Standard Template Library (STL)

Roland Teodorowitsch

Programação Orientada a Objetos - ECo - Curso de Engenharia de Computação - PUCRS

6 de novembro de 2023

Standard Template Library (STL)

Standard Template Library (STL)

- Considerando a utilidade do reuso de software e também a utilidade das estruturas de dados e algoritmos utilizados por programadores a **Standard Template Library (STL)** foi adicionada à biblioteca padrão C++
- A **STL** define componentes genéricos reutilizáveis poderosos que implementam várias estruturas de dados e algoritmos que processam estas estruturas
- Basicamente, a **STL** é composta de **contêineres**, **iteradores** e **algoritmos**

Standard Template Library (STL)

- **Contêineres** são templates de estruturas de dados
  - Possuem métodos associados a eles
- **Iteradores** são semelhantes a ponteiros, utilizados para percorrer e manipular os elementos de um contêiner
- **Algoritmos** são as funções que realizam operações tais como buscar, ordenar e comparar elementos ou contêineres inteiros
  - Existem aproximadamente 85 algoritmos implementados na STL
  - A maioria utiliza iteradores para acessar os elementos de contêineres

Contêineres

Contêineres

- Os contêineres são divididos em três categorias principais
  1. **Contêineres Sequenciais**
     - Estruturas de dados lineares
  2. **Contêineres Associativos**
     - Estruturas de dados não lineares
     - Exploram o conceito de pares chaves/valor
  3. **Adaptadores de Contêineres**
     - São contêineres sequenciais, porém, em versões restringidas

Contêineres

Contêineres	Tipo	Descrição
`vector`	Sequencial	Inserções e remoções no final, acesso direto a qualquer elemento.
`deque`	Sequencial	Fila duplamente ligada, inserções e remoções no início ou no final, sem acesso direto a qualquer elemento.
`list`	Sequencial	Lista duplamente ligada, inserção e remoção em qualquer ponto.

Contêineres

Contêineres	Tipo	Descrição
`set`	Associativo	Busca rápida, não permite elementos duplicados.
`multiset`	Associativo	Busca rápida, permite elementos duplicados.
`map`	Associativo	Mapeamento um-para-um, não permite elementos duplicados, busca rápida.
`multimap`	Associativo	Mapeamento um-para-um, permite elementos duplicados, busca rápida.
`stack`	Adaptadores	*Last-in, first-out* (LIFO).
`queue`	Adaptadores	*First-in, first-out* (FIFO).
`priority_queue`	Adaptadores	O elemento de maior prioridade é sempre o primeiro elemento a sair.

Funções Comuns aos Contêineres

- Todos os contêineres da STL fornecem funcionalidades similares, sendo que muitas operações genéricas se aplicam a todos os contêineres

Funcionalidade	Descrição
`Construtor default`	Fornece a inicialização padrão do contêiner.
Construtor Cópia	Construtor que inicializa um contêiner para ser a cópia de outro do mesmo tipo.
Destrutor	Simplesmente destrói o contêiner quando não for mais necessário.
`empty`	Retorna `true` se não houver elementos no contêiner e `false` caso contrário.
`size`	Retorna o número de elementos no contêiner.
`operator=`	Atribui um contêiner a outro.
`operator<`	Retorna `true` se o primeiro contêiner for menor que o segundo e `false` caso contrário.

Funções Comuns aos Contêineres

Funcionalidade	Descrição
`operator<=`	Retorna `true` se o primeiro contêiner for menor ou igual ao segundo e `false` caso contrário.
`operator>`	Retorna `true` se o primeiro contêiner for maior que o segundo e `false` caso contrário.
`operator>=`	Retorna `true` se o primeiro contêiner for maior ou igual ao segundo e `false` caso contrário.
`operator==`	Retorna `true` se o primeiro contêiner for igual ao segundo e `false` caso contrário.
`operator!=`	Retorna `true` se o primeiro contêiner for diferente do segundo e `false` caso contrário.
`swap`	Troca os elementos de dois contêineres.

- **Atenção**: Os operadores <, <=, >, >=, == e != não são fornecidos para o contêiner `priority_queue`.

