HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

BÁO CÁO BÀI THỰC HÀNH

Kiểm thử xâm nhập

Gỡ rối chương trình C++ với GDB

Giảng viên: Đinh Trường Duy

Nhóm lớp: 01

Sinh viên: Hoàng Trung Kiên

Mã sinh viên: B20DCAT098

Hà Nội – 2024

Mục lục

1. Mục đích.

Bài thực hành này hướng dẫn sử dụng tiện ích GDB để gỡ lỗi một chương trình C++. Gỡ lỗi là điều không thể tránh khỏi. Mỗi lập trình viên tại một thời điểm nào đó trong sự nghiệp phải gỡ lỗi một đoạn mã. Có nhiều cách để bắt đầu gỡ lỗi, từ in thông báo ra màn hình, sử dụng trình gỡ lỗi hoặc chỉ nghĩ về những gì chương trình đang làm và đưa ra một phỏng đoán về vấn đề. Trước khi một lỗi có thể được sửa, nguồn của lỗi phải được xác định. Ví dụ, với các lỗi phân đoạn, sẽ rất hữu ích khi biết lỗi seg đang xảy ra trên dòng mã nào. Khi dòng mã được đề cập đã được tìm thấy, sẽ rất hữu ích nếu biết về các giá trị trong phương thức đó, cái gì đã gọi phương thức và tại sao (cụ thể) lại xảy ra lỗi. Sử dụng trình gỡ lỗi làm cho việc tìm kiếm tất cả thông tin này trở nên đơn giản.

2. Yêu cầu đối với sinh viên.

Có kiến thức về ngôn ngữ lập trình C++
Biết cách biên dịch và thực thi một chương trình C++ bất kỳ.
Tìm hiểu về gdb

3. Nội dung thực hành.

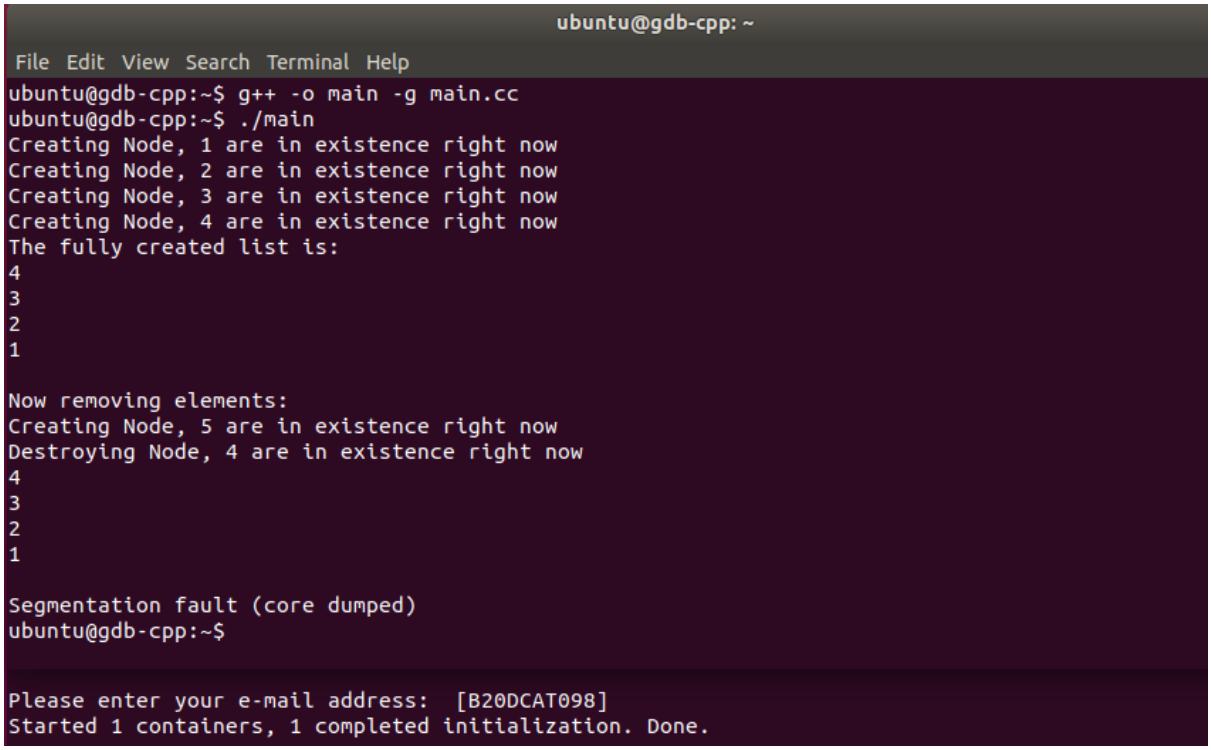
- Khởi động bài lab: labtainer -r gdb-cpp

```
student@ubuntu:~/labtainer/labtainer-student$  labtainer -r gdb-cpp
latest: Pulling from labtainers/gdb-cpp.gdb-cpp.student
b79abeadfe19: Pull complete
70d301b10709: Pull complete
d19a52571a2a: Pull complete
0673281907ba: Pull complete
290dc58d5cbd: Pull complete
b6b5245c382e: Pull complete
a70ac3558eb1: Pull complete
ceb4281d01fe: Pull complete
dcae9abb9e28: Pull complete
0e08b374be84: Pull complete
Digest: sha256:13b7a01037ee57761c0c38d38047d55228cfca366adf684c22d214e1f46b5c7c
Status: Downloaded newer image for labtainers/gdb-cpp.gdb-cpp.student:latest
non-network local connections being added to access control list

Please enter your e-mail address:  [B20DCAT098]
```

