Capítulo 8

Standard Template Library

A STL é uma parte extremamente importante da biblioteca padrão do C++. Na prática, programadores passam a maior parte do tempo programando somente com a STL, usando pouco as outras facilidades oferecidas pela biblioteca padrão completa. Na STL, estão declaradas estruturas de dados muito versáteis e diversos algoritmos úteis, de forma a facilitar a programação.

O uso dessa biblioteca evita que seja necessário, por exemplo, implementar e depurar listas encadeadas.Além disso, houve uma grande preocupação com o desempenho das classes e funções, de modo que dificilmente uma implementação própria pode oferecer vantagens consistentes de performance, sem falar nas facilidades que a STL provê.

8.1 Conhecendo a STL

Pode-se dividir a STL em três tipos de componentes: containers, iteradores e algoritmos. Containers são estruturas que contém objetos de outras classes, como por exemplo listas, pilhas, mapas, árvores, vetores, etc.. Para que se entenda como funcionam essas classes, é extremamente necessário que se esteja familiarizado com o conceito de templates.

Iteradores são, de certa forma, ponteiros. Na realidade eles não são ponteiros, porque eles não apontam simplesmente para uma palavra na memória, e existem operações que podem ser feitas sobre ponteiros que não podem ser feitas sobre iteradores, como aritmética de ponteiros, por exemplo. Entretanto, mesmo com estas diferenças, iteradores têm um uso semelhante a ponteiros: eles fornecem uma interface de acesso a elementos internos de containers. Algoritmos são, como o nome diz, algoritmos prontos que a STL fornece. Eles permitem, por exemplo, ordenar containers, obter elementos máximos de conjuntos, e outras coisas interessantes. Com algoritmos STL, pode-se acumular todos os valores de um vetor, essencialmente resolvendo um problema de integração numérica em uma linha de código.

Pode-se também aplicar uma função arbitrária a cada elemento de um con-

Contêiner	Categoria	Descrição
Vector	Sequencial	Inserções/Retiradas rápidas, apenas no fim. Acesso direto a qualquer elemento.
List	Sequencial	Inserções/Retiradas rápidas.
Deque	Sequencial	Inserções/Retiradas rápidas, no início e no fim. Acesso direto a qualquer elemento.
String	Sequencial	Vetor de caracters típico.
Set	Associativo	Pesquisa rápida. Duplicatas não permitidas.
Multiset	Associativo	Pesquisa rápida. Duplicatas permitidas.
Map	Associativo	Mapeamento um para um. Pesquisa rápida usando chaves. Duplicatas não permitidas.
Multimap	Associativo	Mapeamento um para um. Pesquisa rápida usando chaves. Duplicatas permitidas.
Stack	Adaptador	Primeiro a entrar, último a sair (FILO)
Queue	Adaptador	Primeiro a entrar, primeiro a sair (FIFO)
Priority_Queue	Adaptador	Mais alta prioridade, primeiro a sair

junto, também com uma linha de código. Uma característica interessante dos algoritmos STL é que eles são genéricos. Por exemplo, o algoritmo sort, que implementa ordenação de listas, é versátil o suficiente para ordenar quase qualquer vetor de elementos arbitrários. Este "quase" deve ser levado em consideração. Para que o sort funcione, a classe dos elementos internos ao vetor deve prover um operador $<$ bem definido. Além disso, não existem restrições, o que implica que o sort pode ordenar elementos de qualquer classe, desde que o predicado de ordem seja especificado para esta classe!

8.2 Contêiners

Contêiners são inseridos, de acordo com suas características, em três categorias: seqüenciais, associativos e adaptadores de containers. A justificativa para essa separação provém do uso de cada um dos grupos de containers. Abaixo, uma lista com os principais containers, alocados em seus respectivos grupos, seguidos de uma descrição breve de sua utilização.

8.3 Funções Membro Comuns

Existem alguns membros básicos que são comuns a todos os containers da STL. Estes membros permitem fazer checagens simples, como obter o número de elementos em um container, testar se este está vazio, e etc..

