Se parece fácil demais, algo deve estar errado.

Templates

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Templates

Estamos usando templates o tempo todo.

Exemplo:

```
std::list<int> listaInteiros;
```

é uma instância de uma **especialização de um template.**

O que é um template?

Considere o exemplo de uma classe que representa uma pilha simples.

Note que **a classe foi implementada no .hpp.**

Vai facilitar os próximos passos.

Forçou algumas mudanças no makefile.

Verifique.

Pilha.hpp

```cpp
#ifndef PILHA_HPP
#define PILHA_HPP
class Pilha{
   public:
       constexpr static 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
```

Pilha

A classe representa uma pilha de inteiros.

Com o que aprendemos 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 aprendemos até o momento, como proceder se precisarmos de uma pilha de inteiros, e também de uma pilha de doubles?

Control+C / Control+V na classe e a modificamos para doubles.

Teríamos duas classes, a PilhaInteiros e a PilhaDoubles.

No mínimo, contraprodutivo.

Ideia de jerico: popular a classe de ponteiros para void.

Templates

Vamos definir que a classe funciona **para um tipo T qualquer.**

Templates

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

Exemplo:

```
#ifndef PILHA_HPP
#define PILHA_HPP

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

Templates

Onde mais?

```
#ifndef PILHA_HPP
#define PILHA_HPP

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

Templates

Um T é empilhado.

O valor recebido pelo pop é armazenado e um ponteiro para T.

```
#ifndef PILHA_HPP
#define PILHA_HPP

class Pilha{
   public:
       constexpr static int MAX_PILHA{10};

        //...

       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;
};
#endif
```

Templates

O próximo passo é indicar que T é um template.

Indicamos que **T deve ser substituído por um tipo real em tempo de compilação.**

Isso é feito logo antes da definição da classe através de

`template <typename T>`

```
#ifndef PILHA_HPP
#define PILHA_HPP

template <typename T>
class Pilha{
   public:
       //...
   private:
       T pilha[MAX_PILHA];
       int topo;
};
#endif
```

Instanciando

```cpp
#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 << "\n";
   }

   std::cout << "Fim\n";

   return 0;
}
```

Instanciando

O **compilador** vai substituir T por int, e gerar uma *Pilha* **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 << "\n";
   }

   std::cout << "Fim\n";

   return 0;
}
```

Instanciando

Você pode criar pilhas de qualquer tipo!

```
int main(){
    Pilha<int> p1;
    Pilha<int*> p2;
    Pilha<Pessoa> p3;
    Pilha<Pilha<Pessoa*>> p4;

    //...

    return 0;
}
```

Polimorfismo

Templates oferecem um tipo de polimorfismo.

- Um algoritmo pode ser expresso de forma a comportar vários tipos.
- Polimorfismo: do Grego “muitas formas”.
- Existem outros tipos de polimorfismo em C++: exemplo -> funções virtuais.

Templates oferecem um modelo de **programação genérica.**

Em C++ templates são chamados de polimorfismo de tempo de compilação, ou polimorfismo paramétrico.

- Veja mais no capítulo 24.1 de Bjarne Stroustrup. The C++ Programming Language (2013).

Polimorfismo Paramétrico

Em C++ os templates oferecem polimorfismo paramétrico.

- Resolvido em tempo de compilação.
- Os templates em C++ são compilados para o tipo específico.

No exemplo, internamente temos 3 pilhas compiladas.

- Para inteiro, para double, e para Pessoa.

```
int main(){
    Pilha<int> p1;
    Pilha<double> p2;
    Pilha<Pessoa> p3;

    //...

    return 0;
}
```

Polimorfismo Paramétrico

As templates em C++ são compilados para o tipo específico.

+ Vantagens:
    + 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?

Polimorfismo Paramétrico

```
int main(){
    Pilha<int> p1;
    Pilha<double> p2;
    Pilha<Pessoa> p3;

    //...

    return 0;
}
```

As templates em C++ são compilados para o tipo específico.

- 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 através de herança e especialização (Veja capítulo 25.3 de Stroustrup (2013) e Stroustrup 1994).

"Então, essas técnicas podem ser usadas para reduzir o problema [de code bloating ...]. 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, 2013).

Polimorfismo Paramétrico

```
int main(){
    Pilha<int> p1;
    Pilha<double> p2;
    Pilha<Pessoa> p3;

    //...

    return 0;
}
```

As templates em C++ são compilados para o tipo específico.

- 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 através de herança e especialização (Veja capítulo 25.3 de Stroustrup (2013) e Stroustrup 1994).
    - **Por experiência**, o compilador pode gerar erros difíceis de entender quando usamos templates.
        Muitos erros são gerados apenas na etapa de linkedição.

WTF? ⟶