- Build và chạy:

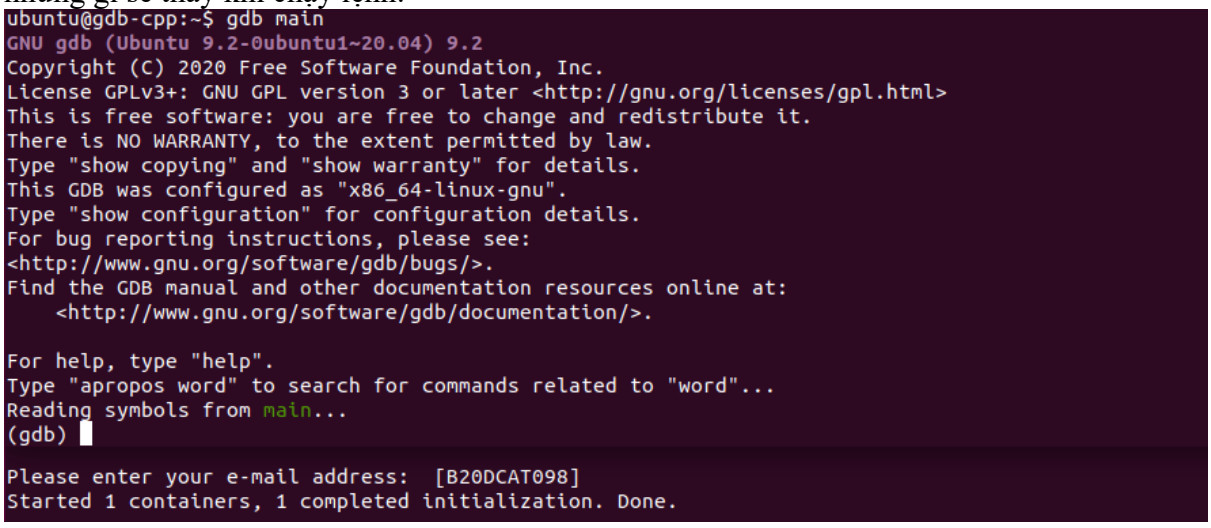
Mã và một tệp makefile đơn giản nằm trên máy tính gdb-cpp khởi động khi chạy lab này. Mã chương trình này rất đơn giản và bao gồm hai định nghĩa lớp, một nút và một danh sách liên kết. Ngoài ra còn có một chương trình đơn giản để kiểm tra danh sách. Tất cả mã được đặt vào một tệp duy nhất để minh họa quá trình gỡ lỗi dễ dàng hơn. Chương trình và makefile nằm trên máy tính được tạo khi lab khởi động và có thể được nhìn thấy trong thư mục chính.



- Nạp một chương trình trong gdb:

Trước tiên phải khởi chạy trình gỡ lỗi. Trình gỡ lỗi được gọi là gdb và ta có thể cho nó biết tệp nào cần gỡ lỗi tại dấu nhắc shell. Vì vậy, để gỡ lỗi main, ta gõ gdb main.

gdb sẽ hiện ra và đợi người dùng gõ lệnh. Tâ cần chạy chương trình để trình gỡ lỗi có thể giúp xem điều gì sẽ xảy ra khi chương trình bị treo. Nhập run tại dấu nhắc (gdb). Đây là những gì sẽ thấy khi chạy lệnh:



```
(gdb) run
Starting program: /home/ubuntu/main
Creating Node, 1 are in existence right now
Creating Node, 2 are in existence right now
Creating Node, 3 are in existence right now
Creating Node, 4 are in existence right now
The fully created list is:
4
3
2
1

Now removing elements:
Creating Node, 5 are in existence right now
Destroying Node, 4 are in existence right now
4
3
2
1


Program received signal SIGSEGV, Segmentation fault.
0x000055555555586c in Node<int>::next (this=0x0) at main.cc:28
28        Node<T>* next () const { return next_; }
(gdb)
```

Chương trình bị lỗi. Vì vậy hãy xem loại thông tin nào có thể thu thập. Kiểm tra sự cố, ta có thể thấy chương trình ở dòng 28 của main.cc, điểm này chỉ đến 0 và chúng ta có thể thấy dòng mã đã được thực thi. Nhưng ta cũng muốn biết cái gì đã gọi phương thức này và muốn có thể kiểm tra các giá trị trong các phương thức gọi. Vì vậy, tại dấu nhắc gdb, nhập backtrace cho kết quả sau:

```
(gdb) backtrace
#0  0x000055555555586c in Node<int>::next (this=0x0) at main.cc:28
#1  0x0000555555555763 in LinkedList<int>::remove (this=0x55555556aeb0,
    item_to_remove=@0x7fffffffe43c: 1) at main.cc:77
#2  0x00005555555553b1 in main (argc=1, argv=0x7fffffffe558) at main.cc:120
(gdb)
```

