ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chương 9

Gỡ lỗi và kiểm thử

Tổng quan

Mục đích:

Giới thiệu về các cách tiếp cận khác nhau của việc gỡ lỗi và kiểm thử

Mục tiêu:

Nắm vững các kĩ thuật của việc gỡ lỗi và các phương pháp kiểm thử cũng như độ bao phủ của kiểm thử

Nội dung

1. Gỡ lỗi

2. Kiểm thử

1. Gỡ lỗi

1.1. Khái niệm

1.2. Phân loại lỗi

1.3. Quy trình gỡ lỗi

1.4. Lời khuyên khi gỡ lỗi

1.5. Gdb

1.1. Khái niệm

- Gỡ lỗi là quá trình định vị và gỡ bỏ các lỗi của chương trình
- Ước tính có tới 85% thời gian của việc gỡ lỗi là định vị ra nơi xảy ra lỗi và 15% là sửa chữa các lỗi này
- Nếu chương trình được thiết kế với cấu trúc tốt, được viết bằng phong cách lập trình tốt và áp dụng các kỹ thuật viết chương trình hiệu quả, bẫy lỗi thì chi phí cho việc gỡ rối sẽ được giảm thiểu.

1.2. Phân loại lỗi

- Có thể phân loại thành lỗi khi biên dịch (compile time), lỗi run-time.
- Lỗi khi biên dịch:
  - Các lỗi khiến chương trình bị sai cú pháp và không thể biên dịch được.
  - Ngoài ra cũng có thể bao gồm các cảnh báo của trình biên dịch như: biến cục bộ không được dùng hoặc gọi đến hàm chưa được khai báo (vẫn chạy được nếu dùng dynamic linking)
- Lỗi run-time: các lỗi chỉ phát hiện ra khi tiến hành chạy chương trình

1.2. Phân loại lỗi

- Các lỗi run-time thường gặp:
  - Truy cập phần tử ngoài mảng
  - Lỗi gán không hợp lệ: biến toàn cục vô ý bị chỉnh sửa, thao tác gán thay vì so sánh (`a = b` thay vì `a == b`)
  - Sử dụng các biến chưa được khởi tạo ban đầu
  - Các case thực hiện xuyên suốt cho nhau do thiếu break
  - Không giải phóng bộ nhớ
  - Viết nhầm điều kiện (x < 5 thay vì viết x > 5)
  - Xử lý file (quên không đóng file hoặc mở/đóng file liên tục)

1.3. Quy trình gỡ lỗi

Quy trình sẽ trải qua các bước sau:

1) Xác định/mô tả về lỗi
2) Thu thập dữ liệu về trường hợp xảy ra lỗi
3) Dự đoán hoặc định vị về nơi/thời điểm xảy ra lỗi
4) Chạy lại hoặc dùng gdb để kiểm tra dự đoán/định vị của mình
5) Nếu bước (4) phát hiện ra dự đoán/định vị của mình sai, quay lại bước (2)
6) Nếu bước (4) khẳng định dự đoán/định vị, tiến hành sửa lỗi

1.3. Quy trình gỡ lỗi (2)

Để có thể đưa ra các dự đoán/định vị, hãy cố gắng trả lời các câu hỏi sau:

1. Liệu có chú thích nào bị đóng không đúng cách (khiến một số câu lệnh cũng bị biến thành chú thích)
2. Liệu tất cả các biến đều được khởi tạo?
3. Liệu tất cả các vòng lặp đều kết thúc?
4. Liệu tất cả các tham số truyền vào cho các hàm đều hợp lệ?
5. Liệu các khối { } đều đặt đúng chỗ?

1.4. Lời khuyên khi gỡ lỗi

- Giữ lại một bản mô tả kĩ lưỡng về lỗi (người phát hiện lỗi, thời điểm, tình huống tái hiện lỗi, mức độ nguy hiểm, ai đã sửa, thời điểm sửa, phiên bản sửa)
- Giữ lại một bản sao chép mã nguồn trước khi cố gắng sửa lỗi. Như vậy có thể quay lại được phiên bản cũ nếu lỗi sửa mãi không được hoặc quá trình sửa lại gây thêm nhiều lỗi khác.
- Khi sửa xong phải kiểm thử tích hợp nhằm xác nhận phiên bản mới không ảnh hưởng đến mã nguồn cũ.

