

Programação Orientada a Objetos

Introdução à Orientação a Objetos

- Inicialização e Destruição de Objetos

Para onde vamos ...

- Agora veremos ...
 - III.3. Inicialização e destruição
- Referência básica
 - *Thinking in C++*, Vol. 1
 - Capítulo 6



III.3. Inicialização e destruição

- Grande parte dos erros em um programa C ocorre quando o programador se esquece de inicializar ou de “limpar” uma variável
 - Isto é particularmente verdadeiro com bibliotecas, quando um “programador cliente” não sabe como inicializar as estruturas de dados, ou mesmo não sabe que ele deveria fazer isto
 - A “limpeza final” é um problema sério, porque muitas vezes os programadores simplesmente usam suas estruturas de dados e se esquecem que ao final de seu uso devem fazer uma “faxina” ...



III.3. Inicialização e destruição

- Em C++, os processos de inicialização e de “faxina final” podem ser automatizados nos objetos que criamos
 - Funções especialmente concebidas para isto: os **construtores** e os **destrutores**!
- Iremos revisitar a classe CWindow:
 - Nela o processo de inicialização era feito pela função initialize();
 - O usuário da classe poderia se esquecer de chamar initialize() => desastre, pois a janela não estaria inicializada corretamente quando a usássemos
 - Modificaremos isto para que seja feito automaticamente por um construtor



III.3. Inicialização e destruição

- O que são os construtores?
 - São funções que têm o propósito explícito de inicializar objetos
 - Têm o mesmo nome da classe, não retornam nada, têm chamada automática na declaração dos objetos

```
class X {  
    int i;  
public:  
    X(); // Construtor  
};  
  
void f() {  
    X a; // A função X() é executada automaticamente !!!!!  
    // ...  
}
```

III.3. Inicialização e destruição

- O que são os destrutores?
 - Função complementar às funções construtoras de uma classe
 - Sempre que o escopo de um objeto encerra-se, esta função é chamada
 - Cada classe pode ter somente **um** destrutor, que jamais recebe parâmetros
 - O destrutor também não tem nenhum tipo de retorno

```
class Y {  
public:  
    ~Y();  
};
```

III.3. Inicialização e destruição

```
//: C06:Constructor1.cpp
#include <iostream>
using namespace std;

class Tree {
    int height;
public:
    Tree(int initialHeight); // Constructor
    ~Tree(); // Destructor
    void grow(int years);
    void printsize();
};

Tree::Tree(int initialHeight) {
    height = initialHeight;
}

Tree::~Tree() {
    cout << "inside Tree destructor" << endl;
    printsize();
}
```

```
void Tree::grow(int years) {
    height += years;
}

void Tree::printsize() {
    cout << "Tree height is " << height << endl;
}

int main() {
    cout << "before opening brace" << endl;
    {
        Tree t(12);
        cout << "after Tree creation" << endl;
        t.printsize();
        t.grow(4);
        cout << "before closing brace" << endl;
    }
    cout << "after closing brace" << endl;
}
```

III.3. Inicialização e destruição

- Revisitando a classe CWindow:

```
#ifndef CWINDOW_H
#define CWINDOW_H
class CWindow {
    int x, y;           // Posição na tela
    int cx, cy ;      // Largura e altura
    Canvas* my_canvas;
public:
    CWindow (int xp, int yp, int cx, int cy); // Construtor
    ~CWindow() ;          // Destrutor
    int show();           // Mostra na tela
    int move(int newx, int newy);           // Move
    int resize(int newcx, int newcy); // Redimensiona
    int setTextColor(COLORREF cor);   //
    int textOut(int x, int y, char* text); // Texto em x,y
};
#endif // CWINDOW_H ///
```

III.3. Inicialização e destruição

```
#include "CWindow.h"

CWindow::CWindow (int xp, int yp, int cxp, int cyp) {
    // Construtora
    x = xp;
    y = yp;
    cx = cxp;
    cy = cyp;
    my_canvas = new Canvas; // retorna um ponteiro do tipo Canvas
}

CWindow::~CWindow () {
    delete my_canvas; // desaloca memória - é complementar a new
}
```

III.3. Inicialização e destruição

- Em C, as variáveis só podem ser declaradas no início de blocos
- Em C++, elas podem ser declaradas em qualquer parte
 - Isto é feito para que se possa conseguir informação suficiente para inicializar os objetos por meio de seus construtores
 - Um objeto não pode ser criado se ele não for também inicializado
 - Você pode esperar ter as informações necessárias para então definir e inicializar um objeto ao mesmo tempo
 - Um construtor tem como obrigações
 - Inicializar todas as variáveis do objeto
 - Garantir, dentro do possível, a validade dos dados

