11 Debugowanie, gdb

Debugowanie to proces znajdowania i usuwania błędów (*bugów*) w działaniu programu. Jak znaleźć taki błąd? Podstawowe „ręczne" sposoby to:

* patrzenie w kod źródłowy,
* wzbogacenie kodu o „printf-y" – wypisywanie różnych kontrolnych wartości.

Możemy także sięgnąć po narzędzia wspierające proces debugowania:

* skorzystać z *debugera* – programu specjalnie przeznaczonego do tego celu,
* użyć opcji kompilatora – wygenerować kod wyposażony w sprawdzanie poprawności.

Debugger to specjalne środowisko w którym uruchamiany jest program, pozwalające śledzić jego działanie linijka po linijce. Umożliwia zatrzymanie programu w każdej chwili i sprawdzenie jego stanu (zmienne, wyrażenia, stos wywołań funkcji). Przykładowo możemy sprawdzić, że w 25 linijce zmienna `X` ma wartość `10`, a w linijce 36 dla bloku `if-then-else` zostanie wykonany `else`.

Co daje debugger, czego nie dadzą nam `printf`-y?

- wstawianie i usuwanie wypisywania zabiera czas, wymaga edycji i rekompilacji,
- po zdebugowaniu musimy usunąć (lub wyłączyć) dodatkowy kod,
- wykonując kod dowiemy się tylko tyle ile wypiszemy,
- w debuggerze, nie musimy modyfikować kodu
- w chwili wykrycia błędu program jest zatrzymany, dowiadujemy się gdzie wystąpił błąd oraz możemy zacząć sprawdzać jego stan w momencie zatrzymania,
- jesteśmy w stanie interaktywnie śledzić źródło błędu, wykryć niespodziewane usterki,
- możemy zmodyfikować wartości w trakcie wykonania!

Popularnym debuggerem tekstowym jest `GNU Debugger` – `gdb`, na którym będziemy bazować. Istnieje do niego wiele nakładek graficznych, oprócz tego większość zintegrowanych środowisk programistycznych posiada wbudowany debugger – niezależnie od wersji, główne możliwości jakie dają tego typu programy są do siebie bardzo zbliżone.

Debugowanie kompilatorem

Zanim omówimy debugowanie, krótkie spojrzenie jakie możliwości dostarcza kompilator.

-fsanitize=address/leak/undefined/... , sprawdzanie poprawności dostępu do pamięci, wykrywanie wycieków, ostrzeganie o operacjach o niezdefiniowanym przez standard wyniku

-fcheck-pointer-bounds , sprawdza czy wskaźniki wskazują na rozsądne rzeczy

-fstack-protector , sprawdza czy funkcje nie nadpisują kawałków stosu

Opcje te produkują znacznie wolniejsze i większe wersje programów, które dokonują dodatkowych sprawdzeń w trakcie wykonania. Zaletą tego podejścia jest automatyzacja – bez zastanawiania się nad debugowaniem, na testowych programach, można wykryć pospolite błędy.

Debugowanie z `gdb`

Przygotowanie programu do debugowania – kompilacja `-g`

Przed uruchomieniem debuggera dla danego programu, należy mu najpierw dostarczyć odpowiednio skompilowany plik. W przypadku kompilatora `GCC` używamy opcji `-g`:

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

Parametr `-g` powoduje, że w pliku obiektowym powstaje specjalna sekcja gdzie zapisane są odwołania do kodu źródłowego. Program wie gdzie pojawiają się nowe linie kodu źródłowego, jak nazywają się zmienne globalne i lokalne, itd.

W przypadku kompilacji z opcją `-g` nie powinno się stosować opcji optymalizacyjnych typu `-O2`, gdyż mogą one powodować nieprawidłowości przy nawigacji w czasie sesji debuggera. Parametr `-Og` udostępnia tylko te optymalizacje, które nie zakłócają debugowania.

Uruchomienie programu

Aby uruchomić debugger, wystarczy wpisać `gdb` – przechodzimy wtedy w tryb interaktywny, w którym `gdb` oczekuje poleceń użytkownika. W poniższych przykładach będziemy stosować prompt `(gdb)` dla oznaczenia komend wewnątrz powłoki gdb.

Typowe uruchomienie debugowania programu:

```
$ gdb <program>
(gdb) run 2 5               # uruchomienie programu z argumentami
```

bądź:

```
$ gdb
(gdb) file <program>        # wgranie programu
(gdb) run 2 5 < in > out    # argumenty oraz przekierowanie input/output
```

Program zostaje uruchomiony jako proces podrzędny i jest wykonywany do momentu gdy:

* zakończy działanie,
* zostanie wywołany sygnał przerwania (e.g. `SIGINT`, `SIGTERM` etc.),
* zostanie rzucony wyjątek który nie został przechwycony wewnątrz programu,
* zostanie wykonana niepoprawna instrukcja.

