Responda: por que o tiranossauro não consegue coçar as costas?

Concepts

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Pergunta

Para que serve a classe a seguir?

Ela está correta?

```
#ifndef UTIL_HPP
#define UTIL_HPP

#include <list>

template <typename T>
class Util {
  public:
   Util() = default;
   virtual ~Util() = default;

   static T somar(const std::list<T> lista) {
       T soma{0};
       // a partir do C++ 11, podemos fazer um foreach
       for (T const& valor : lista) soma += valor;
       return soma;
   }
};
#endif
```

Exemplo

```cpp
#include <iostream>

#include "util.hpp"

int main() {
   std::list<int> valores;
   valores.push_back(1);
   valores.push_back(2);
   valores.push_back(3);

   std::cout << Util<int>::somar(valores) << '\n';

   std::cout << "Fim!!!\n";
   return 0;
}
```

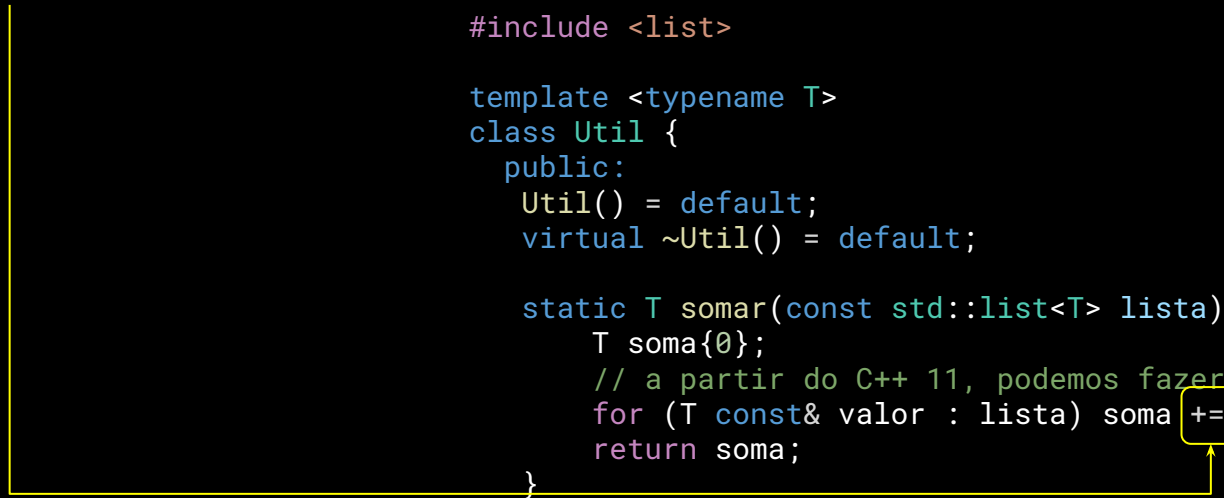
```cpp
#ifndef UTIL_HPP
#define UTIL_HPP

#include <list>

template <typename T>
class Util {
  public:
   Util() = default;
   virtual ~Util() = default;

   static T somar(const std::list<T> lista) {
       T soma{0};
       // a partir do C++ 11, podemos fazer um foreach
       for (T const& valor : lista) soma += valor;
       return soma;
   }
};
#endif
```

Dica: Em C++ isso se chama
Range-Based for

Exemplo

O compilador resolve em tempo de compilação.

Verifica se T é um tipo, e se ele é capaz de realizar as operações necessárias.



```cpp
#ifndef UTIL_HPP
#define UTIL_HPP

#include <list>

template <typename T>
class Util {
  public:
   Util() = default;
   virtual ~Util() = default;

   static T somar(const std::list<T> lista) {
       T soma{0};
       // a partir do C++ 11, podemos fazer um foreach
       for (T const& valor : lista) soma += valor;
       return soma;
   }
};
#endif
```

Pergunta

Para que serve a classe a seguir?

Retorna somatórios de itens de uma lista.

Exemplo: somar todos os os valores de uma lista de inteiros.

Ela está correta?

Sim.

