G

Użycie debugera GNU

Zagadnienia

W tym rozdziale poruszymy następujące zagadnienia:

- użycie polecenia `run` do uruchamiania programu w debugerze
- użycie polecenia `break` do zdefiniowania punktu przerwania
- użycie polecenia `continue` do kontynuowania działania programu
- użycie polecenia `print` do obliczania wartości wyrażenia
- użycie polecenia `set` do zmiany wartości zmiennych podczas działania programu
- użycie poleceń `step`, `finish` i `next` do sterowania sposobem działania programu
- użycie polecenia `watch` do sprawdzenia, w jaki sposób element danych jest modyfikowany podczas wykonywania programu
- użycie polecenia `delete` do usunięcia punktu przerwania lub punktu obserwacji

G.1. Wprowadzenie

W rozdziale 2. poznałeś dwa rodzaje błędów — kompilacji i logiczne — a także dowiedziałeś się, jak można wyeliminować z kodu błędy kompilacji. **Błędy** logiczne nie przeszkadzają w zakończeniu kompilacji sukcesem, ale mogą prowadzić do nieprawidłowego działania programu po jego uruchomieniu. Kompilator GNU zawiera oprogramowanie o nazwie *debuger*, które pozwala monitorować sposób wykonywania programu, co umożliwia wychwytywanie w nim błędów logicznych i ich usuwanie.

Debuger stanie się jednym z Twoich najważniejszych narzędzi programistycznych. Wiele środowisk IDE oferuje własne debugery podobne do dostarczanego wraz z kompilatorem GNU lub oferuje interfejs graficzny dla debugera GNU. W tym dodatku przedstawimy kluczowe możliwości debugera GNU. W dodatku F natomiast omówiliśmy funkcje i możliwości, jakie oferuje debuger dostarczany z Visual Studio.

G.2. Punkty przerwania oraz polecenia run, stop, continue i print

Analizę debugera rozpoczniemy od przyjrzenia się **punktom przerwania**, czyli znacznikom, które można zdefiniować w dowolnym wykonywalnym wierszu kodu. Gdy podczas działania programu będzie miał zostać wykonany wiersz zawierający punkt przerwania, działanie programu zostanie wstrzymane, co pozwoli programiście sprawdzić wartości zmiennych, aby ustalić, czy program zawiera błędy logiczne. Na przykład można sprawdzić wartość zmiennej przechowującej wynik obliczeń i tym samym ustalić, czy zostały one przeprowadzone prawidłowo. Warto w tym miejscu dodać, że próba ustawienia punktu przerwania w niewykonywalnym wierszu kodu (np. komentarzu) spowoduje zdefiniowanie tego punktu przerwania w następnym wykonywalnym wierszu kodu w tej funkcji.

Omawiając funkcjonalności debugera, posłużymy się przedstawionym na listingu G.1 programem, którego działanie polega na znalezieniu największej spośród trzech podanych liczb. Wykonywanie programu rozpoczyna się od funkcji `main()` zdefiniowanej w wierszach od 8. do 22. Trzy liczby całkowite są w wierszu 15. pobierane za pomocą funkcji `scanf()`. Następnie te liczby są w wierszu 19. przekazywane funkcji `maximum()`, która znajduje największą z nich. Znaleziona wartość jest zwracana funkcji `main()` za pomocą polecenia `return` zdefiniowanego w wierszu 36. Następnie ta wartość zostanie wyświetlona przez polecenie `printf()` w wierszu 19.