Podstawowe komendy:

`help`	pomoc ogólna, wypisuje klasy komend
`help <komenda>`	pomoc na temat danej komendy lub klasy komend
`quit`	wyjście z programu

Warto wiedzieć, że `gdb` udostępnia skróty nazw komend (pierwsza litera bądź prefiks). Przykładowo: `q` → `quit`, `h` → `help`. W razie niejednoznaczności zostaniemy poinformowani o pasujących nazwach.
W powłoce `(gdb)` działają takie rzeczy jak: `cd`, `pwd`. Komenda `shell` włączy basha, po wyjściu z którego jesteśmy dalej w `gdb`.

Analiza post-mortem

Załóżmy, że uruchomiliśmy program (`run`) i nastąpiła niepoprawna operacja, skutkująca przerwaniem programu. Zostaniemy o tym poinformowani i powrócimy do powłoki gdb, jednakże w przeciwieństwie do zwykłego uruchomienia, program jest nadal w pamięci – możemy podejrzeć w jakim był stanie w chwili gdy doszło do błędu.

```
Program received signal SIGFPE, Arithmetic exception.
0x00000000004005df in nwd (a=2, a@entry=12, b=0) at nwd.c:10
10                  t = a % b;
```

Wypisany zostanie komunikat (przykład powyżej), z którego możemy dowiedzieć się o:

* rodzaju zdarzenia,
* miejscu zdarzenia (adres instrukcji i numer linijki, funkcja i wartości jej argumentów),
* kodzie źródłowym w miejscu wystąpienia błędu.

Już ta informacja jest bardzo przydatna. Możemy jednak dowiedzieć się dużo więcej.

podgląd zmiennych

`info args`	wyświetla aktualne wartości argumentów fukcji
`info locals`	wyświetla aktualne wartości zmiennych lokalnych

Uwaga! Jeśli funkcja zmienia wartości swoich argumentów, wypisywane są ich aktualne wartości a nie te z którymi funkcja była pierwotnie wywołana. Debugger czasami próbuje odtworzyć pierwotną wartość argumentu co wypisywane jest jako `<nazwa>@entry`.

stos wywołań funkcji

`bt, backtrace`	podgląd aktualnego stosu wywołań funkcji; każda z pozycji określa *ramkę* (ang. *frame*), z lokalnymi zmiennymi danego wywołania

```
#0  0x0000000000004005d6 in nwd (a=2, b=0) at nwd.c:10
#1  0x0000000000400654 in main (argc=3, argv=0x7fffffffe498) at nwd.c:24
```

Po ramkach możemy poruszać się, sprawdzając wartości zmiennych w kolejnych funkcjach.

`f, frame`	wyświetl bieżącą ramkę
`frame <n>`	przejdź do ramki numer `<n>` (to ten numer po znaku `#`)
`down/up <n>`	przeskocz w dół / w górę o `<n>` ramek

Komendy typu `info args/locals` zależą od aktualnie wybranej ramki.

podgląd kodu źródłowego

`list`	wyświetla 10 linijek w okół aktualnego miejsca
`list <n>`	wyświetla 10 linijek wokół linii `<n>`.
`list <b>,<e>`	wyświetla linijki on `<b>` do `<e>`.

wypisywanie wartości/wyrażeń

`print <X>`	wypisz wartość zmiennej <X>
`print <wyrazenie>`	wykonaj `<wyrazenie>` i wypisz jego wynik (akceptuje składnie zbliżoną do C++)

`ptype <wyrazenie>` zwraca typ zmiennej/obliczonego wyrażenia

Przykłady:

```
(gdb) frame 1
#1  0x0000000000400654 in main (argc=3, argv=0x7fffffffe498) at nwd.c:24
24          int result = nwd(a,b);
(gdb) info local
a = 12
b = 14
result = 0
```

```
(gdb) print argv
$3 = (char **) 0x7fffffffe498
(gdb) print argv[1]
$4 = 0x7fffffffe72f "12"
(gdb) print argv[2]
$5 = 0x7fffffffe732 "14"
(gdb) print atoi(argv[1])+atoi(argv[2])
$6 = 26
```

Każde wywołanie zwraca wynik w postaci `$<n> = ...` – numeru `<n>` możemy użyć jako pomocniczej zmiennej do budowania kolejnych wyrażeń:

```
(gdb) print atoi($4)+atoi($5)
$7 = 26
(gdb) ptype atoi($4)+atoi($5)
type = int
```

Breakpointy

Jak dotąd zakładaliśmy, że w kodzie znajduje się "katastrofalny" błąd który prowadzi do zatrzymania działania całego programu. Częściej jednak mamy do czynienia z nieprawidłowym działaniem, które nie przerywa działania programu. Aby prześledzić dokładnie jak program zachowuje się w jakimś miejscu, należy umieścić tak zwanego *breakpointa*.

