データの格納と参照について

渡辺大地

1　データ構造とアルゴリズム

言うまでもなく、プログラミングという行為を究極的に表すと、「**データ**」と「**処理**」をどのように組み合わせるかということである。長年の研究と実践の積み重ねから、それぞれに体系だった理論が構築されており、それぞれ「データ構造」「アルゴリズム」と呼ばれている。今回は、このうち「データ構造」についての実践的な利用を学んでみよう。

1.1　「コンテナ」という概念

プログラミングを行うとき、データをどのように扱うかは大変重要で難しい問題となることが多い。本来、人間が何か仕事を行う際に、どのような「記憶」が作業過程で必要となるかは意識しないことが多い。しかし、これをプログラムで記述する際にはなんとかして厳密に**形式化**する必要がある。この「形式化」が厄介な代物である。しかも、プログラムを新規に作成する度に新たな形式を考えなければならず、それをコーディングする必要もある。これはなかなか面倒な作業となる。そんなとき、データの格納や取得を簡単に指示できるような「秘書」のような存在を、誰もが切望したくなる。

当然ながら、そんな都合のいい存在があるわけはなく、結局は自分でコーディングするしかない。しかし、ある程度のデータベース的なモジュールを単独で準備し、それを利用することで毎回のコーディングを避けることくらいはできそうなものである。実際、プロのプログラマは「秘書」の役割を果たしてくれる機能を持つモジュールを、自作して利用しているか、あるいは他の誰かが作ったモジュールを利用しているかのいずれかを実行している。

上に挙げたような、データの格納や取得を目的として作成されたもののうち、システムやアプリケーションであればそれは「**データベース (Data Base)**」と呼ばれる。それに対し、プログラムの中で利用するモジュールであった場合には「**コンテナ (Container)**」と呼ばれることが多い。C++ や Java などのクラスライブラリをサポートする言語では、コンテナをクラスで明確に実装していることが多く、そのようなクラスを「**コンテナクラス**」と呼ぶ。今回は、この「コンテナクラス」の役割を実用を考えてみる。

1.2　コンテナに要求される機能

コンテナクラスの機能は端的に言ってしまえばたったの二つであり、それは「格納」と「参照」である。理想論を言えば、コンテナクラスにはあらゆるものをどんな方法でも「格納」ができて、あらゆるものをどんな方法で「参照」できることが望ましい。そして、そのようなコンテナクラスを作成することは不可能ではない。しかし、そのようなクラスを利用することは実は現実的な選択肢ではない。なぜなら、**汎用性**は**効率**を下げる要因になるからである。

大抵の場合、コンテナクラスは複数の種類が準備されていて、それぞれに機能の**得手不得手**というものがある。中には、あまりに効率が悪くなるために最初からサポートされない機能もそれぞれに存在したりもする。従って、プログラマは自分がどのコンテナクラスを選択すべきかを適切に判

断する必要があるのである。この後の節では、様々なコンテナクラスの概念、機能、特徴、具体的な利用を紹介していくが、その際に不適切な利用例も合わせて紹介する。

1.3 コンテナとイテレータ (反復子)

大抵のコンテナクラスには、「**イテレータ (反復子)**」という仕組みが用意されている。イテレータというのは、簡単に言うと添字番号を用いずに要素に順番にアクセスする手段のことである。イテレータの実装は言語やライブラリによって様々なものだが、大体次のような方法を提供している。

1. まず、最初の格納要素を返す。
2. 次に、順番に次の要素を返すようなメソッドが準備されており、それを用いる。
3. 終端まで至ったかどうかを判断する手段が提供されており、終端に至ったら処理を終了する。

C++ における具体的な利用方法は後の節で述べる。

2 配列

2.1 配列の概念

配列は、プログラミング言語の誕生と共に存在する最も原始的なコンテナである。概念は非常に単純なもので、ある型の変数が列状に並んでいるものと捉えることができる。

2.2 配列の機能

配列の機能としては、次のようなものが挙げられる。

- 配列の作成は、格納するデータの型名及び配列の長さを指定する。
- データの格納及び参照は、両方とも添字番号を利用する。

2.3 配列の特徴

配列は、なんといってもその長所として

「どのようなプログラミング言語でも文法レベルでサポートされている。」

であることが挙げられる。また、ほとんどのプログラミング言語においては非常に高速に代入、参照を行うことのできる手段である。

しかし、短所は次に挙げるように非常に多い。

- 最初に指定したサイズから可変でない。サイズの伸縮を行いたい場合は、新規に別の配列を作成してデータを全てコピーするしかない。
- 添字番号以外にデータにアクセスする手段がない。
- 木構造など、直線的でないデータ構造を表現できない。

