Министерство цифрового развития, связи и массовых коммуникаций
Федеральное Государственное Бюджетное Образовательное Учреждение Высшего Образования «Сибирский Государственный Университет Телекоммуникаций и Информатики»

Кафедра вычислительных систем

Курсовая работа
по дисциплине «Технологии разработки программного обеспечения»
на тему «Генератор паролей»

Выполнил:
ст. гр. ИС-142
Наумов Алексей

Проверил:
ст. преподаватель Токмашева Е.И.

Новосибирск, 2022

Содержание

Введение и постановка задачи 3
Техническое задание 3
Описание выполненного проекта..... 5
Общая структура……..... 5
Получение гарантированного символа..... 5
Получение символов общего алфавита..... 6
Получение случайного символа 7
Хеширование 7
Guards (проверки для предотвращения ошибок) 7
Unit-тесты 8
Личный вклад в проект..... 9
Приложение. Листинг программы 10

Введение и постановка задачи

Разработать генератор паролей, работающий в терминале (командной строке), принимающий аргументы командной строки при запуске программы, в зависимости от которых генерируются пароли с заданными опциями.

Техническое задание

1. **Функциональность**

Реализуемый продукт — генератор паролей для терминала *(командной строки)*. Используется для создания устойчивых ко взлому паролей, опции к которым можно задать в аргументах к программе. С помощью аргументов можно задать длину паролей в символах, получить разное их количество. Пароли могут быть состоящими как непосредственно из цифр и букв латинского алфавита, так и с использованием специальных символов *(!, $, %, & и т.д.)*.

Сценарии использования (примеры):

Генерация одного пароля стандартной длины (8) со стандартными опциями:

```
$ pawg
akbspiun
```

Генерация трёх паролей длиной в 10 символов со стандартными опциями:

```
$ pawg 3 10
aslkrjnabs fajvmmcnal dffjasjwfg
```

Генерация пяти паролей длиной в 5 символов с хотя бы одной цифрой и заглавной буквой, вывод результата в столбик:

```
$ pawg 5 5 -c -n -1
1H9fA
j4cNA
3D9ja
LA01B
kK5aq
```

Генерация трёх паролей длиной в 5 символов, основанных на хешировании строки hello, вывод в столбик:

```
$ pawg 3 5 -1 -sha1 hello
aaf4c
9cf5c
1eec5
```

Генерация одного пароля длиной в 10 символов, используя спецсимволы и цифры:

```
$ pawg 1 10 -y -n
aa2jk&js1!
```

2. **Формат входных данных**

   $ pawg [N] [length] [args]
   N — количество генерируемых паролей
   length — длина пароля
   args — аргументы командной строки

3. **Интерфейс**
   Программа является приложением для терминала (командной строки), соответственно никакого графического интерфейса не требует.

4. **Аргументы командной строки**

Аргумент	**Расшифровка**	**Назначение**
-h	help	Вывести справку и помощь по программе
-c	capitalize	Включить в пароль хотя бы одну букву верхнего регистра
-n	numbers	Включить в пароль хотя бы одну цифру
-1	one column	Вывести пароли в один столбец
-y	special symbols	Включить в пароль хотя бы один специальный символ
-s	secure	Сгенерировать хаотичный пароль со всеми возможными символами *(алиас флагов -n -y -c)*
-H *(-sha1)*	SHA-1 Hash	Использовать хеш SHA-1 заданного сида *(строки)* как генератор непроизвольных знаков. Запомнив генерирующий сид, можно восстановить пароль или серию паролей.