`break <n>`	wstawia breakpoint w aktualnym pliku źródłowym w linii `<n>`.
`break <nazwa>`	wstawia breakpoint w pierwszej instrukcji funkcji `<nazwa>`.
`info breakpoints`	wyświetla wszystkie stworzone breakpointy

Po uruchomieniu, gdy program osiągnie linijkę ze wstawionym *breakpointem*, jego działanie zostanie zatrzymane (przed wykonaniem tej linijki). Możemy przeglądać stan programu, tak jak w przypadku post-mortem. Dodatkowo możemy sterować dalszym działaniem.

`continue`	wznawia działanie programu (aż do zakończenia lub kolejnego breakpointa)
`step <n>`	wykonuje bieżącą linię i przechodzi do kolejnej (może być w innej funkcji); opcjonalny argument `<n>` oznacza powtórzenie tego `<n>` razy
`next <n>`	wykonuje program aż zostanie osiągnięta kolejna linia w bieżącej funkcji; wywołania funkcji (wewnątrz bieżącej linijki) wykonywane są w całości
`until`	wykonuje program aż do linii o wyższym numerze w bieżącej funkcji
`finish`	wykonuje program aż do wyjścia z aktualnej funkcji

Jeśli chcemy wykonać tą samą komendę co poprzednio, wystarczy sam `<Enter>`.

zarządzanie breakpointami

`tbreak ...`	wstawia jednorazowy breakpoint
`ignore <b> <n>`	ignoruje `<n>` razy breakpoint `<b>`, zatrzymanie dopiero za `n+1` razem
`delete`	usuwa wszystkie breakpointy
`delete <b>`	usuwa breakpoint o numerze `<b>`
`clear ...`	usuwa breakpointy z zadanego miejsca
`disable <b>`	deaktywuje breakpoint o numerze `<b>`
`enable <b>`	aktywuje breakpoint o numerze `<b>`

zmiana wartości

W momencie zatrzymania programu, można zmieniać wartość zmiennych:
`set var <zmienna>=<wartosc>`

Przykłady:

```
(gdb) break nwd.c:7
Breakpoint 1 at 0x4005c2: file nwd.c, line 7.
(gdb) run 12 14
Starting program: nwd 12 14
Breakpoint 1, nwd (a=12, b=14) at nwd.c:7
7               if (a<b)
```

```
(gdb) continue
Continuing.
Breakpoint 1, nwd (a=14, b=12) at nwd.c:7
7               if (a<b)
(gdb)
Continuing.
...
(gdb)
Continuing.
...
(gdb)
Continuing.
Program received signal SIGFPE, Arithmetic exception.
0x00000000004005d6 in nwd (a=2, b=0) at nwd.c:10
10                  t = a % b;
```

Watchpointy

Podobnie jak breakpointy, *watchpointy* służą przerywaniu działania programu. Podczas gdy breakpointy umieszcza się w kodzie, *watchpointy* działają na danych. Są to punkty obserwacji zmiany wartości wyrażenia – gdy nastąpi takowa, debugger zatrzyma program.

`watch <w>`	obserwuje wyrażenie `<w>` i przerywa program gdy nastąpi zmiana
`watch -l <w>`	obserwuje adres pamięci zawierający wynik `<w>`
`rwatch <w>`	obserwuje wyrażenie `<w>`, przerywa gdy odczytywana jest wartość

Watchpointy dzielą numeracje z breakpointami i wszystkie polecenia operujące na tych numerach (np `delete`, `ignore` czy `condition`).

Podgląd kodu źródłowego

Jak pisaliśmy powyżej, aby zobaczyć kod źródłowy można używać komendy `list`. Wygodniej jednak pracować z bardziej interaktywnym podglądem – tak zwanym *Text User Interface* (TUI). Domyślnie będzie zawierał wewnątrz terminala "okno" z kodem, a poniżej linię poleceń. TUI możemy uruchomić przy starcie gdb:
`gdb -tui program`
lub poprzez kombinację klawiszy `Ctrl-x a`, `Ctrl-x Ctrl-a` czy `Ctrl-x Ctrl-A` (każdą z tych kombinacji można też wyłączyć tekstowy interfejs).