Funções Comuns a Contêineres Sequenciais e Associativos

Funcionalidade	Descrição
`max_size`	Retorna o número máximo de elementos de um contêiner.
`begin`	As duas versões deste método retornam um `iterator` ou um `const_iterator` para o primeiro elemento do contêiner.
`end`	As duas versões deste método retornam um `iterator` ou um `const_iterator` para a posição após o final do contêiner.
`rbegin`	As duas versões deste método retornam um `reverse_iterator` ou um `const_reverse_iterator` para o primeiro elemento do contêiner invertido.
`rend`	As duas versões deste método retornam um `reverse_iterator` ou um `const_reverse_iterator` para a posição após o final do contêiner invertido.
`erase`	Apaga um ou mais elementos do contêiner.
`clear`	Apaga todos os elementos do contêiner.

Arquivos de Cabeçalho da STL

Arquivo	Observação
<vector>	Vetor
<list>	Lista
<deque>	Fila duplamente ligada
<queue>	Contém queue e priority_queue
<stack>	Pilha
<map>	Contém map e multimap
<set>	Contém set e multiset
<bitset>	Conjunto de bits (vetor em que cada elemento é um bit – 0 ou 1)

Contêineres

- é necessário garantir que os elementos armazenados em um contêiner suportam um conjunto mínimo de operações
  - Quando um elemento é inserido em um contêiner, ele é copiado.
    - Logo, o operador de atribuição deve ser sobrecarregado se necessário
    - Também deve haver um construtor cópia
  - Contêineres associativos e alguns algoritmos requerem que os elementos sejam comparados
    - Pelo menos os operadores == e < devem ser sobrecarregados.

Iteradores

Iteradores

- **Iteradores** são utilizados para apontar elementos de contêineres sequenciais e associativos
  - Entre outras coisas
  - Algumas funcionalidades como `begin` e `end` retornam **iteradores**
- Se um `iterador i` aponta para um elemento:
  - `++i` aponta para o próximo elemento
  - `*i` se refere ao conteúdo do elemento apontado por `i`

Iteradores

- Os **iteradores** são objetos declarados no arquivo de cabeçalho `<iterator>`
- Existem basicamente dois tipos de objetos **iteradores**:
  - `iterator`: aponta para um elemento que pode ser modificado
  - `const_iterator`: aponta para um elemento que não pode ser modificado
- Por exemplo, é possível criar **iteradores** para o `cin` e o `cout`
  - São sequências de dados, assim como os contêineres
  - é possível percorrer os dados captados por um `cin` e os dados a serem escritos por um `cout`

Iteradores

- Exemplo de iteradores para `cin` e `cout`

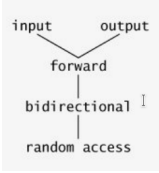
```
#include <iostream>
#include <iterator> // ostream_iterator e istream_iterator

using namespace std;

int main() {
  cout << "Informe␣dois␣inteiros:␣";
  // cria istream_iterator para ler valores de int a partir de cin
  istream_iterator<int> inputInt(cin);
  int number1 = *inputInt; // le int a partir da entrada padrao
  ++inputInt; // move iterador para o proximo valor de entrada
  int number2 = *inputInt; // le int a partir da entrada padrao
  // cria ostream_iterator para gravar valores int em cout
  ostream_iterator<int> outputInt(cout);
  cout << "A␣soma␣eh:␣";
  *outputInt = number1 + number2; // gera saida do resultado para cout
  cout << endl;
  return 0;
}
```

- Saída:

```
Informe dois inteiros: 12 25
A soma eh: 37
```

Categorias de Iteradores

Categoria	Descrição
input	Utilizado para ler um elemento de um contêiner. Só se move do início para o final, um elemento por vez. Não pode ser utilizado para percorrer um contêiner mais que uma vez.
output	Utilizado para escrever um elemento em um contêiner. Só se move do início para o final, um elemento por vez. Não pode ser utilizado para percorrer um contêiner mais que uma vez.
forward	Combina os iteradores *input* e *output* e retém a sua posição no contêiner.
bidirectional	Combina o iterador *forward* com a capacidade de se mover do final para o início. Pode ser utilizado para percorrer um contêiner mais que uma vez.
random access	Combina o iterador *bidirectional* com a capacidade de acessar diretamente qualquer elemento. Ou seja, pode saltar uma quantidade arbitrária de elementos.



- Usar o iterador "mais fraco" maximiza a reusabilidade.
- Por exemplo, um algoritmo que usa um iterador *forward* aceita iteradores *forward*, *bidirectional* ou *random access*.

