UNIVERSIDADE FEDERAL DE SANTA CATARINA - CAMPUS JOINVILLE

CENTRO DE ENGENHARIAS DA MOBILIDADE

Introdução a Programação de C++

Mariana Cândido Perassa

NOVEMBRO / 2016

Sumário

Sumário

1. Introdução 1

1.1 Compilando o primeiro programa

2. Dados

2.1. Bibliotecas

2.2. Entrada e saída

2.3. Variáveis

2.3.1. Tipos

2.3.2. Atribuição de valores

2.4. Operadores

3. Vetores e matrizes

3.1. Declaração do vetor

3.2. Atribuição de valores ao vetor

3.3. Declaração da matriz

3.4. Atribuição de valores à matriz

4. Laços e comandos

4.1. Comandos condicionais

4.2. Laços de repetição

5. Funções

5.1. Definição

5.2. Declarando uma Função

5.3. Chamando uma Função

5.4. Passagem por valor

5.5. Passagem por referência

6. Ponteiros

6.1. Declaração de ponteiros

7. Alocação de dados

7.1. Alocação estática
7.2. Alocação dinâmica

8. Arquivos

9. Práticas de programação

10. Exercícios de Revisão

11. Referências Bibliográficas

1. Introdução

Esta apostila foi desenvolvida como um projeto de ensino do Programa de Educação Tutorial do Centro de Engenharias da Mobilidade (PET-CEM). O presente trabalho apresenta uma introdução a programação de computadores utilizando a linguagem C++, contendo os principais conceitos e exemplos de aplicações.

1.1 Compilando o Primeiro Programa

Iniciaremos o estudo com um pequeno exemplo. O programa apenas escreve “Alo, mundo” no console, como todo bom primeiro programa em uma linguagem de programação. A seguir, cada parte do programa será vista em detalhe.

```
// Alo mundo, um programa elementar

#include <iostream>
using namespace std;

int main (){

cout << "Alo, mundo\n";
return 0;
}

// Fim de main ()
```

Pode se observar que a primeira linha é precedida por //. Essa é a sintaxe do comentário de uma linha do C++. O compilador deve ignorar tudo aquilo que estiver à direita de duas barras, mesmo que não no começo. Comentários iniciados por /* e terminados por */ também são utilizados principalmente para fazer comentários de várias linhas. Já o comando “\n” funciona como um escape, mudando o cursor para uma linha abaixo.

A diretiva #include é utilizada para incluir definições de dados e código que serão utilizados por nosso programa, mas já foram compilados e estão disponíveis em uma biblioteca padrão.

O namespace é um recurso utilizado para evitar que, quando são construídos grandes programas, nomes de variáveis, classes e funções conflitem.

O std é uma proposição geral de funções capaz de armazenar, copiar e invocar qualquer função, expressão, objeto ou ponteiro.

Dados podem ser impressos enviando-os para a saída padrão, cout. Da mesma forma, dados podem ser lidos através da entrada padrão, cin. Também nesse caso o tipo de dado lido depende da variável para a qual se está fazendo a leitura.

Em C++, todo bloco de comandos deve estar entre chaves (“{ }”). Segundo as regras de hierarquia de escopo, qualquer variável declarada dentro de um bloco é visível apenas dentro do mesmo (inclua-se qualquer sub-bloco), sendo destruída assim que o bloco é finalizado. Essas são chamadas variáveis locais, em contraponto às variáveis declaradas fora de qualquer bloco, chamadas variáveis globais.

Todas as declarações e comandos da linguagem devem ser terminados por ‘*;*’. Esse sinal não serve apenas como separador nas declarações, mas serve para identificar composição de seqüência entre os comandos, isto é, primeiro é executado um e depois o outro. Utilize vários comandos de impressão para gerar uma saída mais longa.

2. Dados

2.1. Bibliotecas

A biblioteca padrão é uma coleção de classes, funções e variáveis escritas na própria linguagem para facilitar o desenvolvimento de aplicações. Dentre as diversas bibliotecas existentes, no curso abordaremos as seguintes:

<iostream> - Manipulação de fluxo de dados padrão do sistema (entrada padrão e saída padrão)

<cstring> - Manipulação de cadeia de caracteres, especializada para o tipo de dado char (o qual será abordado posteriormente)

<time.h> - Processamento de horas e datas

<cstdlib> - Responsável pelas conversões de números, alocações na memórias e outras funções. Com ela podemos converter um "char" em um "double", criar um número randômico - rand( ) -, alocar na memória, realocar na memória, desalocar da memória, execução de comandos do sistema operacional respectivamente.

