"O desenvolvimento de classes é uma atividade interessante, criativa e intelectualmente desafiadora, com o objetivo de criar classes de valor" (Deitel e Deitel, 2017).

Tipos de Casting

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR

Antes de começar

Note no projeto que foram declarados construtores explícitos para `ProfessorAdjunto`.

Caso contrário o projeto não compilaria.

O construtor de Professor seria implicitamente removido.

Muitas de nossas classes não tem, por exemplo, construtor de cópia ou atribuição na hierarquia.

Isso tudo deveria existir (faça você mesmo).

Castings

Diferente de C, em C++ é raro precisarmos de castings.

Pense duas vezes antes de fazer um cast.

“geralmente um cast é indicativo de um erro de design.” (Stroustrup, 2013).

Casting estilo C

Resumo: **evite castings no estilo C.**

Castings na forma (tipo)variavel podem fazer qualquer tipo de casting, sem que isso fique claro para o programador.

Pode invocar um `static_cast`, `reinterpret_cast`, `const_cast`, ou uma combinação deles!

Casting estilo C

Resumo: **evite castings no estilo C.**

Castings na forma (tipo)variavel podem fazer qualquer tipo de casting, sem que isso fique claro para o programador.

Pode invocar um `static_cast`, `reinterpret_cast`, `const_cast`, ou uma combinação deles!

“Infelizmente, castings estilo C podem converter de um ponteiro de uma classe para outra que seja uma base privada. Nunca faça isso, e torça para um aviso do compilador caso o faça por engano. Castings estilo C são muito mais perigosos do que conversões nomeadas (veremos na aula) já que a notação torna mais complicado entender qual tipo de conversão está sendo feita ... “ (Stroustrup, 2013).

static_cast

- Um `static_cast<tipo>()` faz uma conversão entre tipos relacionados. Exemplos:
  - Conversão de um ponteiro para outro da mesma hierarquia de classes;
  - Conversão de um tipo primitivo para outro (ex.: float para int).
  - Veja mais em 11.5.2 de Stroustup (2013).
- O compilador resolve a conversão em tempo de compilação.
  - **Nenhuma checagem em tempo de execução é realizada.**

Exemplo

```
unsigned short int Pessoa::getIdade() const {
   return static_cast<unsigned short int>(idade);
}

void Pessoa::setIdade(const unsigned short int novaIdade) {
   if (novaIdade > 120)
      throw std::invalid_argument{"Idade Invalida."};
   idade = static_cast<unsigned char>(novaIdade);
}
```

Pergunta

O que há de estranho com o trecho a seguir?

```cpp
#include <iostream>

#include "CPF.hpp"

void imprimirCpf(const ufpr::CPF& cpf){
   std::cout << "O CPF eh: " << cpf << '\n';
}

int main(){
   ufpr::CPF cpf{11111111111};

   imprimirCpf(cpf);
   imprimirCpf(22222222222);

  return 0;
}
```

Pergunta

O que há de estranho com o trecho a seguir?

```cpp
#include <iostream>

#include "CPF.hpp"

void imprimirCpf(const ufpr::CPF& cpf){
    std::cout << "O CPF eh: " << cpf << '\n';
}

int main(){
    ufpr::CPF cpf{11111111111};

    imprimirCpf(cpf);
    imprimirCpf(22222222222);

  return 0;
}
```

A função aceita apenas cpfs, mas estamos passando um unsigned long.
E o compilador aceitou!

Conversão Implícita

O compilador encontrou uma forma de construir um `CPF` a partir de um `long int`, e o fez.

Essa conversão é automática, e chamada de **conversão implícita**.

```
class CPF{
   public:
      CPF(const unsigned long numero);
      //...
};
```

```
#include <iostream>

#include "CPF.hpp"

void imprimirCpf(const ufpr::CPF& cpf){
   std::cout << "O CPF eh: " << cpf << '\n';
}

int main(){
   ufpr::CPF cpf{11111111111};

   imprimirCpf(cpf);
   imprimirCpf(22222222222);

  return 0;
}
```

Conversão Implícita

Conversões implícitas podem muitas vezes gerar confusão ou até erros de lógica.

Para evitar conversões implícitas, marque os construtores da classe como `explicit` nos headers.

Exemplo

Agora a conversão implícita não é feita.

Mas você pode forçar o uso do construtor explicitamente, usando um `static_cast`.

