Ferramentas de desenvolvimento

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

Projecto exemplo

Descrição

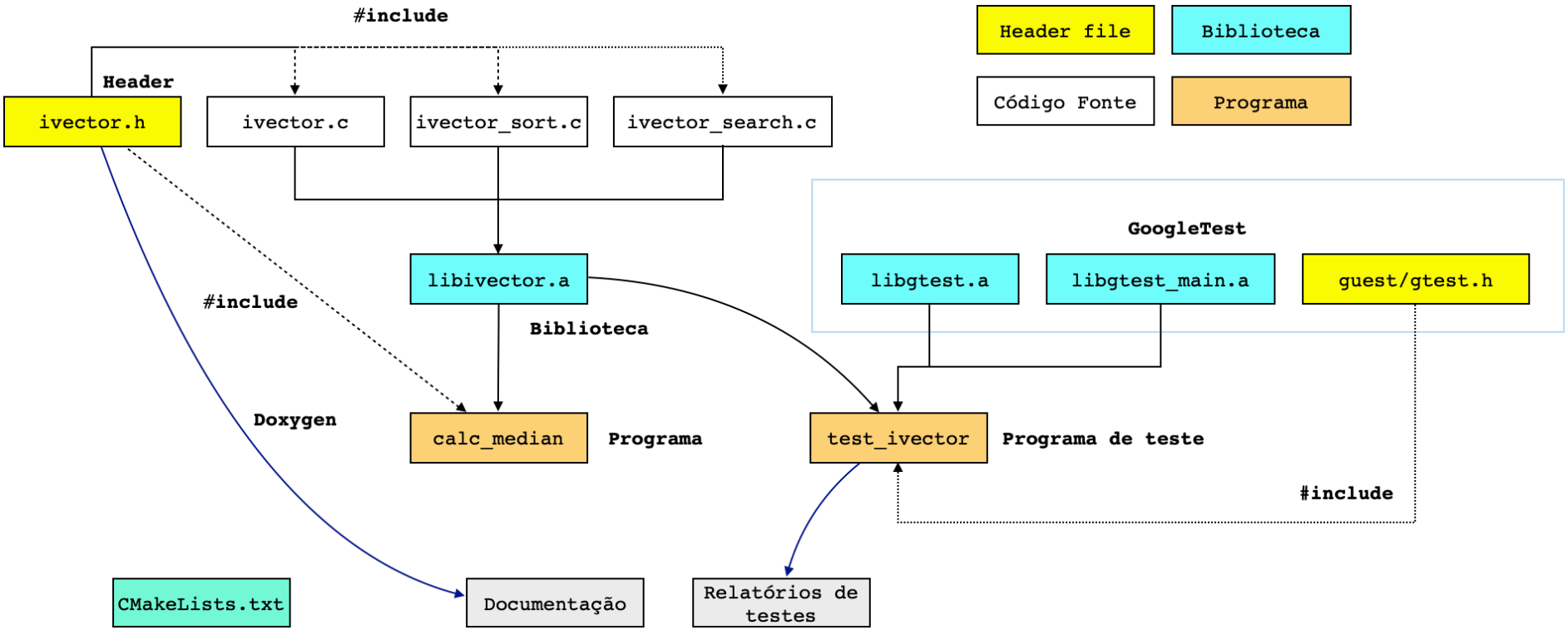
Componentes de um projecto organizado usando a ferramenta CMake:

- Biblioteca simples `ivector` (vector de inteiros, variante da Ficha 3).
- Programas que usam a biblioteca `ivector`.
- Documentação gerada usando Doxygen.
- Testes unitários usando a biblioteca GoogleTest.

Disponível aqui.

Vamos ilustrar o uso do projecto na linha de comando e também no ambiente do CLion.

Perspectiva geral



[→ abrir imagem completa em janela separada]

Organização

- `CMakeLists.txt`: configuração do projecto
- `ivector`: directório para a biblioteca `ivector`;
- `gtest`: directório para a biblioteca GoogleTest
- `programs`: alguns programas exemplo;
- `tests`: testes sobre a biblioteca.

Uso do CMake

CMake - introdução

O CMake é uma ferramenta "open-source" para desenvolvimento de programas.

Características essenciais:

- Automatização de tarefas usuais no desenvolvimento de software
  - compilação
  - testes
  - geração de documentação
  - preparação de ficheiros para distribuição
- Multi-plataforma
  - a mesma especificação base pode ser usada em conjunto com diversos compiladores e sistemas operativos
  - gera configurações para vários "build systems" (make, Ninja, Android Studio, Visual Studio, ...)

CMakeLists.txt

Um projecto CMake é normalmente descrito por uma pasta de ficheiros contendo na sua raíz um ficheiro de texto com nome `CMakeLists.txt`.