LISTING G.1. Znalezienie największej wartości spośród trzech liczb całkowitych

```
1 // Plik: figG_01.c
2 // Znalezienie największej liczby całkowitej spośród trzech podanych
3 #include <stdio.h>
4
5 int maximum( int x, int y, int z ); // Prototyp funkcji
6
7 // Wykonywanie programu rozpoczyna się od funkcji main()
8 int main( void )
9 {
10    int number1; // Pierwsza liczba całkowita
11    int number2; // Druga liczba całkowita
12    int number3; // Trzecia liczba całkowita
13
14    printf( "Podaj trzy liczby całkowite: " );
15    scanf( "%d%d%d", &number1, &number2, &number3 );
16
17    // number1, number2 i number3 to argumenty
18    // wywołania funkcji maximum()
19    printf( "Największa liczba to: %d\n", maximum( number1, number2, number3 ) );
20 } // Koniec funkcji main()
21
```

```
22 // Definicja funkcji maximum()
23 // x, y i z to parametry
24 int maximum( int x, int y, int z )
25 {
26    int max = x; // Przyjęcie założenia, że x to największa wartość
27
28    if ( y > max ) { // Jeżeli wartość y jest większa niż max, należy przypisać wartość y do max
29       max = y;
30    } // Koniec pętli if
31
32    if ( z > max ) { // Jeżeli wartość z jest większa niż max, należy przypisać wartość z do max
33       max = z;
34    } // Koniec pętli if
35
36    return max; // max to największa wartość
37 } // Koniec funkcji maximum()
```

W następujących krokach będziesz korzystać z punktów przerwania i różnych poleceń debugera w celu przeanalizowania wartości zmiennej `number1` zdefiniowanej w wierszu 10. programu z listingu G.1:

1. **Kompilacja programu przeznaczonego do debugowania**. Aby móc używać debugera, program musi być skompilowany wraz z opcją `-g` powodującą wygenerowanie informacji dodatkowych, które debuger potrzebuje do pracy z programem. Oto polecenie, za pomocą którego należy skompilować program omawiany w tym dodatku:

   ```
   $ gcc -g figG_01.c
   ```

2. **Uruchomienie debugera**. Wydaj polecenie `gdb ./a.out` (listing G.2). *Polecenie gdb* uruchamia debuger i wyświetla jego znak zachęty (`gdb`), po którym można wydać polecenia.

LISTING G.2. Uruchomienie debugera GNU wraz z programem

```
$ gdb ./a.out
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html> This is free
➥software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...

(gdb)
```

1. **Działanie programu w debugerze**. Uruchomienie programu w debugerze wymaga wydania polecenia `run` (listing G.3). Jeżeli przed uruchomieniem programu w debugerze nie zdefiniowałeś żadnych punktów przerwania, program zostanie wykonany do końca.

LISTING G.3. Uruchomienie programu bez zdefiniowanych punktów przerwania

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Podaj trzy liczby całkowite: 22 85 17
Największa liczba to: 85

Program exited normally.
(gdb)
```

4. **Dodanie punktów przerwania za pomocą debugera GNU**. Dodaj punkt przerwania w wierszu 14. pliku *figG_01.c*, wydając polecenie `break 14`. *Polecenie **break*** wstawia punkt przerwania w wierszu podanym jako jego argument (np. 22, 85 i 17). Można dodać dowolną liczbę punktów przerwania. Każdy z nich jest identyfikowany przez liczbę oznaczającą kolejność dodania danego punktu. Dlatego identyfikatorem pierwszego punktu przerwania jest `Breakpoint 1`. Dodaj następny punkt przerwania, tym razem w wierszu 19., co wymaga wydania polecenia `break 19` (listing G.4). Ten nowy punkt przerwania będzie miał identyfikator `Breakpoint 2`. Po uruchomieniu programu nastąpi wstrzymanie jego działania w każdym wierszu zawierającym zdefiniowany punkt przerwania, a debuger przejdzie do *trybu debugowania*. Punkty przerwania mogą być dodawane nawet po rozpoczęciu procesu debugowania. (*Uwaga*: jeżeli nie masz numerowanego listingu kodu, zawsze możesz skorzystać z *polecenia **list***, aby wyświetlić kod źródłowy z numerami wierszy. Więcej informacji na temat polecenia `list` otrzymasz, gdy wydasz w debugerze polecenie `help list` po znaku zachęty `gdb`).

LISTING G.4. Zdefiniowanie dwóch punktów przerwania w programie

