Bulls and Cows

COLLABORATORS			
	TITLE : Bulls and Cows		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Jordi Fita	February 6, 2018	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
86024768ea09	2011-08-01	The download links of bullsandcows are now images instead.	jfita
02966c43ff92	2011-04-14	Removed the laying bit about rand() mod x not being normal.	jfita
2f571afc1f35	2011-03-31	Merged	jfita
6d069a050252	2011-03-28	Corrected the screenshot's size in bulls and cows.	jfita
a6bd81fb8b84	2011-03-31	Fixed a couple types and explained why I'm choosing string for the numbers instead of intergers in bulls and cows.	jfita
06d4f661307e	2011-03-28	Added the screenshot to bulls and cows.	jfita
2254a0f1d22a	2011-03-28	Fixed the directive of including ctime for bullsandcows.	jfita
b6ca632a243d	2011-03-28	Added the download section to bulls and cows.	jfita
d0c7f43c77eb	2011-03-28	Reworded some paragraphs of bulls and cows for better understanding.	jfita
aecb3cecdade	2011-03-28	Removed the apostrophe from bullsandcows' atangle directives.	jfita

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
619a7a1790df	2011-03-28	Added the first draft of bulls and cows.	jfita

Contents

1 Overview 1

2 Generating the Secret Code 3

3 Player's Input 5

4 Counting Bulls and Cows 6

5 Keep Playing 7

A Makefile 8

B License 9

List of Figures

1 Screenshot of *Bulls and Cows* with some guesses. 1

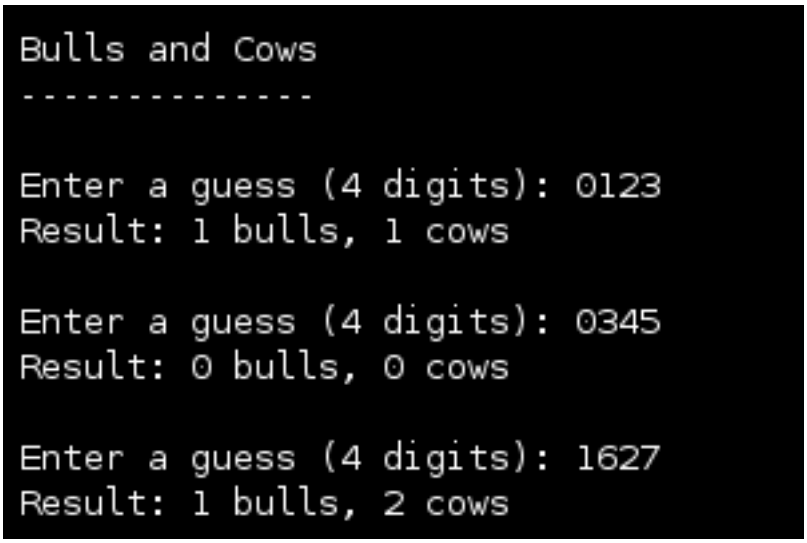

Figure 1: Screenshot of *Bulls and Cows* with some guesses.

Bulls and Cows is a code-breaking game for two players in which players think a secret number that has non-repeating digits. Then, in turns, each player tries to guess the other player's number. If the player's guess matching digits are on the right positions, they are *bulls*. If on different positions, they are *cows*. For example:

- Secret number: 1357
- Guess: 1723
- Answer: 1 bull, 2 cows.

The first player to guess correctly the opponent's number wins the game.

Download Source Code

1 Overview

In this game's version, written in C++ as a console application, the computer generates a random number. The generated number is, by default, four digits long but can be set to be between 3 to 6 digits. Then, the player tries to find that number using as few guesses as possible. At each player's guess, the computer tells how many *bulls* and *cows* there are in the guess using the rules explained above.

```
<<bullsandcows.cpp>>=
//
// Bulls & Cows - A simple code-breaking game.
// Copyright (c) 2011 Jordi Fita <jfita@geishastudios.com>
//
<<license>>
//
<<includes>>

<<function to generate secret code>>

<<function to read player guess>>
```

```
<<function to compute the number of bulls and cows>>

int
main(int argc, char *argv[])
{
    try
    {
        std::cout << "\nBulls and Cows\n--------------\n\n";

        <<get secret code length>>
        <<initialize random number generator>>
        bool playing = true;
        do
        {
            <<generate secret code>>
            <<reset number of bulls and cows>>
            do
            {
                <<ask for the player guess>>
                <<check how many bulls and cows there are in the guess>>
            }
            <<until the secret code is discovered>>
            <<ask whether to play again>>
        }
        while (playing);
        return EXIT_SUCCESS;
    }
    <<catch and show errors>>
    return EXIT_FAILURE;
}
```