Valgrind - debugger użycia pamięci

GDB nie jest najlepszym narzędziem do analizowania użycia pamięci. Pozwala wykryć takie błędy jak:

* Dostęp do niewłaściwej pamięci (pamięci niezaalokowanej, zwolnionej, poza dozwolonym zakresem)
* Użycie niezainicjowanej wartości
* Bezpowrotnie utracenie zarezerwowanej pamięci (tzw. *memory leak*)
* Wielokrotne zwalnianie pamięci

Najprostrzej uruchomić debugowanie valgrindem za pomocą:
`valgrind <program> <argumenty-do-programu>`

Jeśli mamy jakiś memory leak, dostaniemy wtedy wynik podobny do następującego:
```
==8980== LEAK SUMMARY:
==8980==    definitely lost: 52 bytes in 2 blocks
==8980==    indirectly lost: 0 bytes in 0 blocks
==8980==      possibly lost: 0 bytes in 0 blocks
==8980==    still reachable: 72,704 bytes in 1 blocks
==8980==         suppressed: 0 bytes in 0 blocks
==8980== Rerun with --leak-check=full to see details of leaked memory
```

Warto tutaj zaznaczyć, że określenie czy jakaś pamięc nie jest osiągalna jest trudne. Uwzględniane są nie tylko zmienne lokalne i globalne, ale też odnośniki pośrednie i tablice. Dlatego poza pozycjami `still reachable` i `definitely lost` pojawiają się pozycje dla których valgrind nie ma pełnej pewności.

Jeśli gdzieś w programie wykonaliśmy niewłaściwy zapis do pamięci pojawi się też komunikat:
```
==12313== Invalid write of size 4
==12313==    at 0x400662: clear(int*, int) (main.cpp:9)
==12313==    by 0x400697: main (main.cpp:14)
==12313==  Address 0x5ab5c94 is 0 bytes after a block of size 20 alloc'd
```

```
==12313==    at 0x4C2E80F: operator new[](unsigned long) (in
   /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==12313==    by 0x400629: allocate(int) (main.cpp:3)
==12313==    by 0x400682: main (main.cpp:13)
```

Podaje on w której linii nastąpił zapis, oraz gdzie w okolicy znajduje się pamięć która rzeczywiście jest poprawnie zarezerwowana i gdzie została ona zarezerwowana. Pozwala to na znajdowanie błędów np. z przekroczeniem zakresu tablicy.

Valgrind i gdb

Valgrind nie posiada interaktywnej powłoki jak `gdb`. Aby debugować program z uwzględnieniem błędów pamięci trzeba połączyć działanie tych dwóch programów. Gdybyśmy jednak uruchomili `gdb valgrind program` to będziemy debugować `valgrind` a nie nasz program. Nie o to nam chodzi. Podobnie `valgrind gdb ...` będzie analizować `gdb` a nie nasz program.

Na szczęście `gdb` ma możliwość debugowania procesów już istniejących poprzez dopinanie się do nich. W naszym przypadku dokonujemy tego w trzech krokach:

* `valgrind --vgdb=yes --vgdb-error=0 <program> <argumenty>`
  Uruchamia program w trybie który może zostać "przechwycony" przez gdb. Program `vgdb` to mały program/serwer służący do komunikacji pomiędzy `valgrind`-em a `gdb`. Program ten przyjmuje komendy od `gdb`, wykonuje ich działanie i zwraca wynik. Z kolei parametr `--vgdb-error` określa ile błędów dopuszczamy zanim serwer debugujący będzie aktywny. Przekazanie parametru 0 powoduje, że debugger uruchamia się od razu i program sam się nie wykona.
* W drugim oknie konsoli uruchamiamy `gdb` z tym samym programem jako argument:
  `$ gdb <program>`
  Argumentów nie trzeba podawać. Potrzebny jest nam tylko obraz pliku obiektowego wraz z symbolami debugującymi.
* Następnie musimy połączyć `gdb` z aktywnym serwerem `vgdb`:
  `(gdb) target remote | vgdb` Jeśli aktywnych jest kilka różnych serwerów `vgdb` to trzeba podać numer PID procesu który nas interesuje.

Tak przygotowany program możemy już uruchomić. Nie wywołujemy `run`, ale `continue`. Jednakże, działanie teraz zostanie przerwane z chwilą wykrycia błędu przez valgrind:

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x0000000000400662 in clear (array=0x5ab5c80, size=6) at main.cpp:9
9               array[i]=0;
(gdb)
```

Teraz możemy używać `gdb` w normalny sposób i dokładnie przeanalizować moment w którym doszło do niepoprawnej operacji.