# Classes e funções amigas

Paulo Ricardo Lisboa de Almeida

UDESC
UNIVERSIDADE
DO ESTADO DE
SANTA CATARINA

# Atualizando os objetos associados

- Na aula passada foi solicitado que
  - Ao modificar a sala de aula de uma *Disciplina*, a *SalaAula* fosse informada disso automaticamente
  - Ao adicionar uma *Disciplina* em uma *SalaAula*, a disciplina fosse informada disso automaticamente

# Possível solução

SalaAula.cpp

```
void SalaAula::adicionarDisciplina(Disciplina* disciplina){
        disciplinasMinistradas.push_back(disciplina);
        if(disciplina->getSalaAula() != this)
                disciplina->setSalaAula(this);
}
```

Disciplina.cpp

```
void Disciplina::setSalaAula(SalaAula* salaAula){
        std::list<Disciplina*> aux = salaAula->getDisciplinas();
        std::list<Disciplina*>::iterator it;
        // verifica se a sala de aula já contem a disciplina para nao entrar em loop
        for(it = aux.begin(); it != aux.end(); it++){
                if ((*it) == this) {
                        break;
                }
        }
        // adiciona apenas se não continha a disciplina
        if (it == aux.end()) salaAula->adicionarDisciplina(this);
        // verifica se a sala é diferente e não nula, pra nao apagar algo que nao
        // existe nem apagar a disciplina da sala certa na lista de disciplinas em SalaAula
        if(this->salaAula != nullptr && this->salaAula != salaAula)
                this->salaAula->removerDisciplina(this);
        this->salaAula = salaAula;
}
```

Exemplo criado por Lais Vossen

UDESC

# Possível solução

SalaAula.cpp

```
void SalaAula::adicionarDisciplina(Disciplina* disciplina){
        disciplinasMinistradas.push_back(disciplina);
        if(disciplina->getSalaAula() != this)
                disciplina->setSalaAula(this);
}
```

Disciplina.cpp

```
void Disciplina::setSalaAula(SalaAula* salaAula){
        std::list<Disciplina*> aux = salaAula->getDisciplinas();
        std::list<Disciplina*>::iterator it;
        // verifica se a sala de aula já contem a disciplina para nao entrar em loop
        for(it = aux.begin(); it != aux.end(); it++){
                if ((*it) == this) {
                        break;
                }
        }
        // adiciona apenas se não continha a disciplina
        if (it == aux.end()) salaAula->adicionarDisciplina(this);
        // verifica se a sala é diferente e não nula, pra nao apagar algo que nao
        // existe nem apagar a disciplina da sala certa na lista de disciplinas em SalaAula
        if(this->salaAula != nullptr && this->salaAula != salaAula)
                this->salaAula->removerDisciplina(this);
        this->salaAula = salaAula;
}
```

A busca poderia ser mais eficiente em uma estrutura como um *set* (comumente implementados como árvores binárias) ou unordered_set (comumente implementados como tabelas hash)

UDESC

# Opções

- No momento então conhecemos duas opções
  - Os sets e afins não comunicam as alterações para os objetos associados (ou agregados)
    - Nesse caso, que usará as classes terá a responsabilidade de manter tudo consistente
  - Os sets e afins comunicam os objetos associados (ou agregados) as suas alterações
- **Vantagens? Desvantagens?**

# Opções

- No momento então conhecemos duas opções
  - Os sets e afins não comunicam as alterações para os objetos associados (ou agregados)
    - Nesse caso, que usará as classes terá a responsabilidade de manter tudo consistente
      + Baixo overhead
      - O programador agora deve conhecer mais detalhes, e cuidar deles
        - Pode facilmente deixar o sistema inconsistente
  - Os sets e afins comunicam os objetos associados (ou agregados) as suas alterações
      + Transparência
        - As classes gerenciam suas relações, diminuindo a responsabilidade do programador
      - Overhead
        - As lógicas envolvidas podem ser complexas e custar caro para a máquina, principalmente quando envolvem listas como no exemplo anterior
          - Precisamos iterar a lista toda para descobrir se o item já estava lá!

