

# Programação e Desenvolvimento de Software 2

## Gerenciamento de Memória

---

Prof. Julio Cesar S. Reis  
[julio.reis@dcc.ufmg.br](mailto:julio.reis@dcc.ufmg.br)

# Gerenciamento de Memória

- O que provavelmente irá acontecer se tentarmos executar o código abaixo?

```
double vetor[21000000000];  
    //...  
cout << "Fim!" << endl;
```

# Gerenciamento de Memória

- O que irá acontecer se tentarmos executar o código abaixo?

```
double vetor[21000000000];  
//...  
cout << "Fim!" << endl;
```

- Provavelmente “Fim” não irá aparecer
  - Estouro da pilha

# Gerenciamento de Memória

- Já estamos bastante habituados a resolver este tipo de situação... como?

# Gerenciamento de Memória

- Já estamos bastante habituados a resolver este tipo de situação... como?

```
double *vetor = new double[21000000000];  
  
//...  
  
delete vetor;  
  
cout << "Fim!" << endl;
```

- Uso de new, delete
- Se existir a quantidade necessária de memória, vai funcionar!

# Qual o problema?

- O desenvolvedor C++
  - Quando ele aloca uma memória dinamicamente, por exemplo, e esquece de desalocar essa memória.
- Memory leaks
- E se pudéssemos, por exemplo, fazer new sem a necessidade de delete?
  - RAI, Smart Pointers

# RAII

---

# Aquisição de Recurso É Inicialização (RAII)

- Padrão de projeto de software para C++
- Combina a aquisição e liberação de recursos com inicialização e destruição de objetos
- Uso de memória:
  - inicia-se na declaração
  - termina quando o objeto sai do escopo (fim da execução ou lançamento de exceção)



# Aquisição de Recurso É Inicialização (RAII)

- Fundamentos:

- O recurso é liberado no destruidor (por exemplo, fechando um arquivo)

- Instâncias da classe são alocadas em pilha (e não no heap)

- O recurso é adquirido no construtor (por exemplo, abrir um arquivo). Esta parte é opcional, mas comum.

# Aquisição de Recurso É Inicialização (RAI)

- "Aquisição de recursos" do RAI é onde você começa algo que deve ser finalizado posteriormente, como por exemplo:

- Abrir um arquivo (que deve ser fechado mais tarde)
- Alocar alguma memória (e desalocá-la depois)

# Aquisição de Recurso É Inicialização (RAII)

- "Inicialização" do RAII significa que a aquisição ocorre dentro do construtor
  - A abertura de um arquivo via construtor

# Gerenciamento de Arquivos

```
#include <stdio>

class Arquivo{
    std::FILE* ptr_arquivo;
public:
    Arquivo(const char* nome_arquivo)
        : ptr_arquivo(std::fopen(nome_arquivo, "w+")){
        //...
    }
    ~Arquivo(){
        //...
    }
    void escreve(const char* texto){
        //...
    }
};
```

# Gerenciamento de Arquivos

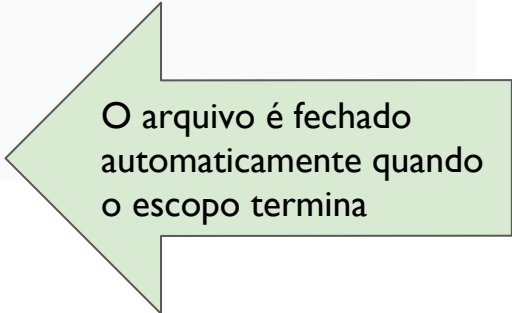
## ■ O uso da classe...

```
void exemplo_uso() {  
  
    Arquivo arquivoLog("arquivoLog.txt");  
  
    Arquivo.escreve("Olá, log da aula de PDS2!");  
  
}
```

# Gerenciamento de Arquivos

## Exemplo de uso da classe...

```
void exemplo_uso() {  
  
    Arquivo arquivoLog("arquivoLog.txt");  
  
    Arquivo.escreve("Olá, log da aula de PDS2!");  
  
}
```



O arquivo é fechado automaticamente quando o escopo termina

# Gerenciamento de Arquivos

- A classe Arquivo encapsula o gerenciamento do recurso \*FILE, adquirindo e liberando automaticamente a memória
- O arquivo é fechado automaticamente quando seu escopo termina

# Smart Pointers (Ponteiros Inteligentes)

---



# Ponteiros Inteligentes

- Implementações para auxiliar a manipulação de ponteiros
- São objetos que armazenam ponteiros para objetos alocados dinamicamente (no heap)
- Seu funcionamento é similar ao de um ponteiro tradicional

# Mas, qual a diferença?

- Eles deletam automaticamente o objeto apontado no momento certo (após o término da utilização)
  - Provém facilidade de desalocação automática de memória
  - Prevenção de memory leaks
- São úteis para assegurar a destruição do objeto apontado em caso de exceção