Tipos Predefinidos de Iteradores

Tipo Predefinido	Direção do ++	Capacidade
`iterator`	Início para o Final	Leitura/Escrita
`const_iterator`	Início para o Final	Leitura
`reverse_iterator`	Final para o Início	Leitura/Escrita
`const_reverse_iterator`	Final para o Início	Leitura

- Nem todos os tipos predefinidos o são para todos os contêineres
- As versões const são utilizadas para percorrer contêineres de **somente leitura**
- **Iteradores invertidos** são utilizados para percorrer contêineres na **direção inversa**

Operações em Iteradores

Operação	Tipo de Iterador	Descrição
`++p`	TODOS	Pré-incremento
`p++`	TODOS	Pós-incremento
`*p`	`input`	Referencia o conteúdo apontado
`p = p1`	`input`	Atribui um iterador a outro
`p == p1`	`input`	Compara dois iteradores quanto à igualdade
`p != p1`	`input`	Compara dois iteradores quanto à desigualdade
`*p`	`output`	Referencia o conteúdo apontado
`p = p1`	`output`	Atribui um iterador a outro

Operações em Iteradores

Operação	Tipo de Iterador	Descrição
`p += i`	`random access`	Incrementa o iterador em `i` posições
`p -= i`	`random access`	Decrementa o iterador em `i` posições
`p + i`	`random access`	Resulta em um iterador posicionado em `p+i` elementos
`p - i`	`random access`	Resulta em um iterador posicionado em `p-i` elementos
`p[i]`	`random access`	Retorna uma referência para o elemento a `i` posições a partir de `p`
`p < p1`	`random access`	Retonar `true` se o primeiro iterador estiver antes do segundo no contêiner
`p <= p1`	`random access`	Retonar `true` se o primeiro iterador estiver antes ou na mesma posição do segundo no contêiner
`p > p1`	`random access`	Retonar `true` se o primeiro iterador estiver após o segundo no contêiner
`p >= p1`	`random access`	Retonar `true` se o primeiro iterador estiver após ou na mesma posição do segundo no contêiner

Algoritmos

Algoritmos

- A STL inclui aproximadamente 85 **algoritmos**
  - Podem ser utilizados genericamente, em vários tipos de contêineres
- Os **algoritmos** operam indiretamente sobre os elementos de um contêiner usando iteradores
  - Vários deles utilizam pares de iteradores, um apontando para o início e outro apontando para o final
  - Frequentemente os **algoritmos** também retornam iteradores como resultado
  - Este desacoplamento dos contêineres permite que os algoritmos sejam genéricos
- A STL é extensível
  - Ou seja, é possível adicionar novos algoritmos a ela

Algoritmos

- Entre os vários algoritmos disponíveis, é possível encontrar:
  1. Algoritmos alteradores de sequências:
     `copy`, `remove`, `reverse_copy`, `copy_backward`, `remove_copy`, `rotate`, `fill`, `remove_copy_if`, `rotate_copy`, `fill_n`, `remove_if`, `stable_partition`, `generate`, `replace`, `swap`, `generate_n`, `replace_copy`, `swap_ranges`, `iter_swap`, `replace_copy_if`, `transform`, `partition`, `replace_if`, `unique`, `random_shuffle`, `reverse`, `unique_copy`
  2. Algoritmos não alteradores de sequências:
     `adjacent_find`, `find`, `find_if`, `count`, `find_each`, `mismatch`, `count_if`, `find_end`, `search`, `equal`, `find_first_of`, `search_n`
  3. Algoritmos numéricos (definidos em <numeric>):
     `accumulate`, `partial_sum`, `inner_product`, `adjacent_difference`

Exemplos de Algoritmos

- Ordenação:

Algoritmo	Descrição
`sort`	Ordena os elementos do contêiner
`stable_sort`	Ordena os elementos do contêiner preservando a ordem relativa dos equivalentes
`partial_sort`	Ordena parcialmente o contêiner
`partial_sort_copy`	Copia os menores elementos e os ordena no contêiner de destino
`nth_element`	Ordena o n-ésimo elemento

- Busca Binária:

`lower_bound`	Retorna um iterador para o limite esquerdo
`upper_bound`	Retorna um iterador para o limite direito
`equal_range`	Retorna os limites que incluem um conjunto de elementos com um determinado valor
`binary_search`	Testa se um valor existe em um intervalo

Exemplos de Algoritmos

- Intercalação:

Algoritmo	Descrição
`merge`	Intercala os elementos de dois intervalos e coloca o resultado em outro intervalo
`inplace_merge`	Intercala os elementos de dois intervalos e coloca o resultado no mesmo intervalo
`includes`	Testa se um intervalo ordenado inclui outro intervalo ordenado, se cada elemento de um intervalo é equivalente a outro do segundo intervalo
`set_union`	Calcula a união entre dois intervalos de valores
`set_intersection`	Calcula a interseção entre dois intervalos de valores
`set_difference`	Calcula a diferença entre dois intervalos de valores
`set_symmetric_difference`	Calcula a diferença simétrica entre dois intervalos de valores

