

Narzędzia do debugowania

Przygotował Arkadiusz Skiba

CZĘŚĆ I - Core dump

Core dump to plik zawierający zapis stanu programu (jego pamięci) z chwili, w której wystąpiło żądanie jego utworzenia. Mogło ono zostać spowodowane awarią aplikacji lub działaniem użytkownika.

Automatyczne generowanie pliku core

Plik core, czyli zrzut pamięci, zostanie automatycznie wygenerowany przez jądro systemu, gdy aplikacja ulegnie awarii i nie jest w stanie kontynuować swojej pracy. Może tak się stać na przykład, gdy nastąpi próba zapisu do niezaalokowanej pamięci lub wykonania błędnej instrukcji procesora.

Core dump na linuxie

Na systemach unixowych - w naszym przykładzie ubuntu - zrzut jest generowany automatycznie gdy program napotka błąd. Koniecznym warunkiem jest ustawienie limitu pliku core poleceniem

\$ ulimit -c unlimited

który domyślnie limit jest ustawiony na zero. Przyjrzyjmy się manualom:

a) \$ man core 5

Tu jak zaraz zobaczymy, mamy wpisane warunki kiedy core dump się nie wygeneruje.

b) \$ man 7 signal

Tu jak zaraz zobaczymy, mamy wylistowane sygnały, które potrafimy wyłączyć i analizować dzięki debugowaniu z plikiem core dump'a

Aby wygenerować plik z core dumpem, wadliwy program po prostu uruchamiamy. Plik pojawi się w bieżącym folderze. Następnie aby sprawdzić przyczynę problemu, wpisujemy komendę:

\$ gdb -q <program> <plik_core>

flaga -q wycisza niepotrzebne informacje

Aby zapisać zrzut całej przestrzeni adresowej do pliku, należy wykonać komendę

\$ objdump -s <plik_core> >> dump.txt

man 5 core

CORE(5)

Linux Programmer's Manual

CORE(5)

NAME

core - core dump file

DESCRIPTION

The default action of certain signals is to cause a process to terminate and produce a core dump file, a disk file containing an image of the process's memory at the time of termination. This image can be used in a debugger (e.g., `gdb(1)`) to inspect the state of the program at the time that it terminated. A list of the signals which cause a process to dump core can be found in `signal(7)`.

A process can set its soft `RLIMIT_CORE` resource limit to place an upper limit on the size of the core dump file that will be produced if it receives a "core dump" signal; see `getrlimit(2)` for details.

There are various circumstances in which a core dump file is not produced:

- * The process does not have permission to write the core file. (By default, the core file is called core or core.pid, where pid is the ID of the process that dumped core, and is created in the current working directory. See below for details on naming.) Writing the core file fails if the directory in which it is to be created is nonwritable, or if a file with the same name exists and is not writable or is not a regular file (e.g., it is a directory or a symbolic link).
- * A (writable, regular) file with the same name as would be used for the core dump already exists, but there is more than one hard link to that file.
- * The filesystem where the core dump file would be created is full; or has run out of inodes; or is mounted read-only; or the user has reached their quota for the filesystem.
- * The directory in which the core dump file is to be created does not exist.
- * The `RLIMIT_CORE` (core file size) or `RLIMIT_FSIZE` (file size) resource limits for the process are set to zero; see `getrlimit(2)` and the documentation of the shell's ulimit command (limit in `csh(1)`).

man 7 signal

First the signals described in the original POSIX.1-1990 standard.

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating-point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process

The signals **SIGKILL** and **SIGSTOP** cannot be caught, blocked, or ignored.