Fragmento do projecto exemplo:

```
cmake_minimum_required(VERSION 3.10)
project(prog_t4)
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 11)
# ...
include_directories(ivector)
add_library(ivector ivector/ivector.c
    ivector/ivector_search.c ivector/ivector_sort.c)
add_executable(calc_median programs/calc_median.c)
target_link_libraries(calc_median ivector)
```

Alguns comandos

`cmake_minimum_required(VERSION v)`

Identifica versão mínima `v` do CMake que deve estar instalada.

```
cmake_minimum_required(VERSION 3.10)
```

`project(name ...)`

Dá nome ao projecto.

Exemplo:

```
project(prog_t3)
```

`set(variable value)`

Define valor `value` para variável `variable`.

Exemplo:

```
set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 11)
```

Variáveis comuns - outros exemplos

```
set(COMMON_FLAGS
  "-Wall -g -fsanitize=address ...)
set(CMAKE_C_FLAGS ${CMAKE_C_FLAGS} ${COMMON_FLAGS})
set(CMAKE_CXX_FLAGS ${CMAKE_CXX_FLAGS} ${COMMON_FLAGS})
```

`CMAKE_C_FLAGS` e `CMAKE_CXX_FlAGS` definem as opções de compilação para C e C++ respectivamente.

Alguns comandos (cont.)

`include_directories(dir)`

Acrescenta `dir` para inclusão de "header" files (mapeada em opção `-Idir` do gcc).

Exemplo:

```
include_directories(ivector)
```

`add_library(name sources)`

Define a biblioteca `name` a partir da compilação de `sources`.

Exemplo:

```
add_library(ivector ivector/ivector.c
                    ivector/ivector_search.c
                    ivector_sort.c)
```

Alguns comandos (cont.)

`add_executable(name sources)`

Define o programa `name` a partir da compilação de `sources`.

Exemplo:

```
add_executable(calc_median programs/calc_median.c)
```

`target_link_libraries(name libraries)`

Define conjunto de bibliotecas `libraries` que são usadas (na fase de ligação) para o programa `name`.

Exemplo:

```
target_link_libraries(calc_median ivector)
```

Alguns comandos (cont.)

`add_subdirectory(subdir)`

Adiciona subdirectório `subdir` à build do projecto Este deverá conter um ficheiro `CMakeLists.txt` próprio.

Exemplo para o código da biblioteca GoogleTest e seu uso em programas de teste:

```
add_subdirectory(gtest)

include_directories(gtest/googletest/include)

add_executable(test_ivector tests/test_ivector.cpp)
target_link_libraries(test_ivector gtest gtest_main ivector)
```

Invocação do cmake

Devemos criar um directório para a build (por ex. subdirectório do projecto), e de seguida executar o utilitário `cmake` para gerar os ficheiros de "build":

```
$ ls
CMakeLists.txt
gtest
ivector
programs
tests
$ mkdir build
$ cd build
$ cmake ..
-- The C compiler identification is AppleClang 10.0.0.1000114
-- The CXX compiler identification is AppleClang 10.0.0.10001
...
-- Build files have been written to: /Users/edrdo/prog/homepag
```

Ficheiros gerados

Por omissão "o build system" a usar é o utilitário `make` que se baseia no uso de "Makefiles", caso o utilitário `make` esteja disponível no sistema. Note que é gerado um ficheiro `Makefile` (entre vários outros).

```
$ ls
...
Makefile
...
```

