Introdução a STL

Paulo Ricardo Lisboa de Almeida

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Standard Template Library

- A STL (Standard Template Library) contém diversas implementações já prontas
  - Pertence a biblioteca padrão do C++
  - Algoritmos muito eficientes
    - Listas encadeadas, vetores redimensionáveis, mapas, árvores, …
  - Evita que “reinventemos a roda” toda vez que um programa precisa ser feito
- Veja em
  - www.cplusplus.com/reference/stl/

STL

- Vamos usar a classe list da STL como exemplo
  - É implementada como uma **lista duplamente encadeada**
  - Veja as funções membro da classe list em
    www.cplusplus.com/reference/list/list/

Instanciando

- Inclua o header da estrutura desejada
  - Para *list*

    #include<list>

- Para criar uma instância

  std::list<TIPO_ITENS_LISTA> minhaLista;

A STL é implementada utilizando Templates (conceito de polimorfismo paramétrico). Estudaremos em detalhes no futuro. Por enquanto só “aceite” que devemos passar o tipo da lista entre < >

Algumas funções membro da classe list

- push_back
  - Adiciona o elemento no final da lista
- push_front
  - Adiciona o elemento no início da lista
- remove
  - Remove determinado elemento da lista
- empty
  - Retorna verdadeiro se a lista está vazia, ou falso caso contrário

Exemplo

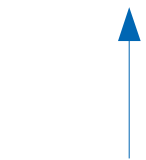
```
#include <iostream>
#include <list>

#include "Pessoa.hpp"

int main(){
        std::list<int> lista;
        lista.push_back(10);
        lista.push_back(20);
        lista.push_front(30);

        return 0;
}
```

Acessando os elementos da lista

- Para acessar os elementos da lista, precisamos de um **iterador**
  - Mais um conceito que estudaremos mais profundamente no futuro
  - Por enquanto, vamos aprender apenas a **usar** um iterador
- Você pode considerar um iterador como um objeto capaz de percorrer um contêiner (lista, vetor, árvore, …)
  - O iterador **aponta** para o elemento atual
  - Pode receber o próximo elemento
    - Criamos um loop, onde fazermos o iterador apontar para o próximo elemento a cada iteração, até terminar a lista

Declarando um iterator

```
std::TIPO_CONTAINER<TIPO_OBJETOS_CONT>::iterator itInt;
```



list, vector, ...

Int, double, Pessoa,...

Exemplo

```
#include <iostream>
#include <list>

#include "Pessoa.hpp"

int main(){
	std::list<int> lista;
	std::list<int>::iterator itInt;

	lista.push_back(10);
	lista.push_back(20);
	lista.push_front(30);

	for(itInt = lista.begin(); itInt!= lista.end(); itInt++)
		std::cout << *itInt << std::endl;

	return 0;
}
```

Exemplo

```
#include <iostream>
#include <list>

#include "Pessoa.hpp"

int main(){
	std::list<int> lista;
	std::list<int>::iterator itInt;

	lista.push_back(10);
	lista.push_back(20);
	lista.push_front(30);

	for(itInt = lista.begin(); itInt!= lista.end(); itInt++)
		std::cout << *itInt << std::endl;

	return 0;
}
```

Incrementar um iterador é o mesmo que passar para o próximo item

retorna um iterador para o início da lista

retorna um iterador para o fim da lista

itInt **aponta** para o item atual

UDESC

Teste você mesmo

- Crie uma lista de Ponteiros para Pessoa
  - Insira 3 pessoas na lista
  - Itere nessa lista, exibindo o nome de cada pessoa

Teste você mesmo

```
std::list<Pessoa*> pessoas;
pessoas.push_back(new Pessoa{"Joao", 15});//adicionando no final da lista
pessoas.push_back(new Pessoa{"Maria", 16});//adicionando no final da lista
Pessoa* p3{new Pessoa{"Pedro", 20}};
pessoas.push_front(p3);//adicionando no início da lista

for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)//a lista contém ponteiros para pessoas, e não pessoas
        std::cout << (*itPes)->getNome() << std::endl;//o itPes é um ponteiro que aponta pra um ponteiro de pessoa!
```