Przykład nr 1: segmentation fault

<https://pastebin.com/qPmXpLUP>

```
#include <iostream>

int main()
{
    char* c = NULL;
    *c = 'd';
    std::cout<<*c;
    return 0;
}
```

- 1) g++ main.cpp -o program
- 2) ./program
- 3) gdb -q program core

```
arek@arekUbuntu:~/Desktop/CoreDump$ gdb -q program core
Reading symbols from program...(no debugging symbols found)...done.

warning: core file may not match specified executable file.
[New LWP 3854]
Core was generated by `./prog'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x000055983c1fe80e in main ()
(gdb) █
```

Przykład nr 2: arithmetic exception

<https://pastebin.com/qPmXpLUP>

```
#include <iostream>
int main(){
    int input=5;
    int i = input/2;
    int c;
    for (i>0; i--;) {
        c= input%i;
        if (c==0 || i == 1)
            std::cout << "not prime" << std::endl;
        else
            std::cout << "prime" << std::endl;
    }
    return 0;
}
```

- 1) g++ main2.cpp -o program
- 2) ./program
- 3) gdb -q program core

```
arek@arekUbuntu:~/Desktop/CoreDump$ gdb -q program core
Reading symbols from program...(no debugging symbols found)...done.
[New LWP 4417]
Core was generated by './program'.
Program terminated with signal SIGFPE, Arithmetic exception.
#0  0x000055e30326c8ee in main ()
(gdb) Quit
(gdb)
[23]+  Stopped                               gdb -q program core
```


Samodzielne wygenerowanie pliku core

Plik zrzutu możemy również wygenerować samodzielnie za pomocą polecenia `gcore`. Nie wymaga to przerywania pracy programu - proces ten będzie działał poprawnie również po wykonaniu zrzutu.

Oto przebieg przykładowej sesji, w której wygenerowano plik core dla aktualnie uruchomionego shella Bash:

```
$ ps
```

```
PID TTY      TIME CMD
```

```
15665 pts/7    00:00:00 bash
```

```
$ gcore -o myBashDump 15665
```

```
0x00007f4ba32bd58a in __GI___waitpid (pid=-1, stat_loc=0x7ffed73b4280, options=10) at  
../sysdeps/unix/sysv/linux/waitpid.c:29
```

```
29      return SYSCALL_CANCEL (wait4, pid, stat_loc, options, NULL);
```

```
Saved corefile myBashDump.15665
```

WINDOWS

System windows umożliwia generowanie zrzutów pamięci w celach recovery. W zaawansowanych ustawieniach systemu możemy określić typ zrzutu jaki powinien się wygenerować w przypadku crasha.

Dostępne możliwości:

- automatic memory dump
- small memory dump
- kernel memory dump
- complete memory dump
- active memory dump

Start-up and Recovery

System start-up

Default operating system:

Windows 10

☐ Time to display list of operating systems: 0 seconds

☐ Time to display recovery options when needed: 30 seconds

System failure

☒ Write an event to the system log

☒ Automatically restart

Write debugging information

Automatic memory dump

Dump file:

%SystemRoot%\MEMORY.DMP

☒ Overwrite any existing file

☐ Disable automatic deletion of memory dumps when disk space is low

OK Cancel

CZĘŚĆ II - Debugger wbudowany w CLion

W tej części prezentacji pokażę wam różne możliwości jakie oferuje debugger gdb w środowisku programistycznym CLion na przykładzie programu "gra w życie"

