C++プログラミングII

第3回 テンプレート

岡本秀輔

成蹊大学理工学部

関数テンプレート

関数テンプレート(復習)

- 関数で使う型情報を引数にした型板（テンプレート）
- 呼び出し指定により具体的な関数を生成
- 引数の型はコンパイラが推定可能/明示も可能
  - 文字列リテラルは const char*と推定されるので注意

```
template<typename T> // T はテンプレート引数
void print_smaller(T a, T b) {
    if (a < b)
        std::cout << a <<"\n";
    else
        std::cout << b <<"\n";
}

int main() {
    print_smaller(5, 3); // 3, T=int
    print_smaller(2.5, 4.8); // 2.5, T=double
    print_smaller("abc", "xyz"); // 不定,T=const char*
    print_smaller<string>("abc", "xyz"); // abc
}
```

print_smallerへのオブジェクトの適用

- 関数テンプレートの実引数はクラスも適用可能
  - std::string はクラスなので既に使っている
- テンプレートがクラスの持つべき機能を決める
  - print_smaller では < と << の演算子が必要

```
class Sales { // 売上情報
    std::string name; int num; // 名前と個数
  public:
    Sales(std::string a, int b) :name(a), num(b){}
    bool operator<(const Sales& p)const
            { return num < p.num; }
    friend auto&
    operator<<(std::ostream& o, const Sales& p)
            { return o << p.name <<": "<< p.num; }
};

int main() {
   Sales a{"abc", 40}, b{"xyz", 20};
   print_smaller(a, b); // xyz:  20
}
```

print_smaller への vector の適用

- std::vector は < 演算子を備えている
- << 演算子があれば print_smaller で利用可能

```
template<typename T>
void print_smaller(T a, T b) { /* 前と同じ */ }

template<typename T>
auto& operator<<(std::ostream& o, const vector<T>& v)
{
   for (auto& e : v) o << e <<" ";
   return o;
}

int main() {
   vector a{3,4,5,6}, b{2,4,8,1,5};
   std::cout << a <<"\n"; // 3 4 5 6
   print_smaller(a, b);   // 2 4 8 1 5
}
```

テンプレート引数のデフォルト引数

- テンプレート引数 T のデフォルト値は T{}
- 指定された型の 0 に相当する値となる

```
template<typename T>       // テンプレート引数
T sum(T a, T b, T c =T{}) // デフォルト引数
{
   return a + b + c;
}

int main()
{
   using std::cout;
   cout << sum(1,2)<<" "<< sum(1,2,3)<<"\n"; // 3 6
   using namespace std::string_literals;
   cout << sum("a"s, "b"s)<<" "                // ab
        << sum("a"s, "bc"s, "def"s)<<"\n"; // abcdef
}
```

リファレンスの使用

リファレンス引数

- lvalue リファレンスは対象が変数などの左辺値
- リテラルなどの rvalue は参照できない

```
void print(int& n) {
    cout << &n <<" "<< n <<"\n";
}

int main() {
    int  x {1};
    int& y {x};
    cout << x <<" "<< y <<"\n";
    // int & z  {3} ;  // error:  リテラルは rvalue

    print(x);
    // print(3); // error
}
```

const リファレンス引数

- const リファレンスはリテラルでも式でも OK
- コンパイラが一時変数を作成して参照する

```
void print(const int& n) {
    cout << &n <<" "<< n <<"\n";
}

int main() {
    int  x {1};
    const int& a { x };
    const int& b { 5 };   // リテラル
    const int& c { 5*x }; // 式
    cout << a <<" "<< b <<" "<< c <<"\n";

    print(x);
    print(3);
    print(3*x);
}
```

