組込みレガシーコード TDDレシピ

2019/06/14 組込みTDD勉強会

ご協力のお願い

本セッションではsli.doを利用したアンケートを元に進行させていただきます
ご協力いただける方は以下のURLから本セッションページへのアクセスをお願いいたします

URL: https://www.sli.do/

QRコード:

イベントID: #embtdd

このセッションのゴール

- テストがないコードに対してテストが書ける、というイメージを持つ
- テストが現場の課題を解くのに役立つ、というイメージを持つ

このセッションでお話しないこと

- 設計に関すること
- いわゆるテスト駆動開発(TDD)のお話

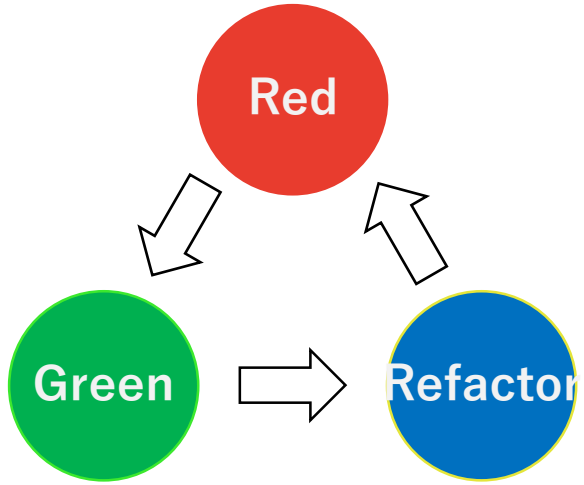
テスト駆動開発(TDD)？

テスト駆動とは、プログラム開発手法の一種で、プログラムに必要な各機能について、最初にテストを書き、そのテストが動作する必要最低限な実装をとりあえず行った後、コードを洗練させる、という短い工程を繰り返すスタイルである。

Wikipedia より

テスト駆動開発(TDD)のサイクル

1. 次の目標を考える
2. その目標を示すテストを書く
3. そのテストを実行して失敗させる（**Red**）
4. 目的のコードを書く
5. 2 で書いたテストを成功させる（**Green**）
6. テストが通るままでリファクタリングを行う（**Refactor**）
7. 1 ～ 6 を繰り返す



レガシーコード？

レガシーとは？

1　遺産。先人の遺物。

2　時代遅れのもの。「レガシーシステム」

[補説]本来、過去に築かれた、精神的・物理的遺産の意であるが、近年、「首相としてのレガシーを作る」のように、後世に業績として評価されることを期待した、計画中の事業の意でも用いられるようになった。

デジタル大辞泉（小学館）より

レガシーとは？

保守または拡張が困難な既存のプロジェクトなら、なんでも「レガシー」(legacy)と呼ぶことにしている。

Chris Birchall 著『レガシーソフトウェア改善ガイド』より

レガシーとは？

保守または拡張が困難な既存のプロジェクトなら、なんでも「レガシー」(legacy)と呼ぶことにしている。

Chris Birchall 著『レガシーソフトウェア改善ガイド』より

レガシーコードとは？

単にテストの無いコードである

マイケル・C・フェザーズ著 『レガシーコード改善ガイド』より

レガシーコードとは？

> テストの無いコードは悪いコードである。どれだけうまく書かれているかは関係ない。どれだけ美しいか、オブジェクト指向か、きちんとカプセル化されているかは関係ない。テストがあれば、検証しながらコードの動きを素早く変更することができる。テストがなければコードが良くなっているのか悪くなっているのかが本当にはわからない。