Exemplos de Algoritmos

- *Heap*:

Algoritmo	Descrição
`push_heap`	Adiciona um elemento a um *heap*
`pop_heap`	Remove um elemento de um *heap*
`make_heap`	Cria um *heap* a partir de um intervalo de valores
`sort_heap`	Ordena os elementos de um *heap*

Exemplos de Algoritmos

- Min/Max:

Algoritmo	Descrição
`min`	Retorna o menor de dois argumentos
`max`	Retorna o maior de dois argumentos
`min_element`	Retorna o menor elemento de uma sequência
`max_element`	Retorna o maior elemento de uma sequência
`lexicographical_compare`	Comparação lexicográfica (menor que)
`next_permutation`	Transforma uma sequência na próxima permutação (ordem lexicográfica)
`prev_permutation`	Transforma uma sequência na permutação anterior (ordem lexicográfica)

Exemplos Completos

vector

- A classe `vector` é genérica, logo, deve-se definir o tipo na declaração de um objeto
- Este contêiner é dinâmico
  - A cada inserção o contêiner se redimensiona automaticamente
- O método `push_back` adiciona um elemento ao final do `vector`
- Outros possíveis métodos incluem:
  - `front`: determina o primeiro elemento
  - `back`: determina o último elemento
  - `at`: determina o elemento em uma determinada posição, mas antes verifica se é uma posição válida
  - `insert`: insere um elemento em uma posição especificada por um iterador

vector: exemplo 1

```
#include <iostream>
#include <vector>
using namespace std;
int main () {
  vector<int> v1;    // vetor vazio de inteiros
  cout << "v1.size()=" << v1.size() << endl;
  for ( vector<int>::iterator it = v1.begin();
        it != v1.end(); ++it)
      cout << ' ' << *it;
  cout << '\n';

  vector<int> v2 (4,100);    // quatro ints com 100
  cout << "v2.size()=" << v2.size() << endl;
  for ( vector<int>::iterator it = v2.begin();
        it != v2.end(); ++it)
      cout << ' ' << *it;
  cout << '\n';

  vector<int> v3 (v2.begin(),v2.end()); // it. de v2
  cout << "v3.size()=" << v3.size() << endl;
  for ( vector<int>::iterator it = v3.begin();
        it != v3.end(); ++it)
      cout << ' ' << *it;
  cout << '\n';
```

```
  vector<int> v4 (v3);  // uma copia de v3
  cout << "v4.size()=" << v4.size() << endl;
  for (vector<int>::iterator it = v4.begin();
      it != v4.end(); ++it)
    cout << ' ' << *it;
  cout << '\n';

  // o construtor tambem pode ser usado com arrays
  int myints[] = {16,2,77,29};
  vector<int> v5 (myints, myints + sizeof(myints) / sizeof(
  cout << "O conteudo de v5 eh:";
  for ( vector<int>::iterator it = v5.begin();
        it != v5.end(); ++it)
      cout << ' ' << *it;
  cout << '\n';
  cout << *v5.begin() << endl;
  cout << v5.begin()[0] << endl;
  cout << v5[0] << endl;
  cout << *(v5.end()-1) << endl;
  cout << v5[v5.size()-1] << endl;

  return 0;
}
```