用語

一般の局所変数の場合：

lvalue リファレンス

- 指定：`T& n{a};`
- 意味：`n` は `a` を参照している

const リファレンス

- 指定：`const T& cn{a};`
- 意味：`cn` は `a` を参照し変更しない

関数の引数の場合：

(lvalue) リファレンス引数

- 指定：`void f(T& n);`
- 意味：`n` は実引数の変数を参照している

const リファレンス

- 指定：`void g(const T& cn);`
- 意味：`cn` は実引数を参照し変更しない

関数テンプレートの改善

- オブジェクトの値渡し引数は効率が悪い場合あり
  - 大きなサイズの場合、コピーに時間がかかる
  - const リファレンス引数が良い
- const リファレンス引数はリテラルも実引数に使用可
  - わずかに無駄な処理が入る

```
template<typename T> // const リファレンス引数
void print_smaller(const T& a, const T& b) {
    if (a < b)
        cout << a <<"\n";
    else
        cout << b <<"\n";
}

int main() {
    vector a{3,4,5,6}, b{2,4,8,1,5};
    print_smaller(a, b); //  2 4 8 1 5
    print_smaller(3, 5);  // 3
    print_smaller(b[1], 5); // 4
}
```

標準ライブラリでは

- std::min():小さい方を返す

```
template<typename T>
const T& min( const T& a, const T& b );
```

- std::max():大きい方を返す

```
template<typename T>
const T& max( const T& a, const T& b );
```

- どちらも < 演算子を使う
- クラスのオブジェクトでも OK

lvalue リファレンスを返す関数

- 関数の呼び出し側で戻された変数を操作できる

```
int& func(int& x) { return x; } // リファレンスを返す
int main() {
    int a {1};
    std::cout << func(a) <<"\n";  // 1 戻った a を出力
    int& b { func(a) };           // 戻った a を b で参照
    ++ b;
    std::cout << a <<"\n";        // 2
    func(a) = 3;                  // 戻った a を直接変更
    std::cout << a <<"\n";        // 3
}
```

- 演算子の連続的な利用

```
ostream& operator<<(ostream& o, const MyVec& v);
MyVec a, b;
cout << a << b; // (cout << a) << b の意味
```

リファレンスを返す際の注意

- 返した変数が関数終了後も利用可能であること

```
// 局所変数のリファレンスを返すの不可
int& bad() {
    int x;     // bad() 関数の局所変数
    return x; // x は return 直後に廃棄
}

// std::min, std::max の実引数に式を指定して
// 式結果の一時変数をリファレンスで受け取るのも不可
int& r { std::min(5, a+1) }; // 即廃棄の一時変数
```

```
int& ok { std::max(a, b) }; // a,b は廃棄されないので OK
```

確認クイズ

- 大きい方を出力するプログラムを完成させよ
- `std::max()` を使用すること

```
#include <iostream>
#include <vector>
using std::cout, std::vector, std::string;

template<typename T>
void print_max(const T& a, const T& b) {
    // ここだけで完成させる
}

int main() {
    vector x{3,4,2,5,8}, y{5, 4, 2};
    print_max(x, y); // 5 4 2

    vector<string> s{"ab","cde"}, t{"xyz","0123"};
    print_max(s, t); // xyz 0123
}
```

解答例

- 選んでから処理する

```
template<typename T>
void print_max(const T& a, const T& b) {
   // m は a か b のどちらか
   auto& m { std::max(a,b) }; // const T &になる
   for (auto& e : m)          // e も const
      cout << e <<" ";
   cout <<"\n";
}
```

- さらに短く書くと

```
template<typename T>
void print_max(const T& a, const T& b) {
   for (auto& e : std::max(a,b))
      cout << e <<" ";
   cout <<"\n";
}
```

[] 演算子のメンバ関数

- [] 演算子には通常 lvalue 用と rvalue 用の二つを作る
  - セッター/ゲッター
- const の有無でメンバ関数が多重定義となる
- 使用する場面で適切に選択される

```
class Vec3d {
    std::vector<int> vec;
  public:
    Vec3d(int a=0, int b=0, int c=0) :vec{a,b,c} {}
    int  operator[](size_t i)const { return vec[i];}
    int& operator[](size_t i)      { return vec[i];}
};

int main()
{
   Vec3d x {1,2,1};
   x[2] = 3; // const なしメンバ関数
   std::cout << x[2] <<"\n"; // const メンバ関数
}
```