Obs.: serve como exemplo, mas poderia ser mais genérica, por exemplo, aceitando qualquer container iterável (lista, vetor, array, ...).

```
#ifndef UTIL_HPP
#define UTIL_HPP

#include <list>

template <typename T>
class Util {
  public:
   Util() = default;
   virtual ~Util() = default;

   static T somar(const std::list<T> lista) {
       T soma{0};
       // a partir do C++ 11, podemos fazer um foreach
       for (T const& valor : lista) soma += valor;
       return soma;
   }
};
#endif
```

Pergunta

Quando as coisas podem dar errado?

```
#ifndef UTIL_HPP
#define UTIL_HPP

#include <list>

template <typename T>
class Util {
  public:
   Util() = default;
   virtual ~Util() = default;

   static T somar(const std::list<T> lista) {
       T soma{0};
       // a partir do C++ 11, podemos fazer um foreach
       for (T const& valor : lista) soma += valor;
       return soma;
   }
};
#endif
```

Pergunta

```cpp
#include <iostream>
#include <string>

#include "util.hpp"

int main() {
   std::list<std::string> palavras;
   palavras.push_back("abacate");
   palavras.push_back("carro");
   palavras.push_back("computador");

   std::cout << Util<std::string>::somar(palavras);

   std::cout << "\nFim!!!\n";
   return 0;
}
```

```cpp
#ifndef UTIL_HPP
#define UTIL_HPP

#include <list>

template <typename T>
class Util {
  public:
   Util() = default;
   virtual ~Util() = default;

   static T somar(const std::list<T> lista) {
       T soma{0};
       // a partir do C++ 11, podemos fazer um foreach
       for (T const& valor : lista) soma += valor;
       return soma;
   }
};
#endif
```

Console:
abacatecarrocomputador
Fim!!!

Não podemos multiplicar!!!

```
Util<T>::multiplicar(std::__cxx11::list<T>)
[with T = std::__cxx11::basic_string<char>]':
main.cpp:12:37:   required from here
util.hpp:21:43: error: no match for 'operator*='
(operand types are
'std::__cxx11::basic_string<char>' and 'const
std::__cxx11::basic_string<char>')
   21 |         for (T const& valor : lista)
mult *= valor;
```

```cpp
int main() {
   std::list<std::string> palavras;
   palavras.push_back("abacate");
   palavras.push_back("carro");
   palavras.push_back("computador");

   std::cout << Util<std::string>::multiplicar(palavras);

   std::cout << "\nFim!!!\n";
   return 0;
}
```

```cpp
#ifndef UTIL_HPP
#define UTIL_HPP

#include <list>

template <typename T>
class Util {
  public:
   Util() = default;
   virtual ~Util() = default;

   static T somar(const std::list<T> lista) {
       T soma{0};
       for (T const& valor : lista) soma += valor;
       return soma;
   }

   static T multiplicar(const std::list<T> lista) {
       T mult{1};
       for (T const& valor : lista) mult *= valor;
       return mult;
   }
};
#endif
```

Util

Qualquer classe que implemente as funções necessárias será aceita pelo compilador.

Nem sempre desejamos isso.

No exemplo, é um tanto estranho “somar” strings.

Antes do C++20, só podíamos definir os tipos válidos na documentação, ou fazer alguma gambiarra com asserts para mitigar o problema.

```
#ifndef UTIL_HPP
#define UTIL_HPP

#include <list>

template <typename T>
class Util {
  public:
   Util() = default;
   virtual ~Util() = default;

   static T somar(const std::list<T> lista) {
       T soma{0};
       // a partir do C++ 11, podemos fazer um foreach
       for (T const& valor : lista) soma += valor;
       return soma;
   }
};
#endif
```

Concepts

A partir do C++20, temos Concepts.

Concept: requisitos para que o template possa ser instanciado.

Predicados avaliados em **tempo de compilação**.

Antes de continuar

Pode ser necessário adicionar `-std=c++20` como parâmetro de compilação para habilitar os concepts no seu compilador.

Inclua o header `<concepts>`.

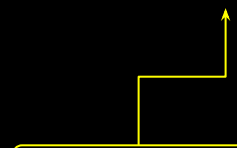
Começando simples

Vamos começar com alguns concepts prontos:

https://en.cppreference.com/w/cpp/concepts

Exemplo

O tipo T deve ser avaliado como um integral em tempo de compilação.



```
template <std::integral T>
class Util {
  public:
   Util() = default;
   virtual ~Util() = default;

   static T somar(const std::list<T> lista) {
       T soma{0};
       for (T const& valor : lista) soma += valor;
       return soma;
   }

   static T multiplicar(const std::list<T> lista) {
       T mult{1};
       for (T const& valor : lista) mult *= valor;
       return mult;
   }
};
#endif
```