<cmath> - Tratamento de funções matemáticas

2.2. Entrada e Saída

A entrada e saída de dados é feita através das funções cin e cout, disponíveis na sua biblioteca padrão.
O operador << permite inserir valores em um fluxo de saída, enquanto o operador >> permite extrair valores de um fluxo de entrada. Segue modelo de entrada e saída de dados:

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

int main (){
char nome[100]; // Declarando variável de caracteres (a qual será citada no tópico seguinte)
int a; // Declarando variável de inteiros (a qual será citada no tópico seguinte)

cout << "Qual o seu nome? "; // Utilizando comando de saída
cin >> nome; // Utilizando comando de entrada

cout << "Quantos anos voce tem? ";
cin >> a;

return 0;
}
```

Também é possível utilizar o comando de saída para retornar ao usuário o resultado de uma operação, por exemplo:

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

int main (){
int a,b;

cout << "Entre com um numero qualquer: ";
cin >> a;

cout << "Entre com o numero que deseja somar ao anterior: ";
cin >> b;

cout << "A soma dos numeros eh: " << a+b <<endl; // "endl" possui função similar a "\n"

return 0;
}
```

2.3. Variáveis

2.3.1. Tipos

As variáveis podem ser de cinco tipos primários distintos:

int: Valores inteiros;

char: Caracteres;

float: Valores em ponto flutuante (reais);

double: Valores em ponto flutuante, com precisão dupla;

bool: Valor false (0) ou true (1).

2.3.2. Atribuição de valores

Para criar uma variável, precisamos declarar o seu tipo, seguido pelo nome da variável e por um caractere de ponto e vírgula: “ int larg; ”.

Para atribuir um valor a uma variável, usamos o operador de atribuição “ = ” (vide tópico seguinte): “ larg = 10; “.

Opcionalmente, podemos combinar esses dois passos, declarando e inicializando a variável em uma só linha: “ int larg = 7; “.

Podemos também definir mais de uma variável em uma só linha. Podemos ainda misturar declarações simples com inicializações.

No caso de vetores, a atribuição de valores deve ser feita entre chaves: “ v [4] = {1,2,3,4,5}“, sendo v[0]=1 a primeira posição.

2.4. Operações

Após criarmos as variáveis, podemos efetuar operações com as mesmas. Segue listagem de operadores e atribuições:

Aritiméticos:	Atribuições:	Comparação

+	Soma
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão

++	Incremento
--	Decremento

Lógicos	
&&	E
||	OU

==	Igual
!=	Diferente
>	Maior que
<	Menor que
>=	Maior ou igual que
<=	Menor ou igual que

Exercício: Crie um programa que leia 3 valores a partir do teclado, e retorne ao console a média aritimética dos mesmos.

Resolução:

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

int main (){
int a, b, c;

cout << "Entre com o valor 1: ";
cin >> a;

cout << "Entre com o valor 2: ";
cin >> b;

cout << "Entre com o valor 3: ";
cin >> c;

cout << " Media: " << (float)(a+b+c)/3 << endl;


return 0;
}
```

O programa acima retornará ao console um resultado float do cálculo efetuado, devido a não restrição dos valores de a,b e c (sendo assim, a divisão da soma pode resultar em um número com vírgula).

3. Vetores e matrizes

Vetores e matrizes, também conhecidos como *arrays*, são estruturas de dados que armazenam coleções de elementos de mesmo tipo tendo cada elemento identificado por um ou mais índices. Vetores são *arrays* unidimensionais, ou seja, só tem uma dimensão; já as matrizes são bidimensionais, ou seja, tem duas dimensões. Além de vetores e matrizes existem outros tipos de *arrays* com diferentes dimensões e são conhecidos como multidimensionais.

3.1. Declaração do vetor

Vetores precisam ser declaradas como quaisquer outras variáveis, para que o compilador conheça o tipo de seus elementos e reserve espaço de memória suficiente para armazená-las.

O que diferencia a declaração de um vetor da de qualquer outra variável é a parte que acompanha o seu nome, isso é, o par de colchetes que envolve um número inteiro, indicando o tamanho da matriz. Declara-se um vetor da seguinte forma:

Tipo nome [tamanho];
Exemplo:

```
int vetor[5];
```

3.3. Declaração da matriz

Matrizes são declaradas da mesma forma que vetores, diferenciando-se apenas pelo acréscimo de um par de colchetes indicando o tamanho de sua segunda dimensão. Declara-se uma matriz da seguinte forma:

Tipo nome [tamanho 1][tamanho 2];
Exemplo:

```
int matriz[5][3];
```

Arrays multidimensionais seguem a mesma lógica de declaração:

Tipo nome [tamanho 1][tamanho 2]...[tamanho n];

3.4. Atribuição de valores

Os elementos de um *array* são sempre numerados por índices iniciados por zero. Por exemplo:

x[1][4] = 5; atribui o valor 5 ao espaço identificado pelo índice 1 (linha 2) e índice 4 (coluna 5).

Além disso, você pode especificar o tipo de elementos que compõem o *array*. O exemplo a seguir faz a demonstração prática dos passos citados até o presente momento:

```
#include <iostream>

using namespace std;

int main(){
    int m[5][3];

    for(int i=0; i<5; i++){
        for(int j=0; j<2; j++){
            cout << (m[i][j] = i+j) << " ";
        }
        cout << endl;
    }

    return 0;
}
```

4. Comandos

4.1. Comandos condicionais

- Comando If:

O comando If permite que executemos algo somente se a sua expressão de teste for verdadeira, caso contrário nada é executado.

Suponhamos que você queira executar um comando qualquer se a expressão de teste for verdadeira e outro se for falsa. Nesse caso utiliza-se o comando If-Else:

```
if ( condição ) {
    comando;
    comando;
} else {
    comando;
    comando;
 }
```

- Comando Switch:

Permite selecionar uma entre várias ações alternativas. Segue exemplo:

```
switch ( x ) {
   case 1:
      escreve(x);
     break;
  case 2:
      registra(x);
     break; //  O comando break, provoca uma saída imediata de um laço ou de um switch
  default:
      comando;
```

```
}
```

4.2. Laços de repetição

- For:

Utilizado quando queremos repetir algo por um número fixo de vezes. Isso significa que utilizamos um laço For quando sabemos de antemão o número de vezes a repetir.

O exemplo a seguir imprime os números de 0 a 19:

```
for (int i = 0; i < 20; i++)  // Se for de interesse do programador imprimir os números de 0 a 20 por
{                                exemplo, pode-se alterar o sinal "<" para "<=".
   cout << i << endl;
}
```

- While:

Utilizamos o laço While, quando este pode ser terminado inesperadamente, por condições desenvolvidas dentro do corpo do laço.

```
while ( condição ) {
    comando 1;
    comando …;
}
```

- Do-While:

Utilizamos do-while quando queremos executar um comando enquanto uma condição for verdadeira. Este comando diferencia-se do while por executar os comandos ao menos uma vez até que a condição seja falsa ao passo que o while só executa o bloco de comando a partir do momento em que a condição se mostra verdadeira.

```
do {
    comando;
} while ( condição );
```

5. Funções

5.1. Definição

Funções são usadas para criar pequenos pedaços de códigos separados do programa principal. Exceto a função MAIN, todas as outras funções são secundárias, o que significa que elas podem existir ou não. Estas são importantes pois retornam valores, ajudam a fragmentar o código em partes menores - mais fáceis de lidar - e ainda por cima podem ser utilizadas mais de uma vez no mesmo programa, poupando minutos de programação e inúmeras linhas de código.

A linguagem C++ já possui funções pré definidas. Temos como exemplo as funções contidas na biblioteca matemática (cmath). São elas:

Função	Comentário
ceil(x)	Arredonda um número real para cima.
cos(x)	Calcula o cosseno de x (em radianos).
exp(x)	Obtém o logaritmo natural *e* elevado à potência x.
fabs(x)	Obtém o valor absoluto de x.
floor(x)	Arredonda um número real para baixo.
log(x)	Obtém o logaritmo natural de x.
log10(x)	Obtém o logaritmo de base 10 de x.
pow(x,y)	Calcula a potência de x elevado a y.
sen(x)	Calcula o seno de x (em radianos)
sqrt(x)	Calcula a raiz quadrada de x.

Além disso, podemos definir funções específicas e chamá-las da mesma maneira. Uma declaração de função, é também chamada de protótipo da função. Nessa declaração devem constar o tipo da função, seu nome e o tipo de cada um de seus parâmetros:

Tipo-valor-retorno nome-da-função (lista de parâmetros) {

instruções

return valor de retorno se houver;

}

5.2. Declarando uma função

Para que seu programa possa compilar sem erros, é necessário que a função sempre esteja acima ou tenha seu protótipo declarado dentro da int main.