A lista abaixo mostra os métodos comuns mais importantes:

Contêiner	#include
Vector	<vector>
List	<list>
Deque	<deque>
String	<string>
Set e Multiset	<set>
Map e Multimap	<map>
Stack	<stack>
Queue e Priority Queue	<queue>

::size() retorna um size_ type (convertível para unsigned int) que representa o número de elementos do container

::empty() retorna um valor booleano dizendo se o container está vazio

::begin() retorna um iterador para o início do container

::end() retorna um iterador para um elemento além do final do container

::rbegin() retorna um iterador reverso para o início reverso do container (ou seja, o último elemento do container)

::rend() retorna um iterador reverso para um elemento antes do início do container

Além destes, os operadores típicos de comparação != e == também estão implementados para contêiners, bem como construtores de cópia.

8.4 Funções Membro Específicas

Obviamente, existem algumas funcionalidades específicas de cada container, que precisam ser implementadas com métodos não-genéricos. Um exemplo disso é a função clear(), que apaga todos os valores de um container, e só é função-membro dos containers seqüenciais ou associativos (também conhecidos como containers de primeira classe).

8.4.1 Arquivos de Cabeçalho

Para utilizar os containers da STL, é necessáro incluir os arquivos de cabeçalho daquele container no programa. Abaixo, os arquivos necessários para cada um dos containers discutido acima.

8.5 Vector

Um dos containers mais simples e utilizados da biblioteca de templates do C++ é a classe Vector. Apesar de ser fundamentalmente diferente de arrays em C, o vector serve para muitos dos mesmos fins. Uma das principais vantagens de se utilizar vector sobre arrays pré-determinados é que um vector não tem limite de tamanho (estritamente falando, um vector tem limite de tamanho máximo, mas este valor é limitado por uma constante normalmente desprezivelmente grande, definida pela quantidade de memória disponível na máquina. Este valor pode ser obtido com o método ::max _ _ size())

A maioria dos conceitos relativos aos objetos Vector já é bastante consolidada em programadores C, de modo que essa apostila mostrará, principalmente, a sintaxe e diferenças básicas entre o uso de arrays e de vetores da STL. Comecemos com um exemplo básico, que utiliza vectors apenas para armazenar e mostrar dados. A classe Aluno deve receber um número indefinido de notas, armazenar em um vector, e imprimir um histórico escolar, a partir desses dados. Sua implementação está em 8.1, 8.2.

Código 8.1: Header da classe Aluno

```
// trecho de aluno.h
#include <vector>
#include <iostream>
using std::vector;
using std::cout;
using std::endl;

class Aluno
{
  public:
  // Construtor
  Aluno();
  // Destrutor
  ~Aluno();
  // Adicionar nota
  void adicionarNota(int nota1);
  // Imprimir historico
  void imprimirHistorico();

  private:
  vector<int> notas;
};
```

8.5.1 Teste de Aprendizado

- Adicione na classe Aluno uma função calculaMedia().
- Troque o uso de push_back() pelo simples notas[n] = nota1.

Código 8.2: Classe usando um vetor

```
// trecho de aluno.cpp
void Aluno::adicionarNota(int nota1)
{
  notas.push_back(nota1);
}

void Aluno::imprimirHistorico()
{
  for(int i = 0; i < notas.size(); ++i)
  cout << "Nota: " << notas[i] << endl;
}
```

O que acontece?

Obs.: Utilize vector::size() para pegar o número de notas já adicionadas, e, conseqüentemente, o índice da próxima variável a ser adicionada.

8.6 Como lidar com capacidade ilimitada, ou como o Vector não é mágico

À primeira vista, vectors parecem desafiar as leis básicas da computação. Eles oferecem sempre o melhor de dois mundos. Temos garantias teóricas de que vectors fornecem um tamanho ilimitado de elementos, sabemos que estes elementos estão contíguos na memória, e sabemos que o acesso a qualquer elemento tem complexidade constante, bem como a inserção no final e a remoção do final. Não é nem um pouco trivial desenvolver uma estrutura de dados com todas estas capacidades ao mesmo tempo. De fato, se levarmos estas definições ao pé da letra, tal estrutura de dados é impossível.