Exemplo

```
int main() {
   std::list<int> list1;
   list1.push_back(1);
   list1.push_back(2);

   std::list<double> list2;
   list2.push_back(1.1);
   list2.push_back(2.2);

   std::list<std::string> list3;
   list3.push_back("Laranja");
   list3.push_back("Abacaxi");

   std::cout << Util<int>::somar(list1) << '\n';// ok
   std::cout << Util<double>::somar(list2) << '\n';// erro de compilação
   std::cout << Util<std::string>::somar(list3) << '\n';// erro de compilação

   std::cout << "Fim!!!\n";
   return 0;
}
```

Exemplo

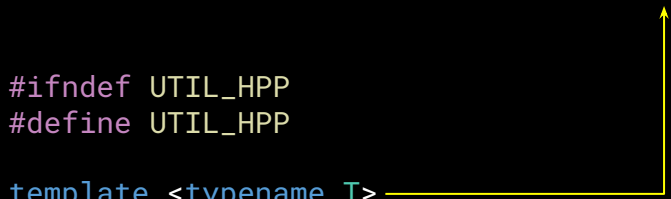
```
int main() {
   std::list<int> list1;
   list1.push_back(1);
   list1.push_back(2);

   std::list<double> list2;
   list2.push_back(1.1);
   list2.push_back(2.2);

   std::list<std::string> list3;
   list3.push_back("Laranja");
   list3.push_back("Abacaxi");

   std::cout << Util<int>::somar(list1) << '\n';// ok
   std::cout << Util<double>::somar(list2) << '\n';// erro de compilação
   std::cout << Util<std::string>::somar(list3) << '\n';// erro de compilação

   std::cout << "Fim!!!\n";
   return 0;
}
```

```
main.cpp:21:29: error: template constraint
failure for ‘template<class T>  requires
integral<T> class Util’
```

Criando seu próprio conceito

Para criar seu próprio conceito, que aceita valores integrais e pontos flutuantes faça, por exemplo:

Definido um novo Concept chamado Numérico, que pode ser usado em qualquer lugar. O Concept é avaliado como verdadeiro se T for um integral ou ponto flutuante.



```
#ifndef UTIL_HPP
#define UTIL_HPP

template <typename T>
concept Numerico = std::integral<T> || std::floating_point<T>;

template <Numerico T>
class Util {
  public:
    // ...
};
#endif
```

Criando seu próprio conceito

```cpp
int main() {
    std::list<int> list1;
    list1.push_back(1);
    list1.push_back(2);

    std::list<double> list2;
    list2.push_back(1.1);
    list2.push_back(2.2);

    std::list<std::string> list3;
    list3.push_back("Laranja");
    list3.push_back("Abacaxi");

    std::cout << Util<int>::somar(list1) << '\n'; // Ok
    std::cout << Util<double>::somar(list2) << '\n'; // Ok
    std::cout << Util<std::string>::somar(list3) << '\n'; // Erro de compilação

    std::cout << "Fim!!!\n";
    return 0;
}
```

Indo Além

O conceito numérico requer dois tipos, T e U, onde se U não for especificado, U = T. São necessários dois objetos, x do tipo T, e y do tipo U.

```
#ifndef UTIL_HPP
#define UTIL_HPP

template <typename T, typename U = T>
concept Numerico = requires(T x, U y) {
    x - y;
    x * y;
    x / y;
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x = x;
    x = 0;
};

template <Numerico T>
class Util {
  public:
    // ...
};
#endif
```

Indo Além

Essas operações precisam ser suportadas, e o conceito de zero precisa existir.

```
#ifndef UTIL_HPP
#define UTIL_HPP

template <typename T, typename U = T>
concept Numerico = requires(T x, U y) {
    x - y;
    x * y;
    x / y;
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x = x;
    x = 0;
};

template <Numerico T>
class Util {
  public:
    // ...
};
#endif
```

Indo Além

```cpp
int main() {
    std::list<int> list1;
    list1.push_back(1);
    list1.push_back(2);

    std::list<double> list2;
    list2.push_back(1.1);
    list2.push_back(2.2);

    std::list<std::string> list3;
    list3.push_back("Laranja");
    list3.push_back("Abacaxi");

    std::cout << Util<int>::somar(list1) << '\n'; // Ok
    std::cout << Util<double>::somar(list2) << '\n'; // Ok
    std::cout << Util<std::string>::somar(list3) << '\n'; // Erro de compilação

    std::cout << "Fim!!!\n";
    return 0;
}
```

Avaliando

Podemos avaliar as expressões via static_assert

```
#include <cassert>
#include <iostream>
#include <string>

#include "util.hpp"

int main() {
   static_assert(Numerico<double>); // Ok
   static_assert(Numerico<int>); // Ok
   static_assert(Numerico<int, double>); // Ok
   static_assert(Numerico<std::string>); // Falha no assert

   std::cout << "Fim!!!\n";
   return 0;
}
```

