

[prog2] Programming C++ (C6) Exercise Guide (Ex03)

10/12, Thursday 3rd period.

Ex03 について

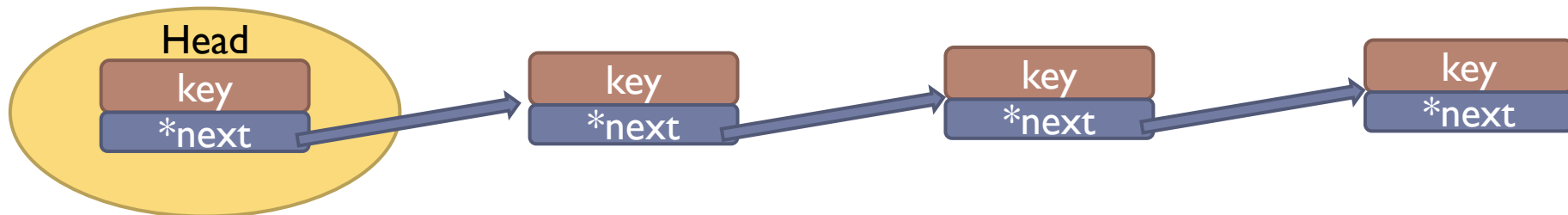
▶ list

連結リストを簡単に実装できる道具 とでも思えばよい

- **vector** と同じように STLコンテナの1つ

[ProgC/alg1 まで]

- 構造体に**データ(key)**と**自分自身のアドレスを持つポインタ(*next)**を定義する
- nextに、次に繋げたい構造体のアドレスを格納する
- nextを使って辿っていける状態にすることでリスト構造を表現



list (クラス)を使う

- ・ list は クラス で定義されているものだが、とりあえず今は深く考えない（後半で、詳しく扱う）⇒ 道具としての利用方法

■ 使い方 ※ <list> をincludeすること

int 型 の値をキーとするリストを持ちたいとき: `list<int> vec;`

double 型の値をキーとするデータを持ちたいとき: `list<double> vec;`

`list<保持する型の名前> 変数名;`

- ・ 定義してあれば、構造体のlistという構成も可能
⇒ STLコンテナの仲間ということで、定義はvectorと全く同じ

vector が解ってしまえば、
list も やってることは同じでしょ？

list (クラス)を使う

list<int> ls; を例に...

■ おさえておきたい機能

ls.push_back(x); // int型の変数 x を list に追加

ls.begin(); // “ls”という名前のlistの先頭の位置

ls.end(); // “ls”という名前のlistの末尾の位置

イテレータに関しては

list<int>::iterator it; のように宣言。(結局、これもvectorと同じ)

list (クラス)を使う

list<int> ls; を例に...

ls.push_back(x); のイメージ

便宜上、*next を書いているが、
listコンテナを使えば、
新しいノードの確保や解放、リスト間の接続
([関節参照](#))は自動的に行われるので
プログラマにとっては、気にする必要が無い

list <int> ls; 空っぽ


ls.push_back(1);



ls.push_back(-3);



ls.push_back(6);



ls.begin();



ls.end();