III.3. Inicialização e destruição

- Boa prática: definir variáveis o mais perto possível do seu local de uso e sempre inicializá-las quando são definidas - é uma questão de segurança
 - Ao reduzir a duração da disponibilidade da variável dentro do escopo, reduz-se a chance dela ser mal utilizada em alguma outra parte do escopo
 - Além disso, melhora a legibilidade – o leitor não tem que desviar sua atenção para outro local para obter o tipo da variável

III.3. Inicialização e destruição

- Inicialização agregada à definição:

```
int a[5] = { 1, 2, 3, 4, 5 }; // inicializa todas as 5 posições
int b[6] = {9}; // inicializa o primeiro com 9 e atribui 0 aos demais
int c[10] = {0}; // forma abreviada de inicializar sem usar o for
// nota: a inicialização não ocorre se não incluir a lista de valores
int c[] = { 1, 2, 3, 4 }; // define o tamanho do vetor após
// conhecer a lista de valores
// nota: para saber o tamanho, usar (sizeof c / sizeof *c)
struct X { int i; float f; char c; };
X x1 = { 1, 2.2, 'c' }; // inicialização da estrutura
//nota: seus dados são públicos => ok)
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} }; // inicialização de
// vetor da estrutura
```

III.3. Inicialização e destruição

- Construtor *default*
 - É um construtor que pode ser chamado sem argumentos
 - Se uma classe não tem nenhum construtor, é criado automaticamente pelo compilador um construtor *default*
 - A criação de qualquer construtor pelo programador faz com que este construtor *default* não seja mais criado automaticamente!

- Exemplo:

```
class Y {  
    int i;  
};  
class X {  
    int i;  
    X(int k);  
};  
Y a, b, c[4];      // OK, existe construtor default na classe Y  
X d(3);          // OK  
X e, f[3] ;        // Erro! não existe construtor default para X
```

III.3. Inicialização e destruição

- Importante: problemas pela falta do construtor *default*
 - Situações nas quais não existem detalhes suficientes para realizar a inicialização (e não se tem construtor *default*)
 - Ex: construtor definido e uso de inicialização agregada à definição
 - struct Y { float f; int i; Y(int a); };
 - Y y2[2] = { Y(1) }; // erro: não se sabe como inicializar o 2º elemento
 - Y y3[7]; // erro: não se sabe como inicializar os elementos do vetor
 - Y y4; // erro: não se tem construtor *default*
 - O construtor *default* é tão importante que, se não há nenhum construtor para uma estrutura (struct ou classe), é criado automaticamente um

// Automatically-generated default constructor

```
class V {  
    int i; // private  
}; // No constructor  
int main() {  
    V v, v2[10];  
}
```

Nota: o construtor gerado pelo compilador não realiza inicialização de valores. Se você quiser inicializar, crie explicitamente um construtor *default*. De uma maneira geral, você deve criar seus construtores evitando deixar o compilador assumir isso para você.

III.3. Inicialização e destruição

- O que será impresso pelo programa?

```
//: C06:Multiarg.cpp
#include <iostream>
using namespace std;

class Z {
    int i, j;
public:
    Z(int ii, int jj);
    void print();
};

Z::Z(int ii, int jj) {
    i = ii;
    j = jj;
}

void Z::print() {
    cout << "i = " << i << ", j = " << j << endl;
}

int main() {
    Z zz[] = { Z(1,2), Z(3,4), Z(5,6), Z(7,8) };
    for(int i = 0; i < sizeof zz / sizeof *zz; i++)
        zz[i].print();
}
```

```
i = 1, j = 2
i = 3, j = 4
i = 5, j = 6
i = 7, j = 8
```

III.3. Inicialização e destruição

12. Modifique os programas abaixo para que os mesmos utilizem construtores e destrutores.

```
///: C05:Handle.h
#ifndef HANDLE_H
#define HANDLE_H

class Handle {
    struct Cheshire;
    Cheshire* smile;
public:
    void initialize();
    void cleanup();
    int read();
    void change(int);
};

#endif // HANDLE_H ///:~
```

```
///: C05:Handle.cpp {O}
#include "Handle.h"
#include "../require.h"

struct Handle::Cheshire {
    int i;
};

void Handle::initialize() {
    smile = new Cheshire;
    smile->i = 0;
}

void Handle::cleanup() {
    delete smile;
}

int Handle::read() {
    return smile->i;
}

void Handle::change(int x) {
    smile->i = x;
}
```

```
///: C05:UseHandle.cpp
#include "Handle.h"
int main() {
    Handle u;
    u.initialize();
    u.read();
    u.change(1);
    u.cleanup();
}
```