O que a STL "esconde" aqui é que, na realidade, a inserção não é feita em tempo constante, mas sim em tempo constante amortizado. O funcionamento desta inserção é bastante simples: Quando inicializamos um vector, a STL reserva um espaço de memória contíguo de tamanho, digamos, n, para alguns poucos elementos, e mantém este espaço disponível para nosso vector.

Conforme vamos populando ele com elementos, este espaço vai sendo ocupado e, quando queremos inserir o n+1-ésimo elemento, faltaria espaço contíguo. Quando isto acontece, a STL procura outro local de memória contíguo, desta vez com 2n espaços, e move todo o conteúdo do vector para lá. Como o tamanho da nova área de memória aumenta exponencialmente, a probabilidade de que, dado um push_back() qualquer, este precise uma realocação, tende a 0 conforme fazemos mais e mais push_backs(). Por isso que chamamos a complexidade de inserção de constante amortizada.

É interessante notar que esta realocação de memória pode invalidar ponteiros sobre o container, e é por isso que ponteiros sobre containers não devem ser

usados. Este processo de alocação amortizada justifica um método bastante útil da classe vector, que é o método reserve(). Este método serve para que possamos determinar um tamanho para o espaço de memória inicial que o vector deve ocupar.

Por exemplo, se soubermos que nosso vector irá ocupar "aproximadamente" 100 elementos, podemos fazer uma chamada a reserve(100), que irá reservar um espaço de memória contígua de 100 elementos para nosso vector. Se nos garantirmos que a inserção irá só até o centésimo elemento, aí sim esta terá complexidade constante. Note, porém, que o método reserve() não popula o vector com nenhum elemento, de forma que esta chamada não altera o resultado de size() e empty()!

O exemplo 8.3 mostra outro detalhe que deve ser observado quando se trata de containers de tamanho arbitrário: apesar de que podemos, em princípio, inserir quantos elementos quisermos dentro de um vector, só podemos acessar aqueles que já foram inseridos. Isto funciona da mesma forma que arrays em C tradicional, ou seja, se um array tem 5 elementos, o acesso ao elemento 6 (inclusive ao 5) gerará comportamento indefinido. É importante observar que isto sumariza a diferença entre infinitos elementos e número arbitrário de elementos. Um vector deve ser pensado exatamente como um array C que pode ter seu tamanho modificado em tempo de execução.

Código 8.3: Demonstração de Vetores

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

using std::vector;
using std::cin;

int main(int argc, char **argv)
{
  vector<int> numeros;
  for(int i = 0; i < 4; ++i)
  {
    numeros.push_back(i);
  }
  vector<int> numeros2;
  numeros2.reserve(3);
  for(int i = 0; i < 4; ++i)
  {
    numeros2[i] = i;
  }

  numeros2[5000] = 5;