vector: exemplo 2

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int>::const_iterator it;
  vector <int> v1;
  cout << "v1.size()=" << v1.size() << endl;
  cout << "v1.max_size()=" << v1.max_size() << endl;
  if (v1.empty())
     cout << "vazio...";
  else
     for (it=v1.begin(); it!=v1.end(); ++it)
         cout << *it << ' ';
  cout << endl;
  cout << "v1.push_back(12);" << endl;
  v1.push_back(12);
  cout << "v1.push_back(34);" << endl;
  v1.push_back(34);
  cout << "v1.push_back(56);" << endl;
  v1.push_back(56);
  cout << "v1.size()=" << v1.size() << endl;
  for (it=v1.begin(); it!=v1.end(); ++it)
      cout << *it << ' ';
  cout << endl;
  return 0;
}
```

vector: exemplo 3

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int>::const_iterator it;
  vector<int>::iterator it2;
  vector <int> v2(10);
  cout << "v2.size()=" << v2.size() << endl;
  cout << "v2.max_size()=" << v2.max_size() << endl;
  if (v2.empty()) cout << "vazio..." << endl;
  else {
     for (it=v2.begin(); it!=v2.end(); ++it)
         cout << *it << ' ';
     cout << endl;
     int x = 10;
     for (it2=v2.begin(); it2!=v2.end(); ++it2) {
         *it2 = x;
         x += 10;
     }
     for (it=v2.begin(); it!=v2.end(); ++it)
         cout << *it << ' ';
     cout << endl;
     v2.pop_back();
     v2.pop_back();
     cout << "2 x v2.pop_back();" << endl;
     cout << "v2.size()=" << v2.size() << endl;
     for (it=v2.begin(); it!=v2.end(); ++it)
         cout << *it << ' ';
     cout << endl;
     v2.erase(v2.begin()+1);
     cout << "v2.erase(v2.begin()+1);" << endl;
     cout << "v2.size()=" << v2.size() << endl;
     for (it=v2.begin(); it!=v2.end(); ++it)
         cout << *it << ' ';
     cout << endl;
     // Apaga elementos de ind. 4 (5.) ate ind. 5 (6.)
     v2.erase(v2.begin()+4,v2.begin()+6);
     cout << "v2.erase(v2.begin()+4,v2.begin()+6);" << endl;
     cout << "v2.size()=" << v2.size() << endl;
     for (it=v2.begin(); it!=v2.end(); ++it)
         cout << *it << ' ';
     cout << endl;
     it=v2.begin();
     for (int i=0; i<v2.size(); ++i) cout << *(it+i) << ' ';
     cout << endl;
     for (int i=0; i<v2.size(); ++i) cout << it[i] << ' ';
     cout << endl;
     for (int i=0; i<v2.size(); ++i) cout << v2[i] << ' ';
     cout << endl;
  }
  return 0;
}
```

list

- No exemplo a seguir, os seguintes métodos da classe list são usados:
  - `sort`: ordena a lista em ordem crescente
  - `unique`: remove elementos duplicados
  - `remove`: apaga todas as ocorrências de um determinado valor da lista
- Existem outros como:
  - `reverse`: inverte a lista
  - `merge`: intercala listas
  - `remove_if`: remove elementos que atendam um critério

list: exemplo 1

```
#include <iostream>
#include <list>
using namespace std;
void mostraLista(string nome,list<int> l) {
  cout << nome << " = {";
  list<int>::iterator it;
  for (it = l.begin(); it != l.end(); it++) {
      if (it != l.begin()) cout << ',';
      cout << *it;
  }
  cout << "}" << endl;
}
int main () {
  // lista de 4 inteiros com valor 100
  list<int> first (4,100);
  mostraLista("first", first);
  // lista criada a partir de outra
  list<int> second (first.begin(),first.end());
  mostraLista("second", second);
  // lista criada com construtor de copia
  list<int> third (second);
  mostraLista("third", third);
  // a lista pode ser criada a partir de um array
  int v[] = { 2, 3, 5, 7, 11};
  list<int> fourth (v, v + sizeof(v)/sizeof(int) );
  mostraLista("fourth", fourth);
  list<int> fifth;        // lista de inteiros vazia
  mostraLista("fifth", fifth);
  fifth.push_front( 1 ); //insere na frente
  mostraLista("fifth", fifth);
  fifth.push_front( 2 );
  mostraLista("fifth", fifth);
  fifth.push_front( 3 );
  mostraLista("fifth", fifth);
  fifth.push_back( 4 );  //insere no final
  mostraLista("fifth", fifth);
  fifth.push_back( 1 );
  mostraLista("fifth", fifth);
  fifth.sort();           // ordena a lista
  mostraLista("fifth", fifth);
  fifth.remove( 4 );      // remove todos os 4s
  mostraLista("fifth", fifth);
  fifth.unique();         // remove elementos duplicados
  mostraLista("fifth", fifth);
  fifth.pop_front();      // remove elemento do inicio
  mostraLista("fifth", fifth);
  fifth.pop_back();       // remove elemento do fim
  mostraLista("fifth", fifth);
  fifth.clear();          // esvazia a lista
  mostraLista("fifth", fifth);
  return 0;
}
```

deque

- Não possui, por exemplo, os métodos `sort()`, `remove()` e `unique()`
- O método `push_front()` está disponível apenas para `list` e `deque`
- O operador `[]` permite acesso direto aos elementos do `deque`
  - Também pode ser utilizado em um `vector`
- Em geral, um `deque` possui um desempenho levemente inferior em relação a um `vector`
  - No entanto, é mais eficiente para fazer inserções e remoções no início