マイケル・C・フェザーズ著 『レガシーコード改善ガイド』より

こんなことはありませんか？

元の仕様がわからない

仕様について記載されたドキュメントが無い

変更のたびにどこかでエラーが起こる

仕様について記載されたドキュメントが古い

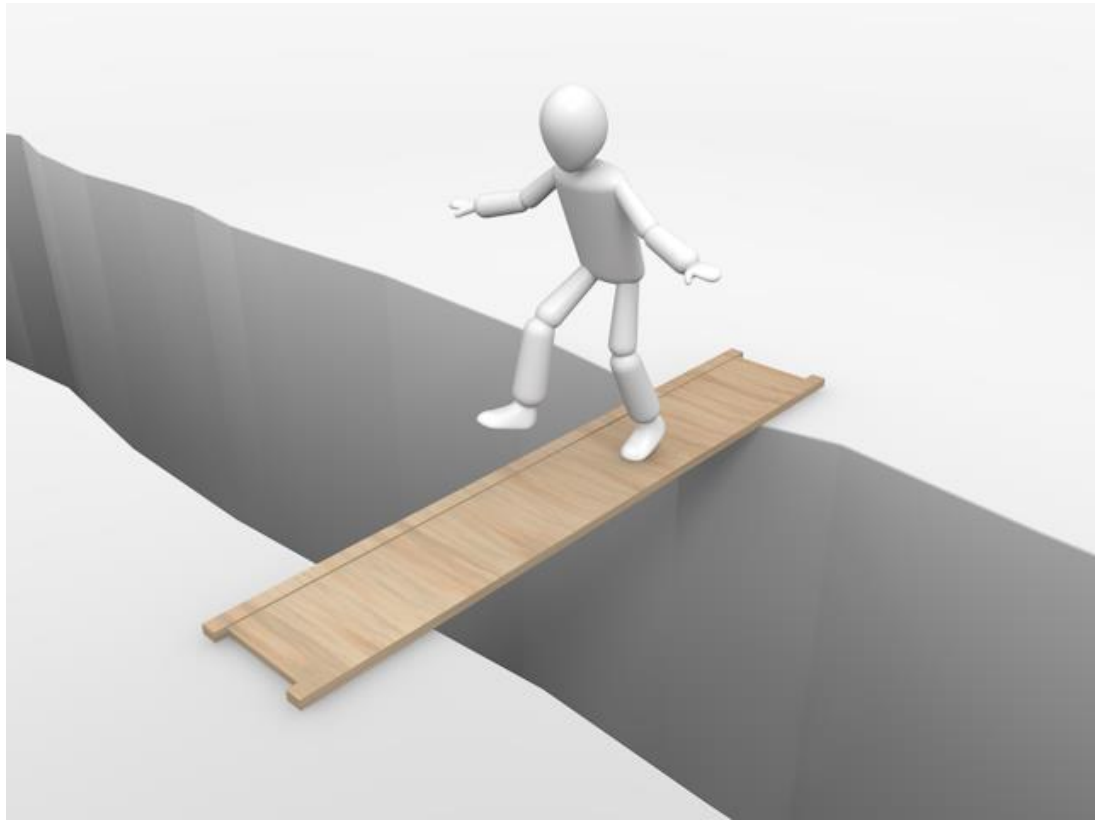
変更の影響範囲がわからない

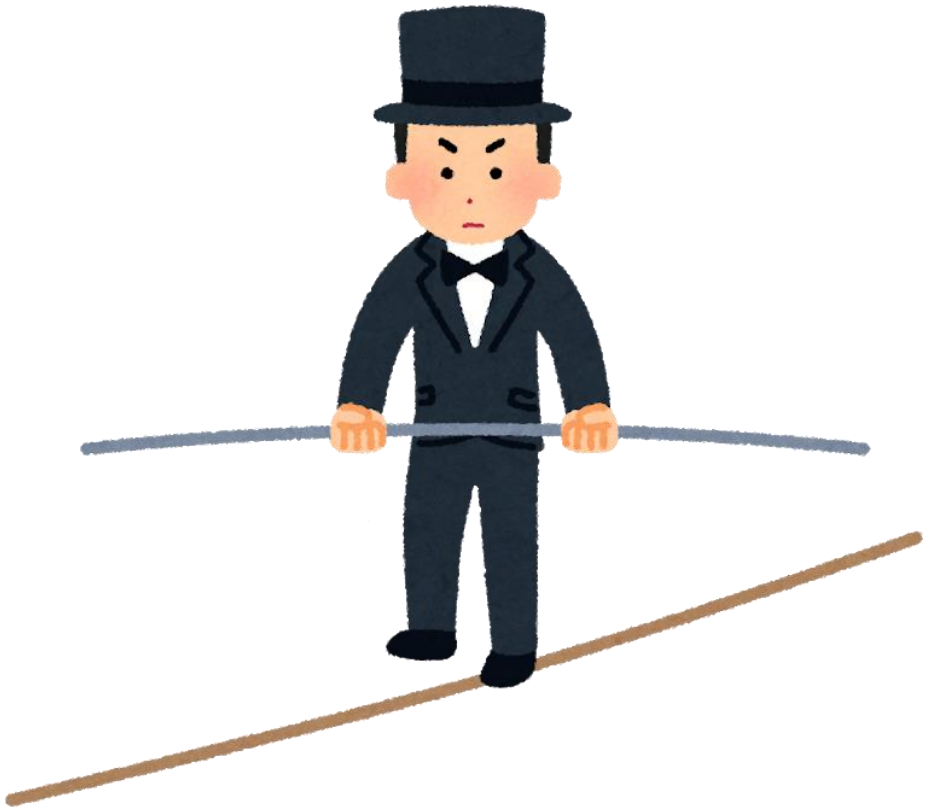
必要なのかわからないコードがある

ビルドの度にお祈りする

実行する度にお祈りする

URL: https://www.sli.do/　イベントID: #embtdd





さらに組込み現場では

アセンブラのコードが
埋め込まれている

人への依存

性能向上の為に練り込
まれたコードがある

OSへの依存

ハードウェアへの
依存

依存度が高くなる

バージョンごとの
互換性を保たなけ
ればならない

簡単に過去のソー
スを変更できない

ビルド～実行まで
に数時間かかる

動作を確認できる
環境が手元に無い

URL: https://www.sli.do/　イベントID: #embtdd

こんな問題解決できるのか？

URL: https://www.sli.do/　イベントID: #embtdd

全ては無理…

諦めるのはまだ早い

ここからレシピのご紹介

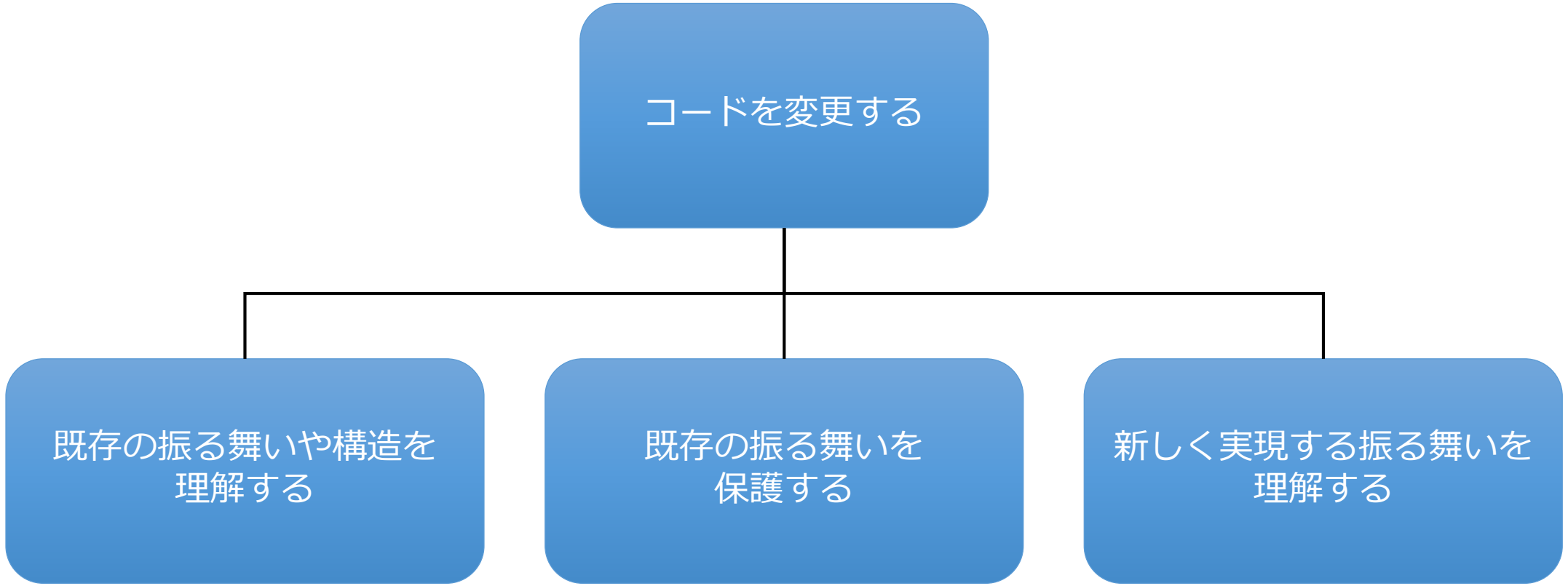
第１部：はじめの一歩

第２部：テストを書くときに役立つレシピ

レシピ

1. Privateをテストする
2. Stubを使う
3. テスティングフレームワークの機能を知る
4. 単体テストで不再現バグを予防する
5. テスティングフレームワークの特徴
6. IDEと仲良くする

既存コードに変更を入れる（1/2）



既存コードに変更を入れる（2/2）

既存の振る舞いや構造を理解する

仕様書や設計書を読む

コードを読む

既存の振る舞いを保護する

コードのどこを変更するか？

変更箇所が影響する機能はどの位あるか？

どんなテストをすれば良いか？

今回は自動テストを使って

「振る舞いや構造の理解」と「振る舞いの保護」を実現する

1. 仕様化テスト（1/2）

手順

1. テストハーネス（テストコード）の中で対象のコードを呼び出す
2. 失敗するとわかっている表明（アサート）を書く
3. 失敗した結果から実際の振る舞いを確認する
4. コードが実現する振る舞いを期待するように、テストを変更する
5. 以上の手順を繰り返す

マイケル・C・フェザーズ著 『レガシーコード改善ガイド』より
※（）内はこちらで入れた補足文になります

1. 仕様化テスト（2/2）

ポイント

- 仕様書がないケースでも使用出来る
- 実際のシステムの動きを文書化することに注力する※

※バグを見つけることを目的としない

2. ファーストテストの壁を超える

```
void レガシーコードに新たなテストを追加する()
{
  コンパイルを成功させる();

  リンクを成功させる();

  while (テストがクラッシュする()) {
    実行時の依存関係を見つける();
    実行時の依存関係を解決する();
  }
}
```

ジェームズ・W・グレニング著 『テスト駆動開発による組み込みプログラミング』より

2. ファーストテストの壁を超える

```
void レガシーコードに新たなテストを追加する()
{
  コンパイルを成功させる();

  リンクを成功させる();

  while (テストがクラッシュする()) {
    実行時の依存関係を見つける();
    実行時の依存関係を解決する();
  }
}
```

関数にデータ構造とパラメータを与える
(ヌルポインタやシンプルなリテラル値)
#include を追加する

ジェームズ・W・グレニング著 『テスト駆動開発による組み込みプログラミング』より

2. ファーストテストの壁を超える

```
void レガシーコードに新たなテストを追加する()
{
  コンパイルを成功させる();

  リンクを成功させる();

  while (テストがクラッシュする()) {
    実行時の依存関係を見つける();
    実行時の依存関係を解決する();
  }
}
```

プロダクトコードの一部をリンクする
スタブを使用する

ジェームズ・W・グレニング著 『テスト駆動開発による組み込みプログラミング』より

2. ファーストテストの壁を超える

```
void レガシーコードに新たなテストを追加する()
{
  コンパイルを成功させる();

  リンクを成功させる();

  while (テストがクラッシュする()) {
    実行時の依存関係を見つける();
    実行時の依存関係を解決する();
  }
}
```

デバッガフル活用！

ジェームズ・W・グレニング著 『テスト駆動開発による組み込みプログラミング』より

対象コードのご紹介

FreeRTOS-Sim：https://github.com/megakilo/FreeRTOS-Sim

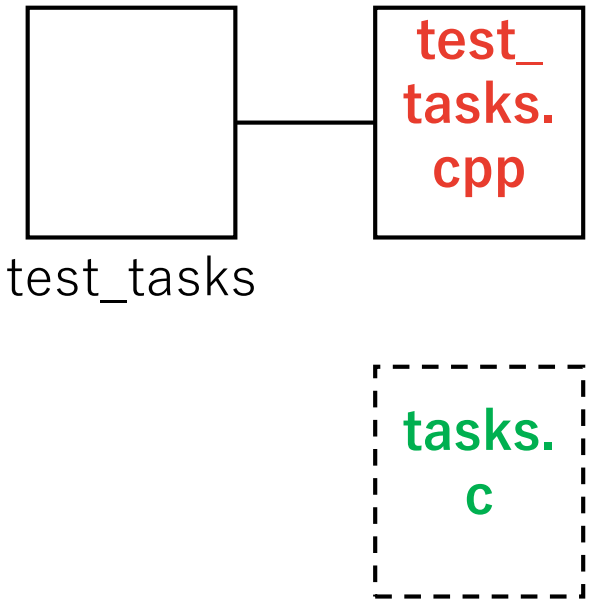
FreeRTOSをPOSIX環境で動作するようにしたシミュレータのコードを使用します

決してこのソースコードが修正が必要なコードというわけではありません
組込み現場に近く、仕様がわからないコードに対してテストを書くためのサンプルとして使用しています

開発環境

製品コード言語：C,　テストコード言語：C++

エディタ：Visual Studio Code、テスティングフレームワーク：CppUTest




```
# この後のデモで追加
# tasks.o: $(MAKETOP)/Source/tasks.c
#   $(CC) -c -o $@ $(CPPFLAGS) $(CFLAGS) $^
test_tasks.o: test_tasks.cpp
    $(CXX) -c -o $@ $(CPPFLAGS) $(CXXFLAGS) $^
test_tasks: test_tasks.o
    $(CXX) -o $@ $^ $(LDFLAGS)
```

■　製品コード

■　テストコード

URL: https://www.sli.do/　イベントID: #embtdd

次のテストをどう書くか？

次にどんなテストを書くか？と考えた場合に、
仕様がわかっている（ある程度想像出来る）ケースと
そうでないケースで洗い出し方が異なる。

今回は、仕様がわからないケースに対しての洗い出し方から見ていく

3. ToDoを洗い出す（1/2）

仕様がわからないケース

1. 変更対象の関数を特定する
2. 関数の戻り値と更新される値を洗い出す
3. 洗い出した値が変更される条件を洗い出す

3. ToDoを洗い出す（2/2）

仕様がわかっている（ある程度想像できる）ケース

「コードの振る舞いを理解できた」、

「コードの振る舞いを保護できた」と自信を持って言うために

必要なテストケースを出来るだけ多く洗い出す。

自動テストを使った場合の違い

やることは変わらない

既存の振る舞いや構造を理解する

仕様書や設計書を読む

コードを読む

既存の振る舞いを保護する

コードのどこを変更するか？

変更箇所が影響する機能はどの位あるか？

どんなテストをすれば良いか？

自動テストを使った場合の違い

既存の振る舞いや構造を理解する

仕様書や設計書を読む

コードを読む

やることは変わらない

既存の振る舞いを保護する

コードのどこを変更

やり方が変わった！

変更箇所が影響する機能はどの位あるか？

どんなテストをすれば良いか？

自動テストの効果

振る舞いや構造の理解

- 実際に動かしてみた結果を確認出来る
- デバッガなどを使って詳細な動きを見ることが出来る

振る舞いを保護する

- 回帰テストのコストを下げることが出来る
- テストケースが増えることで安心感が生まれる

そうは言っても...