```
(gdb) break 14
Breakpoint 1 at 0x80483e5: file figG_01.c, line 14.
(gdb) break 19
Breakpoint 2 at 0x8048440: file figG_01.c, line 19.
```

5. **Uruchomienie programu i rozpoczęcie procesu debugowania**. Wydaj polecenie `run`, aby uruchomić program i rozpocząć proces debugowania (listing G.5). Dotarłszy do punktu przerwania zdefiniowanego w wierszu 14., debuger wejdzie do trybu debugowania. W tym momencie zostaniesz poinformowany o dotarciu do punktu przerwania, a na ekranie będzie wyświetlony kod źródłowy znajdujący się w wierszu (14.), który ma być wykonany jako następny.

LISTING G.5. Wykonywanie programu do momentu napotkania pierwszego punktu przerwania

```
(gdb) run
Starting program: /home/user/AppJ/a.out

Breakpoint 1, main() at figG_01.c:14
14                      scanf("%d%d%d", &number1, &number2, &number3);
(gdb)
```

6. **Użycie polecenia `continue` do wznowienia działania programu**. W debugerze wpisz `continue`. *Polecenie **continue*** powoduje wznowienie działania programu, które jest kontynuowane do momentu napotkania kolejnego punktu przerwania (wiersz 19.). Po wyświetleniu komunikatu *Podaj trzy liczby całkowite* wpisz 22, 85 i 17. Debuger poinformuje o dotarciu do drugiego punktu przerwania (listing G.6). Zwróć uwagę na pojawienie się między danymi wyjściowymi debugera zwykłych danych wyjściowych programu zdefiniowanego w pliku *figG_01.c*.

LISTING G.6. Kontynuowanie działania programu do momentu napotkania drugiego punktu przerwania

```
(gdb) continue
Continuing.
Podaj trzy liczby całkowite: 22 85 17

Breakpoint 2, main() at figG_01.c:19
19            printf("Największa liczba to: %d\n", maximum(number1, number2, number3));
(gdb)
```

7. **Przeanalizowanie wartości zmiennej**. Wydaj polecenie `print number1`, aby wyświetlić bieżącą wartość zmiennej `number1` (listing G.7). *Polecenie **print*** pozwala sprawdzić aktualną wartość dowolnej zmiennej. Ta możliwość może pomóc w wyszukaniu i wyeliminowaniu błędów logicznych w kodzie. W omawianym przykładzie wartość zmiennej wynosi 22 — jest to ta wartość, która została przypisana zmiennej po jej wpisaniu w odpowiedzi na komunikat wyświetlony przez program.

LISTING G.7. Wyświetlenie wartości zmiennej

```
(gdb) print number1
$1 = 22
(gdb)
```

8. **Stosowanie wygodnych zmiennych**. Gdy wydajesz polecenie `print`, wynik jego wykonania zostaje umieszczony w wygodnej zmiennej, takiej jak $1. Jest to utworzona przez debuger zmienna tymczasowa o nazwie rozpoczynającej się od znaku dolara, po którym znajduje się liczba. Zmienne wygodne mogą być wykorzystywane do przeprowadzania operacji arytmetycznych i obliczania wartości wyrażeń boolowskich. Wydaj polecenie `print $1`. Debuger wyświetli wartość zmiennej $1 (listing G.8) zawierającą wartość zmiennej `number1`. Zwróć uwagę, że wyświetlenie wartości zmiennej $1 powoduje utworzenie nowej zmiennej wygodnej — $2.

LISTING G.8. Wyświetlanie wartości wygodnej zmiennej

```
(gdb) print $1
$2 = 22
(gdb)
```

9. **Kontynuowanie działania programu**. Wpisz `continue`, aby kontynuować działanie programu. Ponieważ debuger nie napotka żadnych kolejnych punktów przerwania, program będzie działać aż do końca (listing G.9).

LISTING G.9. Dokończenie wykonywania programu

```
(gdb) continue
Continuing
Największa liczba to: 85

Program exited normally
(gdb)
```

10. **Usunięcie punktu przerwania**. Listę wszystkich punktów przerwania w programie można wyświetlić za pomocą polecenia `info break`. Aby usunąć punkt przerwania, należy wpisać `delete`, spację i numer punktu przerwania przeznaczonego do usunięcia. Zacznij od usunięcia pierwszego punktu przerwania przez wydanie polecenia `delete 1`. Usuń także drugi punkt przerwania. Teraz wydaj polecenie `info break`, aby wyświetlić pozostałe punkty przerwania w programie. Debuger powinien wskazywać na brak zdefiniowanych punktów przerwania (listing G.10).

LISTING G.10. Wyświetlanie i usuwanie punktów przerwania