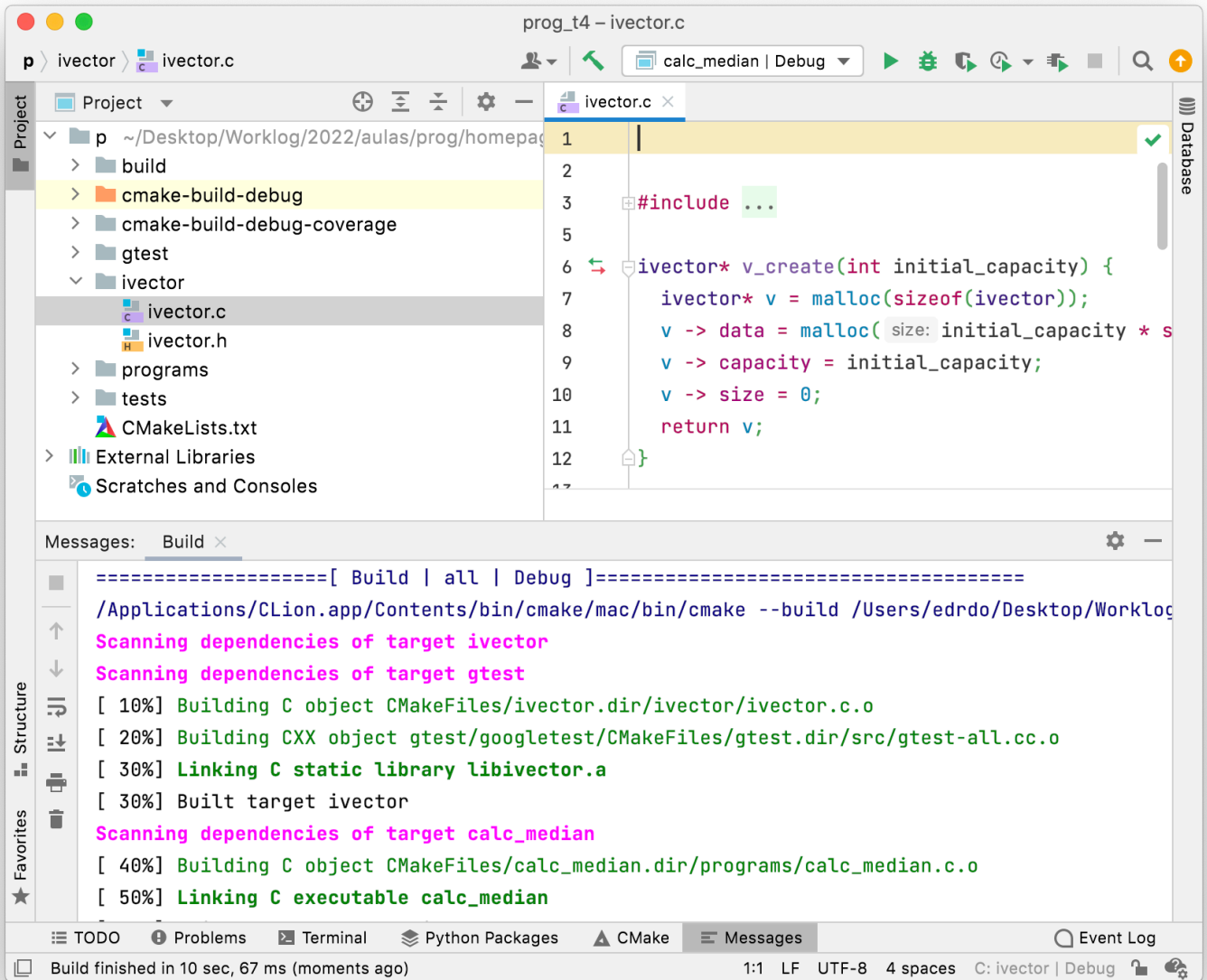
Compilando o projecto

Para compilar o projecto podemos agora executar `make`. Os ficheiros gerados ficaram no directório de build.

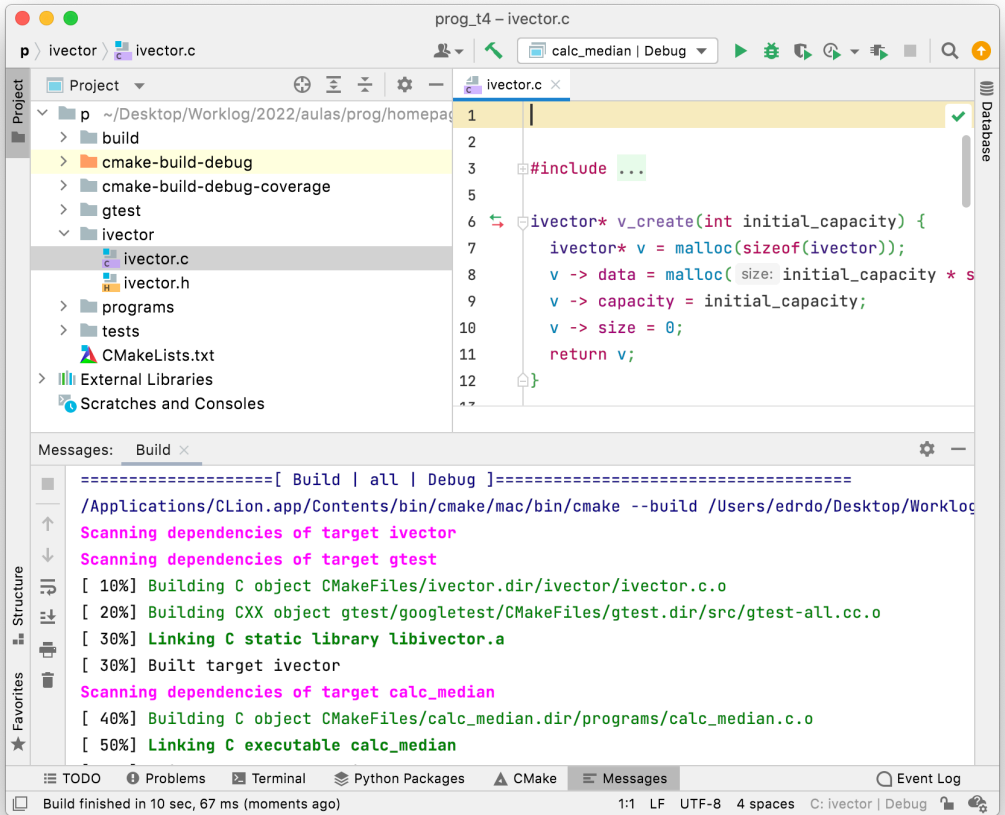
```
$ make
Scanning dependencies of target gtest
[ 10%] Building CXX object gtest/googletest/CMakeFiles/gtest.c
[ 20%] Linking CXX static library ../../lib/libgtest.a
[ 20%] Built target gtest
Scanning dependencies of target ivector
[ 30%] Building C object CMakeFiles/ivector.dir/ivector/ivecto
[ 40%] Linking C static library libivector.a
[ 40%] Built target ivector
...
Scanning dependencies of target calc_median
[ 90%] Building C object CMakeFiles/calc_median.dir/programs/c
[100%] Linking C executable calc_median
[100%] Built target calc_median
```

