Console-Based Checkers Game Release Notes *

Belser, C.
cmb923@drexel.edu

Brennan, Z.
zab37@drexel.edu

Horsey, K.
kth37@drexel.edu

van Rijn, Z.
zwv23@drexel.edu

August 29, 2017

Abstract

Thank you for your purchase of and patience with your new Console-Based Checkers(tm) game. We hope you will be thoroughly satisfied with our implementation, and look forward to serving you and your business needs in the future. Please don't hesitate to reach out to us in the unlikely event that you encounter any issues with your new software, or wish for us to implement additional features.

1 Source Code

You will find the complete source code including helpful scripts and utilities in the included `final_transfer.tar.gz`. We employ the following directory structure:

```
Z:\PROJECTS\CS451-CHECKERS
+---docker
|   +---Dockerfile
|   \---Makefile
+---docs
|   +---design.pdf
|   +---design.tex
|   +---img
|   |   +---crown120.png
|   |   +---crown.png
|   |   +---logo.png
|   |   +---old
|   |   |   +---GUI_Loading_Screen
```

*CS451:002 Group 2, Drexel University

```
|   |   |   +---UML_Client
|   |   |   +---UML_Client.png
|   |   |   +---UML_Game_Logic
|   |   |   +---UML_Game_Logic.png
|   |   |   +---UML_Game_State
|   |   |   \---UML_Game_State.png
|   |   +---piece120.png
|   |   +---piece.png
|   |   +---src
|   |   |   +---UML_PNGs-20170813T183501Z-001.zip
|   |   |   \---UML_XMLs-20170813T183420Z-001.zip
|   |   +---UML_Client.png
|   |   +---UML_Game_Manager.png
|   |   +---UML_Game_State.png
|   |   +---UML_Network.png
|   |   \---UML_Server.png
|   +---release.pdf
|   +---release.tex
|   +---requirements.pdf
|   +---requirements.tex
|   +---testcases.pdf
|   \---testcases.tex
+---Makefile
+---README.md
+---spec
|   +---Evaluation\ form(1).xlsx
|   +---iso-iec-ieee-29148-2011.pdf
|   +---Project\ Description.docx
|   +---Project\ Overview.pptx
|   +---Sample\ Design\ Document.pdf
|   +---Sample\ Requirements\ Document.pdf
|   \---Sample\ Test\ Case\ Document.pdf
+---src
|   +---checkers
|   +---fonts
|   |   \---arcade.h
|   +---fonts.c
|   +---fonts.h
|   +---game
|   |   +---board.c
|   |   +---board.h
|   |   +---display.c
|   |   +---display.h
|   |   +---game_logic.c
|   |   +---game_logic.h
|   |   +---game_manager.c
```

```
|   |   +---game_manager.h
|   |   +---move.c
|   |   +---move.h
|   |   +---user_interface.c
|   |   \---user_interface.h
|   +---main.c
|   +---Makefile
|   +---Makefile.am
|   +---network
|   |   +---client.c
|   |   +---client.h
|   |   +---Makefile
|   |   +---server.c
|   |   \---server.h
|   \---test
|       +---Makefile
|       +---TEST_display.c
|       \---TEST_user_interface.c
\---tests
    +---main.c
    +---Makefile.am
    \---tests.c

12 directories, 70 files
```

Compiling your software is a simple, straightforward process but it requires that you have *Docker* installed. Docker is thoroughly described in the accompanying `design.pdf` document.

Assuming *Docker* is installed, the build process is quite straightforward:

```
make game
```

Additional "test" utilities (demonstrations that individual components work properly) can be compiled using the following, but unsupported, commands:

```
make display
make logic
make network
make user_interface
```

2 Installing

The executable(s) produced by the **Compiling** section are linked statically, which means that they include all necessary libraries and objects for the proper execution of our software.

As such, they are "installed" to the `bin/` directory at the project root, and can be safely copied to any other `x86_64` Linux-based platform.

In the event that our software needs to be run on Windows, two options exist:

1. **via Docker** – simply run `make run-con` to execute the binary inside a Docker container (works on any platform)
2. **recompiling** – use your favorite C compiler to build the requisite libraries (download links are provided in the `Dockerfile`) and then build the project source code. The code is compatible with Windows.

3 Release Notes

There is some odd behavior on some Linux-based systems where the console display output is misaligned, miscolored, or otherwise incorrect.

This issue is caused by the improper configuration of the `TERM` and/or `TERMINFO` environment variable(s). On many popular distributions including Ubuntu and Mint, these are often incorrectly set.

