1ª Aula Prática
Introdução aos Testes Unitários e Mooshak (e ao CLion)
Ciclos, invariantes e manipulação de inteiros, strings e vectores

O principal objetivo desta primeira aula prática é que fique a conhecer o modelo de testes unitários (usando GoogleTest) que será usado em todas as aulas (e também nos testes práticos), bem como o papel do Mooshak para automatizar a tarefa de submeter e verificar código. **No final desta aula deverá garantir que conseguiu fazer download do código dado, que conseguiu compilar e executar, que percebeu como estão a ser feitos os testes e que submeteu com sucesso o seu primeiro programa no Mooshak.**

Nota sobre o ambiente de desenvolvimento: iremos aconselhar a usar **CLion** como IDE nesta unidade curricular. Se preferir pode usar outro ambiente/editor sob a sua responsabilidade.

- CLion: A cross-platform IDE for C and C++ (installation guide) (free academic license)
- Compilador C++: Configure CLion on Windows *(em Linux podem simplesmente usar GCC)*

Instruções

- Faça download do ficheiro ***aed2223_p01.zip*** da página da disciplina e descomprima-o (contém a pasta ***lib,*** a pasta ***Tests*** com os ficheiros ***funWithCycles.cpp, funWithCycles.h*** e ***tests.cpp***, e os ficheiros ***CMakeLists*** e ***main.cpp***)
- No CLion, abra um projeto, selecionando a pasta que contém os ficheiros do ponto anterior
- Se não conseguir compilar, efetuar "Load CMake Project" sobre o ficheiro *CMakeLists.txt*
- Faça a sua implementação no ficheiro *funWithCycles.cpp*
- Note que apenas os testes da função `palindrome` (o 1º exercício) estão descomentados; deverá ir descomentando os outros testes à medida que vai fazendo os exercícios.

Os exercícios base desta aula envolvem a criação de funções estáticas da classe **`FunWithCycles`** e servem para "desenferrujar" o seu C++ e dar-lhe a oportunidade de exercitar o **uso de ciclos** (ex: *for* ou *while*), ao mesmo tempo que verifica se percebeu a noção de **invariante** dada nas aulas teóricas da semana anterior.

Excepto quando explicitamente dito, pode assumir que os casos de teste feitos aos seus programas são "pequenos" e que a correção é a sua principal preocupação (falaremos de eficiência na próxima aula prática).

1. **Palíndromos.** Função a implementar:

```
bool FunWithCycles::palindrome(const std::string & s)
```

Esta função deve devolver `true` se a string *s* for um palíndromo e `false` caso contrário. Uma string é um palíndromo se for exatamente igual ao seu inverso (lido de trás para a frente).

Exemplo de chamada e output esperado:

```
cout << FunWithCycles::palindrome("abba") << endl;
cout << FunWithCycles::palindrome("abcdba") << endl;
cout << FunWithCycles::palindrome("reviver") << endl;
cout << FunWithCycles::palindrome("revive") << endl;
```

```
1
0
1
0
```

Explicação: "abba" e "reviver" são palíndromos; *"abcdba"* e *"revive"* não são.

a) Comece por compilar e executar o projecto e verifique que existem testes que falham (familiarize-se com o interface); espreite em *funWithCycles.cpp* como o código apenas devolve `false` e por isso falha os casos onde devia dar `true` e espreite *test.cpp* para ver como os testes estão implementados.

b) Coloque o seguinte código dentro da função `palindrome` e verifique que agora os testes passam a estar corretos (*nota: existiam outras maneiras de fazer o exercício)*

```
unsigned length = s.size();
for (unsigned i = 0; i < length/2; i++)
   if (s[i] != s[length-i-1])
      return false;
return true;
```

c) Como foi explicado nas aulas teóricas, um **invariante** é uma condição que é sempre verdade imediatamente antes de cada iteração do ciclo e captura a essência do que o algoritmo em causa faz (é "o quê", enquanto que as instruções são o "como"), sendo que a sua veracidade no final do ciclo deve implicar necessariamente a correção do algoritmo. Consegue identificar neste pedaço de código anterior qual o invariante que é mantido?