実際に最初のテストを書くのは大変、時には数時間かかることも...

実践のポイント

- スコープを絞って1つずつクリアする
- モデルやスケッチ、仕様書を上手く利用する

レシピ

1. Privateをテストする
2. Stubを使う
3. テスティングフレームワークの機能を知る
4. 単体テストで不再現バグを予防する
5. テスティングフレームワークの特徴
6. IDEと仲良くする

レシピ１
Privateをテストする

Privateをテストする

- ソースをインクルードする

```
/* product.c */
static int local_val = 0;
static int foo(int param)
{
    return local_val = param;
}
```

```
#include "product.c“
TEST(FOO, BAR)
{
    TEST_EQUAL(0, local_val);
    TEST_EQUAL(1, foo( 1 ));
    TEST_EQUAL(1, local_val);
}
```

メリット

- 製品コードを汚さない
- コード量も少ない

デメリット

- リンク時の多重定義に注意が必要

Privateをテストする

- staticを消して外部参照可能にする①

```
# makefile
CFLAG += -Dstatic=
```

```
/* test.cpp */
extern "C"
{
    extern int bar(int);
}
```

メリット

- 製品コードを汚さない

デメリット

- 静的な意味で使っている部分に影響
- テスト側に参照定義が必要

Privateをテストする

- staticを消して外部参照可能にする②

```
/* product.h */
#undef  LOCAL
#ifdef  UNIT_TEST
#define LOCAL
#else
#define LOCAL static
#endif

#ifdef  UNIT_TEST
LOCAL int baz();
#endif
```

```
/* product.c */
#include ”product.h”
LOCAL int baz()
{
}
```

メリット

- 記憶クラスとスコープを区別できる

デメリット

- 既存の場合、製品コードを修正
- コード量は多め

レシピ２
Stubを使う

Stubを使う

そもそもは「できる限り本物のコードを使いたい」が、それが難しかったり色々な理由で代わりのものを使う事がある。

Stubを使う

Stubを使うシーン

- ハードウェアからの独立
- 生成するのが難しい入力
- 時間のかかるコラボレータを高速化する
- 変動するものへの依存
- 開発中のものへの依存
- 設定困難なものへの依存

Stubを使う

Stubのよくあるパターン

- リンカによる置き換え
- 関数ポインタによる置き換え
- プリプロセッサによる置き換え

Stubを使う

リンカによる置き換え

- 最も一般的なスタブのイメージ
  - テスト対象コード

```
1  void product_code(void)
2  {
3    int sum = sum_func(1, 2);
4    /* 以降なんらかの処理 */
5  }
```

  - 本物コード

```
1  int sum_func(int a, int b)
2  {
3    return a + b;
4  }
```

  - 試験用コード

```
1  int function1(int a, int b)
2  {
3    assert(a == 1);
4    assert(b == 2);
5  }
```

試験時のみ、本物コードではなく試験用コードをビルドしたオブジェクトとリンクすることでスタブへ置き換える

Stubを使う

関数ポインタによる置き換え

- 関数ポインタを引数に渡してコールすることで呼び出し対象の関数を外部に切り出す

- テスト対象コード

```
1  void target_func(void(*func)(int))
2  {
3    int hoge = 0;
4    /* hogeに対する処理 */
5    func(hoge);
6  }
```

- 試験用コード

```
1  void test_func(int a)
2  {
3    assert(a == 0);
4  }
5  void test()
6  {
7    target_func(&test_func);
8  }
```

Stubを使う

プリプロセッサによる置き換え

- gccの-includeオプションによってビルド時にファイルをインクルードさせるテクニック
- 呼び出しの履歴を取得するのに便利
- プリプロセッサによる置き換えによって実際とは異なるコードがテスト対象としてコンパイルされることに注意

Stubを使う

プリプロセッサによる置き換え

- テスト対象コード

  test.c

```
1  void product_code()
2  {
3    func();
4  }
```

本コードビルド時に下記オプションでビルドしstub.hをインクルード

gcc -c -include stub.h test.c

- 試験用コード

  stub.h

```
1  void call_log_func(void);
2
3  #define func() call_log_func(__FILE__, __LINE__)
```

stub.c

```
1  void call_log_func(char* file, int line)
2  {
3    printf('called func: %s:%d\n', file, line);
4    func();
5  }
```

レシピ３
テスティングフレームワークの機能を知る

テスティングフレームワークの機能を知る

- テスティングフレームワークとは？
  - テスト（期待値と結果の比較）を実行
  - テスト結果を見やすく表示

- CppUTestの場合

テストコード

```
CHECK_EQUAL(10, ret);
```

ターミナル

```
 $ ./test_tasks
 ..
 OK (2 tests, 2 ran, 2 checks, 0 ignored, 0 filtered out, 0 ms)
```

CppUTestの起動オプション

- -v (verbose): テスト結果の詳細（テスト名・実行時間）を表示

```
$ ./test_tasks -v
TEST(xTasksCreateTest, setPortMallocSuccess) -- 123 ms
TEST(xTasksCreateTest, setPortMallocFaild) - 21 ms

OK (2 tests, 2 ran, 2 checks, 0 ignored, 0 filtered out, 144 ms)
```

- いつ使う？
  - どんなテストがあるかざっと見たいとき
  - なんかテストに時間かかるようになったなーと思ったとき

CppUTestの起動オプション

- -n (name): 特定のテストだけ実行

```
$ ./test_tasks -v -n Success
TEST(xTasksCreateTest, setPortMallocSuccess) - 21 ms

OK (2 tests, 1 ran, 1 checks, 0 ignored, 1 filtered out, 21 ms)
```

- いつ使う？
  - デバッガを使って動きを解析するとき
- 関連
  - -sn：完全一致したテストだけ実行

CppUTestの起動オプション

- -p (process): プロセスを分離して実行

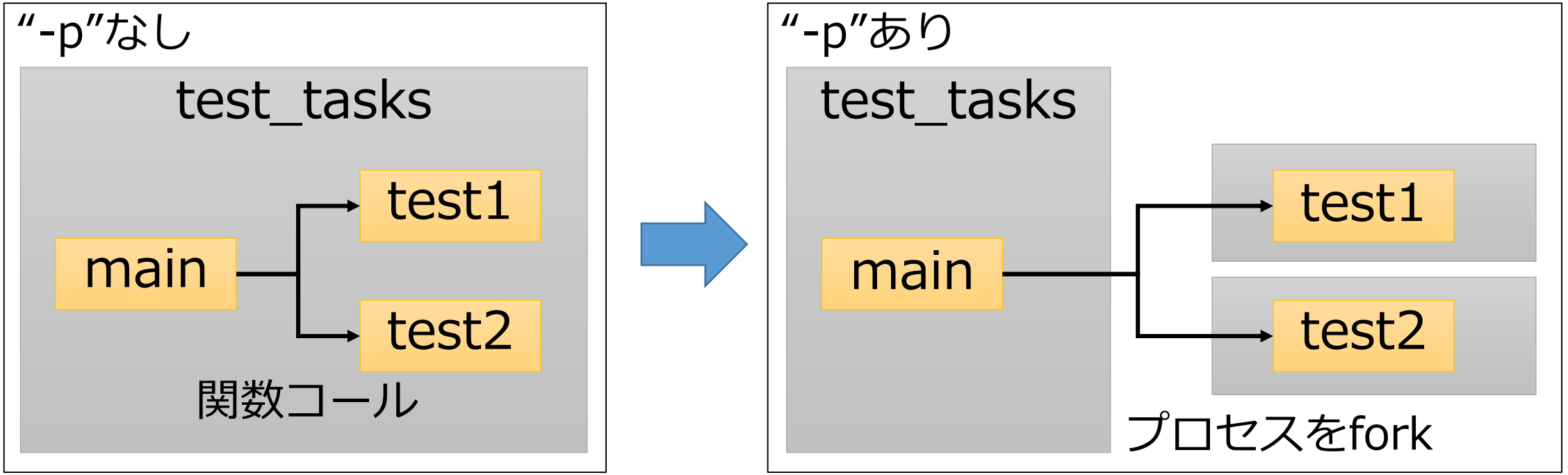
```
$ ./test_tasks  -p
..
OK (2 tests, 0 ran, 0 checks, 0 ignored, 0 filtered out, 5 ms)
```

