Testowanie

Franciszek Pietrusiak

Najprostsza kompliacja

```
g++ -o program program.cpp
```

Flagi kompilacji

https://linux.die.net/man/1/g++

- `-o <nazwa_programu>` kompiluje program i nazywa go `<nazwa_programu>`
- `--std=c++17` mówimy kompilatorowi, że używamy wersji 17 C++
- `-Wall` pokazuje wszystkie najważniejsze Warning-i
- `-Wextra` pokazuje niektóre Warning-i, których `-Wall` nie wykrywa
- `-Wshadow` wykrywa nadpisywanie zmiennych z tymi samymi nazwami
- `-Wunused` informuje o zmiennych/funkcjach których nie użyliśmy w programie
- `-O2` optymalizuje kod (nie stosować przy debugowaniu)
- `-O3` jeszcze bardziej optymalizuje kod
- `-fsanitize=address` wykrywa odwoływanie się do niezarezerwowanych obszarów pamięci
- `-g3` do debugowania w gcd
- `-D_GLIBCXX_DEBUG` pomaga w debugowaniu struktur z STL'a

Assert

Przydaje się gdy chcemy upewnić się, że warunki w naszym kodzie w są spełnione. Aby z niego korzystać należy zincludować bibliotekę `assert.h` (jest też w `bits/stdc++.h`).

Dla przykładu:

```
int musi_byc_trzy = 3;
assert(musi_byc_trzy != 3);
```

wtedy po uruchomieniu programu dostajemy:

```
assert: assert.cpp:6: int main(): Assertion `musi_byc_trzy != 3' failed.
```

Czyli dokładnie wiemy co nie działa i w której linijce. Stosowanie assert'ów sprawia, że nasz kod jest bardziej przewidywalny i czytelniejszy.
Dodając `#define NDEBUG` do nagłówku programu można wyłączyć komunikaty assert'ów.

Makefile

Tworzymy plik tekstowy o nazwie `makefile`:

```
CXXFLAGS = -std=c++17 -Wall -Wextra -pedantic -Wshadow\
         -Wfloat-equal -Wshadow -Wconversion -g
```

Potem kompilujemy program poleceniem:

```
make program
```

Najprostszy rand

```
int RAND(int a, int b) {
        return a + rand() % (b-a+1);
}
```

Taki rand potrzebuje seed'a:

```
srand(time(NULL)); // zmienia się co sekunde
srand(stoi(argv[1])); // sami ustalamy seed'a
```

Argumenty do programu

```
int main (int argc, char *argv[])
```

Testerka w bashu:

```
i=0
while true
do
        ./gen $i > in1
        ./brut < in1 > out1
        ./wzo < in1 > out2
        wynik='diff -w -q out1 out2'
        if [ "$wynik" == "" ]; then # diff zwraca 0 jeśli pliki są takie same
                echo "OK $i"                    # -w ignoruje białe znaki
        else                                  # -q zwraca wart. iff różne pliki
                echo "WA $i"
                echo "Wejscie:"
                cat in1
                echo "Brut:"
                cat out1
                echo "Rozw:"
                cat out2
                break
        fi
        ((i++))
done
```

Przypadki grafów do testowania

1. Ścieżka https://en.wikipedia.org/wiki/Path_graph
2. Gwiazda https://en.wikipedia.org/wiki/Star_(graph_theory)
3. Ścieżka z liśćmi https://en.wikipedia.org/wiki/Caterpillar_tree
4. Klika https://en.wikipedia.org/wiki/Complete_graph
5. Graf "choinka"
6. Cykle

Magiczne linijki

```
ios_base::sync_with_stdio(0);
cin.tie(0), cout.tie(0);
```

Czas i pamięć programu

```
time ./program             // podstawowe informacje (tylko czas)
/usr/bin/time -v ./program // więcej informacji (czas i pamięć)
```

Limit stosu

```
ulimit -s           // sprawdź jaki limit stosu
ulimit -s <rozmiar> // ustaw limit na dany <rozmiar>
ulimit -s unlimited // zdejmij limit stosu
```

Limit pamięci

```
ulimit -v
ulimit -v <rozmiar>
```

Limit obowiązuje dla całego terminala i nie da się go zwiększyć. Aby to zrobić należy włączyć nowy terminal.

gdb

TODO

valgrind

Stosuje się do wykrywania problemów z pamięcią w programie.
Wykonuje się to komendą:

```
valgrind --tool=memcheck --leak-check=yes ./program
```

Drugim zastosowaniem valgrind'a jest analiza zużycia pamięci. Służy do tego komenda:

```
valgrind --tool=massif ./program < test.in
```

Zostaje stworzony plik `massif*` (różna końcówka). Przykładowo `massif.out.12440`. Teraz po wpisaniu komendy:

```
ms_print massif.out.12440
```

Trzecie zastosowanie to analiza czasu. Po wpisaniu komendy:

```
valgrind --tool=cachegrind ./program < test.in
```

Znowu tworzy się plik `cachegrind.out.*`, który otwieramy komendą:

```
kcachegrind cachegrind.out.12480
```

Moja Templatka

```
#include <bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;
using ll = long long;
#define FR first
```

```
#define SD second
#define PB push_back
#define deb(...) logger(#__VA_ARGS__, __VA_ARGS__) // debugging /*
template <typename T> struct tag:reference_wrapper <T>{ using reference_wrapper
<T>::reference_wrapper; };
template <typename T1, typename T2> static inline tag <ostream> operator<<(tag <ostream> os,
pair<T1, T2> const& p){ return os.get()<<"{"<<p.first<<", "<<p.second<<"}", os;}
template <typename Other> static inline tag <ostream> operator<<(tag <ostream> os, Other const&
o){ os.get()<<o; return os; }
template <typename T> static inline tag <ostream> operator <<(tag <ostream> os, vector <T>
const& v){ os.get()<<"["; for (int i=0; i<v.size(); i++) if (i!=v.size()-1) os.get()<<v[i]<<",
"; else os.get()<<v[i]; return os.get()<<"]", os; }
template <typename T> static inline tag <ostream> operator <<(tag <ostream> os, set <T> const&
s){ vector <T> v; for (auto i: s) v.push_back(i); os.get()<<"["; for (int i=0; i<v.size(); i++)
if (i!=v.size()-1) os.get()<<v[i]<<", "; else os.get()<<v[i]; return os.get()<<"]", os; }
template <typename ...Args> void logger(string vars, Args&&... values) { cout<<"[ "<<vars<<" =
"; string delim=""; (..., (cout<<delim<<values, delim=", ")); cout <<" ]\n"; }
/**/
```

Linki do poczytania

https://oi.edu.pl/static/attachment/20180202/propdoc-1.3.pdf