UDESC

Removendo um item

- Para remover um item da lista temos duas opções principais
  - remove
    - Recebe o objeto que deve ser removido (via referência)
  - erase
    - Recebe um *iterator* do item que deve ser removido

Exemplo

```
std::list<Pessoa*> pessoas;
pessoas.push_back(new Pessoa{"Joao", 15});
pessoas.push_back(new Pessoa{"Maria", 16});
Pessoa* p3{new Pessoa{"Pedro", 20}};
pessoas.push_front(p3);

for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
        if((*itPes)->getNome() == "Pedro")//continua até encontrar Pedro
                break;


if(itPes != pessoas.end()){//encontrou ou chegou no final da lista?
        //3 OPÇÕES!!!
}
```

Exemplo

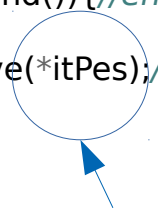
```
std::list<Pessoa*> pessoas;
pessoas.push_back(new Pessoa{"Joao", 15});
pessoas.push_back(new Pessoa{"Maria", 16});
Pessoa* p3{new Pessoa{"Pedro", 20}};
pessoas.push_front(p3);

for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
        if((*itPes)->getNome() == "Pedro")//continua até encontrar Pedro
                break;


if(itPes != pessoas.end()){//encontrou ou chegou no final da lista?
        delete p3;
        pessoas.remove(p3);//passando p3, que já é um ponteiro para pessoa
}
```

Nesse caso, o loop é irrelevante, e só gastamos processamento

UDESC

Exemplo

```
std::list<Pessoa*> pessoas;
pessoas.push_back(new Pessoa{"Joao", 15});
pessoas.push_back(new Pessoa{"Maria", 16});
Pessoa* p3{new Pessoa{"Pedro", 20}};
pessoas.push_front(p3);

for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
        if((*itPes)->getNome() == "Pedro")//continua até encontrar Pedro
                break;


if(itPes != pessoas.end()){//encontrou ou chegou no final da lista?
        delete *itPes;
        pessoas.remove(*itPes);//passando o ponteiro para pessoa que precisa ser apagado
}
```

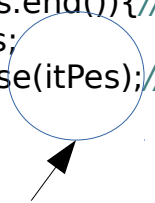


Pegamos o valor do *iterator*, que é justamente um ponteiro para Pessoa. O resultado é o mesmo que o exemplo anterior.

UDESC

Exemplo

```
std::list<Pessoa*> pessoas;
pessoas.push_back(new Pessoa{"Joao", 15});
pessoas.push_back(new Pessoa{"Maria", 16});
Pessoa* p3{new Pessoa{"Pedro", 20}};
pessoas.push_front(p3);

for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
        if((*itPes)->getNome() == "Pedro")//continua até encontrar Pedro
                break;


if(itPes != pessoas.end()){//encontrou ou chegou no final da lista?
        delete *itPes;
        pessoas.erase(itPes);//passando o iterador do item
}
```



Passamos o *iterator* para o erase.

UDESC

Uma questão de desempenho

- No caso de uma lista encadeada, a função *remove* é obrigada a visitar todos os itens da lista até encontrar o item que precisa ser removido
- Qual o problema que criamos com a construção a seguir?

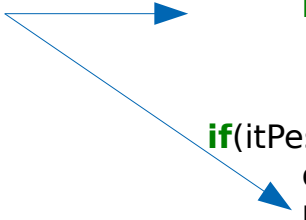
```
std::list<Pessoa*> pessoas;
pessoas.push_back(new Pessoa{"Joao", 15});
pessoas.push_back(new Pessoa{"Maria", 16});
Pessoa* p3{new Pessoa{"Pedro", 20}};
pessoas.push_front(p3);

for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
        if((*itPes)->getNome() == "Pedro")//continua até encontrar Pedro
                break;


if(itPes != pessoas.end()){//encontrou ou chegou no final da lista?
        delete *itPes;
        pessoas.remove(*itPes);//passando o ponteiro para pessoa que precisa ser apagado
}
```