クラステンプレート

クラス テンプレート

- 関数と同じくクラスもテンプレート化が可能
- それぞれ別々の型が作られる
  - Pair<int,int>と Pair<int,double>は別の型
- 変数宣言時のテンプレート引数も省略可能 (C++17)

```
template<typename T, typename K>
class Pair {  // std::pair と同じ構造
  public:
    T x;
    K y;
    Pair(T a, K b) :x{a},y{b}{}
};

int main() {
   Pair<int,int>         a{3, 40};
   Pair<int,double>      b{3, 40.3};
   Pair<double,string> c{3.5, "abc"};
   Pair d{1, 2}; // Pair<int,int>の省略版
}
```

引数での使用

- 関数テンプレートの引数で使う場合には選択肢がある

```
// Pair を必ず使う関数テンプレート
template<typename T, typename K>
void print(const Pair<T,K>& p) { // Pair 専用を明示
   std::cout << p.x <<" "<< p.y <<"\n";
}

// Pair とは一見無関係，.x や .y で Pair に依存する
template<typename T>
void print2(const T& p) { // シンプルだが適用範囲は広い
   std::cout << p.x <<" "<< p.y <<"\n";
}

int main() {
   Pair a{3, 40.3};
   print(a);
   print2(a);
}
```

演算子の多重定義

クラステンプレートで演算子を多重定義する場合の方法

- メンバ関数にはテンプレート引数を使用する *1
- 一般関数は関数テンプレートで指定する *2

```
template<typename T>
class Point {  // 2次元座標系の点
  public:
    T x{}, y{};
    Point(T a, T b): x{a}, y{b}{}
    bool operator==(const Point<T>& p) const // *1
      { return x == p.x && y == p.y); }
};

template<typename T> // *2
auto& operator<<(std::ostream& o, const Point<T>& p)
{
   return o <<"("<< p.x <<","<< p.y <<")";
}
```

よくある間違い

- 既存の演算子との多重定義の重複に注意する

```
// コンパイルできない例
template<typename T> // 広範囲に適用可能なテンプレート
auto& operator<<(std::ostream& o, const T& p)
{
   return o <<"("<< p.x <<","<< p.y <<")";
} // このテンプレートにエラーはない

int main() {
  Point<int> a{1,2};
  cout << "a = "                // ここでエラー
       << a <<"\n";
}
```

- cout << "a = "のコンパイルで混乱発生
- エラー: ambiguous overload for 'operator<<'
- ambiguous: あいまい

Point クラステンプレートの利用

- Point<int>と Point<size_t>は別の型

```
int main()
{
   Point<int> a{3,-4}, b{3,-4}, c{1,2};
   if (a == b) cout <<"a,b => "<< a <<"\n";
   if (a == c) cout <<"why?\n"; // 出力なし
   c = a;  // 代入可能
   if (a == c) cout <<"a,c => "<< c <<"\n";

   Point<size_t> x{3,4}, y{3,4};
   if (x == y) cout <<"x,y => "<< x <<"\n";

   // a = x; // エラー:  型が異なるの代入不可
}
```

vector と Point クラステンプレート

- vector<int>と vector<Point<int>>は同じ使い方
- main() の 2 つの処理がほぼ同じ
  - 型指定、初期値、find のリテラル

```
template<typename T>
size_t find(const vector<T>& v, const T& x) {
    for (size_t i = 0; i < v.size(); i++)
        if (v[i] == x) return i;
    return v.size();
}

int main() {
    vector<int> v {1,2,3,4,5};
    if (auto i{find(v, 3)}; i < v.size())
        cout <<"v["<< i <<"] = "<< v[i] <<"\n";

    vector<Point<int>> vp {{1,2},{2,3},{3,4},{4,5}};
    if (auto i{find(vp, {3,4})}; i < vp.size())
        cout <<"vp["<< i <<"] = "<< vp[i] <<"\n";
}
```