  return 0;
}
```

8.6.1 Observação

Ao testar no Dev-C++, que usa o gcc 3.4.2, o código acima só gerou erro ao acessar o elemento 5000. Porém, ao executar compilando com o g++ no Linux, um acesso ao item de índice 4 é suficiente para gerar um erro de execução. Dessa forma, esse tipo de acesso gera comportamento indefinido, que é suficiente para que seu uso não seja recomendado.

8.7 List

Além dos vetores, uma estrutura muito utilizada em programação é a lista encadeada. Observe que o nome list não quer dizer que é uma lista simplesmente encadeada. De fato, este template STL é uma lista duplamente encadeada (para uma lista especificamente simplesmente encadeada, use a estrutura não-padrão slist).

Listas são recomendadas para tarefas onde é necessário muita performance para inserções no início ou no final, e consulta no início ou no final. De fato, estas operações têm complexidade O(1). Operações arbitrárias no meio da lista, entretanto, como inserção, remoção e consulta, têm complexidade O(n). Uma vantagem das listas para os vectors, por outro lado, é que o tempo constante de listas de fato é constante, e não constante amortizado, pois a estrutura de dados subjacente a uma lista é um encadeamento de ponteiros.

Quando começamos a trabalhar com listas, a utilização de iteradores passa a ser necessária. Enquanto, para varrer um vector, é possível utilizar o operador [], da mesma forma que se varre arrays em C, quando queremos varrer o conteúdo de uma lista, somos obrigados a utilizar iteradores. O exemplo 8.4 mostra o conceito de iteradores em uma varredura linear simples de uma lista:

Inicialmente, a sintaxe destes comandos pode parecer complicada. Lembre-se, entretanto, de exatamente como funciona o comando for: O primeiro comando passado dentro dos parênteses é executado exatamente uma vez antes do laço ser processado. Neste caso, estamos declarando uma variável, chamada pos, de tipo list<int>::iterator, e setando esta variável ao valor especial foo.begin(), que aponta para o primeiro elemento da lista. O segundo comando no parêntese do for é um teste que será executado antes de cada passada do for, inclusive da primeira, e que se falhar, o for será terminado.

Aqui, estamos testando se o valor do iterador é igual ao valor fixo foo.end(), que aponta para um elemento além do último elemento da lista, de forma que este for varre a lista inteira. Finalmente, o último comando no parêntese do for chama o operador++ sobre o iterador, que está definido internamente e significa que o iterador deve receber o valor de seu sucessor imediato. Note que não existe o operador– para iteradores. Para varrer uma lista em ordem reversa, reverse_iterators devem ser utilizados. Veremos iteradores em mais detalhes na seção apropriada.

Código 8.4: Iterando sobre uma lista

```
#include <list>
#include <iostream>
using std::list;
using std::cout;
using std::cin;

int main()
{
  list<int> foo;
  for (list<int>::iterator pos = foo.begin(); // note que este
      for
    pos != foo.end(); ++pos){                 // esta separado em
        duas linhas!