1. Step Into, Step Over, Step Out



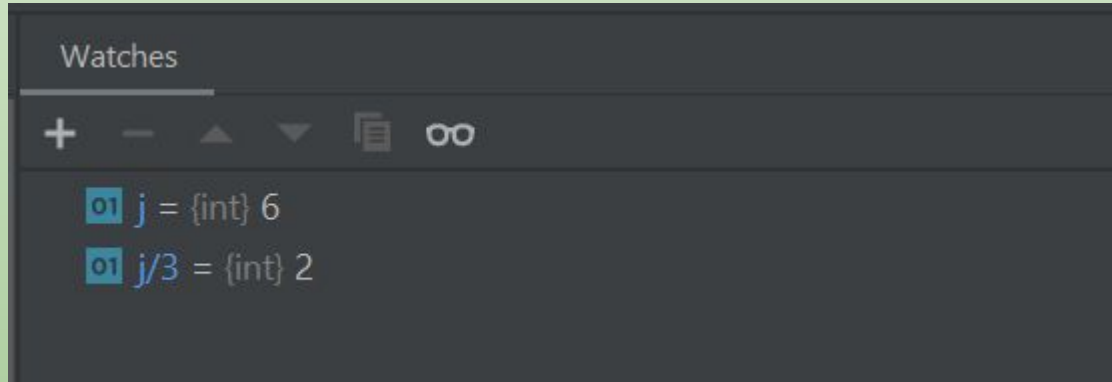
2. Podgląd zmiennych

The screenshot displays the GDB (GNU Debugger) interface, specifically the 'Variables' pane. The 'Frames' pane on the left shows the current execution context, with 'Game::updateBoard Game.cpp:40' selected. The 'Variables' pane on the right shows the state of variables in memory. The variable 'tempBoard' is a 10x10 array of 'Cell' objects. The first two rows are expanded, showing the attributes of each cell. The first cell has 'pos' as 'UPLEFT', 'alive' as 'false', 'neighbours' as '1', and 'c' as a space character. The second cell has 'pos' as 'UP', 'alive' as 'false', 'neighbours' as '1', and 'c' as a space character. The remaining rows are collapsed.

```
Frames
  Thread-1
    Game::updateBoard Game.cpp:40
    Game::run Game.cpp:51
    main main.cpp:6
    __tmainCRTStartup 0x00000000004013c7
    mainCRTStartup 0x00000000004014fb

Variables
  tempBoard = {Cell [10][10]}
    [0] = {Cell [10]}
      [0] = {Cell}
        pos = {Cell::Position} Cell::UPLEFT
        alive = {bool} false
        neighbours = {int} 1
        c = {char} 32 ' '
      [1] = {Cell}
        pos = {Cell::Position} Cell::UP
        alive = {bool} false
        neighbours = {int} 1
        c = {char} 32 ' '
      [2] = {Cell}
      [3] = {Cell}
      [4] = {Cell}
      [5] = {Cell}
      [6] = {Cell}
```

3. Watches



4. Breakpointy

The screenshot shows the Visual Studio Breakpoints window. On the left, a list of breakpoints is shown under the 'Breakpoints' tab. The first section is 'Line Breakpoints', which includes two breakpoints for 'Game.cpp': one at line 52 (selected) and one at line 40. Below this are 'Exception Breakpoints' for 'When any is thrown', 'Python Exception Breakpoint' (with 'Any exception' checked), and 'JavaScript Exception Breakpoints' (with 'Any exception' checked). The right pane shows the configuration for the selected breakpoint at 'Game.cpp:52'. It is 'Enabled', 'Suspend', and has a 'Condition' of 'board[6][6].isActive'. The 'Log' section is checked, with options for 'Breakpoint hit' message and 'Stack trace'. The 'Evaluate and log' section is also checked, with the log message 'Na dzien dzisiejszy dabesta' entered. The 'Remove once hit' option is unchecked. The 'Disable until breakpoint is hit' section is set to '<None>'. The 'After hit' section has 'Disable again' selected. At the bottom, a code editor shows the context of the breakpoint, with lines 50, 51, and 52. Line 52, which contains 'printBoard();', is highlighted with a red background and a breakpoint icon.

Breakpoints

+ - [📄]

▼ ☒ Line Breakpoints

☒ Game.cpp:52

☒ Game.cpp:40

▼ ☐ Exception Breakpoints

☐ When any is thrown

▼ ☒ Python Exception Breakpoint

☒ Any exception

▼ ☐ JavaScript Exception Breakpoints

☐ Any exception

Game.cpp:52

☒ Enabled

☒ Suspend

☒ Condition:

 board[6][6].isActive

Log: ☐ "Breakpoint hit" message ☐ Stack trace

☒ Evaluate and log:

 Na dzien dzisiejszy dabesta

☐ Remove once hit

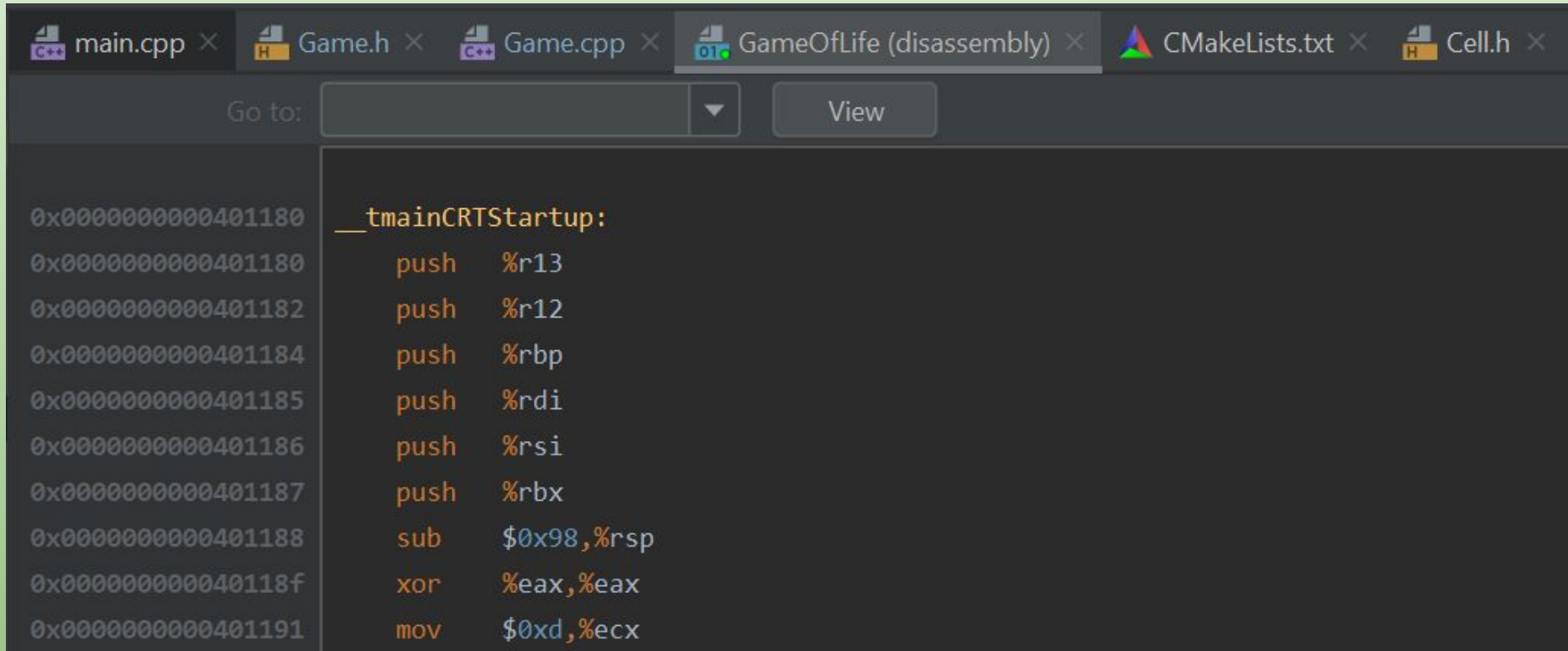
Disable until breakpoint is hit:

 <None>

After hit: ☒ Disable again ☐ Leave enabled

```
50     while (i++ < 100000) {
51         updateBoard();
52  printBoard();
```