1.4. Lời khuyên khi gỡ lỗi (2)

- Cần ghi lại các chú thích để lý giải nguyên nhân sửa đổi của mình (Cả trong mã nguồn và trên github/gitlab/bất kỳ hệ quản lý mã nguồn nào)
- Hạn chế giả sử rằng lỗi là do máy tính bị lỗi phần cứng hoặc phần mềm hệ thống lỗi.
- Cố gắng tổng quát hóa quy luật/nguyên nhân xảy ra lỗi
- Có thể chèn thêm các câu lệnh printf để dò lỗi được nhanh hơn (thay vì debug từng dòng lệnh)

1.4. Lời khuyên khi gỡ lỗi (3)

- Có thể chèn thêm các câu lệnh `printf` để dò lỗi được nhanh hơn (thay vì debug từng dòng lệnh) (tiếp):
  - Chèn tại dòng đầu của hàm (để in ra tham số)
  - Chèn tại ngay trước lệnh `return` (để in ra kết quả)
  - Chèn tại phần đầu của vòng lặp

1.4. Lời khuyên khi gỡ lỗi (4)

- Thêm các điều khiển khi biên dịch
  - Chèn thêm các câu lệnh `printf` cũng có các phiền toái của nó
  - Giải pháp thay thế là thêm các điều khiển khi biên dịch
  - Điều khiển này sẽ khiến chỉ thực thi một số câu lệnh khi một (vài) biến môi trường đạt giá trị nào đó
  - Thường được kết hợp với các `MACRO`

1.4. Lời khuyên khi gỡ lỗi (5)

```
#include <stdio.h>
int foo(int);
int main( ){
      int n = 1;
      #ifdef TEST
            printf("Reached main( ) n = %d\n", n);
      #endif
}
```

1.4. Lời khuyên khi gỡ lỗi (6)

- Khi biên dịch, sẽ thêm tùy chọn tạo ra biến TEST trong khối lệnh gcc:

```
gcc -DTEST -o exam   examp.c
```

- Điều đó cũng tương tự: `#define TEST`
- Một cách khác để hỗ trợ gỡ lỗi là sử dụng hàm assert, đây là hàm có sẵn trong thư viện assert.h
- Nếu MACRO định nghĩa NDEBUG (dùng #define hoặc tùy chọn -D của trình biên dịch) thì các hàm assert bị bỏ qua không thực thi
- Chỉ dùng hàm assert này cho mục đích kiểm thử đơn vị hoặc gỡ lỗi

1.4. Lời khuyên khi gỡ lỗi (7)

- Hàm assert có thể truyền vào bất kỳ biểu thức C nào
- Nếu biểu thức có giá trị 0 khi thực thi, thì một thông báo lỗi sẽ hiện ra và việc thực thi của chương trình sẽ dừng lại.

```
#include <stdio.h>
#include <assert.h>
int main( ) {
      int x = 9, y = 9;
      assert(x = = (y + 1)) ;
      return 0 ;
}
```

1.5. gdb

- Công cụ nổi tiếng nhất để gỡ lỗi trong ngôn ngữ C/C++ là `gdb`
- Có thể tìm hiểu thêm thông tin về công cụ này: `man gdb` hoặc `gdb -help` hoặc `ginfo`
- Nhìn chung các công cụ gỡ lỗi đều có hỗ trợ tính năng:
  - Đặt điểm dừng (breakpoint) tại vị trí bất kỳ trong mã nguồn
  - Thực thi từng câu lệnh sau điểm dừng
  - Kiểm tra giá trị của các biến
  - Phân tích các lỗi liên quan đến bộ nhớ (core dump)

2. Kiểm thử

2.1. Khái niệm

2.2. Phương pháp kiểm thử

2.3. Độ bao phủ kiểm thử

2.4. Các phương pháp đo

2. Kiểm thử

- Khó có thể khẳng định 1 chương trình lớn có làm việc chuẩn hay không
- Khi xây dựng 1 chương trình lớn, 1 lập trình viên chuyên nghiệp sẽ dành thời gian cho việc viết test code không ít hơn thời gian dành cho viết bản thân chương trình
- Lập trình viên chuyên nghiệp là người có khả năng, kiến thức rộng về các kỹ thuật và chiến lược testing

2.1. Khái niệm

- Beizer: Việc thực hiện test là để **chứng minh tính đúng đắn giữa 1 phần tử và các đặc tả của nó.**
- Myers: Là quá trình thực hiện 1 chương trình **với mục đích tìm ra lỗi.**
- IEEE: Là quá **trình kiểm tra hay đánh giá 1 hệ thống hay 1 thành phần hệ thống** một cách thủ công hay tự động để kiểm chứng rằng nó **thỏa mãn những yêu cầu đặc thù** hoặc để **xác định sự khác biệt giữa kết quả mong đợi và kết quả thực tế**

2.2. Phương pháp kiểm thử

- **Black-Box**: Testing chỉ dựa trên việc phân tích các yêu cầu - requirements (unit/component specification, user documentation, v.v.). Còn được gọi là functional testing.
- **White-Box**: Testing dựa trên việc phân tích các logic bên trong - internal logic (design, code, v.v.). (Nhưng kết quả mong đợi vẫn đến từ requirements.) Còn được gọi là structural testing.