```
(gdb) info break
Num   Type         Disp   Enb   Address
1     breakpoint   keep   y     0x080483e5
          breakpoint already hit 1 time
```

```
2     breakpoint   keep   y     0x08048799
          breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb) info break
No breakpoints or watchpoints
(gdb)
```

11. **Wykonywanie programu bez punktów przerwania**. Wydaj polecenie `run`, aby uruchomić program. Następnie po wyświetleniu komunikatu programu podaj wartości 22, 85 i 17. Ponieważ wcześniej usunąłeś wszystkie punkty przerwania, dane wejściowe programu zostaną wyświetlone bez przechodzenia debugera do trybu debugowania (listing G.11).

LISTING G.11. Wykonywanie programu bez zdefiniowanych punktów przerwania

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Podaj trzy liczby całkowite: 22 85 17
Największa liczba to: 85

Program exited normally. (gdb)
```

12. **Użycie polecenia `quit`**. Użyj *polecenia **quit***, aby zakończyć sesję debugowania (listing G.12). Spowoduje to zakończenie działania debugera.

LISTING G.12. Zakończenie działania debugera za pomocą polecenia quit

```
(gdb) quit
$
```

W tej części dodatku wykorzystaliśmy polecenie `gdb` do uruchomienia debugera, a polecenie `run` do rozpoczęcia debugowania programu. Dodaliśmy punkty przerwania w wierszach funkcji `main()`. Polecenie `break` pozwala zdefiniować punkt przerwania w innym pliku bądź w konkretnej funkcji. Jeżeli wpiszesz `break`, a następnie nazwę pliku, dwukropek i numer wiersza, wówczas punkt przerwania zostanie dodany w wierszu o podanym numerze we wskazanym pliku. Z kolei wpisanie `break` i nazwy funkcji spowoduje, że debuger przejdzie do trybu debugowania w trakcie każdego wywołania tej funkcji.

Zobaczyłeś również, że polecenie `help list` dostarcza więcej informacji na temat sposobu działania polecenia `list` debugera. Jeżeli chcesz się dowiedzieć czegoś o debugerze lub jego poleceniach, wydaj polecenie `help` lub `help nazwa_polecenia`, a otrzymasz więcej informacji na dany temat.

Analizowaliśmy również wartość zmiennej za pomocą polecenia `print` i usunęliśmy punkt przerwania za pomocą polecenia `delete`. Wiesz już, jak używać polecenia `continue`, aby działanie programu było kontynuowane po napotkaniu punktu przerwania, a także jak zakończyć działanie debugera przez wydanie polecenia `quit`.

G.3. Polecenia print i set

W poprzedniej części dodatku pokazaliśmy, jak wykorzystać polecenie `print` debugera do analizy wartości zmiennej podczas działania programu. Z tej części dowiesz się, jak za pomocą polecenia `print` analizować wartość znacznie bardziej skomplikowanych wyrażeń. Poznasz również polecenie `set`, które pozwala przypisywać nowe wartości zmiennym. Zakładamy, że katalog roboczy zawiera przykładowy program omawiany w tym dodatku, a kod źródłowy został skompilowany wraz z opcją `-g`, która umożliwia programowi współpracę z debugerem.

1. **Rozpoczęcie debugowania**. Wydaj polecenie `gdb ./a.out`, aby uruchomić debuger GNU.
2. **Dodanie punktu przerwania**. Dodaj punkt przerwania w wierszu 19. kodu źródłowego, wydając polecenie `break 19` (listing G.13).

LISTING G.13. Dodanie punktu przerwania w programie

```
(gdb) break 19
Breakpoint 1 at 0x8048412: file figG_01.c, line 19.
(gdb)
```

3. **Uruchomienie programu i dotarcie do punktu przerwania**. Wydaj polecenie `run`, aby rozpocząć proces debugowania (rysunek G.14). Spowoduje to wykonywanie funkcji `main()` programu do momentu napotkania punktu przerwania w wierszu 19. Wówczas wykonywanie programu zostanie wstrzymane, a debuger przejdzie do trybu debugowania. Polecenie w wierszu 19. jest tym, które ma zostać wykonane jako następne.