Mas seria particularmente estranho. Se você quer um objeto CPF, construa um!!!

```
class CPF{
   public:
       explicit CPF(const unsigned long numero);
       //...
};
```

Dica

A conversão implícita se aplica apenas a construtores com um parâmetro.

Declare todos os construtores com um parâmetro como `explicit` para evitar erros de lógica.

Exceto se você realmente desejar que o construtor seja utilizado para conversões implícitas.

Sobrecarga de operadores de casting

É possível sobrecarregar operadores de conversão para as classes.

Converter o objeto para outros tipos.

Protótipo:

```
MyClass::operator OtherClass() const;
```

Exemplo

```cpp
int main() {
   ufpr::CPF cpf{11111111111};
   unsigned long val{cpf};

   std::cout << val << '\n';

   return 0;
}
```

```cpp
#ifndef CPF_HPP
#define CPF_HPP

#include <ostream>
#include <istream>

namespace ufpr{
class CPF{
   public:
       operator unsigned long() const;
       //...
};
}
#endif
```

```cpp
CPF::operator unsigned long() const{
   return this->numero;
}
```

Operadores de casting explícitos

Operadores de casting podem ser declarados com o explícitos.

Preciso usar `static_cast` explicitamente para fazer a conversão.

Exemplo

```cpp
#include <iostream>

#include "CPF.hpp"

void imprimir(unsigned long valor){
   std::cout << "O valor eh: " << valor << '\n';
}

void imprimir(const ufpr::CPF& cpf) {
   std::cout << "O CPF eh: " << cpf << '\n';
}

int main() {
   ufpr::CPF cpf{11111111111};

   imprimir(cpf);
   imprimir(static_cast<unsigned long>(cpf));

   return 0;
}
```

```cpp
#ifndef CPF_HPP
#define CPF_HPP

#include <ostream>
#include <istream>

namespace ufpr{
class CPF{
   public:
       explicit operator unsigned long() const;
       //...
};
}
#endif
```

reinterpret_cast - The nastiest of them all

Um `reinterpret_cast` é usado para converter, por exemplo, entre ponteiros não relacionados.

Converter um ponteiro de um tipo para qualquer outro tipo.

`Reinterpret` feito em tempo de compilação.

Exemplo

```cpp
#include <iostream>
int main()
{
   int* p = new int(65);
   char* ch = reinterpret_cast<char*>(p);
   std::cout << *p << '\n';
   std::cout << *ch << '\n';
   std::cout << p << '\n';
   std::cout << reinterpret_cast<void*>(ch) << '\n';
   return 0;
}
```

reinterpret_cast

`reinterpret_cast`s são **extremamente perigosos!**

Seu uso é raro.

É usado especialmente quando blocos de memória são alocados, para que dados sejam escritos Byte a Byte.

Conceito de serialização.

Veja um exemplo na Seção 14.8 de Deitel e Deitel (2017).

`reinterpret_cast`s podem ser necessários quando estamos lidando com ponteiros de função.

Quando estamos compatibilizando funções legadas de C para C++.

Em qualquer outro cenário, usar ponteiros para funções em C++ provavelmente está errado.

const_cast

Um `const_cast` é usado para se obter permissão de escrita em uma variável const.

```
#include <iostream>

void imprimir(int* ptr){
   std::cout << "Imprimindo de uma funcao mal escrita" <<
       " que so aceita ponteiros nao const " << *ptr << '\n';
}

int main(void) {
   const int val{100};
   const int *constPtr{&val};
   int *ptr{const_cast <int *>(constPtr)};

   //imprimir(constPtr);//erro de compilação
   imprimir(ptr);


   return 0;
}
```

const_cast

`const_cast` é comumente usado para adequar chamadas de função em códigos legados.

Mais uma vez, pode **ser perigoso.**

> **O uso de `const_casts` em um programa é um péssimo sinal.**

Use somente como estritamente necessário.

dynamic_cast

Um `dynamic_cast` é usado para identificar o tipo de um objeto em tempo de execução.

Quando usado em um ponteiro, faz o cast se possível, ou insere `nullptr` no ponteiro caso contrário.

Quando usado em uma referência, faz o cast se possível, ou então lança uma exceção `bad_cast` caso contrário.

dynamic_cast

O `dynamic_cast` pode ser usado para “subir o nível de um objeto na hierarquia de classes” de maneira segura.

Exemplo: Todo professor é uma pessoa, mas nem toda pessoa é um professor!

