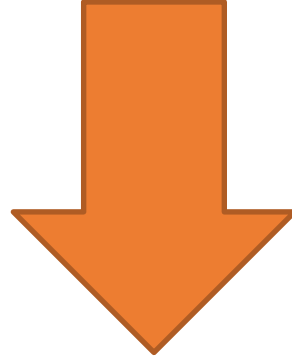


IMD0030 – LINGUAGEM DE PROGRAMAÇÃO I

Aula 16 – Ponteiros Inteligentes.

(material baseado nas notas de aula do Prof. Silvio Sampaio e Prof. Ivan Ricarte -UNICAMP)



Antes de Ponteiros Inteligentes,
Informação útil

Deduzindo tipos com o uso de **auto**

- Em C++11 uma variável pode ser declarada como sendo “do tipo” **auto**
 - Isso diz ao compilador que o tipo da variável deverá ser deduzido de seus inicializadores (por isso devem ser inicializadas)
- Exemplos de uso:
 - `int x;` // a variável x é do tipo int – declaração explícita (tipada)
 - `auto x = 10;` // variável x é deduzida como sendo inteira, já que 10 é um inteiro

Uso comum do auto

```
using std::string;
using std::vector;

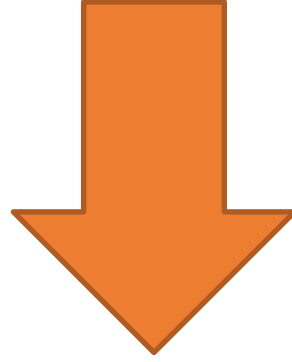
std::vector<std::string> nomes;

//
// Como é hoje
//
for(std::vector<string>::iterator i = nomes.begin() ; i != nomes.end() ; ++i)
{

}

//
// Muito mais fácil de ler.
//
for(auto i = nomes.begin() ; i != nomes.end() ; ++i)
{

}
```



Agora retornamos ao início de Ponteiro
Inteligentes

Alocação dinâmica de memória: Problemas comuns

- Memory Leak
 - Memória que é alocada não é devolvida (liberada)
 - Buffer Overflow
 - Escrever fora da área alocada
 - Memória não inicializada
 - Acesso a memória não inicializada
 - Pode induzir comportamentos “aleatórios”
 - Double-free
 - A área de memória de um ponteiro é liberada mais do que uma vez
 - Origina crash na maioria das libcs
-

Ponteiros inteligentes em C++

In brief, smart pointers are **C++ objects** that **simulate simple pointers** by implementing **operator->** and the unary **operator***. In addition to sporting pointer syntax and semantics, smart pointers often perform useful tasks—such as **memory management or locking**—under the covers, thus freeing the application from carefully **managing the lifetime of pointed-to objects**.

Andrei Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001.

Uso de *smart pointers* para evitar *memory leak*

- Em C++ 11 é possível usar ponteiros inteligentes (*smart pointers*) para alocar memória dinamicamente sem ter que se preocupar com sua liberação após acabar o seu uso

```
void UsaPonteiroBruto()
{
    // Utiliza um ponteiro bruto -- não recomendado.
    Dado* umDado = new Dado();

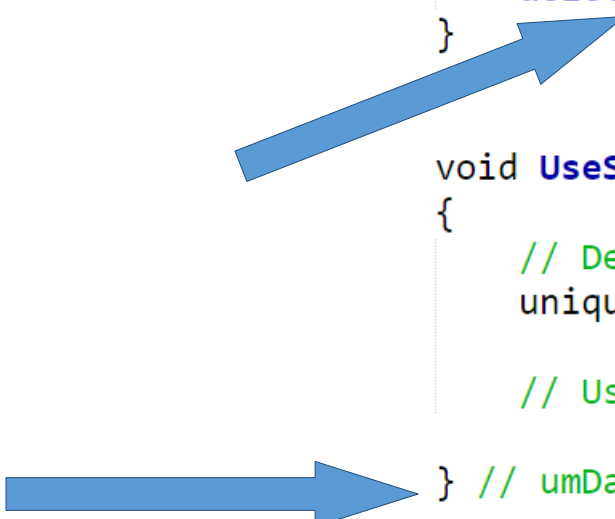
    // Usa o umDado...

    // Não se esqueça de liberar da memória!
    delete umDado;
}

void UseSmartPointer()
{
    // Declara como um ponteiro inteligente.
    unique_ptr<Dado> umDado = new Dado();

    // Usa um Dado...

} // umDado é liberado da memória (removido) automaticamente.
```