Kiểm thử hộp đen

- Black-box testing sử dụng mô tả bên ngoài của phần mềm để kiểm thử, bao gồm các đặc tả (specifications), yêu cầu (requirements) và thiết kế (design).
- Không có sự hiểu biết cấu trúc bên trong của phần mềm
- Các dạng đầu vào có dạng hàm hoặc không, hợp lệ và không không hợp lệ và biết trước đầu hợp lệ và không biết trước đầu ra
- Được sử dụng để kiểm thử phần mềm tại mức: mô đun, tích hợp, hàm, hệ thống và chấp nhận.

Kiểm thử hộp đen (2)

- Ưu điểm của kiểm thử hộp đen là khả năng đơn giản hoá kiểm thử tại các mức độ được đánh giá là khó kiểm thử
- Nhược điểm là khó đánh giá còn bộ giá trị nào chưa được kiểm thử hay không

Kiểm thử hộp trắng

- Còn được gọi là clear box testing, glass box testing, transparent box testing, hoặc structural testing, thường thiết kế các trường hợp kiểm thử dựa vào cấu trúc bên trong của phần mềm.
- WBT đòi hỏi kĩ thuật lập trình am hiểu cấu trúc bên trong của phần mềm (các đường, luồng dữ liệu, chức năng, kết quả)
- Phương thức: Chọn các đầu vào và xem các đầu ra

Kiểm thử hộp trắng (2)

- ***Đặc điểm***
  - Phụ thuộc vào các cài đặt hiện tại của hệ thống và của phần mềm, nếu có sự thay đổi thì các bài test cũng cần thay đổi theo.
  - Được ứng dụng trong các kiểm tra ở cấp độ mô đun (điển hình), tích hợp (có khả năng) và hệ thống của quá trình test phần mềm.

Kiểm thử hộp xám

- Là sự kết hợp của kiểm thử hộp đen và kiểm thử hộp trắng khi mà người kiểm thử biết được một phần cấu trúc bên trong của phần mềm
- Khác với kiểm thử hộp đen
  - Là dạng kiểm thử tốt và có sự kết hợp các kĩ thuật của cả kiểm thử hộp đen và hộp trắng

Những ai cần biết đến kiểm thử

- Programmers
  - *White-box testing*
    - *Ưu điểm: Người triển khai nắm rõ mọi luồng dữ liệu*
    - *Nhược: Bị ảnh hưởng bởi cách thức code được thiết kế/viết*
  - Quality Assurance (QA) engineers
    - *Black-box testing*
    - *Pro: Không có khái niệm về implementation*
    - *Con: Không muốn test mọi logical paths*