  - ぱっと見変わらない？

CppUTestの起動オプション

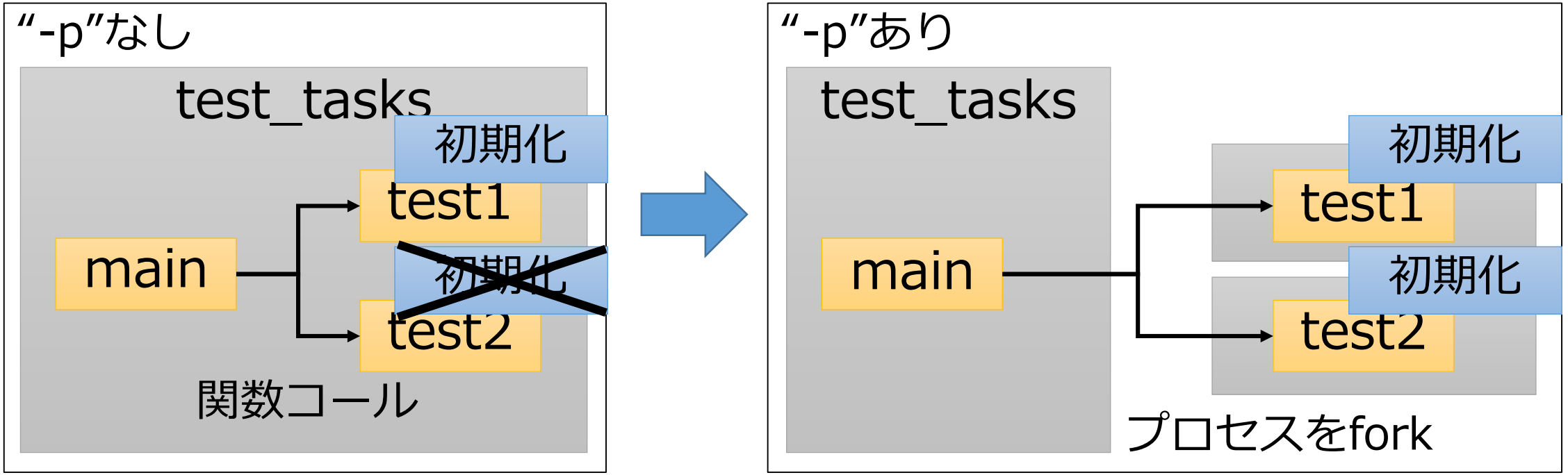
- -p (process): プロセスを分離して実行

```
$ ./test_tasks  -p
..
OK (2 tests, 0 ran, 0 checks, 0 ignored, 0 filtered out, 5 ms)
```



CppUTestの起動オプション

- いつ使う？
  - pthread_once（プロセスで一回だけ実行する仕組み）による初期化をテスト毎に実行したい



CppUTestのメモリリーク検出機能

- Makefileをちょこっと修正するだけで…

```
CFLAGS += -include /usr/include/CppUTest/MemoryLeakDetectorMallocMacros.h
```

- メモリリークを検出してくれる！

```
$ ./test_tasks

test_tasks.cpp:29: error: Failure in TEST(xTasksCreateTest, setPortMallocSuccess)
        Memory leak(s) found.
Alloc num (5) Leak size: 160 Allocated at: stub.c and line: 17. Type: "malloc"
        Memory: <0x211a1e0> Content:
 (略)
Alloc num (4) Leak size: 80 Allocated at: stub.c and line: 17. Type: "malloc"
        Memory: <0x211a130> Content:
 (略)
Total number of leaks:  2
 (略)
..
Errors (1 failures, 2 tests, 2 ran, 2 checks, 0 ignored, 0 filtered out, 0 ms)
```

CppUTestのメモリリーク検出機能

- ちょっとだけ中を見てみる
  - malloc/free, new/deleteをプリプロセッサで再定義

```
#define malloc(a) cpputest_malloc_location(a, __FILE__, __LINE__)
#define calloc(a, b) cpputest_calloc_location(a, b, __FILE__, __LINE__)
#define realloc(a, b) cpputest_realloc_location(a, b, __FILE__, __LINE__)
#define free(a) cpputest_free_location(a, __FILE__, __LINE__)
```

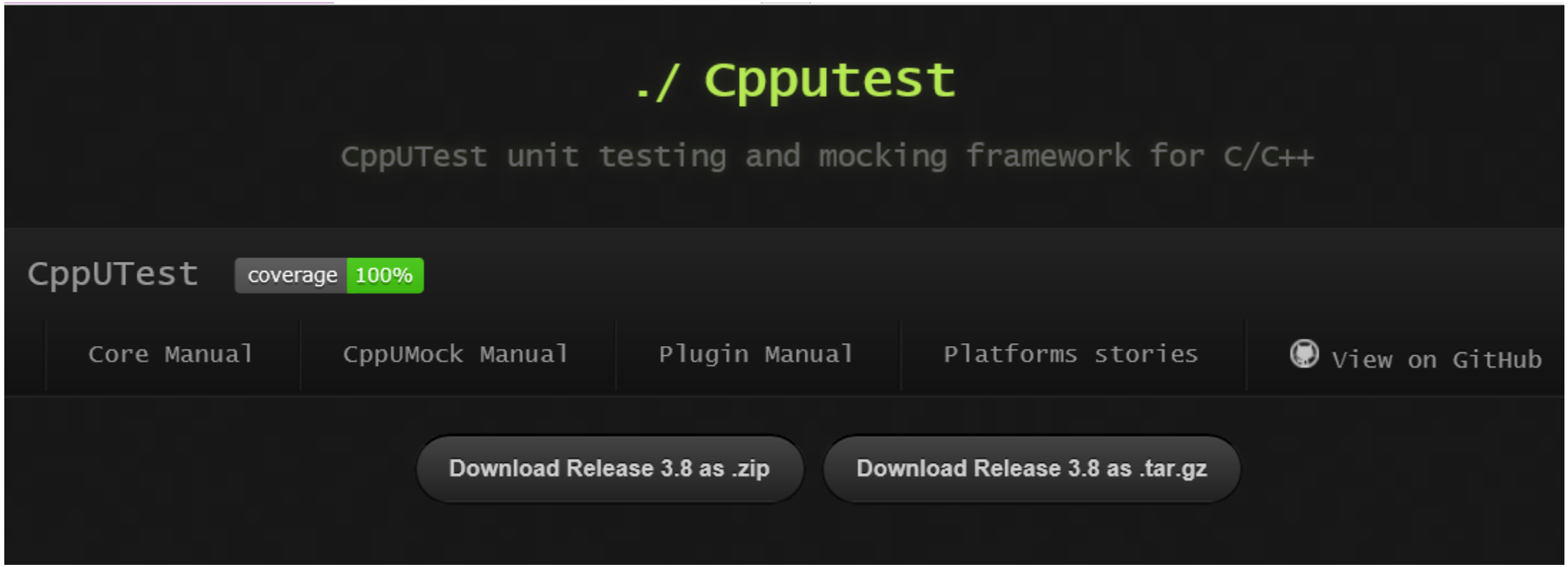
  - テスト実行前と後で、メモリ確保量を比較
    1. **メモリ確保量を計算**
    2. SetUp
    3. テスト実行
    4. TearDown
    5. **メモリ確保量が増えてないかチェック**

CppUTestのメモリリーク検出機能

- 注意点
  - コンパイル済みのオブジェクトでは使えない
    - →strdup(3)とか
  - 解放用のI/Fがないコードではfailしてしまう
    - →部分的にメモリリーク検出をON/OFFすることも可能

```
MemoryLeakWarningPlugin::turnOnNewDeleteOverloads();
MemoryLeakWarningPlugin::turnOffNewDeleteOverloads();
```

公式リファレンス大事



https://cpputest.github.io/manual.html

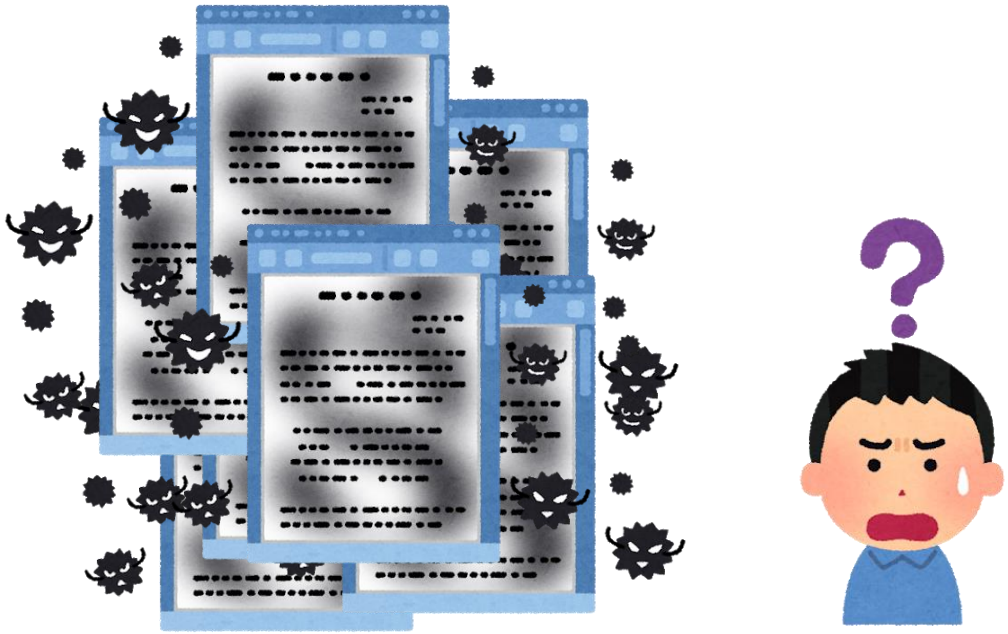
レシピ４
単体テストで不再現バグを予防する

単体テストで不再現バグを予防する

- 不再現バグの例
  - メモリの状態に依存
  - タイミングに依存
  - リソースが枯渇したときだけ発生

単体テストで不再現バグを予防する

- システムテストで不再現バグが発生！
  →再現しないので、
  　残されたログから推測するしかない。。。
  →解析難航



単体テストで不再現バグを予防する

- 単体テスト＋Valgrind
  - 不再現バグにつながるような不正な処理がないか、動的にチェックするツール
  - 単体テストと相性が良い

http://www.valgrind.org/

Valgrindは単体テストと相性が良い

結合テスト+Valgrind

- 大量のエラーでげんなり
- 連続動作する組込システムでは全部リソースリークと判定されてしまう
- Valgrindのオーバーヘッドで結果が変わる（そもそもまともに動かない）

