第5章「シーケンシャルコンテナとstringの解析」

主なトピック

- list型
- イテレータ（反復子）：ポインタの機能を拡張したもの

2種類のデータアクセス法

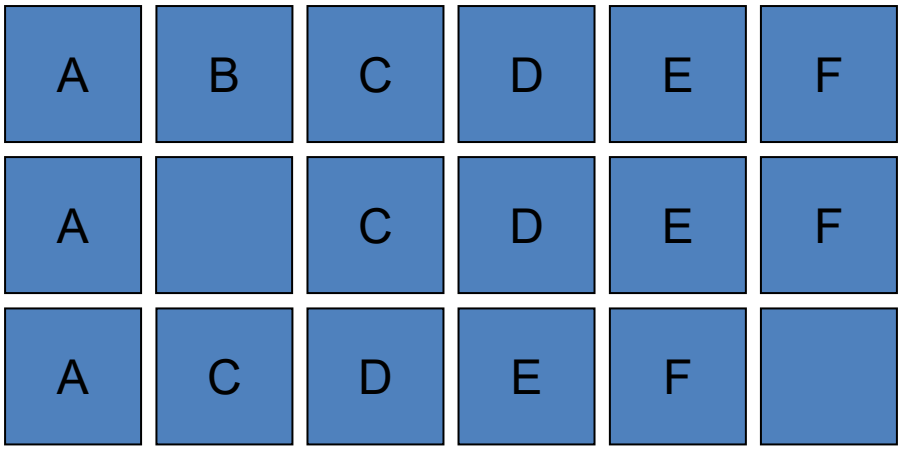
- ランダムアクセス
  - いろんな場所に一発でアクセスできる
  - 配列やvector型, data[i]
- シーケンシャルアクセス
  - 先頭から順番にしかアクセスできない
  - list型，ループとイテレータ（反復子）を使ってアクセス

vector型とlist型の特徴

- 両方ともコンテナ
  - 同じようにデータの追加が可能
- vector型
  - 要素へのアクセスが添え字を使える
- list型
  - 要素の追加や削除が高速

要素を削除するときのイメージ

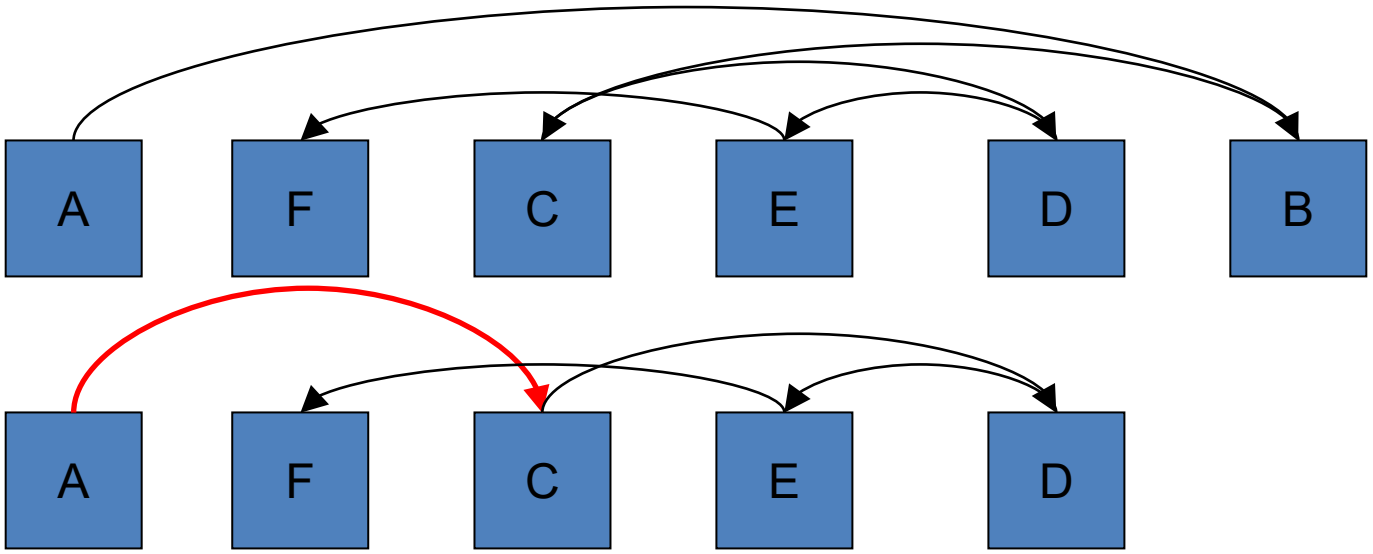
vector: メモリ上に順に並んでいる（線形メモリ空間）



- データを詰める作業が必要
- データ数に依存した処理（データが多いときに前の方のデータを削除すると処理に時間がかかる）
- データの追加も同様

要素を削除するときのイメージ

list: 各データはメモリ上にバラバラに存在
各データが次のデータが何かを格納している



- データの接続だけを変更
- データ数に依存しない処理
- データの追加も同様に高速

list型

- 使い方は，vectorと共通なものもある
  - 変数の宣言
  - データの追加
- しかし，まったく異なる使い方もある
  - ループ
  - データへのアクセス
- ちょっとだけ違う使い方もある
  - ソート

