Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Linguagem de Programação I • IMD0030
– Exercício sobre Polimorfismo de Função –
27 de fevereiro de 2015

Sumário

1 Introdução

O presente documento descreve um exercício de programação envolvendo os conceitos de **programação polimórfica** por meio de templates e **ponteiro para função**. Para motivar uso de tais conceitos, foi desenvolvido um projeto de programação em C++ que inclui *(i)* uma *classe*, para representar retângulos, e *(ii)* um *programa principal* que realiza uma busca sequencial de um retângulo-alvo em uma coleção de retângulos.

A tarefa principal consiste em alterar o projeto de programação, mais especificamente o programa principal, de maneira que o programa consiga efetuar a busca do retângulo-alvo na coleção de acordo com *diferentes critérios de seleção*. O objetivo é realizar a modificação de maneira a não replicar código similar (a busca) para fins específicos (os diversos critérios de seleção).

O ajuste no programa principal envolve um padrão de programação comum em bibliotecas auxiliares de programação, como o STL. Este padrão desacopla uma ação necessária a execução de um algoritmo genérico. Assim, quando um cliente precisar usar o algoritmo genérico, ele necessita prover uma ação específica, adequada aos objetos que serão manipulados pelo algoritmo. Para este exercício o algoritmo é a *busca linear*—i.e. buscar um retângulo em uma coleção—e a ação é a de comparação—i.e. saber se o retângulo R_1 é "igual" ao retângulo R_2 de acordo com determinados critérios de seleção definidos pela aplicação.

2 Programação polimórfica

De maneira simplificada, a programação polimórfica consiste em desenvolver um pedaço de código correspondente a um algoritmo que atua sobre determinadas entidades, *sem* precisar especificar qual o tipo da entidade.

Por exemplo, quando descrevemos o algoritmo de ordenação por seleção (*selection sort*) sabemos que ele pode ser aplicado para uma coleção de números inteiros, caracteres, ou qualquer entidade que suporte o conceito de *ordem total* (i.e. é possível estabelecer uma ordem entre seus elementos). Portanto, idealmente seria muito prático se conseguíssemos implementar o algoritmo de ordenação por seleção **sem** precisarmos especificar o tipo dos elementos a serem ordenados. Felizmente, é possível programar desta forma ao utilizarmos, por exemplo, **templates** do C++.

Uma função template viabiliza a implementação do algoritmo sem especificar sobre qual tipo a função deve atuar. O tipo é passado como "argumento" para a função no momento que ela é utilizada por um código-cliente.

Um exemplo deve esclarecer este conceito. Considere o Código 1 que implementa uma função para realizar a busca sequencial em um vetor de inteiros. Note que na versão apresentada no Código 1 foi necessário indicar que o vetor e o alvo são inteiros (linha 4).

Código 1 Função que realiza a busca sequencial em um vetor de inteiros.

```
1  /**  Busca sequencial padrão.
2    *  Recebe como entrada, respectivamente, o vetor, seu tamanho, e o elemento procurado.
3    *  Retorna índice do elemento procurado no vetor se o encontrar, ou −1 caso contrário.  */
4  int linearSearch( int V[], int sz, int target ) {
5      for ( int i(0); i < sz; ++i ) { // Run through the array looking for the target.
6          if ( V[ i ] == target ) // Have we found it yet?
7              return i; // Yes! Return its position within the array.
8      }
9      return -1; // Sorry, it's not here...
10 }
```

Uma implementação equivalente da mesma função usando templates é exibida no Código 2. As diferenças mais importantes entre as duas versões estão destacadas em cor vermelha no Código 2. Os tipos específico `int` foi substituído por uma "variável" `Obj` (linha 5). Quando o código-cliente

Note que o parâmetro `target` foi modificado de maneira que deixou de ser uma passagem por *valor* para ser uma passagem por *referência constante* apenas por questão de desempenho—passar um endereço normalmente é bem mais eficiente do que copiar um parâmetro.

Com a introdução do template, é possível invocar a **mesma** função de busca sequencial (Código 2) sobre qualquer tipo de vetor, seja de inteiros, de cadeia de caracteres (*string*) ou de retângulos, como é o caso do projeto de programação em questão.

