Um raciocínio lógico leva você de A a B. A imaginação leva você a qualquer lugar que você quiser.

Albert Einstein

Abençoados os que têm sono, pois não tardarão em dormir.

Friedrich Nietzsche

1 Estruturas de dados

As estruturas de dados, na sua maioria dos casos, baseiam-se na nos tipos de armazenamento vistos dia a dia, ou seja, nada mais são que a transformação de uma forma de armazenamento já conhecida e utilizada no mundo real adaptadas para o mundo computacional. Por isso, cada tipo de estrutura de dados possui vantagens e desvantagens e cada uma delas tem sua área de atuação (massa de dados) otimizada.

Os dados manipulados por um programa podem possuir natureza distinta, isto é, podem ser números, letras, frases, etc. Dependendo da natureza de um dado, algumas operações podem ou não fazer sentido quando aplicadas a eles. Por exemplo, não faz sentido falar em somar duas letras - algumas linguagens de programação permitem que ocorra a soma dos valores *ASCII* correspondentes de cada letra. Outras linguagens interpretam a operação de soma como uma *concatenação* de caracteres.

Para poder distinguir dados de naturezas distintas e saber quais operações podem ser realizados com eles, os programas lidam com o conceito de tipo de dados. O tipo de um dado define o conjunto de valores ao qual uma variável pode assumir, bem como o conjunto de todas as operações que podem atuar sobre qualquer valor daquela variável. Por exemplo, uma variável do tipo inteiro pode assumir o conjunto de todos os números e de todas as operações que podem ser aplicadas a estes números.

Os tipos de dados manipulados por um programa podem ser classificados em dois grupos: *atômicos* e *complexos ou compostos*. Os tipos atômicos são aqueles cujos elementos do conjunto de valores são indivisíveis, exemplo: o tipo inteiro, real, caractere e lógico. Por outro lado, os tipos complexos são aqueles cujos elementos do conjunto de valores podem ser decompostos em partes mais simples. Se um tipo de dado por ser decomposto, então o tipo de dado é dito estruturado, e a organização de cada componente e as relações entre eles constitui a disciplina de *Estrutura de Dados*.

2 Dados Homogêneos e Heterogêneos

Uma estrutura de dados, que utiliza somente um tipo de dado, em sua definição é conhecida como *dados homogêneos*. Variáveis compostas homogêneas correspondem a posições de memória, identificadas por um mesmo nome, individualizado por índices e cujo conteúdo é composto do mesmo tipo. Sendo os *vetores* (também conhecidos como estrutura de dados unidimensional) e as matrizes (estruturas de dados bidimensionais) os representantes dos dados homogêneos.

Uma estrutura de dados é chamada de *heterogênea*, quando envolve a utilização de mais de um tipo básico de dado (inteiro ou caractere, por exemplo) para representar uma estrutura de dados. Normalmente, este tipo de dado, é chamado de *registro*.

Um registro é uma estrutura de dados que agrupa dados de tipos distintos ou, mais raramente, do mesmo tipo. Um registro de dados é composto por certo número de campos de dados, que são itens de dados individuais. Registros são conjuntos de dados logicamente relacionados, mas de tipos diferentes (numéricos, lógicos, caractere, etc).

3 Dado Homogêneo - Conhecendo os vetores

O vetor é uma *estrutura de dados linear* que necessita de somente um índice para que seus elementos sejam endereçados. É utilizado para armazenar uma lista de valores do mesmo tipo, ou seja, o tipo vetor permite armazenar mais de um valor em uma mesma variável. Um dado vetor é definido como tendo um número fixo de células idênticas (seu conteúdo é dividido em posições). Cada célula armazena um e somente um, dos valores de dados do vetor. Cada uma das células de um vetor possui seu próprio endereço, ou índice, através do qual pode ser referenciada. Nessa estrutura todos os elementos são do mesmo tipo, e cada um pode receber um valor diferente.

