# **Bài 11. Tìm lỗi chương trình với gdb**

**Mục tiêu:**

1. *Luyện tập sử dụng công cụ GNU gdb trong việc lần bước và tìm lỗi trong chương trình*

**Yêu cầu nộp bài** *: Toàn bộ code và output phần A (trừ bài 1)*

*Với mỗi bài, bạn cần mark-copy nội dung màn hình cmd/terminal, rồi paste vào một file text, chẳng hạn A2.txt, để nộp trong thư mục của bài tập.*

***Lưu ý: chép bài hoặc cho chép bài sẽ dẫn đến trượt môn học!***

## Thực hành

gdb là công cụ hỗ trợ debug, nó nằm trong bộ GNU. Cũng như gcc, đây là công cụ debug được sử dụng rộng rãi nhất. Nó được sử dụng trực tiếp từ dòng lệnh, hoặc được sử dụng gián tiếp thông qua tiện ích debug của các IDE. Trong bài này, bạn sẽ thực hành việc sử dụng trực tiếp gdb cho việc tìm lỗi chương trình.

1. ***Chuẩn bị.*** Hãy mở cửa sổ dòng lệnh, kiểm tra xem lệnh gdb có chạy hay không. Nếu thấy thông báo tương tự như “Command not found”, cần kiểm tra biến môi trường path xem có chứa đường dẫn đến một cài đặt của GNU hay không, với các máy cài DevC++, đường dẫn cần có là C:\Program Files (x86)\Dev-Cpp\MinGW64\bin (có thể thay đổi chút ít theo đường dẫn mà DevC++ được cài tại máy bạn.)
2. ***Dùng gdb tìm lỗi chia cho 0.*** Cho chương trình ví dụ dưới đây, hãy lưu vào một file, chẳng hạn Divide.cpp. Chương trình này sẽ crash do lỗi chia cho 0 ở lần gọi hàm divide thứ hai với tham số b = 0.

#include <iostream>  
using namespace std;   
  
int divide(int, int);   
int main()   
{   
 int x = 4, y = 2;   
 cout << divide(x, y);   
   
 x = 3; y = 0;   
 cout << divide(x, y);   
   
 return 0;   
}   
  
int divide(int a, int b)   
{   
 return a / b;   
}

Để có thể debug, bạn cần dịch chương trình với tham số -g. Lệnh dịch như sau:

g++ -g Divide.cpp -o Divide

Kết quả là file nhị phân thực thi được, có tên Divide.exe (môi trường Windows) hoặc Divide (môi trường Linux).

Khi chạy chương trình. Nếu tại Linux, bạn sẽ thấy thông báo lỗi Floating point exception (core dumped). kèm theo một file có tên *core* chứa thông tin lỗi tại thư mục hiện tại. Còn tại Windows, bạn sẽ không thấy thông báo lỗi đó cũng như file core.

Để bắt đầu debug, bạn chạy lệnh

gdb Divide

Sẽ thấy output đại loại:

GNU gdb (rubenvb-4.7.2-release) 7.5.50.20120920-cvs

Copyright (C) 2012 Free Software Foundation, Inc.

...

Reading symbols from D:\LTNC\samples\Divide.exe...done.

(gdb)

Bạn đang ở trong chương trình gdb, (gdb) chính là dấu nhắc lệnh. Tại đó bạn sẽ gõ những lệnh muốn gdb thực hiện. Đầu tiên là lệnh r để chạy chương trình trong chế độ debug

**Chạy chương trình: r**

(gdb) **r**

Starting program: D:\LTNC\samples\Divide.exe

[New Thread 109464.0x16688]

2

Program received signal SIGFPE, Arithmetic exception.

0x0000000000401efd in divide (a=3, b=0) at Divide.cpp:18

18 return a / b;

Giải nghĩ output trên: 2 là kết quả in ra của lần gọi divide() thứ nhất. Chương trình nhận được lỗi Arithmetic exception tại dòng số 18, với nội dung return a/b; trong file Divide.cpp. Ngữ cảnh của dòng gây lỗi đó là trong lời gọi hàm divide(a=3, b=0).