Uma questão de desempenho

- No caso de uma lista encadeada, a função *remove* é obrigada a visitar todos os itens da lista até encontrar o item que precisa ser removido
- Qual o problema que criamos com a construção a seguir?

```
std::list<Pessoa*> pessoas;
pessoas.push_back(new Pessoa{"Joao", 15});
pessoas.push_back(new Pessoa{"Maria", 16});
Pessoa* p3{new Pessoa{"Pedro", 20}};
pessoas.push_front(p3);

for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
        if((*itPes)->getNome() == "Pedro")//continua até encontrar Pedro
                break;


if(itPes != pessoas.end()){//encontrou ou chegou no final da lista?
        delete *itPes;
        pessoas.remove(*itPes);//passando o ponteiro para pessoa que precisa ser apagado
}
```

Percorremos para encontrar o item, e depois percorremos mais uma vez para remover



UDESC

Uma questão de desempenho

- A função erase recebe um iterator que aponta para o objeto a ser removido, e não o objeto em sí
  - Um iterator armazena internamente informações de posição
  - O item pode ser removido **diretamente!**
  - Em uma lista duplamente encadeada, isso é ainda mais eficiente
    - A remoção de um item em uma lista duplamente encadeada não exige a realocação dos itens posteriores ao item removido

Então ...

Como você julga o trecho?

```
std::list<Pessoa*> pessoas;
pessoas.push_back(new Pessoa{"Joao", 15});
pessoas.push_back(new Pessoa{"Maria", 16});
Pessoa* p3{new Pessoa{"Pedro", 20}};
pessoas.push_front(p3);



delete p3;
pessoas.remove(p3);
```

Note que estamos utilizando remove e o objeto, mas não gastamos tempo com um loop para procurar o objeto! O remove se vira para procurar!

Então ...

Isso está correto!

```
std::list<Pessoa*> pessoas;
pessoas.push_back(new Pessoa{"Joao", 15});
pessoas.push_back(new Pessoa{"Maria", 16});
Pessoa* p3{new Pessoa{"Pedro", 20}};
pessoas.push_front(p3);



delete p3;
pessoas.remove(p3);
```

Note que estamos utilizando remove e o objeto, mas não gastamos tempo com um loop para procurar o objeto! O remove se vira para procurar!

Então ...

Como você julga o trecho?

```
std::list<Pessoa*> pessoas;
	pessoas.push_back(new Pessoa{"Joao", 15});
	pessoas.push_back(new Pessoa{"Maria", 16});
	Pessoa* p3{new Pessoa{"Pedro", 20}};
	pessoas.push_front(p3);


	for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
		if((*itPes)->getNome() == "Pedro")//continua até encontrar Pedro
			break;

	if(itPes != pessoas.end()){//encontrou ou chegou no final da lista?
		delete *itPes;
		pessoas.erase(itPes);//passando o iterador do item
	}
```

Nós mesmos procuramos pelo item. E passamos o iterador para o erase, que não vai gastar tempo procurando mais uma vez.

UDESC

Então ...

Isso está correto!

```
std::list<Pessoa*> pessoas;
	pessoas.push_back(new Pessoa{"Joao", 15});
	pessoas.push_back(new Pessoa{"Maria", 16});
	Pessoa* p3{new Pessoa{"Pedro", 20}};
	pessoas.push_front(p3);


	for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
		if((*itPes)->getNome() == "Pedro")//continua até encontrar Pedro
			break;

	if(itPes != pessoas.end()){//encontrou ou chegou no final da lista?
		delete *itPes;
		pessoas.erase(itPes);//passando o iterador do item
	}
```

Nós mesmos procuramos pelo item. E passamos o iterador para o erase, que não vai gastar tempo procurando mais uma vez.

UDESC

Então ...

Como você julga o trecho?

```
std::list<Pessoa*> pessoas;
	pessoas.push_back(new Pessoa{"Joao", 15});
	pessoas.push_back(new Pessoa{"Maria", 16});
	Pessoa* p3{new Pessoa{"Pedro", 20}};
	pessoas.push_front(p3);


	for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
		if((*itPes)->getNome() == "Pedro")//continua até encontrar Pedro
			break;

	if(itPes != pessoas.end()){//encontrou ou chegou no final da lista?
		delete p3;
		pessoas.remove(p3);
		OU
		pessoas.remove(*itPes);
	}
```

UDESC

Então ...

