Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Linguagem de Programação I • DIM0120
◁ Notas de aula sobre Programação Abstrata (parte 1) ▷
– Ponteiro Genérico e Ponteiro para Função –
29 de agosto de 2018

Sumário

1 Introdução

O presente documento descreve um exercício de programação envolvendo os conceitos de **data abstraction** por meio de *ponteiro* `void` (ou ponteiro genérico) e *ponteiro para função*. Para motivar uso de tais conceitos, foi desenvolvido um projeto de programação com a função *filter* e outros com *busca binária*.

O objetivo é demonstrar que a busca binária pode ser aplicada sobre uma coleção qualquer de dados, desde que eles suportem uma ordem total, e; que a "filtragem" pode ser implementada de maneira independente do tipo de *predicado de seleção* e do *tipo de dado* sobre o qual queremos realizar a filtragem.

Lembre que estes dois projetos já foram trabalhados anteriormente e apresentavam as seguintes limitações: *(i)* o filter funcionava sobre um vetor de inteiros e separava apenas valores positivos não nulos, e; *(ii)* a busca binária (e a linear) foi desenvolvida para operar sobre um vetor de inteiros.

Antes de apresentar a descrição das tarefas, é importante introduzir as duas ferramentas de abstração que precisaremos para resolver os exercícios. São elas **ponteiro genérico** e **ponteiro para função**.

2 Ponteiro Genérico

Sabemos que uma variável ocupa um conjunto de *bytes* dentro da memória atrelada ao programa. A quantidade de *bytes* que uma variável ocupa é determinada pelo compilador de acordo com o **tipo** da variável, seja ele básico, uma estrutura, classe ou um arranjo.

```
1 // Vamos assumir que 'soma' ocupa 4 bytes, iniciando em 0x7fff5665976c
2 int soma=10; // Variável regular.
3 int *p;      // Variável ponteiro não inicializada.
4 p = &soma;   // Variável ponteiro inicializada.
5 *p = 20;     // Alterando conteúdo de 'soma' via ponteiro 'p'.
```

Quando declaramos um variável ponteiro, informamos na sintaxe qual o tipo de dado que o ponteiro aponta, como em `int *p;` na linha 3 no código anterior. Um ponteiro, quando corretamente inicializado (linha 4), guarda um endereço de memória que corresponde ao **início** de um bloco de memória. No exemplo anterior, a variável `p` armazena o endereço `0x7ff5665976c`. Na atribuição da linha 5, armazenamos o valor 20 em `soma`.

Mas um momento...

Como o compilador, "olhando" apenas o `p` na linha 5, "sabe" que ele deve "escrever" o valor 20 em 4 *bytes*? Em outras palavras, como o compilador determina o endereço **final** da memória que o ponteiro aponta?

A resposta é simples, quando o ponteiro `p` foi declarado (linha 3), o compilador guardou a informação de que `p` sempre vai apontar para um grupo de 4 *bytes*. De maneira mais genérica, o compilador armazena a informação sobre a quantidade de *bytes* associada ao ponteiro.

Portanto, saber o tipo de dados do ponteiro é importante pois permite (*i*) aplicar operações como **desreferenciar** um ponteiro para recuperar seu valor, e; (*ii*) realizar **aritmética de ponteiros** quando o ponteiro aponta para um segmento de memória contínuo (por exemplo, um vetor). Tais ações tornam-se possíveis pois o compilador "sabe" exatamente quantos bytes de memória um tipo específico (básico da linguagem ou composto) ocupa.

2.1 Ponteiros sem tipo. Isso pode Arnaldo?

Por outro lado, um ponteiro do tipo `void` é um tipo especial de ponteiro que apenas armazena (aponta para) um endereço de memória; ele não possui, atrelado a si, a informação sobre a

quantidade de *bytes* que compõem o tipo para o qual ele aponta.

```
void *ptr; // ptr é um ponteiro genérico.
```

Portanto, não é possível aplicar as ações descritas anteriores, como desreferenciar ou realizar aritmética de ponteiros, visto que o compilador não saberia como interpretar o conteúdo do endereço. Para poder interpretar corretamente o conteúdo ele deveria saber onde **termina** o conjunto de *bytes* (lembra que o inteiro tem 4 *bytes*?).