Описание выполненного проекта

Общая структура

Для составления пароля используются 2 массива: массив символов **char password[length]** и массив целых чисел **int cells[length]**. В первый помещаются символы генерируемого пароля, во второй — заполнение его ячеек *(1 / 0)*. Для работы с опциями для генерации используется структура Options и функции **initOptions** и **getOptions** для инициализации стандартных опций и получения новых из аргументов.

```
 1 struct Option {
 2     int character_options;
 3     int numeric;
 4     int capitalized;
 5     int special;
 6     int size;
 7     int count;
 8     int column;
 9     int hash;
10     char* seed;
11 };
```

```
 1 struct Option initOptions()
 2 {
 3     struct Option option;
 4     option.size = 8;
 5     option.count = 1;
 6     option.numeric = 0;
 7     option.capitalized = 0;
 8     option.special = 0;
 9     option.column = 0;
10     option.character_options = 0;
11     option.seed = malloc(sizeof(char));
12     return option;
13 }
14
15 struct Option getOptions(struct Option option, int argc, char** argv)
16 {
17     if (argc > 1) {
18         for (int i = 1; i < argc; i++) {
19             if (strcmp(argv[i], "-c") == 0) {
20                 option.capitalized = 1;
21             … < another arguments checking >
```

(перебор введённых аргументов)

Получение гарантированного символа

При наличии флага *(опции)* для генерации пароля, функцией **getGuaranteedIndex** выбирается случайная ячейка с индексом **i**, для которой выполняется проверка, заполнена она или нет (**cells[i] == 1** ?).
Если ячейка не заполнена (**cells[i] = 0**), в неё помещается случайный символ из алфавита заданной опции (для **-c** это от A до Z, для **-n**: от 0 до 9, для -y: от ! до +).

```
 1 int getGuaranteedIndex(int* cells, int size)
 2 {
 3     int guaranteed = getRandom(0, size - 1);
 4     if (cells[guaranteed] == 1) {
 5         while (cells[guaranteed] != 0) {
 6             int i = 1;
 7             guaranteed = getRandom(0, size - 1);
 8             i++;
 9         }
10     }
11     return guaranteed;
12 }
```

Получение символов общего алфавита

При отсутствии дополнительных аргументов (флагов), оставшиеся ячейки заполняются с помощью функции **getFromAlphabet** через цикл **for**, в котором выполняется проверка, заполнена ячейка или нет.

Если ячейка не заполнена, для неё подбирается значение из общего алфавита (для аргументов **-n -c** это, например, **(a-z)+(0-9)+(A-Z)**).

```
 1 char getFromAlphabet(struct Option option)
 2 {
 3     char letter = '~';
 4     for (int i = 0; letter == '~'; i++) {
 5         int characterType = getRandom(1, 4);
 6         switch (characterType) {
 7         case 1:
 8             letter = getLowercase();
 9             break;
10         case 2:
11             if (option.capitalized == 1) {
12                 letter = getCapital();
13             }
14             break;
15         case 3:
16             if (option.numeric == 1) {
17                 letter = getNumber();
18             }
19             break;
20         case 4:
21             if (option.special == 1) {
22                 letter = getSpecial();
23             }
24             break;
25         }
26     }
27     return letter;
28 }
```

Подбор случайного символа

Значения для генерации псевдослучайных символов подбираются по генерации псевдослучайных чисел функцией **getRandom**, используемых для получения определённых символов из ASCII. Для специальных символов это значения от 33 до 47 в функции **getSpecial**, для цифр - от 48 до 57 в функции **getNumber**, для латинских букв верхнего регистра - от 65 до 90 в функции **getCapital**, нижнего регистра - от 97 до 122 в функции **getLowercase**.

```
1 int getRandom(int lower, int upper)
2 {
3     return (rand() % (upper - lower + 1)) + lower;
4 }
```

```
 1 char getLowercase()
 2 {
 3     return (char)getRandom(97, 122);
 4 }
 5
 6 char getCapital()
 7 {
 8     return (char)getRandom(65, 90);
 9 }
10
11 char getNumber()
12 {
13     return (char)getRandom(48, 57);
14 }
15
16 char getSpecial()
17 {
18     return (char)getRandom(33, 47);
19 }
```

Хеширование

При генерации паролей из заданного сида *(строки)* используется внешняя библиотека <sha1.h>, взятая с ресурса **IETF (Internet Engineering Task Force)**.