# Principais Vantagens

- Não precisamos nos lembrar de liberar a memória que foi alocada para uso do objeto
- Não é necessário usar **delete** e **free** em todos os objetos que foram declarados
- Elimina o risco de dangling pointers, que ocorrem quando os ponteiros apontam para objetos já deletados
- Consequentemente, redução da ocorrência de bugs

# Tipos Mais Comuns

- `unique_ptr` (antigo `auto_ptr`)

- Ponteiro único

- Permite um ponteiro por vez

- `shared_ptr`

- Ponteiro compartilhado (vários proprietários)

- Contador de referências

- `weak_ptr`

- uso em conjunto com `shared_ptr`

- posse temporária

# Ponteiros

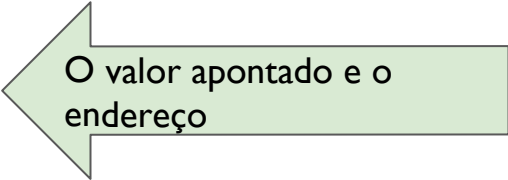
```
#include <iostream>

using namespace std;

int main() {
    int *pnum = new int();
    *pnum = 10;
    cout << *pnum << " -- " << pnum << endl;
    delete pnum;

    return 0;
}
```

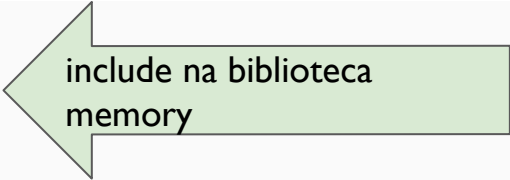
```
$ ./main
10 -- 0x6e1028
```



O valor apontado e o endereço

# Ponteiros Inteligentes

```
#include <iostream>
#include <memory>
```



include na biblioteca  
memory

```
using namespace std;
```

```
int main() {
    int *pnum = new int();
    *pnum = 10;
    cout << *pnum << " -- " << pnum << endl;
    delete pnum;

    return 0;
}
```

# unique\_ptr

## Ponteiro único

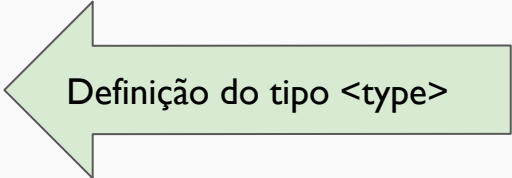
```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    //int *pnum = new int();
    unique_ptr<int>pnum(new int);
    *pnum = 10;
    cout << *pnum << " -- " << &pnum << endl;

    //delete pnum;

    return 0;
}
```



Definição do tipo <type>

# unique\_ptr

```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    //int *pnum = new int();
    unique_ptr<int>pnum(new int);
    *pnum = 10;
    cout << *pnum << " -- " << &pnum << endl;

    //delete pnum;

    return 0;
}
```

Incluo '&' (notação de endereço)

Não preciso mais do delete

```
$ ./main
10 -- 0x6e1028
```



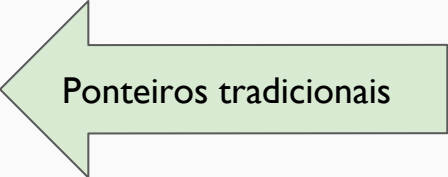
# Strings

```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    string *str = new string("Aula PDS2");
    cout << *str << " -- Tamanho: " << str->size() << endl;
    delete str;

    return 0;
}
```



Ponteiros tradicionais

```
$ ./main
Aula PDS2 -- Tamanho: 9
```

# unique\_ptr

```
#include <iostream>
#include <memory>

using namespace std;

int main() {
    //string *str = new string("Aula PDS2");
    unique_ptr<string>str(new string("Aula PDS2"));
    cout << *str << " -- Tamanho: " << str->size() << endl;
    delete str;

    return 0;
}
```

```
$ ./main
Aula PDS2 -- Tamanho: 9
```

# Classes

```
#include <iostream>
#include <memory>
```

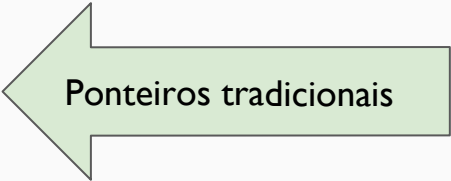
```
using namespace std;
```

```
class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
};
```

```
int main() {
    Aluno *a = new Aluno();
    cout << "Nota: " << a->getNota() << endl;
    delete a;

    return 0;
}
```

```
$ ./main
Nota: 0
```



Ponteiros tradicionais

# Classes

```
#include <iostream>
#include <memory>
```

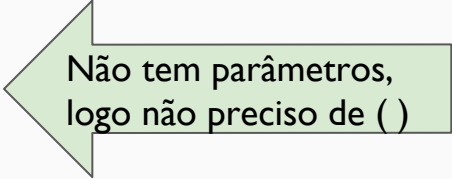
```
using namespace std;
```

```
class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
};
```

```
int main() {
    //Aluno *a = new Aluno();
    unique_ptr<Aluno>a(new Aluno);
    cout << "Nota: " << a->getNota() << endl;
    //delete a;

    return 0;
}
```

```
$ ./main
Nota: 0
```