さらにテンプレート化

- 抽象的な処理方法と具体的な処理対象の分離
- ここまでテンプレート化する利点
  - test の処理がデータ型に依存しているかの確認
  - アルゴリズムの性能調査がしやすい

```
template<typename T>
void test(const vector<T>& v, const T& x) {
   if (auto i{find(v, x)}; i < v.size())
      cout <<"v["<< i <<"] = "<< v[i] <<"\n";
}

int main() {
   test<int>( {1,2,3,4,5}, 3 );
   test<Point<int>>({{1,2},{2,3},{3,4},{4,5}},{3,4});
}
```

テンプレートの特殊化

テンプレートの限界

- 特定の型に対してうまく行かない時がある
  - double では 0.1+0.2 != 0.3 である

```
template<typename T>
bool is_equal(T x, T y) { return x == y; }

int main()
{
    if (is_equal(1+2, 3))  // true
        std::cout << "ok\n";

    std::string s1{"ab"}, s2{"abc"};
    if (is_equal(s1+"c", s2))  // true
        std::cout << "ok\n";

    if (is_equal(0.1+0.2, 0.3)) // false?
        std::cout << "ok\n";
}
```

関数テンプレートの特殊化

- 特定の型専用のパターンを用意できる

```
template<typename T>
bool is_equal(T x, T y) { return x == y; }

// 関数テンプレートの特殊化
template<>  // 固定したいテンプレート引数を省略
bool is_equal(double x, double y) { // 型を固定
    const double eps {0.01}; // epsilon
    return std::abs(x-y) < eps;
}

int main() {
    if (is_equal(1+2, 3))  // true
        std::cout << "ok\n";

    if (is_equal(0.1+0.2, 0.3)) // true
        std::cout << "ok\n";
}
```

別の解決策方法

- 関数テンプレートに対応する多重定義を作る
- 違い：
  - 関数テンプレートの特殊化
    - 使用されなければ無視される
    - ヘッダファイルに向いている
  - 多重定義
    - 使用されなくても機械語は生成される
    - 簡潔に指定できる
    - テンプレートよりも優先

```
template<typename T>
bool is_equal(T x, T y) { return x == y; }

// 関数の多重定義 (template<>がない)
bool is_equal(double x, double y) { // 型を固定
    const double eps {0.01}; // epsilon
    return std::abs(x-y) < eps;
}
```

メンバ関数の特殊化

- メンバ関数を特定の型だけ別の動作にするには
  多重定義ではなく特殊化で対応する

```
template<typename T>
class Pt {  // 名前付きの点
    std::string name;
    T val;
  public:
    Pt(std::string a, T b): name{a}, val{b}{}
    bool is_equal(const T& a) const{ return a==val;}
    // ここで多重定義はできない
};

template<> // メンバ関数の特殊化
bool Pt<double>::is_equal(const double& a) const {
   const double eps {1e-5};
   return std::abs(val-a) < eps;
}
```

まとめ

まとめ

- 関数テンプレート
  - 関数で使う型情報を引数にした型板
  - クラスも実引数として使用可能
  - テンプレートが指定する型の機能を決める
- クラステンプレート
  - クラスで指定する型を引数にする
  - 引数指定の省略でもコンパイラが推定可能
- テンプレートの特殊化
  - 特定の型に対して決まった型の関数を作る
  - 特殊化と多重定義の二種類の方法がある
  - 多重定義はテンプレートよりも優先
  - 多重定義は必ず機械語に変換される
    - 実行ファイルのサイズが大きくなる
- テンプレートを組み合わせて
  抽象的な処理と具体的なものを分離する