

# Programação Orientada a Objetos

Inicialização e Destruição

Sobrecarga de Funções

Constantes

Controle de Visibilidade

# Onde estamos ...

- Na Unidade III já vimos ...
  - III.1. Implementando classes e objetos em C++
  - III.2. Atributos e métodos: controle de acesso e encapsulamento



# Para onde vamos ...

- Agora veremos ...
  - III.3. Inicialização e destruição
  - III.4. Sobrecarga de funções e argumentos *default*
  - III.5. Constantes, funções *inline* e controle de visibilidade
- Referência básica
  - *Thinking in C++, Vol. 1*
    - Parte 2: Capítulos 6 ao 10



# Agenda

## Unidade III - Classes e Objetos

Parte2

- III.1. Implementando classes e objetos em C++
- III.2. Atributos e métodos: controle de acesso e encapsulamento
- III.3. Inicialização e destruição
- III.4. Sobrecarga de funções e argumentos *default*
- III.5. Constantes e controle de visibilidade
- III.6. Ponteiros, referências, atributos dinâmicos, gerenciamento de memória e o construtor de cópia
- III.7. Sobrecarga de operadores e conversão de tipos



### 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:

### 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: não se pode utilizar construtor *default* quando...
  - Existem dados privados na estrutura ou classe
  - Existe pelo menos um construtor definido
    - Exemplo: construtor definido (inicialização agregada à definição)
      - struct Y { float f; int i; Y(int a); };
      - Y y1[] = { Y(1), Y(2), Y(3) }; //OK, chama o construtor definido

// Multiple constructor arguments with aggregate initialization

```
#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();
}
```

### 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();
}
```

## III.4. Sobrecarga de funções

- Em português, uma palavra pode ter diferentes significados
  - Tudo depende do contexto em que ela foi empregada
- A palavra está sendo **sobre carregada** ...
- Isto é particularmente útil em situações corriqueiras
  - Lavar uma camisa e lavar um carro ...
    - Imagine se fossemos forçados a usar um verbo diferente para cada uma destas ações
    - em C somos forçados a usar “**verbos**” diferentes, para funções muito semelhantes

## III.4. Sobrecarga de funções

- Em C teríamos que criar funções com nomes diferentes, quando a única coisa que as diferencia conceitualmente é o tipo de dados recebido

```
int escalar_int(int [], int [], int);  
float escalar_float(float [], float [], int);
```

- Em C++ poderíamos usar um único nome para a função: estaríamos efetuando uma sobrecarga

```
int escalar(int [], int [], int);  
float escalar(float [], float [], int);
```

## III.4. Sobrecarga de funções

- Em C++, outro fator nos força a ter sobrecarga de funções
  - A possibilidade de criarmos vários construtores diferentes
    - Poder criar objetos de diferentes maneiras!
  - Também é possível omitirmos alguns parâmetros da lista de argumentos, quando fizermos a chamada a certas funções
    - Neste caso os argumentos receberão valores “*default*”

## III.4. Sobrecarga de funções

- A idéia da sobrecarga de funções é muito simples
  - Usa-se um mesmo nome para a função, mas uma lista de argumentos diferentes
    - O compilador utiliza o escopo, o nome da função e a lista de argumentos para reconhecer cada função sobrecarregada
    - Apesar das funções terem o mesmo nome, o compilador conseguirá distinguir que função chamar por meio dos parâmetros utilizados

```
void print(char);
```

```
void print(float);
```

```
...
```

```
print('a');
```

```
print(3.1415);
```

## III.4. Sobrecarga de funções

- Uma pergunta que muitas vezes se faz é:  
pode-se fazer a sobrecarga pelo tipo de retorno?

`void f();`

`int f();`

- **Não!** Como o compilador iria saber qual função chamar quando se escrevesse

`f(); ??`

## III.4. Sobrecarga de funções - Construtores

```
class Ponto {  
private:  
    int x,y;  
public:  
    Ponto( );  
    Ponto(int,int);  
};  
  
Ponto::Ponto(int x1,int y1){  
    x=x1;  
    y=y1;  
}  
Ponto::Ponto(){  
    x=0;  
    y=0;  
}
```

```
int main() {  
    Ponto A, B(3,4);  
    Ponto *p= new Ponto(3,4);  
    Ponto *p1 = new Ponto;  
    Ponto Vpoints[10];  
    ...  
}
```

## III.4. Sobrecarga de funções

- No exemplo anterior, os dois construtores executam basicamente o mesmo código
  - Um deles apenas, quando não recebe argumentos, inicializa x e y com zero
- Não poderíamos usar um único construtor?
  - Sim, desde que usássemos argumentos *default!*
    - Valor fornecido na declaração que o compilador insere automaticamente se não for fornecido um valor na chamada
      - Ex: f(int a, int b=0) pode ser chamado por:  
f(150, 2); f(200, 0); f(100);
      - O segundo argumento é automaticamente substituído pelo compilador caso não seja fornecido e o primeiro seja *int*

## III.4. Sobrecarga de funções

- Argumentos *default*

```
class Ponto {  
private:  
    int x,y;  
public:  
    Ponto(int=0,int=0);  
};
```

```
Ponto::Ponto(int x1,int y1){  
    x=x1;  
    y=y1;  
}
```

```
int main() {  
    Ponto A, B(3,4);  
    Ponto *p= new Ponto(3,4);  
    Ponto *p1 = new Ponto;  
    Ponto Vpoints[10];  
    ...  
}
```

## III.4. Sobrecarga de funções

- Argumentos *default*
  - Se os dois construtores executassem ações completamente diferentes, não faria sentido usar argumentos *default*, mas sim continuar a ter sobrecarga das funções
  - As seguintes regras devem ser seguidas:
    - Somente os últimos argumentos da lista podem ter valores *default*
      - Não se pode ter argumento *default* seguido por um não *default*
      - Uma vez que começou a usar valor *default*, todos os demais devem ser *default*
    - Não se pode omitir argumentos no meio da lista
    - Os valores *default* somente são colocados na declaração da função
      - O compilador precisa conhecer os valores *default* antes de usá-los
      - Algumas vezes pode-se colocar estes valores comentados na definição, apenas para fins de documentação

