第7章「連想コンテナを使う」

主なトピック

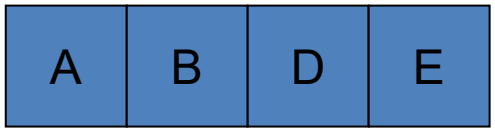
- map型（あるいはKey-Value型，連想記憶）

２種類のコンテナ

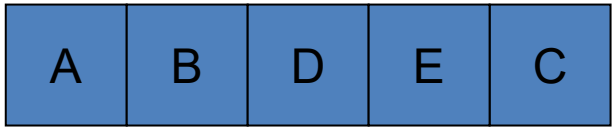
- シーケンシャルコンテナ
  - 列（シーケンス）になって格納
  - push_back関数などでデータを挿入する位置を指定可能
  - vector型，list型
- 連想コンテナ
  - データを追加するときに，データの値に応じて自動的に格納位置が決まる
  - map型

データの追加のイメージ

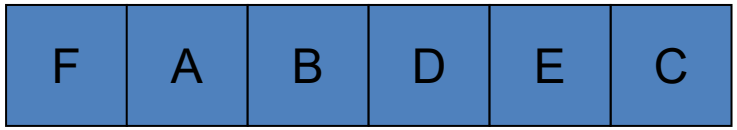
シーケンシャルコンテナ
vector型 c



c.push_back(“C”);

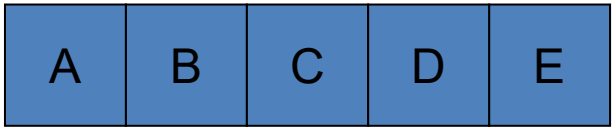


c.push_front (“F”);

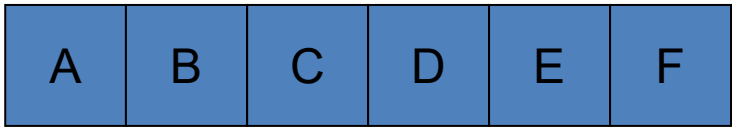


連想コンテナ m

mに“C” を追加



mに “F”を追加



連想コンテナにおけるデータの表現

- 連想コンテナでは，データの格納位置が自動的に変化
- では，どうやってデータにアクセスする？
- データは，キー(Key)と値(Value)の組として表現
  - 例えば，前回までの課題だと
  - 単語(“C++”) と出現回数(10)
  - キー (“C++”)と値(10)
  - “C++”という文字列から10という整数を「連想」させる

連想コンテナにおけるデータの表現

- シーケンシャルコンテナでは,
  - 「添え字」（データの格納位置）を使ってデータにアクセス
  - イテレータを使って先頭から順番にアクセス
  - vector< WordCount > w;
- 連想コンテナでは,
  - キー (“C++”)を使って，値(10)にアクセス
  - イテレータを使って先頭から順番にアクセス
  - map<string, int> m;

map型

- 単語（string型）と出現回数（int型）の連想コンテナ
- map< string, int > counters;
  - map型の変数の宣言
  - map< キーの型，値の型 > コンテナの変数名;
- counters[“c++”]
  - map型の変数にアクセス
  - コンテナ変数名[キー]
  - ++ counters[“c++”]; 出現回数を1増やす
  - cout << counters[“c++”]; 出現回数を出力

map型のサンプルソース

```
//   標準入力から単語を読み込み，その出現回数を1増やす
map< string, int > counters;
string s;
while( cin >> s ) {
   ++ counters[ s ];
}
//  countersに格納された，すべての単語とその出現回数を出力
for( map< string, int >::iterator iter = counters.begin();
   iter != counters.end(); ++iter)  {
   cout << iter -> first << “¥t” << iter -> second << endl;
}
```

サンプルソースの解説

- counters[s]
  - mapでは，あるキーでアクセスした瞬間にその値が初期化される.
  - この場合は，int型のデフォルトの 0 で初期化
  - そのため，キーが出現済みかどうか確認する必要はない
    - 参考：前回の演習のvector< WordCount >の場合
- ++ counters[s]
  - counters[s] = counters[s] + 1と同じ

サンプルソースの解説

- map< string, int >::iterator iter;
  - map用のイテレータの宣言
- iter = counters.begin();
  - countersの先頭の要素
- iter != counters.end()
  - イテレータiterがcountersの最後の要素と違う
- ++ iter
  - イテレータを進める（次の要素に移る）