LISTING G.14. Wykonywanie programu do momentu osiągnięcia punktu przerwania w wierszu 19.

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Podaj trzy liczby całkowite: 22 85 17

Breakpoint 1, main() at figG_01.c:19
19            printf("Największa liczba to: %d\n", maximum(number1, number2, number3));
(gdb)
```

4. **Obliczanie wartości wyrażeń arytmetycznych i boolowskich**. Jak zapewne pamiętasz z części G.2, po przejściu debugera do trybu debugowania można analizować wartości zmiennych programu za pomocą polecenia `print`. To polecenie pozwala również obliczać wartości wyrażeń arytmetycznych i boolowskich. Wpisz `print number1 - 2`, wartością zwrotną tego polecenia jest 20 (listing G.15) — wartość zmiennej `number1` nie została zmodyfikowana. Następnie wpisz `print number1 == 20`. Wyrażenie zawierające `==` zwraca 0, jeśli wynikiem działania polecenia jest `false`, lub zwraca 1, jeśli wynikiem działania polecenia jest `true`. W omawianym przykładzie wartością zwrotną jest 0 (listing G.15), ponieważ zmienna `number1` wciąż zawiera wartość 22.

LISTING G.15. Wyświetlanie wartości wyrażeń w debugerze

```
(gdb) print number1 - 2
$1 = 20
(gdb) print number1 == 20
$2 = 0
(gdb)
```

5. **Modyfikowanie wartości**. Istnieje możliwość zmiany wartości zmiennych w trakcie wykonywania programu w debugerze. Jest to cenna możliwość podczas eksperymentowania z różnymi wartościami, a także wyszukiwania błędów logicznych. Polecenie `set` debugera można wykorzystać do zmiany wartości zmiennej. Wpisz `set number1 = 90`, aby zmienić wartość zmiennej `number1`, a następnie `print number1` w celu wyświetlenia nowej wartości zmiennej (listing G.16).

LISTING G.16. Przypisanie wartości zmiennej podczas działania programu w trybie debugowania

```
(gdb) set number1 = 90
(gdb) print number1
$3 = 90
(gdb)
```

6. **Wyświetlanie wyniku działania programu**. Wydaj polecenie `continue`, aby kontynuować wykonywanie programu. Podczas wykonywania wiersza 19. (listingu G.1) zmienne `number1`, `number2` i `number3` zostają przekazane funkcji `maximum()`. Następnie funkcja `main()` wyświetla największą z przekazanych liczb. Zwróć uwagę, że wynikiem jest 90 (listing G.17). To potwierdza, że wydane wcześniej polecenie `set` debugera spowodowało zmianę wartości zmiennej `number1` z 22 na 90.

LISTING G.17. Użycie zmodyfikowanej zmiennej podczas działania programu

```
(gdb) continue
Continuing.
Największa liczba to: 90

Program exited normally.
(gdb)
```

7. **Użycie polecenia `quit`**. Wydaj polecenie `quit`, aby zakończyć sesję pracy z debugerem (listing G.18). Spowoduje to zakończenie działania debugera.

LISTING G.18. Opuszczenie debugera za pomocą polecenia quit