The simplest remedy is to run the following:

```
TERM=xterm-color ./bin/game
```

4 Unit Testing

We employ the *libcheck* library to perform unit tests. It is a fairly straightforward framework, taking tests in the following format:

```
#include <check.h>

START_TEST(my_test)
{
        ck_assert_int_eq(B, 0);
        ck_assert_int_eq(E, 1);
        ck_assert_int_eq(F, 2);
}
END_TEST
```

More information can be found in either the accompanying `requirements.pdf` or `design.pdf` documents.

Note: the *libcheck* library operates differently from most other "code coverage" tools in that, due to limits with the C programming language, cannot provide a line-by-line coverity report. As such, the output to our unit tests is the program return code of `0`.

5 Code Coverage

As far as code coverage is concerned, we make use of the GNU coverage too *gcov*. It works like this.

First, source code is compiled with the additional flags:

```
-fprofile-arcs \
-ftest-coverage \
-fprofile-dir=./data \
-fprofile-generate=./data
```

Then, there are two choices:

1. Run the main game executable, to see if there is any "dead code" (making sure to test all possible functionalities).
2. Run the unit-testing framework(s), to exercise as much of the underlying code as possible, in an effort to determine *how much* of the source code is *covered* by the unit tests. The accuracy of this number strongly depends on the previous point.

Unfortunately, since the client's requirements in the domain of unit testing and code coverage were no specified, we make no assumptions as to a hypothetical *minimum* percentage of code covered. We simply demonstrate that the functionality and ability to use these tools and run these tests, is properly implemented. You can verify that this is the case by running the following command(s):

```
make cover
```

This action will perform the aforementioned steps, generating profiling "notes" containing structural information about the software, as well as profiling statistics when the executable is tested. The tool *Valgrind* may also be used to profile C code in much the same fashion. In summary, the concept is simple: determine code coverage on the unit testing framework.

Of course, unit testing and coverity analysis are definitely good metrics, but they describe only observable behaviors of the system and cannot find logic errors better than the humans that programmed them.

6 Static Code Analysis

We use the tool *cppcheck* to perform static analysis of our code, as well as the standard C library *MUSL* and any other header files that we include. It operates by checking each source file:

```
Checking src/fonts.c ...
[/usr/include/stdio.h:102]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
```

```
[/usr/include/stdio.h:112]: (information) Skipping configuration
    'GCC_SCANF;scanf' since the value of 'scanf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[/usr/include/curses.h:1663]: (information) Skipping configuration
    '__MINGW32__;trace' since the value of 'trace' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
1/12 files checked 7\% done
Checking src/game/board.c ...
2/12 files checked 19\% done
Checking src/game/display.c ...
[/usr/include/ncurses.h:1663]: (information) Skipping configuration
    '__MINGW32__;trace' since the value of 'trace' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
3/12 files checked 30\% done
Checking src/game/game_logic.c ...
4/12 files checked 42\% done
Checking src/game/game_manager.c ...
5/12 files checked 53\% done
Checking src/game/move.c ...
6/12 files checked 65\% done
Checking src/game/user_interface.c ...
7/12 files checked 76\% done
Checking src/main.c ...
8/12 files checked 84\% done
Checking src/network/client.c ...
[./src/network/client.c:88]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[./src/network/client.c:92]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[./src/network/client.c:136]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[./src/network/client.c:146]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[./src/network/client.c:150]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
9/12 files checked 88\% done
Checking src/network/server.c ...
[./src/network/server.c:117]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[./src/network/server.c:146]: (information) Skipping configuration
```

```
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[./src/network/server.c:152]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[./src/network/server.c:176]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[./src/network/server.c:188]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[./src/network/server.c:193]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[./src/network/server.c:197]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
[./src/network/server.c:206]: (information) Skipping configuration
    'GCC_PRINTF;printf' since the value of 'printf' is unknown.
    Use -D if you want to check it. You can use -U to skip it explicitly.
10/12 files checked 92\% done
Checking src/test/TEST_display.c ...
11/12 files checked 96\% done
Checking src/test/TEST_user_interface.c ...
12/12 files checked 100\% done
```

Static code analysis may reveal bits of *truly* dead code, or code that is entirely inaccessible during normal execution of the program. It can reveal logic errors, as well, but there is a tradeoff between performance and accuracy. Not even the slowest, most thorough, expensive tools can detect all errors with 100% certainty, but a static code analyzer is definitely a step in the right direction.