```
void fn (int x /* = 0 */ ) { // ...}
```

## III.4. Sobrecarga de funções

- Argumentos *default*

- Exemplos:

f( int a =5, int b, float c); // NÃO PODE

g(int a, float b =0., float c=3.); // OK!

chamadas:

g(1), g(2, 7.), g(4, 5., 6.); // OK!

g(1, , 7); // NÃO PODE !

- Argumentos *default* devem ser utilizados para tornar as chamadas a funções mais simples, quando houver uma grande lista de argumentos que podem receber valores típicos
    - O valor *default* deve ser o valor que mais provavelmente vai ocorrer para aquele argumento, de forma que os programadores clientes poderão muitas vezes ignorá-lo ou utilizá-lo somente quando for necessário modificar este valor padrão

## III.4. Sobrecarga de funções

- Argumentos *default*
  - Cuidado para não confundir o compilador usando a sobrecarga e argumentos *default* simultaneamente onde não poderia ...
    - void impressao(int,float);
    - void impressao(int, float, char='a');

quem o compilador chamaria ?

- impressao(1, 5.) ;



## III.4. Sobrecarga de funções

13. Modifique a classe abaixo para usar argumentos default no construtor. Teste o construtor criando dois objetos diferentes.

```
///: C07:Stash3.h
#ifndef STASH3_H
#define STASH3_H

class Stash {
    int size;      // Size of each space
    int quantity; // Number of storage spaces
    int next;      // Next empty space
    // Dynamically allocated array of bytes:
    unsigned char* storage;
    void inflate(int increase);
public:
    Stash(int size); // Zero quantity
    Stash(int size, int initQuantity);
    ~Stash();
    int add(void* element);
    void* fetch(int index);
    int count();
};

#endif // STASH3_H
```

```
//: C07:Stash3.cpp (O)
#include "Stash3.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
    size = sz;
    quantity = 0;
    next = 0;
    storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
    size = sz;
    quantity = 0;
    next = 0;
    storage = 0;
    inflate(initQuantity);
}

Stash::~Stash() {
    if(storage != 0) {
        cout << "freeing storage" << endl;
        delete []storage;
    }
}
```

```
int Stash::add(void* element) {
    if(next >= quantity) // Enough space left?
        inflate(increment);
    // Copy element into storage,
    // starting at next empty space:
    int startBytes = next * size;
    unsigned char* e = (unsigned char*)element;
    for(int i = 0; i < size; i++)
        storage[startBytes + i] = e[i];
    next++;
    return(next - 1); // Index number
}

void* Stash::fetch(int index) {
    require(0 <= index, "Stash::fetch (-)index");
    if(index >= next)
        return 0; // To indicate the end
    // Produce pointer to desired element:
    return &(storage[index * size]);
}

int Stash::count() {
    return next; // Number of elements in CStash
}

void Stash::inflate(int increase) {
    assert(increase >= 0);
    if(increase == 0) return;
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i]; // Copy old to new
    delete []storage; // Release old storage
    storage = b; // Point to new memory
    quantity = newQuantity; // Adjust the size
}
```

## III.4. Sobrecarga de funções

14. Uma definição da classe Time correta pode incluir os dois construtores a seguir? Se não, explique o por quê.

```
Time (int h=0, int m=0, int s=0);  
Time ();
```

## III.5. Constantes e controle de visibilidade

- Capítulos 8, 9 e 10 do livro
  - III.5.1. Constantes
  - III.5.2. Funções *inline*
  - III.5.3. Controle de visibilidade de nomes

## III.5.1. Constantes

- Objetivo na linguagem C++
  - Permitir que o programador especifique o que pode ser modificado ou não em seu programa
  - Fornece maior controle e segurança a um programa em C++
    - Compilador acusa erro se você utilizar tipos incompatíveis em uma operação envolvendo constante ou alterar uma constante de forma acidental
- Especificação na linguagem C++
  - Utiliza a palavra reservada **const**, que pode ter diferentes significados, dependendo de onde ela aparece
  - Para criar uma **constante** no programa, evitando “números mágicos”:

```
const int bufsize = 100;      // substitui #define BUFSIZE 100: macro na qual o
...                          // pré processador substitui valor, não ocupa memória
char buffer[bufsize];
```

## III.5.1. Constantes

- Vantagem do uso de constantes
  - Sobre o `#define`
    - Além de atuar de forma semelhante, uma constante tem um tipo, que pode ser verificado pelo compilador: uma macro não tem
  - Outras vantagens adicionais
    - Constantes também podem ser usadas para garantir que uma variável, depois de inicializada, não vai ter seu valor modificado ao longo do programa
    - Em geral não ocupa memória (isso depende da complexidade do tipo de dado e do compilador – sempre aloca com uso em ponteiros e com `extern`)
    - Possibilita realizar cálculos envolvendo os valores constantes em tempo de compilação (importante na definição de arrays)
    - Pode ser usado com todos os tipos (predefinidos ou definidos pelo usuário)
    - Facilita a manutenção de programas
- ➔ Boa prática: usar `const` ao invés de `#define`

## III.5.1. Constantes

- Outras características
  - Uma constante **tem que ser inicializada** na declaração
    - Pois não pode haver uma atribuição para uma constante ...
  - É visível somente dentro do arquivo onde está definida
    - Se necessário, usar *extern* para obter valor externo e ser reconhecida fora do arquivo
      - `extern const int bufsize; // força alocar memória, pois o valor é externo`
  - É possível inicializar constante com valor produzido em tempo de execução
    - `const char c = cin.get(); const char c2 = c + 'a';`
  - Possibilita gerar código mais eficiente por eliminar (em vários casos) a alocação de memória e a leitura de variáveis constantes
  - Pode ser empregado em vários contextos
    - Objetos, dados e função membros, argumentos de função, retornos de função,
    - ...

## III.5.1. Constantes

