Linguagem de Programação I

Aula 7 - Ponteiro genérico e ponteiro de função

Objetivos da aula

- **Introduzir os conceitos de ponteiro genérico e ponteiro de função**
- Para isso, estudaremos:
  - Uso de parâmetros genéricos em funções
  - Algoritmo da busca binária polimórfica

- Ao final da aula espera-se que o aluno seja capaz de:
  - Criar e usar variáveis de tipo genérico
  - Implementar funções polimórficas em C

A importância do tipo em ponteiros

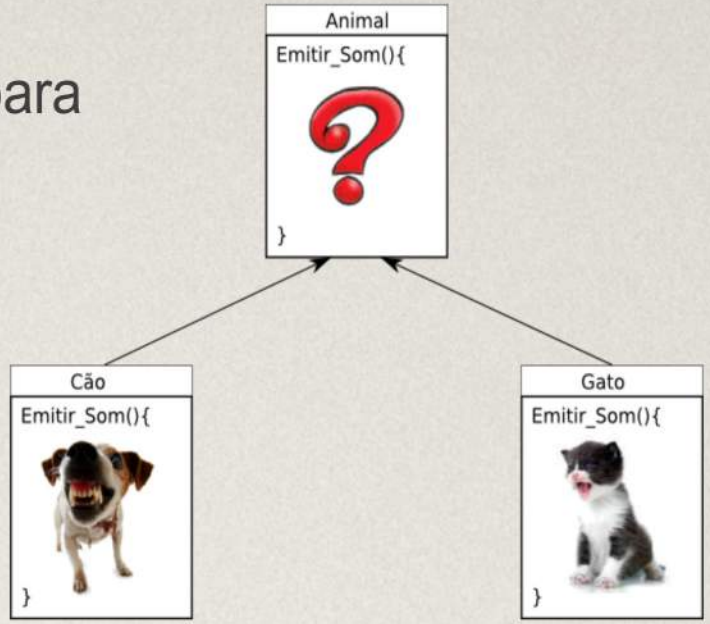
- Quando declaramos
  - `int *p`, obtemos um endereço de memória...
  - `float *p`, obtemos um endereço de memória...
  - `struct Data *p`, obtemos um endereço de memória...
- Se tudo é endereço de memória, por que precisamos declarar o tipo de dado que estará armazenado nesse endereço?

1. Útil para "desreferenciar" o endereço (acessar o conteúdo)
2. Útil para realizar aritmética de ponteiros (ex: `p++`, `p--`)

Mas quando seria útil usar o **ponteiro genérico** (`void*`) que permite apontar para "qualquer tipo"?

Ponteiro genérico

- Ponteiro genérico (`void*`) é útil como artifício para permitir polimorfismo em linguagem C
  - Polimorfismo permite diferentes tipos serem tratados da mesma forma



- Exemplo:
  - O algoritmo da busca binária que implementamos para o tipo `int` na aula de recursividade é o mesmo para outros tipos...
  - Se precisarmos fazer uma busca para os tipos `float`, `char`, `struct Cao` e `struct Gato`, teremos que implementar uma função para cada tipo?

Polimorfismo

- Existem vários tipos de polimorfismo
  - Ad hoc (com sobrecarga de funções ou operadores)
  - Paramétrico (com o uso de *templates*, *generics* e similares)
  - De subtipos (com orientação a objetos, através de subclasses)

- C não possui uma forma explícita para fazer polimorfismo
  - Exceto se usarmos uma união (`union` visto em ITP) ou um ponteiro genérico (`void*`)

- C++ possui diversas formas de polimorfismo e portanto não requer o uso do recurso do ponteiro genérico (veremos isso posteriormente)

Solução com ponteiro genérico

- Uma solução para a busca binária seria receber os parâmetros do arranjo e do valor a ser procurado como `void*`
- Mas, se declararmos a seguinte assinatura para a busca binária...

```
int binarySearch( void *array, void *value, int n )
```

Como poderemos saber em que tipo converter (cast) os parâmetros para poder comparar os seus valores?

Solução 1: passar mais um parâmetro para especificar o tipo e tratá-lo com um grande switch-case (um case para cada tipo).

Mas, seria +/- equivalente a escrever várias sub-rotinas

Solução 2: passar uma função para comparar os valores

Passar uma função como parâmetro?!

- Ideia da solução

```
1.  int binarySearch( TIPO array[], TIPO value, int low, int high, FUNCTION compare )
2.  {
3.      if( low > high )                            // não há elementos em array
4.          return -1;
5.
6.      int mid  = ( low + high ) / 2;              // índice do meio
7.
8.      int comp = compare( value, array[mid] );    // compara os elementos
9.
10.     if( comp == 0 )                             // são iguais --> achou
11.         return mid;
12.
13.     else if( comp > 0 ) // comp > 0 --> value > array[mid] --> busca a direita
14.         return binarySearch( array, value, mid + 1, high, comp );
15.
16.     else // value < array[mid] --> busca a esquerda
17.         return binarySearch( array, value, low, mid - 1, comp );
18. }
```

Tipo: ponteiro de função

- Como visto anteriormente, o código de máquina um programa encontra-se em uma região da memória chamada segmento de código
  - Logo, sempre existe um endereço de memória para o início de cada função presente no programa
- O ponteiro que aponta para o endereço de memória do início de uma função chama-se ponteiro de função
- Pode-se:
  - passar um ponteiro de função para outra função
  - usar um ponteiro de função como um campo de um registro
  - ou... usá-lo como se usa qualquer outro tipo de dado!!!

