XXV OI, zawody II stopnia – Usuwanie usterek z rozwiązań

Przetestowanie swojego rozwiązania przed wysłaniem jest bardzo ważne, ponieważ dzięki temu można wykryć błędy, których wpływ na ocenę będzie widoczny dopiero po odsłonięciu wyników. Przemyślane testowanie pomaga też zlokalizować błędy w programie.

Poniżej przedstawiamy różne możliwości usuwania usterek z rozwiązań dostępne podczas zawodów II stopnia, a także na większości komputerów z systemem Linux. Jeśli nie masz w domu do dyspozycji komputera z systemem Linux, możesz spróbować przetestować wybrane funkcje w trakcie dnia próbnego albo na Windowsowych wersjach opisanych programów (choć te drugie mogą działać trochę inaczej). Zaznaczamy tylko, że Jury w trakcie zawodów II stopnia nie będzie mogło udzielać żadnych wskazówek na temat wymienionych poniżej metod.

Jest to pierwsza wersja tego dokumentu. Jeśli masz jakieś sugestie co do jego zawartości, napisz do nas na `olimpiada@oi.edu.pl` lub skontaktuj się z nami przez dział „Pytania" podczas II etapu.

1 Kompilacja w C i C++

Jeśli chcemy przetestować swoje rozwiązanie, warto w poleceniu kompilacji umieścić dodatkowe flagi `-Wall -Wextra -pedantic`, dzięki czemu „poprosimy" kompilator, aby zwrócił nam uwagę na usterki w naszym kodzie. Większość zgłaszanych przez kompilator usterek może powodować błędne działanie programu, dlatego warto ich nie ignorować lub – jeśli tak – robić to w pełni świadomie.

Kolejną ważną flagą jest flaga `-g3` (rozszerzona wersja flagi `-g`), która dołączy do pliku binarnego specjalne symbole ułatwiające debuggowanie kodu (patrz kolejne sekcje). Więcej o flagach debuggingowych:
`file:///usr/local/share/doc/gcc-4.8/gcc/Debugging-Options.html` – link lokalny
`https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html`

A co z optymalizacją (flaga `-O2`)? Optymalizacje często utrudniają debuggowanie kodu, więc na potrzeby debuggowania rozwiązania najczęściej warto je wyłączyć. Warto jednak pamiętać, że włączenie optymalizacji może powodować częstsze ujawnianie się błędów w kodzie. Jest też bardzo przydatne przy profilowaniu (patrz sekcja o `valgrind`). Więcej o optymalizacjach:
`file:///usr/local/share/doc/gcc-4.8/gcc/Optimize-Options.html` – link lokalny
`https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

Flaga `-static` może powodować wykrywanie błędów w programach ich pozbawionych (zawsze i głównie przez program `valgrind`). Dlatego (w dużym uproszczeniu) można tę flagę pominąć na potrzeby testowania programu, o ile ona nie jest źródłem błędów wykonania programu.

Istnieją jeszcze dwie bardzo przydatne flagi, choć nie polecamy ich mieszać z opisanymi niżej sposobami analizy programów – powinny one być używane zupełnie niezależnie od nich. Pierwsza z nich to `-fsanitize=address`, która powoduje dodanie do programu kodu wykrywającego błędne obchodzenie się z adresami. Po skompilowaniu programu z użyciem tej flagi normalnie go uruchamiamy – jeśli problem zostanie wykryty, to dodany kod przerwie jego działanie i wypisze stosowną informację. Druga to `-fsanitize=undefined`, która działa analogicznie do poprzedniej, ale wykrywa miejsca, w których program może zachowywać się w nieprzewidywalny sposób. Więcej o takich flagach w `gcc`:
`file:///usr/local/share/doc/gcc-4.9/gcc/Debugging-Options.html` – link lokalny
`https://gcc.gnu.org/onlinedocs/gcc-4.9.0/gcc/Debugging-Options.html#Debugging-Options`
`https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`