```
#include <iostream>
using namespace std;
const int i = 100;           // Typical constant
const int j = i + 10;        // Value from const expr
char buf[j + 10];          // Still a const expression
```

Constantes tratadas em tempo de compilação (não alocam memória)

```
int main() {
    cout << "type a character & CR:";
    const char c = cin.get(); // Can't change
    const char c2 = c + 'a';
    cout << c2; // ...
}
```

Constantes tratadas em tempo de execução (alocam memória)

## III.5.1. Constantes

- Uso de constantes com ponteiros
  - Ponteiro para constante
    - Formas: const tipo\* var;      ou      tipo const\* var;
      - Ex: const int\* u; // u é um ponteiro que aponta para constante inteira
      - *u* pode apontar para qualquer coisa (não é constante), mas o que ele aponta não pode ser alterado
  - Ponteiro constante
    - Forma: tipo var=valor;      tipo\* const ponteiro = &var
      - Ex: int d=1; int\* const p = &d;
        - *p* é um ponteiro constante que aponta para um inteiro
        - É necessário existir um endereço inicial, imutável para o ponteiro
        - O conteúdo pode alterar – ex: \*p = 2 // legal
  - Ponteiro constante para objeto constante
    - Engloba as duas formas acima: neste caso, nem o ponteiro nem o conteúdo do objeto podem ser alterados
      - Ex: int a=1; const int\* const pa1 = &a;    ou    int const\* const pa2=&a

## III.5.1. Constantes

- Ponteiros e constantes: exemplo
  - const pode se aplicar ao endereço para o qual o ponteiro aponta ou se aplicar ao conteúdo apontado

```
//: C08:PointerAssignment.cpp
int d = 1;
const int e = 2;
int* u = &d; // OK -- d not const
//!int* v = &e; // Illegal -- e const
int* w = (int*)&e; // Legal but bad practice
int main() {} //:~
```

const int \*v = &e;

De toda forma, é possível utilizar um cast para forçar a atribuição, mas não é uma boa prática

Não se pode atribuir um endereço de um objeto constante para um ponteiro, pois senão o objeto poderia ser alterado por este ponteiro

## III.5.1. Constantes

- Passagem de parâmetros em C++
  - A passagem de parâmetros *default* é feita por cópia

```
void f(int i) { i = 50; }
```
  - Porém, pode-se passar por referência e altera o valor

```
void g(int& i) { i = 20; }
```
  - Outra forma de modificar uma variável é usar ponteiros

```
void h(int* pti) { *pti = 30; }
```
  - O retorno *default* também é por cópia

```
int f1() { int i = 4; return i; }
```

```
int f1() { int i = 4; return i; }
int main() {
    int j = 10;
    f(j); // j continua com o valor 10
    g(j); // j assume o valor 20
    h(&j); // j assume o valor 30
    j = f1(); // atribui-se a j o valor 4
}
```

## III.5.1. Constantes

- Uso de constantes na passagem de parâmetros e retorno
  - Na passagem de parâmetros constantes por cópia
    - O uso de *const* não causa alteração significativa, uma vez que os argumentos passados não serão alterados pela função
    - É apenas uma promessa de que os valores originais não serão alterados pela função.

```
void f1(const int i) {  
    i++; // Illegal --- compiler-timer error  
}
```

## III.5.1. Constantes

- **Retorno de valor constante**
  - Ao retornar valor, também ocorre cópia → também não se faz necessário para os tipos primitivos
  - Para outros tipos, não podem ser atribuídos a outro objeto ou modificados
- **Na passagem e retorno de endereços (ponteiro ou referências)**
  - Sempre que possível, usar *const* ao passar um endereço para uma função
    - O uso de *const* previne a ocorrência de alteração inadequada
    - Além disso, possibilita usá-la junto a outros objetos que forem *const*
  - A sintaxe por referência, além de mais simples, é mais eficiente

## III.5.1. Constantes

- Passando referências constantes em funções
  - Esta é a forma mais adequada de passar parâmetros para funções, onde se quer **evitar a cópia** e ainda garantir que não haja modificação de valores internos

```
class X
{
    // Um monte de atributos ==> é mais eficiente evitar a cópia!
};

void g ( X x) {}          // Passagem por cópia (ineficiente)
void g1(X& x) {}          // Passagem por referência (OK, pode modificar)
void g2(const X& x) {}    // Passagem por referência constante
                           (OK, não pode modificar)
```

## III.5.1. Constantes

- Constantes em classes

- Podemos ter atributos e funções membro constantes

```
class X {  
    const int i;           // atributo constante  
    public:  
        X(int ii);  
        void f() const; // função membro constante  
};
```

- Também podemos ter objetos constantes

```
const X a;           // objeto constante
```

## III.5.1. Constantes

- Uso de atributo constante dentro de uma classe
  - Aloca espaço específico em cada objeto
  - Representa um valor que é inicializado uma vez e não pode mais ser alterado
    - “Esta constante é para toda a vida do objeto”
  - Cada objeto pode ter um valor diferente para esta constante
    - Ao criar uma constante para uma classe, esta não pode ser inicializada
      - A inicialização ocorre no construtor de cada objeto, em um local específico
        - **Lista de inicialização do construtor**
        - Apresentada apenas na **definição** do construtor
        - Ocorre após a lista de argumentos, precedida do “:” e vem antes do “{“
        - Indica que deve ser executada antes de qualquer código do construtor
        - Criada para uso pela herança (a ser visto posteriormente)
        - Nota: a idéia desta lista foi estendida também para os tipos primitivos, existindo construtores *default* para eles em substituindo à atribuição
          - Ex: float pi (3.14159) é o mesmo que float pi = 3.14159

## III.5.1. Constantes

- Exemplo de atributo constante em uma classe inicializado com a **lista de inicialização do construtor**

```
class X {  
    const int i; // atributo constante  
public:  
    X(int ii);  
        // etc ...  
};  
X::X(int sz) { i = sz; } // Não pode inicializar assim, pois i é constante  
X::X(int sz) : i(sz) {} // OK!
```

- Nota: se quiser utilizar constante em tempo de compilação para uma classe, utilizar **static const tipo nomeVar = valor;**
  - Neste caso, só existirá uma instância para a classe, independentemente do número de objetos criados para a classe

## III.5.1. Constantes

- Exemplo de uso geral da lista de inicialização de construtores