Алгоритм:

1. Получение количества паролей, длины паролей и генерирующей строки *(сида)* в аргументах: **$ pawg <N> <length> -sha1 <seed>**
2. Создание массива хешей по количеству паролей: **char hashed[N][SHA1-LENGTH]**
3. Получение полного хеша SHA1 из строки: **hashed[0] = SHA1(seed)**
4. Генерация последующих полных хешей, где **hashed[k] = SHA1(hashed[k-1])**
5. Вывод первых **length** символов для пароля от каждого хеша

```
1 if (option.hash == 1) {
```

```
2     char* hash = getHashFromSeed(option.seed);
3     option.seed = hash;
4     result = malloc(sizeof(char) * option.size);
5     for (int j = 0; j < option.size; j++) {
6         result[j] = hash[j];
7     }
8 }
```

Таким образом, при утере пароля и количества символов в нём, пользователь сможет восстановить пароль/серию паролей, используя свою генерирующую строку.

Guards (проверки для предотвращения ошибок)

1. Длина пароля должна быть больше количества его символьных опций (e.g. **-n -c -y**)
2. Длина пароля при генерации хеша не может быть больше 40, т.е. длины хеша SHA1
3. При получении невалидного аргумента выводится предупреждение о несчитывании аргумента и его пропуск. Программа в этих случаях работает корректно и даже при отсутствии валидных аргументов просто сгенерирует стандартный пароль.
4. При наличии флага генерации пароля из хеша SHA1, но отсутствии сида, выводится справка об использовании программы.

```
 1 if (option.hash == 1 && option.size > 40) {
 2         printf("max hash password length equals to 40 : letters after
   40'th won't generate\n");
 3     }
 4
 5 if (strcmp(argv[i], "-H") == 0 || strcmp(argv[i], "-sha1") == 0) {
 6     if (!argv[i + 1]) {
 7         printHelp();
 8         option.size = 0;
 9     }
10 }
11
12 if (option.character_options > option.size) {
13     printf("size of password is less than character options you've
   specified\n");
14     option.size = 0;
15 }
16
17 else {
18     printf("can't recognise parameter %s : skipping it\n",
19     argv[i]);
20 }
```

Unit-тесты

В работе используется библиотека **ctest.h** для модульного тестирования всего приложения. Так как программа разделена на модули и отдельные функции, тесты покрывают сценарии верной/неверной работы и пользовательского ввода во всех

реализованных функциях.

Примеры (экземпляры из файлов main.c и test.c): (скриншот полного выполнения всех тестов)

```
allenvox@MacBook-Pro-Yuriy cw-is-142_password-generator % bin/pawg-test
TEST 1/24 check_random:valid [OK]
TEST 2/24 check_random:invalid [OK]
TEST 3/24 check_isnumber:valid [OK]
TEST 4/24 check_isnumber:invalid [OK]
TEST 5/24 check_initoptions:valid [OK]
TEST 6/24 check_initoptions:invalid [OK]
TEST 7/24 check_getoptions:valid [OK]
TEST 8/24 check_getoptions:invalid can't recognise parameter abc : skipping it
[OK]
TEST 9/24 check_sha1:valid [OK]
TEST 10/24 check_sha1:invalid [OK]
TEST 11/24 check_getlowercase:valid [OK]
TEST 12/24 check_getlowercase:invalid [OK]
TEST 13/24 check_getcapital:valid [OK]
TEST 14/24 check_getcapital:invalid [OK]
TEST 15/24 check_getspecial:valid [OK]
TEST 16/24 check_getspecial:invalid [OK]
TEST 17/24 check_getnumber:valid [OK]
TEST 18/24 check_getnumber:invalid [OK]
TEST 19/24 check_getfromalphabet:valid1 [OK]
TEST 20/24 check_getfromalphabet:valid2 [OK]
TEST 21/24 check_getfromalphabet:invalid1 [OK]
TEST 22/24 check_getfromalphabet:invalid2 [OK]
TEST 23/24 check_getguaranteedindex:valid [OK]
TEST 24/24 check_getguaranteedindex:invalid [OK]
RESULTS: 24 tests (24 ok, 0 failed, 0 skipped) ran in 0 ms
```

```c
#define CTEST_MAIN
#define CTEST_SEGFAULT
#define CTEST_COLOR_OK

#include <ctest.h>

int main(int argc, const char** argv)
{
    return ctest_main(argc, argv);
}
```

