Iteradores

Prof. Salles Magalhaes

Introducao

Ate o momento vimos tres tipos de listas:

- Contiguidade

- Simplesmente encadeadas

- Duplamente encadeadas

Podemos percorrer as listas de forma **uniforme** utilizando os “iteradores” e funcoes como next/prev.

Iteradores permitem acesso, modificacao de elementos, percurso (iteracao), etc.

Introducao

Alem da uniformidade, iteradores ajudam na eficiencia: operacoes permitidas normalmente sao as eficientes para suas estruturas de dados (em geral *O(1)*).

Importancia de iteradores sera melhor observada em estruturas de dados mais complexas (ex: arvores)

Introducao

Os iteradores mostrados como exemplo não são exatamente iguais a iteradores presentes, por exemplo, na STL.

Hoje: como criar iteradores mais úteis, versáteis e similares a iteradores “de verdade”.

Problema: necessidade do uso do identificador do objeto, funcoes (next, prev, etc).

Solução: iteradores mais “independentes” da estrutura de dados (“lembram” a qual estrutura de dados pertencem)

Versao inicial dos nossos iteradores

```
int main() {
  MyList2<char> v;
  v.push_back('a');
  for(int i=1;i<10;i++) {
    v.push_back('a'+i);
  }

  for(MyList2<char>::iterator it = v.begin();it!=v.end();it = v.next(it)) {
    cout << v.deref(it) << " ";
  }
  cout << endl;

  return 0;
}
```

Nova versao do iterador (similar a do STL)

```
MyList2<char> v;
v.push_back('a');
for(int i=1;i<10;i++) {
  v.push_back('a'+i);
}

for(MyList2<char>::iterator it = v.begin();it!=v.end();it++) {
  cout << (*it) << " ";
}
cout << endl;

return 0;
```

Exemplos de diferenca

	Primeira versao dos iteradores	Iteradores da STL
Incremento	v = v.next(it)	it++
Decremento (*)	v = v.prev(it)	it--
Copia (do iterador)	it = it2	it = it2
Derreferencia (leitura/escrita)	v.derref(it)	*it; , cout << *it ; , *it = 5;
Pular 9 elementos (*)	-	it = it+9
Imprimir o elemento 9 posicoes a frente (*)	-	cout << *(it+9) << endl;
Comparar iteradores		if(it == it2) , if(it < it2) (*)

* quando disponivel

Lista por contiguidade

Implementação simples usando ponteiros!

Exemplo

Implementacao: não precisa mudar NADA na implementacao!

- Ponteiros se comportam exatamente como queremos (lista por contiguidade)

Veja implementacao e testes (TestaMyVec.cpp)

Lista por contiguidade

Esse tipo de iterador e’ chamado de iterador de acesso aleatorio bidirecional

- Permite uma grande variedade de operacoes

- Incremento usando inteiros, decremento usando inteiros, comparacao, etc

Veja a tabela em : http://www.cplusplus.com/reference/iterator/

Pergunta: nosso iterador para elementos do MyVec pode perder validade (enquanto o MyVec ainda existe)?

Veja o exemplo “IteradorMyVecValidade.cpp”

Lista por contiguidade

Não ha garantias de que os iteradores para MyVec continuarao validos após insercoes no vetor! (mesmo se as posicoes apontadas pelo vetor não forem modificadas!!)

Exercicio:

- Por que essa garantia não existe?

- Teria como modificar a classe iterador para garantir essa validade?

Lista duplamente encadeada

Primeira versao de iterador para lista encadeada: ponteiro para nodo

- Podemos implementar um “iterador” mais avancado de forma similar a lista por contiguidade?

Lista duplamente encadeada

Primeira versao de iterador para lista encadeada: ponteiro para nodo

- Podemos implementar um “iterador” mais avancado de forma similar a lista por contiguidade?

-- Não: nodos não estao contiguos na memoria!