```
class Parte {  
    int t;  
public:  
    Parte(int tt) ;  
};  
Parte::Parte(int tt) : t(tt) {}
```

```
class Todo {  
    const int size;  
    Parte b;  
public:  
    Todo(int sz, int bt);  
};  
Todo::Todo(int sz, int bt) : size(sz), b(bt) {}
```

Esta lista de inicialização pode ser usada antes do corpo da função mesmo sem ter atributo *const*. No caso específico de *const*, ela é obrigatória

## III.5.1. Constantes

- Objetos constantes e funções membro constantes

```
const int i = 1;  
const X b(2); // objeto constante
```

- Como o compilador pode garantir que um objeto do tipo especificado vai permanecer constante?
  - Permitindo que somente funções membro constantes (funções que o compilador garante que não modificam o estado do objeto) sejam chamadas sobre os objetos constantes!

```
class X {  
    int i;  
public:  
    X(int ii);  
    int f() const;  
    int g();  
};  
X::X(int ii) : i(ii) {}  
int X::f() const { return i; }  
int X::g() { i = 20; }
```

```
void h(const X& rx) {  
    rx.f(); // OK, f() é constante  
    rx.g(); // ERRO!!!  
}  
int main() {  
    X x1(10);  
    const X x2(20);  
    x1.f(); //OK  
    x2.f(); //OK  
    x1.g(); //OK  
    x2.g(); // ERRO!!! }
```

## III.5.1. Constantes

- Objetos constantes e funções constante
  - Portanto, **funções membro constante** são aquelas funções para as quais se tem a garantia de se manter constante o objeto sobre o qual as funções são chamadas
    - Elas não modificam os atributos dos objetos!
    - São as únicas que podem ser chamadas por objetos constantes
      - São seguras de serem chamadas por todos os objetos
    - O *const* deve aparecer na declaração e na definição
      - Caso contrário, o compilador considera 2 funções distintas
    - Acusam erro caso a função tente alterar qualquer membro da função ou chamar outra função membro não constante
    - Construtores e destrutores não podem ser funções *const*, pois sempre realizam alteração no objeto

## III.5.1. Constantes

- **Regra importante:** faça com que todas as funções membro que não precisem modificar atributos do objeto sejam constantes
  - Desta forma, elas poderão ser chamadas sobre objetos ou referências constantes!

```
void h(const X& refcons, X& ref) { // contexto do exemplo anterior
    refcons.f();      // OK, f() é const
    ref.f();          // OK, f() pode ser chamada por qualquer objeto
    ref.g();          // OK, nem g() nem ref são const
    refcons.g();     // ERRO - g() não é const!
}
```

- **Volatile:** avisa ao compilador para desconsiderar cuidados com o dado
  - “Este dado pode ser alterado fora do conhecimento deste compilador”
    - Usado para dados provenientes de multitarefa, interrupções, threads
  - A sintaxe e as formas de uso são semelhantes a *const*
    - *const volatile*: não pode ser alterado por um programador cliente, mas pode ser alterado por um agente externo

# Exercício

15. Considerando as seguintes definições, indique quais comandos são ilegais e justifique.

```
int i = 0;
const int j = 1;
int *p1;
const int *p2;
int * const p3 = &i;
```

- a) p1 = &i;
- b) p1 = &j;
- c) p2 = &i;
- d) p2 = &j;
- e) p2 = &i; (\*p2)++;
- f) (\*p3)++;
- g) p3 = &j;

## Exercicio

16. Localize os erros na seguinte classe e explique como corrigi-los:

```
#include <iostream>
using namespace std;

class Example {
    const int data;
    int count;

public:
    Example() {};
    int getCount() const {return count++;}
};
```

## Exercício

17. Faz sentido um construtor ser `const`? E um destrutor? Por quê?
18. Por quê é interessante declarar como `const` todas as funções membro que não modificam atributos de objetos de uma classe?

## III.5.2. Funções *inline*

- Funções *inline*: necessidade
  - Uso de macros
    - Uma das maneiras que a linguagem C (e C++) possui para permitir uma maior eficiência do código
    - Vantagens do uso
      - Permite que ocorra uma simples substituição de código pelo pré-processador, evitando:
        - Ter uma chamada a função, com todo o custo associado
        - Criar os argumentos e copiar seus valores
        - Fazer um CALL em Assembly, retornando o valor e efetuando um RETURN em Assembly
      - Possui a conveniência e a legibilidade de uma chamada de função, porém sem o custo adicional associado

## III.5.2. Funções *inline*: Macro

- Funções *inline*: necessidade
  - Macros: exemplos de uso que geram problemas
    - `#define F(x) (x + 1)`
      - Uso da macro  
`F(1)`
      - Expandido pelo pré-processador  
`(1+1)`
    - Problema 1
      - Desdobramento incorreto em função de erro ao defini-la  
`#define F (x) (x + 1) // note espaço entre F e (x)`  
`F(1) ==> (x) (x + 1) (1)`
    - Problema 2
      - Precedência incorreta de operadores  
`#define CUBO(x) x*x*x`  
`CUBO(a+b) → a+b*a+b*a+b = a + 2.ba + b`

Solução Problema 2:  
uso de parênteses  
`#define CUBO(x)  
(x)*(x)*(x).`  
`CUBO(a+b) →  
(a+b)*(a+b)*(a+b)`

## III.5.2. Funções *inline*: Macros

- Funções *inline*: necessidade
  - Macros: exemplos de uso que geram problemas
    - **Problema 3:** não se pode implementar uma função membro de uma classe como macro!