- データの挿入ができない。

ただし、比較的最近に生まれた言語ではこれらの短所のうちの幾つかを克服したものもある。

総じて、C や Fortran といった大昔の言語ならばいざ知らず、C++ や Java といった言語でわざわざ使う理由はどこにもないというのが結論となる。もちろん、関数やメソッド内の一時的な記憶領域、クラス内のメンバデータとして利用される分には積極的に利用することには差し支えないが、データベースとしての役割を配列に求めるのは酷である。そのような機能は、それぞれの言語の専用コンテナクラスにまかせるべきである。コンテナクラスでは、前述した配列の短所の多くを補い、かつ速度や効率も妥協する必要がないよう設計されている。

3 ベクトルクラス

3.1 ベクトルクラスの概念

コンテナクラスとして最初に紹介するのは**ベクトル (Vector) クラス**である。ベクトルクラスは、これまで配列が果たしてきた役割をそのまま継承し、それに伸縮自在な機能を持つということを目的として生み出された。従って、配列の持つ全ての長所をあわせ持ち、さらに便利になったものと考えて良いだろう。

ベクトルクラスは、内部では単純に配列をそのまま持っていることが多い。ただし、格納データの個数が配列のサイズを上回りそうになったとき、内部で新たに配列を作成してそこに格納するように処理されている点が異なる。これを実現するには、

1. さらに大きな配列を作成し、既存データを新配列にコピーしてから、旧配列を捨てる。
2. 新規の配列を作成し、既存配列より後ろのデータはそこに格納するようにする。(従って、ベクトルクラスは内部で複数の配列を持つことになる。)

という二つのスタンスが考えられ、前者は参照速度を、後者は格納速度を優先した案である。通常、ベクトルクラスはデータの格納速度よりも参照速度の方が重要視されるという理由で、大抵の場合は前者の案が採用されている。

3.2 ベクトルクラスの特徴

前述したように、ベクトルクラスは配列が持つ特徴をそのまま引き継いでいる。長所は以下に挙げる通りである。

- 伸縮自在である。
- 添字番号によるアクセスが可能であり、かつ大変高速である。
- 全てのコンテナクラスの中で、最もデータ格納の効率がよい。

一方、短所は以下に挙げる通りである。

- 添字番号以外のデータアクセス手段が存在しない。
- 末尾データ以外の場所へのデータの挿入や削除ができないか、あるいはできてもかなり効率が悪い。

- 木構造など、直線的でないデータ構造を表現できない。

ベクトルクラスは、良くも悪くも配列の拡張である。添字番号によってアクセスすることを念頭においた場面では絶大な威力を発揮し、そうでない場面ではまるで役に立たないことになる。ただし、末尾要素に対して簡単に追加、削除が行える点は非常に便利で有用である。また、ベクトルクラスでは添字番号による参照がイテレータの役割を果たすことができるため、単に効率のことだけを考えるならばイテレータをあえて用いる必要性はない。強いて上げれば、他のコンテナクラスとの互換性を維持するために用いられるのが普通である。

3.3 C++ でのベクトルクラス

C++ に準備されているコンテナクラスは、**STL** と呼ばれる C++ 用の標準ライブラリの中にある。**テンプレート (Template)** と呼ばれる文法を用いて利用可能で、「vector テンプレート」というものでベクトルクラスを作成することになる。

形式は以下のようなものとなる。

```
vector<データ型名>　コンテナ変数名;
```

例えば、int 型のベクトルクラス arrayInt を作成する場合には、以下のような記述となる。

```
vector<int>     arrayInt;
```

このようにして作成したコンテナに対し、次のようなメンバ関数を利用することができる。「TYPE」は、ベクトルクラスの格納データ型を指す。他にもメンバ関数は多く存在するが、当面これだけあれば困ることはないだろう。

push_back(TYPE)
　末尾に要素を追加する。サイズは自動的に 1 大きくなる。

pop_back()
　末尾の要素を削除する。サイズは自動的に 1 小さくなる。

size(void)
　現在の格納データの長さを返す。

resize(int)
　格納データの長さを変更する。

clear()
　格納データの長さを変更する。

また、データそのものへのアクセスは (有難いことに) 配列とまったく同じ [] 演算子を用いることで可能である。次のプログラムは、arrayInt に 1 から 100 を、arraySum[i] に $1+2+\cdots+i$ を格納するサンプルである。

```
#include <vector>

void main()
{
        vector<int>     arrayInt;
        vector<int>     arraySum;
        int             i;

        for(i = 0; i < 100; i++) {
                arrayInt.push_back(i);
        }
        arraySum.resize(100);
        arraySum[0] = arrayInt[0];

        for(i = 1; i < 100; i++) {
                arraySum[i] = arraySum[i-1] + arrayInt[i];
        }
}
```