Estrutura mínima de um ponteiro inteligente

```
template <typename T>
class PonteiroInteligente {
public:
    PonteiroInteligente (T* _ponteiro): ponteiro(_ponteiro);
    ~PonteiroInteligente () {
        delete ponteiro;
        std::cout << "Ponteiro liberado." << std::endl;
    };
    T* operator->() const { return ponteiro; };
    T& operator*() const { return *ponteiro; };
private:
    T* ponteiro;
}
```

mais eficiente ao
copiar objetos grandes
e permite operações
como *T = <value>

indica que o método é “read-only” e não pode modificar
membros não-estáticos da classe

Usando o PonteiroInteligente

```
#include "ponteirointeligente.h"
```

```
int main(int argc, char const *argv[])
```

```
{
```

```
    PonteiroInteligente<int> ptr(new int(80));
```

```
    std::cout << (*ptr) << std::endl;
```

```
    return 0;
```

```
}
```

Libera a memória apontada.

Aloca memória e cria o ponteiro.

Faz uso do ponteiro e da área de memória alocada.

Ponteiros Inteligentes no C++

- Recurso incorporado no C++11
 - Classes parametrizadas (definidas na biblioteca **memory**) para ponteiros inteligentes:
 - *unique_ptr*
 - *shared_ptr*
 - *weak_ptr*
 - Seleção da classe de ponteiro reflete a intenção do programador em seu uso
 - Ao contrário do que ocorre com ponteiro tradicional
 - Tornam programação mais simples e código mais robusto
-

O ponteiro `unique_ptr`

- Existe uma única referência para o objeto apontado
 - Ponteiro não pode ser “copiado”
 - Não pode ser atribuído a outro ponteiro diretamente, armazenado em um container ou passado como argumento para uma função
 - Posse do ponteiro pode ser transferida
 - Conteúdo é movido e área anterior passa a ser inválida
 - Mover é sempre mais eficiente que copiar
 - Substitui **`auto_ptr`** (é outro ponteiro inteligente)
 - Para transferir a posse do ponteiro explicitamente, introduz função **`move()`**
 - **`std::make_unique()`** permite criar um ponteiro inteligente **`unique_ptr`** (C++14)
 - “Don’t write new”
-

Exemplo: unique_ptr

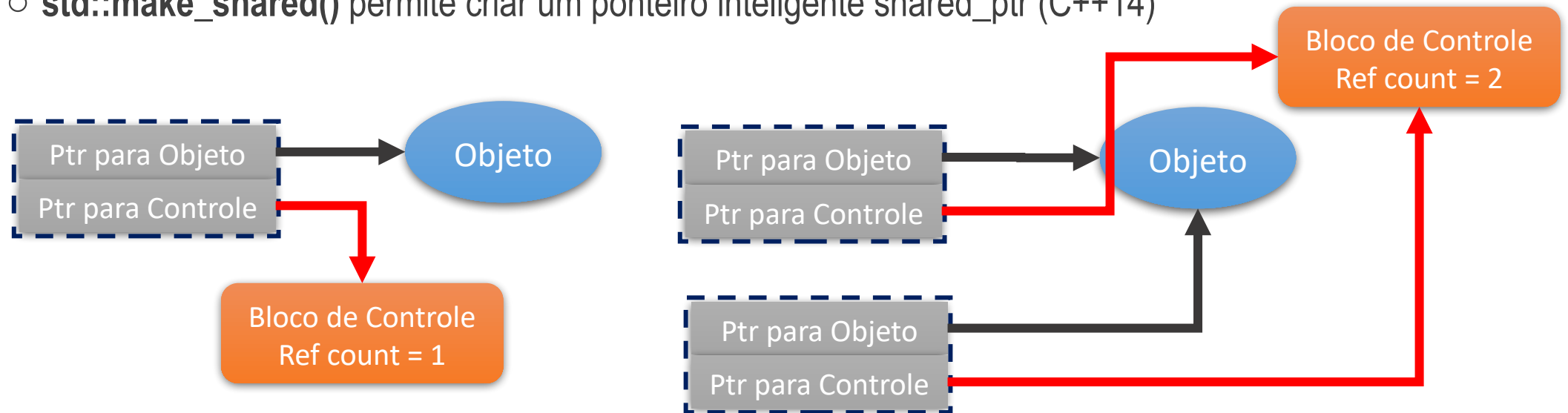
```
#include <iostream>
#include <memory>
```

```
int main(int argc, char const *argv[])
{
    std::unique_ptr<int> ptr1(new int);
    std::unique_ptr<int> ptr2(nullptr);
    *ptr1 = 25;
    std::cout << (*ptr1) << std::endl;
    // ptr2 = ptr1; // Causaria erro!
    ptr2 = std::move(ptr1); // É preciso transferir a posse
    //std::cout << (*ptr1) << std::endl; // Causaria erro!
    std::cout << (*ptr2) << std::endl;
    return 0;
}
```

