SCC-211
Lab. Algoritmos Avançados

Capítulo 4
Ordenação

João Luís G. Rosa

Ordenação em C

- C provê a função **`qsort`** que implementa o algoritmo *QuickSort* (**`stdlib.h`**)

```
#include <stdio.h>
#include <stdlib.h>

int values[] = { 40, 10, 100, 90, 20, 25 };

int compare (const void * a, const void * b) {
  return ( *(int*)a - *(int*)b );
}

int main () {
  int n;
  qsort (values, 6, sizeof(int), compare);
  for (n=0; n<6; n++)
     printf ("%d ",values[n]);
  return 0;
}
```

http://www.cplusplus.com

Ordenação em C

- C provê a função **`bsearch`** que implementa a busca binária (**`stdlib.h`**)

```
#include <stdio.h>
#include <stdlib.h>

int compareints (const void * a, const void * b) {
  return ( *(int*)a - *(int*)b );
}

int values[] = { 10, 20, 25, 40, 90, 100 };

int main () {
  int * pItem;
  int key = 40;
  pItem = (int*) bsearch (&key, values, 6, sizeof (int), compareints);
  if (pItem!=NULL)
    printf ("%d is in the array.\n",*pItem);
  else
    printf ("%d is not in the array.\n",key);
  return 0;
}
```

http://www.cplusplus.com

Ordenação em STL

- *C++ standard library* provê mais de um algoritmo de ordenação. Alguns dos principais são:
  - **`sort()`** – geralmente baseado no *quicksort*: *O(n log n)* na média e $O(n^2)$ no pior caso;
  - **`partial_sort()`** – geralmente baseado no *heapsort*. Tem pior caso *O(n log n)*, mas para a maior parte das entradas é de 2 a 5 vezes mais lento que o *quicksort*. Pode parar a ordenação quando os *k* primeiros elementos estão ordenados.
  - **`stable_sort()`** – geralmente baseado no *mergesort*. Preserva a ordem os elementos de mesmo valor (estável). Pode ter complexidade *O(n log n)* ou *O(n log n log n)*.

Ordenação em STL

- Os algoritmos de ordenação **`sort`** e **`stable_sort`** possuem as seguintes formas de chamada:
  - `void sort(RandomAccessIterator beg, RandomAccessIterator end)`
  - `void sort(RandomAccessIterator beg, RandomAccessIterator end,BinaryPredicate op)`
- Ordenam os elementos no intervalo [**`beg, end`**) com o operador <.
- op é um operador que pode substituir <.
- É necessário que o container tenha suporte a iteradores aleatórios.

Ordenação em STL: Exemplo 1

```
#include<vector>
#include<algorithm>
#include<functional>

using namespace std;

int main() {
        vector<int> v;
        int i;

        v.push_back(5);
        v.push_back(1);
        v.push_back(10);

        sort(v.begin(), v.end());

        for(i=0; i<v.size(); i++)
                printf("%d \n", v[i]);

        sort(v.begin(), v.end(), greater<int>());

        for(i=0; i<v.size(); i++)
                printf("%d \n", v[i]);
}
```

(Josuttis, 1999)

Ordenação em STL: Exemplo 2

```
#include<vector>
#include<algorithm>
#include<string>

using namespace std;

bool lessLength(const string& s1, const string& s2) {
        return s1.length() < s2.length();
}

int main() {
        vector<string> c1;
        vector<string> c2;
        int i;

        c1.push_back("1xxx");    c1.push_back("2x");
        c1.push_back("3x");      c1.push_back("4x");
        c1.push_back("5xx");     c1.push_back("6xxxx");
        c1.push_back("7xx");     c1.push_back("8xxx");
        c1.push_back("9xx");     c1.push_back("10xxx");
        c1.push_back("11");      c1.push_back("12");
        c1.push_back("13");      c1.push_back("14xx");
        c1.push_back("15");      c1.push_back("16");
        c1.push_back("17");
```

(Josuttis, 1999)

Ordenação em STL: Exemplo 2

```
        c2 = c1;

        sort(c1.begin(), c1.end(), lessLength);

        for(i=0; i<c1.size(); i++)
                printf("%s \n", c1[i].c_str());

        stable_sort(c2.begin(), c2.end(), lessLength);

        for(i=0; i<c2.size(); i++)
                printf("%s \n", c2[i].c_str());
}
```

(Josuttis, 1999)

Ordenação em STL: Exemplo 2

- **`Sort`**:

17
2x
3x
4x
16
15
13
12
11
9xx
7xx
5xx
8xxx
14xx
1xxx
10xxx
6xxxx

- **`Stable-sort`**:

2x
3x
4x
11
12
13
15
16
17
5xx
7xx
9xx
1xxx
8xxx
14xx
6xxxx
10xxx

Ordenação em STL

- O algoritmo de ordenação **`partial_sort`** pode interromper a ordenação quando os primeiros elementos (até **`sortEnd`**) estão ordenados.
  - `void partial_sort(RandomAccessIterator beg, RandomAccessIterator sortEnd, RandomAccessIterator end)`
  - `void partial_sort(RandomAccessIterator beg, RandomAccessIterator sortEnd, RandomAccessIterator end, BinaryPredicate op)`

Ordenação em STL: Exemplo 3

```
#include<vector>
#include<algorithm>

using namespace std;

int main() {
    vector<int> v;

    v.push_back(5); v.push_back(4);
    v.push_back(1); v.push_back(2);
    v.push_back(6); v.push_back(0);

    partial_sort(v.begin(), v.begin()+5, v.end());

    partial_sort(v.begin(), v.end(), v.end());
}
```

(Josuttis, 1999)

Exercício 3: Football aka Soccer

PC/UVa IDs: 110408/10194, Popularity: B, Success rate: average Level: 1

Football the most popular sport in the world (Americans insist to call it "Soccer", but we will call it "Football"). As everyone knows, Brazil is the country that have most World Cup titles (four of them: 1958, 1962, 1970 and 1994). As our national tournament have many teams (and even regional tournaments have many teams also) it's a very hard task to keep track of standings with so many teams and games played!

So, your task is quite simple: write a program that receives the tournament name, team names and games played and outputs the tournament standings so far.

Exercício 3: Football aka Soccer

A team wins a game if it scores more goals than its opponent. Obviously, a team loses a game if it scores less goals. When both teams score the same number of goals, we call it a tie. A team earns 3 points for each win, 1 point for each tie and 0 point for each loss.

Teams are ranked according to these rules (in this order):

- Most points earned.
- Most wins.
- Most goal difference (i.e. goals scored - goals against)
- Most goals scored.
- Less games played.
- Lexicographic order.

Exercício 3: Football aka Soccer

Sample Input

```
2
World Cup 1998 - Group A
4
Brazil
Norway
Morocco
Scotland
6
Brazil#2@1#Scotland
Norway#2@2#Morocco
Scotland#1@1#Norway
Brazil#3@0#Morocco
Morocco#3@0#Scotland
Brazil#1@2#Norway
```

Sample Output

```
World Cup 1998 - Group A
1) Brazil 6p, 3g (2-0-1),
   3gd (6-3)
2) Norway 5p, 3g (1-2-0),
   1gd (5-4)
3) Morocco 4p, 3g (1-1-1),
   0gd (5-5)
4) Scotland 1p, 3g (0-1-2),
   -4gd (2-6)
```

Exercício 3: Football aka Soccer

```
#include<iostream>
#include<map>
#include<vector>
#include<string>
#include<algorithm>

using namespace std;

struct ranking {
  string team;
  string team_lc;
  int points;
  int wins;
  int ties;
  int loses;
  int goals_against;
  int goals;
  int games;
};
```


Exercício 3: Football aka Soccer

```
int main() {
  vector<struct ranking> rank;            // source code edited
  map<string, int> m;

  cin >> nr_tests;
  cin.ignore(1);
  for (test = 0; test < nr_tests; test++) {
    getline(cin,tournament);
    cin >> nr_teams;
    cin.ignore(1);
    for (i = 0; i < nr_teams; i++) {
      getline(cin,r.team);
      str_lower(r.team, r.team_lc);       // lowercase convertion
      r.points = 0;
      r.wins = 0;
      r.ties = 0;
      r.loses = 0;
      r.goals_against = 0;
      r.goals = 0;
      r.games = 0;
      rank.push_back(r);
      m[r.team] = i;
    }
```

Exercício 3: Football aka Soccer

```
cin >> nr_plays;
cin.ignore(1);
for (i = 0; i < nr_plays; i++) {
  getline(cin, team1, '#');
  getline(cin, goal_str, '@');
  goals1 = atoi(goal_str.c_str());
  getline(cin, goal_str, '#');
  goals2 = atoi(goal_str.c_str());
  getline(cin, team2);
```