d) Para mostrar a correção de um ciclo com base num invariante "basta" mostrar quatro coisas:

. **Inicialização:** o invariante é verdade antes do início da primeira iteração
. **Manutenção:** se for verdade no início de uma iteração, continua a ser verdade no início da próxima iteração
. **Terminação:** quando o ciclo termina, o invariante ser mantido implica que o algorito está correto
. **Progresso:** cada iteração leva-nos até mais próximo do final até eventualmente o ciclo terminar

Procure provar que este ciclo funciona tentando mostrar cada uma das 4 propriedades anteriores no código dado. Imagine uma instância (caso de *input*) que quebre cada um dos erros seguintes e tente compreender qual (ou quais) das 4 propriedades anteriores seria quebrada:

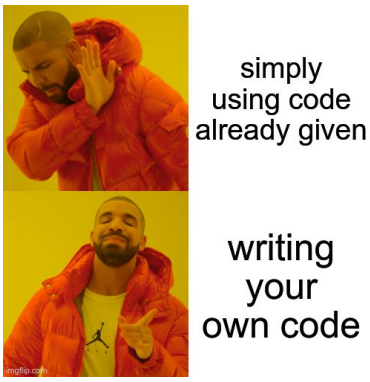
. Se em vez de `i = 0` o ciclo começasse com `i = 1`
. Se em vez de ir ate `i < length/2` fosse até `i < length/2-1`
. Se em vez de `s[i] != s[length-i-1]` fosse `s[i] == s[length-i-1]`
. Se em vez de `i++` tivéssemos `i--`

e) Submeta este exercício 1 no Mooshak. Use os dados de autenticação que recebeu por e-mail (se ainda não recebeu por favor contacte o seu docente das práticas e/ou *@Pedro Ribeiro* no Slack). Basta entrar em aula 1, selecionar o problema correto e submeter o ficheiro *funWithCycles.cpp*.

Verifique que obtém **Accepted** e 100 pontos e carregue no resultado para ver uma tabela de resultados mais detalhada. Experimente modificar o código com pequenos erros (como os 4 indicados anteriormente) e submeta novamente para verificar como o código passou a dar erros como **Wrong Answer** ou **Time Limit Exceeded**. Note também como carregando no número da sua submissão consegue recuperar o código submetido. Em cada problema existem vários testes com pontos associados (sendo que o máximo em cada problema é sempre de 100 pontos).

Nota: a submissão no Mooshak durante as aulas não é obrigatória, mas é fortemente aconselhada – para além de nos permitir a nós (e a vocês) ir seguindo o que estão a fazer, iremos sempre ter testes adicionais (para além dos colocados no zip das aulas) e assim ficam com um backup do vosso código. É também muito provável que se venha a usar o Mooshak para receber e processar as submissões nos testes práticos.

Agora que já conseguiu compilar, executar, perceber os testes unitários e submeter no Mooshak, podemos passar aos problemas "mais a sério" da aula



2. Frases Palíndromo. Função a implementar:

```
bool FunWithCycles::palindromeSentence(const std::string & s)
```

Esta função deve devolver `true` se a string *s* for uma **frase palíndromo** e `false` caso contrário.

Uma string é uma frase palíndromo se for um palíndromo quando ignorarmos todos os caracteres não alfabéticos (tais como espaços, pontuação e números) e se ignorarmos a capitalização das letras (por exemplo, 'a' deve ser considerado o mesmo caracter que 'A').

Exemplo de chamada e output esperado:

```
cout << FunWithCycles::palindromeSentence("Madam, I'm Adam") << endl;
cout << FunWithCycles::palindromeSentence("O bolo do   lobo") << endl;
cout << FunWithCycles::palindromeSentence("o bolo d lobo") << endl;
cout << FunWithCycles::palindromeSentence("Anotaram a data da maratona") << endl;
```

```
1
1
0
1
```

Explicação: "Madam, I'm Adam", "O bolo do lobo" e *"Anotaram a data da maratona"* são frases palíndromo, mas *"o bolo d lobo"* não é.

