SCC-211
Lab. Algoritmos Avançados

Capítulo 2

Estruturas de Dados Lineares – Parte 1

João Luís G. Rosa

Standard Template Library - STL

- Biblioteca de classes que utiliza *templates* para implementar diversas estruturas de dados.
- *Templates* permitem parametrizar a estrutura de dados com um tipo de dados escolhido pelo programador.
- Para utilizá-la é necessário um compilador C++ e a biblioteca instalada (*default*).
- É possível utilizar a biblioteca sem ter que desenvolver um código orientado a objetos: C+-.
- Vários exemplos sobre isso serão vistos no curso.

Standard Template Library - STL

- STL provê uma coleção de *templates* representando *containers*, *iterators*, objetos de função e algoritmos.
- Um *container* é uma unidade, como um vetor, que pode armazenar vários valores. Os *containers* STL são homogêneos, ou seja, armazenam valores do mesmo tipo.
- Algoritmos são receitas para acompanhar determinadas tarefas, como ordenar um vetor ou achar um valor numa lista.

Standard Template Library - STL

- *Iterators* são objetos que permitem que se mova através de um *container* da mesma forma que ponteiros permitem que se mova através de um vetor. São generalizações de ponteiros.
- Objetos de função são objetos que agem como função: podem ser objetos de classe ou ponteiros de funções (que inclui nomes de funções, porque um nome de função age como um ponteiro).
- STL permite que se construa uma variedade de *containers*, incluindo vetores, filas e listas e permite que se execute várias operações, como busca e ordenação.

Programação Genérica

- STL é um exemplo de *Programação Genérica* (PG).
- A programação orientada a objetos (POO) se concentra no aspecto dos dados da programação enquanto que a PG se concentra nos **algoritmos**.
- Em comum: abstração e criação de código reusável.
- Mas filosofias diferentes!

Programação Genérica

- Uma meta da PG é escrever código que seja independente dos tipos de dados.
- Os *templates* são a ferramenta C++ para fazer programas genéricos.
- Os *templates* permitem definir uma função ou classe em termos de um tipo genérico.
- A STL vai além, provendo uma representação genérica dos algoritmos.
- Os *templates* tornam isso possível.

Standard Template Library - STL

- Infelizmente, existem diferenças entre as diferentes implementações da STL.
- É importante saber que para a Maratona:
  - Na final nacional o sistema utilizado atualmente é o Maratona Linux (Debian) com g++ 3.3.5 e STL padrão.
  - Na primeira fase, o sistema é de responsabilidade da sede (mas geralmente Maratona Linux).
- Portanto, o mais seguro é treinar em Linux e evitar eventuais diferenças de compilador!

acm International Collegiate Programming Contest

IBM. event sponsor

Standard Template Library - STL

- STL é uma biblioteca bastante extensa, envolve estruturas de dados básicas e diversos **algoritmos** tais como:
  - Strings;
  - Estruturas de dados básicas e (classes *containers*);
  - Algoritmos de ordenação.
- STL contém uma grande quantidade de estruturas de dados básicas (classes *containers*):
  - Vectors;
  - Maps;
  - Sets;
  - Stacks;
  - Queues;
  - Deques;
  - Priority Queues;
  - Outras...

Standard Template Library - STL

- É importante saber que se deve tomar diversos cuidados ao se utilizar a STL:
  - O mais importante é que a STL foi projetada tendo-se em mente o desempenho;
  - Portanto, os algoritmos estão entre os mais eficientes conhecidos (excelente para maratona!);
  - Entretanto, a biblioteca não oferece muitas verificações de erros cometidos pelo programador!

vectors

- Podem ser considerados vetores dinâmicos.
- Permitem a indexação dos elementos, utilizando a mesma sintaxe dos vetores.
- O primeiro elemento encontra-se na posição 0.
- Para utilizar:

```
#include<vector>
using namespace std;
```

- Para declarar:

```
vector<tipo> <objeto>;
```

- Para inserir um valor:

```
<objeto>.push_back(valor);
```

vectors

- Exemplo:

```
vector<double> exemplo;
exemplo.push_back(1.2);
exemplo.push_back(3.1);
```

- Para os elementos inseridos por **push_back** é possível acessá-los/modificá-los usando um índice:

```
for (i=0; i < exemplo.size(); i++)
      printf("%d", exemplo[i]);
```

- O método **size()** fornece o número de exemplos no **vector**.

vectors

- Cuidado, nunca acesse um elemento além do tamanho do vetor.

```
vector<int> exemplo;
exemplo[2] = 2;                 // Ops...
```

- Todo elemento deve ser primeiro inserido pelo método **push_back()**, ou

```
vector<char> exemplo(10);    // Cria 10 elems
exemplo[9] = 'A';            // Ok!
```

`iterators`

- Entender iteradores é entender a STL.
- Assim como os *templates* tornam os algoritmos independentes do tipo do dado armazenado, os iteradores tornam os algoritmos independentes do tipo de *container* usado.
- Portanto, os iteradores são componentes essenciais da abordagem genérica da STL.