Algumas características do tipo vetor:

- Alocação estática (deve-se conhecer as dimensões da estrutura no momento da declaração)
- Estrutura homogênea
- Alocação sequencial (bytes contíguos)
- Inserção/Exclusão
  - Realocação dos elementos
  - Posição de memória não liberadas

Os vetores também são chamados de *matrizes unidimensionais* por possuírem somente um índice. Os índices de um vetor em C++ irão sempre começar em zero (0). No C++, os vetores também são conhecidos como *arrays*. Um vetor é identificado por um único nome, individualizados por índices, e cujo conteúdo é de um mesmo tipo. O índice indica a posição do elemento na estrutura. Assim, para podermos manipular um particular elemento na estrutura é preciso fornecer o nome de tal estrutura e a sua localização nesta estrutura indicada pelo índice. Utilizamos *estruturas indexadas* quando é necessário armazenar dados na memória principal a fim de poder percorrê-los em uma ordem diferente do que a de sua leitura ou quando precisamos percorrê-los diversas vezes, por exemplo.

Um vetor é uma estrutura de dados que contém um número fixo de dados agrupados por um mesmo tipo, que pode ser qualquer um dos tipos predefinidos na linguagem C++ (*int*, *char*, *float*, *double*), um tipo vetor, um tipo registro ou ainda um tipo definido pelo usuário. A figura 1 ilustra graficamente o *conceito do vetor*. Como pode ser observado na figura, a posição 3 do vetor em o valor 90.

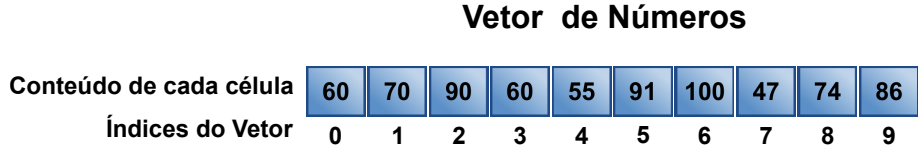


Figura 1: Conceito de vetor.

O programa 1 apresenta o vetor representado na figura 1.

Programa 1: Um vetor de números.

```
1 // um programa manipulando um vetor de números
```

```cpp
// programa_001.cpp
#include "biblaureano.h"

int main(){
    const int CAPACIDADE = 10;

    int numeros[CAPACIDADE] = {60, 70, 90, 60, 55, 91, 100, 47, 74, 86};

    for( int indice = 0; indice < CAPACIDADE; ++indice){
        cout << numeros[ indice ] << " ";
    }

    cout << endl; //impressão do final da linha

    cout << "Game Over!!!" << endl;
    return 0;
}
```

3.1 Entendendo o programa

Para declarar uma variável do tipo vetor, basta informar o nome da mesma e entre [e] (colchetes) a capacidade do vetor.

```cpp
    const int CAPACIDADE = 10;

    int numeros[CAPACIDADE] = {60, 70, 90, 60, 55, 91, 100, 47, 74, 86};
```

Neste caso, além de declarar o vetor inteiro *numeros* o mesmo já recebe um conjunto de valores pré-definidos, que são mostrados na sequência do programa. Observe que a variável *item* faz o papel de índice do vetor, a partir que começa do valor 0 (início do vetor) e é atualizado até chegar no valor 9 (última posição do vetor, isto justifica a condição $indice < CAPACIDADE$):

```cpp
    for( int indice = 0; indice < CAPACIDADE; ++indice) {
        cout << numeros[ indice ] << " ";
    }
```

Lembre-se sempre, um vetor sempre começa na posição zero (0), logo um vetor de 10 posições começa na posição 0 e termina na posição 9.

4 Cinto de utilidades - como guardar tantos itens ?

Você deve conhecer o Batman! Se não conhece, está perdendo muita coisa. Enfim, o Batman tem um cinto de utilidades, onde ele carrega os mais diversos itens (batrangue, batpunhal, batcorda). Em um jogo, como representar todos itens, que podem ser mais mais variados possíveis ? O mais correto seria *concentrar* estes itens em apenas um único local. Veja o programa 2.

