Фреймворк для юнит-тестов — конспект темы

Создаём альтернативу assert

Стандартный макрос assert:

- показывает, какая проверка не сработала в юнит-тесте и не показывает значения сравниваемых выражений;
- прерывает работу программы и не даёт проверять дальше;
- выводит информацию об ошибках в cout и смешивает её с выходными данными;
- зависит от макроса `NDEBUG`.

Чтобы получить значения сравниваемых выражений, нужна шаблонная функция `AssertEqual`. Она принимает два значения произвольного типа, сравнивает их и реагирует, если они неравны:

```
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;

template <typename T, typename U>
void AssertEqual(const T& t, const U& u) {
    if (t != u) {
        cout << "Assertion failed: "s << t << " != "s << u << endl;
        // Аварийно завершаем работу программы
        abort();
    }
}

int main() {
    string hello = "hello"s;
    AssertEqual(hello.length(), 5);

    // Эта проверка не сработает
    AssertEqual(2 + 2, 5);
}
```

`AssertEqual` принимает аргументы различных типов — это позволяет избежать проблем при операциях с целочисленными типами.

Параметр `hint` в `AssertEqual` выведет подсказку — она способствует поиску ошибок.

```
template <typename T, typename U>
void AssertEqual(const T& t, const U& u, const string& hint) {
    if (t != u) {
        cout << "Assertion failed: "s << t << " != "s << u << "."s;
        if (!hint.empty()) {
            cout << " Hint: "s << hint;
        }
        cout << endl;
        abort();
    }
}
```

Используем макросы и улучшаем фреймворк

Встроенные макросы:

`__FILE__` — вместо него препроцессор вставляет в текст программы имя текущего файла исходного кода;

`__LINE__` — вместо него препроцессор вставляет номер текущей строки;

`__FUNCTION_`_ — препроцессор заменяет его на имя текущей функции.

У макроса может быть один или несколько параметров, передаваемых в скобках. При обработке макроса фактические значения его параметров вставляются в исходный код. Если в теле макроса перед именем параметра поставить `#`, при раскрытии макроса вместо параметра появится строка, которая содержит его исходный код.

Макросы, меняющие одну последовательность символов на другую, объявляются директивой `#define`. Они способствуют выводу диагностической информации.

Объявим функцию `LogImpl`, которая принимает выводимую строку, имя функции, имя файла и номер строки исходного кода. Вот как макрос `LOG` может сделать код компактным:

```
#include <iostream>
#include <string>
```

```
using namespace std;

// Функция LogImpl выполняет основную работу
void LogImpl(const string& str, const string& func_name, const string& file_name, int line
_number) {
    cout << file_name << "("s << line_number << "): "s;
    cout << func_name << ": "s << str << endl;
}

// Макрос LOG используется для удобного вызова функции LogImpl
#define LOG(expr) LogImpl(#expr, __FUNCTION__, __FILE__, __LINE__)

int main() {
    // Функцию LogImpl можно вызывать напрямую, но это не очень удобно
    LogImpl("12345"s, __FUNCTION__, __FILE__, __LINE__);

    // Макрос LOG раскрывается в вызов функции LogImpl более удобно
    LOG(12345);
    LOG("hello"s);
    LOG(1 + 10);
}
```

Работаем с cerr

`cerr` (от англ. character error — ошибка символа) — поток вывода сообщений об ошибках. Работа с `cerr` идентична работе с `cout`.

Чтобы информация об ошибках выводилась в `cerr` и не смешивалась с выходными данными программы, замените `cout` на `cerr` в функциях `AssertEqualImpl`, `AssertImpl` и в тестирующих функциях.