
O objetivo desta prática é relembrar as técnicas básicas da programação, assim como os algoritmos e estruturas de dados que vocês aprenderam até aqui.

Nós vamos implementar uma versão computacional do jogo de cartas *Blackjack*. A nossa versão é a seguinte:

Cada carta vale um certo número de pontos: um Ás vale 1, uma figura (valete, dama e rei) vale 10 e toda outra carta vale o número inscrito nela.

Um jogador humano joga contra o computador. Ele começa por tomar cartas ("draw") uma a uma tentando se aproximar o máximo possível de 21 pontos, sem ultrapassar este valor. Se o total de suas cartas ultrapassar 21 pontos, ele perde o jogo. O jogador humano pode decidir parar de pegar novas cartas ("stand"), e é então a vez do computador tomar cartas uma a uma. O vencedor é aquele cuja soma de cartas mais se aproxima de 21 pontos sem ultrapassar este valor.

Figure 1: *Os jogadores de cartas* de Michelangelo, 1594

1. Implementação da classe

Escreva a classe `Card` com os seguintes campos:

`naipe` um caracter que pode ter um entre os seguintes valores:

- 'H' - para copas (Heart)
- 'D' - para ouro (Diamond)
- 'S' - pour espada (Spade)
- 'C' - para paus (Cubs)

`valor` um inteiro entre [1,13] indicando o valor da carta

- 1 - Ás

2-10 - valor inscrito na carta

11 - Valete

12 - Dama

13 - Rei

A classe deve ter um construtor explícito `Card(int naipe, int valor)` que tem como parâmetros dois inteiros definindo o naipe e o tipo da carta. O construtor recebe um valor numérico representando a cor e deve transformá-lo em 'H', 'D', 'S' ou 'C' (este artifício simplificará parte do trabalho mais a frente) segundo a especificação a seguir: 0 = 'H', 1 = 'D', 2 = 'S' e 3 = 'C'.

2. Display

Adicione um método `public string toString()` que devolve uma cadeia de caracteres descrevendo a carta. Por exemplo, para a carta "Dama de Espadas", o método `toString()` irá devolver:

`S12`

Para testar a classe `Card` utilize a seguinte função `main()`:

```
int main(int argc, char** argv){
  Card c0(0,1);
  Card c1(1,10);
  Card c2(2,11);
  Card c3(3,13);
  cout << "" << c0.toString() << " " << c1.toString() << " "
       << c2.toString() << " " << c3.toString();
}
```

O programa deve ter como output:

```
H1 D10 S11 C13
```

3. Jogo de Cartas

As operações "draw" (pegar uma carta) e "shuffle" (embaralhar as cartas) podem ser realizadas sobre a lista de cartas. A operação "draw" tira a primeira carta da lista, enquanto "shuffle" modifica a ordem das cartas da lista de maneira aleatória. Esta última operação será tratada mais a frente.

Para representar a lista de cartas utilizaremos o template `list<T>` da STL (`http://www.sgi.com/tech/stl/`). No nosso caso o tipo do parâmetro `T` será `Card`. Desta forma, a declaração de uma nova lista de cartas `l` se fará por meio da instrução `list<Card> l`. Pesquise na API da STL os métodos do template `list<T>` que você precisará para este trabalho.

Para utilizar o template `list<T>` será necessário adicionar a biblioteca correspondente por meio da diretiva `#include<list>`.

4. A classe `Deck`

Implemente a classe `Deck` comportando um campo:
`cl` : uma lista encadeada de cartas.
Adicione um construtor explícito `Deck()` que cria uma lista `cl` com 52 cartas de jogo, 13 de cada naipe (basta enumerar explicitamente o conjunto de cartas do baralho)

5. Display

Adicione um método `string toString()` à classe `Deck` que retorna uma cadeia de caracteres descrevendo as cartas do monte.
A classe `Deck` será testada utilizando o método de teste a seguir (a ser chamada a partir da função `main`):

```
void test1(){
  Deck d;
  coud << d.toString() << endl;
}
```

O programa deve exibir o resultado a seguir, em uma única linha, cortada aqui para maior legibilidade:

```
[H1, H2, H3, H4, H5, H6, H7, H8, H9, H10, H11, H12, H13,
D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D12, D13,
S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13,
C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13]
```