Тестируемые случаи ниже — проверка функций **getRandom** и **isNumber** из файлов **random.c** и **input.c**. Тест с меткой *valid* завершается с положительным результатом, если полученный результат находится в диапазоне от 0 до 100 — мы завершаем его со сравнением полученных значений функцией **ASSERT_EQUAL** (из **ctest.h**), с меткой *invalid* — если результат вне диапазона (некорректный), и в переменную **expected** (ожидаемое), что он будет в диапазоне. Для выполнения этого теста функция видоизменяется на **ASSERT_NOT_EQUAL**, чтобы убедиться, что результат выполнения функции является верным.

В функцию **isNumber**, тесте *valid*, подаётся верная входная информация, являющаяся числом, но записанным как строка. Тест завершается при получении функцией ожидаемого значения 0, говорящее о том, что строка является числом. *Invalid*-тест получает некорректную входную информацию (строку, содержащую буквы), и завершается сравнением результата с ожиданием результата, равного 1, которая обозначает, что подаваемая строка не является числом.

```c
#include "random.h"
#include "input.h"
#include <ctest.h>

CTEST(check_random, valid)
{
    int value = getRandom(0, 100);
    int result = value >= 0 && value <= 100;
    int expected = 1;
    ASSERT_EQUAL(expected, result);
}

CTEST(check_random, invalid)
{
    int value = getRandom(0, 100);
    int result = value <= 0 || value >= 100;
    int expected = 1;
    ASSERT_NOT_EQUAL(expected, result);
}

```

```
21 CTEST(check_isnumber, valid)
22 {
23     int result = isNumber("1000");
24     int expected = 0;
25     ASSERT_EQUAL(expected, result);
26 }
27
28 CTEST(check_isnumber, invalid)
29 {
30     int result = isNumber("AVC");
31     int expected = 1;
32     ASSERT_EQUAL(expected, result);
33 }
```

Личный вклад в проект

При выполнении данной работы я покрывал всё приложение тестами, исправлял ошибки в коде, разрабатывал техническое задание, проверял работу приложения на разных платформах, создавал функции подбора случайных символов, генерировал идеи для создания аргументов командной строки.

Приложение. Листинг программы

Makefile

```
 1 CFLAGS = -Wall -Wextra -I src -I thirdparty -MMD
 2 DIRGUARD = @mkdir -p $(@D)
 3
 4 all: bin/pawg
 5 -include obj/*.d
 6
 7 bin/pawg: obj/main.o obj/libpawg.a
 8         $(DIRGUARD)
 9         gcc $(CFLAGS) -o $@ $^
10
11 obj/%.o: src/%.c
12         $(DIRGUARD)
13         gcc $(CFLAGS) -c -o $@ $<
14
15 obj/sha1.o: thirdparty/sha1.c
16         gcc $(CFLAGS) -c -o $@ $<
17
18 .PHONY: clean
19 clean:
20         rm -rf obj/ bin/
21
22 .PHONY: test
23 test: bin/pawg-test
24
25 bin/pawg-test: obj/test/main.o obj/test/test.o obj/libpawg.a
26         $(DIRGUARD)
27         gcc $(CFLAGS) -o $@ $^
28
```

```
obj/test/%.o: test/%.c
	$(DIRGUARD)
	gcc $(CFLAGS) -c -o $@ $<

obj/libpawg.a: obj/alphabetic.o obj/random.o obj/hash.o obj/sha1.o
obj/input.o obj/password.o
	ar rcs $@ $^
```