I am using the standard constants `EXIT_SUCCESS` and `EXIT_FAILURE` to signal whether the application finished correctly or due an error. These two constants are defined inside the `cstdlib` header.

```
<<includes>>=
#include <cstdlib>
```

As this is a C++ program, when there are non-recoverable errors I throw exceptions. At minimum, the most basic error handling I can do is prevent the exception to leave `main` and instead output to the console that I've got an error. If the error is derived from the standard `exception` class, then I print the error message that the object holds. Otherwise, I don't have a clue of what error it is.

```
<<catch and show errors>>=
catch (std::exception &e)
{
    std::cerr << "Error: " << e.what() << std::endl;
}
catch (...)
{
    std::cerr << "Unknown error" << std::endl;
}
```

The standard `exception` class is defined in the `stdexcept` header file.

```
<<includes>>=
#include <stdexcept>
```

The `cerr` object that I'm using to show the error message to the console standard error output is declared and instanced in `iostream`.

```
<<includes>>=
#include <iostream>
```

2 Generating the Secret Code

Before I can generate the secret code to guess, first I need to know how many digits this code needs to have. By default, the number of digits is four, but the player can pass a different number, between 3 and 6, as a parameter to the application. Therefore, I initialize the variable that holds the code's length to `4`, but if there is any argument then I try to read it as the new length.

```
<<get secret code length>>=
unsigned int secretCodeLength = 4;
if (argc > 1) {
    <<get the secret code length from the parameters>>
}
```

Unfortunately, the argument list passed to main is an array of C strings (`char *`), but I need an integer. To convert the first argument to an integer, the C++ idiomatic way is to create a *string stream* object to wraps the string in a stream-like object, like `cout` or `cin` are. Then, I can try to retrieve the integer from this stream and check if there was any problem. If I couldn't extract the integer from the stream (i.e., the first parameter is not an integer) or the integer is not within the 3 to 6 range I throw an `invalid_argument` exception telling what is the problem and forcing the game to quit.

```
<<get the secret code length from the parameters>>=
std::istringstream parameters(argv[1]);
if (!(parameters >> secretCodeLength)) {
    throw std::invalid_argument("Invalid parameter.  Usage: bullsandcows length[=4].");
}
if (secretCodeLength < 3 || secretCodeLength > 6) {
    throw std::invalid_argument("The secret code's length must be between 3 and 6");
}
```

The `invalid_argument` class is defined in the `stdexcept` header already included. The `istringstream` class is defined inside `sstream`.

```
<<includes>>=
#include <sstream>
```

Now that I have the length of the code to generate I can create the number. Even though the secret code is a number, I store the secret code in an string. That's because it is far easier in C++ to compare each digit in an string, later when I have the player's input, that to compare the individual digits of integer variables.

```
<<generate secret code>>=
std::string secretCode = generate_secret_code(secretCodeLength);
```

The `string` class is defined in the standard header file `string` which, although is is already pulled in by `sstream` as a dependence, I nevertheless include it here because now I am using `string` directly.

```
<<includes>>=
#include <string>
```

The function to generate the secret code expects an unsigned integer in the range from 3 to 6 and passing a different value is a precondition violation with undefined result. To help catch this error while developing in debug mode, I assert that the value is within the expected range.

```
<<function to generate secret code>>=
std::string
generate_secret_code(unsigned int length)
{
```

```
    assert(length >= 3 && length <= 6 && "Invalid secret code length");

    <<generate secret code string>>
    <<return secret code string>>
}
```

The `assert` macro is defined in the `cassert` header.

```
<<includes>>=
#include <cassert>
```

I need to remember that the generated secret code must not repeat digits. This means that I need a list to know which are the digits available for me to use to generate the code. Initially, this list will contains all ten digits, from `0` to `9`. Each time I use a digit from the list, I remove that digit because I can not use it anymore for this code. The digit from the list is chosen randomly and hence I require direct access to the list's elements. The best standard container to use in this case is `vector` defined in the `vector` standard header.

```
<<includes>>=
#include <vector>
```

```
<<build list of available digits>>=
std::vector<unsigned int> digits;
for (unsigned int digit = 0 ; digit < 10; ++digit) {
    digits.push_back(digit);
}
```

To accumulate the digits and form the complete code I use another string stream. This time, though, instead of converting from string to integer, I am doing to inverse operation: for each digit to generate, I move to a random vector's position, convert and append the digit to the code and finally remove the digit from the list of available digits.

In this case is easier to use directly the `vector` iterator to get the digit's value than to use `vector::at` with the index because to erase the element from the list I require an iterator anyways.