2 Kompilacja w Pascalu

Kompilacja w Pascalu jest bardzo zbliżona do tej w innych dostępnych językach. Aby poznać podstawowe flagi i ich znaczenia warto rzucić okiem do dokumentacji. Podstawowe flagi:
`file:///usr/share/doc/fp-docs/2.6.4/user/user.html#sec177` – link lokalny
`https://www.freepascal.org/docs-html/user/userap1.html`

Za dodanie debuggingowych informacji odpowiada flaga `-g`. Aby móc zobaczyć więcej informacji związanych z tym, co dzieje się w kodzie, może być warto użyć `-gl`.

Podczas kompilacji domyślnie nie są wyświetlane różne komentarze kompilatora. By poprosić kompilator o wyświetlanie jakiegoś typu informacji, wystarczy napisać `-v`, a następnie wymienić, o jakiego typu informacje nam chodzi. Domyślnie wymienione jest tylko `e` – errors. Zachęcamy do włączenia ostrzeżeń, notatek, wskazówek i generalnych informacji poprzez `-vewnhi`, jednak możliwości jest znacznie więcej.

Najprawdopodobniej najbardziej pomocną flagą kompilacji okaże się flaga `-C`. Jest ona odpowiedzialna za dodanie do kodu różnych funkcjonalności przydatnych w wykrywaniu usterek. Zachęcamy do przetestowania `-Ciort`. Wymusi to sprawdzanie IO-checking, przepełnień buforów, wyjść za tablice.

3 valgrind

`valgrind` to potężne narzędzie służące m.in. do wykrywania błędów i profilowania programów. Poniżej skupimy się tylko na jego najprostszych zastosowaniach.

Jeśli skompilujemy program zgodnie z wyżej opisanymi wytycznymi (bez flagi `-static`), a następnie napiszemy w konsoli `valgrind ./program < test.in`, to dostaniemy czytelną informację o większości błędów związanych z obsługą pamięci. Warto tu jednak zaznaczyć, że jeśli stosujemy dość popularne „zabezpieczenia" polegające na stałym rozmiarze tablic `MAX_N` (albo `n + 10`) lub innych podobnych zabiegach, to błędy mogą zostać wykryte dopiero przy dużych testach. Zrezygnowanie, przynajmniej na czas testów, z takich zabezpieczeń, może istotnie wpłynąć zarówno na jakość testów, jak i programów. Nieplanowane odwoływanie się do komórek tablicy, nawet jak są zaalokowane, często jest symptomem znacznie poważniejszego błędu programistycznego, który objawi się tylko przy specyficznych warunkach.

Błędy zgłaszane przez program `valgrind` na pewno nie należą do tych, które można lekceważyć (poza świadomym ignorowaniem wycieków pamięci). Prawie zawsze kończą się one błędem wykonania programu na większych testach lub po prostu nieprzewidywalnym działaniem programu. Zazwyczaj to właśnie tego typu błędy są winne sytuacji, w której program działa różnie w różnych środowiskach uruchomienia (np. na różnych komputerach).

`valgrind` ma jeszcze dwie funkcjonalności, którym poświęcimy chwilę. Pierwsza z nich to analizowanie zużycia pamięci. Komenda `valgrind --tool=massif ./program < test.in` tworzy plik `massif*` z analizą zużycia pamięci Twojego programu. Aby wyświetlić ją w czytelnej formie, należy napisać `ms_print`, a następnie podać nazwę właśnie stworzonego pliku. Przykładowo:

```
$ ms_print massif.out.4313
```

Warto tu zaznaczyć, że ta analiza jest zazwyczaj bliska do tej, którą przeprowadzamy na sprawdzaczkach. Nie musi być jednak ona identyczna. Ze względu na sposób zliczania pamięci na sprawdzaczkach zazwyczaj `massif` wykrywa większe zużycie pamięci o kilka MB (standardowo do 8) niż sprawdzaczki. Może się jednak zdarzyć, że wykryje mniejsze zużycie niż system Olimpiady.