Mas então, para que serve um ponteiro genérico (além de complicar a vida dos alunos de LP1)???

2.2 "Enter the void pointer"

Aparentemente, a impossibilidade de desreferenciar ou aplicar operações aritméticas sobre um ponteiro `void` pode parecer uma grande desvantagem. Porém, ganhamos **flexibilidade** (ou abstração) já que um ponteiro genérico apenas guarda um endereço de início de um bloco de memória; ora, se tal ponteiro não está "amarrado" a um tipo específico, basta informar ao compilador (manualmente) quantos *bytes* ele deve considerar ao "interpretar" o ponteiro.

Desta forma, (1) sabendo qual o **endereço inicial do ponteiro** e (2) informando manualmente **quantos bytes o compilador deve considerar**, podemos fazer o ponteiro `void` apontar para qualquer tipo de dado!!! Lembre-se que o compilador precisa apenas destas 2 informações para interpretar a memória. Em outras palavras, um ponteiro `void` pode apontar para um valor inteiro, um valor real, um vetor de *strings*, uma estrutura, ou mesmo um objeto mais complexo. Veja o exemplo abaixo.

```
1  int nValue;
2  float fValue;
3
4  struct Something {
5      int mat;
6      float score;
7  };
8
9  Something sValue;
10
11 void *ptr;
12 ptr = &nValue; // válido.
13 ptr = &fValue; // válido.
14 ptr = &sValue; // válido.
```

Contudo, o dado apontado não pode ser diretamente desreferenciado; é preciso primeiramente converter o ponteiro para o tipo desejado, através de um *cast*, antes de podermos desreferenciar o ponteiro e ter acesso ao seu conteúdo.

```
1 int value = 5;
2 void *voidPtr = &value;
3
4 //cout << *voidPtr << endl; // ilegal: não posso desreferenciar um ponteiro void.
5 int *intPtr = static_cast<int*>(voidPtr); // É preciso converter p/ ponteiro inteiro...
6 cout << *intPtr << endl; // para então poder desreferenciar o ponteiro normalmente.
```

Este comportamento permite, por exemplo, passar ponteiros genéricos para função. É o que acontece, por exemplo, nas funções `bsearch` e `qsort` do C, definidas em `<stdlib.h>`.

Mas, uma pergunta natural seria: se não é possível saber para qual tipo um ponteiro genérico aponta, como fazer para saber qual *cast* usar para converter o ponteiro? Bom, a princípio, o programador é responsável por manter (ou passar) esta informação de alguma forma. Confira o exemplo abaixo:

```
#include <iostream>

enum Type {
    INT,
    FLOAT,
    CSTRING
};

void printValue(void *ptr, Type type) {
    switch (type) {
        case INT:
            std::cout << *static_cast<int*>(ptr) << '\n'; // cast p/ inteiro e dereferenciar.
            break;
        case FLOAT:
            std::cout << *static_cast<float*>(ptr) << '\n'; // cast p/ float e dereferenciar.
            break;
        case CSTRING:
            std::cout << static_cast<char*>(ptr) << '\n'; // cast p/ char e dereferenciar.
            break;
    }
}

int main() {
    int nValue = 5;
    float fValue = 7.5;
    char szValue[] = "Mollie";

    printValue(&nValue, INT);
    printValue(&fValue, FLOAT);
    printValue(szValue, CSTRING);

    return 0;
}
```

Mas e se o cliente escrevesse o seguinte código, o que aconteceria?

```
int nValue = 5;
printValue(&nValue, CSTRING);
```

A resposta é indefinida!

Como é possível perceber, este tipo de abstração é propenso a erro (caso o tipo correto não seja identificado na conversão) e parece não oferecer um grau de abstração muito alto. É por este motivo que a linguagem C++ oferece outras formas de programar abstratamente, as quais serão abordadas posteriormente. Mais adiante, na Seção 4.5 vamos aprofundar o uso de ponteiros genéricos dentro de um contexto da função `filter()`.

3 Ponteiro para Função

Ponteiro para função é um tipo de dado que guarda endereços de funções (código), ao invés de guardar endereço de uma variável regular. Desta forma, a utilização de ponteiro para função permite **passar uma função como argumento** para outra função.

