6 パラメータ付きのクラス定義、総称

6–1 パラメータ付きのクラス定義

例えば 、例3.7で定義した

「char型データを要素とするスタック」のクラスStackChar

に類似したクラスとして

「double型データを要素とするスタック」のクラス

を構成したい場合は、単に、

◇ 既存の StackChar.h, StackChar.cpp というコード中の
「char」を機械的に「double」に置き換え、

◇ 必要な調整を少し加える

だけで良い。ただ、

この種の作業を繰り返すと本質的に同じコードが多数できるので、

◇ クラスを定義する時間の無駄、

◇ 保持するコードの無駄

◇ 保守も面倒　(←修正は全てのコードに対して行う必要がある)

⇨

◇ こういった無駄を排し、

◇ 本質的に同じコードを１つにまとめ上げる

ために、C++言語では、

扱うデータの型等をパラメータ化したクラス定義

ができる。

例6. 1 (扱うデータの型をパラメータ化したスタック)　例3.7の「char型データを要素とするスタック」のクラス StackChar において、

保持する要素データの型の部分をパラメータ化

して得られたクラス定義の例を次に示す。(ここで、プログラム中の 下線 は StackChar.h, StackChar.cpp からの主要な変更箇所を表す。)

注意：一般に、パラメータ付きのクラス定義を行う場合 、ヘッダファイルの中に実装部も含めるのが無難。

その理由：

例えば仕様部 Stack.h と実装部 Stack.cpp に分けても、「g++ -c Stack.cpp」によって出来る Stack.o はパラメータ付きクラスのメンバ関数群に過ぎず、実際に利用したい Stack<char> 等のメンバ関数群ではない。

$\Longrightarrow$ リンク時に「‘Stack<char>::Stack(int)’ に対する定義されていない参照です」といった類のエラー

```
[motoki@x205a]$  cat -n Stack.h
     1 /* 「扱うデータの型をパラメータ化したスタック」オブジェ */
     2 /* クトのクラス Stack の仕様部                          */
     3
     4 #ifndef ___Class_Stack
     5 #define ___Class_Stack
     6         ↓次に続く定義が型パラメータ TYPE をもつことを明示
     7 template<typename TYPE>
     8 class Stack {  ほぼ char→TYPE という単純な置き換えだけ
                      使用例: Stack<char>, Stack<double>, ...
                                    ↑
                      「 char型を要素とするスタック」のクラス
     9   static int numOfInstances;
                              // これまでに生成したStack<TYPE>
                                                  // インスタンスの個数
    10   const int id;      // インスタンスに固有のid番号
    11   TYPE* element;     // スタック領域へのポインタ
    12   int  size;         // スタックの容量
```

```
13    int  top;          // スタックのtop要素を保持する
                                        // 配列要素の番号
14 public:
15    explicit Stack(int size = 100); // スタック容量=size
                                      // として初期化。(デフォルト
16                                    // コンストラクタの役割も)
17    Stack(const Stack<TYPE>& stack);  //コピーコンストラ...
                          StackChar.h 内の次の関数に相当
                        ↓ StackChar(const std::string& str);
18    Stack(const TYPE a[], const int arraySize);
19                                    // 引数の配列要素を
                                      // スタックに入れて初期化
                    StackChar.h 内の次の関数に相当
                  ↓ StackChar(int size, const std::string& str);
20    Stack(int size, const TYPE a[], //容量がsizeのスタ
21          const int arraySize);     //ック領域を確保し
                                // そこに引数の配列要素を入れて初期化
22    ~Stack() { delete[] element; }
```

```
23    // オブジェクト(もしくはStack<TYPE>クラス全体)
                                        // の情報を提供するための関数群
24    static int getNumOfInstances() {
                                              return numOfInstances; }
25                                    // これまでに生成したStack<TYPE>
                                            // インスタンスの個数を返す
26    int getId() const { return id; }  //スタックのid番号...
27    int getSize() const { return size; }  //スタック容量...
28    int getNumOfElements() const { return top+1; }
                                           //スタック内の要素数を...
29    void showContents() const;     //スタックの状況を出力
30    // Stack<TYPE>型オブジェクトを操作するための関数群
31    void reset() { top = -1; }      // スタックを空にする
32    void resize(int size);          // スタック容量を変更
33    bool isEmpty() const { return (top < 0); }  //スタ
                                              // ックが空か否か
34    void pushdown(TYPE ele);        // pushdown操作
```

```
35    TYPE popup();                          // popup操作
36 };
37
38
39 //=============================================================
40 /* 「扱うデータの型をパラメータ化したスタック」オブジェ */
41 /* クトのクラス Stack の実装部                                  */
42
43 #include <iostream>
44 #include <typeinfo>          ←typeid()演算子を使うので...
45 #include <cstdlib>     // for exit()
46 #include <cstring>     // for memcpy()
47 //#include "Stack.h" ←実装部もヘッダファイル中に入れた
48 //using namespace std;
49 ↑       色々な所からインクルードされる可能性があるので...
```

```
50 // static変数の初期化 ------------------------------
51 template<typename TYPE>
52 int Stack<TYPE>::numOfInstances = 0;
53
54 // 各種コンストラクタ --------------------------------
55 template<typename TYPE>          //スタック容量=sizeとし
                                    //て初期化(デフォルトコン
56 Stack<TYPE>::Stack(int size)    //ストラクタの役割も)
57               : id(numOfInstances++), size(size),
                                                top(-1)
58 {
59   element = new TYPE[size];
60 }
61
```

```
62 template<typename TYPE>          //コピーコンストラクタ
63 Stack<TYPE>::Stack(const <TYPE>& stack)
64                : id(numOfInstances++), size(stack.size),
                                                top(stack.top)
65 {
66   element = new TYPE[stack.size];
67   memcpy(element, stack.element,
                                      sizeof(TYPE)*stack.size);
68 }
69
```