  cout << (*pos) << endl;
  }

  return 0;
}
```

8.8 Outros Containers

A seguir, faremos um breve resumo sobre os outros containers mais específicos da STL, e ainda sobre um tipo de dados específico, o pair.

Deque Um deque é funcionalmente semelhante a um vector. Observe que deque significa double-ended queue, mas isto não quer dizer que um deque seja implementado como uma lista encadeada. De fato, deques são implementados como espaços contíguos na memória, de forma que inserções no meio levam tempo linear, mas acesso ao meio leva tempo constante. A principal diferença entre um vector e um deque é que o deque possui os métodos push_front e pop_front, que fazem papéis análogos ao push_back e pop_back, mas atuam na frente da lista. Da mesma forma que os vectors, estes métodos são implementados com uma técnica de tempo constante amortizado.

String Strings implementam um vetor de caracteres, para possibilitar formas de entrada e saída mais alto-nível do que null-terminated chars em C. Uma string possui complexidades semelhantes ao vector, de forma que também é mantida como área contígua de memória, mas o tipo de seus elementos é char. Existe uma variante interessante de strings, que são as wstrings, que permitem trabalhar com chars de 16 bits. Normalmente isto acontece quando a interface com algum outro programa exige a utilização de wide characters, ou seja, caracteres de 16 bits

Set Um set é, primeiramente, um container associativo. Isto significa que

acesso randômico aos seus elementos não é possível. Entretanto, inserção e remoção são executados em tempo O(log n). É importante notar que para containers associativos, não é possível especificar o local onde o elemento será adicionado, pois estes containers são implementados como árvores binárias balanceadas. Isto significa, adicionalmente, que os elementos são automaticamente guardados em ordem dentro do container. Observe que sets não permitem elementos duplicados. As funções únicas mais importantes de sets são insert, remove e find, que inserem, removem, e localizam, respectivamente, um elemento no set.

Multiset Muito semelhante ao set, a principal diferença entre eles é que enquanto o set não permite colocação de elementos duplicados, um multiset permite.

Map Um Map é um container mais interessante, no sentido de que ele permite buscas do tipo chave-informação. Dessa forma, os elementos de um map são pares (std::pair) onde o primeiro elemento representa a chave de busca e o segundo o valor desejado. O map é implementado também como uma árvore binária balanceada onde a ordenação é feita com respeito às chaves. Dessa forma, é possível buscar um valor informando somente uma chave, com complexidade O(log n). Não permitem duplicatas

Multimap Multimaps são análogos a multisets, no sentido de que são maps que permitem multiplicidade de elementos.

Modificadores de containeres Stacks, Queues e Priority Queues não são containeres propriamente ditos, mas sim modificadores que atuam sobre containeres. Dessa forma, para especificarmos uma stack, por exemplo, primeiramente temos que definir um container normal (um vector, por exemplo), e depois construir a stack informando este vector. O que acontece é que os métodos acessados pelo objeto stack serão limitados a acesso e escrita ao primeiro elemento da pilha, como desejamos. O mesmo acontece para queues e priority queues.

Pair Um Pair não é um container. Ao invés disso, um pair é uma estrutura de dados declarada pela STL que permite formarmos pares de tipos genéricos. Para especificar um par inteiro-string, por exemplo, devemos fazer o seguinte: pair<int,string> foo. Os tipos de dados que podem formar pares são arbitrários, e para acessarmos elementos do pair, existem os atributos .first e .second.

8.9 Iteradores

A definição mais simples de iteradores é razoavelmente óbvia: iteradores iteram sobre o conteúdo de containers STL. Ou seja, iteradores são tipos de dados

específicos que a STL implementa para possibilitar uma forma de acesso uniforme a elementos de qualquer tipo de container. A idéia de interface uniforme é muito importante aqui: se fôssemos implementar um conjunto de containers diferentes a partir do zero, teríamos, possivelmente, classes para listas, filas, arrays, árvores binárias, e etc.

Cada uma de nossas classes possuiria um método diferente para acessar os elementos de dentro dos nossos containers, e o programador que fosse utilizá-los teria que aprender cada um individualmente. O conceito de iteradores da STL resolve este conflito, pois fornece uma única forma de acesso uniforme a elementos de qualquer container.

De certa forma, iteradores podem ser pensados como ponteiros para os elementos dos containers. Esta analogia deve ser tomada com muito cuidado, porém, pois existem coisas que podem ser feitas com ponteiros (apesar de que não são necessariamente recomendadas) que a STL não fornece. O exemplo mais clássico de uma aplicação assim é aritmética de ponteiros.

Já vimos, na seção que trata com listas, um exemplo de declaração de iteradores. Por isso, vamos mostrar um exemplo que utiliza iteradores sobre sets. Fica bastante claro que a estrutura do comando é exatamente igual. A única coisa que muda é o tipo do iterador, que passa a ser do tipo set<int>::iterator.

Código 8.5: Demonstrando Iteradores

```
#include <set>
#include <iostream>

using std::list;
using std::cout;
using std::cin;