Esta flexibilidade oferece maior poder de abstração para uma função, que pode delegar parte da sua operação para o cliente através de uma função que ele/ela deve fornecer via argumento. Para tornar este conceito mais claro, considere, por exemplo, que um programador X deseja implementar o algoritmo de ordenação `quicksort`. Idealmente, o programador X deveria conseguir programar uma função `quicksort` com um grau de abstração tal que seu correto funcionamento não dependesse do tipo de dado que o cliente deseja ordenar. Para que isso aconteça, o programador cliente deveria ser capaz de informar ao programador X como proceder para **comparar** dois elementos quaisquer do vetor, visto que a comparação é a operação básica necessária para realizar uma ordenação (i.e. estabelecer uma ordem total).

Neste caso, o programador X deve especificar que sua função `quicksort` deve receber um *ponteiro para uma função de comparação* que recebe dois elementos (de um mesmo tipo de dado que se deseja ordenar), digamos a e b, por parâmetro (usando ponteiro genérico) e retornar `true` se $a < b$, ou `false` caso contrário. Através da especificação deste requisito, o programador X tem independência suficiente para programar o algoritmo *quicksort* sem se preocupar com o tipo de dado que está sendo ordenado, visto que cabe ao cliente (programador da aplicação) fornecer o código necessário para comparar elementos durante o processo de ordenação.

Para demonstrar a sintaxe para declarar um ponteiro para uma função, considere o exemplo a seguir, no qual

1. declaramos um ponteiro para função regular com a sintaxe via ponteiro (linha 31) ou com classe template (linha 32);
2. declaramos um ponteiro para função como argumento de uma função via ponteiro (linha 21) ou com classe template (linha 22), e;
3. invocamos uma função através de ponteiro ou classe template (linha 24):

```
1  // pointer to functions
2  #include <iostream>
3  #include <functional>  // New (C++11) template class.
4  using namespace std;
5
6  int addition (int a, int b)
7  { return (a+b); }
8
9  int subtraction (int a, int b)
10 { return (a-b); }
11
12 /**!
13  *  Esta função aplica uma operação binária (passada como argumento)
14  *  sobre seus dois argumentos, x e y, e retorna o resultado.
```

```
15  *  @param x Primeiro argumento para a operação binária.
16  *  @param y Segundo argumento para a operação binária.
17  *  @param functocall Ponteiro para uma função que recebe dois 'int' e retorna um 'int'.
18  *
19  *  @return O resultado da operação binária executada sobre x e y.
20  */
21  //int operation (int x, int y, int ( *functocall )(int,int)) {
22 int operation (int x, int y, std::function< int ( int,int) > functocall ) {
23   int g;
24   g = functocall(x,y);
25   return g;
26 }
27
28 int main () {
29   int m,n;
30   // Declarando um ponteiro para função regular.
31   //int ( *minus )(int,int) = subtraction;
32   std::function< int (int,int) > minus = subtraction;
33
34   // Passando para a função a operação de adição para ser executada sobre 7 e 5.
35   m = operation (7, 5, addition);
36   // Passando para a função qual operação deve ser executada entre os dois parâmetros.
37   n = operation (20, m, minus);
38   cout << n;
39   return 0;
40 }
```

Portanto, para declarar um ponteiro para função precisamos especificar a assinatura da função, ou seja, quantos e quais parâmetros ela deve receber e que tipo ela deve retornar.

Ponteiros para função só podem receber endereços de funções que satisfaçam a assinatura especificada na declaração do ponteiro. Em resumo, as sintaxes para declarar um ponteiro para função ou classe template (C++11) é a seguinte:

```
// Declaramos um ponteiro para função com a sintaxe C/C++
tipo_retorno ( * nome_ptr_func ) ( tipo_arg1, tipo_ar2, ... );
// ou usamos a classe template do C++11
#include <functional>
std::function< tipo_retorno( tipo_arg1, tipo_arg2, ... ) > nome_ptr_func;
```

4 Tarefas

A seguir temos as tarefas que devem ser realizadas, em ordem crescente de complexidade. Os exercícios devem ser realizados com base nos arquivos auxiliares fornecidos via Sigaa.