# Terceira opção

- As duas opções anteriores **são válidas**
  - Se são aplicáveis ou não, depende do problema que você precisa resolver
- Estudaremos uma terceira opção, onde o relacionamento ainda é bidirecional, mas as alterações só podem ser feitas a partir de um lado do relacionamento

# Classes e funções amigas

- Uma função ou classe amiga **pode acessar todos os membros** da classe
  - Membros privados, protegidos ou públicos

Veremos no futuro

# Classes e funções amigas

- Dentro de uma classe, podemos declarar que
  - Existe outra **classe** que é amiga
    - ***friend*** *class NomeClasse;*
    - Nesse caso, todas as funções membro de NomeClasse podem acessar **todos** os membros de nossa classe
  - Existe uma **função global** que é amiga
    - ***friend*** *tipoRetorno nomeFuncao(parametros);*
    - Essa função (que não pertence a nenhuma classe específica – função solta como as criadas em C) pode acessar **todos** os membros da classe
  - Existe uma **função membro de determinada classe** que é amiga
    - ***friend*** *tipoRetorno NomeClasseAmiga::nomeFuncaoClasse(parametros);*
    - Somente a função nomeFuncaoClasse da classe NomeClasseAmiga pode acessar **todos** os membros da classe

# Boas práticas

- Boa prática de programação
  - Os amigos da classe devem ser os primeiros itens a serem declarados na classe

# Exemplo

- Vamos usar o conceito de amizade para tornar as classes Disciplina e SalaAula mais autossuficientes
- Primeiro, modifique as funções membro *adicionarDisciplina* e *removerDisciplina* para que sejam **private**
  - Agora o programador já não pode mais invocar essas funções
  - Remova a chamada dessa funções do *main* se necessário

# Exemplo

- Vamos definir que a função *setSalaAula* da classe *Disciplina* pode acessar os membros privados (e públicos e protegidos) da classe *SalaAula*

# Exemplo

```
//...
class SalaAula{
        friend void Disciplina::setSalaAula(SalaAula* salaAula);
    public:
        SalaAula(std::string nome, unsigned int capacidade);

        std::string getNome();
        void setNome(std::string nome);

        unsigned int getCapacidade();
        void setCapcidade(unsigned int capacidade);

        std::list<Disciplina*>& getDisciplinas();
    private:
        void adicionarDisciplina(Disciplina* disciplina);
        void removerDisciplina(Disciplina* disciplina);

        std::string nome;
        unsigned int capacidade;
        std::list<Disciplina*> disciplinasMinistradas;
};
```

UDESC

# Exemplo

- Agora podemos modificar a função membro *setSalaAula* em Disciplina para que ela automaticamente atualize os objetos da classe *SalaAula*
  - Essa função ainda pode acessar os membros *removerDisciplina* e *adicionarDisciplina*, por ser uma função amiga

```
void Disciplina::setSalaAula(SalaAula* salaAula){
        if(this->salaAula != nullptr)//se já existia uma sala, remover a disciplina dessa sala
                this->salaAula->removerDisciplina(this);
        this->salaAula = salaAula;
        if(this->salaAula != nullptr)
                this->salaAula->adicionarDisciplina(this);//adicionar a disciplina na nova sala
}
```

# Exemplo

- Agora o programador não pode acessar as funções para adicionar ou remover disciplinas na sala de aula
- Mas ele ainda pode *setar* a sala de aula das disciplinas
  - As disciplinas se encarregam de atualizar a sala de aula automaticamente
  - Tudo isso sem a utilização de lógicas sofisticadas
    - Baixo *overhead*