今回のサンプルではデータ型として int を利用したが、もちろんユーザが定義したクラスのオブジェクトやポインタ型の変数も格納可能である。

4 リンクリストクラス

4.1 リンクリストクラスの概念

配列やベクトルクラスを扱う場合の難点に、途中要素の挿入や削除がある。配列やベクトルの中間部分に要素を挿入する場合、挿入箇所より後ろのデータを全て一つずつずらす必要がある。削除の場合も同様に前にずらさなければならない。このことが、無視できないパフォーマンスの低下を招く場合、別のデータ構造を検討する価値がある。

リンクリスト は、要素の操作や削除に非常に関して大変良い性能を発揮する。リンクリストでは、管理したいデータに対しそれぞれ「次の要素への参照先」という情報を付加させて置く。概念的には次の図 1 のようなものとなる。

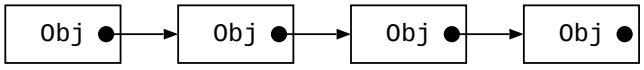

図 1: リンクリストの概念図

この「次の要素への参照先」という情報を利用し、順番に要素にアクセスする手段が提供されることになる。このようなデータの保持方法を用いることにより、中間箇所に対しての要素挿入が図

2 のような操作によって行うことができる。

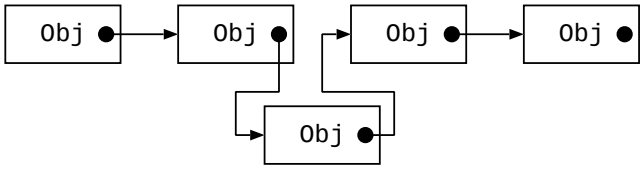

図 2: リンクリストへの要素挿入

逆に、途中要素の削除は図 3 のようにして行うことができる。

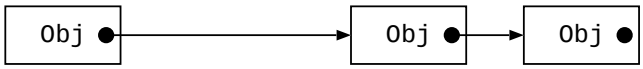

図 3: リンクリストに対する要素削除

図で示したように、挿入や削除に対する操作は、対象要素の前後にしか及ばないため、非常に少ない操作で可能であることがわかる。

この構造では、ある要素の前に位置する要素を参照することが、単純にはできないという問題点がある。これは、次の図 4 のように「前の要素への参照先」という情報を併せて持ち合わせることで解決できる。

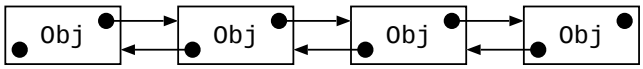

図 4: 前要素へのリンク情報

このように、前後の両方向に参照先があるリンクリストのことを**ダブルリンクリスト** と呼ぶ。それに対し、最初に紹介したような一方向のみのリンクリストのことを **シングルリンクリスト** と呼ぶ。一般的に、あえてシングルリンクリストを選択する理由はほとんどない。実際、C++ や Java のリンクリストクラスはダブルリンクリストを採用している。

4.2 リンクリストクラスの特徴

リンクリストクラスは、次のような長所を持っている。

- 伸縮自在である。
- あらゆる箇所への挿入、削除が非常に高速に行える。
- 前後双方への逐次アクセスが高速に行える。

一方、短所としては次のようなものがある。

- 添字番号によるランダムなアクセスができないか、あるいはできてもかなり効率が悪い。

- 配列としてデータを保持するわけではないので、メモリ管理の効率はベクトルクラスよりも劣る。

リンクリストクラスの機能そのものは、ベクトルクラスのそれと多くの場合互換性を持つ。異なるのは、それぞれのパフォーマンスである。従って、管理したいデータの利用想定によって使い分けることができる。次のようにして採択すればよいだろう。