Programa 2: Batman e o seu cinto de utilidades.

```cpp
// programa com cinto de utilidades
```

```cpp
// programa_002.cpp
#include "biblaureano.h"

int main(){
   //capacidade de itens que o cinto pode carregar
   const int CAPACIDADE_MAXIMA = 15;

   string batCinto[ CAPACIDADE_MAXIMA ];
   int qtdItens = 0;

   batCinto[ qtdItens++ ] = "batcorda"; //posição 0
   batCinto[ qtdItens++] = "batrangue"; //posição 1
   batCinto[ qtdItens++ ] = "bataspirina"; //herói também tem dor de cabeça

   cout << "Seu batCinto tem os seguintes itens:" << endl;
   for( int item = 0; item < qtdItens; ++ item){
       cout << batCinto[ item ] << endl;
   }

   string maoDireita = "batfumaça";
   string maoEsquerda = "batfone";

   cout << "Na mão direita o batman tem:" << maoDireita << endl;
   cout << "Na mão esquerda o batman tem:" << maoEsquerda << endl;

   //pegando itens do cinto
   batCinto[0].swap(maoDireita);
   cout << "Agora, na mão direita o batman tem:" << maoDireita << endl;
   cout << "Seu batCinto tem os seguintes itens:" << endl;
   for( int item = 0; item < qtdItens; ++ item){
       cout << batCinto[ item ] << endl;
   }

   char continua;
   do{
      char confirma;
      string novoItem;
      novoItem = readString("Entre com o nome do novo item:");

      confirma = readChar("Confirma o novo item (s/n)?");
      //carregando o cinto
      if( confirma == 's' ){
         //ops, tem que verificar se a capacidade permite
         if( qtdItens < CAPACIDADE_MAXIMA ){
            batCinto[ qtdItens++ ] = novoItem;
         }
         else{
             cout << "Cinto cheio!!!" << endl;
         }
      }
      continua = readChar("continua (s/n) ?");
   }while( continua == 's');

   cout << "Seu batCinto tem os seguintes itens:" << endl;
   for( int item = 0; item < qtdItens; ++ item){
       cout << batCinto[ item ] << endl;
   }

   cout << "Game Over!!!" << endl;
   return 0;
}
```

O programa 2 é quase idêntico a vários outros programas já vistos em aula. A diferença está no momento da criação da variável *batCinto*, que nada mais é que um *vetor* de strings (figura 2). Define-se como vetor uma variável que possui várias ocorrências de um mesmo tipo. Cada ocorrência é acessada através de um índice. Os índices de um vetor em C++ irão sempre começar de zero, fato que deve ser lembrado pois geralmente este detalhe é um grande causador de problemas. Portanto, para se acessar a primeira ocorrência de um vetor deve-se indicar o índice zero (igual ao a *string*). Um vetor também é conhecido como *array*.

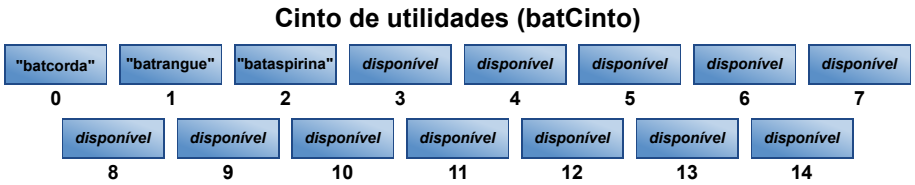


Figura 2: Vetor *batCinto*.

5 Entendendo o programa

Infelizmente, o tamanho de um vetor em C++ deve ser definido no momento da sua definição. É uma boa ideia criar um variável constante para controlar o tamanho máximo de um vetor.

```
1  //capacidade de itens que o cinto pode carregar
2  const int CAPACIDADE_MAXIMA = 15;
```

Como comentado anteriormente, O tamanho de um vetor é dado no momento da sua criação. O tamanho máximo de um vetor é informado entre [ ] (colchetes).