src/main.c

```c
#include "alphabetic.h"
#include "hash.h"
#include "input.h"
#include "password.h"
#include "random.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char** argv)
{
    srand(time(NULL));
    struct Option option = initOptions();
    option = getOptions(option, argc, argv);
    if (option.size == 0) {
        printHelp();
        return 0;
    }
    char* result;
    if (option.hash == 1 && option.size > 40) {
        printf("\e[4mmax hash password length equals to 40 : letters after
"
               "40'th won't generate\e[0m\n");
    }
    for (int i = 0; i < option.count; i++) {
        if (option.hash == 1) {
            char* hash = getHashFromSeed(option.seed);
            option.seed = hash;
            result = malloc(sizeof(char) * option.size);
            for (int j = 0; j < option.size; j++) {
                result[j] = hash[j];
            }
        } else {
            result = generatePassword(option);
        }
        printf("%s ", result);
        if (option.column == 1 || i == option.count - 1) {
            printf("\n");
        }
    }
    free(result);
    return 0;
}
```

src/alphabetic.c

```c
#include "alphabetic.h"
#include "input.h"
#include "random.h"
#include <stdio.h>
#include <string.h>

char getLowercase()
{
    return (char)getRandom(97, 122);
}

char getCapital()
{
    return (char)getRandom(65, 90);
}

char getNumber()
{
    return (char)getRandom(48, 57);
}

char getSpecial()
{
    return (char)getRandom(33, 47);
}

char getFromAlphabet(struct Option option)
{
    char letter = '~';
    for (int i = 0; letter == '~'; i++) {
        int characterType = getRandom(1, 4);
        switch (characterType) {
        case 1:
            letter = getLowercase();
            break;
        case 2:
            if (option.capitalized == 1) {
                letter = getCapital();
            }
            break;
        case 3:
            if (option.numeric == 1) {
                letter = getNumber();
            }
            break;
        case 4:
            if (option.special == 1) {
                letter = getSpecial();
            }
            break;
        }
    }
    return letter;
}
```

src/alphabetic.h

```h
#pragma once

#include "input.h"

char getLowercase();
char getCapital();
char getFromAlphabet(struct Option option);
char getNumber();
char getSpecial();
```

src/input.c

```c
#include "input.h"
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void printHelp()
{
    printf("\n Usage: pawg [N] [L] [args...]\n");
    printf("\t(to generate N passwords with L-digit length and specific "
           "options included in args)\n\n");
    printf(" Arguments:\n");
    printf("\t-h\t  : display help\n");
    printf("\t-c\t  : include at least one uppercase letter\n");
    printf("\t-n\t  : include at least one number\n");
    printf("\t-y\t  : include at least one special character\n");
    printf("\t-s\t  : generate a secure password including all characters
"
           "(alias for -c -n -y)\n");
    printf("\t-H <seed> : generate password based on sha1-hash of
seed\n\n");
}

int isNumber(char* input)
{
    for (int i = 0; input[i]; i++) {
        if (!isdigit(input[i])) {
            return 1;
        }
    }
    return 0;
}

struct Option initOptions()
{
    struct Option option;
    option.size = 8;
    option.count = 1;
    option.numeric = 0;
    option.capitalized = 0;
    option.special = 0;
    option.column = 0;
```

```
43      option.character_options = 0;
44      option.seed = malloc(sizeof(char));
45      return option;
46  }
47
48  struct Option getOptions(struct Option option, int argc, char** argv)
49  {
50      int numargs = 0;
51      if (argc > 1) {
52          for (int i = 1; i < argc; i++) {
53              if (strcmp(argv[i], "-c") == 0) {
54                  option.capitalized = 1;
55                  if (option.character_options != 3) {
56                      option.character_options++;
57                  }
58              } else if (strcmp(argv[i], "-n") == 0) {
59                  option.numeric = 1;
60                  if (option.character_options != 3) {
61                      option.character_options++;
62                  }
63              } else if (strcmp(argv[i], "-y") == 0) {
64                  option.special = 1;
65                  if (option.character_options != 3) {
66                      option.character_options++;
67                  }
68              } else if (strcmp(argv[i], "-s") == 0) {
69                  option.capitalized = 1;
70                  option.numeric = 1;
71                  option.special = 1;
72                  option.character_options = 3;
73              } else if (strcmp(argv[i], "-1") == 0) {
74                  option.column = 1;
75              } else if (
76                      strcmp(argv[i], "-H") == 0
77                      || strcmp(argv[i], "-sha1") == 0) {
78                  if (argc - 1 == i) {
79                      option.size = 0;
80                  } else if (argv[i + 1] != NULL) {
81                      option.hash = 1;
82                      strcpy(option.seed, argv[i + 1]);
83                  }
84              } else if (strcmp(argv[i], "-h") == 0) {
85                  option.size = 0;
86              } else if (isNumber(argv[i]) == 0) {
87                  int number = atoi(argv[i]);
88                  if (numargs == 0) {
89                      option.count = number;
90                      numargs++;
91                  } else {
92                      option.size = number;
93                  }
94              } else if (
95                      strcmp(argv[i - 1], "-H") == 0
96                      || strcmp(argv[i - 1], "-sha1") == 0) {
97              } else {
```

```
 98                 printf("\e[4mcan't recognise parameter %s : skipping
 99 it\e[0m\n",
100                        argv[i]);
101             }
102         }
103     }
104     if (option.character_options > option.size) {
105         printf("\e[4msize of password is less than character options
106 you've "
107                "specified\e[0m\n");
            option.size = 0;
        }
        return option;
    }
```