```
main.cpp:(.text+0x8a): referência não definida para "Pilha<int>::pop(int*)"
collect2: error: ld returned 1 exit status
```

Polimorfismo Paramétrico

```
int main(){
    Pilha<int> p1;
    Pilha<double> p2;
    Pilha<Pessoa> p3;

    //...

    return 0;
}
```

As templates em C++ são compilados para o tipo específico.

- 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 através de herança e especialização (Veja capítulo 25.3 de Stroustrup (2013) e Stroustrup 1994).
    - **Por experiência**, o compilador pode gerar erros difíceis de entender quando usamos templates.
        Muitos erros são gerados apenas na etapa de linkedição.
    - Compilação mais complicada e lenta.

Definição no hpp

Para o caso de templates, a implementação das classes **deve ficar no hpp.**

Caso contrário, o compilador não conseguiria ter informações o suficiente para compilar as classes.

Existem várias gambiarras para implementar no .cpp, mas nunca encontrei alguma que valesse a pena.

As “soluções” geralmente envolvem injetar o .cpp no .hpp via include de alguma forma, o que no final dá na mesma que uma implementação 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.

Exemplo:

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

Non-type parameters

Um Template pode ser definido para receber um **non-type parameter**.

- Um valor integral (ex.: 3.1415, 700, ‘B’, ...) ou uma enumeração.
- Um ponteiro ou referência para:
  - Um objeto.
  - Uma função.

Exemplo

Usar um non-type parameter para especificar o tamanho máximo da pilha.

Sem overheads!

O non-type parameter é substituído em tempo de compilação, de forma similar a um #define em C.

Exemplo

Uma pilha de até 5 elementos.

```cpp
#include <iostream>

#include "Pilha.hpp"

int main(){
   int retorno;
   Pilha<int, 5> p;
   p.push(1);
   p.push(2);
   p.push(3);
   p.push(4);

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

   std::cout << "Fim\n";

   return 0;
}
```

non-type parameter

```cpp
#ifndef PILHA_HPP
#define PILHA_HPP

template <typename T, int MAX_PILHA>
class Pilha{

   //...

   private:
       T pilha[MAX_PILHA];
       int topo;
};
#endif
```

Uma solução melhor

size_t é definido em cstddef

Cuidado! Comparação extras necessárias pois estamos comparando um valor com sinal e um sem.

```
#ifndef PILHA_HPP
#define PILHA_HPP

#include <cstdlib>

template <typename T, std::size_t MAX_PILHA>
class Pilha{
   public:

        // ...

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

size_t

size_t é substituído por algum tipo que possui ao menos 16 bits (a partir da especificação C++11).

Representa o maior tamanho teórico possível de um objeto, incluindo vetores.

A partir do C++14, uma tipo em que o tamanho não pode ser representado por um size_t é considerado **mal formado**.

É substituído pelo tipo correto durante a compilação.

O tipo correto depende da arquitetura da máquina.

Exemplo: o tamanho máximo em um x86 é diferente do de um microcontrolador.

Maior portabilidade.

Em meu x86-64, o size_t é traduzido para um unsigned long.

Array STL

A classe Array da STL utiliza non-type parameters em sua implementação.

Com isso a classe array é muito similar a um array convencional.

www.cplusplus.com/reference/array/array

```
#include <iostream>
#include <array>

int main (){
    std::array<int,5> myarray = { 2, 16, 77, 34, 50 };

    std::cout << "myarray contains:";
    for (std::array<int,5>::const_iterator it{myarray.begin()};
            it != myarray.end(); it++ )
         std::cout << ' ' << *it;
     std::cout << '\n';

    return 0;
}
```

Argumentos default nos Templates

Os templates podem possuir valores default.

Mesmo conceito com parâmetros default em funções.

Caso nada seja especificado, o default é usado.

Disponível a partir do C++11.

Exemplo

Instancie uma pilha padrão!

```cpp
#include <iostream>

#include "Pilha.hpp"

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

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

    std::cout << "Fim\n";

    return 0;
}
```

```cpp
#ifndef PILHA_HPP
#define PILHA_HPP

#include <cstdlib>

template <typename T = int, std::size_t MAX_PILHA = 10>
class Pilha{
    public:

        // ...

    private:
        T pilha[MAX_PILHA];
        int topo;
};
#endif
```

Exemplo

Dica: em compiladores que não implementam o C++17, pode ser necessário escrever `Pilha<> p;`

```cpp
#include <iostream>

#include "Pilha.hpp"

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

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

   std::cout << "Fim\n";

   return 0;
}
```

Antes do C++17 mesmo a pilha padrão precisa dos ‹ › em sua definição. Essa verborreia foi eliminada no C++17, com a introdução do class template argument deduction en.cppreference.com/w/cpp/language/class_template_argument_deduction.

```cpp
#ifndef PILHA_HPP
#define PILHA_HPP

#include <cstdlib>

template <typename T = int, std::size_t MAX_PILHA = 10>
class Pilha{
   public:

       // ...

   private:
       T pilha[MAX_PILHA];
       int topo;
};
#endif
```

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.