list で気を付けること その1

list はシーケンシャルアクセス方式、

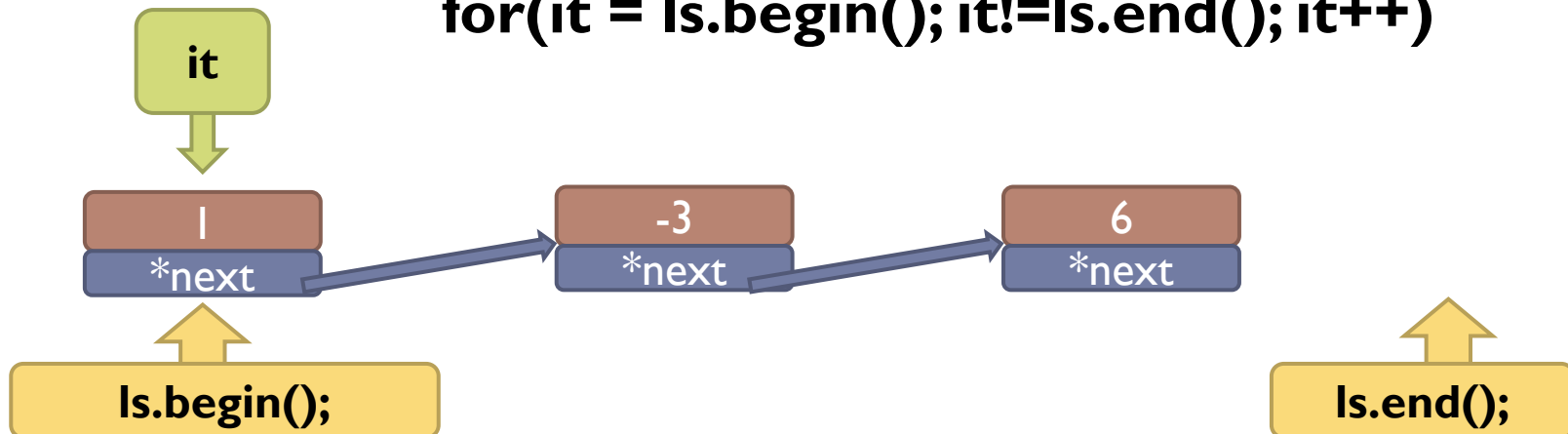
vector はランダムアクセス方式。つまり作成したリストに対して

ls[3]; のように、要素を指定するアクセスが不可能！！

⇒ list へのアクセスは、全てイテレータで実行する

`list<int>::iterator it;`

`for(it = ls.begin(); it!=ls.end(); it++)`

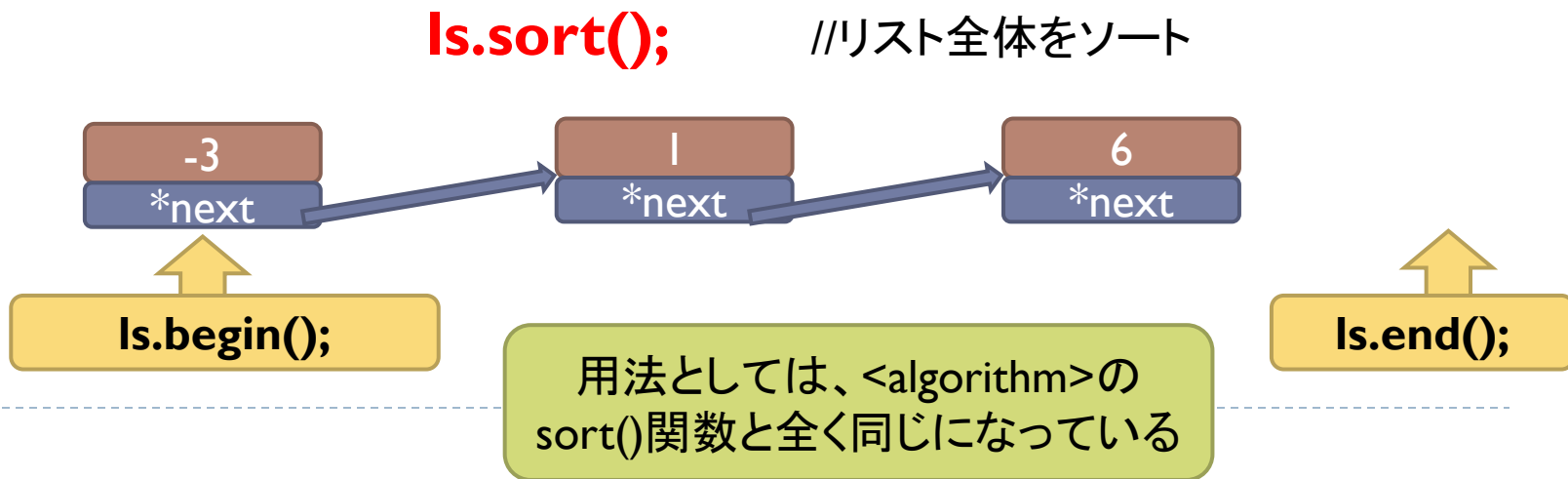


list で気を付けること その2

list はシーケンシャルアクセス方式をとることから、
listの内容をソートをしたいときに

<algorithm> の sort()関数が使えない！！

⇒ <algorithm> の sort()関数は、**ランダムアクセス方式の構造向け**
⇒ list には、専用のソート関数 `list::sort()` が用意されている