src/input.h

```
 1 #pragma once
 2
 3 struct Option {
 4     int character_options;
 5     int numeric;
 6     int capitalized;
 7     int special;
 8     int size;
 9     int count;
10     int column;
11     int hash;
12     char* seed;
13 };
14
15 void printHelp();
16 int isNumber(char* input);
17 struct Option initOptions();
18 struct Option getOptions(struct Option option, int argc, char** argv);
```

src/password.c

```
 1 #include "password.h"
 2 #include "alphabetic.h"
 3 #include "input.h"
 4 #include "random.h"
 5 #include <stdio.h>
 6 #include <stdlib.h>
 7
 8 int getGuaranteedIndex(int* cells, int size)
 9 {
10     int guaranteed = getRandom(0, size - 1);
11     if (cells[guaranteed] == 1) {
12         while (cells[guaranteed] != 0) {
13             int i = 1;
14             guaranteed = getRandom(0, size - 1);
15             i++;
16         }
17     }
```

```
18      return guaranteed;
19  }
20
21  char* generatePassword(struct Option option)
22  {
23      int size = option.size;
24      char* password = malloc(sizeof(char) * size);
25      int* cells = malloc(sizeof(int) * size);
26
27      for (int i = 0; i < size; i++) {
28          cells[i] = 0;
29      }
30
31      if (option.capitalized == 1) {
32          int index = getGuaranteedIndex(cells, size);
33          password[index] = getCapital();
34      }
35
36      if (option.numeric == 1) {
37          int index = getGuaranteedIndex(cells, size);
38          password[index] = getNumber();
39      }
40
41      if (option.special == 1) {
42          int index = getGuaranteedIndex(cells, size);
43          password[index] = getSpecial();
44      }
45
46      for (int i = 0; i < size; i++) {
47          if (cells[i] == 0) {
48              password[i] = getFromAlphabet(option);
49          }
50      }
51
52      free(cells);
53      return password;
54  }
```

src/password.h

```
1  #pragma once
2
3  #include "input.h"
4
5  int getGuaranteedIndex(int* cells, int size);
6  char* generatePassword(struct Option option);
```

src/hash.c

```
 1  #include "hash.h"
 2  #include "sha1.h"
 3  #include <stdint.h>
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6  #include <string.h>
```

```
 7
 8 char* getHashFromSeed(char* seed)
 9 {
10     unsigned char* input = (unsigned char*)seed;
11     SHA1Context sha;
12     uint8_t Message_Digest[20];
13     int err = SHA1Reset(&sha);
14     if (err) {
15         printf("hash reset error\n");
16     }
17     err = SHA1Input(&sha, input, strlen((const char*)input));
18     if (err) {
19         printf("hash input error\n");
20     }
21     err = SHA1Result(&sha, Message_Digest);
22     if (err) {
23         printf("hash result error\n");
24     }
25     char* output = malloc(sizeof(char) * 41);
26     for (int i = 0; i < 20; i++) {
27         char segment[3];
28         sprintf(segment, "%02x", Message_Digest[i]);
29         output[i * 2] = segment[0];
30         output[i * 2 + 1] = segment[1];
31     }
32     output[40] = '\0';
33     return output;
34 }
```