int main()
{
  set<int> foo;
  for (list<int>::iterator pos = foo.begin(); pos != foo.end();
      ++pos)
  {
        cout << (*pos) << endl;
  }
  return 0;
}
```

No exemplo 8.5, podemos notar alguns detalhes interessantes: em primeiro lugar, veja que a STL também segue, de certa forma, a analogia de que iteradores são ponteiros. Isto quer dizer que o operator* está definido para iteradores, e tem a mesma semântica que ele tem para ponteiros (ele retorna o valor apontado pelo ponteiro, ou, neste caso, pelo iterador). Além disso, o operador++ também está definido, e ele serve para passar de um iterador para o próximo. Observe que, mesmo que sets não sejam lineares (lembre que são

árvores binárias), o operador++ está bem definido. Finalmente, devemos observar o significado dos comandos foo.begin() e foo.end() no exemplo acima. foo.begin() é um iterador específico que aponta para o primeiro elemento do container trabalhado. O foo.end(), entretanto, não aponta para o último elemento, e sim para um elemento além do último, seguindo de certa forma o conceito de que índices em C variam de 0 até n-1, e não até n. Abaixo, mostramos os tipos principais de iteradores e suas utilidades:

::iterator Este tipo de iterador declara os iteradores mais comuns, que acessam elementos do container de forma seqüencial da esquerda para a direita e que podem modificar os elementos apontados por eles

::reverse_iterator Este tipo de iterador serve para varrer containers de forma reversa. Como o operador– não está declarado para iteradores normais, não seria possível iterar reversamente por um container sem este iterador especial. Observe que o operador++ está definido para este tipo de iterador da mesma forma que para ::iterator's, mas para ::reverse_iterator's a varredura é feita em ordem reversa!

::const_iterator Semelhante ao ::iterator, mas com a restrição de que os elementos só podem ser acessados, e não modificados, utilizando este iterador. O uso deste iterador existe para podermos varrer containers que foram declarados const. Se tentarmos varrer um const vector<int> com um ::iteretor normal, encontraremos um erro de compilação.

::const_reverse_iterator Análogo ao ::const_iterator, mas se aplica a varredura reversa do container. Além disso, cada container tem definido quatro iteradores especiais que servem para limitar laços for, entre outros usos.

Iremos assumir, para os exemplos abaixo, a existência de um vector<int> chamado foo, mas de fato estes iteradores estão definidos para qualquer container de qualquer tipo.

foo.begin() este iterador é do tipo ::iterator, e aponta para o primeiro elemento do container

foo.end() iterador do tipo ::iterator, aponta para um elemento além do último elemento do container. Acesso a (*foo.end()) gera comportamento indefinido.

foo.rbegin() iterador do tipo ::reverse_iterator, e aponta para o último elemento do container.

foo.rend() iterador do tipo ::reverse_iterator que aponta para um elemento antes do primeiro elemento do container. Acesso a *(*foo.rend())* gera comportamento indefinido.

8.10 Algoritmos

Algoritmos também são uma parte bastante importante da Standard Template Library, já que implementam funções usadas muito freqüentemente de forma eficiente, facilitando muito o trabalho do programador. Um exemplo de algoritmo extremamente utilizado é o de ordenação. Através desse algoritmo, fica fácil organizar um vetor (ou algum outro tipo de estrutura) usando qualquer tipo de relação de ordem. No exemplo 8.6, usaremos a função sort para ordenar inteiros de forma crescente.

Código 8.6: STL Sort

```
#include <algorithm>

int main()
{
  vector<int> numeros;

  for(int i = 0; i < 6; ++i)
  {
     numeros.push_back(50 - 10*i);
  }
  cout << "Imprimir fora de ordem" << endl;

  for(int i = 0; i < 6; ++i)
  {
          cout << numeros[i] << endl;
  }

  std::sort(numeros.begin(), numeros.end());
  cout << "Imprimir em ordem" << endl;