Exercício 3: Football aka Soccer

```
  rank[m[team1]].goals_against += goals2;
  rank[m[team2]].goals_against += goals1;
  rank[m[team1]].goals += goals1;
  rank[m[team2]].goals += goals2;
  rank[m[team1]].games++;
  rank[m[team2]].games++;
  if (goals1 == goals2) {
    rank[m[team1]].points++;
    rank[m[team2]].points++;
    rank[m[team1]].ties++;
    rank[m[team2]].ties++;
  } else {
    if (goals1 > goals2) {
      rank[m[team1]].points += 3;
      rank[m[team1]].wins++;
      rank[m[team2]].loses++;
    } else {
      rank[m[team2]].points += 3;
      rank[m[team1]].loses++;
      rank[m[team2]].wins++;
    }
  }
}
```

Exercício 3: Football aka Soccer

```
    sort(rank.begin(), rank.end(), lessrank);

    cout << tournament << endl;
    for (i = 0; i < nr_teams; i++)
      cout << i+1 <<") " << rank[i].team << " " << rank[i].points << "p, "
           << rank[i].games <<"g (" << rank[i].wins << "-" << rank[i].ties
           << "-" << rank[i].loses << "), "
           << rank[i].goals-rank[i].goals_against << "gd ("
           << rank[i].goals << "-" << rank[i].goals_against << ")" << endl;
    if (test < nr_tests - 1)
      cout << endl;
    m.clear();
    rank.clear();
  }
}
```


Exercício 3: Football aka Soccer

```
bool lessrank(const struct ranking &r1, const struct ranking &r2) {
    if (r1.points == r2.points) {
      if (r1.wins == r2.wins) {
        if (r1.goals-r1.goals_against == r2.goals-r2.goals_against) {
          if (r1.goals == r2.goals) {
            if (r1.games == r2.games) {
              return r1.team_lc < r2.team_lc;
            } return r1.games < r2.games;
          } else return r1.goals > r2.goals;
        } else return r1.goals-r1.goals_against > r2.goals-r2.goals_against;
      } else return r1.wins > r2.wins;
    } else return r1.points > r2.points;
}
```

Exercício: Stacks of Flapjacks (120)

Background

- Stacks and Queues are often considered the bread and butter of data structures and find use in architecture, parsing, operating systems, and discrete event simulation. Stacks are also important in the theory of formal languages.
- This problem involves both butter and sustenance in the form of pancakes rather than bread in addition to a finicky server who flips pancakes according to a unique, but complete set of rules.

Exercício: Stacks of Flapjacks (120)

The Problem

- Given a stack of pancakes, you are to write a program that indicates how the stack can be sorted so that the largest pancake is on the bottom and the smallest pancake is on the top. The size of a pancake is given by the pancake's diameter. All pancakes in a stack have different diameters.
- Sorting a stack is done by a sequence of pancake "flips". A flip consists of inserting a spatula between two pancakes in a stack and flipping (reversing) the pancakes on the spatula (reversing the sub-stack). A flip is specified by giving the position of the pancake on the bottom of the sub-stack to be flipped (relative to the whole stack). The pancake on the bottom of the whole stack has position 1 and the pancake on the top of a stack of *n* pancakes has position *n*.

Exercício: Stacks of Flapjacks (120)

- A stack is specified by giving the diameter of each pancake in the stack in the order in which the pancakes appear.
- For example, consider the three stacks of pancakes below (in which pancake 8 is the top-most pancake of the left stack):

8	7	2
4	6	5
6	4	8
7	8	4
5	5	6
2	2	7

- The stack on the left can be transformed to the stack in the middle via *flip(3)*. The middle stack can be transformed into the right stack via the command *flip(1)*.

Exercício: Stacks of Flapjacks (120)

The Input

- The input consists of a sequence of stacks of pancakes. Each stack will consist of between 1 and 30 pancakes and each pancake will have an integer diameter between 1 and 100. The input is terminated by end-of-file. Each stack is given as a single line of input with the top pancake on a stack appearing first on a line, the bottom pancake appearing last, and all pancakes separated by a space.

Exercício: Stacks of Flapjacks (120)

- **The Output**
  - For each stack of pancakes, the output should echo the original stack on one line, followed by some sequence of flips that results in the stack of pancakes being sorted so that the largest diameter pancake is on the bottom and the smallest on top. For each stack the sequence of flips should be terminated by a 0 (indicating no more flips necessary). Once a stack is sorted, no more flips should be made.

Exercício: Stacks of Flapjacks (120)

- **Sample Input**

```
1 2 3 4 5
5 4 3 2 1
5 1 2 3 4
```

- **Sample Output**

```
1 2 3 4 5
0
5 4 3 2 1
1 0
5 1 2 3 4
1 2 0
```

Referências

- Batista, G. & Campello, R.
  - Slides disciplina *Algoritmos Avançados*, ICMC-USP, 2007.
- Skiena, S. S. & Revilla, M. A.
  - *Programming Challenges – The Programming Contest Training Manual*. Springer, 2003.