。

```
template<typename Container, typename Index>    // c is now a
decltype(auto) authAndAccess(Container&& c,     // universal
                             Index i);          // reference
                                                   c はユニヴァーサ
                                                   ル参照となった
```

上例のテンプレートでは、処理対象のコンテナの型は未知のままです。また、使用するインデックスオブジェクトの型も未知のままです。一般に、未知の型のオブジェクトを値渡しすると無用のコピー動作につながり、性能上の問題やオブジェクトのスライスという動作上の問題（**項目 41** を参照）、また、同僚の嘲笑の槍玉に挙げられるなどの問題がありますが、コンテナのインデックスの場合では、標準ライブラリのインデックス演算の例にならうのは（std::string、std::vector、std::deque などの operator[] 演算）、充分意味があります。そのため、本書でもこの場合は値渡しを維持します。

しかし、**項目 25** の忠告にもあるように、テンプレートの実装はユニヴァーサル参照を std::forward するように変更する必要があります。

[6] 訳者注：値とシンボルの対応付け。束縛とも訳されます。

```
template<typename Container, typename Index>        // final
decltype(auto)                                      // C++14
authAndAccess(Container&& c, Index i)               // version
{                                                      C++14 の最終
  authenticateUser();                                  バージョン
  return std::forward<Container>(c)[i];
}
```

上例は期待した内容をすべて実現してくれますが、C++14 コンパイラが必要です。読者が C++14 コンパイラを持っていなければ、C++11 バージョンのテンプレートを使用せざるを得ません。C++14 バージョンと基本的には同じですが、戻り型を明示する必要があります。

```
template<typename Container, typename Index>        // final
auto                                                // C++11
authAndAccess(Container&& c, Index i)               // version
-> decltype(std::forward<Container>(c)[i])             C++11 の最終
{                                                      バージョン
  authenticateUser();
  return std::forward<Container>(c)[i];
}
```

読者を悩ませる点がもう 1 つあるとすれば、本項目の冒頭で著者が decltype は**ほぼ常に**期待通りの結果となる、すなわち**ごくまれに**予想外の結果となると述べたことでしょう。本当のところを言えば、読者がライブラリをヘビーに実装でもしない限り、問題に出会うことはまずありません。

decltype の動作を**完全に理解**するには、少数の特殊な場合も把握する必要があるでしょう。そのほとんどは本書のような書籍で解説するには非常に分かりにくいものですが、その 1 つを調べれば decltype の内部およびその使用法についての洞察が得られます。

何らかの名前を decltype に与えると、その名前に宣言された型が得られます。名前とは左辺値式ですが decltype の動作には影響しません[7]。単なる名前以上に複雑な左辺値式に対しては、常に左辺値参照型が得られます。すなわち、名前以外の左辺値式が型 T を持つ場合、decltype からはその型として T& が得られます。大部分の左辺値式は本質的に左辺値参照修飾子を内包しているため、この動作が問題になることはまずありません。例えば、左辺値を返す関数については常に左辺値参照が得られます。

この動作には注意を払う必要があります。

```
int x = 0;
```

[7] 正誤表より補足： 本文の「名前とは左辺値式である」が常に真とは限りません。右辺値の名前もわずかながら存在します（「this」など）。また、「単なる名前以上に複雑な左辺値式」は「名前により変数を特定する訳ではない左辺値式」とした方が適切でした（C++ 標準規格に通じている読者ならば、この段落が 7.1.6.2/4 第 1 項をまとめていることが分かるでしょう）。

上例で x は変数名です。そのため、decltype(x) は int となります。しかし、x という名前を丸括弧で囲み「(x)」とすると、単なる名前ではなく、複雑さを備えた式になります。名前としての x は左辺値であり、C++ では (x) という式も左辺値と定義しています。そのため、**decltype((x))** は int& となります。名前を丸括弧で囲むだけで、decltype から得られる型が変わってしまうのです！

C++11 ではこの動作はちょっと変わっている程度でしたが、C++14 では decltype(auto) という記述が可能になったため、うっかり return を少し変更しただけでも、関数に推論される型が影響を受けてしまう事態につながります。

```
decltype(auto) f1()
{
  int x = 0;
  …
  return x;         // decltype(x) is int, so f1 returns int
}                                  decltype(x) は int のため f1 は int を返す

decltype(auto) f2()
{
  int x = 0;
  …
  return (x);       // decltype((x)) is int&, so f2 returns int&
}                           decltype((x)) は int& のため、f2 は int& を返す
```

f2 が f1 と異なる点は、戻り型だけではない点に注意してください。ローカル変数の参照を返しているのです！ これは未定義動作へ向かう特急列車のようなコードです。そんな列車に乗りたい人はいないでしょう。