Declarando variável ponteiro de função

- Sintaxe: `TIPO (*nome_da_variavel) ( PARÂMETROS )`
- Exemplo: Função de impressão

```
1.  #include <iostream>
2.  int max( int a, int b ) { return a > b ? a : b; }
3.  int min( int a, int b ) { return a < b ? a : b; }
4.
5.  // ponteiros de função são também passados como parâmetro (ou retorno)
6.  void imprime( int a, int b, int (*funcao) ( int, int ) )
7.  {
8.      std::cout << funcao( a, b ) << std::endl;
9.  }
10. int main()  // declara uma variável (ponteiroFuncao) ponteiro de função que recebe
11. {           // 2 inteiros como parâmetros e retorna 1 inteiro como resultado */
12.
13.     int (*ponteiroFuncao) ( int, int );
14.     ponteiroFuncao = max;              // ponteiroFuncao armazena o endereço de max
15.
16.     imprime( 5, 9, ponteiroFuncao );  // --> imprime 9
17.     imprime( 5, 9, min );             // --> imprime 5
18.     return 0;
19. }
```

Usando typedef para simplificar

- O uso de ponteiros de função pode se tornar bastante confuso
- É aconselhável usar `typedef` para simplificar o código!

```
#include <iostream>
int max( int a, int b ) { return a > b ? a : b; }
int min( int a, int b ) { return a < b ? a : b; }

/* define um tipo (PonteiroFuncao) que permite apontar para funções que
   recebem 2 inteiros e retornam 1 */
typedef int (*PonteiroFuncao) ( int a, int b );

// usa o tipo para definir o parâmetro funcao
void imprime( int a, int b, PonteiroFuncao funcao )
{
    std::cout << funcao( a, b ) << std::endl;
}
```

Mas, e o polimorfismo? Os parâmetros de funcao não mudam!

Solução: usar o ponteiro genérico (`void*`) nos parâmetros

Solução para a busca binária (1)

- Definir funções com a mesma assinatura (e um typedef para elas)
- Usar `void*` nas assinaturas para representar um tipo qualquer
- Em cada função, desreferenciar o conteúdo para o "seu tipo"

```
1.  // compara dois valores inteiros
2.  int comparaInt( void *a, void *b )
3.  {
4.      int va = *( (int*) a );
5.      int vb = *( (int*) b );
6.      return va - vb;  // 0 se va = vb, neg. se va < vb ou pos. se va > vb
7.  }
8.
9.  // compara duas pessoas em função da idade
10. int comparaPessoa( void *a, void *b )
11. {
12.     Pessoa va = *( (Pessoa*) a );
13.     Pessoa vb = *( (Pessoa*) b );
14.     return va.idade - vb.idade;
15. }
16.
17. typedef int (*CompareFunc) (void*, void*);  // tipo para a função de comparação
```

Solução para a busca binária (2)

- A busca polimórfica

```
1.  int binarySearch( void *array, void *value, int low, int high,
2.                    CompareFunc compare, int size )
3.  {
4.      if( low > high )
5.          return -1;
6.
7.      int mid = ( low + high ) / 2;
8.
9.      // é necessário incluir o parâmetro size para poder acessar um
10.     // elemento do arranjo sem saber seu tipo
11.
12.     void *valueArray = array + size * mid; // equivalente a array[mid]
13.     int comp = compare( value, valueArray );
14.
15.     if( comp == 0 )
16.         return mid;
17.     else if( comp > 0 )
18.         return binarySearch( array, value, mid + 1, high, compare, size );
19.     else
20.         return binarySearch( array, value, low, mid - 1, compare, size );
21. }
```

Solução para a busca binária (3)

- Podemos, então, passar as diferentes funções de comparação

```
1.  #include <iostream>
2.  ...
3.  int binarySearch( void *array, void *value, int low, int high,
4.                   CompareFunc compare, int size ) { ... }
5.  ...
6.  int main()
7.  {
8.      int intArray[] = { 1, 4, 7, 8, 10, 15 };
9.      int intValue = 10;
10.
11.     Pessoa pessoaArray[] = {{ "A", 21 }, { "B", 28 }, { "C", 30 }};
12.     Pessoa pessoaValue = { "C", 30 };
13.
14.     if( binarySearch( intArray, &intValue, 0, 5, comparaInt, sizeof( int ) ) >= 0 )
15.         std::cout << "Inteiro encontrado!" << std::endl;
16.
17.     if( binarySearch( pessoaArray, &pessoaValue, 0, 2, comparaPessoa, sizeof( Pessoa ) ) >= 0 )
18.         std::cout << "Pessoa encontrada!" << std::endl;
19.
20.     return 0;
21. }
```

Resumo da aula

- Polimorfismo é útil para reaproveitar código, definindo funções (ou estruturas) que tratem diferentes tipos de dados
- Ponteiro genérico (`void*`) permite simular polimorfismo em C
- Ponteiro de função é útil em diversas utilidades:
  - Simular polimorfismo em C
  - Programação funcional
    - Funções como elementos de 1ª classe
  - Programação orientada a eventos (callbacks)
    - Passa-se uma função a um serviço (ou módulo), associando-a a um evento
    - Quando o evento ocorre, a função é chamada (callback)
    - Técnica muito usada em interfaces com o usuário (UI)