```
int mult (int a, int b, int c){ // Tipo da funçao: int ; Nome da Função: mult ; Parâmetros: valores inteiros a,b,c
    return a*b*c; // Return = Tipo de retorno
}
```

Para funções que retornam valores, os valores retornados devem obedecer as regras de variáveis, retornando:

- int: para valores inteiros.
- char: para caracteres individuais.
- float: para valores reais.
- double: para reais com precisão.
- bool: para verdadeiro ou falso.
- void: para simplesmente não retornar nada.

Isto é, se você colocar int no tipo da função, ela deverá obrigatoriamente retornar um valor inteiro, da mesma maneira que um valor não pode ser retornado de uma função void.

5.3. Chamando uma função

Para chamar uma função, basta ir até a INT MAIN e acrescentar o código de acordo com sua função. Utilizando o exemplo do tópico acima, temos:

```
#include <iostream>
#include <cstdlib>

using namespace std;

int mult (int a, int b, int c){
	return a*b*c;
}

int main (){
	int valor1, valor2, valor3, resultado;

	cout << "Digite 3 valores que serao multiplicados: ";
	cin >> valor1;
	cin >> valor2;
	cin >> valor3;

	resultado = mult(valor1, valor2, valor3);

	cout << "O resultado da multiplicacao eh: " << resultado << endl;
	return 0;
}
```

Uma função é dita recursiva se, em seu corpo, estiver presente uma instrução de chamada a ela própria:

```
void imprimeVetor(int v[], int t, int i)
{
      if (i == (t-1))
```

```
    {
       cout << v[i] << " ";
    }
    else
    {
       cout << v[i] << " ";
       return imprimeVetor(v, t,(i+1) );     // Repare que a funçao utiliza dela mesma para
    }                                           obtenção do resultado desejado
}
```

5.4. Passagem por valor

É feita uma cópia do valor dos argumentos, que são usados localmente na função. Modificações feitas dentro da função não altera seu valor fora.

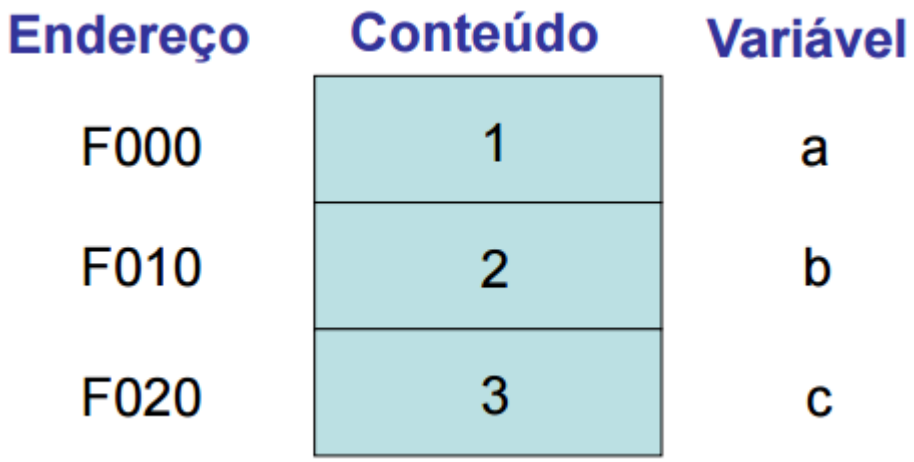
Em C++ todas as chamadas são feitas por valor, a não ser que parâmetros sejam explicitamente passados por referência. Uma exceção são arrays, que são automaticamente passados por referência.

```
void troca (int a, int b) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

No exemplo acima, a função recebe uma cópia e as alterações são efetuadas. Ao final da função, cópias são destruídas e alterações perdidas.