StackChar.h 内の次の関数に相当
↓ StackChar(const std::string& str);

```
70 template<typename TYPE>                    //引数の配列要素を
                                             //スタックに入れて初期化
71 Stack<TYPE>::Stack(const TYPE a[],
                                    const int arraySize)
72                : id(numOfInstances++), size(arraySize),
                                      top(arraySize-1)
73 {
74   element = new TYPE[size];
75   for (int i=0; i<arraySize; i++)
76     element[i] = a[i];
77 }
78
```

StackChar.h 内の次の関数に相当
↓ StackChar(int size, const std::string& str);

```
79 template<typename TYPE>     //容量がsizeのスタック領域
                                //を確保し、そこに引数の
80 Stack<TYPE>::Stack(int size,  //配列要素を入れて初期化
81                     const TYPE a[], const int arraySize)
82               : id(numOfInstances++), size(size),
                                          top(arraySize-1)
83 {
84   element = new TYPE[size];
85   for (int i=0; i<arraySize; i++)
86     element[i] = a[i];
87 }
88
89 // オブジェクトの情報を提供するための関数群 ----------------
90 template<typename TYPE>          // スタックの内容を表示
91 void Stack<TYPE>::showContents() const
```

```
 92 {
 93   std::cout << "stack_of_" << typeid(TYPE).name()
 94         << "(id=" << id << ",size=" << size
 95         << ") has " << top+1 << " elements as follows:"
                                                << std::endl;
 96   std::cout << "   [Bottom] ";
 97   for (int i=0; i<=top; i++)
 98     std::cout << element[i] << "   ";
 99   std::cout << "<--" << std::endl;
100 }
```

StackChar.cpp 内の次の行に相当
↓ cout << "stack_of_char"...

typeid() ... 引数の型情報を表すオブジェクトへの参照を値とする演算子
name() ... type_infoクラスのメンバ関数
typeid(TYPE).name()
... 実行時にTYPEに割当てられたデータ型を表す文字列(処理系に依存)

```
101
102 // <TYPE>型オブジェクトを操作するための関数群 ------------
103 template<typename TYPE>          //スタック容量を変更
104 void Stack<TYPE>::resize(int size)
105 {
106   if (top >= size) {
107     std::cout << "insufficient stack size"
                                          << std::endl;
108     exit(1);
109   }
110   TYPE* temp = new TYPE[size];
111   memcpy(temp, element, sizeof(TYPE)*(top+1));
112   delete[] element;
113   this->size = size;
114   element = temp;
115 }
116
```

```
117 template<typename TYPE>          //pushdown操作
118 void Stack<TYPE>::pushdown(TYPE ele)
119 {
120   if (++top >= size) {
121     this->resize(size+10);
122   }
123   element[top] = ele;
124 }
125
126 template<typename TYPE>          //popup操作
127 TYPE Stack<TYPE>::popup()
128 {
129   if (top < 0) {
130     std::cout << "popup from empty stack"
                                        << std::endl;
131     exit(1);
132   }
```

```
133   return element[top--];
134 }
135
136 #endif
[motoki@x205a]$
```