Sugestão: Comece por criar uma nova *string* contendo a frase apenas com letras minúsculas ("ignorando"), tudo o que não sejam letras e convertendo para minúsculas todas as letras (ex: "Madam, I'm Adam" ficaria "madamimadam"). Pode para isso usar os métodos contidos na biblioteca `<cctype>` (ver documentação no cpluplus.com ou no cppreference.com), que já está nos includes do seu projecto. Depois basta… chamar o método `palindrome` do exercício 1 com esta string que criou!

Nota: Não se esqueça de descomentar os testes unitários deste exercício para testar na sua máquina e quando acertar em todos deve também submeter no Mooshak para obter o seu (merecido) Accepted. Se usou um ciclo no código que criou, consegue identificar o invariante que acabou por usar?

3. Números interessantes.

O Pedro reparou que por baixo do seu portátil estava escrito o número 95726184. Curioso como é, não deixou de reparar que a soma dos seus dígitos era 42! De facto, 9+5+7+2+6+1+8+4=42, o que fez o Pedro pensar que este era um número mesmo "interessante" (pois 42 é o sentido da vida).

Felicíssimo, foi a correr ter com a Vanessa contar a sua descoberta. No entanto, o Luciano não achou a descoberta assim tão fascinante, pois achou que existiam mesmo muitos números com essa propriedade. O Pedro começou a contar a partir do 95726184 e de facto 9 números depois vem o 95726193, cuja soma dos dígitos também é 42. Mas nem sempre a distância é tão curta…

Depois de pensarem mais um pouco, o Pedro e a Vanessa acharam que podiam daqui criar um jogo para desafiarem a Ana, o António, a Filipa, o Luciano e o Rosaldo, que consiste em encontrar o primeiro número "interessante" maior que um dado número. Para impedir que simplesmente eles tentem decorar respostas, eles decidiram que estavam interessados em números cuja soma dos dos dígitos fosse um número também à escolha e não apenas 42.

Função a implementar:

```
int FunWithCycles::nextInterestingNumber(int n, int sum)
```

Esta função deve devolver o menor número maior que *n* cuja soma dos dígitos seja exatamente *sum*.

Restrições: neste exercício, em todos os testes feitos é garantido que $1 \leq n \leq 10^9$, que $1 \leq sum \leq 50$ e que o número a procurar nunca estará mais de 100 mil números acima de n.

Exemplo de chamada e output esperado:

```
cout << FunWithCycles::nextInterestingNumber(95726184, 42) << endl;
cout << FunWithCycles::nextInterestingNumber(424242, 42) << endl;
cout << FunWithCycles::nextInterestingNumber(1, 25) << endl;
cout << FunWithCycles::nextInterestingNumber(299, 13) << endl;
```

```
95726193
429999
799
319
```

Explicação: 95726193 é o menor número maior que 95726184 cuja soma dos dígitos é 42.
429999 é o menor número maior que 424242 cuja soma dos dígitos é 42.
799 é o menor número maior que 1 cuja soma dos dígitos é 25.
319 é o menor número maior que 299 cuja soma dos dígitos é 13.

Sugestão: Comece por implementar a função auxiliar `int FunWithCycles::digitSum(int n)` para que esta devolva o somatório dos dígitos de n (por emplo, `digitSum(123)` deveria devolver 1+2+3=6). Depois, basta fazer um ciclo que tenta todos os números consecutivos a seguir a n, parando quando encontrar um que tenha a soma dos dígitos desejada (usando a função auxiliar em cada número).

Nota: Como anteriormente, neste e nos próximos exercícios não se esqueça de descomentar os testes unitários deste exercício, de submeter no Mooshak e de tentar identificar o invariante em qualquer ciclo que tenha usado. Note também para simplificar a sua habituação, nesta aula apenas tem de submeter o ficheiro *funWithCycles.cpp* no Mooshak; numa aula futura será explicado como submeter mais do que um ficheiro, quando tal for necessário.