-- “Iteracao” precisa seguir apontadores next/prev, derreferencia precisa...

Exercicio: e’ possivel fazer com que o iterador (implementado com ponteiro) se comporte de forma diferente?

Lista duplamente encadeada

Solucao:

- Criar classe iterador

- Iterator contera ponteiro para nodo

- Operacoes: sobrecarga de operadores

Veja solucao em: MyVecNewIterator.h (observe a funcao testaMyList2 em TestaMyList.cpp)

Lista simplesmente encadeada: implementacao e’ similar, mas não e’ possivel ter operacoes de decremento (iterador “não bidirecional”), a não ser que aceitamos fazer tal operacao de forma ineficiente...

Lista duplamente encadeada

Exercicios:

- E’ possivel decrementar o iterador end() atualmente ? Se não, como tornar isso possivel?

- E’ possivel fazer acesso aleatorio com o iterador de lista? Por que sim/nao?

- Os iteradores continuam validos após modificacoes na lista?

- Por que o tipo de retorno da operacao de derreferencia e’ uma referencia?

Nossos iteradores/estr. de dados vs STL

Em C++ (na biblioteca STL) ha classes similares a MyVec (vector) e MyList2 (list).

Os iteradores dessas estruturas de dados sao similares aos nossos.

Iterador para MyVec/vector: iteradores de acesso aleatorio

- Suportam comparacao do tipo <, >, == , etc ; suportam acesso aleatorio ; sao bidirecionais

Iterador para MyList2/list: iteradores bidirecionais

- Suportam operacoes do tipo ++, --, comparacao do tipo ==, != .

Por que não suportam < , > ?

(algumas) Vantagens do uso de iteradores

Acesso uniforme!!! (isso ficara mais claro com estr. de dados mais complexas)

Em TestaMyList.cpp, compare as funcoes testaMyList2 e testaMyVec: sao basicamente identicas!

Note que, devido a uniformidade, podemos usar templates para gerar funcoes que atuam de forma similar em estruturas de dados diferentes (veja testaContainer)

(algumas) Vantagens do uso de iteradores

A partir do C++11 (leia o manual do seu compilador para aprender a ativar essa versao) foi introduzido o “range-based for loop”

for(Tipo t: container) { //leia ”para cada elemento t no container”

}

A ideia e’ que o containter sera iterado (do inicio ao fim) e cada valor do container sera atribuido a t em cada iteracao (se t for uma referencia → t representara uma referencia para cada elemento do container)

Veja um exemplo em: RangeFor.cpp

(algumas) Vantagens do uso de iteradores

O range-based for loop funciona para qualquer container iteravel

Note que ele funciona ate mesmo em arrays automaticos (que não possuem exatamente “iteradores”, mas podem ser iterados usando ponteiros)

Podemos usar esse for ate mesmo nas nossas classes MyVecNewIterator e MyList2NewIterator (veja exemplo em: TestaMyListRangeFor.cpp)

Exercicio: Como o C++ sabe iterar na nossas classes customizadas??

(algumas) Vantagens do uso de iteradores

Como o C++ sabe iterar na nossas classes customizadas??

R: gracas ao: begin(), end(), ++ , != e * (o for usa apenas esses 5 métodos)

Exercicio: o range-based for vai sempre acessar os elementos do seu container do comeco ate o final?

IMPORTANTE: a não ser que seja explicitamente dito o contrario, em Estruturas de Dados você não poderá usar o range-based for (precisamos praticar o uso de iteradores!)

Outros tipos de iteradores

Ha outros tipos de iteradores que podemos criar/usar

Por exemplo, na STL ha o conceito de iterador reverso, que pode ser utilizado para iterar em uma estrutura de dados “de trás para a frente” de forma mais simples.

> Exercicio: por que não podemos simplesmente fazer algo como o abaixo para iterarmos de forma reversa?
>
> ```
> for(iterator it = v.end(); it!= v.begin(); it--)  ?
> ```