Esboço da aula prática 1

Como compilar e linkar com o g++ em linha de comando:

```
$ g++ -std=c++11 -o output file_1.cpp file_2.cpp ... file_n.cpp
```

onde

-std=c++11 é um flag indicando suporte ao C++ 11 (dependendo do sistema, o suporte já é por default).
output é o nome do arquivo executável.
file_1.cpp ... file_n.cpp são os arquivos fonte C++.

Outros flags úteis:
-Wall exibe mensagens de advertência.
-O habilita otimizações.
-g habilita a geração de informações para depuração (compatível com o GDB). Nesse caso é recomendável também usar o flag -Og para otimizar a experiência de depuração.

Para mais informações, consultar o manual:

```
$ man g++
```

Em vez de usar compilação em linha de comando, podemor usar uma IDE multiplataforma como o QtCreator (http://qt-project.org). O QtCreator possui opções de autocompletar, integração com o depurador e capacidade de gerar arquivos makefile para Linux, Windows e Mac com suporte a diferentes compiladores (GCC, MinGW e MSVC).

No QtCreator, basta criar um novo projeto do tipo "console" e adicionar os arquivos .cpp e .h. Para que o compilador dê suporte a C++ 11, devemos alterar o arquivo .pro de modo que a linha:

```
CONFIG += console
```

torne-se

```
CONFIG += console c++11
```

O suporte ao C++ 11 é necessário para utilizarmos a biblioteca chrono (http://en.cppreference.com/w/cpp/chrono) na medição do tempo de execução de trechos de código (mas também vamos usar outras coisas interessantes do C++ 11 como a palavra-chave auto e o for(:) estilo foreach).
Um exemplo de uso das macros do arquivo timing.h é dado a seguir:

```
time_point ts = time_now(); // Tempo inicial
... // Seu código vai aqui
time_point te = time_now(); // Tempo final
int duration = itime_ns(te - ts); // Diferença entre o tempo inicial e final, em nanossegundos
```

O sufixo _ns em itime_ns pode ser substituído por _us (microssegundos), _ms (milissegundos), _s (segundos), _min (minutos), e _h (horas). Para resultados em double, usamos dtime_ns em vez de itime_ns.

Exercícios em aula:

ex1.cpp
1) Implementação da busca linear sobre um arranjo de 10 inteiros (em vez de usar um arranjo estático, usamos um "vector" que implementa um arranjo dinâmico em C++; essa estrutura será utilizada nas próximas aulas).

ex2.cpp
2) Implementação da versão template do código anterior.
Possíveis modificações: utilizar um vetor de strings e um vetor de pair<int, int> para mostrar a facilidade do uso de funções template.

ex3.cpp
3) Implementação da busca binária sobre o código anterior.

ex4.cpp
4) Fazer uma medição de tempo da busca linear e busca binária usando as macros de timing.h.

ex5.cpp
5) Criar um vetor de 100 mil elementos (inteiros no intervalo [1,100000]) e gerar uma cópia contendo uma permutação aleatória desses valores. Medir o tempo da busca linear e busca binária para 10000 buscas aleatórias sobre esses vetores.
Possível exercício: como poderíamos implementar nossa própria função "shuffle" de permutação aleatória?

ex6.cpp
6) Fazer buscas aleatórias (linear e binária) sobre as palavras do arquivo wordlist.txt (dicionário de mais de 400 mil palavras em inglês) e verificar o desempenho.

Para versões genéricas da busca linear e binária podemos usar as funções find e binary_search da biblioteca padrão "algorithm" do C++:
http://www.cplusplus.com/reference/algorithm/find/
http://www.cplusplus.com/reference/algorithm/binary_search/

A função binary_search só retorna um valor booleano. Para encontrar o iterador da chave encontrada, usa-se lower_bound (http://www.cplusplus.com/reference/algorithm/lower_bound/). Um exemplo:

```
template <class Iter, class T>
Iter bin_search (Iter first, Iter last, const T& val)
{
  first = std::lower_bound(first, last, val);
  if (first != last && !(val < *first)) // lower_bound usa ordem parcial estrita
    return first; // Encontrado
  else
    return last;  // Nao encontrado
}
```

Para converter um iterador em um índice de um vetor v, basta subtrair v.begin() do iterador.