4. O Inverno está a chegar.

O Boromir reparou que as temperaturas estão a baixar em Gondor. Olhando para os últimos 10 dias, as temperaturas foram as seguintes:

`23 24 22 21 18 17 17 22 14 13`

Ele calculou as diferenças de temperatura entre dias seguidos, tendo obtido o seguinte resultado:

`+1 -2 -1 -3 -1 0 +5 -8 -1`

Boromir quis então perceber qual a maior sequência de dias seguidos com a temperatura sempre a descer. Para estes 10 dias, esta sequência tem tamanho 4 (`-2 -1 -3 -1`), sendo que existe uma outra sequência de tamanho apenas 2 (`-8 -1`).

Função a implementar:

```
int FunWithCycles::winter(vector<int> v)
```

Dado um vetor *v* com as temperaturas dos últimos dias, esta função deve devolver o maior comprimento de uma sequência consecutiva de descidas de temperatura,

Exemplo de chamada e output esperado:

```
cout << FunWithCycles::winter({23,24,22,21,18,17,17,22,14,13}) << endl;
```

```
4
```

Explicação: este exemplo foi explicado no enunciado do exercício.

Sugestão: não é preciso ter dicas para todos os problemas, pois não?

Exercícios Extra

Os exercícios seguintes permitem-lhe consolidar mais um pouco os seus conhecimentos e refrescar os conhecimentos já adquiridos em UCs anteriores (pode submeter no Mooshak).

5. Brincando com vetores.

a) Função a implementar:

```
int FunWithCycles::count(const vector<int> & v, int n)
```

Dado um vetor *v* e um inteiro *n*, esta função deve devolver o número de ocorrências de *n* no vetor.

Exemplo de chamada e output esperado:

```
cout << FunWithCycles::count({1,2,3,2,6,1,2,2}, 1) << endl;
cout << FunWithCycles::count({1,2,3,2,6,1,2,2}, 2) << endl;
cout << FunWithCycles::count({1,2,3,2,6,1,2,2}, 3) << endl;
cout << FunWithCycles::count({1,2,3,2,6,1,2,2}, 4) << endl;
```

```
2
4
1
0
```

Explicação: no vetor indicado existem:

. 2 ocorrências de 1
. 4 ocorrências de 2
. 1 ocorrência de 3
. 0 ocorrências de 4

b) Função a implementar:

```
bool FunWithCycles::hasDuplicates(const vector<int> & v)
```

Dado um vetor *v* deve devolver `false` se todos os inteiros do vetor ocorrerem só uma vez e `true` caso contrário (ou seja, se existir pelo menos um elemento em duplicado).

Exemplo de chamada e output esperado:

```
cout << FunWithCycles::hasDuplicates({1,2,3,2,6,1,2,2}) << endl;
cout << FunWithCycles::hasDuplicates({2,4,6,8,3,5,7}) << endl;
cout << FunWithCycles::hasDuplicates({10,20,30,10}) << endl;
```

```
1
0
1
```

Explicação: o primeiro e o terceiro vetor têm elementos em duplicado; o segundo vetor não tem.

c) Função a implementar:

```
void FunWithCycles::removeDuplicates(vector<int> & v)
```

Dado um vetor *v* remove todas as ocorrências duplicadas de um inteiro, mantendo apenas a primeira ocorrência de cada um.

Exemplo de chamada e output esperado:

```
vector<int> v = {1,2,3,2,6,1,2,2};
FunWithCycles::removeDuplicates(v);
for (int i : v) cout << i << " ";
cout << endl;
```

```
1 2 3 6
```

Explicação: as ocorrências indicas a vermelho estão duplicadas {1,2,3,**2**,6,**1**,**2**,**2**}

d) Função a implementar:

```
vector<int> FunWithCycles::merge(const vector<int> & v1, const vector<int> & v2)
```

Dados dois vetores *v1* e *v2* ordenados de forma crescente, devolve um novo vetor também ordenado de forma crescente contendo todos os inteiros de *v1* e *v2*.