src/hash.h

```
1 #pragma once
2
3 char* getHashFromSeed(char* seed);
```

test/main.c

```
 1 #define CTEST_MAIN
 2 #define CTEST_SEGFAULT
 3 #define CTEST_COLOR_OK
 4
 5 #include <ctest.h>
 6
 7 int main(int argc, const char** argv)
 8 {
 9     return ctest_main(argc, argv);
10 }
```

test/test.c

```
 1 #include "alphabetic.h"
 2 #include "hash.h"
 3 #include "input.h"
 4 #include "password.h"
 5 #include "random.h"
```

```
 6 #include <ctest.h>
 7 #include <string.h>
 8
 9 CTEST(check_random, valid)
10 {
11     int value = getRandom(0, 100);
12     int result = value > 0 && value < 100;
13     int expected = 1;
14     ASSERT_EQUAL(expected, result);
15 }
16
17 CTEST(check_random, invalid)
18 {
19     int value = getRandom(0, 100);
20     int result = value < 0 || value > 100;
21     int expected = 0;
22     ASSERT_EQUAL(expected, result);
23 }
24
25 CTEST(check_isnumber, valid)
26 {
27     int result = isNumber("1000");
28     int expected = 0;
29     ASSERT_EQUAL(expected, result);
30 }
31
32 CTEST(check_isnumber, invalid)
33 {
34     int result = isNumber("AVC");
35     int expected = 1;
36     ASSERT_EQUAL(expected, result);
37 }
38
39 CTEST(check_initoptions, valid)
40 {
41     struct Option option = initOptions();
42     int result = option.size == 8 && option.capitalized == 0
               && option.numeric == 0 && option.special == 0 &&
   option.column == 0
               && option.character_options == 0 && option.hash == 0;
43     int expected = 0;
44     ASSERT_EQUAL(expected, result);
45 }
46
47 CTEST(check_initoptions, invalid)
48 {
49     struct Option option = initOptions();
50     int result = option.size != 8 || option.capitalized != 0
               || option.numeric != 0 || option.special != 0 ||
   option.column != 0
               || option.character_options != 0 || option.hash != 0;
51     int expected = 1;
52     ASSERT_EQUAL(expected, result);
53 }
54
```

```
 55 CTEST(check_getoptions, valid)
 56 {
 57     struct Option option = initOptions();
 58     char* argv[3];
 59     argv[0] = "pawg";
 60     argv[1] = "-s";
 61     argv[2] = "-n";
 62     option = getOptions(option, 3, argv);
 63     int result = option.capitalized && option.numeric && option.special;
 64     int expected = 1;
 65     ASSERT_EQUAL(expected, result);
 66 }
 67
 68 CTEST(check_getoptions, invalid)
 69 {
 70     struct Option option = initOptions();
 71     char* argv[2];
 72     argv[0] = "pawg";
 73     argv[1] = "abc";
 74     option = getOptions(option, 2, argv);
 75     int result = option.character_options;
 76     int expected = 0;
 77     ASSERT_EQUAL(expected, result);
 78 }
 79
 80 CTEST(check_sha1, valid)
 81 {
 82     int result = strcmp(
                "aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d",
                getHashFromSeed("hello"));
 83     int expected = 0;
 84     ASSERT_EQUAL(expected, result);
 85 }
 86
 87 CTEST(check_sha1, invalid)
 88 {
 89     int result = strcmp(
                "aaf4c61ddcc5e8a2dabede0f3b482cdea9434d",
    getHashFromSeed("hello"));
 90     int expected = 0;
 91     ASSERT_NOT_EQUAL(expected, result);
 92 }
 93
 94 CTEST(check_getlowercase, valid)
 95 {
 96     int result = (int)getLowercase() <= 122 && (int)getLowercase() >= 97;
 97     int expected = 1;
 98     ASSERT_EQUAL(expected, result);
 99 }
100
101 CTEST(check_getlowercase, invalid)