これに関して、

- 型パラメータ付きのクラス Stack が上の様に定義されていれば 、次の様なプログラムを書くこともできる。

```
[motoki@x205a]$  cat -n useStackDouble.cpp
     1 /* 総称的に定義されたクラス Stack.h の利用例                    */
     2 /* (1)配列内のdouble型データ列を
                                   スタックを用いて反転した後出力 */
     3 /* (2)デフォルトコンストラクタの利用                          */
     4 /* (3)コピーコンストラクタの利用                              */
     5
     6 #include <iostream>
     7 #include "Stack.h"
```

```
 8 using namespace std;
 9
10 int main(void)
11 {
12   Stack<double> stackA(100);
13   double a[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
14
15   for (int i=0; i < 5; ++i)
16     stackA.pushdown(a[i]);
17   for (int i=0; !stackA.isEmpty(); ++i)
18     a[i] = stackA.popup();
19   cout << "Reversed data sequence a = ( ";
20   for (int i=0; i<5; ++i)
21     cout << a[i] << "  ";
22   cout << ")" << endl << "---" << endl;
23
24   Stack<double> stackB; //デフォルトコンストラクタ利用
```

```
25    Stack<double> stack[3]; //デフォルトコンストラクタ利..
26    for (int i=0; i<3; ++i) {
27      for (int k=0; k<=i; ++k)
28        stack[i].pushdown(99.99);
29    }
30    stackB.showContents();
31    for (int i=0; i<3; ++i)
32      stack[i].showContents();
33    cout << "---" << endl;
34
35    Stack<double> stackC(a, sizeof(a)/sizeof(double));
36    Stack<double> stackD(stackC);  //コピーコンスト
                                          //ラクタの利用
37    stackD.pushdown(1.0/0.0);
38    Stack<double> stackE = stackD; //コピーコンスト
                                          //ラクタの利用？
```

```
    39    stackE.popup();
    40    stackE.pushdown(0.0/0.0);
    41    stackC.showContents();
    42    stackD.showContents();
    43    stackE.showContents();
    44    cout << "---" << endl;
    45
    46    cout << "生成されたStack<double>インスタンス数 = "
    47         << Stack<double>::getNumOfInstances()
                                                  << endl;
    48 }
[motoki@x205a]$  g++ useStackDouble.cpp
[motoki@x205a]$  ./a.out
Reversed data sequence a = ( 5.5  4.4  3.3  2.2  1.1  )
---
stack_of_d(id=1,size=100) has 0 elements as follows:
  [Bottom] <--
```

```
stack_of_d(id=2,size=100) has 1 elements as follows:
  [Bottom] 99.99  <--
stack_of_d(id=3,size=100) has 2 elements as follows:
  [Bottom] 99.99  99.99  <--
stack_of_d(id=4,size=100) has 3 elements as follows:
  [Bottom] 99.99  99.99  99.99  <--
---
stack_of_d(id=5,size=5) has 5 elements as follows:
  [Bottom] 5.5  4.4  3.3  2.2  1.1  <--
stack_of_d(id=6,size=15) has 6 elements as follows:
  [Bottom] 5.5  4.4  3.3  2.2  1.1  inf  <--
stack_of_d(id=7,size=15) has 6 elements as follows:
  [Bottom] 5.5  4.4  3.3  2.2  1.1  -nan  <--
---
生成されたStack<double>インスタンス数 = 8
[motoki@x205a]$
```

パラメータ付きの定義は 、
利用する際のパラメータ部の指定 (具現化,instantiation)
に従ってコンパイル時に色々な定義を創り出す
$\Longrightarrow$ 多態性をもたらすとも考えられる。

パラメータ付きの定義を用いたプログラミングを一般に
総称的プログラミング (ジェネリックプログラミング)
と呼ぶ。

(補足,どうしても仕様部と実装部を分けたい場合) 例えば
上の例6.1で仕様部と実装部を分けて
Stack<char>, Stack<double>という具現化を行う場合は、

⇨ 実装部のコード Stack.cpp の最後に下記の19行を追加しておけば、
Stack.cpp のコンパイル結果である Stack.o の中には
◇ 具現化前のコードだけでなく、
◇ Stack<char>のメンバ関数の実行コードも
◇ Stack<double>のメンバ関数の実行コードも
含まれる様になる。

```
template int Stack<char>::numOfInstances;
template Stack<char>::Stack(int size);
template Stack<char>::Stack(const Stack<char>& stack);
template Stack<char>::Stack(const char a[],
                            const int arraySize);
template Stack<char>::Stack(int size, const char a[],
```

```
                                                const int arraySize);
template void Stack<char>::showContents() const;
template void Stack<char>::resize(int size);
template void Stack<char>::pushdown(char ele);
template char Stack<char>::popup();

template int Stack<double>::numOfInstances;
template Stack<double>::Stack(int size);
template Stack<double>::Stack(const Stack<double>& stack)
template Stack<double>::Stack(const double a[],
                                                const int arraySize);
template Stack<double>::Stack(int size, const double a[],
                                                const int arraySize);
template void Stack<double>::showContents() const;
template void Stack<double>::resize(int size);
template void Stack<double>::pushdown(double ele);
template double Stack<double>::popup();
```

パラメータ付きの定義の構文：

- 一般に、定義の先頭位置に

  `template <` パラメータ指定`, ..., `パラメータ指定 `>`

  という記述を挿入することにより、定義内の「データ型」や「整定数」をパラメータ化できる。

  ここで、パラメータ指定 としては次の形のものが許される。

  - ◇ `typename` パラメータ名 ←データ型名を表すパラメータ
  - ◇ `class` パラメータ名 ←データ型名を表すパラメータ
  - ◇ `int` パラメータ名 ←整定数を表すパラメータ
  - ◇ `typename` パラメータ名 `=` デフォルトのデータ型名
  - ◇ `class` パラメータ名 `=` デフォルトのデータ型名
  - ◇ `int` パラメータ名 `=` デフォルトの整定数値