4.1 filter — versão 1: criando uma biblioteca simples

Primeiramente você deve trabalhar sobre o projeto *filter*, pasta `[prj_filter_v1]`. Lembre que a função `filter()` foi desenvolvida para receber um vetor de inteiros e seu tamanho, para então "filtrar" no início do vetor apenas elementos positivos ou não-nulos. Por "filtrar" queremos dizer apenas mover os elementos *selecionados* para frente, preservando sua ordem

relativa e sobrescrevendo os elementos que não satisfazem o critério de seleção (no caso, ser um inteiro > 0). Desta forma, a função `filter()` deve retornar o novo tamanho lógico do vetor[1].

Nesta versão o projeto está na forma **monolítica**, ou seja, todo o código está contido em um único arquivo, `src/drive_filter.cpp`. Sua missão consiste em reorganizar o código de maneira a separar a função `filter()`, transformando-a em uma função-biblioteca.

Para isso você precisa criar o arquivo `src/filter.cpp`, o qual deverá conter apenas o código fonte da função `filter()`, ou seja, sua implementação.

Em seguida, você deve criar o cabeçalho `include/filter.h`, o qual deverá conter apenas a declaração da função `filter()`, ou seja, a assinatura da função.

O próximo passo é ajustar o arquivo da função principal, `src/drive_filter.cpp`, apagando código da função `filter()` de lá e incluindo o arquivo cabeçalho recém criado. Lembre-se que o código da função `filter()` agora reside em um arquivo separado!

Por último, você deve ajudar o comando de compilação de maneira que o compilador entenda que o projeto agora é composto por dois arquivos fonte, `src/filter.cpp` e `src/drive_filter.cpp`, e que precisa procurar na pasta `include` o arquivo `filter.h` com o cabeçalho da função `filter()`.

4.2 filter — versão 2: passando uma função predicado

Nesta versão você vai modificar a função `filter()` de maneira que ela possa receber do código cliente um ponteiro para uma **função predicado**. Esta modificação vai permitir que o cliente especifique qual tipo de critério de seleção a função `filter()` deve usar para decidir quais elementos devem permanecer no vetor original.

São exemplos de critérios de seleção: somente números pares, somente números ímpares, números negativos, números acima de 20, etc.

Para viabilizar esta tarefa você deverá alterar o código `filter.cpp` e `filter.h` de maneira que a função passe a receber um ponteiro para uma função predicado. A função predicado terá a seguinte assinatura:

```
bool predicate( int );
```

Isso quer dizer que toda função predicado receber um inteiros e retorna `true` se o elemento deve permanecer no vetor, ou `false`, caso contrário.

Além da alteração do código da função `filter()` mencionado acima, será necessário realizar modificações no arquivo `drive_filter.cpp` de maneira a criar as funções predicados que serão passadas por argumento para a nova versão da função `filter()`. Confira abaixo como seria uma possível chamada da nova versão da função `filter()` a partir da função principal:

```
    filter( V, arrSz, seleciona_primos );
```

[1] Lembre que o tamanho físico do vetor não será alterado, já que não vamos desalocar memória.

onde `V` é o ponteiro representando o vetor, `arrSz` é o tamanho do vetor e `seleciona_primos` é a função predicado que seleciona apenas números primos.

Teste sua resposta com vários predicados diferentes para perceber a versatilidade desta versão e como a abstração de código é importante. Perceba, em especial, que apesar de podermos ter vários predicados diferentes, o código da função `filter()` **não precisa ser alterada** a cada nova função predicado criada. Ou seja, cabe ao cliente programar o predicado que deseja, desta forma desassociando a operação de filtragem do tipo de predicado.

4.3 Busca binária para inteiros

Neste exercício você deve compreender o funcionamento da função `bsearch` do C++ que realiza a busca binária em um vetor. Esta função possui um alto grau de abstração, visto que ela é capaz de receber um ponteiro genérico, sobre o qual a busca será realizada, e um ponteiro para função de comparação, essencial para que a busca seja realizada.