Uso do CLion

O CLion funciona em integração com projectos CMake.



Uso do CLion (cont.)



Instruções básicas:

- Aceda a *File* > *Open* e selecione o ficheiro `CMakeLists.txt`.
- Aceda a *Build* > *Build Project* para compilar o projecto.

Documentação de código usando Doxygen

Doxygen - introdução

O Doxygen é um sistema "open-source" para geração de documentação.

Aspectos fundamentais:

- Documentação gerada a partir de comentários com formatação especial no código fonte.
- A documentação pode ser gerada nos formatos HTML ou PDF (via Latex).
- Muito usado para C e C++. Também funciona para outras linguagens (Java, Python, ...).

Blocos de documentação

Vários formatos para blocos de documentação são aceites (referência. Por exemplo, blocos de documentação são definidos por comentários de linha simples iniciados com `//!` ou `///`:

```
//! Brief description.
//!
//! Continuation of description.
```

ou comentários multi-linha iniciados com `/*!` ou `/**`

```
/*!
  Brief description.

  Continuation of description.
*/
```

Exemplos - estruturas de dados

```
//! C struct for integer vector.
typedef struct {
    //! Internal array that grows as needed.
    int *data;
    //! Capacity of the internal array.
    int capacity;
    //! Number of elements.
    //! This corresponds to the used positions in the interna
    int size;
} ivector;
```

A documentação é inserida em "header files" como `ivector.h`, para posterior disseminação com a distribuição de uma versão do software.

Tipicamente os ficheiros de uma versão incluem apenas os "header files" necessários e binários (programas e bibliotecas) para ligação ou uso com software externo.

Exemplos - estruturas de dados (cont.)

Documentação gerada:

Public Attributes | List of all members

ivector Struct Reference

C struct for integer vector. More...

```
#include <ivector.h>
```

Public Attributes

int *	**data** Internal array that grows as needed.
int	**capacity** Capacity of the internal array.
int	**size** Number of elements. This corresponds to the used positions in the internal array.

Detailed Description

C struct for integer vector.

Exemplos - funções

```
//! Create an integer vector.
//!
//! \param initial_capacity Initial capacity for the vector.
//! \return A newly allocated integer vector.
ivector *v_create(int initial_capacity);


...
//! Add an element to the vector.
//!
//! \param iv Integer vector.
//! \param x Element to add.
void v_add(ivector *iv, int x);
```

Tags `\param` e `\return` usadas para documentação de argumentos e valor de retorno. Também podemos usar `@param` e `@return` em alternativa.

Exemplos - funções (cont.)

Documentação gerada:

Functions

ivector *	**v_create** (int initial_capacity)
void	**v_destroy** (**ivector** *iv)
int	**v_size** (const **ivector** *iv)
int *	**v_data** (**ivector** *iv)
void	**v_add** (**ivector** *iv, int x)
int	**v_get** (const **ivector** *iv, int pos)
void	**v_set** (**ivector** *iv, int pos, int value)
int	**v_search** (const **ivector** *iv, int value)
int	**v_bsearch** (const **ivector** *iv, int value)
void	**v_sort** (**ivector** *iv)

Exemplos - funções (cont.)

Documentação gerada:

◆ v_create()

ivector * v_create (int initial_capacity)

Create an integer vector.

Parameters

initial_capacity Initial capacity for the vector.

Returns

A newly allocated integer vector.

Exemplos - funções (cont.)