Isso é burrice!

```
std::list<Pessoa*> pessoas;
        pessoas.push_back(new Pessoa{"Joao", 15});
        pessoas.push_back(new Pessoa{"Maria", 16});
        Pessoa* p3{new Pessoa{"Pedro", 20}};
        pessoas.push_front(p3);


        for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
                if((*itPes)->getNome() == "Pedro")//continua até encontrar Pedro
                        break;

        if(itPes != pessoas.end()){//encontrou ou chegou no final da lista?
                delete p3;
                pessoas.remove(p3);
                OU
                pessoas.remove(*itPes);
        }
```

UDESC

Removendo itens no loop

- O exemplo anterior remove apenas a primeira referência de Pedro na Lista
- Se a lista conter mais de um Pedro?
  - Vamos criar um algoritmo para isso

Problemas?

- Você consegue ver o problema com o algoritmo a seguir?

```
//removendo todas pessoas chamadas João
for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
    if((*itPes)->getNome() == "Joao")
        pessoas.erase(itPes);
```

Problemas?

- Removemos o item apontado pelo iterador
  - E depois tentamos continuar utilizando o iterador (que foi deletado) no loop.
  - **Falha de segmentação** (memória do programa corrompida)
  - O C++ não vai te avisar sobre esse seu erro, nem em tempo de compilação, nem execução
    - As estruturas da STL não implementam mecanismos de segurança, como os Fail-Fast e Fail-Safe implementados no Java, C# e Python
      - Esses mecanismos custam tempo de processamento
      - C++ sempre vai deixar você dar um tipo no pé, caso a outra opção seja gastar processamento desnecessariamente

```
//removendo todas pessoas chamadas João
for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
      if((*itPes)->getNome() == "Joao")
            pessoas.erase(itPes);//erro
```

Utilizando o retorno de erase

- A função erase retorna um iterador que aponta para o elemento posterior ao removido
  - Basta utilizar esse valor retornado para corrigir o problema

```
//removendo todas referências a João
	itPes = pessoas.begin();
	while(itPes != pessoas.end()){
		if((*itPes)->getNome() == "Joao"){
			delete *itPes;
			itPes = pessoas.erase(itPes);//itPes recebe o próximo item válido da lista
		}else{
			itPes++;
		}
	}
```

Utilizando delete nos itens da lista

- A lista de pessoas armazena ponteiros
  - Precisamos liberar a memória manualmente

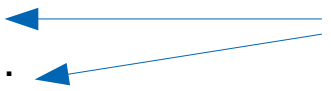
```
//Removendo as demais pessoas da memória
for(itPes = pessoas.begin(); itPes != pessoas.end(); itPes++)
    delete *itPes;

//agora a lista ainda possui os ponteiros, que apontam para endereços inválidos
//para limpar a lista, basta
pessoas.clear();
```

Atenção

- **Não use list para tudo**
  - Entenda que cada estrutura de dados tem seus custos e benefícios
    - Benefícios de uma lista encadeada
      - É muito flexível, e remover/adicionar itens é computacionalmente barato
    - Custos
      - Exige ponteiros extras para cada nodo internamente (+ memória), pode causar invalidações de cache, e é muito custoso acessar um item pelo seu índice (ex.: acesse o item 5)
- Use as estruturas de dados corretas de acordo com o seu problema
  - Problema clássico que sempre encontro em programas Java
    - Programadores usam ArrayList para tudo, sem nem sequer parar para pensar em como um ArrayList é implementado internamente
      - Será que o ArrayList é a solução para todos os problemas?
- Você pode aprender mais sobre estruturas de dados e seus custos nas disciplinas de
  - Estruturas de dados
  - Análise de algoritmos

Exercícios para a próxima aula

- Modifique a classe Disciplina
  - Uma disciplina possui uma membro que é lista de alunos
    - Use *list* ou *vector*
  - Adicione as funções membro:
    - adicionarAluno(Pessoa* aluno);
    - removerAluno(Pessoa* aluno);
    - removerAluno(unsigne long cpf);



Sobrecarga de funções

Referências

- http://www.cplusplus.com/reference/stl/
- DEITEL, P.; DEITEL, H. **C++ how to Program**. [S.l.]: Pearson, 2017. ISBN 9780134448237
- STROUSTRUP, B. **The C++ Programming Language**. Pearson Education, 2013. ISBN 9780133522853.