Ficha 4

Programação (L.EIC009)

Objectivos

- Uso de CMake e CLion
- Documentação de código com Doxygen
- Uso e programação de testes unitários expressos usando GoogleTest

Recursos

Slides das aulas teóricas: HTML PDF

0. Validação do ambiente

CLion e CMake

Na imagem Linux dos laboratórios o CLion está instalado no directório `/opt/clion-2021.2.1`.

Poderá ser conveniente configurar a variável de ambiente `PATH` da seguinte forma:

```
export PATH=/opt/clion-2021.2.1/bin:/opt/clion-2021.2.1/bin/cmake/linux/bin
```

(pode adicionar a linha acima ao final do ficheiro `.bashrc` para tornar a configuração activa cada vez que inicia uma linha de comando).

Depois de configurar `PATH` execute `cmake --version` na linha de comandos para verificar se o CMake está instalado. O CLion por sua vez pode ser lançado com `clion.sh`.

Doxygen

Verifique que o utilitário `doxygen` está instalado executando `doxygen --version`.

Pode instalar o `doxygen` em Linux no seu PC, ex. com o comando `sudo apt-get install doxygen` se usar Ubuntu. Pacotes para outros sistemas

operativos estão disponíveis aqui.

A documentação Doxygen pode de qualquer forma ser validada usando apenas o CLion.

Replit

Pode também usar o projecto no Replit - aqui.

1

- Baixe o arquivo ZIP `prog_p4.zip` e descomprima-o (`unzip prog_p4.zip`)
- Teste a compilação do projeto usando o `cmake`. Crie para o efeito um directório de "build", ex. `build`:

```
$ cd prog_p4
$ mkdir build
$ cd build
$ cmake ..
...
$ make
```

- Pode também testar a compilação do projecto usando o CLion
  - Aceda a `File > Open` selecione o ficheiro `CMakeLists.txt` e indique opção `Open as Project`.
  - Depois de aberto o projecto execute `Build > Build Project`.

1

Veja o "header file" `src/funcs.h` onde pode encontrar a declaração de várias funções.

Documente a função `days_in_month` com anotações Doxygen de forma análoga à função `hexstr_convert` (já documentada).

No directório de "build" execute

```
make doxdocs
```

para gerar a documentação Doxygen, que será colocada no directório `doc/html`.

Abra o ficheiro `index.html` com um browser (ex. Firefox ou Chrome).

No CLion pode validar a documentação focando a declaração da função com o rato.

2

Em `src/test_days_in_month.cpp` encontra uma "test suite" para a função `days_in_month()` - o programa gerado é `test_days_in_month` que pode invocar a partir da linha de comandos ou do CLion.

2.1

A "test suite" tem apenas um teste. Execute-a para ver os resultados.

2.2

Acrescente casos de teste:

- para um mês com 30 dias;
- para o mês de Fevereiro num ano bissexto;
- para o mês de Fevereiro num ano não bissexto;

Recompile e re-execute os testes de seguida.

2.3

Execute os testes em modo de "coverage" no CLion.

Em alternativa faça-o na linha de comando re-invocando o `cmake` num directório de build separado (ex. `covbuild`) e depois usando o script `covtest.sh`

```
$ mkdir covbuild
$ cd covbuild
$ cmake .. -DUSE_GCOV=1 -DCMAKE_C_COMPILER=gcc -DCMAKE_CXX_COMPILER=g++
...
$ make
...
$ chmod +x ../covtest.sh
$ ../covtest.sh ./test_days_in_month > cov.txt
```

Neste último caso, abra depois o ficheiro `funcs.c.gcov` para ver um relatório da cobertura dos testes. O ficheiro `cov.txt` dá por sua vez um relatório geral de cobertura global por funções.

2.3

Se precisar, acrescente mais casos de teste a `test_days_in_month.cpp` até atingir uma taxa de 100 % para cobertura de linhas.

- Inspecione a vista de Coverage no CLion.
- OU abra os ficheiros `cov.txt` e `funcs.gcov.c` gerado pelo uso do script `covtest.sh` no passo anterior.

2.4

Se precisar, acrescente mais casos de teste a `test_days_in_month.cpp` até atingir uma taxa de 100 % desta vez para cobertura de saltos.

3

Considere agora a "test suite" em `test_hexstr_convert.cpp`. O alvo dos testes é desta vez a função `hexstr_convert()`.

3.1

Adicione as configurações necessárias à inclusão do programa correspondente na "build" editando `CMakeLists.txt` (estas devem ser análogas às existentes em `test_days_in_month`).

Certifique-se que o programa compila correctamente.

3.2

Acrescente mais casos de teste a `test_hexstr_convert.cpp` até atingir uma taxa de 100 % para cobertura de saltos.

4

A função exponencial e^x pode ser aproximada por uma soma de n termos (resultantes da expansão em série de Taylor):

$$e^x \sim 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!} = \sum_{i=0}^{n} \frac{x^i}{i!}$$

- Adicione código para a função `exp_taylor()` em `funcs.c`.

- Adicione a "test suite" em `test_exp_taylor.cpp` à compilação (em `CMakeLists.txt`) e valide a sua implementação com a execução dos testes correspondentes. Note o uso da macro `EXPECT_NEAR()`.

5

Considere o código dado em `stack.h` e `stack.c` para uma stack implementada internamente com uma lista simplesmente ligada que vimos antes nas aulas.

5.1

Configure o projecto:

- Por forma a definir a biblioteca `stack` com código fonte `src/stack.c` - use `add_library` de form análoga a `funcs`.
- Adicione a "test suite" `test_stack.cpp` à compilação também.

5.2

Além do significado usual das funções já implementadas em `stack.c` as pretende-se que defina as operações (com código por definir):

- `stack_peek(s, v)`:
  - se stack não vazia: obtém elemento `v` no topo da stack **sem** o remover e finalmente retorna `true`;
  - caso contrário retorna `false`.
- `stack_to_array(s, a)`:
  - copia os elementos da stack para o array dado, começando pelo topo da stack na posição `0` do array e assim sucessivamente para os restantes elementos (o código da função pode assumir que o array dado tem espaço alocado suficiente para a cópia).

Valide o seu código com os testes fornecidos.

5.3

Acerte a configuração de `doxdocs` em `CMakeLists.txt` para incluir também `src/stack.h` na geração de documentação.

Documente (progressivamente) os tipos de dados e declarações de funções em `stack.h` e valide a documentação gerada.