ここから得られる教訓は、decltype(auto) には充分な注意が必要ということです。その型を推論する式の何気ない細部が、decltype(auto) から得られる型に影響を与えます。期待通りの型を推論させるには、**項目 4** で述べた技法を駆使する必要があります。

同時に、大局を見失なわないことも重要です。decltype は予想外の型を推論することも確かにありますが（auto と併用する／しないに関わらず）、通常はまずありません。一般的には、decltype は期待通りの型を返します。名前に対し decltype を用いる場合は特にそうです。この場合の decltype は、その名の通り、名前を宣言した型を返します。

重要ポイント

- decltype はほぼ常に、変数または式の型をそのまま返す。

- 名前ではない、型を T とする左辺値式については、常に T& という型を返す。
- C++14 では decltype(auto) が追加された。auto のように初期化子から型を推論するが、適用される推論規則は decltype のものである。

項目 4：推論された型を確認する

推論した型を表示、確認する方法は複数あり、開発プロセスの段階に応じ変化します。ここでは 3 種類の方法を取り上げます。コーディング時に型推論情報を得る、コンパイル時に得る、実行時に得る、の 3 つです。

IDE のエディタ

IDE のエディタは、マウスカーソルを乗せるだけでプログラムエンティティ（変数、仮引数、関数など）の型を表示する機能を備えているのが一般的です。例えば、次のようなコードがあるとしましょう[8]。

```
const int theAnswer = 42;

auto x = theAnswer;
auto y = &theAnswer;
```

IDE のエディタは恐らく、x に推論した型を int、y については const int* と表示するでしょう。

この動作を実現するには開発中のコードが多かれ少なかれコンパイル可能な状態でなければなりません。IDE が型推論情報を表示できるのは、内部で C++ コンパイラを実行しているためです（もしくは、少なくともコンパイラのフロントエンド）。コンパイラがコードを構文解析、型推論できなければ型を表示できません。

int のような単純（組み込み）型では、IDE が表示する情報は通常問題ありません。しかし次の節で述べるように複雑な型が登場してくると、IDE が表示する情報がそれほど有用とは言えなくなるでしょう。

コンパイラによる診断情報

コンパイラにその推論した型を表示させる上手な方法は、わざとコンパイルエラーを起こさせることです。エラーメッセージにはその原因となった型の情報が、まず間違いなく含まれます。

[8] 訳者注：著者は『銀河ヒッチハイク・ガイド』に登場する「生命、宇宙、そして万物についての究極の疑問の答え」の「42」を意図しています。

例えば、先の例の x と y に推論された型を確認したいとします。クラステンプレートを宣言しますが、**定義はしないでおきます**。次のようなものです。

```
template<typename T>        // declaration only for TD;
class TD;                   // TD == "Type Displayer"
```

TD を宣言だけする。TD は「Type Displayer」の略

上例のテンプレートをインスタンス化しようとすると、その定義が存在しないため、エラーメッセージが出力されます。x と y の型を表示させるには、単にこれらの型の TD をインスタンス化しようとすれば良いのです。

```
TD<decltype(x)> xType;      // elicit errors containing
TD<decltype(y)> yType;      // x's and y's types
```

x と y それぞれの型を含むエラーメッセージが出力される

目的の情報を含むエラーメッセージをまず間違いなく表示してくれるのですから、上例では変数名を *variableName*Type という形式にしてあります（変数名 +Type）。著者が使用しているコンパイラの 1 つが上例をコンパイルした際のエラーメッセージの一部を挙げます（目的の部分は強調表示してある）。

```
error: aggregate 'TD<int> xType' has incomplete type and
    cannot be defined
error: aggregate 'TD<const int *> yType' has incomplete type
    and cannot be defined
```

別のコンパイラを使用しても、字面こそ違え、同様の情報を出力します。

```
error: 'xType' uses undefined class 'TD<int>'
error: 'yType' uses undefined class 'TD<const int *>'
```