Código 2 Função que realiza a busca sequencial em um vetor de inteiros.

```
1  /**  Busca sequencial padrão com template.
2    *  Recebe como entrada, respectivamente, o vetor, seu tamanho, e o elemento procurado.
3    *  Retorna índice do elemento procurado no vetor se o encontrar, ou −1 caso contrário.  */
4  template <typename Obj>
5  int linearSearch( Obj V[], int sz, const Obj & target ) {
6      for ( int i(0); i < sz; ++i ) { // Run through the array looking for the target.
7          if ( V[ i ] == target ) // Have we found it yet?
8              return i; // Yes! Return its position within the array.
9      }
10     return -1; // Sorry, it's not here...
11 }
```

3 Ponteiros para função

Ponteiros para função correspondem exatamente ao que o nome indica: um apontador armazena o endereço de memória de um *segmento de código* correspondente a uma função. Uma das utilidades deste tipo de ponteiro é facilitar a passagem de uma função por parâmetro para outra função, desta forma viabilizando uma programação mais flexível.

Por exemplo, considerando o exemplo de busca sequencial apresentado no Código 2 percebemos que o critério de seleção utilizado para localizar um elemento no vetor é o operador de igualdade `operator==()` utilizado na linha 7. Mas o que significar ser "igual"? Se considerarmos que desejamos comparar retângulos podemos, por exemplo, definir que dois retângulos são iguais quando

- ⋆ seus vértices possuem as mesmas coordenadas Cartesianas;
- ⋆ possuem a mesma forma, ou seja, mesma largura e altura;
- ⋆ possuem a mesma área; ou
- ⋆ possuem a mesma distância para a origem do sistema Cartesiano coordenadas.

A verdade é que qualquer uma destas definições é válida, dependendo da necessidade da aplicação que pretende comparar retângulos. O fato é que a busca sequencial para funcionar, ou seja, para retornar um elemento a partir de uma coleção, precisa apenas de um **critério de seleção**. Normalmente utilizamos o critério de seleção *igualdade*, mas ele pode ser bem diverso, como no caso dos 4 critérios de comparação de retângulos descritos anteriormente.

Para tornar esta relatividade do critério de seleção ainda mais evidente, considere um exemplo em que temos a seguinte coleção de números inteiros: $[4, 10, 8, 17, 8, 21]$. Quando definimos o critério de seleção para busca sequencial como sendo a *"igualdade"*, a busca linear pelo elemento-alvo 8 iria retornar positivamente com o índice 2 (terceiro elemento no vetor). Por outro lado, se o critério de seleção fosse *"ser divisível"* para o elemento-alvo 5 então a busca sequencial retornaria, 1, visto que 10 (segundo elemento o vetor) é o único elemento divisível por 5.

3.1 A Função `comp()`

Portanto, para tornar a busca sequencial mais flexível, seria importante que o critério de seleção utilizado na busca fosse um de seus parâmetros. Isso é possível se utilizarmos ponteiro para função para informar à busca sequencial qual critério de seleção deve ser aplicado durante a busca.

Para tanto, podemos definir uma função de comparação `comp(a,b)` que recebe como entrada duas entidades, a e b, e retorna 0 (zero) se a e b satisfazem o critério de seleção, ou um valor $\neq 0$, caso contrário.

Por exemplo, se definirmos que o critério de seleção é a *igualdade* a ser aplicada sobre números inteiros, então a função de comparação poderia ser implementada da seguinte forma:

```
1 int comp( int a, int b ) {
2     return ( a - b );
3 }
```

Assim, se $a = b$ a função retorna 0, caso contrário retorna um número diferente de zero.

Generalizando para templates, temos:

```
1 template < typename Obj >
2 int comp( const Obj& a, const Obj& b ) {
3     return ( a - b );
4 }
```

É importante ressaltar que o versão `comp()` template só funcionará se os objetos passados possuírem a operação de subtração definida.