Iteradores

- Generalização de um ponteiro;
- Permite acessar os elementos de algumas classes *containers* (**`vector`**, **`set`**, **`map`**, etc.).
- São manipulados pelos seguintes operadores sobrecarregados:
  - Incremento e decremento: ++ e --;
  - Igualdade e desigualdade: == e !=;
  - Dereferenciação: *

Iteradores

```
#include<vector>

using namespace std;

int main () {
  vector<int> vet;
  vector<int>::iterator p;

  for (int i=0; i <=4; i++)
    vet.push_back(i);

    for (p=vet.begin(); p!=vet.end(); p++)
      printf("%d", *p);
}
```


Iteradores

- Cuidado: o método **end( )** retorna um sentinela, e não um iterador para o último elemento.
- É necessário declarar:

  `vector<int>::iterator p;`

  uma vez que cada classe *container* implementa o seu iterador.

Iteradores

- Existem três categorias de iteradores:
  - Iteradores de avanço: somente ++;
  - Iteradores bidirecionais: ++ e --;
  - Iteradores de acesso aleatório: ++, -- e [].
- Todo iterador de acesso aleatório também é um iterador bidirecional.
- Todo iterador bidirecional também é de avanço.

Iteradores

- Existem duas sub-categorias:
  - Iteradores contantes: não permitem modificar os dados;
  - Iteradores mutáveis: permitem modificar.
- Iteradores constantes servem para trechos de código que não devem alterar os dados.
- Declarados com **`const_iterator`**.

Iteradores

- Existem também iteradores reversos. Eles resolvem o problema:

```
for (p = o.end(); p != o.begin(); p--)  //Ops...
```

- O método **`end()`** retorna um sentinela, e não um iterador para o último elemento.

```
reverse_iterator rp;
for (rp = o.rbegin(); rp!= o.rend(); rp++)
  // Ok!
```

- Para um **`reverse iterator`** o operador ++ caminha para o início do *container*.

`map`

- Cria associações entre pares de valores. Os elementos do **`map`** são mantidos pela ordem da chave. Portanto, é útil quando a ordem das chaves é importante.
- Não permite duas chaves iguais.
- **`#include<map>`**
- Declaração: **`map<TChave, TData>`**

map

- Operações:
  - **`m.size()`** – Fornece o número de elementos no **`map`**;
  - **`m.empty()`** – Retorna true se o **`map`** estiver vazio;
  - **`m.insert(E)`** – Insere **`E`** no **`map`**;
  - **`m.erase(C)`** – Remove o elemento de chave **`C`**;
  - **`m.find(C)`** – Retorna um iterador para o elemento de chave **`C`**, ou **`m.end()`** se não existe **`C`**;
  - **`m1==m2`** – true se **`m1`** e **`m2`** possuem os mesmos elementos.

map

```
#include<cstdio>
#include<map>
using namespace std;

int main() {
    map<char, int> m;
    map<char, int>::iterator p;

    m.insert(pair<char , int>('c', 2));
    m.insert(make_pair('b', 4));
    m['a'] = 5;
    printf("%d\n", m['b']);
    for (p=m.begin();p!=m.end(); p++)
        printf("%c %d\n", p->first, p->second);
}
```

map

```
#include<cstdio>
#include<cstring>
#include<map>

using namespace std;

struct ltstr {
    bool operator()(const char* s1, const char* s2) const {
        return strcmp(s1, s2) < 0;
    }
};

int main()
{
    map<const char*, int, ltstr> months;

    months["january"] = 31;
    months["february"] = 28;
    months["march"] = 31;

    printf("june -> %d", months["june"]);
}
```


Filas, Pilhas e **deques**

- São estruturas lineares simples.
- Geralmente, não são a solução direta para um problema, mas podem ser úteis como estruturas de dados intermediárias.
- Por exemplo, uma fila é útil para a implementação de uma busca em largura em grafos e uma pilha para uma busca em profundidade.

deque

- É uma estrutura flexível, similar aos **`vector`**s que permite inserção de elementos em ambas as extremidades.
- Características:
  - Acesso aleatórios aos elementos;
  - Inserção e remoção de elementos em tempo constante nas extremidades;
  - Inserção e remoção de elementos em tempo linear no meio.

deque

- **`#include <deque>`**
- Declaração: **`deque <T>`**
- Operações:
  - **`d.size()`** – Fornece o número de elementos na **`deque`**;
  - **`d.empty()`** – Fornece true se a **`deque`** estiver vazia;
  - **`d.front()`** – Fornece o elemento na frente da **`deque`**;
  - **`d.back()`** – Fornece o elemento no final da **`deque`**;
  - **`d.push_front(E)`** – Insere **`E`** no início da **`deque`**;
  - **`d.push_back(E)`** – Insere **`E`** no final da **`deque`**;

deque

- Operações
  - **`d.pop_front()`** – Remove o elemento na frente da **`deque`**;
  - **`d.pop_back()`** – Remove o elemento no final da **`deque`**;
  - **`d.insert (I, E)`** – insere **`E`** na posição dada pelo **`iterador I`**;
  - **`d.erase(I)`** – remove o elemento na posição **`I`**;
  - **`d.clear()`** – apaga a **`deque`**;
  - **`d1 == d2`** – true se **`d1`** e **`d2`** possuem os mesmos elementos na mesma ordem.