```
(gdb) quit
$
```

W tej części dodatku wykorzystałeś polecenie `print` debugera do obliczenia wartości wyrażeń arytmetycznych i boolowskich. Dowiedziałeś się również, jak za pomocą polecenia `set` zmodyfikować wartość zmiennej podczas działania programu.

G.4. Kontrolowanie sposobu działania programu za pomocą poleceń step, finish i next

Czasami zachodzi potrzeba wykonania programu wiersz po wierszu, aby w ten sposób znaleźć i usunąć błędy. Wykonywanie programu w taki sposób może pomóc w sprawdzeniu, czy kod funkcji działa zgodnie z oczekiwaniami. Polecenia przedstawione w tej części dodatku umożliwiają wykonywanie funkcji wiersz po wierszu, jednoczesne wykonanie wszystkich poleceń w funkcji lub wykonanie tylko pozostałych poleceń funkcji (jeśli część z nich została już wykonana).

1. **Uruchomienie debugera**. Wydaj polecenie `gdb ./a.out`, aby uruchomić debuger GNU.
2. **Dodanie punktu przerwania**. Dodaj punkt przerwania w wierszu 19., wydając polecenie `break 19`.
3. **Uruchomienie programu**. Wydaj polecenie `run`, aby uruchomić program, a następnie w odpowiedzi na wyświetlony komunikat wpisz liczby 22, 85 i 17. Debuger poinformuje o napotkaniu punktu przerwania i wyświetli kod zdefiniowany w wierszu 19. Wstrzyma działanie programu i będzie oczekiwał na wpisanie kolejnego polecenia.

4. **Użycie polecenia `step`**. *Polecenie **step*** pozwala wykonać następne polecenie w programie. Jeżeli jest nim wywołanie funkcji, kontrola nad przebiegiem działania programu zostanie przekazana do wywołanej funkcji. Polecenie `step` umożliwia wejście do funkcji i przeanalizowanie jej poleceń. Na przykład można wykorzystać polecenia `print` i `set` do wyświetlania i modyfikowania zmiennych wewnątrz funkcji. Wpisz `step`, aby wejść do funkcji `maximum()` (listing G.1). Debuger wskazuje na ukończenie kroku i wyświetla następne polecenie do wykonania (listing G.19) — w omawianym przykładzie jest to wiersz 26. w pliku *figG_01.c*.

LISTING G.19. Użycie polecenia step w celu wejścia do funkcji

```
(gdb) step
maximum (x=22, y=85, z=17) at figG_01.c:26
26       int max = x;
(gdb)
```

5. **Użycie polecenia `finish`**. Po wejściu do funkcji wydaj polecenie `finish`. Spowoduje ono wykonanie pozostałej części kodu funkcji i zwrot kontroli nad przebiegiem działania programu do miejsca, w którym nastąpiło wywołanie funkcji. Polecenie `finish` wykona pozostałe polecenia funkcji, a następnie wstrzyma działanie w wierszu 19. funkcji `main()` (listing G.20). Wartość zwrócona przez funkcję `maximum()` zostanie wyświetlona. W długich funkcjach prawdopodobnie zechcesz sprawdzić jedynie kilka kluczowych wierszy kodu, a następnie będziesz kontynuować pracę z kodem komponentu wywołującego tę funkcję. Polecenie `finish` okazuje się użyteczne w sytuacjach, gdy nie chcesz przechodzić wiersz po wierszu przez pozostałą część funkcji.

LISTING G.20. Użycie polecenia finish w celu dokończenia działania funkcji i zwrotu wyniku do komponentu wywołującego tę funkcję

```
(gdb) finish
Run till exit from #0 maximum ( x = 22, y = 85, z = 17) at figG_01.c:26
0x0804842b in main() at figG_01.c:19
19            printf("Największa liczba to: %d\n", maximum(number1, number2, number3));
Value returned is $1 = 85
(gdb)
```

6. **Użycie polecenia `continue` w celu kontynuowania działania programu**. Wpisz `continue`, aby kontynuować wykonywanie programu aż do zakończenia jego działania.
7. **Ponowne uruchomienie programu**. Punkty przerwania pozostają aż do zakończenia sesji debugowania, w której zostały zdefiniowane. Dlatego punkty przerwania dodane w *kroku 2.* pozostały. Wpisz `run`, aby uruchomić program, a następnie wpisz `22`, `85` i `17` po wyświetleniu komunikatu programu. Podobnie jak w *kroku 3.*, program będzie działał do momentu napotkania punktu przerwania w wierszu 19., a następnie debuger wstrzyma jego działanie i będzie oczekiwał na kolejne polecenie (listing G.21).

LISTING G.21. Ponowne uruchomienie programu

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Podaj trzy liczby całkowite: 22 85 17

Breakpoint 1, main() at figG_01.c:19
19            printf("Największa liczba to: %d\n", maximum(number1, number2, number3));
(gdb)
```

8. **Użycie polecenia `next`**. W debugerze wpisz `next`. To polecenie działa podobnie jak `step`, z wyjątkiem sytuacji, gdy następnym poleceniem do wykonania jest wywołanie funkcji. W takim przypadku wywołana funkcja zostanie wykonana w całości, a program przejdzie do następnego po wywołaniu funkcji wiersza zawierającego kod przeznaczony do wykonania (listing G.22). W *kroku 4.* polecenie `step` pozwoliło wejść do wywołanej funkcji. Natomiast w tym kroku polecenie `next` wykonuje funkcję `maximum()` i wyświetla największą spośród wprowadzonych liczb. Następnie debuger wstrzymuje działanie w wierszu 20.