6. Pegando uma carta

Adicione um método `Card draw()` à classe `Deck` que retira e retorna a primeira carta do monte. Se o monte está vazio, `draw` deve retornar `NULL`.
O método `draw` será testado utilizando-se o programa teste a seguir:

```
void test2(){
  Deck d;
  cout << "Drawn cards: " << endl;
  for (int i=0; i<52; i++)
    cout << d.draw().toString() << " ";
  cout << endl;
}
```

O programa deve exibir como resultado:

```
Drawn cards: H1 H2 H3 H4 H5 H6 H7 H8 H9 H10 H11 H12 H13 D1 D2 D3 D4 D5 D6 D7 D8
D9 D10 D11 D12 D13 S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 C1 C2 C3 C4 C5 C6
C7 C8 C9 C10 C11 C12 C13
```

7. Embaralhando as cartas

A técnica mais utilizada nos casinos para embaralhar as cartas é chamada *riffle*. Ela se decompõe em duas etapas:

1. Nós separamos o baralho arbitrariamente em duas pilhas.
2. Nós reunimos as duas pilhas efetuando um *riffle* (`http://www.youtube.com/watch?v=4V0OcENYqe8`), de modo a deixar que as cartas caiam uma a uma se entrelaçando.

Do ponto de vista matemático, nós modelamos o *riffle* da seguinte maneira:

- O local onde o baralho de n cartas é separado é escolhido aleatoriamente segundo uma lei binomial, i.e., a probabilidade de que a primeira pilha tenha k cartas é dada por $C(n,k)/2^n$, onde $C(n,k)$ designa o coeficiente binomial "k entre n": $C(n,k) = n!/k!(n-k)!$.
- Quando o *riffle* é feito, a próxima carta a cair é a primeira de uma das duas pilhas. A probabilidade que ela pertença à primeira pilha é $a/(a+b)$, onde a e b designam os números respectivos de cartas contidas em cada uma das pilhas.

O fato de utilizar uma lei binomial ao invés de uma lei uniforme para a separação do baralho se justifica na prática pelo fato de que é muito mais provável cortar o baralho no meio do que nas bordas. Quanto à distribuição de probabilidade entre as duas pilhas no momento do *riffle*, ela reflete bem a simetria do problema.

8. Selecionando a posição da separação do baralho

Adicione um método `int cut(int n)` à classe `Deck`, que determina o posicionamento da separação aleatória de baralho de n cartas seguindo uma lei binomial. O valor retornado pelo método `cut` corresponde ao número de cartas da primeira pilha. Existem várias maneiras de se implementar a lei binomial, faça uma pesquisa. Para gerar um número pseudo-aleatório com probabilidade uniforme vocês podem utilizar o método `rand()` (`http://www.cplusplus.com/reference/clibrary/cstdlib/rand/`)

9. "Cortando" o baralho

Dada a posição de separação do baralho em duas partes, queremos agora efetuar a divisão do baralho propriamente dita. Implemente um método `list<card> split(list<Card> l,`

`int c)` na classe `Deck` que separa a lista de cartas `l` na posição designada pelo índice `c`. A primeira pilha, armazenada na lista retornada pelo método, deve conter `c` cartas, enquanto a segunda pilha, armazenada em `l` deve conter `l.size()-c` cartas ao fim da operação.

O método `split` pode ser testado por meio da função a seguir:

```
void test4(){
  int c[3] = {0,26,52};
  for (int i=0; i<2; i++){
    cout << "Cut = " <<  c[i];
    Deck d;
    list<Card> l = d.split(d.cl, c[i]);
    cout << "First heap: " << l.toString();
    cout << "Second heap: " << d.cl.toString();
  }
}
```

Este programa cria três jogos ordenados de 52 cartas e os separa em dois, respectivamente antes da primeira carta, após a vigésima sexta, e após a quinquagésima segunda. O resultado deve ser como a seguir:

```
Cut = 0
First heap: []
Second heap: [H1, H2, H3, H4, H5, H6, H7, H8, H9, H10, H11, H12, H13, D1, D2, D3,
 D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10,
 S11, S12, S13, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13]
Cut = 26
First heap: [H1, H2, H3, H4, H5, H6, H7, H8, H9, H10, H11, H12, H13, D1, D2, D3,
 D4, D5, D6, D7, D8, D9, D10, D11, D12, D13]
Second heap: [S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, C1, C2, C3,
 C4, C5, C6, C7, C8, C9, C10, C11, C12, C13]
Cut = 52
First heap: [H1, H2, H3, H4, H5, H6, H7, H8, H9, H10, H11, H12, H13, D1, D2, D3,
 D4, D5, D6, D7, D8, D9, D10, D11, D12, D13, S1, S2, S3, S4, S5, S6, S7, S8, S9,
 S10, S11, S12, S13, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13]
Second heap: []
```