```
class X {  
    int i;  
public:  
    #define VAL(X)::i // Error
```
    - Por questões de eficiência, haveria a tendência de deixar todos os atributos da classe como *public* !
    - **Problemas adicionais:** macros não permitem variáveis locais, blocos, verificação de tipos, chamadas recursivas, etc.
  - Solução para todos os problemas das macros:  
**funções *inline* !**

## III.5.2. Funções *inline*

- Funções *inline*: conceito

```
inline int par (int num) {return((num%2==0)? num : num+1);}
```

```
inline float cubo (float x) {return x*x*x;}
```

```
inline float dobro (float x) {return 2*x;}
```

- Objetivo

- O especificador ***inline*** dá uma dica para o compilador de que ele deve tentar expandir o código para a função "*inline*" (como a macro fazia), evitando o “*overhead*” associado à chamada à função
  - É apenas uma sugestão, não obriga o compilador a realizar desta forma

- Todo o processo estará sob o controle do compilador

- Não se pode garantir que toda chamada a uma função *inline* vai ser realmente gerada "*inline*"
- Para tornar a geração de código *inline* possível, você deve incluir o corpo da função dentro da declaração

## III.5.2. Funções *inline*

- Funções *inline*: características

- Toda função definida no corpo de uma classe é *inline*

```
class Circulo{  
private:  
    int raio;  
    Ponto centro;  
public:  
    Circulo(Ponto ce, int ra): centro(ce), raio(ra){ }  
    float area( ) const { return PI*raio*raio; }  
};
```

→ Circulo::Circulo() e Circulo::area() são *inline*!

- Uma função não pertencente à classe também pode se tornar *inline*
  - Basta colocar o especificador *inline* no início de sua declaração e incluir também o corpo da função

## III.5.2. Funções *inline*

- Funções *inline*: funcionamento
  - Ao se deparar com uma função *inline*, o compilador coloca na tabela de símbolos (assinatura+ retorno + corpo da função)
  - Ao localizar seu uso, ele verifica se a chamada está correta, se o retorno é compatível, faz ajustes entre tipos (se necessário) e substitui o corpo da função pela sua chamada, eliminando o *overhead* da chamada de função
  - O comportamento lógico é o mesmo, o que melhora é o desempenho
  - O código *inline* ocupa espaço, mas se a função for pequena, ocupa menos espaço que o código gerado para sua chamada
    - Se a função for grande, causará duplicação de código sempre que for chamada, reduzindo o benefício de se ter maior eficiência na chamada
  - Situações nas quais o compilador não consegue aplicar *inline*
    - Caso a função seja muito complicada (ex: com laços)
    - Caso seja necessário usar o endereço da função
      - Neste caso, ele precisa armazenar para obter o endereço

## III.5.2. Funções *inline*

- Funções *inline*: aplicação
  - Se justifica para funções pequenas
    - Nas quais o *overhead* da chamada às funções seja maior que o causado pela duplicação de código
    - Seu objetivo é fornecer oportunidade de otimização ao compilador
  - Um uso típico de funções *inline* em classes são as chamadas **funções de acesso a atributos** (get / set)
    - Mas podem ser utilizadas também em situações mais sofisticadas

```
class Access {  
    int i;  
public:  
    int get() const { return i; }  
    void set(int ii) { i = ii; }  
};
```

## III.5.2. Funções *inline*

- Funções *inline*: aplicação
  - Muitas vezes, se usam funções sobrecarregadas para acessar / modificar os atributos por meio de **accessors e mutators**
    - Accessors: funções que lêem informações do estado do objeto
    - Mutators: funções que alteram o estado do objeto

```
class Rectangle {  
    int wide, high;  
public:  
    Rectangle(int w = 0, int h = 0) : wide(w), high(h) {}  
    int width() const { return wide; }           // width for read / get  
    void width(int w) { wide = w; }             // width for set  
    int height() const { return high; }          // height for read / get  
    void height(int h) { high = h; }             // height for set  
};  
int main() {  
    Rectangle r(19, 47);  
    // Change width & height  
    r.height(2 * r.width()); // height for set, width for read  
    r.width(2 * r.height()); // width for set, height for read  
}
```

## III.5.2. Funções *inline*

- Funções *inline*: aplicação
  - O uso de funções *inline* na definição da classe faz com que sua interface fique “poluída”
  - Apresenta detalhes de implementação que deveriam estar sendo escondidos do usuário da classe
    - Para evitar isto e ainda poder utilizar funções *inline*, pode-se definir funções membro *inline* fora da definição da classe

## III.5.2. Funções *inline*

```
class Rectangle {  
    int wide, heigh;  
public:  
    Rectangle(int w = 0, int h = 0);  
    int width() const;  
    void width(int w);  
    int height() const;  
    void height(int h);  
};  
  
inline Rectangle::Rectangle(int w, int h) : width(w), height(h) {}  
inline int Rectangle::width() const { return width; }  
inline void Rectangle::width(int w) { width = w; }  
inline int Rectangle::height() const { return height; }  
inline void Rectangle::height(int h) { height = h; }
```

→ Para que a substituição de código possa ser efetuada pelo compilador, a definição das funções também deve ser feita no arquivo .h !

# Exercício

19. Modifique o programa abaixo de forma que a macro BAND() funcione de forma correta.

```
#include <iostream>
using namespace std;

#define BAND(x) (( (x)>5 && (x)<10) ? (x) : 0)

int main() {
    for(int i = 4; i < 11; i++) {
        int a = i;
        cout << "a = " << a;
        cout << "\t BAND(++a)=" << BAND(++a);
        cout << "\t a = " << a << endl;
    }
}
```

# Exercício

20. No programa abaixo, troque todas as funções membro para funções inline. Troque também a função **initialize()** para o construtor.