1. データがある程度大規模となり、かつ中間箇所への挿入、削除が頻繁になると予測されるなら、リンクリストを用いる。
2. 添字番号によるランダムアクセスの速度が重要になりそうなら、ベクトルを用いる。
3. 上記の双方の特徴を持つのであれば、イテレータ等を用いていつでも変更が可能となるようなコード作りを心掛け、実際にベンチマークを計って決定する。
4. あまりパフォーマンスを気にする必要がなさそうならば、ベクトルの方を用いるのが無難である。

4.3 C++ でのリンクリストクラス

C++ では、リンクリスト用のテンプレートとして「list」が利用できる。形式次のようなものである。

```
list<データ型名>　コンテナ変数名;
```

リンクリストクラスの作成方法は、ベクトルクラスの場合と同様である。リンクリストクラスでは、次のようなメンバ関数がベクトルクラスと同様に利用可能である。(「TYPE」はリンクリストクラスの格納データ型を指す。)

push_back(TYPE)
　末尾に要素を追加する。サイズは自動的に 1 大きくなる。

pop_back()
　末尾の要素を削除する。サイズは自動的に 1 小さくなる。

size(void)
　現在の格納データの長さを返す。

clear()
　格納データの長さを変更する。

さらにリンクリストでは、先頭に対する追加「`push_front(TYPE)`」と削除「`pop_front()`」も利用できる。

ベクトルクラスの場合、配列と同様な方法で各要素に対してアクセスすることができたが、リンクリストクラスの場合はイテレータを用いる必要がある。まず、リンクリストのイテレータを作成する。イテレータの作成形式は以下のようなものである。

list<データ型名>::iterator　イテレータ名;

また、イテレータをコンテナから得るには「begin()」というメンバ関数を用いる。具体的には、以下のような記述となる。

```
        list<int>               listInt;
        list<int>::iterator     listIterator;

listIterator = listInt.begin();
```

イテレータは、C++ においてはあたかも要素を指すポインタ変数のように挙動する。つまり、要素にアクセスするにはアスタリスクを用い、次の要素を指すには「++」演算子を用いる。終端に至ったかどうかは、コンテナの「end()」メンバ関数の値と同じかどうかを比較することになる。

次のプログラムは、リンクリストに 1 から 100 までを保存し、それをイテレータを用いて一つずつ表示するものである。

```
#include <list>
#include <iostream>

void main()
{
        list<int>               Container;
        list<int>::iterator     Iterator;
        int                     i;

        for(i = 1; i <= 100; i++) {
                Container.push_back(i);
        }

        for(Iterator = Container.begin();
            Iterator != Container.end(); Iterator++) {
                cout << "Num = " << *Iterator << "\n";
        }
}
```

要素の挿入と削除には、イテレータを用いる。コンテナクラスには、次のメンバ関数が用意されている。ここで、「TYPE」はコンテナクラスの格納データ型を、「ITE」はコンテナクラスのイテレータを指す。

insert(ITE, TYPE)

イテレータが示す位置の直後に要素を挿入する。

insert(ITE, int num, TYPE)

イテレータが示す位置の直後に要素を num 個挿入する。

erase(ITE)

イテレータが示す位置の要素を削除する。

find(TYPE)

与えられたデータがコンテナに格納されている場合、その箇所を示すイテレータを返す。もし格納されていなかった場合は end() と同じ値を返す。

例えば、次のコードはコンテナ「Container」の最初から 10 番目にある要素を削除する。

```
Container.erase(Container.begin() + 9);
```

また、次のコードは Container 中から「7」を全て削除する。

```
while((Iterator = Container.find(7)) != Container.end()) {
        Container.erase(Iterator);
}
```

5　ハッシュマップ、ハッシュセット

5.1　ハッシュマップ、ハッシュセットの概念

コンテナクラスによって格納されているデータから、あるキーワードでデータを検索したいとする。このとき、ベクトルクラス、リンクリストクラスのいずれの場合でも先頭から比較していくことが必要となる。これは、格納データが多数となる場合にはかなり時間のかかる処理となることがある。また、一般的に検索作業には非常に高速な処理が求められることが多い。

このような用途に「**ハッシュテーブル**」というデータ構造が役立つ。ハッシュテーブルには、要素の順番という概念がない。しかし、非常に高速にデータの格納と検索を実現する。ハッシュテーブルは、「**ハッシュコード**」という整数値をオブジェクトごとに計算する。