unique_ptr<int> ptr1 = make_unique<int>();

O ponteiro `shared_ptr`

- Ponto para um recurso que pode ser compartilhado
 - Com controle do número de referências
 - Somente quando última referência deixa de existir, o recurso é liberado
 - Ocupa o dobro de um ponteiro tradicional
 - `std::make_shared()` permite criar um ponteiro inteligente `shared_ptr` (C++14)



Exemplo: shared_ptr

```
1 #include <iostream>
2 #include <memory>
3 using namespace std;
4
5 class A {
6 public:
7     void show() { cout << "A::show()" << endl; }
8 };
9
10 int main()
11 {
12     shared_ptr<A> p1(new A);
13     cout << p1.get() << endl;
14     p1->show();
15     shared_ptr<A> p2(p1);
16     p2->show();
17     cout << p1.get() << endl;
18     cout << p2.get() << endl;
19
20     cout << p1.use_count() << endl;
21     cout << p2.use_count() << endl;
22
23     p1.reset();
24     cout << p1.get() << endl;
25     cout << p2.use_count() << endl;
26     cout << p2.get() << endl;
27
28     return 0;
29 }
```

0x625010

A::show()

A::show()

0x625010

0x625010

2

2

0

1

0x625010

```
1 // shared_ptr::get example
2 #include <iostream>
3 #include <memory>
4
5 int main () {
6     int* p = new int (10);
7     std::shared_ptr<int> a (p);
8
9     if (a.get()==p)
10         std::cout << "a and p point to the same location\n";
11
12     // three ways of accessing the same address:
13     std::cout << *a.get() << "\n";
14     std::cout << *a << "\n";
15     std::cout << *p << "\n";
16
17     return 0;
18 }
19
```

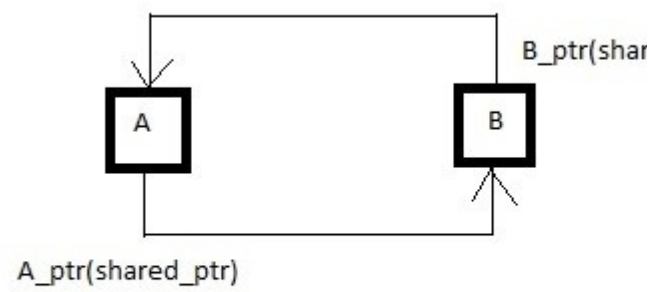
Risco em compartilhar referências

- **Referências cíclicas**

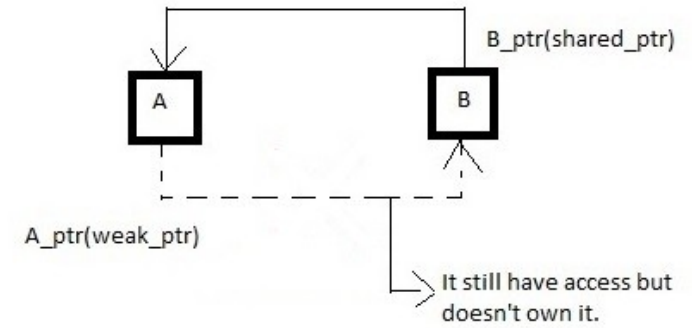
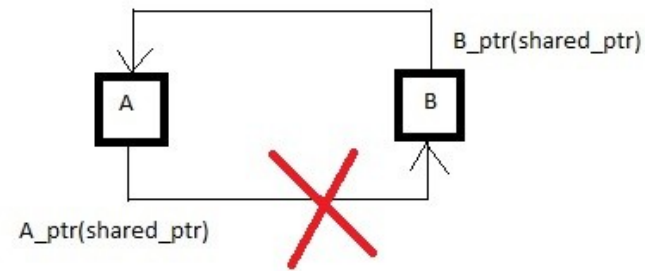
- Um objeto mantém referências circulares entre objetos