Outro Exemplo

Aceita qualquer tipo T que possua a função getNome()

```
#ifndef UNIDADE_COORDENACAO
#define UNIDADE_COORDENACAO

#include <iostream>

template <typename T>
class UnidadeCoordenacao {
  public:
   UnidadeCoordenacao() = default;
   virtual ~UnidadeCoordenacao() = default;

   void setCoordenador(const T* coordenador) {
       this->coordenador = coordenador;
   }

   void iniciarSemestre() {
       std::cout << "Iniciando o semestre\n";
       std::cout << "Coordenador de curso " << coordenador->getNome() << '\n';
   }
  private:
   const T* coordenador;
};
#endif
```

Outro Exemplo

```
#include <cassert>
#include <iostream>
#include <string>

#include "Pessoa.hpp"
#include "Professor.hpp"
#include "ProfessorAdjunto.hpp"
#include "UnidadeCoordenacao.hpp"
#include "Util.hpp"

int main() {
   Pessoa p{"Joao"};
   UnidadeCoordenacao<Pessoa> un;
   un.setCoordenador(&p);
   un.iniciarSemestre();

   std::cout << "Fim!!!\n";
   return 0;
}
```

Somente Professores

Agora apenas professores são aceitos.

```cpp
#include <iostream>

#include "Professor.hpp"
#include "ProfessorAdjunto.hpp"
#include "UnidadeCoordenacao.hpp"

int main() {
   ProfessorAdjunto p{"Joao", 11111111111, 60, 40};
   UnidadeCoordenacao<Professor> un;
   un.setCoordenador(&p);
   un.iniciarSemestre();

   std::cout << "Fim!!!\n";
   return 0;
}
```

```cpp
#ifndef UNIDADE_COORDENACAO
#define UNIDADE_COORDENACAO

#include <concepts>
#include <iostream>

#include "Professor.hpp"

template <typename T>
concept ConceptProfessor = std::derived_from<T, Professor>;

template <ConceptProfessor T>
class UnidadeCoordenacao {
  public:
   //...

  private:
   const T* coordenador;
};

#endif
```

Em Java

Em Java temos uma versão (muito) mais simples de se usar, mas muito mais limitada, através do uso dos Bounded Type Parameters.

Veja mais em:
docs.oracle.com/javase/tutorial/java/generics/bounded.html

Um conceito similar ao do Java é implementado em C#.

```java
public class NaturalNumber<T extends Integer> {

    private T n;

    public NaturalNumber(T n)  { this.n = n; }

    public boolean isEven() {
        return n.intValue() % 2 == 0;
    }

    // ...
}
```

Mais

Esse foi o básico de orientação a objetos em C++.

Existem vários outros conceitos interessantes, fica para uma próxima:

- Boost.
- Expressões Lambda.
- Concorrência.
- Módulos (Promessa de resolver o caos dos arquivos .hpp, .cpp, e includes).
- ...

Exemplo - Boost C++

A Boost C++ oferece uma enorme quantidade de bibliotecas extras para o C++.

Extensão padrão e oficial do C++.

Itens que vão para o C++ ISO são primeiramente implementados na Boost.

Smart Pointers, loops foreach, classes de data/hora, ...

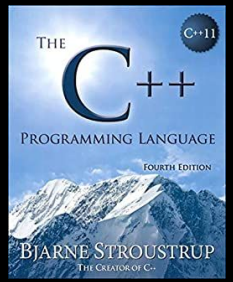
www.boost.org

```
boost::gregorian::date currentDay;
std::map<boost::posix_time::ptime,mtopk::ImageData*>::const_iterator it{imgData->begin()};
```

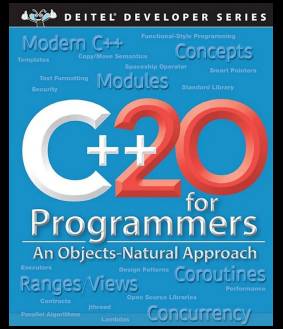

The problem of being faster than light is that you can only live in darkness

Referências

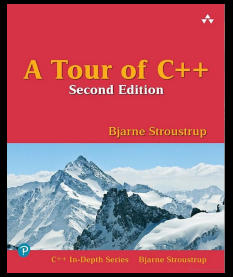
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



Deitel, P., Deitel, H. C++20 for Programmers: An Objects-Natural Approach. 2022.



Stroustrup, B. . A Tour of C++. 2a Ed. 2022.



www.boost.org

Licença