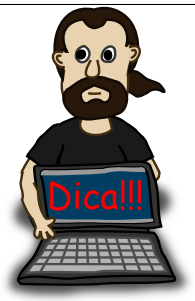
```
1  string batCinto[ CAPACIDADE_MAXIMA ];
```

Na aula anterior, nós acessamos posições diferentes de uma *string*. Na prática, uma *string* é um tipo de vetor de caracteres. Relembrando que devemos indicar a posição (índice) do nosso cinto de utilidades:

```
1  int qtdItens = 0;
2
3  batCinto[ qtdItens++ ] = "batcorda"; //posição 0
4  batCinto[ qtdItens++] = "batrangue"; //posição 1
5  batCinto[ qtdItens++ ] = "bataspirina"; //herói também tem dor de cabeça
6
7  cout << "Seu batCinto tem os seguintes itens:" << endl;
8  for( int item = 0; item < qtdItens; ++ item) {
9     cout << batCinto[ item ] << endl;
10 }
```

Lembre-se de como funciona o operador de pós-incremento ++. Primeiro é retornado o valor e somente depois é realizado o incremento. Seria o mesmo que escrever:

```
1  int qtdItens = 0;
2
3  batCinto[ qtdItens ] = "batcorda"; //posição 0
4  qtdItens++;
5  batCinto[ qtdItens ] = "batrangue"; //posição 1
6  qtdItens++;
7  batCinto[ qtdItens ] = "bataspirina"; //herói também tem dor de cabeça
8  qtdItens++;
9
10 cout << "Seu batCinto tem os seguintes itens:" << endl;
11 for( int item = 0; item < qtdItens; ++ item) {
12     cout << batCinto[ item ] << endl;
13 }
```



Assim como uma *string*, um vetor começa na posição 0. No nosso exemplo, o nosso cinto de utilidades tem 15 espaços (começando em 0 e indo até 14). Caso você tente acessar a posição 15 ou uma posição negativa do vetor o seu programa se comportará de forma desastrosa, podendo inclusive travar.

Após ter carregado o cinto com valores iniciais, podemos utilizar itens do cinto. Lembre-se que uma *string* é um objeto, e que todo objeto tem habilidades (métodos) que podem ser chamados. No nosso exemplo, o nosso herói está guardando um item no cinto e pegando outro:

```
1   string maoDireita = "batfumaça";
2   string maoEsquerda = "batfone";
3
4   cout << "Na mão direita o Batman tem:" << maoDireita << endl;
5   cout << "Na mão esquerda o Batman tem:" << maoEsquerda << endl;
6
7   //pegando itens do cinto
8   batCinto[0].swap(maoDireita);
9   cout << "Agora, na mão direita o Batman tem:" << maoDireita << endl;
10  cout << "Seu batCinto tem os seguintes itens:" << endl;
11  for( int item = 0; item < qtdItens; ++ item) {
12      cout << batCinto[ item ] << endl;
13  }
```

Qualquer habilidade (método) de uma classe *string* pode ser utilizada. Basta indicar o nome do vetor e a posição, separado por um ponto (.) e o nome da habilidade. Por exemplo, vamos supor que você quer eliminar o conteúdo da posição 3 do cinto:

```
1   batCinto[3].clear();
```

Apenas lembre-se que o método *clear()* limpa o conteúdo de uma *string*, neste caso, a posição 3 do nosso vetor será limpa mas esta mesma posição continuará disponível para uso (o vetor não diminui de tamanho).

E finalmente damos a possibilidade do nosso herói acrescentar novos itens no seu cinto de utilidades:

```
    char continua;
    do {
        char confirma;
        string novoItem;
        novoItem = readString("Entre com o nome do novo item:");

        confirma = readChar("Confirma o novo item (s/n)?");
        //carregando o cinto
        if( confirma == 's' ) {
            //ops, tem que verificar se a capacidade permite
            if( qtdItens < CAPACIDADE_MAXIMA ) {
                batCinto[ qtdItens++ ] = novoItem;
            }
            else {
                cout << "Cinto cheio!!!" << endl;
            }
         }

         continua = readChar("continua (s/n) ?");
    } while( continua == 's');
```