5. Podgląd programu w trybie dissassembly



```
main.cpp × Game.h × Game.cpp × GameOfLife (disassembly) × CMakeLists.txt × Cell.h ×  
Go to:  View  
  
0x0000000000401180 __tmainCRTStartup:  
0x0000000000401180     push    %r13  
0x0000000000401182     push    %r12  
0x0000000000401184     push    %rbp  
0x0000000000401185     push    %rdi  
0x0000000000401186     push    %rsi  
0x0000000000401187     push    %rbx  
0x0000000000401188     sub     $0x98,%rsp  
0x000000000040118f     xor     %eax,%eax  
0x0000000000401191     mov     $0xd,%ecx
```


CZĘŚĆ III - Unit testy i raport z pokryciem

Test jednostkowy – metoda testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów (jednostek) programu – np. metod lub obiektów w [programowaniu obiektowym](#) lub procedur w [programowaniu proceduralnym](#). Testowany fragment programu poddawany jest testowi, który wykonuje go i porównuje wynik (np. zwrócone wartości, stan obiektu, zgłoszone wyjątki) z oczekiwanymi wynikami – tak pozytywnymi, jak i negatywnymi (niepowodzenie działania kodu w określonych sytuacjach również może podlegać testowaniu).

Zaletą testów jednostkowych jest możliwość wykonywania na bieżąco w pełni zautomatyzowanych testów na modyfikowanych elementach programu, co umożliwia często wychwycenie błędu natychmiast po jego pojawieniu się i szybką jego lokalizację zanim dojdzie do wprowadzenia błędnego fragmentu do programu. Testy jednostkowe są również formą specyfikacji

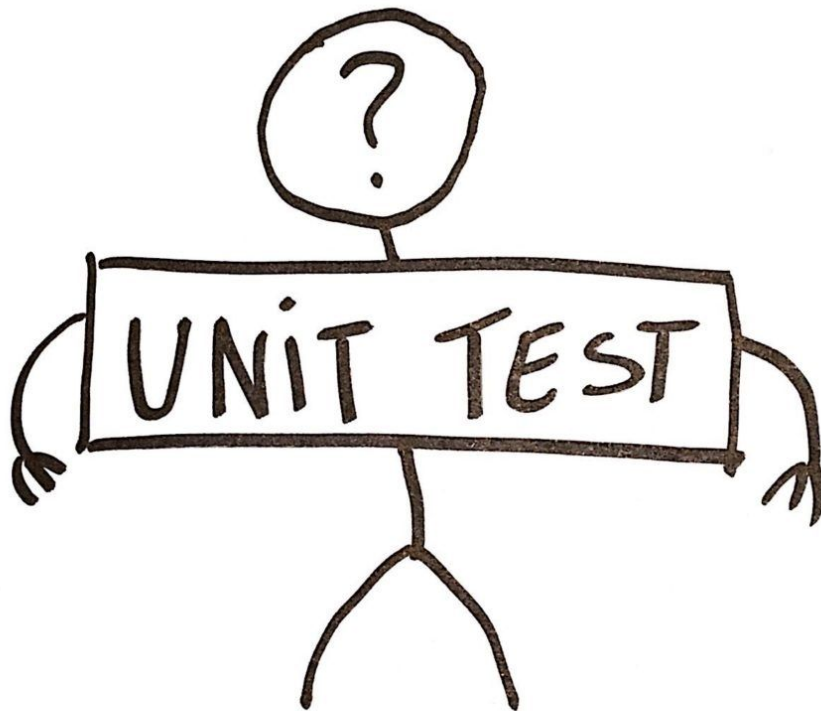
Testy można podzielić na następujące warianty:

- analiza ścieżek
- użycie klas równoważności
- testowanie wartości brzegowych
- testowanie składniowe
- mocking

Przykładowe Frameworki do UT w C/C++:

- Google Test
- CppUTest
- Boost Test Library
- CppUnit

W przykładach będziemy używać Google Test



Przykładowe testy w Google Test:

```
#include "gtest/gtest.h"
#include "main.cpp"

TEST(GreaterTest, AIsGreater){
    EXPECT_EQ(3, GreatestOfThree(3, 1, 2));
};

TEST(GreaterTest, BIsGreater){
    EXPECT_EQ(3, GreatestOfThree(1, 4, 2));
};

TEST(GreaterTest, CIsGreater){
    EXPECT_EQ(3, GreatestOfThree(1, 2, 3));
};

int main(int argc, char**argv)
{
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Do makra `EXPECT_EQ` podajemy oczekiwaną wartość, oraz wynik testowanej funkcji.

Jeśli nasz kod jest pisany w stylu umożliwiającym zastosowanie TDD (Test Driven Development) warto napisać testy jednostkowe przed napisaniem kodu. Często pozwala to na uniknięcie błędów i jest formą prewencji przed koniecznością debuggowania.

Uruchomiony test:

```
[=====] Running 1 test from 1 test case.  
[-----] Global test environment set-up.  
[-----] 1 test from TestCells  
[  RUN   ] TestCells.first  
[      OK ] TestCells.first (0 ms)  
[-----] 1 test from TestCells (1 ms total)  
  
[-----] Global test environment tear-down  
[=====] 1 test from 1 test case ran. (1 ms total)  
[  PASSED ] 1 test.
```

Coverage - Czyli po polsku pokrycie. W programowaniu rozumiane jako pokrycie kodu testami. Jest to procentowa ilość kodu przez jaki przechodzą napisane przez nas Unit Testy. Warto testować i trzymać jak największą część projektu, dzięki czemu kod jest bardziej odporny na błędy, a same testy są pewnego typu dokumentacją kodu.

Do wygenerowania raportu z pokryciem unit testami w C++ służy narzędzie **gcov**

Do wygenerowania front-endowej nakładki na raport w C++ służy narzędzie **lcov**

Ostatnim krokiem w celu wygodnej analizy jest konwersja raportu do formatu **html**

Na głównej stronie pliku html widzimy pliki w naszym projekcie oraz procentowe pokrycie funkcji w każdym z pliku. Aby funkcja była zaliczona jako 'przetestowana', co najmniej raz jakiś test musi przez nią przejść.




LCOV - code coverage report

Current view: [top level](#) - /home/arek/Desktop/GameOfLife

Test: [main_coverage.info](#)

Date: 2020-04-01 13:08:24

	Hit	Total	Coverage
Lines:	38	126	30.2 %
Functions:	12	16	75.0 %

Filename	Line Coverage ↕		Functions ↕	
Cell.h		100.0 % 1 / 1	100.0 %	1 / 1
Game.cpp		21.4 % 24 / 112	50.0 %	4 / 8
main_test.cpp		100.0 % 13 / 13	100.0 %	7 / 7

Generated by: [LCOV version 1.13](#)

Tu widzimy przykład, gdzie widać ilość przetestowanego kodu w konkretnym pliku:

```
0 :         break;
:     }
0 :     case Cell::LEFT: {
0 :         if (board[i - 1][0].alive) tempCell.neighbours++;
0 :         if (board[i + 1][0].alive) tempCell.neighbours++;
0 :         if (board[i - 1][1].alive) tempCell.neighbours++;
0 :         if (board[i + 1][1].alive) tempCell.neighbours++;
0 :         if (board[i][1].alive) tempCell.neighbours++;
0 :         break;
:     }
: }
:
1 : if (!tempCell.alive && tempCell.neighbours == 3)
: {
0 :     tempCell.alive = true;
: }
1 : else if (tempCell.alive && (tempCell.neighbours != 2 && tempCell.neighbours != 3))
: {
1 :     tempCell.alive = false;
: }
1 : }
:
0 : void Game::init() {
:
0 :     board[2][0].alive=true;
0 :     board[2][1].alive=true;
0 :     board[2][2].alive=true;
0 :     board[0][1].alive=true;
0 :     board[1][2].alive=true;
0 : };
:
3 : Cell Game::board[10][10];
```

**Czerwony kolor -
brak testów**

**Niebieski kolor -
kod
przetestowany**

KONIEC :-)

Dziękuję za uwagę