To get a random iterator from the list, then, I get the iterator of the first digit and I move forward the iterator a random number between 0 and the current list's size. To get this random number I map the output of the `rand` function's result [0,`MAX_RAND`) range to [0, list's size) using the modulo.

```
<<generate secret code string>>=
<<build list of available digits>>
std::ostringstream secretCode;
for (unsigned int codeDigit = 0 ; codeDigit < length ; ++codeDigit)
{
    std::vector<unsigned int>::iterator digit = digits.begin() + (rand() % digits.size());
    secretCode << *digit;
    digits.erase(digit);
}
```

The standard C function `rand` is declared in the `cstdlib` header file already included, but before using this function is it best to initialize the random number generate with a *seed* by calling the `srand` function. This seed is used by the `rand` function to generate a sequence of random numbers. Therefore, this seed needs to be different each time the application runs. A good enough seed value in this case is to use the current time in which the application begins. Such a value can be retrieved by the standard C `time` function declared in the `ctime` header.

```
<<includes>>=
#include <ctime>
```

```
<<initialize random number generator>>=
srand(time(0));
```

The last thing to do to generate the secret code is to return the resulting string after concatenating the digits. As the function is expected to return an `string` and not an `ostringstream`, I have to retrieve the string that the stream built for me.

```
<<return secret code string>>=
return secretCode.str();
```

3 Player's Input

With the secret code generated, now I am in the position to ask the player her guess. Like the secret code, the guess is an string of digits and has the same length as the secret code. I store the guess in a string because is easier to compare with the secret code and to know its length.

```
<<ask for the player guess>>=
std::string guess = get_player_guess(secretCodeLength);
```

Even though the guess is ultimately an string, I need to get only digits from the player. Here again I am leveraging this to the stream class, although in this case the stream is `cin` instead of a string stream. And so, I need to be more cautious because now the stream can enter in a *failed* state due to conversion errors.

If I try to extract an integer from the stream and it fails, this means that the player entered something that is not a number. Errors happen, so I want to ask the player to enter the guess again, explaining that the previous input was not correct. But when the extraction couldn't be performed, `cin` gets marked as *failed* and I can't read anything else from it until I clear the stream before trying to read again. Also, when there is an error, I tell `cin` to ignore everything else that could still be in the stream's buffer after the failure until it find an end of line character (\n).

I have to be aware that the user could close the standard input instead of entering a guess and then I am unable to read anymore. In this case, I thrown an exception.

```
<<make sure the player enters an integer>>=
unsigned int guessNumber;
do
{
    std::cout << "Enter a guess (" << length << " digits): ";
    if (std::cin.fail())
    {
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    }
    std::cin >> guessNumber;
    if (std::cin.fail())
    {
        std::cout << "This is not a valid number\n";
    }
}
while(std::cin.fail() && !std::cin.eof());
if (std::cin.eof())
{
    throw std::runtime_error("Can't read from standard input");
}
```

The numeric limits class is defined in the `limits` header.

```
<<includes>>=
#include <limits>
```

After I make sure that the player entered only digits, I can convert this guess to a string using again a string stream. After converting the guess to a string, I can compare its length to the secret code's length and inform the user if the guess' length is not valid. If the length is *less* than the required digits, then I assume that the user entered or wants to enter zeros in front of the digits and pad the string with `0` until fill the minimum width.

```
<<convert the guess to string and check its length>>=
std::ostringstream guessStream;
guessStream.fill('0');
guessStream.width(length);
guessStream << guessNumber;
guess = guessStream.str();
if (guess.size() != length)
{
    std::cout << "Invalid guess.  It must be " << length << " digits\n";
}
```

I must keep asking for the guess until the player enters a numeric guess whose size matches the secret code's length. This function's precondition is that the length must be greater than 0 for this to work.

```
<<function to read player guess>>=
std::string
get_player_guess(unsigned int length)
{
    assert(length > 0 && "Tried to get a zero-length guess");

    std::string guess;
    do
    {
        <<make sure the player enters an integer>>
        <<convert the guess to string and check its length>>
    }
    while (guess.size() != length);

    return guess;
}
```

4 Counting Bulls and Cows

To count how many bulls and cows there are in the guess, I loop on each position of the secret code and look at the guess at the same position. If the characters are the same, then I've found a bull. Otherwise, I have to check whether the character is *at any position* in the guess. If so, then I've got a cow.

```
<<count bulls and cows>>=
unsigned int bulls = 0;
unsigned int cows = 0;
for (size_t codeIndex = 0 ; codeIndex < secretCode.size() ; ++codeIndex)
{
    if (secretCode[codeIndex] == guess[codeIndex])
    {
        ++bulls;
    }
    else if (guess.find(secretCode[codeIndex]) != std::string::npos)
    {
        ++cows;
    }
}
```

This assumes that `secretCode` and `guess` are of the same length, which is the precondition imposed by the function that computes the number of bulls and cows. If this precondition, albeit checked in debug mode by the assert, is violated then the result is undefined.