単体テスト+Valgrind

- テストごとにチェックできるので解析が容易！
- テストには終了がある
  ⇒リークを判定できる
- 単体テストはタイミングに依存しない

Valgrindを使ってみる

- 単体テスト実行

```
$ ./test_tasks
..
OK (2 tests, 2 ran, 2 checks, 0 ignored, 0 filtered out, 1 ms)
```

Valgrindを使ってみる

- 単体テストをValgrindでチェック

```
$ valgrind ./test_tasks
==9721== Memcheck, a memory error detector
==9721== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==9721== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==9721== Command: ./test_tasks
==9721==
..
OK (2 tests, 2 ran, 2 checks, 0 ignored, 0 filtered out, 1 ms)
(中略)
==9721== ERROR SUMMARY: 0 errors from 0 contexts                    om 0)
```

Memory errorは
見つからなかった

検出例 ～二重free～

- チェック対象

```
01: #include <stdlib.h>
02:
03: int
04: main (void)
05: {
06:   char *ptr = malloc (1024);
07:   free (ptr);
08:   free (ptr);
09:   return 0;
10: }
```

free済ポインタを
freeしている

- Valgrind出力

```
$ valgrind ./test
Memcheck, a memory error detector
(中略)
Invalid free() / delete / delete[] / realloc()
   at 0x4C2EDEB: free
   by 0x400593: main (test.c:8)
 Address 0x5204040 is 0 bytes inside
     a block of size 1,024 free'd
   at 0x4C2EDEB: free
   by 0x400587: main (test.c:7)
 Block was alloc'd at
   at 0x4C2DB8F: malloc
   by 0x400577: main (test.c:6)
(中略)
ERROR SUMMARY: 1 errors from 1 contexts
```

検出例 ～free後にアクセス～

- チェック対象

```
01: #include <stdlib.h>
02:
03: int
04: main (void)
05: {
06:   char *ptr = malloc (1024);
07:   free (ptr);
08:   ptr[0] = 'a';
09:   return 0;
10: }
```

free済ポインタに
アクセスしている

- Valgrind出力

```
$ valgrind ./test
Memcheck, a memory error detector
(中略)
Invalid write of size 1
   at 0x40058C: main (test.c:8)
 Address 0x5204040 is 0 bytes inside
     a block of size 1,024 free'd
   at 0x4C2EDEB: free
   by 0x400587: main (test.c:7)
 Block was alloc'd at
   at 0x4C2DB8F: malloc
   by 0x400577: main (test.c:6)
(中略)
ERROR SUMMARY: 1 errors from 1 contexts
```

検出例 ～範囲外アクセス～

- チェック対象

```
01: #include <stdlib.h>
02:
03: int
04: main (void)
05: {
06:   char *ptr = malloc (1024);
07:   ptr[1024] = 'a';
08:   free (ptr);
09:   return 0;
10: }
```

確保したメモリ外にアクセスしている

- Valgrind出力

```
$ valgrind ./test
Memcheck, a memory error detector
(中略)
Invalid write of size 1
   at 0x400586: main (test.c:7)
 Address 0x5204440 is 0 bytes after
     a block of size 1,024 alloc'd
   at 0x4C2DB8F: malloc
   by 0x400577: main (test.c:6)
(中略)
ERROR SUMMARY: 1 errors from 1 contexts
```

検出例 ～メモリリーク～

- チェック対象

```
01: #include <stdlib.h>
02:
03: int
04: main (void)
05: {
06:   char *ptr = malloc (1024);
07:   return 0;
08: }
```

確保したメモリが
解放されていない

- Valgrind出力

option追加必要

```
$ valgrind --leak-check=full ./test
Memcheck, a memory error detector
(中略)
1,024 bytes in 1 blocks are definitely lost in
loss record 1 of 1
   at 0x4C2DB8F: malloc
   by 0x400537: main (test.c:6)
(中略)
ERROR SUMMARY: 1 errors from 1 contexts
```

検出例 ～fdリーク～

- チェック対象

```
01: #include <fcntl.h>
02:
03: int
04: main (void)
05: {
06:   int fd = open (".tmp",
                   O_CREAT, 0666);
07:   return 0;
08: }
```

openしたfdが
closeされていない

- Valgrind出力

option追加必要

```
$ valgrind --track-fds=yes ./test
Memcheck, a memory error detector
(中略)
FILE DESCRIPTORS: 4 open at exit.
Open file descriptor 3: .tmp
   at 0x4F31040: __open_nocancel
   by 0x400546: main (test.c:6)
(中略)
ERROR SUMMARY: 0 errors from 0 contexts
```

fdリークはエラー扱い
ではないので要注意

検出例 ～スレッド間ロック漏れ～

- チェック対象

```
01: #include <pthread.h>
02:
03: int g_count = 0;
04:
05: static void *
06: thread_start (void *arg)
07: {
08:   g_count++;
09:   return NULL;
10: }
11:
12: int
13: main (void)
14: {
15:   pthread_t tid0, tid1;
16:   pthread_create (&tid0, NULL, &thread_start, NULL);
17:   pthread_create (&tid1, NULL, &thread_start, NULL);
18:   pthread_join (tid0, NULL);
19:   pthread_join (tid1, NULL);
20:   return 0;
21: }
```

2スレッドが同期せずに read modify write

- Valgrind出力

option追加必要

```
$ valgrind --tool=helgrind ./test
Helgrind, a thread error detector
(中略)
Possible data race during read of size 4 at 0x60104C by thread #3
Locks held: none
   at 0x4006BE: thread_start (test.c:8)
   by 0x4C34DB6: ???
   by 0x4E476B9: start_thread (pthread_create.c:333)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
   at 0x4006C7: thread_start (test.c:8)
   by 0x4C34DB6: ???
   by 0x4E476B9: start_thread (pthread_create.c:333)
 Address 0x60104c is 0 bytes inside data symbol "g_count"
(中略)
ERROR SUMMARY: 2 errors from 2 contexts
```

Helgrind(thread error検出モード)実行時は、これまでのMemory errorは検出されないので要注意

検出例 ～スレッド間ロック順不正～

- チェック対象

```
01: #include <pthread.h>
02:
03: pthread_mutex_t mtx0 = PTHREAD_MUTEX_INITIALIZER;
04: pthread_mutex_t mtx1 = PTHREAD_MUTEX_INITIALIZER;
05:
06: int
07: main (void)
08: {
09:   /* Lock 0 -> 1 */
10:   pthread_mutex_lock (&mtx0);
11:   pthread_mutex_lock (&mtx1);
12:   pthread_mutex_unlock (&mtx1);
13:   pthread_mutex_unlock (&mtx0);
14:
15:   /* Lock 1 -> 0 */
16:   pthread_mutex_lock (&mtx1);
17:   pthread_mutex_lock (&mtx0);
18:   pthread_mutex_unlock (&mtx0);
19:   pthread_mutex_unlock (&mtx1);
20:   return 0;
21: }
```

ロック順序が一定でない

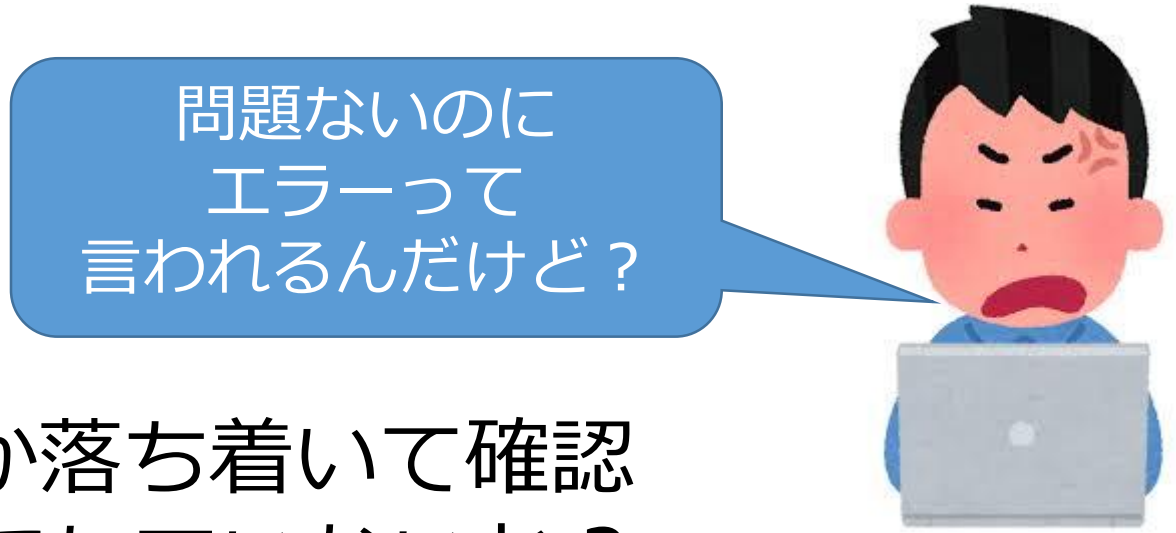
- Valgrind出力

option追加必要

```
$ valgrind --tool=helgrind ./test
Helgrind, a thread error detector
(中略)
Thread #1: lock order "0x601060 before 0x6010A0" violated
Observed (incorrect) order is: acquisition of lock at
0x6010A0
   by 0x40068B: main (test.c:16)
 followed by a later acquisition of lock at 0x601060
   by 0x400695: main (test:17)

Required order was established by acquisition of lock at
0x601060
   by 0x400663: main (test.c:10)
 followed by a later acquisition of lock at 0x6010A0
   by 0x40066D: main (test.c:11)
(中略)
ERROR SUMMARY: 1 errors from 1 contexts
```