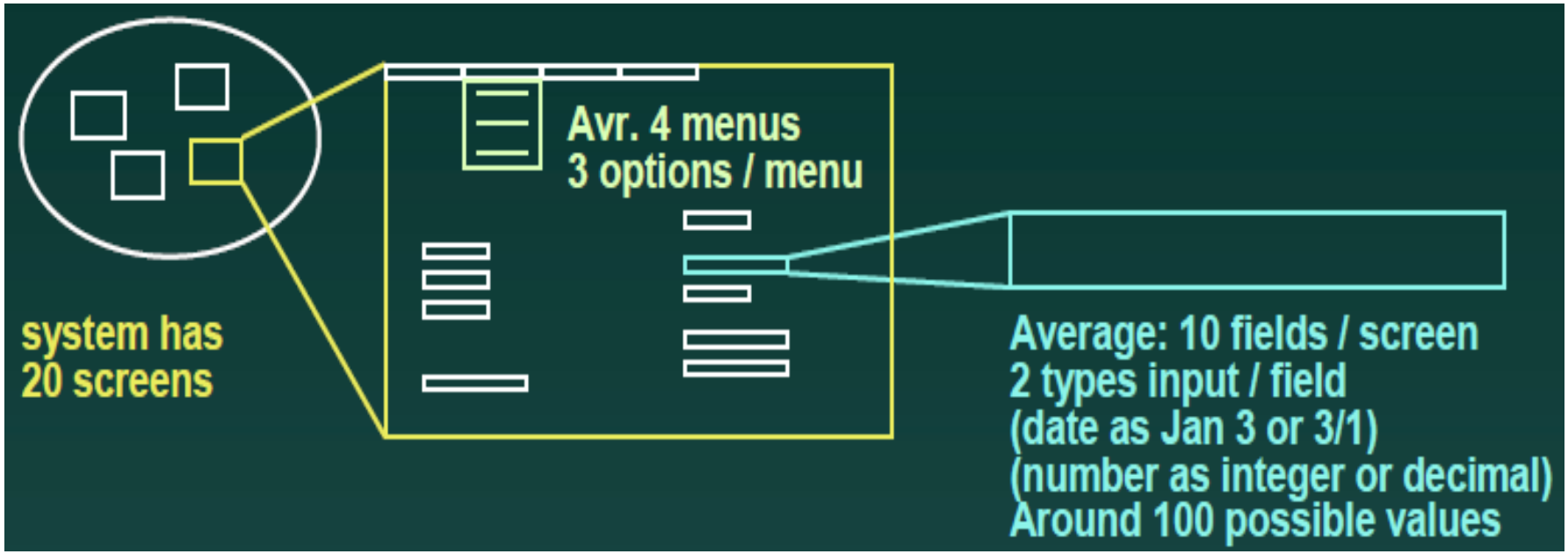
Các mức độ kiểm thử

- Unit: kiểm thử các công việc nhỏ nhất của lập trình viên để có thể lập kế hoạch và theo dõi hợp lý (vd : function, procedure, module, object class,…)
- Component: kiểm thử tập hợp các units tạo thành một thành phần (vd: program, package, task, interacting object classes,…)
- Product: kiểm thử các thành phần tạo thành một sản phẩm (subsystem, application,…)

Các mức độ kiểm thử (2)

- System: kiểm thử toàn bộ hệ thống
- Việc kiểm thử thường:
  - Bắt đầu = functional (black-box) tests,
  - Rồi thêm = structural (white-box) tests, và
  - Tiến hành từ unit level đến system level với một hoặc một vài bước tích hợp

Kiểm thử tất cả mọi thứ?



Chi phí cho 'exhaustive' testing:

20 x 4 x 3 x 10 x 2 x 100 = 480,000 tests

Nếu 1 giây cho 1 test, 8000 phút, 133 giờ, 17.7 ngày

(chưa kể nhầm lẫn hoặc test đi test lại)

nếu 10 secs = 34 wks, 1 min = 4 yrs, 10 min = 40 yrs

Bao nhiêu testing là đủ?

- Không bao giờ đủ!
- Khi bạn thực hiện những test mà bạn đã lên kế hoạch
- Khi khách hàng/người sử dụng thấy thỏa mãn
- Khi bạn đã chứng minh được/tin tưởng rằng hệ thống hoạt động đúng, chính xác
- Phụ thuộc vào risks for your system

Bao nhiêu testing là đủ? (2)

- Dùng RISK để xác định:
  - Cái gì phải test trước
  - Cái gì phải test nhiều
  - Mỗi phần tử cần test kỹ như thế nào? Tức là đâu là trọng tâm
  - Cái gì không cần test (tại thời điểm này…)

2.3. Độ bao phủ kiểm thử

- Độ bao phủ kiểm thử (test coverage) là một độ đo xác định xem các trường hợp kiểm thử có thực sự bao trùm mã ứng dụng hay không, nói cách khác có bao nhiêu phần trăm dòng mã được thực hiện khi chạy các trường hợp kiểm thử đó.
- Áp dụng cho kiểm thử hộp trắng.

2.3. Độ bao phủ kiểm thử (2)

- Khi ta đo đạc các giá trị độ bao phủ trong một tập các lần thực thi các trường hợp kiểm thử:
  - Nếu sớm đạt giá trị 100% thì nghĩa là thừa các trường hợp kiểm thử
  - Nếu toàn bộ các lần thực thi không đạt 100% nghĩa là cần bổ sung các trường hợp kiểm thử mới
  - Nếu bổ sung mà mãi không đạt được giá trị 100% nghĩa là mã nguồn có những nhánh không thể thực thi được.