Documentação gerada:

◆ v_add()

```
void v_add ( ivector *  iv,
             int        x
           )
```

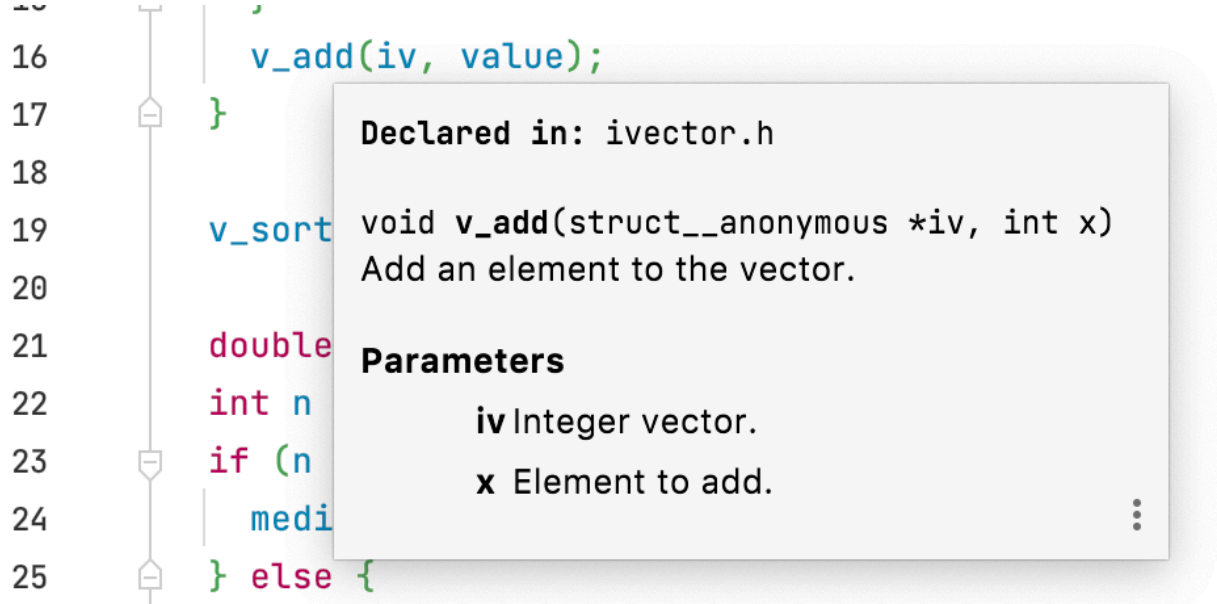
Add an element to the vector.

Parameters

iv Integer vector.

x Element to add.

Integração com CLion



O CLion tem suporte para anotações Doxygen e mostra a documentação de forma interactiva no contexto de edição de código.

Integração com CMake

O módulo FindDoxygen de integração com o CMake define o comando `doxygen_add_docs` para automatizar a geração de documentação.

O Doxygen tem de estar instalado previamente.

Forma geral:

```
find_package(Doxygen)
...
doxygen_add_docs(target_name sources)
```

Integração com CMake (cont.)

Exemplo de uso:

```
find_package(Doxygen)

if(DOXYGEN_FOUND)
    set(DOXYGEN_OUTPUT_DIRECTORY
        ${PROJECT_BINARY_DIR}/docs)
    doxygen_add_docs(documentation
       ${PROJECT_SOURCE_DIR}/ivector)
else(DOXYGEN_FOUND)
     message("Doxygen not found.")
endif(DOXYGEN_FOUND)
```

Integração com CMake (cont.)

Exemplo de uso:

```
$ cmake ..
...
-- Found Doxygen: /Applications/Doxygen.app/Contents/Resources
  (found version "1.9.2 (caa4e3de211fbbef2c3adf58a6bd4c86d0eb7
  found components: doxygen dot
...
$ make documentation
make documentation
[100%] Generate API documentation for documentation
Doxygen version used: 1.9.2
...
Running dot...
lookup cache used 6/65536 hits=36 misses=7
finished...
[100%] Built target documentation
```

Testes usando GoogleTest

Testes de software

- Projectos de software empregam tipicamente testes para validar o correcto funcionamento do código.
- **Teste de software**: procedimento que valida a execução de um item de software face a um comportamento esperado expresso por **asserções**. Diz-se que um teste **passa** quando assim é, e que **falha** caso contrário.
- **Testes unitários**: incidem sobre uma determinada "unidade" de software . A unidade pode ter várias granularidades, por ex. referir-se a uma única função simples, ou a uma API via sequência de chamadas de funções.
- GoogleTest é a biblioteca para escrita de testes de software que vamos empregar.

Caso de teste

Um caso de teste ("test case") sobre X, o **sujeito do teste** (ou **SUT, "subject under test**) é definido por:

1. Definição de dados de entrada (inputs).
2. Interações com X usando dados de entrada.
3. Asserções que validam os dados de saída (outputs) devolvidos e/ou estado de X após as interações.

Exemplo GoogleTest

```
#include <gtest/gtest.h>
#include <arrayalg.h>
...
TEST(binary_search, leftmost) {
    int a[] = { 1, 4, 5, 6, 7, 8 };
    int r = binary_search(a, 6, a[0]);
    EXPECT_EQ(0, r);
}
```

Sujeito de teste: função `binary_search`.