Valgrindの誤検出（False Positive）



- 対策
  - まずは本当に問題ないか落ち着いて確認
    - 不必要に複雑な実装にしていないか？
  - エラー抑制の設定が可能
    - `--gen-supressions` ⇒ 設定ファイル作成
    - `--supressions` ⇒ 設定ファイル指定
    - テスティングフレームワーク内のエラーや、大人の事情で対処できないエラーの抑制にも

戻る

レシピ５
テスティングフレームワークの特徴

テスティングフレームワークの選び方

多くのテスティングフレームワークが存在しているため、
自分達の目的にあったものを選ぶことが重要。

選ぶ際には、以下のような点に気を付けると良い。

- テストをどこで実行するか？（PC or 実機）
- 何をテストしたいか？
- 誰がテストを書くか？（どんな言語を知っているか？）

テストを書く上であると良い機能

1. テストディスカバリ
書いたテストを自動的に実行してくれる

2. Fixture サポート
複数テストに共通する準備・後処理の実施、ヘルパー関数の定義

3. 2値比較可能なアサート
assert(expected == actual); ではなく、 assert(expected, actual) 形式
アサートはシンプルな記法で、かつ、失敗時の情報が多いことが大切

テストを更に活用するために

1. メモリリーク検出

2. 他ツール連携（Jenkins etc...）
テスト結果を所定の形式で出力して他のツールのインプットに出来る
テストを定期的に実行してレポートを表示したい、時などに有効

フレームワーク比較

	Unity	MinUnit	CppUTest	GoogleTest
言語	C	C	C/C++	C/C++
テスト ディスカバリ	×	×	○	○
Fixture の サポート	△	×	○	○
2値比較可能な アサート	○	×	○	○
メモリリーク 検出	×	×	○	×
他ツール連携	×	×	○	○
特徴	フットプリントが小さい。テストディスカバリ、Fixtureも部分的にサポートしている。	3行のソースコードで出来ている極小のフレームワーク。導入の敷居が低い。	C++の知識がなくても使用可能。 複数のOSプラットフォームをサポート。	C++の知識がある程度必要。機能豊富で日本語ドキュメントも充実。bitレベルでの比較は対応していない。

戻る

レシピ６
IDEと仲良くする

IDEと仲良くする（ 1/2 ）

目的

テストコードを書く、リファクタリングをする

⇒ これまでより実装量が増える

素早く、安全に実装を進めるために、IDEの機能をフル活用する

IDEと仲良くする（ 2/2 ）

手順

1. キーボードショートカットを知る
2. IDEで出来ることを知る
   - スニペット
3. IDEで出来ないことをスキルとして身に付ける

参考：和田 卓人 (監修), Kevlin Henney (編集), 夏目 大 (翻訳)
『プログラマが知るべき97のこと』

キーボードショートカット抜粋

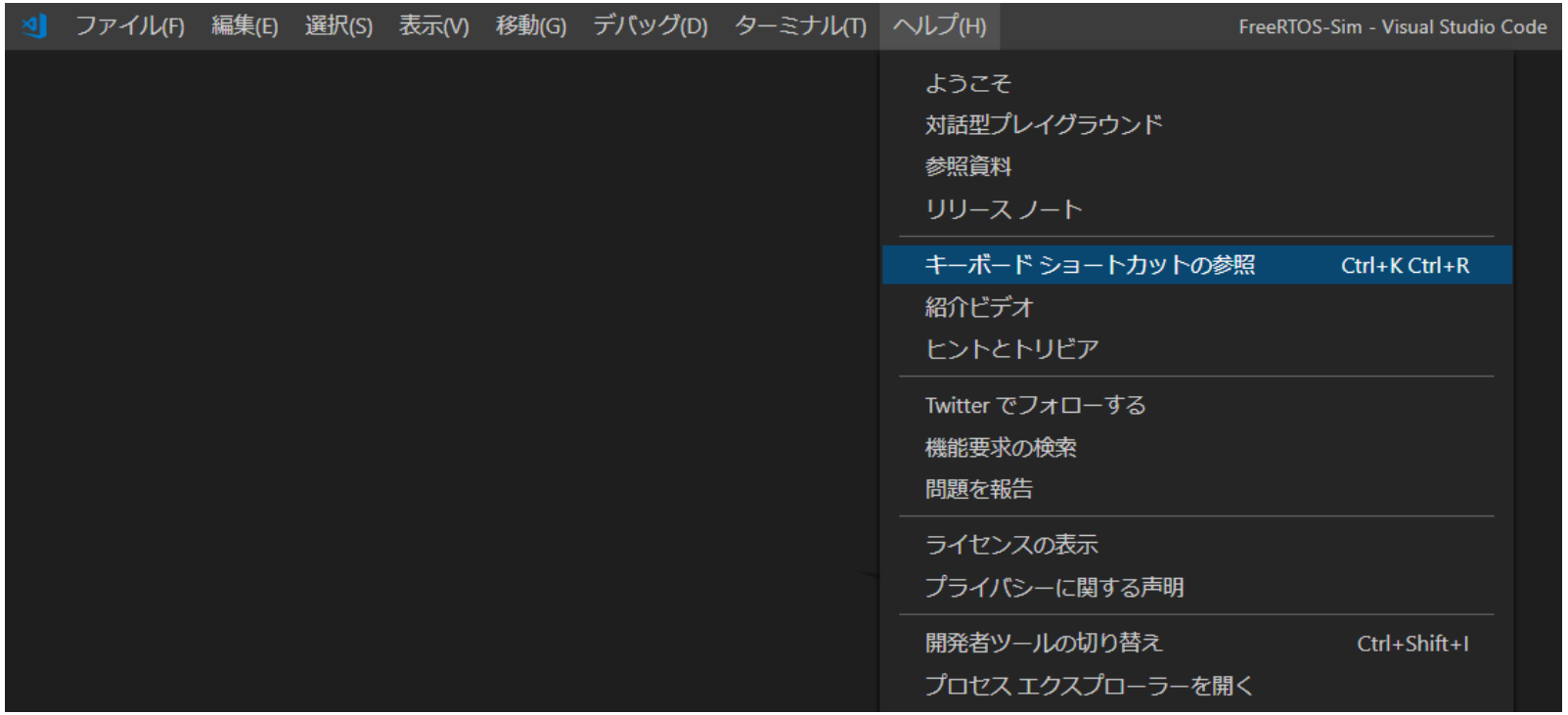
Visual Studio Code (Windows)

動作	ショートカットキー
行の切り取り（未選択時）	Ctrl + x
行のコピー（未選択時）	Ctrl + c
矩形選択	Ctrl + Shift + Alt + 矢印
ワークスペース内のシンボルへ移動	Ctrl + t（Ctrl + p, # 入力でも代用可）
ファイル内のシンボルへ移動	Ctrl + Shift + o（Ctrl + p, @ 入力でも代用可）
対応する括弧に移動	Ctrl + Shift + ¥
宣言 / ファイルに移動	F12
戻る / 進む	Alt + ← / →