**Để xem mã nguồn xung quanh vùng xuất hiện lỗi**, dùng **lệnh l** (viết tắt của list)

(gdb) **l**

13 return 0;

14 }

15

16 int divint(int a, int b)

17 {

18 return a / b;

19 } (gdb)

Để **xem thứ tự các lời gọi hàm đang nằm trong stack** tại thời điểm xảy ra lỗi, dùng lệnh **where**

(gdb) **where**

#0 0x0000000000401efd in divide (a=3, b=0) at Divide.cpp:18

#1 0x0000000000401ed2 in \_\_fu0\_\_ZSt4cout () at Divide.cpp:11

Output trên cho thấy lỗi xảy ra bên trong lời gọi hàm divide(3,0), lời gọi hàm đó xuất phát từ bên trong lời gọi hàm tại dòng 11 (nói cách khác là main() gọi divide(3,0) và lỗi phát sinh bên trong divide(3,0))

Hiện tại, ta đang ở trong ngữ cảnh xảy ra lỗi, đó lời gọi hàm trên cùng (divide(3,0).

**Để xem giá trị của biến, ta dùng lệnh p và tên biến**, chẳng hạn muốn xem giá trị biến a và biến b:

(gdb) **p a**

$1 = 3

(gdb) **p b**

$2 = 0

Ta thử xem giá trị của x.

(gdb) **p x**

No symbol "x" in current context.

Kết quả là trong ngữ cảnh hiện tại không có x. Nhớ rằng ta đang ở trong ngữ cảnh của lời gọi hàm divide() với các biến địa phương a và b. Trong khi đó, x là biến địa phương của hàm main() nên nó không nằm trong ngữ cảnh hiện tại.

Để chuyển ngữ cảnh ra ngoài hàm divide, ta dùng lệnh up

(gdb) **up**

#1 0x0000000000401ed2 in \_\_fu0\_\_ZSt4cout () at Divide.cpp:11

11 cout << divide(x, y);

Output cho thấy ngữ cảnh đã chuyển ra dòng 11 trong hàm main.

Giờ ta có thể hỏi giá trị của x

(gdb) **p x**

$3 = 3

Muốn **chuyển ngữ cảnh** theo hướng ngược lại của chuỗi lời gọi hàm, **đi sâu hơn vào trong vùng xuất hiện lỗi**, ta dùng **lệnh down**.

(gdb) **down**

#0 0x0000000000401efd in divide (a=3, b=0) at Divide.cpp:18

18 return a / b;

Ta đã quay lại ngữ cảnh của divide(3,0).

**Thoát khỏi gdb bằng lệnh q** (quit).

Lỗi trong ví dụ này quá đơn giản. Nhưng nó trình diễn cách ta có thể di chuyển và xem xét các thông tin tại thời điểm xảy ra lỗi, và từ đó tìm ra nguyên nhân lỗi.

Nếu bạn có file core khi chương trình crash thì bạn cũng có thể dùng gdb để “quay lại” thời điểm xảy ra lỗi bằng lệnh **gdb crash core**. Trong đó crash là tên chương trình vừa bị crash.

1. ***Tìm lỗi sử dụng bộ nhớ.*** Dùng chương trình ví dụ sau đây, lưu vào file chẳng hạn tên crash.cpp.

#include <iostream>   
using namespace std;   
  
void setint(int\*, int);   
int main()   
{   
 int a;   
 setint(&a, 10);   
 cout << a << endl;   
   
 int\* b;   
 setint(b, 10);   
 cout << \*b << endl;   
   
 return 0;   
}   
  
void setint(int\* ip, int i)  
{  
 \*ip = i;   
}

Hãy biên dịch crash.cpp ở chế độ hỗ trợ debug (xem cách làm tại bài trước). Chạy gdb cho crash đã được biên dịch và chạy crash trong đó. Ta có output đại loại như sau:

...

Reading symbols from D:\LTNC\samples\crash.exe...done.

(gdb) **r**

Starting program: D:\LTNC\samples\crash.exe

[New Thread 14672.0x21de0]

10

Program received signal SIGSEGV, Segmentation fault.

0x0000000000402017 in setint (ip=0x7fef48a4c9f, i=10) at crash.cpp:20

20 \*ip = i;

Output cho thấy lỗi xảy ra tại dòng 20, trong ngữ cảnh lời gọi hàm setint (ip=0x7fef48a4c9f, i=10).

Tiếp tục dùng lệnh where để xác định chính xác ngữ cảnh sinh lỗi và nguyên nhân của lỗi.