パラメータ付きのクラス定義に関する諸注意：

- パラメータ付きのクラス(テンプレートクラスもしくはクラステンプレートという)を利用したい場合、具現化したクラスを

  クラス名< パラメータ指定 , . . . , パラメータ指定 >

  という形で表す。

- 一般に、パラメータ付きのクラス定義を行う場合は、ヘッダファイルの中に仕様部だけでなく実装部も含める方が無難である。

- テンプレートクラスにフレンド関数を指定することもできる。

  その際、
  - ◇ パラメータに依存しないフレンド関数を指定した場合は、そのフレンド関数は全ての具現化クラスのフレンド関数になる。
  - ◇ パラメータに依存したフレンド関数を指定した場合は、そのフレンド関数は具現化クラスのフレンド関数になる。

- テンプレートクラス内で宣言された static なデータメンバに関しては、個別の具現化クラス毎にデータ領域が確保される。

  例えば、例6.1で定義されたStackクラスの具現化クラスとして Stack<char> とStack<double> が利用される場合、
  ◇ Stack<char> の static なデータメンバである
  Stack<char>::numOfInstances と
  ◇ Stack<double> の static なデータメンバである
  Stack<double>::numOfInstances
  は別の変数領域として確保され、具現化クラス毎に通しのid番号が振り分けられることになる。

- 整定数値を表すパラメータ... テンプレートクラス内の配列メンバの要素数をパラメータ化したい場合等に利用できる。

例6. 2 (添字の有効性をチェックする機能を備えた配列)　指定されたパラメータTYPEに対して「添字の有効性をチェックする機能を備えたTYPE型配列」のクラスをもたらすテンプレートクラスの定義例を次に示す。

```
[motoki@x205a]$  cat -n SafeArray.h
     1 /* 「添字の有効性をチェックする機能を備えた配列」をインス */
     2 /* タンスとする型パラメータ付きクラス SafeArray の仕様部 *
     3
     4 #ifndef ___Class_SafeArray
     5 #define ___Class_SafeArray
     6
     7 template<typename TYPE>
     8 class SafeArray {
     9   TYPE* basePtr;          //pointer to the 1st element
    10   int size;               //array size
    11 public:
    12   explicit SafeArray(int n=100);        //create a Safe
    13   SafeArray(const SafeArray<TYPE>& a);  //copy constru
```

```
14    SafeArray(const TYPE a[], int n);      //copy a standa
15    ~SafeArray() { delete[] basePtr; }
16    int getSize() { return size; }
17    typedef TYPE* iterator;
18    iterator begin() { return basePtr; }
19    iterator end() { return basePtr + size; }
20    TYPE& operator[](int i);     //range-checked element
21    SafeArray<TYPE>& operator=(const SafeArray<TYPE>& a);
22 };
23
24 //=========================================================
25 /* 「添字の有効性をチェックする機能を備えた配列」をインス */
26 /* タンスとする型パラメータ付きクラス SafeArray の実装部 *
27
28 #include <cassert>
29
```

← 入れ物の実装方法に依存しない形でデータ群の繰り返し処理を記述するため

← 代入演算子 =

```
30 template<typename TYPE> //create a SafeArray of size n
31 SafeArray<TYPE>::SafeArray(int n): size(n)
32 {
33   assert(n > 0);
34   basePtr = new TYPE[size];
35   assert(basePtr != 0);
36 }
37
38 template<typename TYPE>          //copy constructor
39 SafeArray<TYPE>::SafeArray(const SafeArray<TYPE>& a)
40 {
41   size = a.size;
42   basePtr = new TYPE[size];
43   assert(basePtr != 0);
44   for (int i=0; i<size; ++i)
45     basePtr[i] = a.basePtr[i];
46 }
```

```
47
48 template<typename TYPE>          //copy a standard array
49 SafeArray<TYPE>::SafeArray(const TYPE a[], int n)
50 {
51   assert(n > 0);
52   size = n;
53   basePtr = new TYPE[size];
54   assert(basePtr != 0);
55   for (int i=0; i<size; ++i)
56     basePtr[i] = a[i];
57 }
58
59 template<typename TYPE>          //range-checked element
60 TYPE& SafeArray<TYPE>::operator[](int i)
61 {
62   assert (0<=i && i<size);
63   return basePtr[i];
```

```
64 }
65                                                    代入演算子 =
66 template<typename TYPE>                    //assignment operator
67 SafeArray<TYPE>& SafeArray<TYPE>::
                          operator=(const SafeArray<TYPE>& a)
68 {
69   assert (a.size == size);
70   for (int i=0; i<size; ++i)
71     basePtr[i] = a.basePtr[i];
72   return *this;
73 }
74
75 #endif
[motoki@x205a]$
```