Eventualmente, vamos desenvolver uma versão da função `filter()` com este nível de abstração. Porém, antes devemos entender melhor como uma função com este nível de abstração é *invocada* do lado da aplicação cliente.

Para isso, você deve partir do arquivo `bsearch_int.cpp`, compreender que desejamos fazer uma busca em um vetor de inteiros, e completar o código de maneira que a busca seja efetivamente realizada.

Em particular, estude na referência do `bsearch` o que cada um de seus parâmetros significa. Com isso, você vai notar que falta definir a função `compare()` que deve ser passada para o `bsearch`. Confira na assinatura do `bsearch` como deve ser o tipo da função `compare()`.

4.4 Busca binária para estudantes

Para deixar claro a versatilidade da busca binária devido a uma programação com abstração, neste exercício você deve adaptar o programa `bsearch_students.cpp` para que a busca binária seja realizada sobre um vetor de *estudantes*.

A definição do tipo `Student`, que representa um estudante, é feita no início do arquivo. A função principal cria um vetor de estudantes e tenta realizar a busca binária sobre o mesmo. Assim como no exercício anterior, sua missão é modificar ou completar o código recebido de maneira que a busca binária funcione corretamente.

4.5 filter — versão 3: passando uma vetor genérico

Neste exercício você deve implementar a versão final da função `filter()`. Esta versão busca alcançar o grau máximo de abstração, funcionando da mesma forma que a função `bsearch`. Isso quer dizer que a nova função `filter()` deverá receber um vetor genérico de elementos que precisam ser filtrados de acordo com um predicado também passado como argumento.

Antes de projetar uma solução para o problema em questão, vamos analisar algumas questões importantes, cujas respostas servirão de guia para a programação correta da solução desejada.

4.5.1 Uso de ponteiros genéricos e ponteiro para função

Vamos iniciar a análise de requisitos da função `filter()` a partir de seus argumentos e tomando a função `bsearch` como exemplo-guia. A nova função `filter()` deve receber um ponteiro para um vetor genérico de elementos, ou seja, um ponteiro `void`. Então temos a primeira pergunta:

"O vetor passado pode ter seus elementos modificados pela função?"

Se a resposta for "não", então devemos passar o vetor como `const void * vet`. Porém, se a função, por definição, deve ter a capacidade de alterar o vetor passado como argumento, então ele deve ser passado como `void * vet`. Qual destas duas opções descreve a situação da função `filter()`?

Portanto a função `filter()` deve receber como argumento *(i)* um ponteiro genérico para o vetor, *(ii)* o número de elementos no vetor, *(iii)* o tamanho, em bytes, de cada elemento do vetor, e; *(iv)* um ponteiro para a função predicado que desejamos usar para selecionar os elementos a permanecerem no vetor.

Contudo, existe um fato novo com relação ao ponteiro para a função predicado. Faça uma comparação entre a assinatura do ponteiro para função requerido por `bsearch` com o ponteiro para função requerido por `filter()` na versão anterior:

```
int ( *comp )(const void*, const void*) );
```

e

```
bool ( *predicate ) ( int );
```

O que há de diferente entre as versões? Você deve rapidamente perceber que o ponteiro para função de `bsearch` recebe dois ponteiros genéricos de entrada, que são justamente os elementos que desejamos comparar. No caso do `predicate()`, a função recebe apenas um elemento, que é o elemento a ser testado de acordo com o critério de seleção. Mas para que a função predicado seja genérica, é preciso que ela também receba um ponteiro genérico no lugar de um **inteiro**, visto que agora queremos filtrar um vetor de "qualquer coisa". Em outras palavras, o predicado deve ser aplicado sobre "qualquer coisa", e não apenas sobre um inteiro. Sendo assim, a assinatura do novo ponteiro para predicado ficaria:

```
bool ( *predicate ) ( const void * );
```

Utilizamos o modificador de tipo `const` no argumento do predicado porque sabemos que a função **não deve alterar o elemento passado como argumento**, ou seja, a função apenas deve analisar o elemento e retornar `true` ou `false`.