メッセージ文章は異なりますが、著者が試したすべてのコンパイラが目的の情報を出力してくれました[9]。

実行時出力

printf を用いた方法では実行時まで結果が分かりません（printf を推奨している訳ではないが）。しかし、この方法は情報の書式を完全に制御できる利点があります。重要となるのは、目的の型の、表示に適したテクスト表現です。「そんなに大変じゃない」と読者は思うかも知れません。

[9] 正誤表より補足： Linux の Intel C++ コンパイラ バージョン 15.0.2 で、未定義テンプレートをインスタンス化しようとすると、型を表示しませんでした。

「typeid と std::type_info::name で良いじゃないか」と。x と y に推論した型を表示するこの探求では、次のように記述すれば良いと思われるでしょう。

```
std::cout << typeid(x).name() << '\n';    // display types for
std::cout << typeid(y).name() << '\n';    // x and y
                                                x と y の型を表示する
```

この方法は、x や y のようなオブジェクトに対し typeid を実行すると std::type_info オブジェクトが得られ、また std::type_info は name というメンバ関数を持っており、型を表す C スタイルの文字列（const char*）を生成するという点を前提としています。

処理系は std::type_info::name が意味ある内容を返すよう努めていますが、その保証はありません。つまりこの方法の有効性には処理系により差異があります。例えば GNU および Clang コンパイラでは、x の型を「i」、y の型を「PKi」と表示します。意味が分かればこの表示も納得できます。「i」は「int」を、「PK」は「pointer to ~~konst~~ const」を意味します（両コンパイラともこの種の「変形された」型（mangled、修飾された型、マングルされた型）を復元するツール c++filt に対応している）。Microsoft のコンパイラの表示はもう少し分かりやすくなっており、x は「int」、y は「int const *」と表示します。

上記の x と y の型は正しいのですから、型を表示する問題はすべて解決済みと考えるかも知れません。しかし、まぁそう慌てないで。もう少し複雑な例も考えてみましょう。

```
template<typename T>                  // template function to
void f(const T& param);               // be called
                                          呼び出されるテンプレート関数

std::vector<Widget> createVec();      // factory function
                                                    factory 関数

const auto vw = createVec();          // init vw w/factory return
                                          factory 関数の戻り値で vw を
                                          初期化
if (!vw.empty()) {
  f(&vw[0]);                          // call f          f を呼び出す
  …
}
```

上例はユーザ定義型（Widget）、STL のコンテナ（std::vector）、auto で宣言した変数（vw）を含んでおり、コンパイラが推論する型を確認したい要求が高まる場面です。例えば、テンプレートの型仮引数 T や関数 f の仮引数 param にどの型が推論されたのかなどです。

typeid で簡単に表示する方法が分かりやすいでしょう。目的の型を表示するコードを f に追加してみます。

```
template<typename T>
void f(const T& param)
```

```
{
  using std::cout;

  cout << "T =     " << typeid(T).name() << '\n';      // show T
                                                          T を表示
  cout << "param = " << typeid(param).name() << '\n'; // show
  …                                                   // param's
}                                                     // type
                                                    param の型を表示
```

GNU および Clang コンパイラにより作成した実行ファイルは、次のように表示します。

```
T =     PK6Widget
param = PK6Widget
```

このコンパイラが表示する PK は「pointer to const」を意味することはすでに分かっていますので、残る謎は 6 という数字です。これは単に対象のクラス名の文字数を表しています（Widget）。最終的に上記表示は、T も param も、const Widget* という型であると言っています。

Microsoft のコンパイラを用いると、次の表示になります。

```
T =     class Widget const *
param = class Widget const *
```

上記の異なる 3 つのコンパイラは同じ情報を表示しており、正しいとうかがえます。しかし、もっとよく見てください。f というテンプレートの param に宣言した型は const T& です。それなのに T と param が同じ型というのはおかしくありませんか？ T を int だとすると、param の型は const int& となるべきで、まったく異なる型のはずです。

残念ながら std::type_info::name が返す型は信頼できません。この場合では、3 つのコンパイラが表示する param の型はすべて誤りです。さらに言えば、誤ることが本質的に**求められている**のです。仕様によれば、std::type_info::name はテンプレート関数に値を渡した際の仮引数の型を返すとされています。**項目 1** で述べたように、型が参照だった場合、その参照性は無視され、さらには const もその意味を失うのです（volatile も）。これが param の型である const Widget * const & が、const Widget* と表示される原因です。まず型が持つ参照性が落とされ、次にポインタが持つ const 性が落とされるのです。

