Templates

Paulo Ricardo Lisboa de Almeida

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

Templates

- Estamos usando templates o tempo todo
  - Exemplo
    - *std::list<int> listaInteiros;*
      - é uma instância de uma **especialização** de um template

Mas o que é um template

- Considere o exemplo de uma classe que representa uma pilha a seguir
  - Note que a classe foi implementada no .hpp
    - Vai facilitar os próximos passos
    - Isso forçou algumas mudanças no makefile
      - Verifique

Pilha

```
#ifndef PILHA_HPP
#define PILHA_HPP
class Pilha{
    public:
        static const int MAX_PILHA{10};

        Pilha():topo{-1}{ }
        ~Pilha(){ }

        bool push(const int valor){
            if(this->estaCheia())
                return false;
            this->topo++;
            this->pilha[this->topo] = valor;
            return true;
        }

        bool pop(int* const retorno){
            if(!this->estaVazia()){
                *retorno = this->pilha[topo];
                this->topo--;
                return true;
            }
            return false;
        }

        bool estaVazia() const{
            if(topo < 0)
                return true;
            return false;
        }

        bool estaCheia() const{
            if(topo >= MAX_PILHA -1)
                return true;
            return false;
        }
    private:
        int pilha[MAX_PILHA];
        int topo;
};
#endif
```

UDESC

Pilha

- A classe representa uma pilha de inteiros
- Com o que a prendemos até o momento, como proceder se precisarmos de uma pilha de inteiros, e também de uma pilha de doubles?

Pilha

- A classe representa uma pilha de inteiros
- Com o que a prendemos até o momento, como faríamos se precisássemos de uma pilha de inteiros, mas também de uma pilha de doubles?
  - Contros+C / Control+V na classe e a modificamos para doubles
    - Teríamos duas classes, a PilhaInteiros e a PilhaDoubles
    - No mínimo, contraprodutivo

Templates

- Para resolver esse problema e criar uma pilha **genérica**, vamos usar Templates
- Vamos definir que a classe não funciona para inteiros, mas sim para um tipo **T qualquer**

Comumente chamamos de T (de **T**emplate), e se tivermos mais de um template, os próximos são U, V ...

Templates

- O primeiro passo na classe Pilha é substituir onde necessário o tipo dos dados de **int** para **T**
- Exemplo

Um tipo “T qualquer”

```
#ifndef PILHA_HPP
#define PILHA_HPP
class Pilha{
	public:
		//...
	private:
		T pilha[MAX_PILHA];
		int topo;
};
#endif
```

Templates

- Onde mais devemos alterar int por T?

Templates

Empilhamos um T

Recebemos um ponteiro para T

```
//...
    public:
        bool push(const T valor){
            if(this->estaCheia())
                return false;
            this->topo++;
            this->pilha[this->topo] = valor;
            return true;
        }

        bool pop(T* const retorno){
            if(!this->estaVazia()){
                *retorno = this->pilha[topo];
                this->topo--;
                return true;
            }
            return false;
        }
//...
    private:
        T pilha[MAX_PILHA];
        int topo;
//...
```

UDESC

Templates

- O próximo passo é indicar que T é um template
  - Indicamos que T deve ser substituído por um tipo real
- Fazemos isso logo antes da definição da classe através de
  template <typename T>
- Na classe pilha:

```
#ifndef PILHA_HPP
#define PILHA_HPP

template <typename T>
class Pilha{
    //...
};
#endif
```

Criando instâncias

```
#include <iostream>

#include "Pilha.hpp"

int main(){
	int retorno;
	Pilha<int> p;
	p.push(1);
	p.push(2);
	p.push(3);
	p.push(4);

	while(!p.estaVazia()){
		p.pop(&retorno);
		std::cout << retorno << std::endl;
	}


	std::cout << "Fim" << std::endl;

	return 0;
}
```

Criando instâncias

Desejamos uma pilha de inteiros. O compilador vai substituir T por int, e gerar uma classe de pilhas especializada em inteiros.

```
#include <iostream>

#include "Pilha.hpp"

int main(){
	int retorno;
	Pilha<int> p;
	p.push(1);
	p.push(2);
	p.push(3);
	p.push(4);

	while(!p.estaVazia()){
		p.pop(&retorno);
		std::cout << retorno << std::endl;
	}


	std::cout << "Fim" << std::endl;

	return 0;
}
```