deque: exemplo 1

```
#include <iostream>
#include <deque>

using namespace std;

void mostraDeque(string nome,deque<int> l) {
  cout << nome << " = {";
  deque<int>::iterator it;
  for (it = l.begin(); it != l.end(); it++) {
      if (it != l.begin()) cout << ',';
      cout << *it;
  }
  cout << "}" << endl;
}

int main () {
  // deque de 4 inteiros com valor 100
  deque<int> first (4,100);
  mostraDeque("first", first);
  // deque criado a partir de outro
  deque<int> second (first.begin(),first.end());
  mostraDeque("second", second);
  // deque criado com construtor de copia
  deque<int> third (second);
  mostraDeque("third", third);
  // o deque pode ser criado a partir de um array
  int v[] = { 2, 3, 5, 7, 11};
  deque<int> fourth (v, v + sizeof(v)/sizeof(int) );
  mostraDeque("fourth", fourth);
  // deque de inteiros vazio
  deque<int> fifth;
  mostraDeque("fifth", fifth);
  fifth.push_front( 1 ); //insere na frente
  mostraDeque("fifth", fifth);
  fifth.push_front( 2 );
  mostraDeque("fifth", fifth);
  fifth.push_front( 3 );
  mostraDeque("fifth", fifth);
  fifth.push_back( 4 );  //insere no final
  mostraDeque("fifth", fifth);
  fifth.push_back( 1 );
  mostraDeque("fifth", fifth);
  fifth.pop_front();      // remove elemento do inicio
  mostraDeque("fifth", fifth);
  fifth.pop_back();       // remove elemento do fim
  mostraDeque("fifth", fifth);
  fifth.clear();          // esvazia o Deque
  mostraDeque("fifth", fifth);
  return 0;
}
```

Lista de Exercícios

Exercício 1

1. Considere a definição da classe Turma abaixo.

```
#ifndef _TURMA_HPP
#define _TURMA_HPP
#include <string>
#include <vector>

using namespace std;

class Turma {
private:
  string codigo;
  int creditos;
  string nome, turma;
  vector <int> diaSemana;
  vector <string> horario;
public:
  Turma(string c="", int cr=0, string n="", string t="");
  void adicionaEncontro(int ds, string h);
  string obtemCodigo() const;
  string obtemNome() const;
  string obtemTurma() const;
  string str() const;
  bool operator==(Turma &t) const;
  bool operator<(Turma &t) const;
};
#endif
```

Exercício 1

1. (Continuação)
   Implemente a classe `Turma` em um arquivo `Turma.cpp` e também uma aplicação que crie uma lista (usando o contêiner `list` da STL) com no mínimo 5 turmas. Nessa aplicação, depois de adicionar as turmas na lista e de mostrar a lista, utilize o método `sort()` do contêiner `list` para ordenar a lista. Por fim, mostre a lista ordenada.
   O método `void adicionaEncontro(int ds, string h)` recebe um dia da semana e um horário, adicionando ambos com `push_back()`, respectivamente, nas variáveis de instância `vector<int> diaSemana` e `vector<string> horario`.
   Observe que a ordenação padrão necessita e utiliza os métodos sobrecarregados para os operadores "==" e "<" (estes métodos devem utilizar como critério de ordenação apenas o código da turma).

Exercício 2

2. Usando as implementações da classe `Turma` do Exercício 1, `Turma.hpp` e `Turma.cpp`, altere a aplicação para usar `vector` no lugar de `list`.
   Será necessário fazer alterações na estratégia de ordenação, pois o contêiner `vector` NÃO possui o método `sort()`, sendo necessário usar a função `sort()` declarada no arquivo de cabeçalho `algorithm`.
   Além de mostrar a lista ordenada pelo critério padrão (ou seja pelo código), mostre a lista ordenada usando o seguinte conjunto de critérios: nome (primeiro critério), código (segundo critério – será usado quando os nomes forem iguais) e turma (terceiro critério – será usado quando nomes e códigos forem iguais). Para isto implemente a comparação entre turmas em uma função `bool compTurma(Turma &t1, Turma &t2)`.