```
#include <iostream>
#include <cstring> // memset()
using namespace std;
const int sz = 20;

class Holder {
public:
    int a[sz];
    void initialize();
};

class Pointer {
    Holder *h;
    int *p;
public:
    void initialize(Holder* h);
    // Move around in the array:
    void next();
    void previous();
    void top();
    void end();
    int read();
    void set(int i);
};
```

```
void Holder::initialize() {
    memset(a, 0, sz * sizeof(int));
}

void Pointer::initialize(Holder *rv) {
    h = rv;
    p = rv->a;
}

void Pointer::next() {
    if(p < &(h->a[sz - 1])) p++;
}

void Pointer::previous() {
    if(p > &(h->a[0])) p--;
}

void Pointer::top() {
    p = &(h->a[0]);
}

void Pointer::end() {
    p = &(h->a[sz - 1]);
}
```

```
int Pointer::read() {
    return (*p);
}

void Pointer::set(int i) {
    (*p) = i;
}

int main() {
    Holder h;
    Pointer hp, hp2;
    int i;

    h.initialize();
    hp.initialize(&h);
    hp2.initialize(&h);
    for(i = 0; i < sz; i++) {
        hp.set(i);
        hp.next();
    }
    hp.top();
    hp2.end();
    for(i = 0; i < sz; i++) {
        cout << "hp = " << hp.read()
            << ", hp2 = " << hp2.read() << endl;
        hp.next();
        hp2.previous();
    }
}
```

### III.5.3. Controle de visibilidade de nomes

- Manipular nomes é uma atividade fundamental na programação
  - Quando o projeto fica grande, o volume de nomes a manipular acaba ficando muito grande
    - Necessário ter estratégias para gerenciá-los
    - *static*: “algo que mantém sua posição” - visto sob 2 aspectos:
      - Localização física na memória (memória estática)
        - Alocado uma única vez em um endereço físico
          - Área especial para dados estáticos (não usa a pilha de execução)
        - Visibilidade proporcionada ao nome
          - Controlada pelo escopo
            - Diversas formas: *namespace*, classe, objeto, arquivo fonte, ...
      - Vantagem sobre nomes globais: acesso + restrito, melhor controle
        - Facilita localizar erros, não alterada fora do escopo definido e sem sobreposição de nomes

### III.5.3. Controle de visibilidade de nomes

- Variáveis locais *static*

```
int count ( )  
{  
    static int num = 0; // inicializada uma única vez  
    int x = 0;          // inicializada várias vezes  
    num++;  
    ...  
    return 0;  
}
```

- A inicialização de **num** é feita somente na *primeira vez* que a função é chamada
  - Na realidade, na primeira vez que o programa passa pela declaração da variável
  - **num** mantém o seu valor de uma chamada para a outra
    - Isto significa que a variável *static* não é criada na pilha, mas sim na área de variáveis estáticas do programa

### III.5.3. Controle de visibilidade de nomes

- Objetos *static*
  - O construtor de um objeto *static* é chamado apenas 1 vez
    - No momento que a *thread* de execução executa pela primeira vez o código da função
    - Se a função não for executada nenhuma vez, não é construído
  - O destrutor de um objeto *static* somente é executado ao final do programa
  - Objeto *static* x global
    - O construtor de um objeto global é executado antes do main() começar sua execução (é diferente do que ocorre com o *static*)
    - O destrutor de um objeto global é executado somente ao final da execução do programa (é semelhante ao que ocorre com o *static*, caso tenha sido construído)

### III.5.3. Controle de visibilidade de nomes

```
#include <iostream>
using namespace std;
class Obj {
    char c; // Identifier
public:
    Obj(char cc) : c(cc) {
        cout << "Obj::Obj() for " << c << endl;
    }
    ~Obj() {
        cout << "Obj::~Obj() for " << c << endl;
    }
};
Obj a('a'); // Global (static storage)
// Constructor & destructor always called
void f() {
    static Obj b('b');
}
void g() {
    static Obj c('c');
}
int main() {
    cout << "inside main()" << endl;
    f(); // Calls static constructor for b
    g();
    cout << "leaving main()" << endl;
}
```

Obj a é global: construtor chamado antes de main() e o destrutor chamado ao final do programa

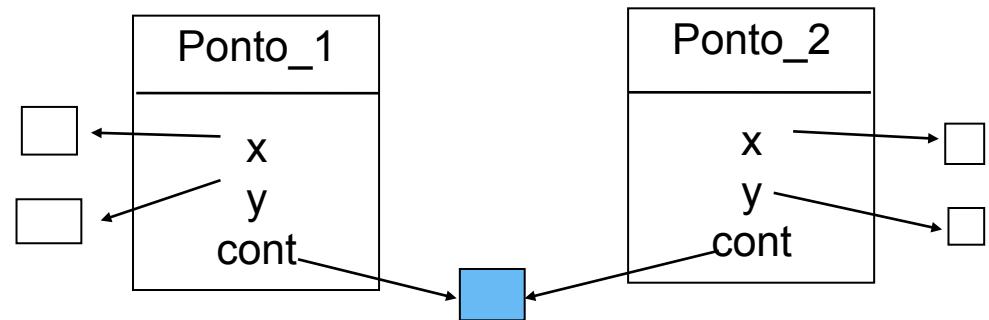
Obj b é static: construtor chamado ao chamar a função pela primeira vez e o destrutor ao final do programa

```
Obj::Obj() for a
inside main()
Obj::Obj() for b
Obj::Obj() for c
leaving main()
Obj::~Obj() for c
Obj::~Obj() for b
Obj::~Obj() for a
```

### III.5.3. Controle de visibilidade de nomes

- Atributos de classe: atributos *static*
  - *Atributos estáticos servem para implementar o conceito de “Atributo de uma Classe”*
  - Usado quando deseja-se que *todos* os objetos de uma determinada classe compartilhem de um certo dado
    - Único para toda a classe, independente do número de objetos
    - Todos os objetos compartilham e podem se comunicar por ele

```
class Ponto{  
    int x, y;  
    static int cont;  
    // ....  
};  
Ponto Ponto_1, Ponto_2;
```



### III.5.3. Controle de visibilidade de nomes

- Atributos de classe: atributos *static*
  - Pode ter escopo público, privado ou protegido
  - Pode ser acessado por funções membro ou por outras partes do programa, quando público
  - Pode ser acessado antes da existência de qualquer objeto da classe
    - Seu espaço de memória é reservado antes da criação de qualquer objeto
    - Sua definição tem que ocorrer fora da classe (sem *inline*) e só uma vez

```
class Ponto{  
    int x, y;  
    static int cont;  
    // ....  
};
```

int Ponto :: cont = 7;

### III.5.3. Controle de visibilidade de nomes

- Atributos *static* constantes
  - Todo membro estático deve ser redeclarado fora da classe, porém dentro do escopo do arquivo da classe
  - Neste ponto ele pode ser inicializado. Não se pode inicializar um membro estático não constante dentro da definição da classe!

