Testing non-compiling code

Arno Lepisk

arno@lepisk.se

@arno_l f00ale

StockholmCpp 22-02-24

What do I mean by non-compilable code?

Code which at first sight appears correct

... but is forbidden by some constraint

Examples are in C++17

some things work in C++11 with some tweaks

Example 1 - Non-copyable types

```
class NoCopy {
  public:
    NoCopy();
    NoCopy(const NoCopy&) = delete;
    NoCopy & operator=(const NoCopy&) = delete;
};
```

How do we test the non-copy property?

Compile and check that it doesn't compile

```
void test_nocopy() {
  NoCopy obj;
  NoCopy copy = obj;
}
```

```
void test_noassign() {
  NoCopy obj;
  NoCopy copy;
  copy = obj;
}
```

Integration in build-system?

Write a script which wraps the compiler?

What about CI?

Wrap in CMake/CTest

```
function(add_fail_testcase NAME FILE)
  add_executable(${NAME} EXCLUDE_FROM_ALL ${FILE})
  add_test(NAME ${NAME}
      COMMAND ${CMAKE_COMMAND} --build
            ${CMAKE_BINARY_DIR} --target ${NAME})
  set_tests_properties(${NAME} PROPERTIES
                                WILL_FAIL On)
endfunction()

add_fail_testcase(test_name test_file.cpp)
```

Running

```
$ ctest
Test project /Users/arno/git/project/build
Start 1: test1
1/3 Test #1: test1 ...........  Passed    0.12 sec
Start 2: test2
2/3 Test #2: test2 ...........  Passed    0.10 sec
Start 3: test3
3/3 Test #3: test3 ...........  Passed    0.10 sec

100% tests passed, 0 tests failed out of 3

Total Test time (real) =   0.33 sec
```

Drawbacks

- Slow
- Need careful design of testcases

Can we do better?

Type traits!

```
#include <type_traits>

static_assert(!std::is_copy_constructible_v<NoCopy>);
static_assert(!std::is_copy_assignable_v<NoCopy>);
```

```
#define TEST(x) static_assert(x,#x)
```

```
#define TEST(x) EXPECT_TRUE(x) // gtest
```

```
TEST((!std::is_copy_constructible_v<NoCopy>));
```

pre C++17

```
#include <type_traits>

TEST(!std::is_copy_constructible<NoCopy>::value);
TEST(!std::is_copy_assignable<NoCopy>::value);
```

Check other properties

```
#include <type_traits>

TEST(std::is_default_constructible<MyClass>::value);
TEST(std::is_move_constructable<MyClass>::value);

TEST(std::has_virtual_destructor<MyClass>::value);
```

Example 2 - methods that shouldn't be callable

```
class MyClass {
  public: int my_ifc(int);
  protected: int internal_func();
};
```

How do we check that someone doesn't change `internal_func` to `public`?

std::is_invocable

```
std::is_invocable<Fn, Arg1, ...>
```

true if Fn can be called with arguments Arg1...

```
std::is_invocable_r<R, Fn, Arg1, ...>
```

Also checks return value

Usage

Notice: The use of `decltype` here does not work. Please use the method described under example 3 instead

```
TEST(
 std::is_invocable_r_v<int,
                       decltype(&MyClass::my_ifc),
                       MyClass*,
                       int>);
```

```
TEST(
 !std::is_invocable_v<
   decltype(&MyClass::internal_func),
   MyClass*>);
```

Example 3 - Non-existence of functions

```
class MyClass1 {
  public: void func();
};
class MyClass2 {
};
```

Helper to check if function is callable

```
class func_caller {
public:
  template<typename T>
  auto operator()(T && t) ->
    decltype(std::forward<T>(t).func());
};
```

```
TEST(std::is_invocable_v<func_caller, MyClass1>);
TEST(!std::is_invocable_v<func_caller, MyClass2>);
```

Example 4 - overloaded operators

```
TEST(std::is_invocable_v<std::plus<>, int, int>);
TEST(std::is_invocable_v<std::plus<>,
                         std::string, std::string>);

TEST(std::is_invocable_v<std::minus<>, int, int>);
TEST(!std::is_invocable_v<std::minus<>,
                         std::string, std::string>);
```

Add missing operator functor

```
struct lshift {
  template< class T, class U>
  constexpr auto operator()(T&& lhs, U&& rhs) const
   -> decltype(std::forward<T>(lhs) << std::forward<U>(rhs))
  {
    return std::forward<T>(lhs) << std::forward<U>(rhs);
  }
};
```

```
TEST(std::is_invocable_v<lshift, std::ostream&,
                                 int>);
TEST(!std::is_invocable_v<lshift, std::ostream&,
                                  MyClass>);
```

Things that do not work

- Things that use internal static asserts
- Calling things which cause warnings + `-Werror` (eg float -> int conversion)

Arno Lepisk

arno@lepisk.se / arno.lepisk@hiq.se

 @arno_l  f00ale