The cake is a lie!

Namespaces

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Conflitos

- Podemos ter conflitos nos mais diversos níveis
  - Funções “soltas” com mesmo nome em diferentes headers;
  - Constantes com mesmo nome;
  - Classes com mesmo nome em diferentes APIs;
  - ...
- Criamos a classe Pessoa.
  - E se importarmos um conjunto de bibliotecas que já define uma classe com esse nome?

Namespaces

Esses são exemplos de **conflitos de nome**.

Namespaces podem ajudar a resolver esses conflitos.

Um **namespace define um escopo dentro do qual podemos ter classes, variáveis, funções, ...**

Namespaces

Para declarar um namespace, basta usar

```
namespace nomeNamespace{

//itens que compõem o namespace

}
```

Convenção

Vamos seguir a convenção do Google, com nomes de namespaces em minúsculo e separados por underscore quando necessário.

https://google.github.io/styleguide/cppguide.html#Namespace_Names

Exemplo

A classe Disciplina agora pertence ao *namespace* **ufpr**

Disciplina.hpp

```
namespace ufpr{
class SalaAula;//forward declaration
class Disciplina {
  public:
    //...
  private:
   std::string nome;
   unsigned short int cargaHoraria;
   Professor *professor;
   SalaAula *sala;

   std::list<ConteudoMinistrado *> conteudos;
   std::list<Pessoa *> alunos;
};
}  // namespace ufpr
#endif
```

Disciplina.cpp

```
#include "Disciplina.hpp"

#include <iostream>

#include "SalaAula.hpp"

using namespace ufpr;

Disciplina::Disciplina(const std::string& nome) : nome{nome},
sala{nullptr} {}

Disciplina::Disciplina(const std::string& nome, SalaAula* const sala)
   : Disciplina{nome} {
   this->setSalaAula(sala);
}

//...
```

Outra Forma

A classe Disciplina agora pertence ao *namespace* **ufpr**

Disciplina.hpp

```
namespace ufpr{
class SalaAula;//forward declaration
class Disciplina {
  public:
    //...
  private:
   std::string nome;
   unsigned short int cargaHoraria;
   Professor *professor;
   SalaAula *sala;

   std::list<ConteudoMinistrado *> conteudos;
   std::list<Pessoa *> alunos;
};
}  // namespace ufpr
#endif
```

Disciplina.cpp

```
#include "Disciplina.hpp"

#include <iostream>

#include "SalaAula.hpp"

namespace ufpr{

Disciplina::Disciplina(const std::string& nome) : nome{nome},
sala{nullptr} {}

Disciplina::Disciplina(const std::string& nome, SalaAula* const sala)
   : Disciplina{nome} {
   this->setSalaAula(sala);
}

//...
}
```

Exemplo

SalaAula.hpp

```
#ifndef SALA_AULA_H
#define SALA_AULA_H

#include <string>
#include <list>

#include "Disciplina.hpp"

namespace ufpr{
class SalaAula{
   friend void Disciplina::setSalaAula(SalaAula* salaAula);

   public:
       SalaAula(std::string nome, unsigned int capacidade);
       virtual ~SalaAula();
       std::string getNome();
       void setNome(std::string nome);
       unsigned int getCapacidade();
       void setCapcidade(unsigned int capacidade);
       std::list<Disciplina*>& getDisciplinas();
   private:
       std::string nome;
       unsigned int capacidade;
       std::list<Disciplina*> disciplinasMinistradas;
};
}
#endif
```

SalaAula.cpp

```
#include "SalaAula.hpp"

using namespace ufpr;

SalaAula::SalaAula(std::string nome, unsigned int
capacidade)
   :nome{nome}, capacidade{capacidade}{
}

SalaAula::~SalaAula(){
   std::list<Disciplina*>::iterator
            it{disciplinasMinistradas.begin()};
   for( ; it != disciplinasMinistradas.end(); it++)
       (*it)->anularSalaAula();
}

std::string SalaAula::getNome(){
   return this->nome;
}

//...
```

namespaces

Agora tanto `SalaAula` quanto disciplina pertencem ao `namespace ufpr`.

Ao usar essas classes, é necessário indicar agora que estamos nos referenciando a esse espaço.

Uma forma é usar o operador de resolução de escopo `::`

```
#include <iostream>

#include "SalaAula.hpp"

int main() {
   ufpr::SalaAula sala{"Lab info 1", 30};

   std::cout << sala.getNome() <<'\n';

   return 0;
}
```

Outras formas

É possível **importar um item específico de um espaço de nome.**

```
using nomeEspaço::item;
```

Exemplo:

```
#include<iostream>

#include"SalaAula.hpp"

using ufpr::SalaAula;

int main(){
   SalaAula sala{"Lab info 1", 30};

   std::cout << sala.getNome() << '\n';

   return 0;
}
```

Importando todos os itens

Finalmente, você pode **importar todos os itens de um espaço de nome.**

```
using namespace nomeEspaço;
```

Exemplo:

```
#include<iostream>

#include"SalaAula.hpp"

using namespace ufpr;

int main(){
   SalaAula sala{"Lab info 1", 30};

   std::cout << sala.getNome() << '\n';

   return 0;
}
```

Boas práticas

Evite importar namespaces (using ...).

