DEBUGGING & TESTING

DEBUGGING 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.

Gỡ rối - debbuging

- Các chương trình đã viết có thể đã có nhiều lỗi ? – tại sao phần mềm lại phức tạp vây ?
- Sự phức tạp của Ct liên quan đến cách thức tương tác của các thành phần của ct đó, mà 1 phần mềm lại bao gồm nhiều thành phần và các tương tác giữa chúng
- Nhiều kỹ thuật làm giảm số lượng các thành phần tương tác :
  - Che giấu thông tin
  - Trừu tượng hóa …
- Có các kỹ thuật nhàm đảm bảo tính toàn vẹn thiết kế phần mềm
  - Documentation
  - Lập mô hình
  - Phân tích các yêu cầu
  - Kiểm tra hình thức
- Nhưng chưa có 1 kỹ thuật nào làm thay đổi cách thức xây dựng phần mềm => luôn xuất hiện lỗi khi test, phai loại bỏ = gỡ rối !

- Ngay LTV chuyên nghiệp cũng tốn nhiều thời gian cho gỡ rối !
- Luôn rút kinh nghiệm từ các lỗi trước đó
- Viết code và gây lỗi là điều bình thường – vấn đề là làm sao để không lặp lại
- LTV giỏi là người giỏi gỡ rối
- Gỡ rối không đơn giản, tốn thời gian => cần tránh gây ra lỗi. Các cách làm giảm thời gian gỡ rối là :
  - Thiết kế tốt
  - Phong cách LT tốt
  - Kiểm tra các ĐK biên
  - Kiểm tra các “khẳng định” – assertion và tính đúng đắn trong mã nguôn
  - Thiết kế giao tiếp tốt, giới hạn việc sử dụng dữ liệu toàn cục
  - Dùng các công cụ kiểm tra
- Phòng bệnh hơn chữa bệnh !!

- Động lực chính cho việc cải tiến các ngôn ngữ LT là cố gắng ngăn chặn các lỗi thông qua các đặc trưng ngôn ngữ như :
  - Kiểm tra các giới hạn biên của các chỉ số
  - Hạn chế không dùng con trỏ, bộ dọn dẹp, các kiểu dữ liệu chuỗi
  - Xác định kiểu nhập/xuất
  - Kiểm tra dữ liệu chặt chẽ.
- Mỗi ngôn ngữ cũng có những đặc tính dễ gây lỗi : lệnh goto, biến toàn cục, con trỏ trỏ tới vùng không xác định, chuyển kiểu tự động…
- LTV cần biết trước những đặc thù để tránh các lỗi tiềm ẩn, đồng thời cần kích hoạt mọi khả năng kiểm tra của trình biên dịch và quan tâm đến các cảnh báo
- Ví dụ : so sánh C,Pascal, VB …

Phân loại lỗi

- Có thể phân thành lỗi syntax, lỗi run-time và lỗi logic.
- Lỗi syntax:
  - 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
- Lỗi logic: chương trình vẫn chạy đúng nhưng do tư duy sai, thuật toán sai dẫn đến sai kết quả, là loại lỗi khó phát hiện nhất