4.5.2 Funções para manipulação de memória

Outra pergunta importante que devemos considerar é: *"Dentro da função* `filter()`*, como fazer para saber para qual o tipo de dado para o qual devemos converter o ponteiro genérico para que possamos aplicar as operações de movimentação de dados?"* Esta pergunta é fundamental, pois já sabemos que a linguagem C++ não permite aplicarmos operações, como a atribuição ou operadores relacionais, sobre ponteiros genéricos; é necessário convertê-los para um tipo para que tais operações sejam viáveis.

A resposta para esta pergunta é: não vamos converter para tipo algum! Mas então, como realizar as operações? Bom, vamos fazer uso de funções especiais que manipulam **blocos de memória**, uma das característica de uma linguagem de nível médio, como é o caso de C/C++.

Através de funções que conseguem movimentar blocos de bytes dentro do vetor, podemos fazer a movimentação dos elementos sem nos preocuparmos em saber qual o seu tipo específico. Isso explica porque este tipo de função genérica precisa receber como um de seus argumentos **o tamanho de cada elemento do vetor em bytes**!

A função que será necessária para implementar esta versão do `filter()` é `memcpy`. De acordo com a referência, esta função copia uma certa quantidade de bytes de uma posição de memória (leia-se, ponteiro) para outra posição de memória. No caso do `filter()`, vamos usar o `memcpy` para movimentar para frente do vetor os elementos selecionados pelo predicado. Outras funções de manipulação de memória que eventualmente podem ser úteis, mas que não são necessária neste projeto são: `memmove` e `memset`.

4.5.3 Aritmética de ponteiros

Outro aspecto importante que precisamos utilizar na implementação do `filter()` é a **aritmética de ponteiros**. Basicamente ela será usada para "saltar" (*offset*) dentro do vetor para posições específicas. Por exemplo, considere o exemplo abaixo:

```
1 int main()
2 {
3     int vet[] = {21, -3, 15, -37, 43};
4
5     std::cout << vet[ 2 ] << std::endl;     // Imprimir o 3ro elemento.
6     std::cout << *(vet + 2) << std::endl;   // Imprimir o 3ro elemento.
7     std::cout << vet << std::endl;          // Imprimir o endereço 'base' do vetor.
8     std::cout << vet + 2 << std::endl;      // Imprimir o endereço 'offset' do vetor.
9 }
```

O programa acima cria um vetor de inteiros e em seguida usa aritmética de ponteiros para acessar e imprimir um elemento na posição 2 (terceiro elemento do vetor) na saída padrão. Isso é feito com os comandos `vet[2]` ou `*(vet+2)`, os quais são essencialmente idênticos, visto que o nome do vetor é um ponteiro e pode ser utilizado com indexação via colchetes ou usando ponteiro $+$ deslocamento (*offset*). Os últimos dois comandos imprimem na saída padrão o endereço base (de início) do vetor e o endereço de base $+$ um deslocamento (*offset*) de 2 inteiros (ou 8 bytes). Uma possível saída do programa seria:

```
15
15
0x77d9d0d5e910
0x77d9d0d5e918
```

Note que o segundo endereço de memória corresponde ao primeiro endereço **mais** 8 bytes de deslocamento, que corresponde exatamente a "saltar" os dois primeiros inteiros, visto que cada inteiro possui 4 bytes de comprimento.

No exemplo acima, para fazermos uma atribuição do elemento na última posição para a terceira posição poderíamos usar

```
vet[2] = vet[4]; // Atribuindo o último elemento para a terceira posição.
```

mas com a função `memcpy` ficaria

```
memcpy( vet+2, vet+4, sizeof(int)); // Atribuindo o último elemento para a terceira posição.
```

que significa mover quatro bytes que correspondem a um inteiro do *endereço origem* `vet+4` para o *endereço destino* `vet+2`. A versatilidade desta operação está no fato de que para o `memcpy` funcionar ele não precisa saber que se trata de um vetor de inteiros; se informarmos a quantidade certa de bytes que precisamos copiar, a função fará seu trabalho corretamente!

4.5.4 Juntando todas as peças do quebra-cabeça

Copiar um elemento de um lugar para outro dentro de um vetor é justamente a operação fundamental da função `filter()`. Como a função recebe como argumento a quantidade de bytes que corresponde a um elemento, torna-se fácil copiar elementos de uma posição para outra. Precisamos apenas determinar os endereços de origem e destino da cópia.