Vì vậy, ngoài những gì đã biết về phương thức hiện tại và các biến cục bộ, bây giờ ta cũng có thể xem những phương thức nào được gọi và tham số của chúng là gì. Ví dụ, có thể thấy rằng chương trình đã được gọi bởi LinkedList <int> :: remove () trong đó mục tham số cần loại bỏ ở địa chỉ 0x7fffffffe43c. Nó có thể giúp hiểu lỗi nếu biết giá trị của mục cần xóa, vì vậy cần xem tiếp giá trị tại địa chỉ của mục cần xóa. Điều này có thể được thực hiện bằng lệnh x sử dụng địa chỉ làm tham số. (”X” có thể được coi là viết tắt của “exam”.) Đây là kết quả của việc chạy lệnh:

```
(gdb) x 0x7fffffffe43c
0x7fffffffe43c: 0x00000001
(gdb)
```

Vì vậy, chương trình gặp sự cố trong khi cố gắng chạy LinkedList <int> :: remove với tham số là 1. Bây giờ ta đã thu hẹp vấn đề xuống một hàm cụ thể và một giá trị cụ thể cho tham số - Các điểm ngắt: Bây giờ ta biết segfault đang xảy ra ở đâu và khi nào, chúng ta muốn xem chương trình đang làm gì trước khi nó bị treo. Một cách để làm điều này là thực hiện lần lượt từng câu lệnh của chương trình cho đến khi đi đến điểm thực thi mà ta muốn xem điều gì đang xảy ra. Cách làm này đúng, nhưng đôi khi ta có thể muốn chỉ chạy đến một phần mã cụ thể và dừng thực thi tại thời điểm đó để có thể kiểm tra dữ liệu tại vị trí đó. Nếu đã từng sử dụng trình gỡ lỗi, bạn có thể quen thuộc với khái niệm điểm ngắt (breakpoint). Về cơ bản, điểm ngắt là một dòng trong mã nguồn nơi trình gỡ lỗi sẽ ngắt thực thi. Trong ví dụ này, để xem mã trong LinkedList <int> :: remove (), ta đặt một điểm ngắt ở dòng 52 của main.cc.

```
(gdb) break 52
Breakpoint 1 at 0x5555555555f9: file main.cc, line 52.
```

```
(gdb) condition 1 item_to_remove==1
(gdb)
```

- Lệnh Step:

Như vậy ta đã đặt một điểm ngắt có điều kiện và bây giờ muốn xem qua hàm này từng dòng một và xem liệu có thể xác định được nguồn gốc của lỗi hay không. Điều này được thực hiện bằng cách sử dụng lệnh step. gdb có một tính năng hay là khi nhấn enter mà không cần gõ lệnh thì lệnh cuối cùng sẽ tự động được sử dụng. Bằng cách đó, chúng ta có thể thực hiện một cách đơn giản bằng cách nhấn vào phím enter sau khi bước đầu tiên đã được nhập. Đây là kết quả:

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ubuntu/main
Creating Node, 1 are in existence right now
Creating Node, 2 are in existence right now
Creating Node, 3 are in existence right now
Creating Node, 4 are in existence right now
The fully created list is:
4
3
2
1

Now removing elements:
Creating Node, 5 are in existence right now
Destroying Node, 4 are in existence right now
4
3
2
1


Breakpoint 1, LinkedList<int>::remove (this=0x55555556aeb0,
    item_to_remove=@0x7fffffffe43c: 1) at main.cc:52
52          Node<T> *marker = head_;
```

```
(gdb) step
53          Node<T> *temp = 0;  // temp points to one behind as we iterate
(gdb)
55          while (marker != 0) {
(gdb)
56            if (marker->value() == item_to_remove) {
(gdb)
Node<int>::value (this=0x7ffff7f1344e <std::ostream::put(char)+94>)
    at main.cc:30
30        const T& value () const { return value_; }
(gdb)
LinkedList<int>::remove (this=0x55555556aeb0,
    item_to_remove=@0x7fffffffe43c: 1) at main.cc:75
75            marker = 0;  // reset the marker
(gdb)
76            temp = marker;
(gdb)
77            marker = marker->next();
(gdb)
Node<int>::next (this=0x55555556b360) at main.cc:28
28        Node<T>* next () const { return next_; }
(gdb)

Program received signal SIGSEGV, Segmentation fault.
0x000055555555586c in Node<int>::next (this=0x0) at main.cc:28
28        Node<T>* next () const { return next_; }
(gdb)
```

4. Checkwork.

```
student@ubuntu:~/labtainer/labtainer-student$ checkwork
Results stored in directory: /home/student/labtainer_xfer/gdb-cpp
Labname gdb-cpp

Student              |    gdb_commands |
==================== | =============== |
B20DCAT098           |              17 |
What is automatically assessed for this lab:

        gdb_commands: How many gdb commands issued by the student
```