Debugging Heuristic

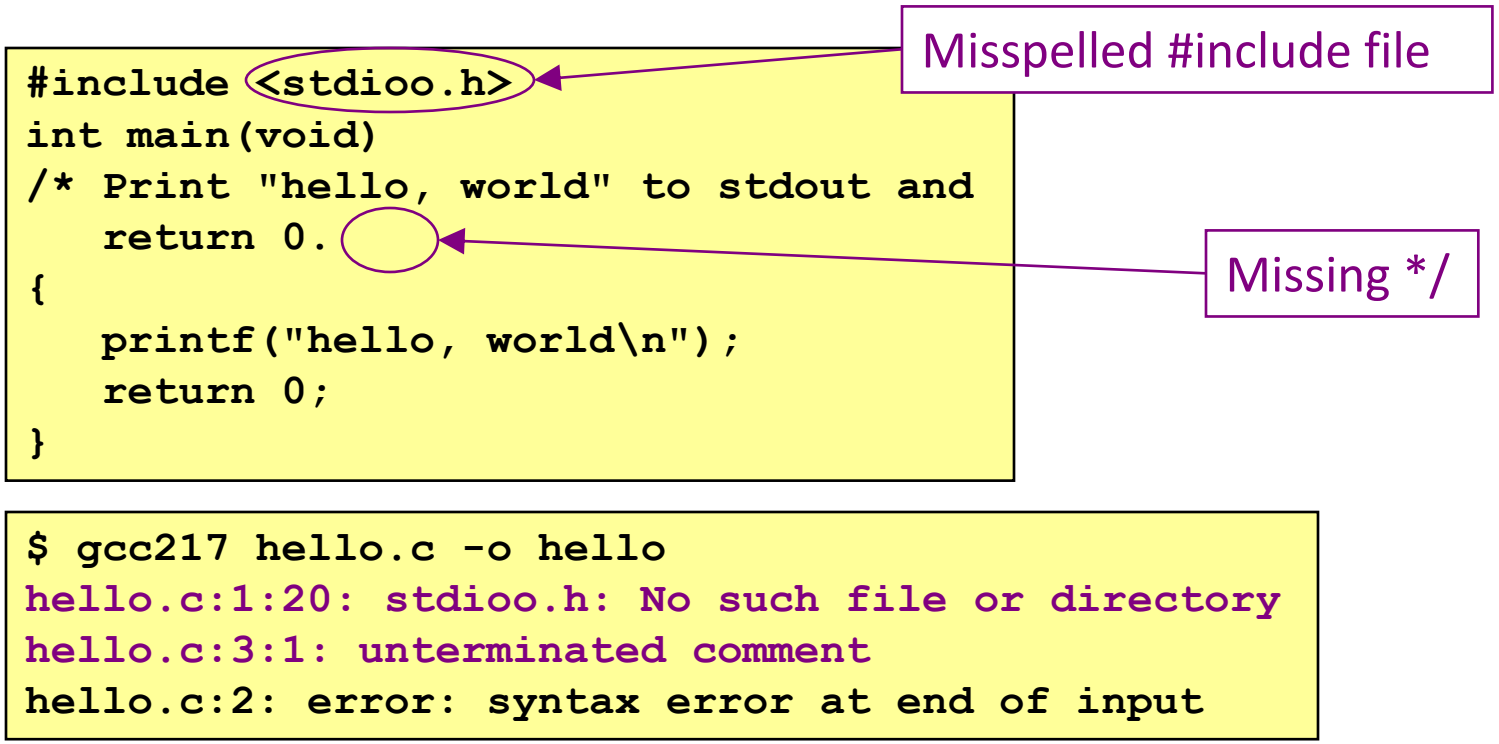
Debugging Heuristic	When Applicable
(1) Understand error messages	Build-time
(2) Think before writing	Run-time
(3) Look for familiar bugs	
(4) Divide and conquer	
(5) Add more internal tests	
(6) Display output	
(7) Use a debugger	
(8) Focus on recent changes	

Understand Error Messages

Gỡ rối khi **build-time** dễ hơn lúc **run-time**, khi và chỉ khi ta…

(1) Hiểu đc các thông báo lỗi!!!

- Một số là từ **preprocessor**



Understand Error Messages (tt)

(1) Hiểu đc các thông báo lỗi!!!

- Một số là từ **compiler**

```
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0. */
{
   printf("hello, world\n")
   retun 0;
}
```

Misspelled keyword

```
$ gcc217 hello.c -o hello
hello.c: In function `main':
hello.c:7: error: `retun' undeclared (first use in this function)
hello.c:7: error: (Each undeclared identifier is reported only once
hello.c:7: error: for each function it appears in.)
hello.c:7: error: syntax error before numeric constant
```

Understand Error Messages (tt)

(1) Hiểu đc các thông báo lỗi!!!

- Một số là từ **linker**

```
#include <stdio.h>
int main(void)
/* Print "hello, world" to stdout and
   return 0. */
{
   prinf("hello, world\n")
   return 0;
}
```

Misspelled function name

Compiler **warning** (not **error**): prinf() is called before declared

Linker error: Cannot find definition of prinf()

```
$ gcc217 hello.c -o hello
hello.c: In function `main':
hello.c:6: warning: implicit declaration of function `prinf'
/tmp/cc43ebjk.o(.text+0x25): In function `main':
: undefined reference to `prinf'
collect2: ld returned 1 exit status
```