Considerando que a função `filter()` recebe um vetor através de um ponteiro genérico e que não é possível realizar aritmética de ponteiros sobre ponteiros genéricos, precisamos primeiramente converter este ponteiro para um tipo básico, a saber: **caractere sem sinal** ou `unsigned char`. Escolhemos um caractere sem sinal pois este tipo em geral possui apenas 1 byte de comprimento; assim, a aritmética de ponteiro terá como unidade básica 1 byte. Confira a seguir como poderíamos implementar uma função que recebe um vetor de inteiros por meio de um ponteiro genérico e faz uma atribuição do quarto elemento do vetor sobre o segundo elemento do mesmo vetor.

```
1  #include <iostream>
2  #include <string>
3  #include <cstring>
4
5  void func( void * myVet, size_t bytes )
6  {
7      // Ambos ponteiros apontam para o mesmo lugar.
8      unsigned char *p = static_cast<unsigned char*>( myVet );
9      // Depois da conversão é possível aplicar aritmética de ponteiros.
10
11     memcpy( p+2*bytes, p+4*bytes, bytes ); // Equivalente a p[2] = p[4];
12 }
13
```

```
14 int main()
15 {
16     int vet[] = {21, -3, 15, -37, 43};
17
18     std::cout << ">> Before:[ ";
19     for ( const auto & e: vet )
20         std::cout << e << " ";
21     std::cout << "]\n";
22
23     func( vet, sizeof(int) );
24
25     std::cout << ">> After:[ ";
26     for ( const auto & e: vet )
27         std::cout << e << " ";
28     std::cout << "]\n";
29 }
```

Note a importância de saber calcular os endereços de origem e destino da cópia a partir do ponteiro original do vetor e dos deslocamentos calculados de acordo com o tamanho do tipo básico em bytes, informado à função através do segundo argumento. Isso é feito na linha 11, na chamada da função `memcpy`.

A saída do programa acima seria:

```
>> Before:[ 21 -3 15 -37 43 ]
>> After:[ 21 -3 43 -37 43 ]
```

perceba que o quinto elemento (43) foi copiado **sobre** o terceiro elemento (15).

Resta agora combinar todos os elementos estudados até aqui para compor a versão final do `filter()`, isto é, ponteiros genéricos, ponteiros para função (que recebe um ponteiro genérico), aritmética de ponteiros e função de manipulação de memória.

Certifique-se que sua solução funciona aplicando a função `filter()`, por exemplo, sobre um vetor de *string*.

```
std::string V[] = { "multiplicar", "acordar", "perdoar", "multicampeao", "saturar",
                    "multipla escolha", "tornado" };
```

Considere como predicado, por exemplo, "filtrar" apenas palavras cujas 5 primeiras letras sejam "multi" (prefixo).

5 Considerações Finais

Neste exercício apresentamos o primeiro contato com programação abstrata utilizando o conceito de **ponteiro genérico** e **ponteiro para função**.

Estas duas formas de abstração permitem que operações sejam aplicadas sobre coleções de elementos genéricos e que ações específicas (funções) sejam passadas para o código. Assim, é possível programar (abstratamente) funções que desempenham certas ações cujos detalhes devem ser fornecidos pelo programador-cliente.

Foram dados como exemplos de algoritmo com este nível de abstração a função `qsort` e `bsearch`, do C++.

Apesar de ser um tipo de abstração poderoso e que funciona tanto em C quanto em C++, a implementação é razoavelmente complicada e depende muito do entendimento em baixo nível de como a memória é organizada.

Nas próximas aulas apresentaremos abordagens alternativas e mais simples de abstração típicas da linguagem C++. Por exemplo, no lugar de ponteiros genéricos, utilizaremos **templates**.

$\sim$ FIM $\sim$

Referências

[1] Alex, *LearnCpp.com: Tutorials to help you master C++ and object-oriented programming*, Chapter 6: Arrays, Strings, Pointers, and References, http://www.learncpp.com, 2007–2017.

[2] cplusplus.com, *C++ Language Tutorial*, Chapter: Pointers, http://www.cplusplus.com/doc/tutorial/pointers/, 2000–2017.