これに関して、

- 型パラメータ付きのクラス SafeArray が上の様に定義されていれば、次の様なプログラムを書くこともできる。

```
[motoki@x205a]$ cat -n useSafeArrayInt.cpp
     1 // 型パラメータ付きのクラス SafeArray を利用した例
     2
     3 #include <iostream>
     4 #include <iomanip>
     5 #include <string>
     6 #include "SafeArray.h"
     7 using namespace std;
     8
     9 void showContentsOf(string arrayName,
                           SafeArray<int> a);
    10
    11 int main()
    12 {
    13   SafeArray<int> a(10), b(10);
    14
    15   for (int i=0; i<a.getSize(); i++) {
```

```
16      a[i] = i+10;
17      b[i] = i*i;
18    }
19    showContentsOf("a", a);
20    showContentsOf("b", b);
21
22    a = b;                                          代入演算子 =
23    cout << "a = b;" << endl;
24    cout << "==> ";
25    showContentsOf("a", a);
26 }
27
28 void showContentsOf(string arrayName,
                                          SafeArray<int> a)
29 {
30    cout << arrayName << ".size=" << a.getSize()
                                              << ",   "
```

```
31          << arrayName << "=(";
32    for (SafeArray<int>::                データ保持方法に依存しない形
                    iterator p=a.begin(); p!=a.end(); ++p)
33      cout << right << setw(4) << *p;                    反復子
34    cout << ")" << endl;
35 }
[motoki@x205a]$  g++ useSafeArrayInt.cpp
[motoki@x205a]$  ./a.out
a.size=10,  a=(  10  11  12  13  14  15  16  17  18  19)
b.size=10,  b=(   0   1   4   9  16  25  36  49  64  81)
a = b;
==> a.size=10,  a=(   0   1   4   9  16  25  36  49  64  81)
[motoki@x205a]$
```

6–2 型パラメータ付きの関数定義

通常の関数内に現れるデータ型の部分をパラメータ化することもできる。

型パラメータ付きの関数... テンプレート関数, 関数テンプレート

例6. 3 (配列全体をコピーする型パラメータ付き関数)

2つのテンプレート関数

◇ copy() ... TYPE2型配列をTYPE1型配列に要素単位でコピー

◇ print() ... TYPE型配列の要素列を出力

を定義し使用した例を次に示す。

```
[motoki@x205a]$  cat -n useTemplateCopy.cpp
     1 // 型パラメータ付きの関数 copy() の定義・利用例
     2
     3 #include <iostream>
     4 #include <iomanip>
```

```
 5 #include <string>
 6 using namespace std;
 7
 8 template<typename TYPE1, typename TYPE2>
 9 void copy(TYPE1 x[], TYPE2 y[], int size);
10
11 template<typename TYPE>
12 void print(string arrayName, TYPE x[], int size);
13
14 int main()
15 {
16   double a[5] = {1.1, 2.2, 3.3, 4.4, 5.5}, b[5];
17   int m[5];
18
19   print("a", a, 5);
20   ::copy(b, a, 5);
```

標準ライブラリ std 内の copy() 関数を適用候補から除外するため

```
21   print("b", b, 5);
22
23   ::copy(m, a, 5);
24   print("m", m, 5);
25 }
26
27 template<typename TYPE1, typename TYPE2>
28 void copy(TYPE1 x[], TYPE2 y[], int size)
29 {
30   for (int i=0; i<size; ++i)
31     x[i] = y[i];
32 }
33
```

```
34 template<typename TYPE>
35 void print(string arrayName, TYPE x[], int size)
36 {
37   cout << arrayName << "[" << size << "] = {"
                                << scientific << fixed;
38   for (int i=0; i<size-1; ++i)
39     cout << x[i] << ", ";
40   cout << x[size-1] << "}" << endl;
41 }
[motoki@x205a]$  g++ useTemplateCopy.cpp
[motoki@x205a]$  ./a.out
a[5] = {1.100000, 2.200000, 3.300000, 4.400000, 5.500000}
b[5] = {1.100000, 2.200000, 3.300000, 4.400000, 5.500000}
m[5] = {1, 2, 3, 4, 5}
[motoki@x205a]$
```

例6. 4 (2つの変数の中身を交換する型パラメータ付き関数)