# Teste você mesmo

```
//...
int main(){
        Pessoa prof1{"João", 40};
        Disciplina dis1{"C++"};
        dis1.setProfessor(&prof1);


        Pessoa prof2{"Maria", 30};
        Disciplina dis2{"Java"};
        dis2.setProfessor(&prof2);


        SalaAula sala{"F204", 20};
        dis1.setSalaAula(&sala);
        dis2.setSalaAula(&sala);
        //sala.adicionarDisciplina(dis1);//o programador não tem direito a invocar essa função (que agora é private)


        std::cout << sala.getNome() << std::endl;
        std::list<Disciplina*>& disciplinas{sala.getDisciplinas()};
        std::list<Disciplina*>::iterator it;
        for(it=disciplinas.begin(); it != disciplinas.end(); it++)
                std::cout << (*it)->getNome() << std::endl;


        return 0;
}
```

UDESC

# Propriedades

- Para que a classe (ou função) B seja amiga de A, **a classe A precisa explicitamente declarar que B é sua amiga**
- A amizade **não é simétrica**
  - Se B é amiga de A, não significa que A é amiga de B
- Amizade **não é transitiva**
  - Se A é amiga de B, e B é amiga de C, não podemos inferir que A é amiga de C
- Amizade **não é herdada**
  - “Os filhos dos seus amigos não são seus amigos”
    - Veremos herança adiante

# Classes e funções amigas

- Se você acha que a amizade é uma coisa linda que deve ser cultivada, reveja seus conceitos
  - **Pelo menos em C++**
  - Em C++ a amizade é um conceito perigoso que deve ser usado com cautela

# Quando usar

- Sempre pense muito bem antes de usar o conceito de amizade
  - Muitas vezes podemos resolver o problema **sem usar classes amigas**
- Quais problemas as classes amigas podem trazer?

# Quando usar

- Sempre pense muito bem antes de usar o conceito de amizade
  - Muitas vezes podemos resolver o problema **sem usar classes amigas**
- Quais problemas as classes amigas podem trazer?
  - Quebra de encapsulamento
    - Para que definimos que existem dados e funções privadas, se começamos a popular o programa com outras classes que “ignoram” as regras de acesso?
    - Pode tornar as coisas mais difíceis de se dar manutenção

# Quando usar

- C++ assume que os programadores são alfabetizados e bem alimentados!
  - São capazes de tomar decisões sobre quando e como usar determinados recursos
  - Veja o comentário de Bjarne Stroustrup sobre a possível quebra de encapsulamento gerada pelas classes amigas
    - www.stroustrup.com/bs_faq2.html#friend
- **Com grandes poderes vêm grandes responsabilidades!**

# Friendship em outras linguagens

- A única linguagem O.O. que conheço que implementa o conceito de classes amigas é o C++
  - Outras linguagens preferem não implementar, principalmente para evitar quebras de encapsulamento que podem ser perigosas
    - Assumem que o programador vai fazer besteira com seus “super poderes”

# Friendship em outras linguagens

- A falta do conceito de amizade em outras linguagens fazem com que elas precisem implementar outros mecanismos, já que em alguns (raros) momentos, precisamos dessa “quebra de encapsulamento”
- Exemplo: como funciona a visibilidade *protected* no Java?

# Friendship em outras linguagens

- A falta do conceito de amizade em outras linguagens fazem com que elas precisem implementar outros mecanismos, já que em alguns (raros) momentos, precisamos dessa “quebra de encapsulamento”
- Em Java por exemplo, a visibilidade *protected* se estende para todas as classes do mesmo pacote, e não apenas para as suas classes filhas
  - **Opinião:** essa é uma péssima ideia do Java, já que
    - A quebra de encapsulamento se torna compulsória para todos os membros protegidos da classe
    - Muitos programadores não se dão conta que as classes do mesmo pacote podem acessar os itens protegidos da sua classe
      - Podem erroneamente assumir que somente as classes filhas acessam

# Templates e sobrecarga de operadores

- O conceito de amizade pode se tornar ainda mais poderoso (e perigoso) quando utilizado em combinação com *Templates* e sobrecarga de operadores
  - Veremos adiante na disciplina

# Referências


- 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.
- https://www.learncpp.com/cpp-tutorial/10-4-association/
- https://en.cppreference.com/w/cpp/language/friend