LISTING G.22. Użycie polecenia next w celu całkowitego wykonania funkcji

```
(gdb) next
Największa liczba to: 85
20 } // Koniec funkcji main()
(gdb)
```

9. **Użycie polecenia `quit`**. Wydaj polecenie `quit`, aby zakończyć sesję debugowania (listing G.23). Gdy program jest wciąż uruchomiony, polecenie `quit` powoduje jego natychmiastowe zakończenie zamiast wykonania pozostałych poleceń funkcji `main()`.

LISTING G.23. Opuszczenie debugera za pomocą polecenia quit

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
$
```

W tej części dodatku pokazaliśmy, jak polecenia `step` i `finish` pozwalają debugować funkcje wywoływane podczas działania programu. Dowiedziałeś się również o możliwości pominięcia wywołania funkcji za pomocą polecenia `next`. Wiesz już także, że wydanie polecenia `quit` powoduje zakończenie sesji debugowania.

G.5. Polecenie watch

*Polecenie **watch*** nakazuje debugerowi obserwowanie danych składowych. Gdy dane składowe będą miały zostać zmienione, debuger poinformuje o tym fakcie. W tej części dodatku wykorzystasz polecenie `watch`, aby sprawdzić, jak wartość zmiennej `number1` jest modyfikowana podczas działania programu.

1. **Uruchomienie debugera**. Wydaj polecenie `gdb ./a.out`, aby uruchomić debuger GNU.
2. **Dodanie punktu przerwania i uruchomienie programu**. Dodaj punkt przerwania w wierszu 19., wydając polecenie `break 19`. Następnie uruchom program za pomocą polecenia `run`. Debuger i program wstrzymają działanie po napotkaniu punktu przerwania w wierszu 14. (listing G.24).

LISTING G.24. Wykonywanie programu do momentu osiągnięcia pierwszego punktu przerwania

```
(gdb) break 14
Breakpoint 1 at 0x80483e5: file figG_01.c, line 14.
(gdb) run
Starting program: /home/user/AppJ/a.out

Breakpoint 1, main () at figG_01.c:14
14          printf("Podaj trzy liczby całkowite: ");
(gdb)
```

3. **Obserwowanie elementu składowego**. Zdefiniuj obserwowanie zmiennej `number1`, wydając polecenie `watch number1` (listing G.25). Ten punkt obserwacji zostanie oznaczony etykietą `watchpoint 2`, ponieważ zastosowanie mają takie same etykiety jak w przypadku punktów przerwania. Obserwować można dowolną zmienną lub element składowy obiektu aktualnie znajdującego się w zasięgu. Gdy wartość obserwowanego elementu ulegnie zmianie, debuger przejdzie do trybu debugowania i poinformuje o zmianie wartości.

LISTING G.25. Zdefiniowanie punktu obserwacji elementu danych

```
(gdb) watch number1
Hardware watchpoint2: number1
(gdb)
```

4. **Kontynuowanie działania programu**. Wpisz `continue` w celu kontynuowania działania programu. Po wyświetleniu komunikatu *Podaj trzy liczby całkowite* wpisz trzy liczby całkowite. Debuger poinformuje o zmianie wartości zmiennej `number1` i przejdzie do trybu debugowania (listing G.26). Starą wartością zmiennej `number1` jest jej wartość sprzed inicjalizacji. Ta wartość może być inna po każdym uruchomieniu programu. Taka nieprzewidywalna (i często niepożądana) wartość pokazuje, dlaczego tak duże znaczenie ma inicjalizacja wszystkich zmiennych w C przed ich użyciem.

LISTING G.26. Wejście do trybu debugowania po zmianie wartości zmiennej