この、ハッシュテーブルを利用して構築されるコンテナが「**ハッシュマップ**」である。ハッシュマップは、概念的には次の図 5 のようなものである。

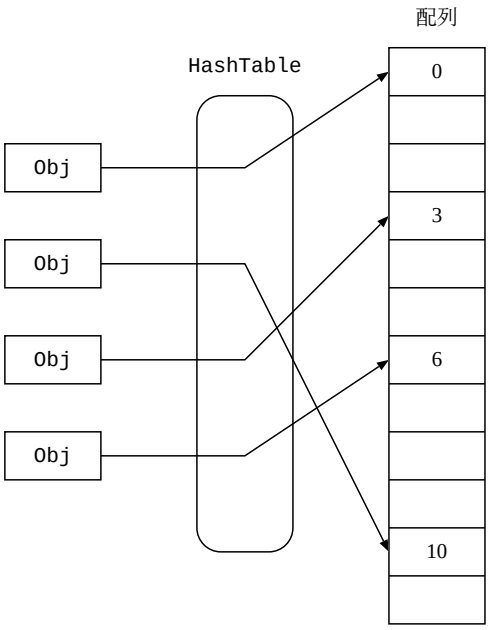

図 5: ハッシュマップの概念図

まず、ハッシュテーブルによって各オブジェクトのハッシュコードを生成し、そのハッシュコードを添字として配列内に格納する。これにより、コンテナ内のデータの有無は単純にハッシュテーブルの計算時間に依存することとなる。一般的に、ハッシュテーブルは大変高速に計算できるように設計されるので、ベクトルやリンクリストを用いるよりもはるかに高速に検索作業が可能となる。

ここで、もし異なるオブジェクトが同一のハッシュコードを出力するような場合はどうすればよいか？ これは理論的に十分ありえる話である。なぜなら、格納する配列のサイズは無限にとるわけにはいかず、少なくとも格納したいデータ個数の方が格納する配列よりも大きい場合には重複が避けられない。これには大きく次の三種類の回避方法が用いられる。

- 格納配列内のデータ構造をリンクリストにしておき、一つのハッシュコードで複数のデータが格納できるようにしておく。
- 格納配列内の、まだ空いている箇所に格納する。
- 格納配列の大きさを拡張し、新たなハッシュテーブルを生成する。

理論的には一番目が最も効率が良く合理的だが、リンクリストが長くなりすぎると検索時間に影響を及ぼす。そこで、実践的には一番目と三番目を組み合わせて構築されることが多い。

ハッシュセットは、ハッシュマップが「キー + 値」という組み合わせでデータが表現されいるのに対し、値の部分が存在しないものと考えれば良い。

5.2 ハッシュマップ、ハッシュセットの特徴

ハッシュマップクラス及びハッシュセットクラスは、次のような長所を持っている。

- 伸縮自在である。(むしろ、データの大きさという概念がない。)
- 要素の挿入、削除、検索が非常に高速に行える。

一方、短所としては次のようなものがある。

- 格納データの中で順番という概念を持たせることができない。
- データを格納する効率は悪い。

ハッシュマップクラスは、機能面だけで見れば最強のコンテナクラスである。キーはどのような型を持つこともでき、参照時間もごくわずかという特性を持っている。従って、このデータ構造をついつい多用してしまいたくなる。しかし、ハッシュマップは非常にメモリを必要とする場合があることと、ハッシュテーブルの再構築にはそれなりに負担がかかるが無視できない。従って、ベクトルクラスやリンクリストクラスを用いた方が結果的にパフォーマンスが向上することも少なくない。

一方、ハッシュセットクラスはある意味非常に利用用途が限られるクラスである。しかし、データの有無のみが問題となる場面では非常に有効なデータ構造である。

5.3 C++ でのハッシュマップクラス

C++ では、ハッシュマップを実現するテンプレートとして「map」が利用できる。形式は次のようなものである。

```
map<キー型名, 値型名>  コンテナ変数名;
```

よくあるパターンとして、キーには string 型の文字列が用いられることが多い。C++ では、キーに配列演算子を用いて指定することができる。次のプログラムは、キーとして string 型を、値として整数を持つようなサンプルコードである。

```
map<string, int>        mapInt;
string                  str;

str = "orange";
mapInt["apple"] = 10;
mapInt[str] = 20;

cout << mapInt[str] << "\n";
```