deque - Exemplo

```
#include<cstdio>
#include<deque>

using namespace std;

int main() {
    deque<char> d;
    int i;

    d.push_back('c');
    d.push_front('a');
    d.insert(d.begin() + 1, 'b');
    for (i=0; i<d.size(); i++)
        printf("%c", d[i]);
    printf("%c", d.front());
    printf("%c", d.back());
    d.pop_front();
    d.pop_back();
    printf("%c", d[0]);
    return 0;
}
```

stacks

- Pilhas
  - **#include <stack>**
  - Declaração: **stack <T>**
  - Operações:
    - **s.size()** – Fornece o número de elementos na pilha;
    - **s.empty()** – Fornece true se a pilha estiver vazia;
    - **s.top()** – Fornece o elemento no topo da pilha;
    - **s.push(E)** – Insere uma cópia de **E** no topo da pilha;
    - **s.pop()** – Remove o elemento no topo da pilha;
    - **s1 == s2** – true se **s1** e **s2** possuem os mesmos elementos.

stacks - Exemplo

```
#include<cstdio>
#include<stack>

using namespace std;

int main() {
    stack<int> s;
    s.push(1);
    s.push(2);
    s.push(3);
    printf("%d\n", s.top());
    s.pop();
    printf("%d\n", s.top());
}
```

queues

- Filas
  - `#include <queue>`
  - Declaração: `queue <T>`
  - Operações:
    - `q.size()`– Fornece o número de elementos na fila;
    - `q.empty()` – Fornece true se a fila estiver vazia;
    - `q.front()` – Fornece o elemento na frente da fila;
    - `q.back()` – Fornece o elemento no final da fila;
    - `q.push(E)` – Insere `E` no final da fila;
    - `q.pop()` – Remove o elemento na frente da pilha;
    - `q1 == q2` – true se `q1` e `q2` possuem os mesmos elementos.

Classes Containers

vector	**vector<T>::iterator** **vector<T>::const_iterator** **vector<T>::reverse_iterator** **vector<T>::const_reverse_iterator** **(iteradores de acesso aleatório)**	**#include<vector>**
maps	**map<T>::iterator** **map<T>::const_iterator** **map<T>::reverse_iterator** **map<T>::const_reverse_iterator** **(iteradores aleatórios, chaves constantes)**	**#include<map>**

Classes Containers

deque	deque<T>::iterator deque<T>::const_iterator deque<T>::reverse_iterator deque<T>::const_reverse_iterator (iteradores de acesso aleatório)	#include<deque>
stack	sem iteradores	#include<stack>
queue	sem iteradores	#include<queue>

WERTYU

PC/Uva IDs: 110301/10082, Popularity: A, Sucess rate: high, Level 1

A common typing error is to place your hands on the keyboard one row to the right of the correct position. Then "Q" is typed as "W" and "J" is typed as "K" and so on. Your task is to decode a message typed in this manner.

WERTYU

Input

Input consists of several lines of text. Each line may contain digits, spaces, uppercase letters (except "Q", "A", "Z"), or punctuation shown above [except back-quote (`)]. Keys labeled with words [Tab, BackSp, Control, etc.] are not represented in the input.

Output

You are to replace each letter or punctuation symbol by the one immediately to its left on the QWERTY keyboard shown above. Spaces in the input should be echoed in the output.

WERTYU

Sample Input

O S, GOMR YPFSU/

Sample Output

I AM FINE TODAY.

Exercício: WERTYU

```
#include<map>
#include<string>
#include<iostream>

using namespace std;

char *str[] = {"`1234567890-=",
               "QWERTYUIOP[]\\",
               "ASDFGHJKL;'",
               "ZXCVBNM,./"};
```


Exercício: WERTYU

```
int main() {
  map<char, char> m;
  int i, j;
  string line;

  for (i=0; i < 4; i++)
    for (j=1; j < strlen(str[i]); j++)
      m[str[i][j]] = str[i][j-1];

  while (getline(cin, line)) {
    for (i=0; i < line.length(); i++)
      if (line[i] == ' ')
        printf("%c", line[i]);
      else
        printf("%c", m[line[i]]);
    printf("\n");
  }
}
```

Referências

- Batista, G. & Campello, R.
  - Slides disciplina *Algoritmos Avançados*, ICMC-USP, 2007.
- Prata, S.
  - *C++ Primer Plus*. Waite Group Press, 1998.
- Skiena, S. S. & Revilla, M. A.
  - *Programming Challenges – The Programming Contest Training Manual*. Springer, 2003.