- **Solução:** uso do **weak_ptr**

- Ponteiro inteligente para uso em conjunto com **shared_ptr**
 - Um **weak_ptr** fornece acesso a um objeto que pertence a um ou mais instâncias de **shared_ptr**, mas não participa de contagem de referência
 - Necessária em alguns casos para quebrar referências circulares entre instâncias **shared_ptr**
-



Circular Reference



```

1  #include <iostream>
2  #include <memory>
3  int main()
4  {
5      // PROBLEM: ref will point to undefined data!
6      int* ptr = new int(10);
7      int* ref = ptr;
8      delete ptr;
9
10     // SOLUTION: check expired() or lock() to determine if pointer is valid
11     // empty definition
12     std::shared_ptr<int> sptr;
13
14     // takes ownership of pointer
15     sptr.reset(new int);
16     *sptr = 10;
17
18     // get pointer to data without taking ownership
19     std::weak_ptr<int> weak1 = sptr;
20
21     // deletes managed object, acquires new pointer
22     sptr.reset(new int);
23     *sptr = 5;
24
25     // get pointer to new data without taking ownership
26     std::weak_ptr<int> weak2 = sptr;
27
28     // weak1 is expired!
29     if(auto tmp = weak1.lock())
30         std::cout << "weak1 value is " << *tmp << '\n';
31     else
32         std::cout << "weak1 is expired\n";
33
34     // weak2 points to new data (5)
35     if(auto tmp = weak2.lock())
36         std::cout << "weak2 value is " << *tmp << '\n';
37     else
38         std::cout << "weak2 is expired\n";
39 }

```

weak1 is expired
weak2 value is 5

Prefira o `std::make_unique` e `std::make_shared`

- Dicas úteis:
 - Prefira usar o `std::make_unique` e `std::make_shared` como substituto ao uso direto do `new`
 - Recurso do C++14, o `std::make_unique` constrói um `std::unique_ptr` do ponteiro cru que o comando `new` produz
 - Recurso do C++11, o `std::make_shared` constrói um `std::shared_ptr` do ponteiro cru que o comando `new` produz
-

Exemplo: `std::make_unique`

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

int main(int argc, char const *argv[])
{
    std::unique_ptr<int> ptr1 = std::make_unique<int>(25);
    std::unique_ptr<int> ptr2(nullptr);
    std::cout << (*ptr1) << std::endl;
    // ptr2 = ptr1; // Causaria erro!
    ptr2 = std::move(ptr1); // É preciso transferir a posse
    //std::cout << (*ptr1) << std::endl; // Causaria erro!
    std::cout << (*ptr2) << std::endl;
    return 0;
}
```

Exemplo: `std::make_shared`

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

void imprime(std::shared_ptr<int> valor) {
    std::cout << "Valor recebido: " << (*valor) << std::endl;
}

int main(int argc, char const *argv[])
{
    std::shared_ptr<int> ptr1 = std::make_shared<int>(33);
    imprime(ptr1);
    std::cout << (*ptr1) << std::endl;
    return 0;
}
```

Novo paradigma: ponteiros inteligentes

- Quando usar ponteiros tradicionais em C++?
 - Praticamente **NUNCA**
 - Quando usar os ponteiros inteligentes em C++?
 - Apenas quando a semântica de ponteiros for necessária
 - Quando um objeto precisa ser compartilhado
 - Quando é necessário fazer uma referência polimórfica (funções com mesmo nome, herança, funções virtuais, etc)
 - Para as demais situações, usar as classes da biblioteca padrão de C++ (STL)
-

?