Ở một môi trường khác, chẳng hạn Linux, chương trình có thể không crash tại một hàm mà ta viết, main hay setint, khi đó sẽ khó tìm lỗi hơn do không có thông tin về ngữ cảnh và biến địa phương, chẳng hạn đây là kết quả tại một môi trường Linux:

$ gdb crash   
(gdb) r   
Starting program: /home/tmp/crash   
10   
10   
Program received signal SIGSEGV, Segmentation fault.   
0x4000b4d9 in \_dl\_fini () from /lib/ld-linux.so.2   
  
(gdb) where   
#0 0x4000b4d9 in \_dl\_fini () from /lib/ld-linux.so.2   
#1 0x40132a12 in exit () from /lib/libc.so.6   
#2 0x4011cdc6 in \_\_libc\_start\_main () from /lib/libc.so.6   
  
#3 0x080485f1 in \_start ()   
(gdb)

Khi đó, ta sẽ có nhu cầu **lần bước chương trình** để tìm dòng lệnh gây lỗi.

Ta dùng các lệnh sau:

* **b main** để đặt breakpoint tại đầu hàm main, mục đích là để khi chạy chương trình, nó sẽ dừng lại tại điểm đó để đợi lệnh điều khiển gdb của ta. Ta có thể thay main bằng các đầu hàm khác.
* **r** để chạy chương trình tới điểm breakpoint tiếp theo. Có thể đặt nhiều điểm breakpoint.
* **n** để chạy một lệnh tiếp theo trong code.
* Phím Enter để chạy lệnh debug gần nhất. Chẳng hạn sau lệnh n, nếu ta bấm enter thì gdb sẽ lại chạy lệnh n, ta sẽ được hiệu ứng chạy dần từng lệnh.

Dưới đây là output của việc đặt breakpoint tại đầu hàm main và sau đó chạy lần bước từng dòng một.

(gdb) **b main**

Breakpoint 1 at 0x401e81: file crash.cpp, line 6.

(gdb) **r**

Starting program: D:\chauttm\a-teaching\LTNC\samples\crash.exe

[New Thread 126376.0x1c7a0]

Breakpoint 1, main () at crash.cpp:6

6 {

(gdb) **n**

8 setint(&a, 10);

(gdb) (gõ Enter để chạy lệnh gần nhất là n)

9 cout << a << endl;

(gdb) (gõ Enter để chạy lệnh gần nhất là n)

10

12 setint(b, 10);

(gdb) (gõ Enter để chạy lệnh gần nhất là n)

Program received signal SIGSEGV, Segmentation fault.

0x0000000000402017 in setint (ip=0x7fef48a4c9f, i=10) at crash.cpp:20

20 \*ip = i;

(gdb)

Program received signal SIGSEGV, Segmentation fault.

0x0000000000402017 in setint (ip=0x7fef48a4c9f, i=10) at crash.cpp:20

20 \*ip = i;

(gdb)

[Inferior 1 (process 126376) exited with code 030000000005]

PHần cuối của output cho thấy lệnh setint tại dòng 12 gọi đến lệnh \*ip=i tại dòng 20 và gây lỗi segmentation fault.

1. ***Debug tràn mảng.*** Hãy chọn từ các tuần trước một bài tập thử nghiệm vòng lặp làm tràn mảng để debug bằng gdb cho đến khi tìm được điểm gây lỗi trong code, copy toàn bộ output của màn hình gdb vào một file text. Hãy chọn từ các tuần trước một bài tập thử nghiệm tràn mảng để debug.
2. ***Debug lỗi con trỏ.*** Hãy chọn từ các tuần trước một bài tập thử nghiệm lỗi để debug bằng gdb cho đến khi tìm được điểm gây lỗi trong code, copy toàn bộ output màn hình của gdb vào file text. Cần ghi rõ bạn chọn debug bài nào, nộp kèm mã nguồn cho bài đó.

## Phụ lục: Các lệnh gdb cơ bản

* **b main** - Đặt breakpoint ở đầu chương trình
* **b** - Đặt breakpoint tại dòng hiện tại
* **b N** - Đặt breakpoint tại dòng N
* **b +N** - Đặt breakpoint sau N dòng kể từ dòng hiện tại
* **b fn** - Đặt breakpoint tại đầu hàm "fn"
* **d N** - Xóa breakpoint thứ N
* **info break** - liệt kê các breakpoint hiện có
* **r** - Chạy chương trình từ đầu cho đến khi gặp breakpoint hoặc lỗi
* **c** - Chạy tiếp chương trình cho đến khi gặp breakpoint hoặc lỗi
* **f** - chạy cho đến hết hàm hiện tại
* **s** - chạy lệnh tiếp theo
* **s N** - chạy N lệnh tiếp theo
* **n** - giống s, nhưng không nhảy vào trong các hàm
* **p var** - in giá trị hiện tại của biến "var"
* **bt** - in stack trace
* **u** - Đi lên một bậc trong stack (ra ngoài)
* **d** - Đi xuống một bậc trong stack (vào sâu)
* **q** - Thoát gdb