Ao receber uma pessoa como argumento, como testar de forma segura se ela é um professor?

```cpp
void funcao(ufpr::Pessoa* p){
   ufpr::ProfessorAdjunto* pa{dynamic_cast<ufpr::ProfessorAdjunto*>(p)};
   if(pa != nullptr){
       std::cout << pa->getNome() << '\n';
       std::cout << "Professor adjunto. Posso chamar funcoes especificas a partir de pa agora\n";
       return;
   }
   ufpr::Professor* pr{dynamic_cast<ufpr::Professor*>(p)};
   if(pr != nullptr){
       std::cout << pr->getNome() << '\n';
       std::cout << "Professor. Posso chamar funcoes especificas a partir de pr agora\n";
       return;
   }
   std::cout << p->getNome() << '\n';
   std::cout << "Classe base\n";
   return;
}

int main(void) {
   ufpr::Professor prof{"Joao", 11111111111, 100, 40};
   ufpr::ProfessorAdjunto profAd{"Maria", 22222222222, 120, 40};
   ufpr::Pessoa pessoa{"Pedro", 33333333333};

   funcao(&prof);
   funcao(&profAd);
   funcao(&pessoa);

   return 0;
}
```

```
void funcao(ufpr::Pessoa* p){
   ufpr::ProfessorAdjunto* pa{dynamic_cast<ufpr::ProfessorAdjunto*>(p)};
   if(pa != nullptr){
       std::cout << pa->getNome() << '\n';
       std::cout << "Professor adjunto. Posso chamar funcoes especificas a partir de pa agora\n";
       return;
   }
   ufpr::Professor* pr{dynamic_cast<ufpr::Professor*>(p)};
   if(pr != nullptr){
       std::cout << pr->getNome() << '\n';
       std::cout << "Professor. Posso chamar funcoes especificas a partir de pr agora\n";
       return;
   }
   std::cout << p->getNome() << '\n';
   std::cout << "Classe base\n";
   return;
}

int main(void) {
   ufpr::Professor prof{"Joao", 11111111111, 100, 40};
   ufpr::ProfessorAdjunto profAd{"Maria", 22222222222, 120, 40};
   ufpr::Pessoa pessoa{"Pedro", 33333333333};

   funcao(&prof);
   funcao(&profAd);
   funcao(&pessoa);

   return 0;
}
```

Essa gambiarra é só um exemplo!

Uma maneira melhor de resolver nesse caso seria fazer várias sobrecargas da função, onde uma sobrecarga aceita ProfessorAdjunto, outra aceita Professor, ...

dynamic_cast

Mais uma vez, use `dynamic_cast` com cautela.

Uso extensivo de `dynamic_casts` é um forte indicativo de problemas de design.

Geralmente o problema pode ser resolvido usando inversão de controle e funções virtuais, sem nunca envolver `dynamic_casts`.

dynamic_cast

`Dynamic_cast` é feito usando:

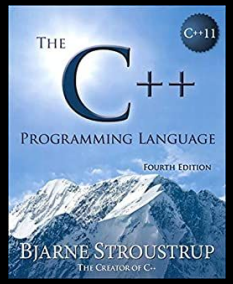
- instanceof em Java;
- objeto.IsInstanceOfType(Tipo) em .Net;

Exercícios

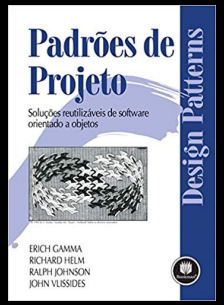
1. Declare todos construtores que recebem um parâmetro como `explicit`, exceto construtores que você julgar fazerem sentido como conversores implícitos.
2. Escreva um operador de casting para CPF, que retorna o CPF em uma string.

Referências

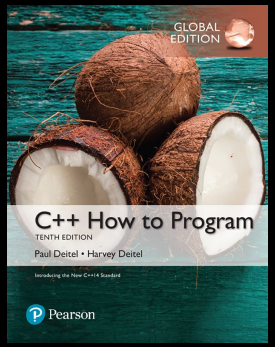
Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 2013.



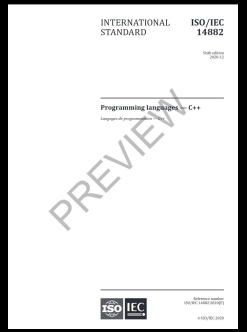
Gamma, E. Padrões de Projetos: Soluções Reutilizáveis. Bookman. 2009.



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