debuggery

Debugger to zewnętrzny program do analizy kodu, którego celem jest znajdywanie błędów w działaniu innych programów (czyli *bugów*). Wewnątrz debuggera możemy wywołać własny program i śledzić jego działanie linijka po linijce, posiadając możliwość zatrzymania go w każdej chwili i sprawdzenia jego stanu (zmienne, wyrażenia, stos wywołań funkcji). Debuggery nadają się do szukania błędów wykonania i niewłaściwych zachowań w środku programów, choć już nie do wycieków pamięci.

Tak naprawdę debugger służy do **upewniania się**, że dany program działa dobrze, tzn. w każdym jego miejscu spełnione są nasze założenia względem niego. Na przykład możemy wierzyć, ze w 25 linijce zmienna `X` ma wartość `10`, a w linijce 36 dla bloku `if-then-else` zostanie wykonany `else`. Debugger może zweryfikować nasze przekonania.

Dlaczego używanie debuggera jest lepsze od wstawiania w kodzie `printf`ów, `cout`ów czy `println`ów?

(i) Wstawianie i usuwanie wypisywania związane jest z dekoncentracją, czasochłonnymi edycjami i rekompilacjami.
(ii) Debugger dostarcza więcej informacji (np. w przypadku błędu wykonania można otrzymać informacje o jego miejscu i aktualnych wartościach zmiennych).
(iii) Debugger pozwala na więcej niż zwykły `printf` (ręczne wywołanie funkcji, zmiana wartości zmiennych, itp.).

Debuggery możemy podzielić na działające w trybie tekstowym i graficznym. Popularnym debuggerem tekstowym jest `GNU Debugger` (w skrócie `gdb`), choć istnieje też do niego wiele nakładek graficznych. Większość zintegrowanych środowisk programistycznych (ang. *integrated development environment – IDE*) posiada wbudowany debugger. Mimo, że omówimy sposób działania debuggerów jedynie na przykładzie `gdb`, to przedstawione idee powinny jednak w zupełności wystarczyć do korzystania z dowolnego innego podobnego typu programu.