Do testów warto używać pesymistycznych dużych testów (choć niekoniecznie maksymalnych). Jeśli zbyt szybko program zakończy działanie, to analiza nie będzie dokładna. Więcej informacji z przykładami:
`file:///usr/share/doc/valgrind/html/ms-manual.html` – link lokalny
`http://valgrind.org/docs/manual/ms-manual.html`

Druga ze wspomnianych funkcjonalności narzędzia `valgrind` to profilowanie czasu. Pisząc `valgrind --tool=cachegrind ./program < test.in`, tworzymy plik ze statystykami działania programu (na danym teście). Można je wyświetlić, pisząc `kcachegrind`, a następnie podając nazwę właśnie stworzonego pliku (domyślnie `cachegrind.out.*`). Na przykład:

```
$ kcachegrind cachegrind.out.4505
```

Warto tu podkreślić, że analiza jest przeprowadzana na innym systemie niż sprawdzaczkowy (w szczególności nie ma `oitimetool`), więc analizy czasów mogą nie być dokładne. Niemniej, bardzo wiele statystyk (jak np. liczba wywołań funkcji) będzie dobrze odpowiadać temu, jak program będzie działał na sprawdzaczce. Więcej informacji z przykładami:
`file:///usr/share/doc/valgrind/html/cl-manual.html` – link lokalny
`http://valgrind.org/docs/manual/cl-manual.html`

3.1 Profilowanie stosu – massif

Aby użyć valgrinda do badania pamięci na stosie warto użyć dodatkowych flag informacji. Dokładniejsze wytłumaczenie znajduje się poniżej, a najdokładniejsze w dokumentacji, ale często zadziała następujące polecenie:

```
$ valgrind --tool=massif --stacks=yes --max-stackframe=1073741824 \
     --main-stacksize=1073741824 ./program < test.in
```

Zgodnie z tym, co zostało napisane w dokumentacji, domyślnie `valgrind` mierzy tylko pamięć alokowaną poprzez `malloc, calloc, realloc, memalign, new, new[]`. Nie mierzy zatem pamięci alokowanej przez niskopoziomowe funkcje takie jak `mmap, mremap` i `brk` oraz tej używanej na stosie.

Oczywiście, najprostszym obejściem jest przejście na używanie pamięci alokowanej dynamicznie na przykład za pomocą kontenerów z stdliba np. `std::vector<int> table_name(size)` zamiast `int table_name[size]`.

Jeśli jednak zmniejszenie ilości pamięci używanej na stosie do pomijalnej wielkości nie jest preferowanym rozwiązaniem, można włączyć uwzględnianie pamięci alokowanej na stosie. Służy do tego flaga massif `-stacks=yes`. Zwolni to jednak istotnie czas profilowania.

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

To jednak może nie wyświetlać całej użytej pamięci. Rozwiązaniem tej niedogodności jest manipulowanie wartością `-max-stackframe`. Najczęściej wystarczy tę wartość ustawić na odpowiednia dużą (w stosunku do pamięci alokowanej na stosie przez program).

```
$ valgrind --tool=massif --stacks=yes --max-stackframe=1073741824 \
     ./program < test.in
```

Więcej o profilowaniu stosu:
`file:///usr/share/doc/valgrind/html/manual-core.html` – link lokalny
`http://valgrind.org/docs/manual/manual-core.html`

Jeśli potrzebne jest wiele pamięci, to warto też zmienić rozmiar stosu.

```
$ valgrind --tool=massif --stacks=yes --max-stackframe=1073741824 \
     --main-stacksize=1073741824 ./program < test.in
```

Więcej możliwości opisanych jest w dokumentacji. Warto tu jedynie dodać, że `valgrind` przyjmuje zazwyczaj dużo założeń, jak choćby to, że na początku stos jest pusty.