```
class Ponto{  
    int x, y;  
    static int cont;  
    // ....  
};  
int Ponto :: cont = 7;
```

- Se o atributo *static* for também constante, pode ser inicializado na própria classe

```
class Buffer{  
    static const int bufsize = 256;  
    char buff[bufsize];  
    // ....  
};
```

### III.5.3. Controle de visibilidade de nomes

- Funções membro *static*
  - Tipo de função membro especial cujas principais características são:
    - Trabalha para a classe como um todo e não para cada objeto
      - Evita ter que criar uma função global (vista e acessível fora da classe)
    - Pode ser chamada sem estar associada a um objeto da classe
      - Basta utilizar a qualificação de escopo da classe - Classe::f()
    - Não pode manipular membros não estáticos da classe
      - Foram criadas para manipular os membros estáticos da classe (dados e outras funções)
      - Não possui o ponteiro *this* (por isso não acessam membros comuns)
  - ➔ Nota: funções membro não estáticas podem acessar dados e funções estáticas

### III.5.3. Controle de visibilidade de nomes

- Funções membro *static*: exemplo

```
#include <iostream>
using namespace std;

class Ponto{
    int x, y;
    static int cont; // dado estático
public:
    Ponto() { cont++; } // soma ocorrências
    ~Ponto();
    static void mostra() { // função estática
        cout << cont << " "; // só dado estático
    }
};
int Ponto::cont=0; // inicialização externa
Ponto::~Ponto () {
    cout << "Destruindo Ponto " << cont << endl;
    cont--;
}
// subtrai ocorrências
int main() {
    Ponto::mostra(); // 0 (sem objeto declarado)
    Ponto p1; // p1 criado (cont = 1)
    p1.mostra(); // 1
    {
        Ponto p2; // p2 criado (cont = 2)
        p2.mostra(); // 2
    } // p2 destruído (cont = 1)
    Ponto p2; // novo p2 criado (cont = 2)
    p2.mostra(); // 2
```

```
0 1 2 Destruindo Ponto 2
2 Destruindo Ponto 2
Destruindo Ponto 1
```

### III.5.3. Controle de visibilidade de nomes

- **Namespaces: necessidade**
  - Em projetos grandes, a falta de controle sobre o espaço (escopo) de nomes pode ser um problema
    - Os nomes mais comuns já foram usados, sendo necessário colocar nomes grandes e/ou complicados para garantir que sejam diferentes
    - Podemos usar `typedef` para tentar simplificar, mas não seria uma solução elegante e suportada em todos os ambientes
  - **Solução em C++: usar *namespaces***
    - Subdividir o espaço de nomes globais em partes gerenciáveis,
      - Coloca o nome dos membros do *namespace* em um espaço específico e separado (semelhante a um encapsulamento)

### III.5.3. Controle de visibilidade de nomes

- Namespaces: aplicação
  - Mecanismo do C++ para agrupar *logicamente* nomes
    - Se quaisquer declarações (variáveis, funções, classes, ...) forem relacionadas entre si, elas poderão ser colocadas em um mesmo espaço de nomes para expressar este fato
  - Existe um outro mecanismo de *agrupamento físico* que é o uso de arquivos
    - Os namespaces vêm adicionar a este mecanismo o *agrupamento lógico*

### III.5.3. Controle de visibilidade de nomes

- *Namespaces*: sintaxe

- A sintaxe de criação de um *namespace* parece com a da classe

```
namespace MyLib {  
    // Declarations  
}  
int main() {}
```

- Isto produz um novo *namespace* contendo as declarações que forem feitas no espaço apropriado
    - Existem algumas diferenças em relação às definições de classes:
      - A definição de um *namespace* pode aparecer apenas no escopo global (ou aninhado a outro *namespace*)
      - Não é necessário usar um ; após a definição do *namespace*
      - A definição de um *namespace* pode ser continuada em múltiplos arquivos de cabeçalho (vai sendo complementada)

### III.5.3. Controle de visibilidade de nomes

- Namespace: exemplo

```
//Header1.h
#ifndef HEADER1_H
#define HEADER1_H
namespace MyLib {
    void f();
    // ...
}
#endif // HEADER1_H

//Header2.h
#ifndef HEADER2_H
#define HEADER2_H
#include "Header1.h"
// Add more names to MyLib
namespace MyLib { // NOT a redefinition!
    void g();
    // ... MyLib agora contém f() e g()
}
#endif // HEADER2_H
```

```
//Continuation.cpp
#include "Header2.h"
int main() {

    MyLib::f();
    MyLib::g();

}
```

### III.5.3. Controle de visibilidade de nomes

- **Namespaces:** características
  - Um *namespace* é um escopo com nome
  - Quanto maior o programa, mais úteis são os *namespaces* para expressar a separação lógica de suas partes
  - Idealmente toda entidade em um programa deve estar em algum *namespace*, para indicar o seu papel lógico no programa
    - A exceção é `main()`, que deve ser global
- **Namespaces:** usos
  - Seu uso pode ocorrer basicamente de 3 formas:
    - Pela qualificação explícita (resolução de escopo)
    - Pela diretiva *using*
    - Pela declaração *using*
  - Estas formas serão detalhadas a seguir

### III.5.3. Controle de visibilidade de nomes

- *Namespaces*: uso pela qualificação explícita
  - Explicita o escopo dos objetos que estão em um *namespace*

```
namespace X {  
    class Y {  
        public:  
            void f();  
    };  
    class Z;  
    void func();  
}  
class X::Z { // namespace para classe  
    int u, v, w;  
public:  
    Z(int i);  
    int g();  
};  
// namespace para funções da classe  
X::Z::Z(int i) { u = v = w = i; }  
int X::Z::g() { return u = v = w = 0; }
```