Criando instâncias

Podemos instancias pilhas de **qualquer tipo!**

```
#include <iostream>

#include "Pilha.hpp"
#include "Pessoa.hpp"

int main(){
	int retorno;
	Pilha<int> p;
	Pilha<double> p;
	Pilha<Pessoa*> p;
	//...

	return 0;
}
```

Internamente

- As templates em C++ são **compilados para o tipo específico**
- No exemplo ao lado, internamente tempos 3 pilhas compiladas
  - para inteiro, para double, e para Pessoa*

```
#include <iostream>

#include "Pilha.hpp"
#include "Pessoa.hpp"

int main(){
	int retorno;
	Pilha<int> p;
	Pilha<double> p;
	Pilha<Pessoa*> p;
	//...

	return 0;
}
```

Internamente

- As templates em C++ são **compilados para o tipo específico**
- Vantagem
  - O código de máquina gerado é tão eficiente quanto se tivéssemos criado cada uma das versões da pilha individualmente e manualmente
  - Type-safety: o compilador consegue verificar erros de conversão de tipo durante a compilação
- **Desvantagens?**

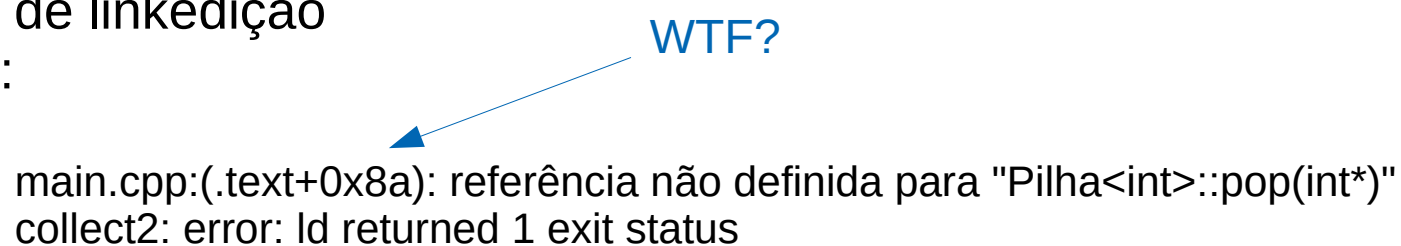
```
#include <iostream>

#include "Pilha.hpp"
#include "Pessoa.hpp"

int main(){
	int retorno;
	Pilha<int> p;
	Pilha<double> p;
	Pilha<Pessoa*> p;
	//...

	return 0;
}
```

Internamente

- Desvantagens
  - A compilação se torna mais complexa
  - O compilador precisa da definição completa da classe para substituir os parâmetros e gerar o binário
  - **Por isso definimos tudo no .hpp**
- Por experiência, o compilador pode gerar erros difíceis de entender quando usamos templates
  - Principalmente devido ao fato de que muitos erros são gerados apenas na etapa de linkedição
  - Exemplo:

Internamente

- Desvantagens
  - A compilação se torna mais complexa
  - O compilador precisa da definição completa da classe para substituir os parâmetros e gerar o binário
  - **Por isso definimos tudo no .hpp**
- Por experiência, o compilador pode gerar erros difíceis de entender quando usamos templates
  - Principalmente devido ao fato de que muitos erros são gerados apenas na etapa de linkedição
  - Exemplo:



Internamente

- Desvantagens
  - *Code Bloat*: o binário final pode se tornar muito grande
    - Lembre-se que no nosso exemplo temos um binário para cada versão da pilha
  - Isso pode ser resolvido facilmente através de herança
    - Veremos adiante
    - "*Então, a derivação pode ser usada para reduzir o problema [de code bloating …] ? Essa técnica se provou eficaz em usos reais. Pessoas que não usam esse tipo de técnica descobriram que o código replicado por custar megabytes de espaço mesmo em programas de tamanho moderado*" (Stroustrup, 1994).