list型のサンプルソース

```
#include <list>
//  前回の授業で定義した構造体 Student_info read 関数compare関数を使う
int main()
{
   list< Student_info > students;
   Student_info record;
   while( read( cin, record) ){
        students.push_back( record );
   }
   students.sort( compare );
   list< Student_info >::iterator iter = students.begin();
   while( iter != students.end() )    {
        //  画面にIDとFirstName, LastNameを出力
        cout << (*iter).ID << " " << (*iter).FirstName << " "
                << (*iter).LastName << endl;
        ++ iter;
   }
}
```

list型のサンプルソースの解説

- #include <list>
  - list型を使うときに必要なヘッダーファイル
- list< Student_info > students;
  - 変数の宣言
  - list<型名> 変数名;
- students.push_back(record);
  - データ(record)をlist型の変数 students の末尾に追加
- ここまではvector型と同じ操作法

list型のサンプルソースの解説

- students.sort(compare);
  - リスト型のデータのソート
  - list<double> のように型の大小関係が自明のときは compare 関数を省略することができる
  - vector型のときは，sort(students.begin(), students.end(), compare)
    - 上のように3つの引数をとる sort 関数を利用できるのはランダムアクセス型の変数のとき
    - vectorの他には，string型など
- vector型のときは for 文（インデックス）が使えた
- list型では 添え字は使えない，その代わりにイテレータ（反復子）と while 文を使

list型のサンプルソースの解説

- list< Student_info >::iterator iter = students.begin();
  - list< Student_info >::iterator という型の iter という名前の変数を宣言
  - iter に students の先頭要素であるstudents.begin() で初期化
  - この iter という変数がイテレータ（反復子）
- while(iter != students.end()) {}
  - iter が students.end() と異なる間ループを実行

list型のサンプルソースの解説

- cout << (*iter).ID
  - cout に イテレータ(iter)が指しているデータ（Student_info型の変数）のIDを出力する
  - 通常の構造体の変数だと record.ID
  - イテレータ（反復子）の場合は (*iter).ID
- ++ iter;
  - イテレータ（反復子）をインクリメント，次の要素に進む
- ポインタによく似ている

イテレータ（反復子）と添え字（インデックス）

- イテレータ（反復子）は添え字（インデックス）に似ているところもある
  - ++ iter; // 整数を1増やすのに似ている
- しかし，違うところもある
  - iter = students.begin();
    - 上の文は，studentsに蓄えられた先頭を指し示す，整数を代入しているわけではない
  - (*iter).ID
    - (*iter)とするとイテレータ（反復子）がデータを指すことになる
    - 配列やvectorだと，students[i]
- ポインタによく似ている

イテレータ（反復子）とは

- コンテナに含まれている要素を指し示すもの
  - 要素そのものではない
    - cout << iter.ID とはできない
  - 要素にアクセスするときは，イテレータの先頭に＊を付ける
    - cout << (*iter).ID なら大丈夫
    - または，cout << iter->ID と書いてもOK
  - 次の要素を指したいときは，イテレータをインクリメント
    - ++ iter
- イテレータはポインタに，新しい機能（抽象化・一般化，機能整理）を付加したもの

正確に言うと

- これまでは，students.begin()は students の先頭要素を示すと言っていた
- 正確には，studentsの先頭の要素を指すイテレータ
- 同様に students.end() は最後の要素の一つ後を指すイテレータ
- 繰り返しになるが
  - イテレータは要素そのものではない
  - しかし，要素を直接扱ってるように手軽に扱える

イテレータの種類

- list< Student_info >::iterator iter;
  - 通常のイテレータ,
  - 読み書き可能（イテレータが指す要素の内容を書き換えできる）
    - iter-> ID = “s0001”;
- list< Student_info >::const_iterator iter;
  - コンスト・イテレータ
  - 読み込みのみ可能（内容を書き換えは不可）
    - iter-> ID = “s0001”; とするとエラーになる

文字列処理

- string型はvector型と同じようにループを組むことができる

```
#include <cctype>
string s = “Sample text”;
for( string::size_type i = 0; i != s.size(); ++i ) {
    //  文字列sを各文字単位で小文字に変換して出力
    cout << tolower( s[ i ] );
}
```

- tolower(c)はcを小文字に変換する関数
  - #include <cctype>が必要

文字列処理

- s.substr(i, j);
  - i から始まり j 文字分の新しいstringを生成
  - 先頭を1文字削るには
    - string tmp = s.substr(1, s.size() -1);
  - 末尾を1文字削るには
    - string tmp = s.substr(0, s.size() - 1);

演習課題5のポイント

- 構造体を作る，例えば
```
struct WordCount {
    string Word;
    int Count;
};
```
- 標準入力ストリームから読み込んだ文字列をテキスト処理
- その後WordCount型に格納してから，list型に保存する
- そして，ソートと出力

演習課題5のポイント

```
string s;
list<WordCount> words;
while(cin >> s)        {
    //  sの文字列処理をここに入れる
    int isFound =0;
    list<WordCount>::iterator iter = words.begin();
    while( iter != words.end() )      {
        if( (*iter).Word == s) {
            isFound = 1;
            //   回数を1増やす処理を入れる
            break;
        }
        ++iter;
    }
    if(isFound == 0)   {
       //   データをwordsに追加する処理を入れる
    }
}
```