GNU Debugger

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
```

Uwaga! W przypadku kompilacji z opcją `-g` nie powinno się stosować dodatkowo żadnych opcji optymalizacyjnych typu `-O2`, bo powodują one nieprawidłowości przy nawigacji w czasie sesji debuggera.

Plik wykonywalny podajemy programowi `gdb` wpisując po prostu: `gdb program`. Nic nie stoi jednak na przeszkodzie, aby wywołać `gdb` bez parametru. Wtedy trzeba jednak koniecznie skorzystać z polecenia `file` prezentowanego poniżej. Jeżeli użyjemy opcji `-q`, to przy starcie `gdb` nie wyświetli informacji o swojej licencji.

Podstawowe komendy:

`help`	pomocy ogólna;
`help` *komenda*	pomoc dla danej komendy;
`run`	uruchomienie programu w celu debugowania;
`run` *arg1 arg2 ...*	uruchomienie programu z podanymi argumentami;
`run <` *plik*	uruchomienie programu dla danych z pliku;
`file` *nazwa-programu*	załadowanie (przeładowanie) pliku wykonywalnego do `gdb`;
`quit`	wyjście z `gdb`.

Warto wiedzieć, że `gdb` udostępnia skróty nazw komend. Dzięki temu wystarczy wpisać np. samo `q` zamiast `quit`. W razie niejednoznaczności zostaniemy poinformowani o pasujących nazwach.

Co ciekawe, raz podane argumenty i przekierowania przy poleceniu `run` będą zapamiętane i przy ponownym użyciu tej komendy bez parametrów zostaną one automatycznie użyte. Co jednak zrobić, jeżeli w pewnym momencie chcemy użyć komendy `run` bez argumentów? Powinniśmy użyć komendy `set args`, która wyczyści nam listę parametrów. W ogólności polecenie to służy do ustawiania argumentów (przekierowań) wywołania polecenia `run` bez jego wykonywania.

Breakpointy to miejsca, w których chcemy, aby debugger się zatrzymał w celu sprawdzenia stanu pewnych parametrów (jak wartości zmiennych i wyrażeń, zawartość stosu). Po uruchomieniu debuggera wykonywany program zatrzyma się na pierwszym napotkanym breakpoincie (albo wykona się i zakończy swoje działanie w przypadku ich braku).

Poza breakpointami istnieją też *watchpointy* – punkty obserwacji zmiany wartości wyrażenia. Ilekroć nastąpi taka zmiana, debugger zatrzyma się. Obsługa watchpointów jest podobna jak w przypadku breakpointów, więc wiele z poniższych komend będzie działało również dla nich.

Komendy związane z breakpointami i watchpointami:

`break` *numer-linii*	wstawienie breakpointa (linia);
`break` *program.c:nr-linii*	`--||--` (program z wielu plików źródłowych);
`break` *nazwa-funkcji*	`--||--` (funkcja w `C/C++`);
`break` *klasa::nazwa-metody(typy-arg)*	`--||--` (metoda w `C++`);
`tbreak` *numer-linii*	wstawienie jednorazowego breakpointa;
`watch` *wyrażenie*	wstawienie watchpointa;
`info breakpoints`	lista ponumerowanych breakpointów;
`info watchpoints`	lista ponumerowanych watchpointów;
`condition` *nr-breakpointa warunek*	uczynienie breakpointa warunkowym;
`ignore` *nr-breakpointa liczba-razy*	opuszczenie breakpointa podaną liczbę razy;
`delete`	usunięcie wszystkich breakpointów;
`delete` *nr-breakpointa*	usunięcie podanego breakpointa;
`clear` *nazwa-funkcji*	usunięcie breakpointów z początku funkcji;
`clear` *numer-linii*	usunięcie breakpointów z podanej linii;

`disable` *nr-breakpointa*	dezaktywacja breakpointa;
`enable` *nr-breakpointa*	aktywacja dezaktywowanego breakpointa.

Komendy nawigacyjne w trakcie działania komendy `run`:

`Ctrl-C`	przerwanie wykonywania programu;
`kill`	zakończenie wykonywania programu (w celu restartu lepiej użyć po prostu `run`);
`continue`	wznowienie działania programu do najbliższego zatrzymania;
`step`	wykonanie jednej linii i skok do wnętrza funkcji;
`next`	wykonanie jednej linii, nawet jeśli to funkcja;
`until`	wykonywanie linii aż do osiągnięcia linii o wyższym numerze;
`finish`	wykonywanie linii aż do wyjścia z danej funkcji.

Jeżeli nie wpiszemy komendy (zostawimy pustą linię) i naciśniemy `Enter`, to wykona się ostatnio używane polecenie. Jest to bardzo przydatne szczególnie w przypadku komend `step` i `next`. Dobry nawykiem przy debugowaniu jest używanie komendy `next` zamiast `step`, gdyż `step` potrafi wejść do wnętrza funkcji, do których na pewno nie będziemy chcieli zaglądać (jak `scanf` czy `printf`). Jeśli natomiast interesuje nas przetestowanie wnętrza danej funkcji, to najlepiej wstawić w środku niej breakpointa.

Nim omówimy komendy związane ze stosem wywołań funkcji w programie (czyli ze stosem *ramek*) kilka słów o samym stosie. Gdy tworzony jest proces dla naszego programu, przydzielana mu jest pamięć. Proces nie jest świadom, gdzie ta pamięć leży i czy np. jest w spójnym kawałku. Wszystko co proces wie, to jak duża jest jego pamięć i że powinien ją adresować startując od adresu `0x00`. Procesy nic nie wiedzą o pamięci innych procesów, a nawet gdyby nieświadomie próbowały zacząć pisać po nie swoim bloku pamięci – od razu zostaną zablokowane. Proces dzieli swoją pamięć na kilka części, jadną z nich jest tak zwany stos ramek. Każda funkcja wykonywanego programu rozpoczyna działanie odkładając na stos swoja ramkę, czyli rezerwując pamięć na swoje zmienne lokalne oraz informacje gdzie ma przekazać sterowanie po zakończeniu. W praktyce pierwsza ramka na stosie naszych programów zawsze będzie pochodzić od funkcji `main()`.

Debugując powinniśmy pamiętać, że program wykonując instrukcje funkcji będącej na wierzchu stosu ramek nie ma dostępu do zmiennych lokalnych funkcji leżących głebiej na stosie. Jednak zmienne te mają zaalokowaną pamięć i ich wartość jest trzymana do czasu powrotu wykonania do ich funkcji. Używając odpowiednich komend debugera możemy *chodzić* po ramkach (komenda `frame`) i przeglądać wartości ich zmiennych.

`backtrace`	numerowany stos ramek;
`where`	to samo co `backtrace`;
`call` *nazwa-funkcji(argumenty)*	ręczne wywołanie funkcji, nowa ramka na stosie;
`frame` *numer-ramki*	skok do podanej ramki;
`info frame`	informacja o aktualnej ramce;
`info locals`	informacja o zmiennych lokalnych aktualnej ramki;
`info args`	informacja o argumentach wywołania aktualnej ramki.

Inne wartościowe komendy:

`list`	po zatrzymaniu (np. na breakpoincie) wyświetlanie 10 linijek wokół miejsca zdarzenia;
`list` *numer-linijki*	wyświetlanie 10 linijek wokół podanej;
`list` *nr-linijki-1,nr-linijki-2*	wyświetlanie zakresu linijek;
`print` *wyrażenie*	wypisywanie wartości wyrażenia, przykłady wyrażeń: `a`, `3*a-1`, `a==2`, `a-b`, `*wsk`, `tab`, `tab[3]`, `tab[3]@9`;
`ptype` *zmienna*	wypisywanie definicji typu zmiennej;
`set var` *zmienna = wartość*	zmiana wartości zmiennej w trakcie sesji debuggera.

W czasie sesji z programem `gdb` najlepiej mieć otwarte również okno z kodem źródłowym. Pewną alternatywą dla powyższego rozwiązania jest `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).

Przykład 1. Poniżej trochę przykładów wywołań komendy `print` (skrót: `p`). Pojawiają się też wywołania `ptype` (`pt`) oraz `set print pretty` (do eleganckiego wypisywania struktur). `$` odnosi się do ostatnio wypisywanej wartości, natomiast wyrażenie w postaci `$n` nawiązuje do `n`-tej wypisywanej wartości.