  for(int i = 0; i < 6; ++i)
  {
    cout << numeros[i] << endl;
  }
}
```

O algoritmo sort é normalmente o algoritmo mais utilizado da STL e, portanto, merece alguma atenção especial. Internamente, o padrão STL dita que o algoritmo utilizado para ordenamento é o algoritmo introsort. Basicamente, o introsort ordena os elementos inicialmente utilizando um quicksort, e quando o nível de recursão atinge um determinado limite, o algoritmo passa a executar um heapsort.

A vantagem desta abordagem é que a complexidade deste algoritmo é garantidamente O(n log n). Outro detalhe importante é que o algoritmo utilizado pela STL não é um algoritmo de ordenamento estável. Para um algoritmo estável, utilize a alternativa stable_sort. Abaixo, mostramos alguns algoritmos comuns que a STL implementa, com uma breve descrição de sua utilidade:

count(iterator first, itertor last, const T &value) a função count da STL recebe dois iteradores (semelhante ao sort) e mais um valor, e retorna o número de ocorrências do valor value dentro do intervalo definido pelos dois iteradores

count_if(iterator first, iterator last, predicate pred) esta função é semelhante à função count, mas ela retorna o número de elementos dentro do intervalo definido pelos iteradores que satisfaz à condição imposta pelo predicado pred, que deve ser uma função booleana de um argumento do tipo do container dos iteradores utilizados

max_element(iterator first, iterator last) retorna um iterador para o elemento de valor máximo dentro do intervalo definido por first e last. Esta função utiliza o comparador operator< do tipo do container para fazer as comparações.

min_element(iterator first, iterator last) semelhante a max_element, mas retornando um iterador para o elemento mínimo

max(const T &v1, const T &v2) dados dois valores de um mesmo tipo, como por exemplo max(1,2), retorna o valor máximo entre eles

min(const T &v1, const T &v2) semelhante a max, mas retornando o valor mínimo

8.10.1 Usando o Algoritmo Sort para qualquer Ordenação

O algoritmo sort é extremamente amplo, de modo que não se limita a ordenações crescentes e decrescentes de inteiros. Nessa seção, veremos como utilizar funções próprias para ordenar estruturas através do sort. Para a demonstração, será usado o exemplo 8.7, que coloca os inteiros em ordem decrescente.

8.11 Usando a STL com Classes Próprias

A biblioteca padrão do C++ tem como alicerce, em sua construção, a facilidade de extensão do seu uso. A partir disso, é redundante dizer que as estruturas criadas podem ser utilizadas não só com os tipos básicos da linguagem, mas também com classes criadas pelo programador. No exemplo 8.8, ordenaremos nossos Livros pelo ano de publicação (o maior antes), e em caso de igualdade, pelo título do livro.

Como se pode ver, basta declarar um operador do tipo <, que determina qual será o critério utilizado para definir se um objeto Livro é ou não menor do que o outro. Depois, é só usar a função sort no vetor. O C++ ordena automaticamente o vetor utilizando o critério definido pelo operador.

Código 8.7: Usando Função para Sort

```
#include <algorithm>
// fun
bool antes(int i, int j) { return (i > j); }

int main()
{
  vector<int> numeros;
  for(int i = 0; i < 6; ++i)
  {
    numeros.push_back(10*i);
  }

  cout << "Imprimir fora de ordem" << endl;

  for(int i = 0; i < 6; ++i)
  {
    cout << numeros[i] << endl;
  }

  std::sort(numeros.begin(), numeros.end(), antes);
  cout << "Imprimir em ordem" << endl;
  for(int i = 0; i < 6; ++i)
  {
    cout << numeros[i] << endl;
  }
}
```

Código 8.8: STL com Classes Próprias

```
// trecho de livro.cpp
// Operador <
// (vai determinar se um Livro deve ser menor na ordem ou nao)

bool Livro::operator<(const Livro &book) const
{
  if(anoDePublicacao > book.getAno())
  return true;
  else
  {
   if(anoDePublicacao == book.getAno())
        {
          if(titulo < book.getTitulo())
          return true;
        }
  }
  return false;
};

/*
trecho de main.cpp

std::sort(livros.begin(), livros.end());