10. *Riffle*

Uma vez feita a separação do baralho em duas partes, nós devemos juntar as duas pilhas. Implemente um método `list<Card> riffle (list<Card> l1, list<Card> l2)` à classe `Deck`, que junta as duas listas de cartas `l1` e `l2` segundo o modelo aleatório do *riffle* descrito acima, depois retorna o resultado da fusão sob a forma de uma nova lista de cartas.

O método *riffle* será testado utilizando o método a seguir:

```
void test5(){
  Deck d;
  list<Card> l = d.split(d.cl, 26);
  cout << "First heap: " <<  l.toString() << endl;
  cout << "Second heap: " << d.cl.toString() << endl;
  cout<< "Riffle result: " << d.riffle(l, d.cl).toString() << endl;
}
```

Esta função cria duas pilhas de 26 cartas "cortando" um baralho com 52 cartas ordenadas, depois efetua o *riffle* com as duas pilhas resultantes. O resultado do *riffle* deve ser o seguinte:

```
First heap: [H1, H2, H3, H4, H5, H6, H7, H8, H9, H10, H11, H12, H13, D1, D2, D3,
 D4, D5, D6, D7, D8, D9, D10, D11, D12, D13]
Second heap: [S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, C1, C2, C3
, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13]
Riffle result: [H1, S1, S2, H2, H3, S3, S4, S5, S6, S7, S8, S9, H4, H5, S10, S11
, S12, H6, S13, C1, H7, H8, H9, C2, C3, H10, H11, H12, C4, H13, C5, C6, D1, C7,
D2, D3, C8, D4, D5, D6, D7, D8, D9, C9, D10, D11, C10, D12, C11, C12, D13, C13]
```

11. Combinar tudo

Adicione um método `void riffleShuffle(int n)` à classe `Deck`, que repete n vezes a sequência de operações (`cut + split + riffle`) sob a lista de cartas do `Deck` d. Teste o método `riffleShuffle` com a função de teste a seguir:

```
void test6(){
  Deck d;
  cout << d.toString() << endl;
  cout << d.riffleShuffle(7).toString() << endl;
  cout << endl;
  cout << d.toString() << endl;
}
```

Esta função cria um baralho de 52 cartas que é então embaralhado 7 vezes por meio do *riffle shuffle*. Ele imprime o baralho antes e depois do embaralhamento. O resultado deve ser algo bem embaralhado conforme a seguir:

```
[H1, H2, H3, H4, H5, H6, H7, H8, H9, H10, H11, H12, H13, D1, D2, D3, D4, D5, D6,
 D7, D8, D9, D10, D11, D12, D13, S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11,
 S12, S13, C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12, C13]

[C7, D10, D9, H5, S12, S5, C6, D11, H6, S1, D2, D7, D8, C1, C9, D12, H10, C3,
 S13, H4, C12, C13, C4, H13, H1, H12, S7, C5, C11, S10, H9, H8, S3, D3, H11,
 H7, H2, S8, D4, D13, D1, D5, S4, C8, C2, S9, S6, C10, H3, S11, S2, D6
```

12. Jogando *Blackjack* http://www.youtube.com/watch?v=PsK1c9ZBpuw

Temos agora à disposição todos os elementos para desenvolvermos o jogo *Blackjack*

Implemente a classe `Blackjack` comportando o método `int getPoints(Card c)`. Este método retorna o número de pontos da carta passada como argumento, conforme definido na introdução.

Para testar o seu código, utilize o método a seguir:

```
void test1(){
  Card c0(0,1);
  Card c1(0,7);
  Card c2(1,10);
  Card c3(2,11);
  Card c4(3,13);
  Blackjack bj;
  cout << bj.getPoints(c0) << endl;
  cout << bj.getPoints(c1) << endl;
  cout << bj.getPoints(c2) << endl;
  cout << bj.getPoints(c3) << endl;
  cout << bj.getPoints(c4) << endl;
}
```

O programa deve imprimir:

```
1
7
10
10
10
```

13. Os pontos de uma mão

Adicione à classe `Blackjack` um método `int getScore(list<Card> l)`, que recebe como parâmetro uma lista de cartas e que retorna a soma dos pontos das cartas da lista `l`.