例2.5で定義した関数をパラメータ化し、テンプレート関数

◇ swap() ... TYPE型の2つの変数の中身を交換

を定義し使用した例を次に示す。

```
[motoki@x205a]$  cat -n useTemplateSwap.cpp
     1 // 型パラメータ付きの関数 swap() の定義・利用例
     2
     3 #include <iostream>
     4 #include <cstring>
     5 using namespace std;
     6
     7 template<typename TYPE>
     8 void swap(TYPE& x, TYPE& y);
     9
    10 void swap(char* x, char* y);
    11
    12 int main()
```

```
13 {
14    int i=1, j=22;
15    double a=33.3, b=444.4;
16    char str1[]="abcd", str2[]="1234";
17    ↓                    std内のswap()を適用候補から除外するため
18    ::swap(i, j);
19    cout << "i=" << i << ", j=" << j << endl;
20
21    ::swap(a, b);
22    cout << "a=" << a << ", b=" << b << endl;
23
24    ::swap(str1, str2);     2つのswap()関数を適用可
                              ⇒非テンプレート関数の方を優先
25    cout << "str1=" << str1 << ", str2=" << str2
                                                << endl;
26 }
27
```

```
28 template<typename TYPE>
29 void swap(TYPE& x, TYPE& y)
30 {
31   TYPE temp;
32
33   temp = x;
34   x = y;
35   y = temp;
36 }
37
38 void swap(char* x, char* y)
39 {                          ↓          標準ライブラリstd内のmax()
40   int maxLength = max(strlen(x), strlen(y));
41   char* temp = new char[maxLength+1];
42
43   strcpy(temp, x);
44   strcpy(x, y);
```

```
    45    strcpy(y, temp);
    46    delete[] temp;
    47 }
[motoki@x205a]$  g++ useTemplateSwap.cpp
[motoki@x205a]$  ./a.out
i=22, j=1
a=444.4, b=33.3
str1=1234, str2=abcd
[motoki@x205a]$
```

ここで、

- 一般に、適用可能な関数が複数あった時は 、次の順に優先
  ① 自明な型変換でシグネチャ合致する非テンプレート関数
  ② 適切な型パラメータ設定でシグネチャ合致するテンプレート関数
  ③ 非テンプレート関数上での通常の選択手順による関数 (2.11節)

6–3 標準テンプレートライブラリ (STL)

C++言語においては、標準テンプレートライブラリ (STL) の中に、
◇ 標準的なデータ構造やアルゴリズムを実装したコードのテンプレートが用意されていて、
◇ それらを組み合わせて自由に総称的プログラミングを進められる。

STLの中核は次の3種類の要素

- コンテナ · · · データ (群) を保持するための入れ物オブジェクト
- 反復子 (イテレータ) · · · ポインタに似たオブジェクトで、
  コンテナ内の全てのデータを扱う繰り返し処理を、採用したコンテナの実装方法に依存しない形で書き表すことを可能にする。
  (処理するコードを変えずにコンテナの実装方法を変更可。)
- アルゴリズム · · · 用意されたコンテナの下での、標準的 (で効率の良い) 操作手段を提供する。

例6. 5 (反復子を用いた繰り返し)　例2.8のプログラム
use_vectorType.cppに現れる繰り返し処理の中で、
「v[i]」という要素表現は配列やベクトルに固有

⇨ 反復子を用いて書き換え

(下線は例2.8のuse_vectorType.cpp からの主要な変更箇所)

[motoki@x205a]$ cat -n useSTL_vectorInt_iterator.cpp

```
 1 // STL内のテンプレートクラス vector<int> を利用した例
 2
 3 #include <iostream>
 4 #include <iomanip>
 5 #include <vector>
 6 using namespace std;
 7
 8 void showContentsOf(vector<int> v);
 9
10 int main()
11 {
```

```
12    vector<int> v;
13
14    showContentsOf(v);
15
16    for (int i=0; i<10; i++)
17      v.push_back(i);
18    showContentsOf(v);
19
20    for (int i=0; i<5; i++)
21      v.pop_back();
22    showContentsOf(v);
23
24    for (vector<int>::iterator p=v.begin(); p!=v.end(); p++)
25      *p *= *p;
26    showContentsOf(v);
27  }
```

反復子　最初の要素

↓　↓　最後の要素の次

↓　↑

次の要素を指す様に

```
28
29 void showContentsOf(vector<int> v)
30 {
31   vector<int>::iterator p=v.begin();
32
33   cout << "要素数 = " << v.size() << endl;
34   cout << "内容   = (";
35   if (v.size() > 0)
36     cout << right << setw(2) << *(p++);
37   for ( ; p!=v.end(); p++)
38     cout << ", "<< right << setw(2) << *p;
39   cout << " )" << endl;
40 }
[motoki@x205a]$  g++ useSTL_vectorInt_iterator.cpp
[motoki@x205a]$  ./a.out
要素数 = 0
内容   = ( )
```

```
要素数 = 10
内容   = ( 0,  1,  2,  3,  4,  5,  6,  7,  8,  9 )
要素数 = 5
内容   = ( 0,  1,  2,  3,  4 )
要素数 = 5
内容   = ( 0,  1,  4,  9, 16 )
[motoki@x205a]$
```

ここで、

- プログラム **24∼25行目** の部分は例**2.8**では次の様に記述。

```
for (int i=0; i<v.size(); i++)
  v[i] *= v[i];
```

  ベクトルや配列に固有の表し方である `v[i]` という表現が除去されたことが確認される。

例6. 6 (コンテナlist<double>, ライブラリnumeric)