Exercícios 3, 4 e 5

3. Escreva um programa usando o contêiner `vector` da STL que declara um vetor de inteiros. Leia valores inteiros do terminal, enquanto eles NÃO forem negativos, armazenando os valores lidos no vetor. Por fim, imprima o vetor de inteiros na saída padrão.
4. Escreva um programa que explora STL, o qual captura uma sequência arbitrária de dígitos binários com `cin` e armazena em um contêiner. Enquanto o valor recebido é diferente de 1 ou 0. Considere que o primeiro valor inserido é o *bit* menos significativos. Ao final, apresente a representação binária informada e o valor decimal sem sinal desta representação binária.
5. Implemente um programa que lê `n` palavras da entrada-padrão, e mais uma palavra-chave. Você deve localizar e imprimir as palavras que foram digitadas que possuem a palavra-chave como substring.
   Dica: use o método `find()` da classe `string`. Este método retorna `string::npos` se não encontrar nada.

Exercício 6

6. Escreva um programa que recebe de entrada nomes e telefones correspondentes.
   a. A entrada deve ser `n` nomes e `n` telefones. Defina uma classe e seus métodos para armazenamento de tais informações.
   b. Na saída, imprima a lista `[<nome> <telefone>]` ordenada pelos nomes. Crie uma função para comparação. Explore o algoritmo sort de STL. Veja exemplo em: `http://www.inf.pucrs.br/~pinho/PRGSWB/STL/stl.html`.
   c. Após isso, o programa deve ler um nome da entrada padrão e imprimir seu telefone correspondente, se este existir. Se não existir, imprima uma mensagem apropriada.

Exercício 7

7. Uma pilha (*stack*) é um tipo abstrato de dados que tem associadas as seguintes operações:
   - `push()` – coloca um novo elemento no topo da pilha
   - `pop()` – retira um elemento do topo da pilha
   - `top()` – devolve o elemento no topo da pilha (sem a alterar!)
   - `clear()` – esvazia a pilha

   Defina:
   - Uma classe genérica em C++ que implemente uma pilha com todas as operações mencionadas e utiliza como estrutura de armazenamento um *array*. (Sugestão: utilize a classe `vector` da STL.)
   - Uma classe genérica em C++ que implementa uma pilha com todas as operações mencionadas e utiliza como estrutura de armazenamento uma lista ligada. (Sugestão: utilize a classe `list` da STL.)

Exercício 8

8. Considere um sistema de impressão que utiliza duas políticas de atendimento dos trabalhos de impressão:
   - P1 – imprime pela ordem de chegada;
   - P2 – imprime o menor trabalho primeiro.

   Assuma que o tempo de impressão por página é uma constante e que o número máximo de páginas é 50.
   - Gere 10 sessões de impressão e meça o tempo médio que um utilizador espera pelo seu trabalho se utilizar a política P1 em cada sessão. Para cada sessão de impressão gere aleatoriamente T trabalhos, cada um com um número de páginas também aleatório. Para o armazenamento dos trabalhos, utilize a classe queue da STL.
   - Repita o item anterior utilizando, agora, a política P2.
   - Identifique a melhor política, no sentido em que minimiza o tempo médio de espera dos utilizadores.

Fontes de consulta

Fontes de consulta

- Geral: `http://en.cppreference.com/w/cpp/container`
- `vector`: `http://en.cppreference.com/w/cpp/container/vector`
- `list`: `http://en.cppreference.com/w/cpp/container/list`

Creditos

Creditos

- Estas lâminas contêm trechos de materiais disponibilizados pelos professores Rafael Garibotti e Edson Moreno.

Soluções

Exercício 1: Turma.cpp

```
#include <sstream>
#include <iomanip>
#include "Turma.hpp"

Turma::Turma(string c, int cr, string n, string t) : codigo(c),creditos(cr),nome(n),turma(t) {}

void Turma::adicionaEncontro(int ds, string h) {
  diaSemana.push_back(ds);
  horario.push_back(h);
}

string Turma::obtemCodigo() const { return codigo; }
string Turma::obtemNome() const { return nome; }
string Turma::obtemTurma() const { return turma; }

string Turma::str() const {
  stringstream ss;
  ss << codigo << "-" << setw(2) << setfill('0') << creditos << "␣";
  ss << nome << "␣-␣Turma␣" << turma << ":";
  for (int i=0; i<diaSemana.size(); ++i) ss << "␣" << diaSemana[i] << horario[i];
  return ss.str();
}

bool Turma::operator==(Turma &t) const { return codigo == t.codigo; }
bool Turma::operator<(Turma &t) const { return codigo < t.codigo; }
```