6 Uma nova versão do mesmo programa

No programa 2 colocamos os itens iniciais do cinto em cada posição, mas a linguagem C++ permite que no momento da definição do vetor já possamos acrescentar os itens. Para tal, basta declarar o vetor e atribuir a lista de valores entre { e } (chaves) e separados por vírgulas (,). Veja o programa 3:

Programa 3: Batman e o seu cinto de utilidades.

```
// programa com cinto de utilidades
// uma nova versão
// programa_003.cpp
#include "biblaureano.h"

int main(){
    //capacidade de itens que o cinto pode carregar
    const int CAPACIDADE_MAXIMA = 15;

    string batCinto[ CAPACIDADE_MAXIMA ] ={"batcorda","batrangue","bataspirina"};

    cout << "Seu batCinto tem os seguintes itens:" << endl;
    for( int item = 0;
          item < CAPACIDADE_MAXIMA //garante que não será ultrapassado
          && !batCinto[item].empty(); //verifica se a posição não está vazia
          ++ item){
         cout << batCinto[ item ] << endl;
    }
    cout << "Game Over!!!" << endl;
    return 0;
}
```

6.1 Entendendo o programa

Repare que estamos criando o nosso cinto de utilidades (vetor) com 15 posições (como no programa 2), mas agora os valores iniciais estão entre entre { e } (chaves) e separados por vírgulas (,).

```
    const int CAPACIDADE_MAXIMA = 15;

    string batCinto[ CAPACIDADE_MAXIMA ] ={"batcorda","batrangue","bataspirina"};
```

No programa 2, nós sabíamos *exatamente* qual a posição que paramos no vetor (pela utilização da variável *qtdItens*). Como neste caso não sabemos quais as posições utilizadas, podemos verificar se a posição está vazia ou não. Só temos que tomar o cuidado e não ultrapassar o final do vetor:

```
1    cout << "Seu batCinto tem os seguintes itens:" << endl;
2    for( int item = 0;
3          item < CAPACIDADE_MAXIMA //garante que não será ultrapassado
4          && !batCinto[item].empty(); //verifica se a posição não está vazia
5          ++ item) {
6        cout << batCinto[ item ] << endl;
7    }
```

O programa verifica se: (a) a capacidade do cinto já não foi ultrapassado e (b) se a posição do cinto está vazia.

Lembre-se da tabela verdade, de como é utilizando o operador lógico && (E lógico)!

```
1    item < CAPACIDADE_MAXIMA //garante que não será ultrapassado
2    && !batCinto[item].empty(); //verifica se a posição não está vazia
```

Se a primeira condição for falsa, a segunda nem será verificada, pois F && qualquer outro valor sempre resultará em falsidade. O teste inverso não poderia ser realizado, pois o final do vetor já poderia ter sido ultrapassado e resultaria em erro:

```
1    !batCinto[item].empty(); //verifica se a posição não está vazia
2    && item < CAPACIDADE_MAXIMA //garante que não será ultrapassado
```

Neste caso teremos um erro de execução (que muitas vezes não é detectado facilmente), pois a posição não existe e a verificação se o limite do vetor foi ultrapassado é realizado depois do erro.

7 Resolvendo um problema passo-a-passo

Vejamos um exemplo do uso de vetores: desejamos calcular a média das *scores* de um grupo de jogadores, e indicar o percentual de jogadores que tiraram *scores* acima da média. Os passos necessários para cumprir esta tarefa são:

1. O primeiro passo é ler os *scores*. Para isso, primeiro precisamos saber quantos jogadores existem, e depois efetuar a leitura de cada uma:

```
1    int qtdjogadores;
2    qtdjogadores = readInt("Entre com a quantidade de jogadores:");
```

2. A seguir definimos um vetor para receber estes *scores* e realizamos a leitura do *score* de cada jogador.

```
1    int score[ qtdjogadores ];
```

```
    for( int jogador = 0; jogador < qtdjogadores; ++jogador) {
       cout << "Entre com o score do jogador " << jogador+1 << ":";
       score[ jogador ] = readInt();
    }
```

