11 Debugowanie, gdb

Uruchomienie programu - ustawianie argumentów

Zanim uruchomimy nasz program, musimy określić:

* Program który będziemy debugować. Wykonujemy to komendą
  `(gdb) file <program>`
  która wczytuje plik programu z bierzącego katalogu. Można też, jeszcze w chwili uruchamiania debuggera, wywołać go z parametrem:
  `$ gdb <program>`
* Argumenty które mają być przekazane do programu. Określamy je komendą:
  `(gdb) set args <argumenty>`
  Można też, jeszcze w chwili uruchamienia gdb, wywołać jako:
  `$ gdb --args <program> <argumenty>`

* Katalog roboczy. Domyślnie jest to katalog w którym uruchomiliśmy gdb. Można go dowolnie zmieniać komendą `cd`, podobnie jak w bashu. Podobnie też `pwd` wypisuje aktualny katalog.
* Zmienne środowiskowe. Domyślnie środowisko jest takie same jak to w którym uruchomiliśmy gdb. Można to jednak zmienić na potrzeby programu który debugujemy:

`path <katalog>`	dodaje `<katalog>` do zmiennej środowiskowej `PATH`
`set environment <zmienna> = <wartosc>`	ustawia wartośc zmiennej środowiskowej `<zmienna>` na `<wartosc>`.
`unset environment <zmienna>`	usuwa zmienną środowiskową `<zmienna>`

Oczywiście wszystkie te zmiany mają zasięg lokalny. Nasze środowisko basha pozostaje niezmienione.

Debugowanie obiektów polimorficznych

Debugowanie obiektów C++ może stanowić trochę problemów. Załóżmy, że mamy klasę bazową `Base` i z niej dziedziczącą `Derived`. Jeśli dany wskaźnik `ptr` o typie `Base*` wskazuje na obiekt typu `Derived`, to `gdb` domyślnie debuguje go jako zwykły obiekt `Base`.

```
(gdb) print ptr
$1 = (Base *) 0x613c20
(gdb) print *ptr
$2 = {_vptr.Base = 0x400878 <vtable for Derived+16>, x = 1}
(gdb) print ptr->y
There is no member or method named y.
```

Jeśli klasa zawiera metody wirtualne (a zwykle tak jest przy typach polimorficznych),

to oprócz zwykłych pól klasy zawiera specjalny wskaźnik `_vptr`. Jest to typowe rozwiązanie przy realizacji metod wirtualnych: `_vptr` to wskaźnik na tablicę adresów metod wirtualnych dla danego obiektu. Każda klasa posiada jedną, globalną tablicę. Podczas debugowania, adres takiej tablicy może posłużyć do identyfikacji prawdziwego typu obiektu. W naszym przypadku możemy odkryć, że nasz wskaźnik `ptr` najprawdopodobniej wskazuje na obiekt typu `Derived`, bo `vptr` wskazuje do „wirtualnej tablicy klasy `Derived`" (`vtable for Derived`).

Problem można łatwo rozwiązać zmieniając sposób pracy komendy `print`.

```
(gdb) set print object
(gdb) print *ptr
$3 = (Derived) {<Base> = {_vptr.Base = 0x400878 <vtable for
   Derived+16>, x = 1}, y = 0}
(gdb) print ptr1->y
$4 = 0
```

GDB wykorzystał informację o rzeczywistym typie zmiennej i wypisał pełną informację o obiekcie. Zaznaczone jest która część należy do klasy `Base`, a która do `Derived`. Ponadto, rzeczywisty typ zmiennej wykorzystywany jest do realizacji operatora `->`, co poprzednio było niemożliwe.

Jeśli klasa jest skomplikowana a to co wypisuje `print` nieczytelne, to możemy zmienić tryb pracy `print` w następujący sposób:

```
(gdb) set print pretty
(gdb) print *ptr
$5 = (Derived) {
  <Base> = {
    _vptr.Base = 0x400878 <vtable for Derived+16>,
    x = 1
  },
  members of Derived:
  y = 0
}
```

W ten sposób każde pole obiektu jest wypisane w osobnej linijce, co w zamyśle ma ułatwić czytelność komunikatu. Niestety, jeśli obiekt jest duży, takie szczegółowe wypisywanie wszystkich informacji może utrudnić a nie ułatwić pracę z debugowaniem. Jeśli wiemy, które najczęściej pola obiektu są nam potrzebne, to najlepiej po prostu napisać własną funkcję debugującą dla danej klasy. Nie musimy tej funkcji nigdzie w kodzie uruchamiać, wystarczy, że ona istnieje.

Dla przykładu jeśli zdefiniujemy:

```
virtual void Base::debug() { std::cout << "Base(x=" << x << ")\n"; }
virtual void Derived::debug() {
    std::cout << "Derived(x=" << x << ", y=" << y << ")\n";
}
```

To podczas działania programu możemy taką metodę wywołać:

```
(gdb) print ptr->debug()
```

```
Derived(x=1, y=0)
$1 = void
```

albo jeśli nie interesuje nas wynik wywołania:

```
(gdb) call ptr->debug()
Derived(x=1, y=0)
```

W ten sposób to my dokładnie decydujemy co zostaje wypisane. Może to być szczególnie przydatne jeśli debugujemy drzewa i grafy przy dużej liczbie odwołań.

Debugowanie kompilatorem

Poza konieczną do debugowania za pomocą `gdb` flagą `-g` kompilator sam w sobie ma też parę opcji umożliwiających debugowanie prostych błędów. Kilka wybranych flag:

-fsanitize=address sprawdza dostępy do niezarezerwowanej bądź wcześniej zwolnionej pamięci

-fsanitize=leak wykrywa wycieki pamięci

-fsanitize=undefined ostrzega gdy wykonywana jest operacja, której wynik nie jest zdefiniowany przez standard, ma wiele podrzędnych opcji odpowiadających za poszczególne operacje

-fsanitize=thread wykrywa potencjalne problemy z dostępem do danych wątków, nie może być stosowany równocześnie z wieloma innymi flagami

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

-fstack-protector sprawdza czy funkcje nie nadpisują kawałków stosu do nich nie należących

Większość z tych opcji produkuje znacznie wolniejsze i większe wersje programów, które w żadnym razie nie nadają się do używania w praktyce. Główną ich zaletą jest natomiast automatyzacja – bez zastanawiania się nad debugowaniem można wykryć wiele pospolitych błędów.