```
(gdb) continue
Continuing.
Podaj trzy liczby całkowite: 22 85 17
Hardware watchpoint 2: number1

Old value = -1208401328
New value = 22
0xb7e6c692 in _IO_vfscanf() from /lib/i686/cmov/libc.so.6
(gdb)
```

5. **Kontynuowanie działania programu**. Wpisz `continue` — program dokończy wykonywanie funkcji `main()`. Debuger usuwa punkt obserwacji zmiennej `number1`, ponieważ zostanie ona usunięta po zakończeniu działania funkcji `main()`. Usunięcie punktu obserwacji powoduje przejście debugera do trybu debugowania. Ponownie wpisz `continue`, aby zakończyć działanie programu (listing G.27).

LISTING G.27. Kontynuowanie działania programu aż do jego zakończenia

```
(gdb) continue
Continuing.
Największa liczba to: 85

Watchpoint 2 is deleted because the program has left the block in which its expression is valid.
0xb7e4aab7 in exit() from /lib/i686/cmov/libc.so.6
(gdb) continue
Continuing

Program exited normally
(gdb)
```

1. **Ponowne uruchomienie debugera i wyzerowanie wartownika zmiennej**. Wydaj polecenie `run` w celu ponownego uruchomienia debugera. Raz jeszcze zdefiniuj obserwowanie zmiennej `number1` przez wydanie polecenia `watch number1`. Ten punkt obserwacji będzie oznaczony etykietą `watchpoint 3`. Wpisz `continue`, aby kontynuować wykonywanie programu (listing G.28).

LISTING G.28. Wyzerowanie wartownika elementu składowego

```
(gdb) run
Starting program: /home/users/AppJ/a.out

Breakpoint 1, main () at figG_01.c:14
14          printf("Podaj trzy liczby całkowite: ");
(gdb) watch number1
Hardware watchpoint 3: number1
(gdb) continue
Continuing
Hardware watchpoint 3: number1

Old value = -1208798640
New value = 22
0xb7e0b692 in _IO_vfscanf() from /lib/i686/cmov/libc.so.6
(gdb)
```

7. **Usunięcie punktu obserwacji elementu składowego**. Zakładamy, że element składowy chcesz obserwować jedynie przez pewien okres czasu w trakcie działania programu. W omawianym przykładzie w celu usunięcia punktu obserwacji zmiennej `number1` należy wydać polecenie `delete 3` (listing G.29). Wpisz `continue` — program zakończy działanie bez ponownego wejścia do trybu debugowania.

LISTING G.29. Usunięcie punktu obserwacji elementu składowego

```
(gdb) delete 3
(gdb) continue
Continuing.
Największa liczba to: 85

Program exited normally.
(gdb)
```

W tej części dodatku użyliśmy polecenia `watch`, aby umożliwić debugerowi informowanie o zmianie wartości zmiennej. Polecenie `delete` natomiast posłużyło nam do usunięcia punktu obserwacji elementu składowego jeszcze przed zakończeniem działania programu.

G.6. Podsumowanie

W tym dodatku pokazaliśmy, jak wstawiać i usuwać punkty przerwania w debugerze. Punkt przerwania pozwala wstrzymać wykonywanie programu, aby można było przeanalizować wartości zmiennych za pomocą polecenia `print` debugera, co będzie pomocne podczas wyszukiwania i usuwania błędów logicznych. Polecenie `print` wykorzystaliśmy do analizy wartości wyrażenia, a polecenie `set` — do zmiany wartości zmiennej. Omówiliśmy także wybrane polecenia debugera (m.in. `step`, `finish` i `next`), które pozwalają sprawdzić, czy funkcja jest wykonywana prawidłowo. Dowiedziałeś się również, jak używać polecenia `watch` do monitorowania elementu składowego w trakcie jego cyklu życiowego. Zobaczyłeś także, że polecenie `info break` wyświetla listę wszystkich punktów przerwania i punktów obserwacji zdefiniowanych w programie, polecenie `delete` zaś usuwa pojedyncze punkty przerwania i punkty obserwacji.