3. Na sequência podemos proceder ao cálculo da média: precisamos somar todos as *scores* e depois dividir a soma pelo número de jogadores:

```
    //somatória de todos os scores
    int soma = 0;
    for( int jogador = 0; jogador < qtdjogadores; ++jogador) {
       soma += score[ jogador ];
    }

    //cálculo da média
    float media = (float) soma/qtdjogadores;
```

4. Agora que obtivemos a média, devemos percorrer novamente os *scores*, contando quantos estão acima da média:

```
    int qtdAcima = 0;
    for( int jogador = 0; jogador < qtdjogadores; ++jogador) {
       if( score[ jogador ] > media) {
          ++qtdAcima;
       }
    }
```

5. Para depois calcularmos a porcentagem de jogadores que estão acima da média.

```
    float porcentagem = (100.0 * qtdAcima)/qtdjogadores;
```

6. Finalmente, podemos escrever os resultados:

```
    cout << "Média dos scores:" << media << endl;
    cout << "porcentagem de jogadores acima da média:" << porcentagem << endl;
```

Juntando aos trechos de código acima as declarações de variáveis necessárias, teremos um programa C++ completo (programa 4).

Programa 4: Resolvendo um problema passo-a-passo.

```
// resolvendo um problema passo-a-passo
// programa_006.cpp
#include "biblaureano.h"

int main(){
   int qtdjogadores;
   qtdjogadores = readInt("Entre com a quantidade de jogadores:");
   int score[ qtdjogadores ];

   //leitura dos scores individuais
   for( int jogador = 0; jogador < qtdjogadores; ++jogador){
      cout << "Entre com o score do jogador " << jogador+1 << ":";
      score[ jogador ] = readInt();
   }

   //somatória de todos os scores
   int soma = 0;
   for( int jogador = 0; jogador < qtdjogadores; ++jogador){
      soma += score[ jogador ];
   }

   //cálculo da média
   float media = (float) soma/qtdjogadores;

```

```
25     //verificação de quantos jogadores estão acima da média
26     int qtdAcima = 0;
27     for( int jogador = 0; jogador < qtdjogadores; ++jogador){
28        if( score[ jogador ] > media){
29           ++qtdAcima;
30        }
31     }
32
33     float porcentagem = (100.0 * qtdAcima)/qtdjogadores;
34
35     cout << "Média dos scores:" << media << endl;
36     cout << "porcentagem de jogadores acima da média:" << porcentagem << endl;
37
38     cout << "Game Over!!!" << endl;
39     return 0;
40  }
```

8 Obtendo o tamanho de um vetor

Na linguagem C++ não existe um comando que retorne o tamanho de um vetor, mas é possível obter o tamanho de um vetor utilizando o operador *sizeof*. Utiliza-se o operador *sizeof* para obter o tamanho total em bytes do vetor e posteriormente basta dividir pelo tamanho em bytes do tipo básico do vetor. O programa 5 ilustra este conceito ao obter o tamanho de um vetor de *int* e de um vetor de *float*.

Programa 5: Obtendo o tamanho de um vetor.

```
1  // pegando o tamanho de um vetor
2  // programa_007.cpp
3  #include "biblaureano.h"
4
5  int main(){
6     int numeros[]= {1,1,2,3,5,8,13,21,34};
7     for( int indice = 0; indice < sizeof(numeros)/sizeof(int); ++indice){
8        cout << "numeros[" << indice << "]=" << numeros[ indice ] << endl;
9     }
10
11    float outrosNumeros[]= {0.01,0.02,0.03, 3.1234566};
12    for( int indice = 0; indice < sizeof(outrosNumeros)/sizeof(float); ++indice){
13       cout << "outrosNumeros[" << indice << "]=" << outrosNumeros[ indice ] << endl;
14    }
15    cout << "Game Over!!!" << endl;
16    return 0;
17 }
```

Se você ultrapassar o fim de um vetor durante uma operação de atribuição, então os valores adicionais irão sobrepor outros dados da memória. Estes valores serão armazenados em sequência, seguindo os elementos do vetor na memória. Como não foi reservado espaço para guardá-los, eles sobreporão outras variáveis ou até mesmo uma parte do código do próprio programa que por acaso esteja na memória. Isto acarretará em resultados imprevisíveis, e nenhuma mensagem de erro do compilador avisará o que está ocorrendo.