Não tem parâmetros,  
logo não preciso de ( )

# Classes

```
#include <iostream>
#include <memory>
```

```
using namespace std;
```

```
class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
};
```

```
int main() {
    //Aluno *a = new Aluno();
    unique_ptr<Aluno>a(new Aluno);
    unique_ptr<Aluno>b=a;
    cout << "Nota: " << a->getNota() << endl;
    //delete a;

    return 0;
}
```



# Classes

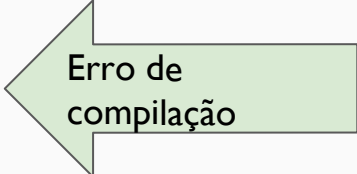
```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
};

int main() {
    //Aluno *a = new Aluno();
    unique_ptr<Aluno>a(new Aluno);
    unique_ptr<Aluno>b=a;
    cout << "Nota: " << a->getNota() << endl;
    //delete a;

    return 0;
}
```



Erro de  
compilação

Permite um único ponteiro por vez

# shared\_ptr

## Ponteiro compartilhado

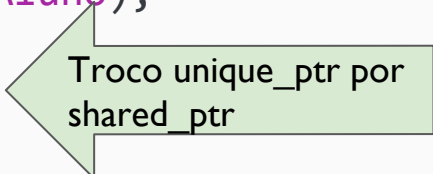
```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
    void setNota(int n){
        this->nota=n;
    }
};
```

```
int main() {
    shared_ptr<Aluno>a(new Aluno);
    shared_ptr<Aluno>b=a;
    a->setNota(70);
    b->setNota(80);
    cout << "Nota a: " << a->getNota() << endl;
    cout << "Nota b: " << b->getNota() << endl;

    return 0;
}
```



Troco unique\_ptr por shared\_ptr

# shared\_ptr

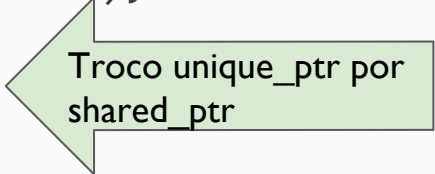
```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
    void setNota(int n){
        this->nota=n;
    }
};
```

```
int main() {
    shared_ptr<Aluno>a(new Aluno);
    shared_ptr<Aluno>b=a;
    a->setNota(70);
    b->setNota(80);
    cout << "Nota a: " << a->getNota() << endl;
    cout << "Nota b: " << b->getNota() << endl;

    return 0;
}
```



Troco unique\_ptr por  
shared\_ptr

■ O que será impresso?



# shared\_ptr

```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
    void setNota(int n){
        this->nota=n;
    }
};
```

```
int main() {
    shared_ptr<Aluno>a(new Aluno);
    shared_ptr<Aluno>b=a;
    a->setNota(70);
    b->setNota(80);
    cout << "Nota a: " << a->getNota() << endl;
    cout << "Nota b: " << b->getNota() << endl;

    return 0;
}
```

```
$ ./main
Nota a: 80
Nota b: 80
```

■ Por que?

# shared\_ptr

```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
    void setNota(int n){
        this->nota=n;
    }
};
```

```
int main() {
    shared_ptr<Aluno>a(new Aluno);
    shared_ptr<Aluno>b=a;
    a->setNota(70);
    //b->setNota(80);
    cout << "Nota a: " << a->getNota() << endl;
    cout << "Nota b: " << b->getNota() << endl;

    return 0;
}
```

```
$ ./main
Nota a: 70
Nota b: 70
```

Eles estão usando o mesmo ponteiro

# Qual a solução neste caso?

# Qual a solução neste caso?

```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    int nota = 0;
    int getNota(){
        return nota;
    }
    void setNota(int n){
        this->nota=n;
    }
};
```

```
int main() {
    unique_ptr<Aluno>a(new Aluno);
    unique_ptr<Aluno>b(new Aluno);
    a->setNota(70);
    b->setNota(80);
    cout << "Nota a: " << a->getNota() << endl;
    cout << "Nota b: " << b->getNota() << endl;

    return 0;
}
```

```
$ ./main
Nota a: 70
Nota b: 80
```

# Utilizando listas de inicialização

```
#include <iostream>
#include <memory>

using namespace std;

class Aluno{
public:
    const char* nome;
    int nota;

    Aluno(const char* n, int nt):
nome(n), nota(nt){
    //...
}
};

int main() {
    unique_ptr<Aluno>a(new Aluno{"Julio",90});

    cout << "Nome: " << a->nome << endl;
    cout << "Nota: " << a->nota << endl;

    return 0;
}
```

```
$ ./main
Nome: Julio
Nota: 90
```