Inputs: argumentos da função `binary_search`.

Outputs: resultado de `binary_search`.

Asserção: igualdade (`r`) com o resultado esperado (`0` neste caso) como expresso pela asserção `EXPECT_EQ`.

Exemplo GoogleTest (cont.)

```
// test_binary_search.cpp
#include <gtest/gtest.h>
#include <arrayalg.h>
...
TEST(binary_search, leftmost) {
    int a[] = { 1, 4, 5, 6, 7, 8 };
    int r = binary_search(a, 6, a[0]);
    EXPECT_EQ(0, r);
}
TEST(binary_search, rightmost) {
    int a[] = { 1, 4, 5, 6, 7, 8 };
    int r = binary_search(a, 6, a[5]);
    EXPECT_EQ(5, r);
}
```

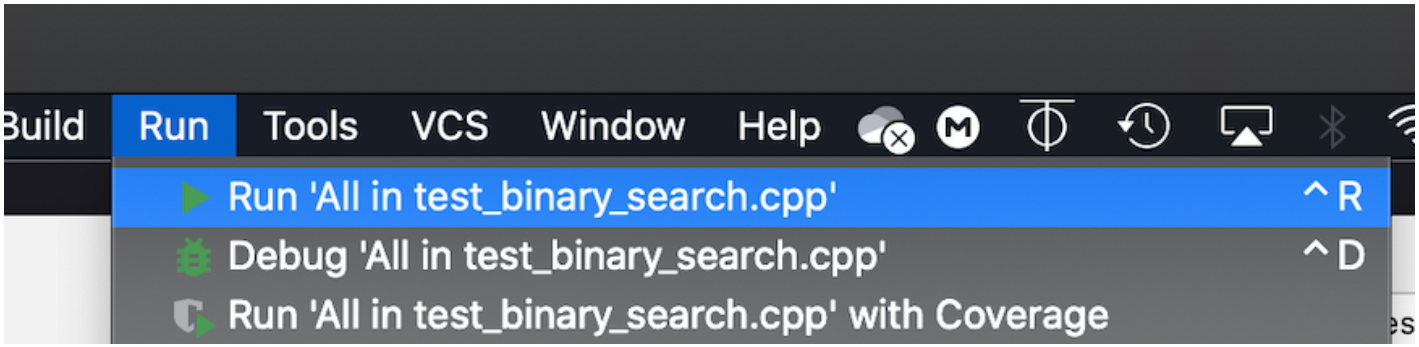
Vários casos de teste agrupados num programa de teste formam uma "**test suite**". Não existe qualquer ordem lógica para a execução dos casos de teste (que executam de forma independente uns dos outros).

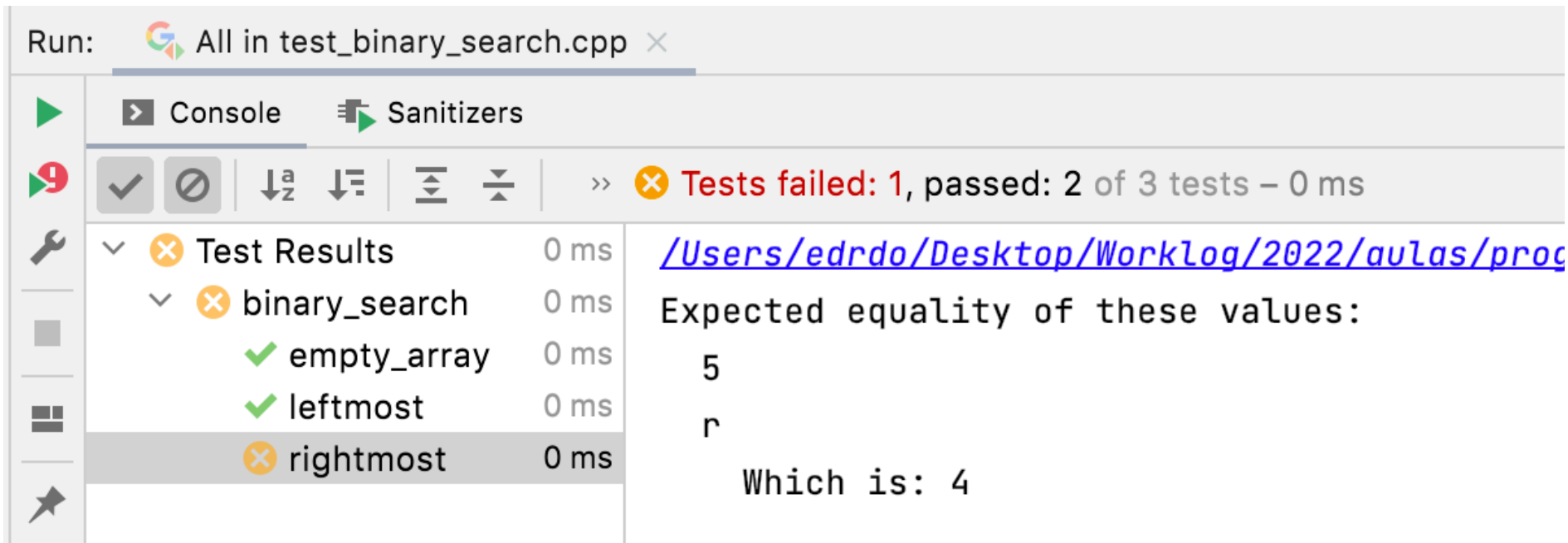
Exemplo GoogleTest (cont.)