例題 1

list サンプル（ストリームからきた入力を list へ push）

```
1  #include <iostream>
2  #include <list>           // listを使う
3
4  using namespace std;
5
6  void Listprint(list<int>); // listの内容を全部表示する
7  list<int>::iterator Finditem(int, list<int> &); // listから、指定の1つの値を存在するか調べる
8
9
10
11 int main(){
12     int input;
13     list<int> ls; //int型の要素を持つlist
14     list<int>::iterator itr; // listを参照するためのイテレータ
15
16     cout << "Input insert numbers." << endl;
17     while(cin >> input){
18         ls.push_back(input); // vectorと同じように push_back()で末尾へ挿入できる
19     }
20     cin.clear(); // cinの失敗（終了）情報を消去（あとでまたcinを使うため）
21
22     Listprint(ls); //listの内容を全部表示
23 }
```

※ 入力をやめない限り、無限ループするので Ctrl+D で入力を終了すること

例題 1

list サンプル (listの中身を表示)

```
59
60  /*
61  listを1つ渡して、最初から順番にアクセスすることで、全要素を列挙する
62  listの場合は listdata[i] などの表記は不可。
63  ⇒データ構造上、この書き方では処理が遅くなるので敢えて実装されていない
64  */
65  void Listprint(list<int> listdata){ ← STLコンテナも関数の引数
66                                     に利用できる
67      cout << "[List contents]";
68      list<int>::iterator itr; //イテレータでアクセスする
69      for(itr=listdata.begin();itr!=listdata.end();itr++){
70          if(itr!=listdata.end()) cout << " -> ";
71          cout << *itr; // *itr でイテレータが指す内容を参照できる
72      }
73      cout << endl;
74  }
```

ランダムアクセスができないため、イテレータを用いた処理が必要。
コードに何度も書くと汚くなるので、さっさと関数化するのが吉。

例題 1

list サンプル (ソート)

```
24
25     cout << endl << "Apply sort algorithm..." << endl;
26     /*
27     <algorithm>のソート sort()はランダムアクセスできる (a[5]等と書いて直接アクセス) データ構造用。
28     リスト構造ではシーケンシャルアクセスであるため使用不可 (リストを順番に辿って要素を探す)
29     そのため、list用に提供されている専用のメンバー関数 sort() を使う。
30     */
31
32     ls.sort(); //リストのソートはメンバー関数 .sort()を用いる。
33     Listprint(ls); //ソート後の状態
34
```

リストの名前.sort();でそのソート全体を整列する。

Listprint(ls); (さっき定義した関数)を使うと、結果は確認できる

例題 1

list サンプル (要素を消す)

```
35
36 //指定要素を消す
37 cout << "Input delete numbers." << endl;
38 while(cin >> input){
39     itr = Finditem(input,ls); //入力データがlistにあるか探す
40
41     if(itr == ls.end()) cout << "Not found such number, "<< input << " ." <<endl; //無ければ消せない
42     else{//あったら消せる
43         ls.erase(itr); // erase()で消すには、消したい場所のイテレータを与える
44         cout << input << " was deleted from the list." << endl;
45         Listprint(ls);//消した後の状態
46     }
47
48     if(ls.empty()){//listが空になったらループを抜ける
49         cout <<"The list became empty. Bye." << endl;
50         break;
51     }
52 }
53
54 return 0;
55 }
```

リストの名前`.erase(イテレータ);`でリスト内の指定のノードを削除できる。

⇒ 戻り値は 消したノードの次のノードを指すイテレータ

⇒ 消したノードのメモリも自動的に解放される

例題 1

list サンプル（指定要素を検索する）

```
78  /*
79  指定の要素 key が list中にあるかどうか
80  あれば、そのkeyのイテレータ、無ければ endの位置が返ってくる
81
82  listは間接参照にしておかないと、データの中身は同じでも、
83  main関数のlistと別のメモリ空間に置かれているため
84  イテレータが正しく対応せずに、main関数で*itrを参照したときにバグる
85  （アドレスの値を書き換える操作を行う訳ではないので、 & で十分）
86  */
87
88  list<int>::iterator Finditem(int key, list<int> &listdata){
89
90      list<int>::iterator itr;
91
92      for(itr=listdata.begin();itr!=listdata.end();itr++){
93          if(*itr == key) break;
94      }
95      return itr; //見つけた場所を返す
96  }
```