Obviamente, podemos substituir o código da função de comparação conforme a necessidade. Se quisermos usar como critério de seleção *"a ser divisível por b"* o código ficaria:

```
1 template < typename Obj >
2 int comp( const Obj& a, const Obj& b ) {
3     return ( a % b ); // resto da divisão inteira.
4 }
```

3.2 Passando `comp()` para a busca

Para passarmos a função de comparação (critério de seleção) de nosso interesse, precisamos passar um ponteiro para o código apropriado. Isso é feito acrescentando um parâmetro *ponteiro para função* na lista de argumentos da função de busca.

O Código 3 ilustra a criação e utilização de ponteiros para função, bem como a passagem de ponteiro para uma função.

4 Sua tarefa

4.1 Parte 1: busca linear polimórfica

A sua tarefa consiste em alterar o projeto de programação de maneira que ele possa trabalhar com 4 critérios de seleção para localizar retângulos-alvo em uma coleção de retângulos.

Código 3 Programa `ptrMinMax.cpp`: Exemplo de uso de ponteiro para função.

```
1  int max( int a, int b ) { return ( a > b ) ? a : b; }
2  int min( int a, int b ) { return ( a < b ) ? a : b; }
3
4  int whichOne( int V[], int sz, int ( *ptFunc ) ( int, int ) ) {
5      if ( sz <= 0 ) return -1;            // Evitar vetores vazios ou inválidos.
6      int dummy = V[ 0 ];                  // Primeiro elemento.
7      for( int i(1); i < sz; ++i ) {       // Aplicar a ação sobre todos elementos
8          dummy = ptFunc( dummy, V[ i ] );
9      }
10     return dummy;
11 }
12
13 int main ( ) {
14     // Ponteiro para uma função que recebe dois inteiros e retorna um inteiro.
15     int ( *p1 ) ( int, int ) = nullptr; // Inicialmente nulo.
16     int A[] = { 32, 2, 16, 64, 4, 1, 8 }, sz = 7;
17
18     p1 = max; // p1 aponta para a função max().
19     cout << "Valor retornado: " << p1( 3, 8 ) << endl;
20
21     p1 = min; // p1 agora aponta para a função min().
22     cout << "Valor retornado: " << p1( 3, 8 ) << endl;
23
24     cout << "Maior de todos: " << whichOne( A, sz, max ) << endl;
25     cout << "Menor de todos: " << whichOne( A, sz, min ) << endl;
26
27     return 0;
28 }
29 ------------------ SAÍDA DO PROGRAMA ------------------
30 $ ./ptrMinMax
31 Valor retornado: 8
32 Valor retornado: 3
33 Maior de todos: 64
34 Menor de todos: 1
35 $
```

Cada critério corresponderá a uma função de comparação específica. Os critério são aqueles definidos na Seção 3. Para provar a versatilidade de *templates* na função de busca linear e de *ponteiros para função*, acrescente ao programa principal código para realizar uma busca linear em um vetor de inteiros (usando a mesma função de busca) considerando os seguintes critérios de seleção:

1. *Igualdade*: se os números forem iguais entre si.
2. *Divisível*: se um número dividir outro.
3. *Primo entre si*: se os números forem mutuamente primos[1].

[1]Dois números são considerados mutuamente primos se não existir um inteiro maior do que 1 que divida os dois simultaneamente.

4.2 Parte 2: busca binária polimórfica

Modifique o programa principal (ou crie outro programa) que implemente a **busca binária polimórfica** (iterativa ou recursiva), ou seja, com versatilidade similar à busca sequencial. Aplique a busca sobre a coleção de retângulos e a coleção de inteiros.

Algumas mudanças são necessárias, contudo:

- ⋆ A função `comp(a,b)` agora compara a e b de maneira que retorna um valor < 0 se $a < b$, valor 0 se $a = b$ ou um valor > 0 se $a > b$.
- ⋆ O arranjo de retângulos deve ser ordenado (manualmente) segundo o *tamanho de sua área*—este é o critério de seleção da busca.
- ⋆ O vetor de inteiros deve ser rearranjado em ordem crescente.
- ⋆ O critério de busca para o vetor de inteiros é a *igualdade*.

$\sim$ FIM $\sim$