もっと知りたい方は

[ヘルプ] → [キーボードショートカットの参照] より

ショートカットの一覧が見れる（英語）



URL: https://www.sli.do/　イベントID: #embtdd

IDEを知る

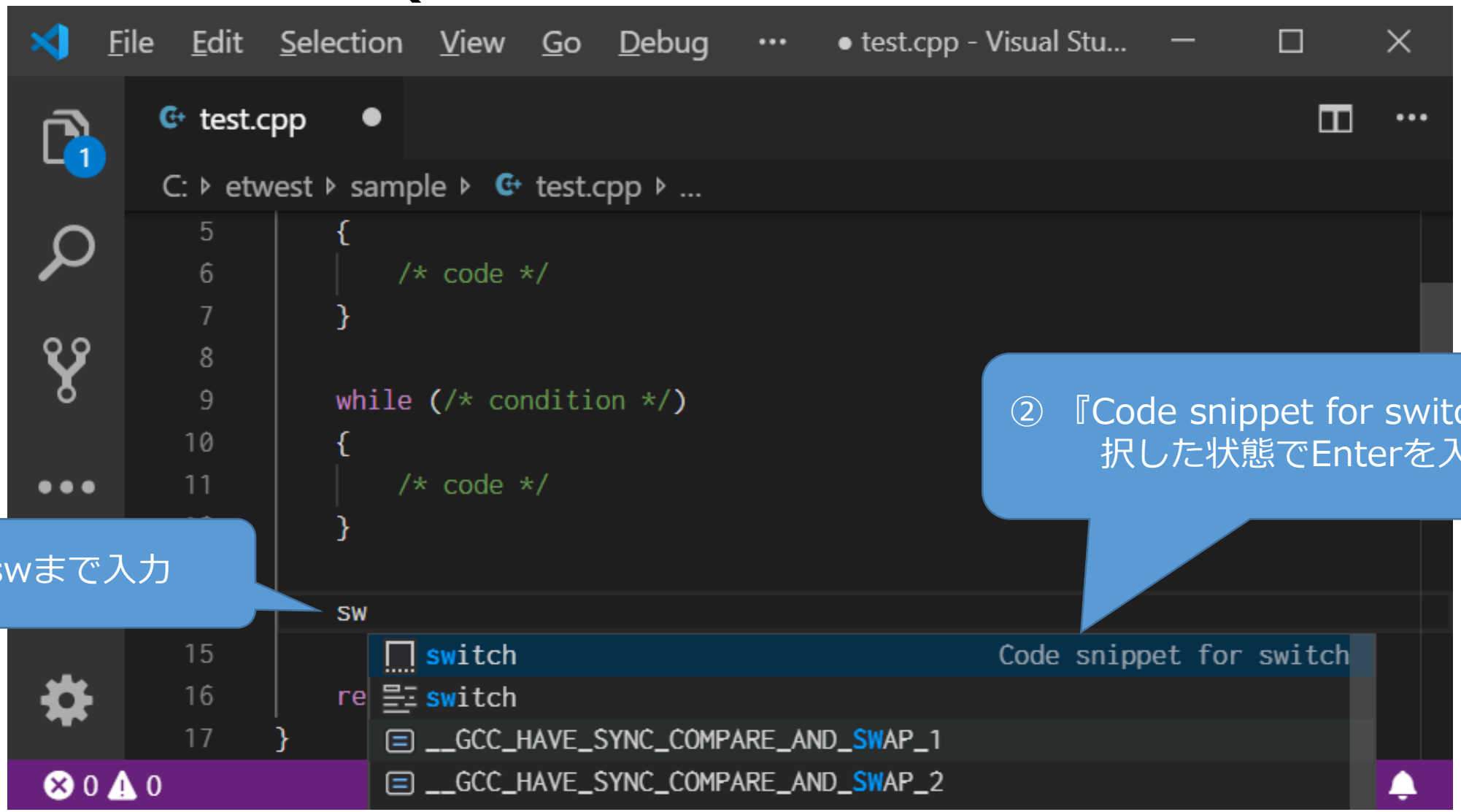
スニペット

コードの「切れ端」、「断片」のこと。

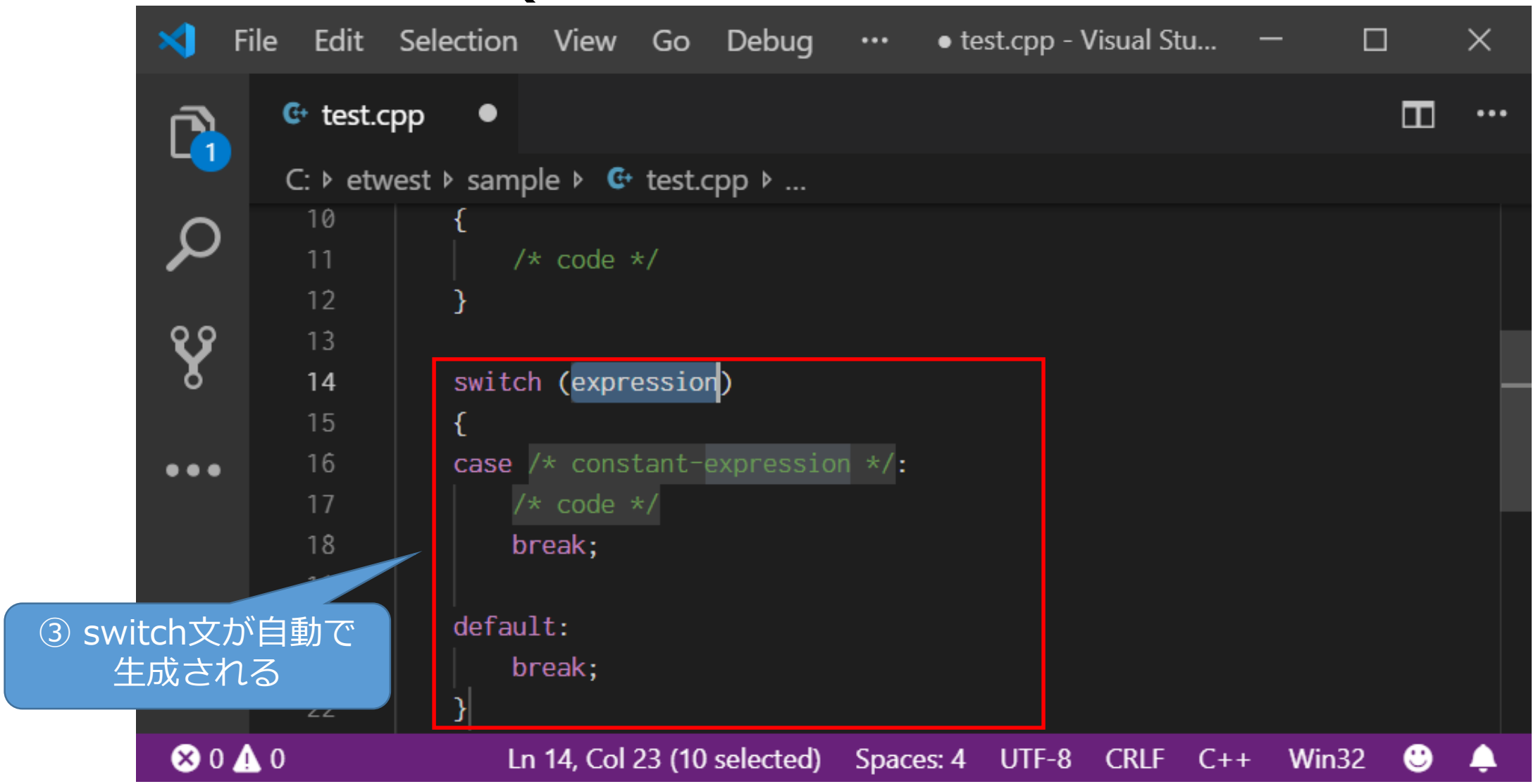
予めあるものに加えて独自のスニペットも作成可能なため、

よく使うコードをスニペットとして登録すると作業を効率化出来る。

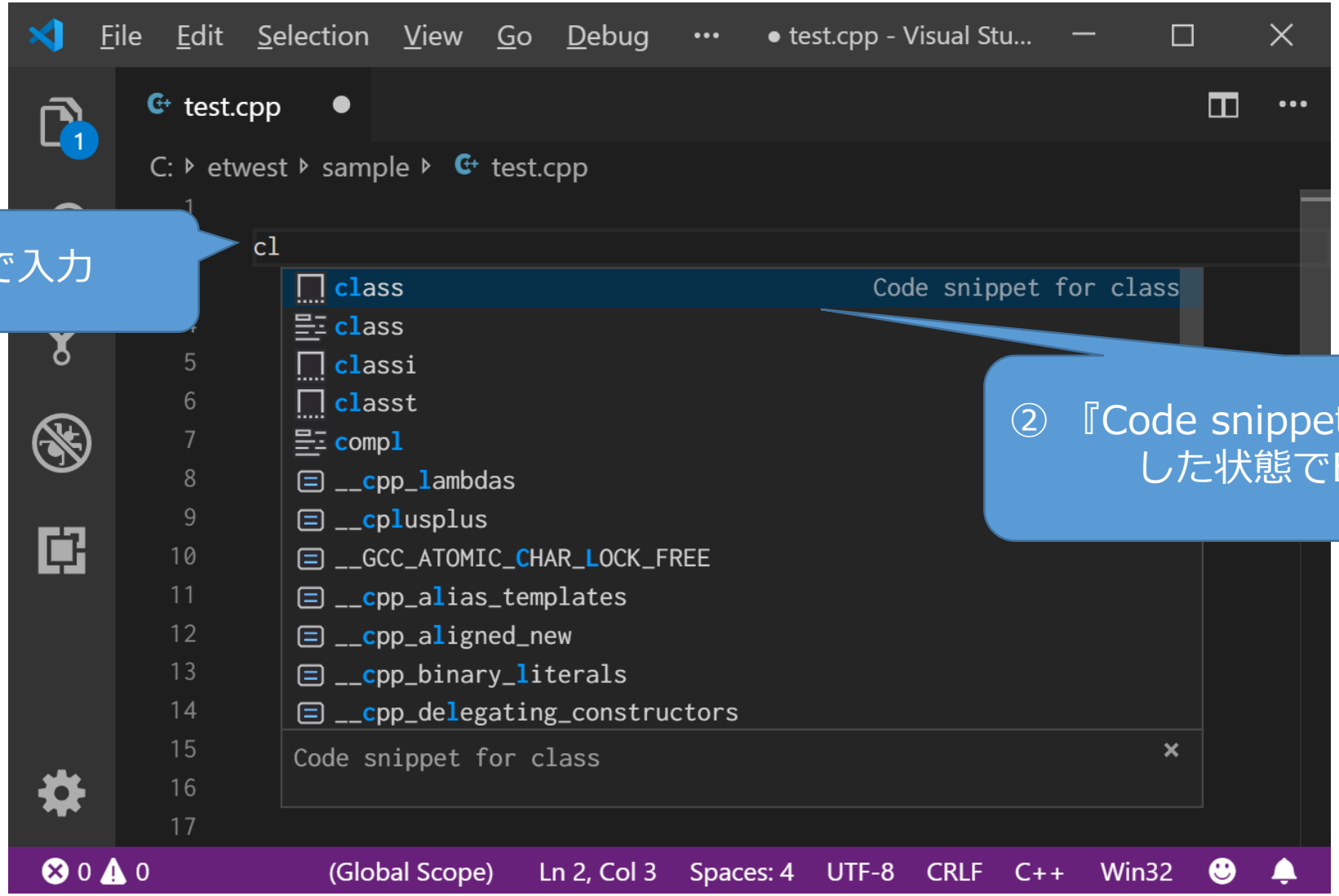
スニペット (Visual Studio Codeの場合)



スニペット (Visual Studio Codeの場合)