やはりこれも残念な点ですが、IDE のエディタが表示する情報も信頼できません。または少なくとも信頼をもって便利に使うことはできません。同じサンプルコードで、ある IDE のエディタは T の型を次のように表示することが分かっています（何ら加工せずに載せる[10]）。

[10] 訳者注：原文は “I am not making this up.” ピューリツァー賞も受賞した米作家 Dave Barry が好んで使ったフレーズで、著者はこれを拝借しています。この作家の著作には『*Dave Barry Does Japan*』というのもあります。

```
const
std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,
std::allocator<Widget> >::_Alloc>::value_type>::value_type *
```

また、同じ IDE のエディタは param の型を次のように表示します。

```
const std::_Simple_types<...>::value_type *const &
```

上例は T の場合よりはましに見えます。途中の「...」には混乱させられますが、「この部分の T はまとめて省略するよ」と IDE のエディタが言っていると分かればなんとかなるでしょう。少しばかりの幸運があれば、読者の開発環境ではもっと上手に表示されることもあるでしょう。

運頼みではく、ライブラリを頼りにしたければ、std::type_info::name や IDE が正しく表示できない場合でもちゃんとした型を表示するよう開発された、Boost の TypeIndex ライブラリの存在は嬉しいニュースでしょう（一般に Boost.TypeIndex と表記される）。このライブラリは標準 C++ には含まれておらず、IDE でも、また TD のようなテンプレートでもありません。Boost はプラットフォーム非依存なオープンソースのライブラリです（http://boost.org で公開されている）。異常なほど細かいところにまでこだわる企業弁護士が見ても文句がないライセンスで公開されており、Boost ライブラリを用いたコードは、標準ライブラリとほぼ同等の可搬性を備えています。

Boost.TypeIndex を用い、先に挙げた関数 f の正確な型情報を表示してみます。

```
#include <boost/type_index.hpp>

template<typename T>
void f(const T& param)
{
  using std::cout;
  using boost::typeindex::type_id_with_cvr;

  // show T                                                    T を表示
  cout << "T =     "
       << type_id_with_cvr<T>().pretty_name()
       << '\n';

  // show param's type                                  param の型を表示
  cout << "param = "
       << type_id_with_cvr<decltype(param)>().pretty_name()
       << '\n';
  …
}
```

上例を実行すると、関数テンプレート boost::typeindex::type_id_with_cvr が型実引数をとり（目的の型）、const、volatile、参照の修飾子を**落とさない**点が大きな意味を持ちます（そのため、const、volatile、参照（reference）を表す「with_cvr」がテンプレート名に付いている）。戻り値は boost::typeindex::type_index オブジェクトであり、そのメンバ関数 pretty_name から std::string が得られます。人間が読みやすい書式の型名です。

先に挙げた f の実装を呼び出す場面をもう一度考えてみましょう。typeid を用いると、param の誤った型が返されます。

```
std::vector<Widget> createVec();      // factory function
                                                  factory 関数
const auto vw = createVec();          // init vw w/factory return
                                         factory の戻り値で vw を初期化
if (!vw.empty()) {
  f(&vw[0]);                          // call f          f を呼び出す
  …
}
```

GNU および Clang コンパイラでは、Boost.TypeIndex を使用すると次のように（正しく）表示されます。

```
T =     Widget const*
param = Widget const* const&
```

Microsoft コンパイラも実質的に同じ内容を表示します。

```
T =     class Widget const *
param = class Widget const * const &
```

このようなほぼ完全な統一性は良いことですが、IDE のエディタ、コンパイラのエラーメッセージ、Boost.TypeIndex などのライブラリはコンパイラが推論する型の確認に使える道具に過ぎないと覚えておくことも重要です。いずれも有用ですが、最終的には**項目 1** から**項目 3** までで述べた型推論の理解、把握に勝るものはありません。

重要ポイント

- 推論された型は、通常は、IDE のエディタ、コンパイラのエラーメッセージ、Boost の TypeIndex ライブラリを用いると確認できる。
- 一部の方法では、表示される型が正確でも有用でもない。そのため、C++ の型推論規則に対する理解が必要不可欠であることは変わらない。