102 {
103     int result = (int)getLowercase() > 122 || (int)getLowercase() < 97;
104     int expected = 0;
105     ASSERT_EQUAL(expected, result);
```

```
106 }
107
108 CTEST(check_getcapital, valid)
109 {
110     int result = (int)getCapital() <= 90 && (int)getCapital() >= 65;
111     int expected = 1;
112     ASSERT_EQUAL(expected, result);
113 }
114
115 CTEST(check_getcapital, invalid)
116 {
117     int result = (int)getCapital() > 90 || (int)getCapital() < 65;
118     int expected = 0;
119     ASSERT_EQUAL(expected, result);
120 }
121
122 CTEST(check_getspecial, valid)
123 {
124     int result = (int)getSpecial() <= 47 && (int)getSpecial() >= 33;
125     int expected = 1;
126     ASSERT_EQUAL(expected, result);
127 }
128
129 CTEST(check_getspecial, invalid)
130 {
131     int result = (int)getSpecial() > 47 || (int)getSpecial() < 33;
132     int expected = 0;
133     ASSERT_EQUAL(expected, result);
134 }
135
136 CTEST(check_getnumber, valid)
137 {
138     int result = (int)getNumber() <= 57 && (int)getNumber() >= 48;
139     int expected = 1;
140     ASSERT_EQUAL(expected, result);
141 }
142
143 CTEST(check_getnumber, invalid)
144 {
145     int result = (int)getNumber() > 57 || (int)getNumber() < 48;
146     int expected = 0;
147     ASSERT_EQUAL(expected, result);
148 }
149
150 CTEST(check_getfromalphabet, valid1)
151 {
152     struct Option option = initOptions();
153     int result
154             = getFromAlphabet(option) >= 97 && getFromAlphabet(option) <=
    122;
155     int expected = 1;
156     ASSERT_EQUAL(expected, result);
157 }
158
159 CTEST(check_getfromalphabet, valid2)
```

```
160 {
161     struct Option option = initOptions();
162     option.capitalized = 1;
163     option.special = 1;
164     option.numeric = 1;
165     char s = getFromAlphabet(option);
166     int cap = s >= 65 && s <= 90;
167     int low = s >= 97 && s <= 122;
168     int num = s >= 48 && s <= 57;
169     int spec = s >= 33 && s <= 47;
170     int result = cap || low || num || spec;
171     int expected = 1;
172     ASSERT_EQUAL(expected, result);
173 }
174
175 CTEST(check_getfromalphabet, invalid1)
176 {
177     struct Option option = initOptions();
178     int result = getFromAlphabet(option) < 97 || getFromAlphabet(option)
    > 122;
179     int expected = 0;
180     ASSERT_EQUAL(expected, result);
181 }
182
183 CTEST(check_getfromalphabet, invalid2)
184 {
185     struct Option option = initOptions();
186     char s = getFromAlphabet(option);
187     int cap = s >= 65 && s <= 90;
188     int num = s >= 48 && s <= 57;
189     int spec = s >= 33 && s <= 47;
190     int result = cap || num || spec;
191     int expected = 0;
192     ASSERT_EQUAL(expected, result);
193 }
194
195 CTEST(check_getguaranteedindex, valid)
196 {
197     int cells[5];
198     cells[0] = 1;
199     cells[1] = 1;
200     cells[2] = 1;
201     cells[3] = 0;
202     cells[4] = 1;
203     int result = getGuaranteedIndex(cells, 5);
204     int expected = 3;
205     ASSERT_EQUAL(expected, result);
206 }
207
208 CTEST(check_getguaranteedindex, invalid)
209 {
210     int cells[5];
211     cells[0] = 0;
212     cells[1] = 0;
213     cells[2] = 0;
```

```
214     cells[3] = 1;
215     cells[4] = 0;
216     int result = getGuaranteedIndex(cells, 5);
217     int expected = 3;
218     ASSERT_NOT_EQUAL(expected, result);
219 }
```