◇ list<double> ... 双方向連結リストでdouble型データ群を保持するオブジェクトのクラス、

◇ accumulate() ... 汎用数値アルゴリズムのライブラリnumeric内の関数の使用例を示す。

↑ 反復子等を用いて、コンテナのデータ保持方式に依らない実装

```
[motoki@x205a]$  cat -n useSTL_listDouble.cpp
     1 // STL内のテンプレートクラス list<double> を利用した例
     2
     3 #include <iostream>
     4 #include <iomanip>
     5 #include <list>
     6 #include <numeric>          // for accumulate()
     7 using namespace std;
     8
     9 void showContentsOf(list<double>& listD);
```

```
10
11 int main()
12 {
13   double a[5] = {3.3, 2.2, 5.5, 1.1, 4.4};
14   list<double> z;
15
16   for (int i=0; i<5; ++i)
17     z.push_front(a[i]);
18   showContentsOf(z);
19
20   z.sort();
21   showContentsOf(z);
22
23   cout << "sum = " << accumulate(z.begin(),
                                    z.end(), 0.0) << endl;
```

←リストの先頭要素の直前に挿入

←保持データ群を小さい順に並び替え

↓
(第1引数の指す要素)+
...+(第2引数の指す要素の手前)
+(第3引数の値)

```
24 }
25
26 void showContentsOf(list<double>& listD)
27 {
28   list<double>::iterator p=listD.begin();
29
30   cout << "要素数 = " << listD.size() << endl;
31   cout << "内容   = (";
32   if (listD.size() > 0)
33     cout << *(p++);
34   for ( ; p!=listD.end(); p++)
35     cout << ", "<< *p;
36   cout << " )" << endl;
37 }
```

```
[motoki@x205a]$  g++ useSTL_listDouble.cpp
[motoki@x205a]$  ./a.out
要素数 = 5
```

```
内容    = (4.4, 1.1, 5.5, 2.2, 3.3 )
要素数 = 5
内容    = (1.1, 2.2, 3.3, 4.4, 5.5 )
sum = 16.5
[motoki@x205a]$
```

例6. 7 (コンテナmap<string,int,less<string>␣>)

◇ (連想)コンテナmap ... 数学的な意味での「写像のグラフ」、i.e.

何らかの写像 $f:K \to V$ の対応関係

を表す集合 $\{(x,f(x))|x \in K\}$

を保持するオブジェクト

を利用して、

標準入力から入力した単語の出現回数を保持

した例を次に示す。

```
[motoki@x205a]$  cat -n useSTL_mapStringInt.cpp
     1 // STL内のテンプレートクラス
                    // map<string,int,less<string> >を利用した例
     2 //    単語列を入力し、各単語の出現回数を表示する
     3
     4 #include <iostream>
     5 #include <iomanip>
     6 #include <string>
     7 #include <map>
```

```
 8 using namespace std;
 9
10 int main()
11 {
12   map<string,int,less<string> > word_freq;
13   map<string,int,less<string> >::iterator p;
14   string word;
15
16   //標準入力から単語を次々に入力し順にword_freq上に記録
```

(string型,int型)という形の2項組のデータ集合を第1要素の小さい順に保持するオブジェクトの型

↓

↑ キー

↑ キー値を基に検索

↑ 半角空白を削除するとエラー

```
17    cin >> word;
18    while (cin.good()) {
19      if (word[word.size()-1]==',' 
                                  || word[word.size()-1]=='.'
20          || word[word.size()-1]==';'
                                 || word[word.size()-1]==':')
21        word = word.substr(0,word.size()-1);   ←
22      if (word.size() == 0)          単語最後のコンマ等の除去
23        continue;      ↓    wordを第1要素にもつ2項組を探す
24      p = word_freq.find(word);
25      if (p != word_freq.end())
26        ++(p->second);                  ←2項組の2番目の要素
27      else                ↓          新規2項組(word,1)を挿入
28        word_freq.insert(make_pair(word, 1));
                                    ↑       //word_freq[word]=1;
29      cin >> word;          2項組(word,1)を新規に構成
```

```
30    }
31
32    //word_freq内に保存されているデータを表示
33    cout << word_freq.size()
                    << " distinct words appeared: " << endl;
34    int num = 0;←現出力行に既に出力した2項組の個数を保持
35    cout << "{ ";
36    p = word_freq.begin() ;
37    if (p!=word_freq.end()) {  ↓      2項組の1番目の要素
38      cout << setw(15) << p->first << ":"
                                    << setw(3) <<p->second;
39      ++num;                                2項組の2番目の要素
40      ++p;
41    }
42    for ( ; p != word_freq.end(); ++p, ++num) {
43      if (num >= 3) {
44        cout << "," << endl
```

```
45              << "   ";
46        num = 0;
47      }else {
48        cout << ", ";
49      }
50      cout << setw(15) << p->first << ":"
                                    << setw(3) <<p->second;
51    }
52    cout << " }" << endl;
53
54    //入力エラーが起きてなかったか確認
55    if (!cin.eof())
56      cout << "Warning: input error." << endl;
57 }
[motoki@x205a]$  g++ useSTL_mapStringInt.cpp
[motoki@x205a]$  cat useSTL_mapStringInt.data
C++, invented at Bell labs by Bjarne Stroustrup in the mid-198
```

```
is a powerful modern successor language to C.  C++ adds to C t
concept of class, a mechanism for providing user-defined types
also called abstract data types.  C++ supports object-oriented
programming by these means and by providing inheritance and
runtime type binding.
[motoki@x205a]$  ./a.out < useSTL_mapStringInt.data
42 distinct words appeared:
{           Bell:  1,          Bjarne:  1,              C:
            C++:  3,      Stroustrup:  1,              a:
       abstract:  1,           adds:  1,           also:
            and:  2,             at:  1,        binding:
             by:  3,         called:  1,          class:
        concept:  1,           data:  1,            for:
             in:  1,     inheritance:  1,       invented:
             is:  1,           labs:  1,       language:
          means:  1,      mechanism:  1,      mid-1980s:
         modern:  1, object-oriented:  1,             of:
```

```
          powerful:  1,     programming:  1,       providing:
           runtime:  1,       successor:  1,        supports:
               the:  2,           these:  1,              to:
              type:  1,           types:  2,   user-defined:
[motoki@x205a]$
```