サンプルソースの解説

- iter -> first, iter -> second
  - この連想コンテナは，単語(string)と出現回数(int)の組を格納
  - 実際は string型と int型の pair (組)として格納
  - この場合は，pair< const string, int >
  - 一般に，pair< const K, V >
    - K: キーの型，V: 値の型
  - キーの型がconstなのは，一度登録したキーを途中で変更されないように保護するため

サンプルソースの解説

- mapのイテレータはpairを指している
- iter -> first
  - pairの第1要素，キー（文字列型の単語）
  - (*iter).first　としても良い
- iter -> second
  - pairの第2要素，値（int型の出現回数）
  - 同様に(*iter).second

連想コンテナの原理

- どうして，添え字でなくて，キー（文字列）でデータにアクセスできる？
- ハッシュ
  - mapの内部で，キー（文字列）から添え字（のようなもの）に変換し，実際はその添え字で配列のようなデータにアクセス
  - キー（文字列）から添え字への変換法が重要
    - 違うキーは，違う添え字に変換すべき
    - 細かく分けすぎると，配列のサイズが大きくなる

行単位での文字列の読み込み

- getline(is, s)
  - 入力ストリーム is から1行（改行まで）読み込み，改行文字を除いた文字列を s に格納する
  - s に以前に保存されていたデータは破棄される
  - 戻り値は，is への参照

行から単語の切り出し

- split 関数の作成
  - 引数: 1行の内容の文字列
    - 文字列の内容を変更しないようにコピーを渡すstring型か
    - 定数型の参照, const string&
  - 戻り値: 切り出された複数の単語を格納したvector<string>
  - vector<string> split(const string& str)

行から単語の切り出し

- str に対して，以下を繰り返す
- 先頭のスペースをとばす
  - スペース以外の文字が最初に現れる場所を探す
  - そこを単語の先頭とする
- 次に，そこから単語の区切り目を探す
  - スペースが最初に現れる場所を探す
  - そこを単語の末尾とする
- 単語の先頭と末尾が異なっていたら，vectorに登録

行から単語の切り出し

- string::iterator i, j;
- i = find_if(i, str.end(), not_space);
- j = find_if(i, str.end(), space);
- string(i, j) が単語となる
  - イテレータ i と j の間のシーケンスからなるコンテナ（文字列）を生成
- bool space(char c){ return(isspace(c)); }
- bool not_space(char c){ return(!isspace(c)); }

デフォルト引数

- 関数の引数が省略されたときに，自動的に使われる値
- 関数のプロトタイプ，または関数の宣言のどちらかでのみ，指定可能
  - void test1(double a = 1.0, double b = 10.0);
  - test1(); // aには1.0, bには10.0
  - test1(5.0); // aには5.0, bには10.0
  - test1(5.0, 20.0); // aには5.0, bには20.0
- デフォルト変数は最も右側から作用

左辺値

- コンパイルエラーで，lvalueとか現れたことありませんか？
- lvalue：左辺値，代入式の左辺に指定して良い値
  - 実体のあるオブジェクト
  - 一時的な値ではない
  - a=15/100としたとき，aは左辺値，15/100は左辺値ではない（計算が終わったら破棄される）
  - 参照を戻す関数の戻り値は左辺値として使えるが，変数の寿命に注意

変数の寿命に注意

動作は不安定：関数内の自動（スタック）変数だから

```
int main()
{
  func1()= 1;
}
int& func1()
{
  int a=0;
  return(a);
}
```

左の例と本質的に同一

```
int main()
{
  *func2()= 1;
}
int* func2()
{
  int a=0;
  return(&a);
}
```

動作は不安定：関数内の自動（スタック）変数だから

```
int main()
{
  func3()= 1;
}
int& func3()
{
  int a[1];
  a[0] = 0;
  return(a[0]);
}
```

動作は安定：vectorがヒープ変数だから

```
int main()
{
  func4()= 1;
}
int& func4()
{
  vector<int> a;
  a.push_back(0);
  return(a[0]);
}
```

演習7のポイント

- map< string, vector<int> > counters;
- クエリ（検索単語）の読み込み
- while (1行単位で読み込み) { // getline
  - 行番号を計算
  - 行を単語に分解 // split
  - 分解された単語に対して
    - counters[クエリ].push_back(行番号);
- }
- counters[クエリ]の行番号を出力