Ao ser executado, uma "test suite" faz um relatório do sucesso ou falha por cada um dos caso de teste:

```
$ ./test_binary_search
[==========] Running 3 tests from 1 test suite.
...
[       OK ] binary_search.leftmost (0 ms)
[ RUN      ] binary_search.rightmost
/Users/edrdo/prog/homepage/teoricas/03/project/tests/test_bina
Expected equality of these values:
  5
  r
    Which is: 4
[  FAILED  ] binary_search.rightmost (0 ms)
[----------] 3 tests from binary_search (0 ms total)
...
[  PASSED  ] 2 tests.
[  FAILED  ] 1 test, listed below:
```

Execução de testes no CLion





GoogleTest - uso de C++

O ficheiro `test_binary_search.cpp` tem extensão `cpp`, indicando que se trata de código C++. A macro de pré-processador `TEST` "esconde" de forma simples a definição de código C++ relativamente verboso e repetitivo.

Para possibilitar a inter-operabilidade de C++ e C as definições de um "header" C devem estar incluídas num bloco `extern "C"`:

```
#ifndef __header_h__
#define __header_h__
#ifdef __cplusplus // C++ context defined?
extern "C" {
#endif
... // declarations
#ifdef __cplusplus
}
#endif
#endif
```

GoogleTest - macros de asserções

Uma asserção verifica se determinada condição é verdade. Um caso de teste é interrompido e dado como falhado quando uma asserção não se verifica.

Tipos simples de asserções:

Asserção	Valida que ...
`EXPECT_TRUE(condition)`	`condition` se verifica.
`EXPECT_FALSE(condition)`	`condition` não se verifica.
`EXPECT_EQ(expected, observed)`	`expected == observed`
`EXPECT_NE(expected, observed)`	`expected != observed`

Veja a documentação para vários outros tipos de asserções.

Cobertura de código

Para aferir o código que foi exercitado por testes de software, a compilação pode ser configurada para fornecer dados de **cobertura**.

Há duas medidas básicas de cobertura:

- **Cobertura de linhas ("Line Coverage")**: indicação das linhas de código exercitadas;
- **Cobertura de saltos ("Branch Coverage")**: mais refinado, condições de salto no código fonte, por ex. em relação a instruções de escolha (`if`, `switch-case`, ...) ou ciclo (`for`, `while`, ...).

Cobertura de código no projecto exemplo

Requisitos para uso linha de comandos Linux: `-DUSE_GCOV=1` na invocação de `cmake`; gcov instalado; execute `covtests.sh` (incluido no projecto).

```
$ cmake .. -DUSE_GCOV=1
$ make
$ ../covtests.sh ./test_binary_search
Running main() from /home/runner/progt3/project/gtest/googlete
[==========] Running 3 tests from 1 test suite.
...
Function 'binary_search'
Lines executed:100.00% of 10
Branches executed:100.00% of 6
Taken at least once:100.00% of 6
Calls executed:100.00% of 2
```

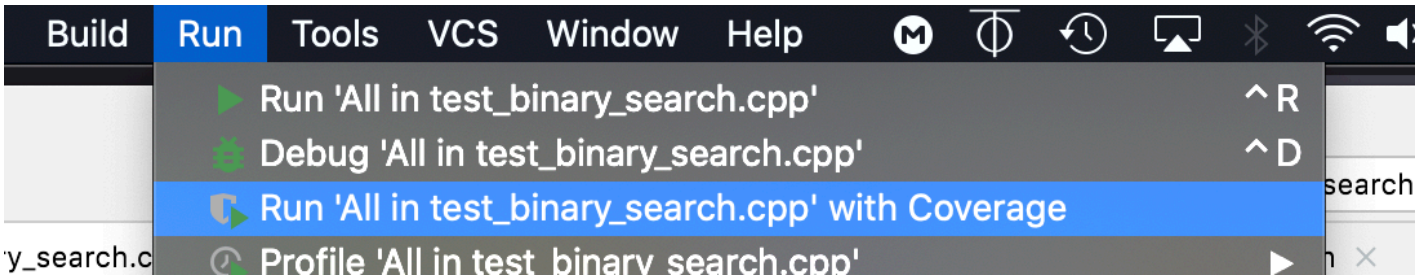
Cobertura de código no projecto exemplo (cont.)