もし、指定したキーでハッシュマップ内にデータが存在しなかった場合は、新たに格納データが生成される。既に同一キーのデータが存在していた場合は、前のデータを上書きする。

map テンプレートが持つメンバ関数には、以下のようなものが特に有用である。

find(key)

key が示すデータのイテレータを返す。もし key が示すデータがコンテナ内に存在しない場合、end() 関数の返り値と同じ値を返す。

begin()
　イテレータを返す。

end()
　イテレータの終端値を返す。

size()
　現在の格納要素数を返す。

erase(key)
　キーが表す要素を (もしあれば) コンテナ内から削除する。

erase(iterator)
　イテレータが表す要素をコンテナ内から削除する。

erase()
　コンテナ内の全要素を消去する。

あるキーを持つデータが既にコンテナ内に存在するかどうかを判別するには、find() 関数を用いる。この関数は、もしデータがコンテナ内にあればそのイテレータを返し、なければ end() 関数の返り値と同じ値を返す。従って、データの有無の検索は以下のようにすればよい。

```
map<string, int>        map;

if(map.find() != map.end()) {
        // 要素があった場合
} else {
        // 要素がなかった場合
}
```

一方、ハッシュマップ内の全データを順次処理していくには、リンクリストクラスと同様にイテレータを用いる。基本的にはリンクリストクラスと利用方法は変わらないが、キーが「first」、データが「second」という要素に入っている点が異なっている。次のプログラムは、ハッシュマップ内の全データを出力するものである。

```
map<string, int>                map;
map<string, int>::iterator      p;
                :
                :
                :
for(p = map.begin(); p != map.end(); p++) {
        cout << "Key = " << p->first << "\n";
        cout << "Value = " << p->second << "\n";
}
```

ハッシュセットを表すテンプレートは「**set**」である。これの利用方法はほとんどハッシュマップと同様なので割愛する。

6 サンプルプログラム

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3: #include <FL/Fl.h>
 4: #include <FL/Fl_Input.h>
 5: #include <FL/Fl_Multi_Browser.h>
 6: #include <FL/Fl_Window.h>
 7:
 8: // (VC++ で開発する人は下の 2 行のコメントを外す。
 9: // #define snprintf _snprintf
10: // using namespace std;
11:
12: static bool     inputStatus;
13:
14: // input に入力があった場合に更新される。
15: void status_change(Fl_Widget *w, void *v)
16: {
17:     inputStatus = true;
18: }
19:
20: int main(int argc, char *argv[])
21: {
22:     Fl_Window           win(300, 400, "STL TEST");
23:     Fl_Multi_Browser    *browser = new Fl_Multi_Browser(10, 10, 280, 340);
24:     Fl_Input            *input = new Fl_Input(10, 360, 280, 30);
25:     char                buffer[512];
26:     int                 i, num;
27:
28:     // 入力用テキストボックス
29:     inputStatus = false;
30:     input->when(FL_WHEN_ENTER_KEY | FL_WHEN_NOT_CHANGED);
31:     input->labelsize(12);
32:     input->textfont(FL_COURIER_BOLD);
33:     input->callback(status_change);
34:
35:     // 出力用テキストエリア
36:     browser->labelsize(12);
37:     browser->has_scrollbar(Fl_Browser_::BOTH_ALWAYS);
38:
39:     win.end();
40:     win.show();
41:
42:     for(i = 0; i < 100; i++) {
43:         snprintf(buffer, 511, "i = %d", i);
44:         // 出力用テキストエリアに文字列を追加
45:         browser->add(buffer);
46:     }
47:
48:     while(true) {
49:         if(win.visible() == 0) {
50:             if(Fl::wait() == 0) {
51:                 break;
52:             } else {
53:                 continue;
54:             }
55:         }
```

```
56:         if(Fl::check() == 0) break;
57:
58:         if(inputStatus == true) {
59:             // 入力用テキストボックスに入力があった場合。
60:             inputStatus = false;
61:
62:             // 入力を整数に変換
63:             num = atoi(input->value());
64:             snprintf(buffer, 511, "num = %d", num);
65:             browser->add(buffer);
66:             browser->bottomline(99999999);
67:         }
68:     }
69:
70:     return 0;
71: }
```

7 問題

(1) 最初の「i = ～」が出力する部分を省き、vector コンテナを用いて入力された数値が保存されるように変更せよ。

(2) 「sum」という文字列が入力された場合、これまで入力された数値の合計を出力するように拡張せよ。

(3) (1), (2) の機能を list コンテナを用いて実現せよ。

(4) (3) に対し、「del」という文字列を入力した場合に、もし同じ数値の要素があった場合は最初の要素以外は削除するように拡張せよ。