5.5. Passagem por referência

É passada uma referência à variável de parâmetro (seu endereço). As modificações feitas dentro da função, alteram o valor original fora da mesma utiliza-se o caractere “e comercial” (&), que indica que aquele parâmetro é uma referência para a variável. Para facilitar o entendimento, vamos analisar o esquema a seguir:



Neste caso, sabemos que:
&a = F000 (endereço de a);
&b = F010 (endereço de b);
&c = F020 (endereço de c);
a = 1, b = 2, c = 3 (valores das variáveis)

Na prática, alterando o programa do tópico anterior, teríamos:

```
void troca (int &a, int &b) {
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

Onde &a e &b são endereços que contém o conteúdo tmp.

6. Ponteiros

Os ponteiros são variáveis que apontam para endereços na memória. Portanto para entender os ponteiros, precisamos entender como funciona a memória do computador. A memória é dividida em locais de memória, que são numeradas seqüencialmente. Cada variável fica em um local único da memória, conhecido como endereço de memória.

6.1. Declaração de ponteiros

Declaramos um ponteiro utilizando “*” e, como as demais variáveis, eles possuem tipos:

- int *a; // Ponteiro para inteiro
- double *x/ // Ponteiro para double

No primeiro exemplo, caso o ponteiro “a” não possuir conteúdo (vazio), devemos declará-lo NULL:

- a = NULL;

7. Alocação de dados

Quando guardamos um dado, na programação, denominamos o processo de “alocação”. Existem dois tipos de alocação: estática e dinâmica.

7.1. Alocação estática

Este é o tipo de alocação que vimos até então. Na alocação estática definimos um tamanho de memória e alocamos uma variável ou array antes da execução do programa. Vejamos um exemplo de alocação estática:

```
int vetor[10];
```

Aqui temos a dimensão do vetor predefinida e a quantidade de memória utilizada para armazená-lo também. Em programas simples este é um tipo de alocação eficaz, o problema começa quando não sabemos inicialmente a dimensão da variável ou array ou quantidade de memória necessária. Aqui entra a alocação dinâmica.

7.2. Alocação dinâmica

A alocação dinâmica é usada quando não sabemos de antemão o volume de dados a ser armazenado pela variável ou array ou a dimensão do array e só descobriremos ao longo da execução do programa. Para realizar a alocação utiliza-se do comando *new*. Vejamos como alocar dinamicamente uma matriz com suas dimensões (linhas *l* e colunas *c*):

```
int** matriz = new int*[l];
for(int i=0; i<=l; i++){
	matriz[i] = new int[c];
}
```

Para apagar um array ou variável alocado dinamicamente utiliza-se o comando *delete*. Vejamos como desalocar a matriz alocada no exemplo anterior:

```
for(int i=0; i<=l; i++){
	delete [] matriz[i];
}
delete []matriz;
```

8. Arquivos

Até o momento temos manipulado os dados internamente na memória, sem manter as alterações após o encerramento do programa. Com a utilização de arquivos podemos tornar o armazenamento de dados persistente, retomar processamentos anteriores e utilizar os dados gerados em outros programas ou para outros fins.

Para manipular arquivos utilizaremos uma nova biblioteca: *fstream*. Essa nova biblioteca faz parte das bibliotecas de entrada e saída de dados. Veremos a seguir como abrir e fechar arquivos, persistir (gravar) dados em um arquivo e como lê-los:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(){

    string arquivo, linha, texto;
    /*
    *cria uma string chamada arquivo_entrada;
    *cria uma string chamada linha para armazenas as linhas do arquivo;
    *cria uma string chamada texto que será armazenada no arquivo;
    */

    cout << "Nome do arquivo de entrada com a extensão (ex.: arquivo.txt): ";
    cin >> arquivo;

    ifstream arquivo_entrada(arquivo); // abre o arquivo informado pelo usuário

    //verifica se o arquivo foi aberto corretamente
    if(arquivo_entrada.isopen()) {
        while(!arquivo_entrada.eof()){ //repete a ação enquanto o arquivo não chega ao fim
            getline(arquivo_entrada, linha); //pega o conteudo das linhas de arquivo_entrada e armazena em linha
            cout << linha << endl;
            arquivo_entrada >> texto; //ao invés de utilizar o cin, utiliza-se arquivo_entrada para persistir um texto no arquivo
        }
        arquivo_entrada.close(); //fecha o arquivo
    }
    else{
        cout << "Não foi possivel abrir o arquivo de entrada!" << endl;
    }

    return 0;
}
```

9. Práticas de programação

Cada programador tem seu estilo de programação, mas existem algumas coisas que ajudam a padronizar e organizar melhor os códigos, são as chamadas práticas de programação. Em geral, as práticas de programação tendem a diminuir o número de linhas do código e deixá-lo mais eficiente, vejamos algumas práticas:

- Identação: é a sequência estrutural do código. Essa prática deixa o programa bem organizado e ajuda a visualizar os blocos de comando e entender melhor o funcionamento do código. Vejamos abaixo um exemplo de código identado e um de código não identado:

```cpp
#include <iostream>

using namespace std;

int main(){
    int a, b;

    cout << "Digite um número a: " << endl;
    cin >> a;

    cout << "Digite um número b: " << endl;
    cin >> b;

    if(a>b){
        for(int i=a; i<b; i--){
            cout << i << endl;
        }
    }else if(a<b){
        for(int i=a; i<b; i++){
            cout << i << endl;
        }
    }else{
        cout << a << " = " << b << endl;
    }
    return 0;
}
```

```cpp
#include <iostream>

using namespace std;

int main(){
int a, b;
cout << "Digite um número a: " << endl;
cin >> a;
cout << "Digite um número b: " << endl;
cin >> b;
if(a>b){
for(int i=a; i<b; i--){
cout << i << endl;
}
}else if(a<b){
for(int i=a; i<b; i++){
cout << i << endl;
}
}else{
cout << a << " = " << b << endl;
}
return 0;
}
```

- Nomenclaturas: quanto as numenclaturas, nomes de variáveis devem ser iniciados com letras minusculas, em caso de mais de uma palavra, elas devem ser começados em letras maiúscula a partir da segunda (exemplo: numero, numeroDePalavras) e jamais devem começar com números. Funções devem sempre iniciar com letras maiúsclas.

10. Exercícios de Revisão

1. Escreva um programa para encontrar o maior valor entre cinco inteiros. Os valores devem ser lidos do teclado.

2. Implemente um programa para calcular o fatorial de um número N, lido do teclado.

3. Leia o nome de três competidores em um campeonato e suas respectivas pontuações. Ao final, informe qual o competidor/pontuação que ficou em primeiro, segundo e terceiro lugar.

4. Leia 10 números a partir do teclado e escreva primeiro os pares e depois os ímpares.

5. Fazer um programa que lê um valor, um operador (+,-,*,/) e um segundo valor e imprime o resultado da expressão. Use uma função para imprimir cada caso, passando valores e operador como parâmetro.

6. Declare um vetor de tamanho n (leia n do teclado), leia seus elementos e implemente uma função que retorna o maior valor do vetor, uma que retorna o menor valor do vetor, uma que retorna a média do vetor,

uma que encontre um valor x no vetor e retorna sua posição (caso x não exista no vetor, retorne -1) e uma que multiplique os elementos do vetor por um número x. Imprima a saída de cada função.

7. Crie um programa que leia do teclado um número x e um número y e armazene em um vetor a sequencia crescente de x a y se x for menor do que y e a sequencia decrescente se x for maior que y.

8. Faça um programa que some duas matriz de tamanhos [m][n] e apresente a transposta da matriz resultante.

9. Faça uma função recursiva que recebe dois valores, um de base e outro de expoente. Sua função deve retornar o valor de $base^{expoente}$. Use somente a operação de multiplicação.

10. Crie uma função que receba como parâmetros dois vetores de inteiros, v1 e v2, e as suas respectivas quantidades de elementos, n1 e n2. A função deverá retornar um ponteiro para um terceiro vetor, v3, com capacidade para (n1 + n2) elementos, alocado dinâmicamente. O vetore v3 deve conter a concatenação dos vetores v1 e v2. Por exemplo, se v1 = {5, 6, 2, 7} e v2 = {24, 4, 16, 81, 10, 12}, v3 irá conter {5, 6, 2, 7, 24, 4, 16, 81, 10, 12}. Declare assim:

```
int* interseccao(int *v1, int n1, int *v2, int n2);
```

No programa principal teste sua função, chamando tanto com vetores criados de forma estática quanto de forma dinâmica. Imprima o vetor de resultado no programa principal. A função de intersecção não deve imprimir nada, só retornar o vetor resultado. Não se esqueça de desalocar a memória de todos vetores alocados dinâmicamente.

11. Referências Bibliográficas

VIVIANE MIZRAHI, Victorine. **Treinamento em linguagem C - 2. ed**.

Stroustrup, Bjarne. **The C++ Programming Language – Third edition**. Addison-Wesley, 1997