Ficheiro `binary_search.c.gcov` gerado (entre vários outros):

```
function binary_search called 3 returned 100% blocks executed
        3:    4:int binary_search(const int array[], int nelem
        3:    5:  int left = 0, right = nelem -1;
        8:    6:  while (left <= right)  {
branch  0 taken 4
branch  1 taken 1 (fallthrough)
        4:    7:    int pos = left + (right - left) / 2;
        4:    8:    if (value > array[pos]) {
call    0 returned 4
branch  1 taken 1 (fallthrough)
branch  2 taken 3
        1:    9:      left = pos + 1;
        3:   10:    } else if (value < array[pos]){
call    0 returned 3
...
```

Cobertura de código no projecto exemplo (cont.)

No CLion:




```
1
2   #include <arrayalg.h>
3
4   int binary_search(const int array[], int nelem, int value) {
5     int left = 0, right = nelem -1;
6     while (left <= right)  {
7       int pos = left + (right - left) / 2;
8       if (value > array[pos]) {
9         left = pos + 1;
          Hide coverage   value < array[pos]){
11        right = pos - 1;
12      } else {
13        return pos;
14      }
15    }
16    return -1;
17  }
18
```

100% files, 91% lines covered, 57% branches covered i...

Element	Line C...	Branc...
arrayalg.h		
binary_search.c	91% l...	57% ...
linear_search.c		

Cobertura de saltos em mais detalhe

```
int binary_search(const int array[], int nelem, int value) {
  int left = 0, right = nelem -1;
  while (left <= right) /* c1 */  {
    int pos = left + (right - left) / 2;
    if (value > array[pos]) /* c2 */
      left = pos + 1;
    else if (value < array[pos]) /* c3 */
      right = pos - 1;
    else
      return pos;
  }
  return -1;
}
```

Que testes poderão ser necessários para atingir 100 % de cobertura de saltos?

Cobertura de saltos em mais detalhe (cont.)

Casos a cobrir para taxa de 100% para cobertura de saltos em ‘binary_search:

- c_1, : execução do teste atinge `left <= right` e verifica a condição
- $\neg c_1$: atinge `left <= right` e **não** verifica a condição
- c_2, $\neg c_2$: análogo, correspondentes a `value > array[pos]`
- c_3, $\neg c_3$: análogo correspondentes a `value < array[pos]`.

Cobertura de saltos em mais detalhe (cont.)

#	Input - `array,nElem,value`	Resultado esperado	Saltos cobertos
0	`NULL, 0, 1`	`-1`	$\neg c_1$
1	`{ 1 }, 1, 1`	`0`	$c_1, \neg c_2, \neg c_3$
2	`{ 1, 2, 3 }, 3, 3`	`2`	$c_1, c_2, \neg c_2, \neg c_3$
3	`{ 1, 2, 3 }, 3, 1`	`0`	$c_1, \neg c_2, c_3, \neg c_3$
4	`{ 1, 2, 3 }, 3, 0`	`-1`	$c_1, \neg c_1, \neg c_2, c_3$

Para cobertura de saltos a 100 % podemos considerar por exemplo os casos de teste 0 a 3 ou 1 a 4.

Testes com "fixture"

Uma "fixture" define procedimentos que são executados antes e depois de cada caso de teste numa "test suite" para acções de arranque e limpeza. Em `ivector_test.cpp`, que usa explicitamente C++ ... :

```
class ivtest : public testing::Test {
protected:
  ivector* iv;
  void SetUp() override {  iv = v_create(1); }
  void TearDown() override { v_destroy(iv); }
};
```

Sem entrar em detalhes sobre o uso de C++ em si, o objectivo é que:

- Variável `iv` fique acessível ao código de todos os casos de teste;
- `Setup()`, executado antes de cada caso de teste, chama `v_create` para alocar vector `iv` de fresco;
- `TearDown()`, executado depois de cada caso de teste independentemente de sucesso ou não, chama `v_destroy` para libertar memória associada a `iv`.

Testes com "fixture" (cont.)

```
TEST_F(ivtest, add) {
  v_add(iv, 1);
  v_add(iv, 2);
  v_add(iv, 3);
  int* data = v_data(iv);
  EXPECT_EQ(3, v_size(iv));
  EXPECT_EQ(1, data[0]);
  EXPECT_EQ(2, data[1]);
  EXPECT_EQ(3, data[2]);
}
```

Em testes com "fixture" emprega-se a macro `TEST_F` em vez de `TEST`.

O caso de teste valida a funcionalidade de `ivector` com várias chamadas a funções definidas para o tipo de dados.