Các phương pháp gỡ rối

- **Trình gỡ rối :**
  - IDE : kết hợp soạn thảo,biên dịch, gỡ rối …
  - Các trình gỡ rối với giao diện đò họa cho phép chạy chương trình từng bước qua từng lệnh hoặc từng hàm, dừng ở những dòng lệnh đặc biệt hay khi xuất hiện những đk đặc biệt, bên canh đó có các công cụ cho phép định dạng và hiển thị giá trị các biến, biểu thức
  - Trình gỡ rối có thể đc kích hoạt trực tiếp khi có lỗi.
  - Thường để tìm ra lỗi , ta phải xem xét thứ tự các hàm đã đc kích hoạt (theo vết) và hiển thị các giá trị các biến liên quan
  - Nếu vẫn không phát hiện đc lỗi : dùng các BreakPoint hoạc chạy từng bước – step by step
  - Có nhiều công cụ gỡ rối mạnh và hiệu quả, tại sao ta vẫn mất nhiều thời gian và trí lực để gỡ rối ?
  - Nhiều khi các công cụ không thể giúp dễ ràng tìm lỗi, nếu đưa ra 1 câu hỏi sai, trình gỡ rối sẽ cho 1 câu trả lời, nhưng ta có thể không biết là nó đang bị sai

Các phương pháp gỡ rối

- **Có đầu mối , phát hiện dễ ràng :**
  - Khi có lỗi, ta thường đổ cho trình dịch, thư viện hay bất cứ nguyên nhân nào khác …tuy nhiên, cuối cùng thì lỗi vẫn thuộc về CT
  - Rất may là hầu hết các lỗi thường đơn giản và dễ tìm. Hãy khảo sát các đầu mối của việc xuất ra kq có lỗi và cố gắng suy ra nguyên nhân gây ra nó
  - Khi có đc 1 số thông tin về lỗi và nơi xảy ra lỗi, hãy tạm dừng để ngẫm nghĩ xem lỗi xảy ra ntn.
  - Suy luận ngược trở lại trạng thái của CT bị hỏng để xđ nguyên nhân gây ra lỗi
  - Gỡ rối liên quan đến việc lập luận lùi, giống như tìm kiếm các bí mật của 1 vụ án. 1 số vđề không thể xảy ra và chỉ có những thông tin xác thực mới đáng tin cậy. => phải đi ngược từ kết quả để khám phá nguyên nhân, khi có lời giải thích đầy đủ, ta sẽ biết đc vấn đề cần sửa và có thể phát hiện ra 1 số vđề khác

Các phương pháp gỡ rối

- **Tìm các lỗi tương tự :**
  - Khi gặp vđề, hãy liên tưởng đến những trường hợp tương tự đã gặp
  - Vd1 : int n; scanf(“%d”,n); ?
  - Vd2 : int n=1; double d=PI;

    printf(“%d %f \n”,d,n); ??
  - Không khởi tạo biến (với C) cũng sẽ gây ra những lỗi khó lường.

Các phương pháp gỡ rối

- **Kiểm tra sự thay đổi mới nhất**
  - Lỗi thường xảy ra ở những đoạn CT mới đc bổ xung
  - Nếu phiên bản cũ OK, phiên bản mới có lỗi => lỗi chắc chắn nằm ở những đoạn CT mới
  - Lưu ý, khi sửa đổi, nâng cấp : hãy giữ lại phiên bản cũ – đơn giản là comment lại đoạn mã cũ
  - Đặc biệt, với các hệ thống lớn, làm việc nhóm thì việc sử dụng các hệ thống quản lý phiên bản mã nguồn và các cơ chế lưu lại quá trình sửa đổi là vô cùng hữu ích (source safe)

Các phương pháp gỡ rối

- **Tránh mắc cùng 1 lỗi 2 lần :** Sau khi sửa 1 lỗi, hãy suy nghĩ xem có lỗi tương tự ở nơi nào khác không. VD :

```
for (i=1;i<argc;i++) {
  if (argv[i][0] !='-')
        break;
    switch (argv[i][1]) {
        case 'o' :  /* tên tệp ouput   "-oData" */
                outname = argv[i]; break;
        case 'f' :   /* Giá trị biên từ "-f100" */
            from = atoi(argv[i]); break;
        case 't' :   /*  đến "-t500" */
                to = atoi(argv[i]); break;
    }
}
```

Tệp tin sai, vì luôn có –o ở trước tên => outname= &agrv[i][2];

Tương tự ?

Chú ý : nếu đơn giản có thể viết code khi ngủ thì cũng đừng ngủ gật khi viết code

Các phương pháp gỡ rối

- **Gỡ rối ngay khi gặp**
  - **Khi phát hiện lỗi, hãy sửa ngay, đừng để sau mới sửa, vì có thể lỗi không xuất hiện lại (do tình huống)**
  - **Cũng đừng quá vội vàng, không suy nghĩ chín chắn, kỹ càng, vì có thể việc sửa chữa này ảnh hưởng tới các tình huống khác**