Ok! os livros estarao ordenados.
*/
```

8.11.1 Teste de Aprendizado

1. Crie o operador $<$ na classe Aluno, e ordene os alunos pela média das notas

2. Crie uma classe Time, com número de vitórias, empates e derrotas, e organize n times em uma tabela de classificação usando: Maior número de pontos(vitórias*3 + empates), menos número de jogos, maior número de vitórias

8.12 Últimas Considerações

A seguir, falamos sobre algumas considerações importantes para escrever qualquer código que utilize a STL. Estes itens são resumos de alguns capítulos do livro Effective STL, Revised, de Scott Meyers. Seguir estes itens é extremamente importante, para garantir que o programa execute de forma previsível, já que devido à complexidade da STL, se não tomarmos cuidado, o programa pode gerar comportamento indefinido facilmente. O número entre colchetes é a numeração de cada item no livro original.

- Sempre garanta que o objeto usado como tipo de um contâiner possui cópia bem definida[3]: Grande parte da filosofia da STL é baseada em cópias dos objetos que utilizamos. Por exemplo, se declaramos um vector<foo> bar, onde foo é uma classe declarada por nós, quando inserimos um elemento neste vector (com um push_back, por exemplo), a STL não insere exatamente o elemento que queríamos, mas sim uma cópia deste.

  Para tipos de dados básicos, como ints, float, e etc., isto não é importante. Para classes arbitrárias, por outro lado, isso pode ser um problema. Para estas classes, quando uma cópia é feita, o compilador invoca métodos específicos desta classe: o seu construtor de cópia ou seu operador de cópia, também conhecido como operator=. O detalhe aqui é que se nós não implementarmos estes métodos em nossa classe, o compilador assume uma implementação padrão que pode ou não fazer o que queremos. Outro problema é que se nós de fato implementarmos estes métodos, devemos nos garantir de que eles estão corretos, senão as operações feitas sobre contâiners da STL podem gerar comportamento inesperado e dificultar enormemente a depuração do programa.

- Em contâiners, use o membro empty(), ao invés de checar size() $==$ 0[4]; O motivo para esta recomendação é bastante simples. A operação empty() tem complexidade O(1), enquanto o teste de size() $==$ 0 pode, em alguns casos, custar O(n). Para contâiners numerosos, isto se torna um problema.

- Lembre de chamar delete em iteradores antes de apagar o contâiner[7]; Suponha que tenhamos, por exemplo, um vector<int *> foo (um vector de ponteiros para inteiros). Além disso, imagine que para popular este vector, utilizamos um laço for que aloca dinamicamente cada elemento com um operador new e coloca o valor no vector, como o exemplo:

```
vector<int *> foo;
for (int i = 0; i < 5; ++i)
{
  int *x = new int; *x = 3; foo.push_back(x);
}
```

- Cada elemento do vector foo representa uma posição de memória dinâmica alocada pelo Sistema Operacional. Quando apagarmos este vector (seja explicitamente ou seja porque seu escopo local está esgotado), o compilador não chama os deletes correspondentes a esses new's! Isto significa que, ao declarar um código como o acima, nós somos obrigados a fazer um outro laço semelhante, chamando os deletes correspondentes a cada iterador:

  ```
  for (int i = 0; i < 5; ++i){ delete foo[i]; }
  ```

  O exemplo que foi mostrado usa somente vector, mas este raciocínio vale para qualquer container STL.

- Prefira vectors e strings a vetores dinamicamente alocados[13]; Ao utilizar vetores dinamicamente alocados, o programador deve se responsabilizar inteiramente pelo gerenciamento de memória dinâmica feito. Para programas simples, isto pode parecer trivial, mas para um projeto mais complexo, gerenciamento de memória dinâmica é um problema bastante custoso.

  O programador deve garantir, em seu código, que existe exatamente uma chamada delete para cada chamada new correspondente. Se esta chamada delete não estiver presente, haverá com certeza vazamento de memória. Se, por outro lado, a chamada for feita duplicada, o comportamento, de acordo com a definição C++ é, novamente, indefinido. Com estes problemas, não parece fazer sentido adotar vetores dinâmicos ao invés de vectors ou strings, que fazem todo este gerenciamento automaticamente e de forma segura.

- Garanta que operadores para contâiners associativos retornam falso para valores iguais[21]; É bastante comum, como vimos em um exemplo acima, declarar operadores booleanos para poder implementar contâiners associativos (como sets, multisets, etc.) de classes arbitrárias. Dessa forma, implementaremos um operator< dentro de uma classe arbitrária qualquer. Em princípio, a STL permite que o comparador utilizado em um set não seja necessariamente o operator<, mas sim qualquer função que

informarmos, desde que esta satisfaça algumas condições. Assim, é possível, em princípio, estabelecer um set onde o comparador utilizado é um operator$<=$, ao invés de um operator$<$. Isto é uma péssima idéia, e o motivo disto é simplesmente que a STL espera que elementos iguais retornem falso a uma comparação, para poder, entre outras coisas, não inserir duplicatas em sets. Assim, se nosso comparador retornar qualquer coisa diferente de falso para valores iguais, a STL gerará comportamento indefinido.