Lembre-se: **O C++ não avisa a você quando o limite de um vetor foi ultrapassado**.

Providenciar a verificação do limite de um vetor é responsabilidade do programador.

9 Passando vetores para funções

Assim como outros tipos de dados, também é possível passar vetores como argumentos para função. No programa 6 observamos esta situação:

Programa 6: Passando vetores para funções.

```
// pegando o tamanho de um vetor
// programa_008.cpp
#include "biblaureano.h"

void lerNumeros(int numeros[], int quantidade);
void imprimeNumeros( const int numeros[], int quantidade);

int main(){
   int quantidadeNumeros = readInt("Entre com a quantidade de números:");
   int numeros[ quantidadeNumeros ];
   lerNumeros(numeros, quantidadeNumeros);
   imprimeNumeros(numeros, quantidadeNumeros);
   return 0;
}

void lerNumeros(int numeros[], int quantidade){
   for( int i=0; i<quantidade; ++i){
      numeros[i] = readInt("Entre com número para posição " +numeroToString(i)+":");
   }
   return;
}

void imprimeNumeros(const int numeros[], int quantidade){
   for( int i=0; i<quantidade; ++i){
      cout << "Conteúdo da posicao " << i << ":" << numeros[i] << endl;
   }
   return;
}
```

9.1 Entendendo o programa

Inicialmente declara-se os protótipos das funções. Como pode ser observado, não é necessário informar o tamanho do vetor.

```
void lerNumeros(int numeros[], int quantidade);
void imprimeNumeros( const int numeros[], int quantidade);
```

Os vetores são passados por referência para as funções: como os vetores podem ter uma tamanho razoável (dezenas, centenas ou até milhares de posições), a linguagem C++ determina que é mais eficiente existir apenas uma única cópia do vetor na memória, sendo portanto irrelevante o número de funções que a acessem. Assim, não são passados os valores contidos no vetor, somente uma referência para eles.

Repare que a função *lerNumeros* não retorna nenhum valor e recebe o vetor *numeros*. Para evitar que uma função altere o conteúdo de um vetor indevidamente, basta declarar o vetor como constante (*const*) como ocorre na declaração da função *imprimeNumeros*.

Na sequência, é dada a opção ao usuário de informar a quantidade de números e o vetor *numeros* é criado com a quantidade informada. Repare que a passagem do vetor é dados apenas pelo seu nome (*numeros*):

```
int main() {
   int quantidadeNumeros = readInt("Entre com a quantidade de números:");
   int numeros[ quantidadeNumeros ];
   lerNumeros(numeros, quantidadeNumeros);
   imprimeNumeros(numeros, quantidadeNumeros);
   return 0;
}
```

Finalmente temos a definição das funções *lerNumeros* e *imprimeNumeros*. A função responsável pela leitura recebe a o vetor como uma referência e portanto os valores lidos nesta função estarão disponíveis no programa principal. A função responsável pela impressão também recebe o vetor como uma referência, mas como uma referência constante e portanto qualquer alteração no vetor não estará disponível no programa principal. Cabe ressaltar que o controle do limite do vetor é feita pela variável *quantidade*.

```
void lerNumeros(int numeros[], int quantidade) {
   for( int i=0; i<quantidade; ++i) {
      numeros[i] = readInt("Entre com número para posição " +numeroToString(i)+":");
   }
   return;
}

void imprimeNumeros(const int numeros[], int quantidade) {
   for( int i=0; i<quantidade; ++i) {
      cout << "Conteúdo da posicao " << i << ":" << numeros[i] << endl;
   }
   return;
}
```