2.4. Các phương pháp đo

a) Statement Coverage:

Statement Coverage đảm bảo rằng tất cả các dòng lệnh trong mã nguồn đã được kiểm tra ít nhất một lần.

```
if(A)
  F1();
F2();
```

Test Case: `A = 1`
Độ bao phủ Statement Coverage đạt 100%

```
int* ptr = NULL;
if (B)
   ptr = &variable;
*ptr = 10;
```

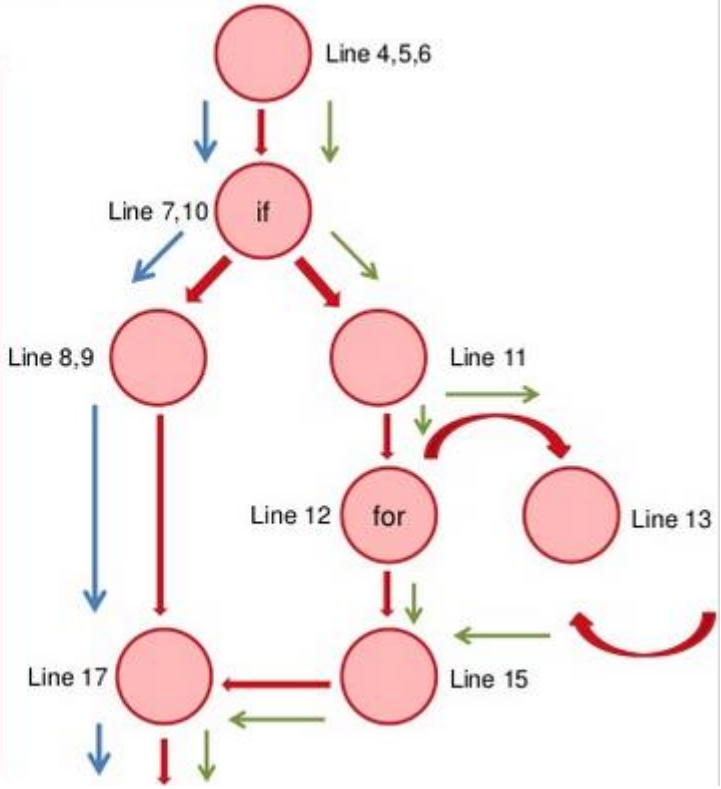
Test Case:
`B = 1`
Độ bao phủ Statement Coverage đạt 100%

2.4. Các phương pháp đo (2)

Ở ví dụ này, để Statement Coverage đạt 100%, chúng ta cần thực thi hai test case với n < 0 (màu xanh dương) và n > 0 (màu xanh lá)

Statement Coverage

```
1 #include <stdio.h>
2 main ()
3 {
4     int i, n, f;
5     printf ("n = ");
6     scanf ("%d", &n);
7     if (n < 0) {
8         printf ("Invalid: %d\n", n);
9         n= -1;
10    } else {
11        f = 1;
12        for (i = 1; i <= n; i++) {
13            f *= i;
14        }
15        printf("d! = %d\n", n, f);
16    }
17    return n;
18 }
```



2.4. Các phương pháp đo (3)

b) Branch Coverage:

Branch Coverage đảm bảo rằng tất cả các nhánh chương trình trong mã nguồn đã được kiểm tra ít nhất một lần.

```
if(A)  F1();
else   F2();
if(B)  F3()
else   F4();
```

Test Case: `A = B = 1`
Và test case: `A = B = 0`
Sẽ giúp độ bao phủ đạt 100%

```
if (A && (B || F1()))
   F2();
else
   F3();
```

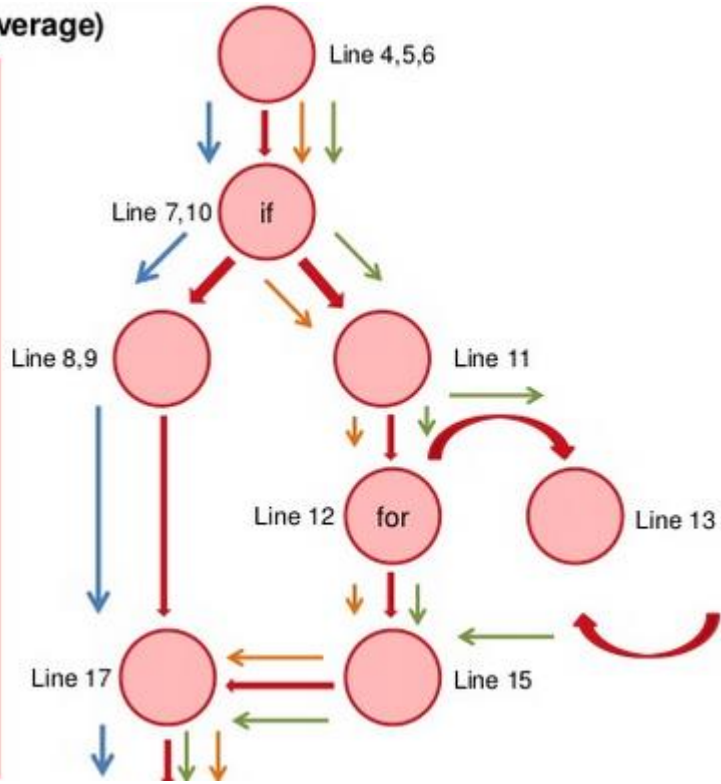
Test Case: `A = B = 1`
Và test case: `A = 0`
Sẽ giúp độ bao phủ đạt 100%
Nhưng độ bao phủ này không đảm bảo rằng hàm F1( ) sẽ được thực thi