Para comparação

Os Generics do Java **não geram uma versão da classe para cada especialização**.

Internamente todos os tipos genéricos comportam o tipo Object, que é uma superclasse da qual todas as classes derivam.

Uma espécie de ponteiro para void.

```
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.

Para comparação

Os Generics do Java **não geram uma versão da classe para cada especialização**.

Internamente todos os tipos genéricos comportam o tipo Object, que é uma superclasse da qual todas as classes derivam.

Uma espécie de ponteiro para void.

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

Por que o Java não aceita uma List‹**double**›, e precisamos fazer essa gambiarra com **Double**?

Generics em Java

Os Generics do Java **não geram uma versão da classe para cada especialização**.

Internamente todos os tipos genéricos comportam o tipo Object, que é uma superclasse da qual todas as classes derivam.

Uma espécie de ponteiro para void.

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

double é um tipo nativo, que por não ser uma classe, não deriva de Object. Como todo tipo genérico internamente aponta para um Object, uma lista não pode conter doubles, apenas Doubles!

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.
    + Compilação 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, ...).

C++ versus Java

Sobrecarga **permitida** em **C++**, já que se tratam de protótipos 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 são iguais

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

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

C++ versus 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"??? Gambiarra para contornar: Design Pattern Factory method.

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

C#

C# implementa mecanismos mais robustos que o Java para seus templates.

Pesquise.

Um bom lugar para iniciar é aqui:

www.jprl.com/Blog/archive/development/2007/Aug-31.html

Python

Python não implementa mecanismos para utilização de templates.

Como não possui tipos, assume que o interpretador vai “se virar” independente do tipo do item que vamos armazenar na nossa pilha, por exemplo.

Concepts

Em Java, é possível fazer `<T extends MinhaClasse>`.

Indicando que `T` pode ser qualquer classe, desde que `T` estenda `MinhaClasse` (conceito de herança).

Isso é possível a partir do C++20 através de **Concepts**.

Leia em Bjarne Stroustrup (2013) onde são dadas algumas ideias sobre como os compiladores deveriam fazer isso, e as possíveis dores de cabeça que seriam criadas ao se implementar em uma linguagem compilada.

Veja em github.com/AnthonyCalandra/modern-cpp-features#concepts exemplos dessa ideia agora funcional no C++20.

Note que apesar de parecerem iguais para o programador, as abordagens do C++ e do Java são bastante diferentes para o problema.

Vídeo

Veja o vídeo demonstrando o problema de code bloat:

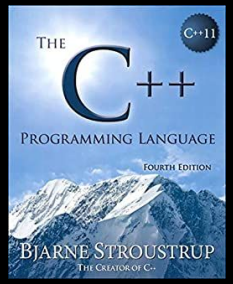
www.youtube.com/watch?v=4UoTr08zI0w

Exercícios

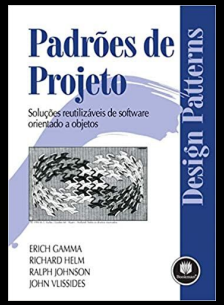
1. Estude sobre funções paramétricas (na aula vimos como parametrizar uma classe inteira - como fazer isso com funções individuais?).
2. Tente usar `pragma once` no seu Template. O que acontece (especialmente no g++)? Dica: pragma once não é padronizado e pode gerar conflitos com templates.
3. Crie uma classe para uma fila circular. Internamente, sua classe deve armazenar os dados da fila em um vetor simples (como feito para a pilha da aula de hoje). Você deve usar templates.
   Faça um main e um makefile para testar.

Referências

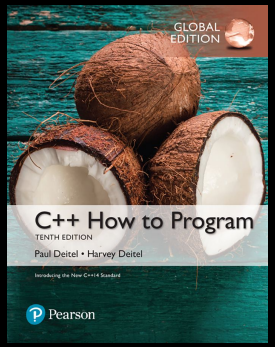
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



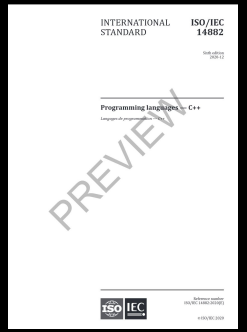
Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.



Deitel, H. M., Deitel, P. J. C++: como programar. 10a ed. Pearson Prentice Hall. 2017.



ISO/IEC 14882:2020 Programming languages - C++: www.iso.org/obp/ui/#iso:std:iso-iec:14882:ed-6:v1:en



Licença