Contrarily, when both the `secretCode` and `guess` have the same length, then the function returns the number of bulls and cows. Given that C++ can't return multiple values, I need to pack these two values in a single element. My choices are to create

a new `struct` that holds the two values or use the standard `pair` class. I usually don't like `pair` very much because its two member attributes are named `first` and `second`, and those names don't give me a clue of what they contain. But in this case, since I only use them to allow me return multiple values, it is fine.

The `pair` class and its helper function, `make_pair`, are both defined in the `utility` header.

```
<<includes>>=
#include <utility>
```

```
<<function to compute the number of bulls and cows>>=
std::pair<unsigned int, unsigned int>
compute_bulls_and_cows(const std::string &secretCode, const std::string &guess)
{
    assert (secretCode.size() == guess.size() && "The secret code's and guess' length must ↩
        be equal");

    <<count bulls and cows>>
    return std::make_pair(bulls, cows);
}
```

The return of this function then can be split again into two variables that I later print onto the console as the guess' result.

```
<<check how many bulls and cows there are in the guess>>=
std::pair<unsigned int, unsigned int> result = compute_bulls_and_cows(secretCode, guess);
bulls = result.first;
cows = result.second;
std::cout << "Result: " << bulls << " bulls, " << cows << " cows\n\n";
```

The `bulls` and `cows` variables are initialized to `0` each time the game begins.

```
<<reset number of bulls and cows>>=
unsigned int bulls = 0;
unsigned int cows = 0;
```

The game must keep asking for guesses until the guess is equal to the secret code, which is the same to say that the number of bulls are equal to the secret code's length. But checking equality with integers is, generally, faster than with strings. In this case though, with strings of max. 6 characters, the difference is microscopic.

```
<<until the secret code is discovered>>=
while (bulls != secretCodeLength);
```

5 Keep Playing

The last remaining thing to do is to tell the player that she won, once she found the secret code, and ask her whether to keep playing or not. Here, I am reading a single character from the standard input. If the player enters `y`, then I stay. Any other input, even an error, means no and then I quit the game.

```
<<ask whether to play again>>=
std::cout << "You found the secret code!\nWant to play again (y/n)? ";
unsigned char reply = 'n';
std::cin >> reply;
playing = reply == 'y';
std::cout << std::endl;
```

A Makefile

Being a simple application, an small Makefile would be sufficient to build and link `bulls and cows` from the source document.

The first thing that needs to be done is to extract and tangle the C++ source therefore, to have a `atangle` installed.

```
<<extract cpp source code>>=
bullsandcows.cpp: bullsandcows.txt
        atangle -r $@ $< > $@
```

Then is possible to link the executable from the extracted C++ source code. Although I have to take into account the platform executable suffix. For Linux and other UNIX systems, the suffix is the empty string, but for Windows I need to append *.exe* to the executable name.

To know which system is the executable being build, I'll use the `uname -s` command, available both in Linux and also in MinGW or Cygwin for Windows. In this case, I only detect the presence of MinGW because I don't want to add yet another dependency to Cygwin's DLL.

```
<<determine executable suffix>>=
UNAME = $(shell uname -s)
MINGW = $(findstring MINGW32, $(UNAME))
```

Later, I just need to check if the substring *MINGW* is contained in the output of `uname`. If the `findstring` call's result is the empty string, then we assume we are building in a platform that doesn't have executable suffix.

```
<<determine executable suffix>>=
ifneq ($(MINGW),)
EXE := .exe
endif
```

With this suffix, I can now build the final executable.

```
<<build bullsandcows executable>>=
bullsandcows$(EXE): bullsandcows.cpp
        g++ -o $@ $<
```

Sometimes, it is necessary to remove the executable as well as the intermediary build artifacts. For this reason, I'll add a target named `clean` that will remove all the files built by the Makefile and only left the original document. I have to mark this target as `PHONY` in case there is a file named `clean` in the same directory as the Makefile.

```
<<clean build artifacts>>=
.PHONY: clean

clean:
        rm -f bullsandcows$(EXE) bullsandcows.cpp
```

As the first defined target is the Makefile's default target, I write the executable first in the Makefile and then all the dependences. After all source code targets, it comes the `clean` target. This is not required, but a personal choice. The final Makefile's structure is thus the following.

```
<<Makefile>>=
<<determine executable suffix>>

<<build bullsandcows executable>>

<<extract cpp source code>>

<<clean build artifacts>>
```

B License

This program is distributed under the terms of the GNU General Public License (GPL) version 2.0 as follows:

```
<<license>>=
// This program is free software; you can redistribute it and/or modify
// it under the terms of the GNU General Public License version 2.0 as
// published by the Free Software Foundation.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307  USA
```