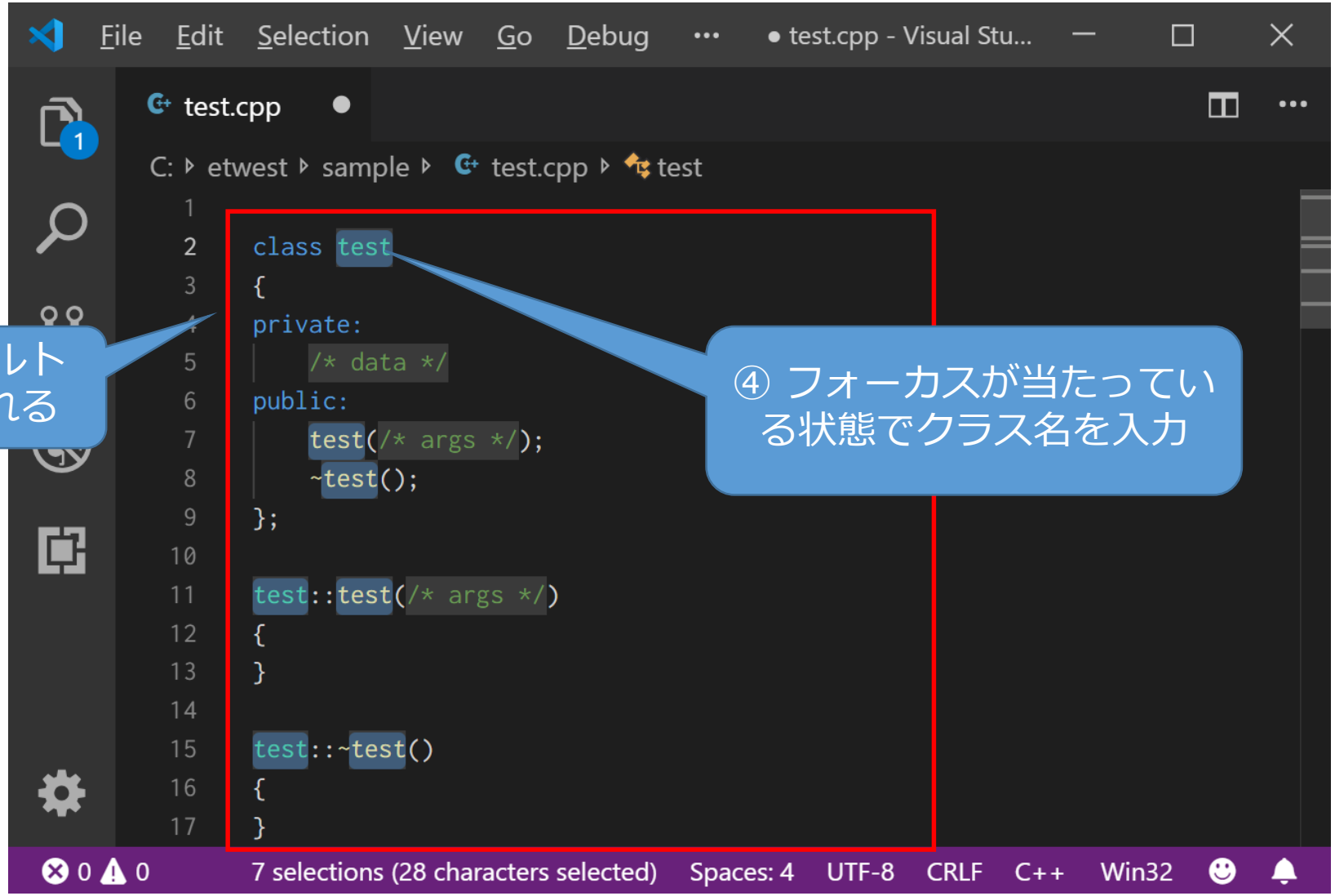
スニペット (Visual Studio Codeの場合)



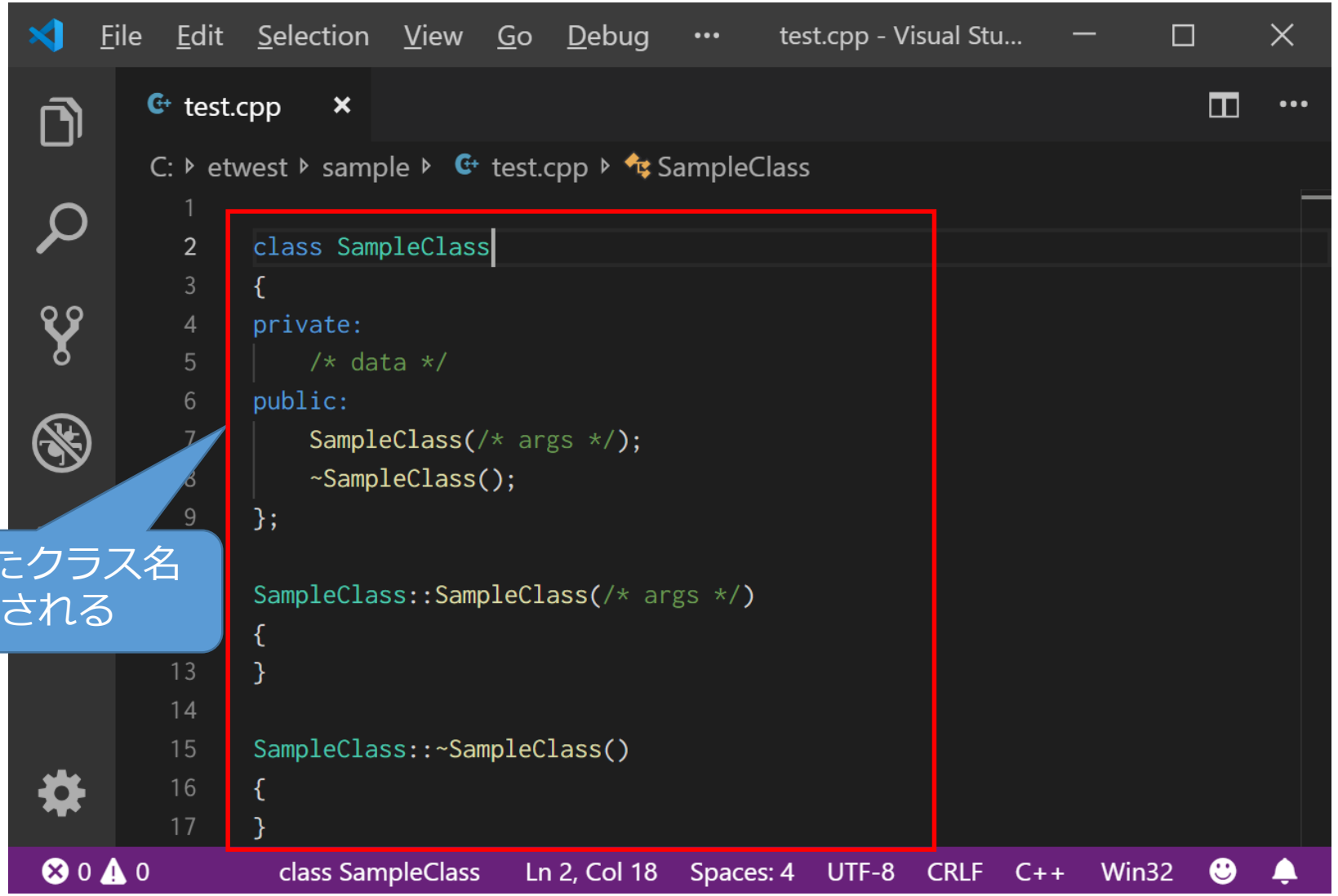
① clまで入力

② 『Code snippet for class』を選択した状態でEnterを入力

スニペット (Visual Studio Codeの場合)



スニペット (Visual Studio Codeの場合)



スキルのこと

- IDEのサポートは年々充実してきている
  - 関数抽出、変数名等などリファクタリングで実施する作業も機能的にサポートするなど
- とはいえ、C/C++系のIDEは他言語に比べると機能が充実しているとは言い難い
- Martin Fowler著 『リファクタリング』などを参考に自身のスキルとして身に付けましょう

レシピ７
組込みTDD勉強会に参加する

組込みTDD勉強会に参加する

若手エンジニアの「組込みでのTDDが学びたい！」という声を受け
実施している東京での勉強会が始まり。
コアメンバ数名の関西転勤を機に、一昨年9月より関西でも実施している。

活動実績）

第七回	：2018/12	実機(STM32)上でのテスト実行
第八回	：2019/1	ペアプログラミング
第九回	：2019/3	レガシーコード対策
第十回	：2019/4	リファクタリング
第十一回	：2019/5	ET West 2019 リハーサル

URL: https://www.sli.do/　イベントID: #embtdd

最後に

私達と共に最初の一歩を
踏み出してみませんか？

参考書籍

- 『レガシーコード改善ガイド』
  マイケル・C・フェザーズ 著
  ウルシステムズ株式会社 監訳
  平澤章／越智典子／稲葉信之／田村友彦／小堀真義 訳
- 『レガシーソフトウェア改善ガイド』
  Chris Birchall 著
  吉川邦夫 訳
- 『テスト駆動開発による組み込みプログラミング』
  James W.Grenning 著
  蛸島昭之 監訳
  笹井崇司 訳

参考書籍

- 『プログラマが知るべき97のこと』
  Kevlin Henney 編
  和田 卓人 監修
  夏目 大 翻訳
- 『リファクタリング―既存のコードを安全に改善する― 』
  Martin Fowler 著
  児玉 公信／友野 晶夫／平澤 章／梅澤 真史 訳