Para testar o seu código, utilize o método de teste a seguir:

```
void test2(){
  list<Card> l;
  Card c0(0,1);
  l.push_back(c0);
  Card c1(1,10);
  l.push_back(c1);
  Card c2(2,11);
```

```
  l.push_back(c2);
  Card c3(3,13);
  l.push_back(c3);
  Blackjack bj;
  cout << bj.getScore(l) << endl;
}
```

O programa deve imprimir :

```
31
```

14. Interação com o jogador humano

A cada etapa do jogo o jogador humano pode pegar uma carta suplementar (comando "draw") ou parar (comando "stand") digitando respectivamente 'd' ou 's' no teclado, confirmado pela tecla enter. Para isto, adicione à classe `Blackjack` o método `char getCommand()` fornecido abaixo.

```
char getCommand(){
  char c;
  do{
   cout << "Enter d for draw or s for stand" << endl;
   cin >> c;
  }while(c != 'd' && c != 's');
  return c;
}
```

15. O jogador humano

Implemente o método `list<Card> humanPlayer(Deck d)` na classe `Blackjack` que recebe como argumento um monte de cartas já inicializado e embaralhado, e que retorna a lista de cartas obtidas pelo jogador humano.

O método `humanPlayer` vai então fazer um loop até que o jogador decide parar de jogar (pressionando 's' no teclado) ou até que a soma de suas cartas ultrapasse 21. A cada iteração o computador irá solicitar a decisão do jogador. Se o jogador humano pressiona 'd', uma carta é retirada do monte, seu valor é impresso na tela e em seguida ela é repassada à lista de cartas do jogador humano.

Se a soma das cartas do jogador ultrapassar 21, o computador imprimirá na tela "You lost, your score is above 21".

Abaixo encontra-se um exemplo de diálogo em que o jogador humano perde porque a soma de suas cartas ultrapassa 21.

```
Enter d for draw, s for stand: d
[D9] -- Score: 9
Enter d for draw, s for stand: d
[D9, D1] -- Score: 10
Enter d for draw, s for stand: d
[D9, D1, H7] -- Score: 17
Enter d for draw, s for stand: d
[D9, D1, H7, H11] -- Score: 27
You lost, your score is above 21
```

Teste o seu método várias vezes através de diferentes listas de cartas escolhidas aleatoriamente.

16. O computador

Se o jogador humano decide parar antes da soma de suas cartas ultrapassar 21, é a vez do computador jogar. Adicione um método `list<Card> computerPlayer(Deck d, int humanScore)` à classe `Blackjack`, que recebe como argumento o monte de cartas já utilizado pelo jogador humano e o score do jogador humano, e retorna a lista de cartas obtidas pelo computador.

A estratégia deste método para o computador é retirar cartas até que seu score atinja 21 ou ultrapasse o score do jogador humano. Quando ele para de pegar cartas, o método `computerPlayer` retorna a lista de cartas obtidas pelo computador.

Implemente agora o método que permita testar a função `computerPlayer`. Teste o seu método executando a função `computerPlayer` várias vezes, com diversos valores do parâmetro `humanScore`, depois imprima na tela os resultados obtidos.

17. Reuna tudo!

Adicione um método `void game()` à classe `Blackjack`. Este método deve inicializar o jogo (i.e., inicializar e embaralhar as cartas) e integrar os métodos `humanPlayer` e `computerPlayer` para jogar *blackjack*. Não esqueça que o computador só joga caso o score do jogador humano não ultrapasse 21.

Quando o jogador humano ganha o método deve imprimir na tela, por exemplo:

```
You won, your score is 21 while the computer scored 27
```

Ao contrário, quando o jogador humano perde deve ser impresso, por exemplo:

```
You lost, your score is 17 while the computer scored 18
```

Em caso de empate (ou seja, ambos obtiveram 21 pontos), será impresso:

```
It is a draw, both you and the computer scored 21
```

*Este trabalho prático é de autoria de Steve Oudot (Poly, France)