```
(gdb) pt argc
type = int
(gdb) pt myfloat
type = float
(gdb) pt argv
type = char **
(gdb) pt mystring
type = unsigned char *
(gdb) p mychar
$26 = 65 'A'
(gdb) p mydouble
$27 = 1.0466666666666666
(gdb) p $
$28 = 1.0466666666666666
(gdb) p $27
$29 = 1.0466666666666666
(gdb) p $26
$30 = 65 'A'

(gdb) pt myIntArray
type = int [10]
```

```
(gdb) pt myIntArray[3]
type = int
(gdb) p myIntArray[3]
$48 = 3
(gdb) p myIntArray[3]@5
$49 = {3, 4, 5, 6, 7}

(gdb) p myStruct
$2 = {name = 0x40014978 "Miles Davis", EyeColour = 1}

(gdb) set print pretty
(gdb) p myStruct
$4 = {
  name = 0x40014978 "Miles Davis",
  EyeColour = 1
}

(gdb) print myStruct.name
$6 = 0x40014978 "Miles Davis"

(gdb) print myStruct->name
$15 = 0x40014978 "Miles Davis"
```

Przykład 2. Jak ustawić watchpointa śledzącego zmianę wartości tablicy statycznej?

```
int a[10]={0,1,2,3,4,5,6,7,8,9};
a[2]=10;

(gdb) watch a

Old value = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
New value = {0, 1, 10, 3, 4, 5, 6, 7, 8, 9}
```

Przykład 3. Jak ustawić watchpointa śledzącego wartości podtablicy tablicy statycznej?

```
(gdb) watch a[3]@4
```

Przykład 4. Jak śledzić wartości tablicy dynamicznej?

```
int *c=new int[20];
c[10]=10;

(gdb) watch *c@20;

Old value = {0 <repeats 20 times>}
New value = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Przykład 5. Jak ustawić watchpointa na zmiennej typu `vector` z biblioteki `STL`?

```
vector<int> d;
for(int i=0; i< 10; i++) d.push_back(i);
d[5]=13;

(gdb) *&d[0]@10
```

Przykład 6. Mamy dany program `bottles.c`:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i;
    for(i=100; i>=0; --i) {
        printf("There are %d bottles standing on the wall.\n", i);
        system("sleep 5");
    }
    return 0;
}
```

Po skompilowaniu z opcją `-g` uruchamiamy go w tle:

```
./bottles &

[1] 36484
There are 100 bottles standing on the wall.
There are 99 bottles standing on the wall.
There are 98 bottles standing on the wall.
```

Jak dostać się do wnętrza procesu o `PID`zie 36484?

```
gdb bottles 36484
(gdb) where
#0  0x00007f9f6026b43e in __libc_waitpid (pid=<optimized out>,stat_loc=
0x7fffed5474c0, options=0) at ../sysdeps/unix/sysv/linux/waitpid.c:32
#1  0x00007f9f601f129e in do_system (line=0x4006ac "sleep 5") at
../sysdeps/posix/system.c:149
#2  0x0000000000400576 in main () at bottles.c:8
```

Alternatywnie możemy uruchomić `gdb` bez argumentów i użyć polecenia `attach 36484`. W celu zakończenia pracy z danym procesem wpisujemy komendę `detach`.

Zadanie 1. Napisz program który inicjalizuje zmienną typu `bool x` na `true`, a później wchodzi do pętli `while(x) {...}`, z której nigdy sam już nie wyjdzie. Uruchom ten program w tle. Następnie przy pomocy `gdb` zmień wartość zmiennej `x` w uruchomionym procesie.