Nunca importe namepsaces (using ...) **em headers.**

Os headers serão importados por outros arquivos, e automaticamente você vai importar os using!

Importar espaços de nome indiscriminadamente é o mesmo que não tê-los!

Pode gerar conflitos de nome difíceis de detectar e corrigir.

Using namespace bla bla bla

"... tome cuidado com diretivas using globais já que seu uso abusivo pode levar exatamente as mesmas colisões que os namespaces foram criados para evitar ... Em particular, não insira um using no escopo global em um header exceto em circunstâncias muito especializadas, já que você não sabe onde um header pode ser incluído" (Stroustrup, 2013).

"*... care should be taken with global using-directives because overuse can lead to exactly the name clashes that namespaces were introduced to avoid ... In particular, don't place a using-directive in the global scope in a header file except in very specialized circumstances because you never know where a header might be included*" (Stroustrup, 2013).

Using namespace bla bla bla

"... care should be taken with global using-directives because overuse can lead to exactly the name clashes that namespaces were introduced to avoid ... In particular, don't place a using-directive in the global scope in a header file except in very specialized circumstances because you never know where a header might be included" (Stroustrup, 2013).

Para facilitar, usamos `using` nos cpps das classes.

Essa pode ser uma circunstância especializada.

É um cpp e não um header.

Estamos importando nosso próprio namespace, o qual temos controle.

O cpp é autocontido e se refere somente às funções de uma classe específica.

O cpp não é importado diretamente para outros arquivos.

Using namespace bla bla bla

"... care should be taken with global using-directives because overuse can lead to exactly the name clashes that namespaces were introduced to avoid ... In particular, don't place a using-directive in the global scope in a header file except in very specialized circumstances because you never know where a header might be included" (Stroustrup, 2013).

Para facilitar, usamos `using` nos cpps das classes.

Essa pode ser uma circunstância especializada.

É um cpp e não um header.

Estamos importando nosso próprio namespace, o qual temos controle.

O cpp é autocontido e se refere somente às funções de uma classe específica.

O cpp não é importado diretamente para outros arquivos.

Obs.: por um raciocínio semelhante, é aceitável fazer `using namespace ...` no `main`, como é feito em Deitel e Deitel 2017. Mas não vamos fazer para não adquirir maus hábitos e acabar usando onde não devemos.

Google

A convenção do Google abomina os using namespaces mesmo dentro de cpps que definem as funções de classes:

https://google.github.io/styleguide/cppguide.html#Namespaces

Aninhando

Namespaces podem ainda ser aninhados

Exemplo:

```
namespace meu_name{
    //...
    namespace aninhado{
        //...
    }
}
```

Comparando

C# implementa mecanismos de namespace baseados na mesma ideia do C++.

Java possui o conceito de pacotes para tratar o mesmo problema.

Namespaces versus pacotes

Os **pacotes em Java são uma versão mais simples e intuitiva** do que os namespaces do C++ (opinião).

Os pacotes do Java são organizados em subdiretórios.

Cada pacote tem seu próprio diretório no Projeto.

Por que C++ não usa a mesma abordagem? Qual o problema?

Namespaces versus pacotes

Os **pacotes em Java são uma versão mais simples e intuitiva** do que os namespaces do C++ (opinião).

Os pacotes do Java são organizados em subdiretórios.

Cada pacote tem seu próprio diretório no Projeto.

Por que C++ não usa a mesma abordagem? Qual o problema?

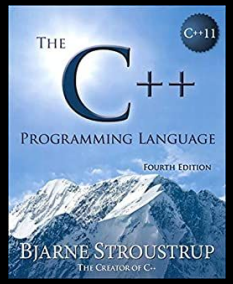
Um sistema operacional com o conceito de árvore de diretórios se torna um requisito para compilar.

Exercícios

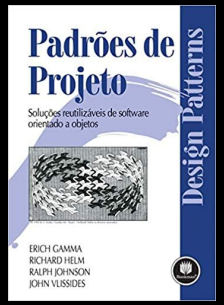
1. Transfira o projeto completo para o namespace ufpr (todas as classes criadas até agora devem fazer parte desse namespace). Instancie alguns objetos de exemplo no main.

Referências

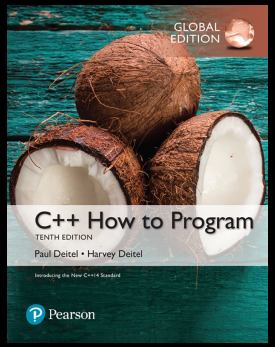
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



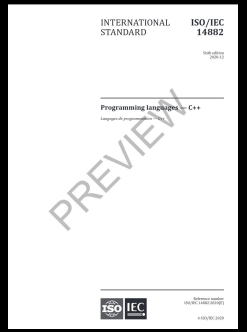
Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.



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