```
// namespace para função  
void X::func() {  
    X::Z a(1); // para objeto  
    a.g();  
}  
int main(){  
    X::Z z1(2), z2(4);  
    X::Y y1, y2;  
    y1.f();  
    z1.g();  
    X::func();  
}
```

### III.5.3. Controle de visibilidade de nomes

- *Namespaces*: uso com a **diretiva using**
  - Torna todos os nomes de um *namespace* disponíveis em um determinado contexto (“importa” todo o *namespace* de uma vez)

```
namespace Int {  
    enum sign { positive, negative };  
    class Integer {  
        int i;  
        sign s;  
        public:  
            Integer(int ii = 0)  
                : i(ii),  
                  s(i >= 0 ? positive : negative)  
            {}  
            sign getSign() const { return s; }  
            void setSign(sign sgn) { s = sgn; }  
            // ...  
    };  
}
```

```
namespace Math {  
    using namespace Int;  
    Integer a, b;  
    Integer divide(Integer, Integer);  
    // ...  
}
```

- Já utilizamos isto: **using namespace std;**

### III.5.3. Controle de visibilidade de nomes

- *Namespaces*: uso com a **declaração using**
  - Possibilita não deixar disponíveis todos os nomes presentes em um *namespace*, mas apenas alguns
  - Introduz um sinônimo local para o nome

```
namespace U {  
    inline void f() {}  
    inline void g() {}  
}  
namespace V {  
    inline void f() {}  
    inline void g() {}  
}  
void h() {  
    using namespace U; // diretiva using (pode utilizar tudo de U)  
    using V::f;         // declaração using (específica para este caso)  
    g();              // chama U::g(); (que é o namespace padrão)  
    f();              // chama V::f(); (V é específico p/ a função f(), cf acima)  
    U::f();           // para chamar a f() de U, precisa explicitar o escopo  
}
```

### III.5.3. Controle de visibilidade de nomes

- **Namespaces:** cuidados
  - O objetivo dos *namespaces* é expressar uma estrutura lógica
    - A forma mais simples de estrutura para a qual ele pode ser usado é a distinção entre o código escrito por uma pessoa e o escrito por outra
    - Quando usamos um único espaço de nomes global, torna-se desnecessariamente difícil compor um programa composto por partes separadas
  - O problema é que cada parte pode definir os mesmos nomes
    - Quando combinados em um mesmo programa, estes nomes colidem ... veja exemplo a seguir

# Namespaces: cuidados

```
// meu.h
char f(char);
int f(int);
class String { /* ... */ };
```

```
// seu.h
char f(char);
double f(double);
class String { /* ... */ };
```

- Fica impossível para alguém usar juntos meu.h e seu.h sem modificações
  - f(): possuem mesmo nome e assinatura
  - Ambos possuem a classe String

- Solução: separar os espaços de nomes

```
namespace Meu {
    char f(char);
    int f(int);
    class String { /* ... */ };
}

namespace Seu {
    char f(char);
    double f(double);
    class String { /* ... */ };
}
```

- Utiliza-se *Meu* e *Seu* por *qualificação explícita*, *declaração* ou *diretiva using*

```
char a, c;
using Seu::String;
String d;
Meu::f(c);  Seu::f(a);
```

```
// declaração using (Seu é o escopo de String)
// qualificação explícita para cada ocorrência
```

### III.5.3. Controle de visibilidade de nomes

- Namespaces podem ser compostos, para gerar novos Namespaces

```
namespace String_dele {  
    class String{ /* ... */};  
}  
namespace Vetor_dela {  
    class Vector {/* ... */};  
}  
namespace Minha_lib {  
    using namespace String_dele;           // diretiva using  
    using namespace Vetor_dela;           // diretiva using  
    void minha_funcao(Vector&);          // declaração de função  
}  
void f() {  
    Minha_lib::String s = "Pixinguinha";  // qualificação explícita  
}  
using namespace Minha_lib;               // diretiva using: a partir daqui,  
void g(Vector &vs) {  
    // ...  
    minha_funcao(vs);                  // não precisa mais explicitar  
}
```

### III.5.3. Controle de visibilidade de nomes

- Namespaces: regras importantes
  - Não utilize **diretivas using** em arquivos .h
    - Utilize apenas nos .cpp, pois o seu alcance fica mais limitado
      - Uso livre de um namespace dentro de um arquivo, sem afetar outros
      - Cada .cpp pode ter o seu namespace, distinto dos demais
      - Se houver problemas de nomes duplicados quando estiver usando mais de uma **diretiva using** nos .cpp, efetue a seleção deles atribuindo **declarações using**
    - Nos .h, ou utilize as **qualificações explícitas** para acessar determinados nomes encapsulados em namespaces, ou utilize **declarações using**, para introduzir apenas os nomes selecionados no escopo

# Exercício

21. Informe o que será impresso pelos códigos abaixo:

a)

```
#include <iostream>
using namespace std;

class X {
    int i;
public:
    X(int ii = 0) : i(ii) {cout<<"Construtor i = "<<i<<endl;}
    ~X() { cout << "Destruitor i = "<< i << endl; }
};

void f() {
    static X x1(47);
    static X x2;
}

int main() {
    f();
    cout<<"fim main"<<endl;
}
```

```
Construtor i = 47
Construtor i = 0
fim main
Destruitor i = 0
Destruitor i = 47
```

b)

```
#include <iostream>
using namespace std;

class X {
    int i;
public:
    X(int ii = 0) : i(ii) {cout<<"Construtor i = "<<i<<endl;}
    ~X() { cout << "Destruitor i = "<< i << endl; }
};

void f() {
    X x1(47);
    X x2;
}

int main() {
    f();
    cout<<"fim main"<<endl;
}
```

```
Construtor i = 47
Construtor i = 0
Destruitor i = 0
Destruitor i = 47
fim main
```

## Exercício

22. Crie uma função que retorna o próximo valor em um sequência de Fibonacci toda vez que ela é chamada. Insira um parâmetro que é um `bool` com o valor default igual a `false` tal que quando você passa o argumento com valor `true` ele começa do início da sequência de Fibonacci. Crie a função `main` para utilizar essa função.