関数に対してアドレス
(イテレータ)を引数にする
ときの方法は2種類あり、

C言語のように * を使うもの
と、 & を使うものある

関節参照で、値の書き換え
も行う場合は * で、
読み込みしか行わない場合
は & を使うとミス・バグが減
らせる。

これも、C言語のリスト実装とよく似ているが、リストのイテレータをやりとりする方法に注意

構造体データをソートする(問題 1)

- ▶ 構造体をvector / list に入れて、
コンテナ全体をソートする

⇒ int や string型の vectorであれば、
型がソートのルールは明らか
(数字が大きい方、アルファベット順、etc...)

```
typedef struct{  
    string name;    //名前  
    int age;        //年齢  
    int marriage;   //結婚歴  
    int nenkin;     //年金受給額  
}Senior;
```

```
vector<Senior> data;
```

構造体などの、データの集合体では「どうソートしてよいか」

コンパイラが解釈できずに、エラーとなる ⇒ **ルールを自分で決めろ**

//以下のようなsort呼び出しはエラーとなる

```
// sort(data.begin(),data.end());
```

// ⇒ Senior構造体には「順序関係」を示すルールが定義されていないため、ソートできない

構造体データをソートする(問題 1)

- ▶ 自分でソートルールを定義する
- ▶ 2つの要素を比較して何らかの順序関係を規定する関数を作る

- ▶ 引数はもちろん、比較させたいデータの「型」

```
//ソートのルールを与える関数 (sort呼び出し時に使用)
bool sort_name(const Senior&, const Senior&);
bool sort_age(const Senior&, const Senior&);
bool sort_marriage(const Senior&, const Senior&);
bool sort_nenkin(const Senior&, const Senior&);
```

```
/*
   ソートルール指示の実体
   構造体を渡して、この関数内にルールを書く
*/
bool sort_name(const Senior& x,const Senior& y){
    return x.name > y.name;
}

bool sort_age(const Senior& x,const Senior& y){
    return x.age > y.age;//2つを比べて、年齢が高い方を
}

bool sort_marriage(const Senior& x,const Senior& y){
    return x.marriage > y.marriage;//2つを比べて、結婚歴の長い方を
}

bool sort_nenkin(const Senior& x,const Senior& y){
    return x.nenkin > y.nenkin;
}
```

この形で書いておけば、
後はsort()関数の引数に渡すだけなので
細かい部分はあまり気にしなくてよい

構造体データをソートする(問題 1)

▶ ソートのルールを記述した関数の名前を、`sort()`の第3引数に与える

⇒ 関数内に記述されたルールによって、ソートを実行する

```
case 0://名前でソート
    cout << "Name";
    sort(data.begin(),data.end(),sort_name);
    break;

case 1://年齢でソート
    cout << "Age";
    sort(data.begin(),data.end(),sort_age);
    break;

case 2://結婚歴でソート
    cout << "Marriage";
    sort(data.begin(),data.end(),sort_marriage);
    break;

case 3://年金受給額でソート
    cout << "Nenkin";
    sort(data.begin(),data.end(),sort_nenkin);
    break;
```

list の場合も使い方は全く同じで
ls.sort(sort_name);
のように、ソートルールを定義した関数の名前を引数に追加するだけ

構造体に対するイテレータ(念の為)

- ▶ `Senior *p;`

という構造体のポインタ型に対して、
`p = &data;` とすると関節参照としては

- ▶ `p-> name;` //アロー演算子

- ▶ `(*p).name;`

で参照できた。イテレータによる操作でもこれは同じなので、
`list<Senior>::iterator it;` が与えられているとき、

- ▶ `it->name;`

- ▶ `(*it).name;`

のようにして同様に参照できる。