例6. 8 (STLアルゴリズム内のfind(),sort())

STLアルゴリズムのライブラリalgorithm内の

◇ find()関数 と

◇ sort()関数

を使用した例を次に示す。

```
[motoki@x205a]$  cat -n useSTL_algorithm.cpp
     1 // STLアルゴリズム内の find(), sort() を利用した例
     2
     3 #include <iostream>
     4 #include <string>
     5 #include <vector>
     6 #include <algorithm>
     7 using namespace std;
     8
     9 void showContentsOf(vector<string> wordSeq);
    10
    11 int main()
```

```
12 {
13   vector<string> wordSeq;
14   vector<string>::iterator where;
15   string word;
16
17   //標準入力から単語を次々に入力し、順にwordSeqに記録
18   cin >> word;
19   while (cin.good()) {
20     wordSeq.push_back(word);    ←動的配列の最後尾に追加
21     cin >> word;
22   }
23   cout << "入力単語列:" << endl;
24   showContentsOf(wordSeq);
25   cout << "---" << endl;
26
```

```
27    //STLアルゴリズム内の find() を利用
                                  (第1,第2引数で指定された範囲) から (第3
                                  引数のデータ) を探す。
                                  ◇あった→最初の要素へのポインタを返す。
                    ↓             ◇無い→第2引数のポインタ値を返す。
28    where = find(wordSeq.begin(), wordSeq.end(), "C++");
29    cout << "最初の\"C++\"以降の10単語列:" << endl;
30    for (int i=0 ; where!=wordSeq.end() && i<10;
                                                  ++where, ++i)
31      cout << *where << " ";
32    cout << endl
33         << "---" << endl;
34
35    //STLアルゴリズム内の sort() を利用
36    sort(wordSeq.begin(), wordSeq.end());←昇順に並べ替え
37    cout << "整列後の単語列:" << endl;
38    showContentsOf(wordSeq);
```

```
39 }
40
41 void showContentsOf(vector<string> wordSeq) {
42   int lineLength = 0;
43   for (vector<string>::iterator p=wordSeq.begin();
                                    p!=wordSeq.end(); ++p) {
44     if (lineLength+p->size()+1 >70) {
45       cout << endl;
46       lineLength = 0;
47     }
48     cout << *p << " ";
49     lineLength += p->size() + 1;
50   }
51   cout << endl;
52 }
[motoki@x205a]$  g++ useSTL_algorithm.cpp
[motoki@x205a]$  cat useSTL_mapStringInt.data
```

```
C++, invented at Bell labs by Bjarne Stroustrup in the mid-198
is a powerful modern successor language to C.  C++ adds to C t
concept of class, a mechanism for providing user-defined types
also called abstract data types.  C++ supports object-oriented
programming by these means and by providing inheritance and
runtime type binding.
[motoki@x205a]$  ./a.out < useSTL_mapStringInt.data
```

入力単語列:

```
C++, invented at Bell labs by Bjarne Stroustrup in the mid-198
a powerful modern successor language to C. C++ adds to C the c
of class, a mechanism for providing user-defined types, also c
abstract data types. C++ supports object-oriented programming
these means and by providing inheritance and runtime type binc
---
```

最初の"C++"以降の10単語列: ↓冒頭の"C++,"と"C++"を区別したので

```
C++ adds to C the concept of class, a mechanism
---
```

整列後の単語列:

```
Bell Bjarne C C++ C++ C++, C. Stroustrup a a abstract adds als
and at binding. by by by called class, concept data for in
inheritance invented is labs language means mechanism mid-198(
modern object-oriented of powerful programming providing provi
runtime successor supports the the these to to type types, typ
user-defined
[motoki@x205a]$
```