2.4. Các phương pháp đo (4)

Ở ví dụ này, để Branch Coverage đạt 100%, chúng ta cần thực thi ba test case với n < 0 (màu xanh dương), n > 0 (màu xanh lá) và n = 0 (màu vàng)

Decision Coverage (branch coverage)

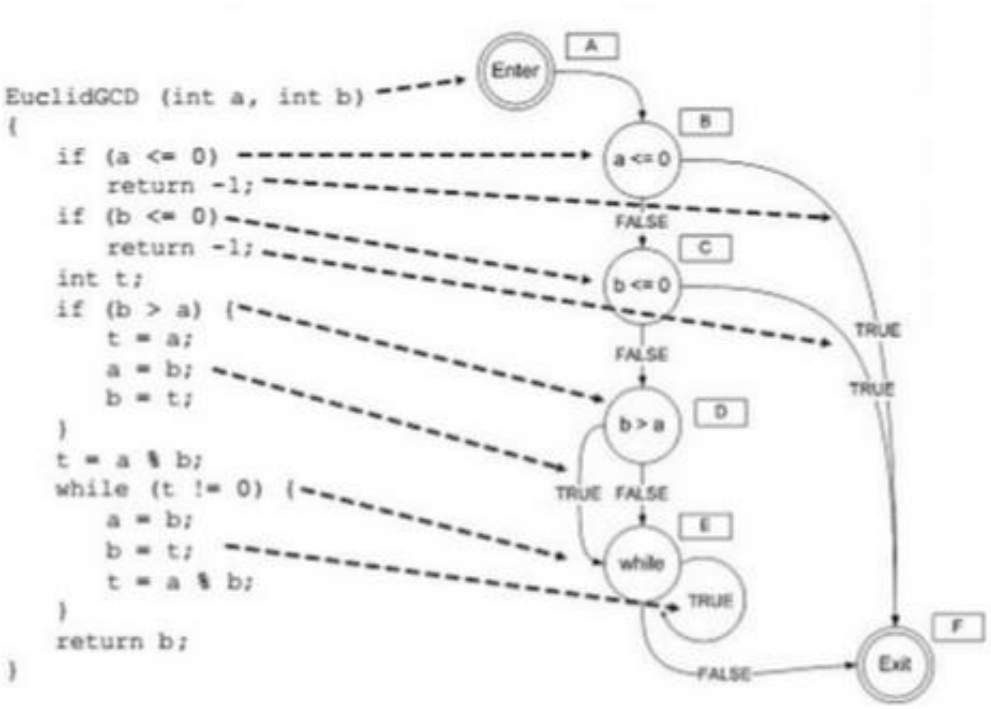
```
1 #include <stdio.h>
2 main ()
3 {
4     int i, n, f;
5     printf ("n = ");
6     scanf ("%d", &n);
7     if (n < 0) {
8         printf ("Invalid: %d\n", n);
9         n= -1;
10    } else {
11        f = 1;
12        for (i = 1; i <= n; i++) {
13            f *= i;
14        }
15        printf("d! = %d\n", n, f);
16    }
17    return n;
18 }
```



2.4. Các phương pháp đo (5)

c) Path Coverage:

Path Coverage đảm bảo rằng tất cả các đường chạy (là tổ hợp của các nhánh) chương trình trong mã nguồn đã được kiểm tra ít nhất một lần.



Với các bộ giá trị (a, b) nào sẽ khiến độ bao phủ đạt 100%???

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions!

soict.hust.edu.vn/ fb.com/groups/soict