4 gdb

Jeśli program kończy się błędem wykonania lub działa w nieprzewidywalny dla nas sposób oraz wszystkie metody wykrywania błędów opisane powyżej zawiodły, warto sięgnąć po ostateczną broń – debugger. Jest to narzędzie służące do wykonywania programów w sposób krokowy. Wiele środowisk programistycznych posiada wbudowany debugger, często będący nakładką na jakiś niezależny od środowiska; poniżej opiszemy podstawowe funkcjonalności debuggera `gdb`, który nie wymaga użycia żadnego konkretnego środowiska programistycznego.

`gdb` ma wiele funkcjonalności, ale my skupimy się na kilku najbardziej podstawowych jego możliwościach. Uruchamiamy go za pomocą polecenia `gdb ./program` (standardowo po skompilowaniu z opcją typu `-g`), a następne komendy wpisujemy już w debuggerze. Oto (niepełna) lista dostępnych komend:

- `run < test.in` – uruchamia program i przekazuje mu wskazany test na standardowe wejście (`r<test.in`).
- `break xx` – ustawia break point na linii *xx* (`b xx`).
- `break foo` – ustawia break point na funkcji *foo* (`b foo`).
- `step` – przechodzi do następnej linii programu, przy czym wchodzi do wnętrza wywołań funkcji (`s`).
- `next` – przechodzi do następnej linii programu, przy czym *nie* wchodzi do wnętrza wywołań funkcji (`n`).
- `continue` – wznawia działanie programu po przerwaniu (`c`).
- `print var` – wypisuje wartość zmiennej *var* (`p var`).
- `backtrace` – wypisuje stos wywołań funkcji (`bt`).
- `list xx` – wypisuje kawałek programu, bliski linii xx (`l xx`).

Po wpisaniu komendy naciskamy Enter. W nawiasach podaliśmy opcje skrócone komend. Przykładowy przebieg krokowego wykonania programu po wywołaniu `gdb ./program`:

```
b main  % program ma się zatrzymać po dojściu do funkcji main
r < test.in  % uruchomienie na teście; program zatrzymuje się na funkcji main
n
n
...
p x  % wypisuje wartość zmiennej x
s  % wchodzi do wnętrza funkcji, na której znajduje się wykonanie krokowe
n
n
...
```

Najciekawsza z powyższych funkcji to `backtrace` (`bt`). Jej najłatwiejsze, najpopularniejsze użycie to uruchomienie programu, poczekanie aż zakończy się błędem wykonania, i wywołanie jej, aby dowiedzieć się gdzie i dlaczego program zakończył się błędem (albo przynajmniej w jakich okolicznościach). Więcej można znaleźć w dokumentacji: o przerywaniu działania:
`file:///usr/share/doc/gdb-doc/html/gdb/Set-Breaks.html` – link lokalny
`https://sourceware.org/gdb/onlinedocs/gdb/Set-Breaks.html#Set-Breaks`
o krokowym wykonywaniu programu:
`file:///usr/share/doc/gdb-doc/html/gdb/Continuing-and-Stepping.html` – link lokalny
`https://sourceware.org/gdb/onlinedocs/gdb/Continuing-and-Stepping.html`
o backtrace:

`file:///usr/share/doc/gdb-doc/html/gdb/Backtrace.html` – link lokalny
`https://sourceware.org/gdb/onlinedocs/gdb/Backtrace.html`

Bardzo pomocne mogą się okazać `watchpoints`, służące do wykrywania, kiedy wartość zmiennej lub wartość pod konkretnym adresem ulega zmianie lub jest odczytywana. Uruchamiamy je za pomocą komendy `watch var` (w skrócie `w var`) i kiedy wartość zmiennej *var* się zmienia, `gdb` nas o tym informuje. Więcej na ten temat można przeczytać w dokumentacji:
`file:///usr/share/doc/gdb-doc/html/gdb/Set-Watchpoints.html` – link lokalny
`https://sourceware.org/gdb/onlinedocs/gdb/Set-Watchpoints.html`