Exercício 1: exercicio1.cpp

```
#include <iostream>
#include <list>
#include <iterator>
#include "Turma.hpp"

using namespace std;

int main() {
  Turma poo1("98718",4,"Programação␣Orientada␣A␣Objetos␣-␣Eco","10");
  poo1.adicionaEncontro(2,"AB");  poo1.adicionaEncontro(4,"AB");
  Turma poo2("98718",4,"Programação␣Orientada␣A␣Objetos␣-␣Eco","11");
  poo2.adicionaEncontro(2,"CD");  poo2.adicionaEncontro(4,"CD");
  Turma aed1("4645",4,"Algoritmos␣e␣Estruturas␣de␣Dados␣I","14");
  aed1.adicionaEncontro(2,"AB");  aed1.adicionaEncontro(4,"AB");
  Turma str("4620J",4,"Sistemas␣de␣Tempo␣Real","30");
  str.adicionaEncontro(3,"JK");  str.adicionaEncontro(5,"JK");
  Turma pp("46528",2,"Programação␣Paralela", "30");
  pp.adicionaEncontro(5,"LM");

  list<Turma> minhasTurmas;
  minhasTurmas.push_back(poo1);
  minhasTurmas.push_back(poo2);
  minhasTurmas.push_back(aed1);
  minhasTurmas.push_back(str);
  minhasTurmas.push_back(pp);

  for (list<Turma>::const_iterator it = minhasTurmas.begin(); it != minhasTurmas.end(); ++it)
      cout << (*it).str() << endl;

  minhasTurmas.sort();

  cout << endl;
  for (list<Turma>::const_iterator it = minhasTurmas.begin(); it != minhasTurmas.end(); ++it)
      cout << (*it).str() << endl;

  return 0;
}
```

Exercício 2: exercicio2.cpp

```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include "Turma.hpp"

using namespace std;

bool compTurma(Turma &t1, Turma &t2) {
  if ( t1.obtemNome() < t2.obtemNome() ) return true;
  if ( t1.obtemNome() > t2.obtemNome() ) return false;
  if ( t1.obtemCodigo() < t2.obtemCodigo() ) return true;
  if ( t1.obtemCodigo() > t2.obtemCodigo() ) return false;
  return t1.obtemTurma() < t2.obtemTurma();
}

int main() {
  Turma poo1("98718",4,"Programação Orientada A Objetos - Eco","10");  poo1.adicionaEncontro(2,"AB");  poo1.adicionaEncontro(4,"AB");
  Turma poo2("98718",4,"Programação Orientada A Objetos - Eco","11");  poo2.adicionaEncontro(2,"CD");  poo2.adicionaEncontro(4,"CD");
  Turma aed1("4645",4,"Algoritmos e Estruturas de Dados I","14");      aed1.adicionaEncontro(2,"AB");  aed1.adicionaEncontro(4,"AB");
  Turma str("4620J",4,"Sistemas de Tempo Real","30");                  str.adicionaEncontro(3,"JK");   str.adicionaEncontro(5,"JK");
  Turma pp("46528",2,"Programação Paralela", "30");                    pp.adicionaEncontro(5,"LM");

  vector<Turma> minhasTurmas;
  minhasTurmas.push_back(poo2);
  minhasTurmas.push_back(poo1);
  minhasTurmas.push_back(aed1);
  minhasTurmas.push_back(str);
  minhasTurmas.push_back(pp);

  for (int i=0; i<minhasTurmas.size(); ++i) cout << minhasTurmas[i].str() << endl;
  sort(minhasTurmas.begin(),minhasTurmas.end());
  cout << endl;  for (int i=0; i<minhasTurmas.size(); ++i) cout << minhasTurmas[i].str() << endl;
  sort(minhasTurmas.begin(),minhasTurmas.end(),compTurma);
  cout << endl;  for (int i=0; i<minhasTurmas.size(); ++i) cout << minhasTurmas[i].str() << endl;

  return 0;
}
```

Exercício 3: exercicio3.cpp

```
#include <iostream>
#include <vector>
#include <iterator>

using namespace std;

int main() {
  vector<int> vetor;
  int n;
  while (true) {
    cin >> n;
    if ( n < 0 ) break;
    vetor.push_back(n);
  }

  cout << "Vetor lido: " << endl;
  for (int i=0; i<vetor.size(); ++i)
      cout << vetor[i] << endl;

  cout << "Vetor lido: (usando um iterador)" << endl;
  for (vector<int>::const_iterator it = vetor.begin(); it != vetor.end(); ++it)
      cout << *it << endl;

  return 0;
}
```