Exemplo de chamada e output esperado:

```
vector<int> v = FunWithCycles::merge({1,5,5,7}, {2,3,6});
for (int i : v) cout << i << " ";
cout << endl;
```

```
1 2 3 5 5 6 7
```

Explicação: o vector resultante é {1,2,3,5,5,6,7} e está ordenado de forma crescente, contendo todos os elementos de {1,5,5,7} (a vermelho) e de {2,3,6} (a verde).

6. Brincando com números primos.

a) Função a implementar:

```
bool FunWithCycles::isPrime(int n)
```

Dado inteiro $n > 1$, esta função deve devolver `true` se o número é primo e `false` caso contrário. Um inteiro maior que 1 é primo se for apenas divisível por 1 e por si próprio.

Exemplo de chamada e output esperado:

```
cout << FunWithCycles::isPrime(13) << endl;
cout << FunWithCycles::isPrime(15) << endl;
cout << FunWithCycles::isPrime(49) << endl;
cout << FunWithCycles::isPrime(89) << endl;
```

```
1
0
0
1
```

Explicação: 13 e 89 são números primos; 15 e 49 não são.

b) Função a implementar:

```
vector<int> FunWithCycles::factorize(int n)
```

Dado um inteiro n > 1, esta função deve devolver um vetor contendo os números primos que constituem a fatorização do número. Os factores primos devem vir por ordem crescente.

Exemplo de chamada e output esperado:

```
vector<int> v1 = FunWithCycles::factorize(1176);
for (auto i : v1) cout << i << " ";
cout << endl;
vector<int> v2 = FunWithCycles::factorize(3037);
for (auto i : v2) cout << i << " ";
cout << endl;
```

```
2 2 2 3 7 7
3037
```

Explicação: 1176 = 2 * 2 * 2 * 3 * 7 * 7 e 3037 é um número primo

c) Função a implementar:

```
vector<int> FunWithCycles::listPrimes(int n)
```

Dado um inteiro n > 1, esta função deve devolver um vetor contendo todos os números primos menores ou iguais a *n*.

Exemplo de chamada e output esperado:

```
vector<int> v = FunWithCycles::listPrimes(100);
for (auto i : v) cout << i << " ";
cout << endl;
```

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Explicação: estão listados todos os primos menores ou iguais a 100

Para realmente calcular os primos passar no tempo limite no Mooshak neste problema (3s para calcular todos os primos até 10 milhões) não deve usar uma estratégia *naive*. Sugerimos que leia e implemente um Crivo de Eratóstenes.

Exercício de Desafio

7) Por vezes vamos colocar alguns desafios algoritmicamente mais complexos para aqueles que quiserem tentar algo mais desafiante. O desafio desta semana é resolver um problema que já resolveu (o exercício 3, o dos números interessantes) mas com testes maiores e tempo limitado, que impede uma solução *naive* itere sobre todos os inteiros consecutivos até descobrir a resposta.

Função a (re)implementar (versão desafio):

```
long long FunWithCycles::fastNextInterestingNumber(long long n, int sum)
```

Esta função deve devolver o menor número maior que *n* cuja soma dos dígitos seja exatamente *sum*.

Restrições: *n* pode ser tão grande quanto 10^{16} , *k* pode ir até 100 e a diferença entre o n e o próximo número cuja soma é sum pode ser tão grande quanto 10^{18}.

Limite de tempo no Mooshak: 1s para 1000 casos de teste

Como este é um problema de desafio, não vamos para já dar nenhuma dica, ficando à espera de ver os vossos programas

Se já tiverem feito tudo e estiverem "presos" neste, e quiserem mesmo fazer o desafio, podem contactar *@Pedro Ribeiro* no Slack para "dosear" as dicas, sabendo que este problema é substancialmente mais difícil que a versão "fácil" original (apesar da brincadeira do meme, este problema é possível de ser resolvido por todos vocês e não necessita de nenhum conhecimento avançado).