Definição no .cpp

- Definimos a classe completa no .hpp para facilitar a compilação
- Podemos definir no .cpp também
  - Existem vários métodos para isso
  - Mas muitos deles complicam o processo de compilação e linkedição
  - Alguns exigem verdadeiras gambiarras, como incluir o .cpp no .hpp
- Por conta disso, é comum implementarmos as classes que envolvem templates no .hpp
  - As classes da STL por exemplo são implementadas completamente nos .hpp
    - Veja em DEITEL e DEITEL, 2017.

Mais de um template

- Uma classe pode ter mais de um template
- A ideia é a mesma
- Exemplos

```
template <typename T,typename U>
class MinhaClasse{
	//...
};
```

Para comparação

- Em **Java**, um conceito similar a templates é implementado através dos Generics
- Para o programador, é quase a mesma coisa
  - Mas internamente, as coisas são muito diferentes

Generics em Java

- Os Generics do Java não geram uma versão da classe para cada especialização
  - Ao invés disso, internamente todos os tipos genéricos comportam o tipo **object**, que é uma superclasse da qual todas as demais classes derivam
  - Exemplo

```
List<Integer> lista1;
List<Double> lista2;
```

Apesar da declaração, as duas listas contém o mesmo tipo de objeto internamente (object) e são iguais. Em Java os “*Templates*” apontam para qualquer coisa.

Generics em Java

- Como internamente os *Generics* do Java não fazem distinção entre os objetos
  - Vantagens
    - Não temos *code bloat*, já que apenas **uma versão** da classe precisa ser compilada
  - Esconde muitas das complexidades do programador
    - Torna a vida do programador mais simples
  - **Desvantagens?**

Generics em Java

- Como internamente os *Generics* do Java não fazem distinção entre os objetos
  - Desvantagens
    - Desempenho reduzido com ponteiros internos extras para as indireções
    - É mais difícil assegurar o type-safety
      - No entanto compiladores modernos Java fazem um bom trabalho quanto a isso
    - Os *Generics* geram problemas de **type erasure**
      - Internamente, os tipos são apagados e tudo vira uma coisa só (Object)
      - A Oracle vende isso **como se fosse uma vantagem!!!**
      - docs.oracle.com/javase/tutorial/java/generics/erasure.html
    - Não podemos usar Generics em tipos primitivos (int, double, float, ...)
      - Nunca se perguntou o motivo do Java fazer a distinção entre Double e double???

Templates versus Generics - Sobrecarga

Sobrecarga **permitida** em **C++**, já que se tratam de protótipos diferentes - os parâmetros são diferentes

```
void imprimirLista(std::list<int>);

void imprimirLista(std::list<double>);
```

Não é permitido em **Java**, já que internamente ambas listas são “List<Object>” e nesse caso os protótipos (assinaturas) são iguais

```
void imprimirLista(List<Integer>);

void imprimirLista(List<Double>);
```

Templates versus Generics - Sobrecarga

- Assumindo que T é um Template (C++) ou Generic (Java)

Permitido em C++. O compilador verifica se o tipo T sendo especializado possui construtor default, e vai efetuar a chamada (caso não tenha construtor default, temos um erro de compilação)

```
T* ptr{new T};
```

Não é permitido em Java. A JVM teria que resolver isso em tempo de execução, mas qual construtor chamar, se internamente T é um “Object qualquer”??? Esse problema é comumente contornado com uso do Design Pattern Factory.

```
T classe = new T();
```

C#

- C# implementa mecanismos mais robustos que o Java para seus templates
  - Pesquise
    - Um bom lugar para iniciar é aqui
      - http://www.jprl.com/Blog/archive/development/2007/Aug-31.html

Python

- E em Python?

Python

- E em Python?

UDESC

Templates

- O conceito de templates se torna ainda mais poderoso com
  - Herança
  - Funções polimórficas
  - Sobrecarga de operadores
- Estudaremos esses conceitos no futuro

Exercícios

1. Implemente uma classe para uma **Fila** utilizando Templates. Internamente, o dado membro que representa a Fila pode ser implementado usando uma list, deque ou outro membro da STL que você julgar interessante (no exemplo da aula usamos um vetor para representar a pilha, o que gerou o problema de que a pilha não pode ter mais que *n* elementos).

Referências

- Stroustrup, B. **The Design and Evolution of C++.** Pearson Education. 1994. ISBN 9780135229477
- 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.