10 Exercícios

1. Escrever um programa que leia 20 números *double* para um vetor e calcule a média dos valores digitados.
2. Escrever um programa que leia 10 números *int* para dentro de um vetor. Após a leitura, o programa deve pedir para ser informado um número e deve dizer se este número foi lido anteriormente ou não. O programa termina quando for digitado o número 0 (zero).
3. Escreva um programa que leia dois vetores *float* de 10 posições e faça a multiplicação dos elementos de mesmo índice, colocando o resultado em um terceiro vetor. Exiba o vetor resultante.
4. Escreva um programa que leia um vetor de 50 posições de *char* e o escreva em outro vetor de forma invertida. Isto é, o elemento de índice 0 passará a ser o 49, 1 à 48, 2 à 47,..,49 à 0.
5. Ler um vetor com 30 letras. Após a leitura, perguntar ao usuário uma letra. Seu programa deve percorrer o vetor e dizer quantas ocorrências daquela letra existe.
6. Leia 20 valores para um vetor *int*. Depois, dado uma sequência de **N** números inteiros, determinar um segmento de soma máxima. Exemplo: Na sequência *5, 2, -2, -7,* ***3, 14, 10, -3, 9****, -6, 4, 1* a soma do segmento é 33.
7. Escreva um programa que leia um vetor gabarito de 10 elementos. Cada elemento do vetor contém um número inteiro 1, 2, 3, 4 ou 5 correspondente as opções corretas de uma prova objetiva. Em seguida o programa deve ler um vetor resposta, também de 10 elementos inteiros, contendo as respostas de um aluno. O programa deve comparar os dois vetores e escrever o número de acertos do aluno. Após ler e conferir o gabarito de um aluno, o programa deve solicitar uma resposta do tipo SIM ou NÃO para continuar o processamento do próximo aluno.
8. Faça um programa que leia dois vetores de caracteres, com 10 elementos cada, e intercale em um terceiro vetor. Imprima o vetor resultante.

9. Faça um programa onde leia uma frase do usuário e informe:
   a) Quantos caracteres tem a frase.
   b) Quantas palavras tem a frase.
   c) Quantas pontuações (.,;!?) tem a frase.
10. Faça um programa que leia uma frase e a mostre embaralhada (apenas palavras trocadas de lugar).
11. Faça um programa que leia uma frase e a mostre embaralhada (caracteres trocados de lugar).
12. Faça um programa que leia um vetor com dez números inteiros e depois o ordene do menor para o maior.
13. Faça um programa que leia um vetor com dez números inteiros e depois o ordene do maior para o menor.
14. No ano de 2097, o presidente dos Estados Unidos da Europa autorizou uma loteria na televisão. A condição seria que um programa de computador deveria mostrar algumas mensagens aleatórias durante os sorteios. A frase aleatória é gerada randomicamente selecionando frases ou palavras de um conjunto de categorias G,A,S e V de acordo com a seguinte regra: uma palavra/frase de cada grupo conforme a sequência G - A - S - V - G - A - S. Por exemplo: *My improved freedom benefits our common responsible national security*. As palavras de cada grupo são:

    Grupo G : my, our common, the party's, my family's, our children's, my fellow Europeans', the government's, the industry's, the consumers', the immigrants', the only truly

    Grupo A : improved, responsible, peacekeeping, free, pro-life, politically correct, integrated, federal, progressive, anti-crime, drug-addicted, gradual, democratic, genetically engineered, racial

    Grupo S :freedom, national security, abuse, opportunity, tax cut, congress, task force, Europe, decision, dialogue, future, community, answer, environment, set of family-values, legislation, discrimination

    Grupo V : benefits, improves, decreases, supports, is built on, is the best guarantee for, creates an opportunity for, forms, is necessary for, will be established to combat

    Faça o programa que gere as frases solicitadas pelo presidente.
15. Faça um programa que resolva a tabela verdade abaixo:

Tabela 1: Tabela verdade.

A	B	C	D	A && !D	B || C || (A && D)	A && B && C || A	(D && B) || A	!B && C && D
false	false	true	true					
true	true	true	false					
true	false	true	false					
true	false	true	true					
false	true	true	true					
true	false	false	false					
false	true	true	false					
false	true	false	false					
true	true	false	false					
true	true	false	true					

Observação: Repare que cada coluna pode ser armazenada em um vetor do tipo lógico e com apenas um laço de repetição você terá como resolver toda a tabela.