Các phương pháp gỡ rối

- **Đọc trước khi gõ vào**
  - **Đừng vội vàng, khi không rõ điều gì thực sự gây ra lỗi và sửa không đúng chỗ sẽ có nguy cơ gây ra lỗi khác**
  - **Có thể viết đoạn code gây lỗi ra giấy=> tạo cách nhìn khác, và tạo cơ hội để nghĩ suy**
  - **Đừng miên man chép cả đoạn không có nguy cơ gây lỗi, hoặc in toàn bộ code ra giấy in => phá vỡ cây cấu trúc**

Các phương pháp gỡ rối

- **Giải thích cho người khác về đoạn code**
  - **Tạo đk để ngẫm nghĩ**
  - **Thậm chí có thể giải thích cho người không phải LTV**
  - **Extrem programming : làm việc theo cặp, pair programming, người này LT, người kia kiểm tra, và ngược lại**
  - **Khi gặp vấn đề, khó khăn, chậm tiến độ, lập tức thay đổi công việc => rút ra khỏi luồng quán tính sai lâm …**

Các phương pháp gỡ rối

- **Làm cho lỗi xuất hiện lại**
  - **Cố gắng làm cho lỗi có thể xuất hiện lại khi cần**
  - **Nếu không đc, thì thử tìm nguyên nhân tại sao lại không đc**
- **Chia để trị**
  - **Thu hẹp phạm vi**
  - **Tập trung vào dữ liệu gây lỗi**

Các phương pháp gỡ rối

- **Hiển thị kết quả để định vị khu vực gây lỗi**
  - **Thêm vào các dòng lệnh in giá trị các biến liên quan, hoặc đơn giản xác định tiến trình thực hiện : "đến đây 1" …**
- **Viết mã tự kiểm tra**
  - **Viết thêm 1 hàm để kiểm tra, gắn vào trước và sau đoạn có nguy cơ, comment lại sau khi đã xử lý lỗi**
- **Tạo log file**
- **Lưu vết**
  - **Giúp ghi nhớ đc các vấn đề đã xảy ra, và giải quyết các vđề tương tự sau này, cũng như khi chuyển giao CT cho người khác..**

Hiển thị KQ ..

In các giá trị tại các điểm nhạy cảm

- Poor:

```
printf("%d", keyvariable);
```

stdout is buffered; CT có thể có lỗi trước khi hiện ra output

- Maybe better:

```
printf("%d\n", keyvariable);
```

In '\n' sẽ xóa bộ nhớ đệm stdout , nhưng sẽ không xóa khi in ra file

- Better:

```
printf("%d", keyvariable);
fflush(stdout);
```

Gọi fflush() để làm sạch buffer 1 cách tường minh

Hiển thị KQ (cont.)

- Maybe even better:

```
fprintf(stderr, "%d", keyvariable);
```

In debugging output ra **stderr**; debugging output có thể tách biệt với các in ân của CT

- Maybe better still:

```
FILE *fp = fopen("logfile", "w");
…
fprintf(fp, "%d", keyvariable);
fflush(fp);
```

Ngoài ra: stderr không dùng buffer

Ghi ra 1 a log file

Các phương pháp gỡ rối

- **Phương sách cuối cùng**
  - **Dùng 1 trình gỡ rối để chạy từng bước**
  - **Nhiều khi vấn đề tưởng quá đơn giản nhưng lại không phát hiện được, ví dụ các toán tử so sánh trong pascal va VB có độ ưu tiên ngang nhau, nhưng với C ?**

    (== và != nhỏ hơn <,<=,>,>= !)
  - **Thứ tự các đối số của lời gọi hàm : ví dụ : strcpy(s1,s2)**
  - **Thứ tự thực hiện các phép toán**

    int m[6]={1,2,3,4,5,6}, *p,*q;

    p=m; q=p+2; *p++ =*q++; *p=*q; ???
  - **Lỗi loại này khó tìm vì bản thân ý nghĩ của ta vạch ra 1 hướng suy nghĩ sai lệch : coi điều không đúng là đúng**
  - **Đôi khi lỗi là do nguyên nhân khách quan : Trình biên dịch, thư viện hay hệ điều hành, hoặc lỗi phần cứng : 1994 lỗi xử lý dấu chấm động trong bộ xử lý Pentium**

- **Các lỗi xuất hiện thất thường :**
  - **Khó giải quyết**
  - **Thường gán cho lỗi của máy tính, hệ điều hành …**
  - **Thực ra là do thông tin của chính CT : không phải do thuật toán, mà do thông tin bị thay đổi qua mỗi lần chạy**
  - **Các biến đã đc khởi tạo hết chưa ?**
  - **Lỗi cấp phát bộ nhớ ? Vd :**

```
char *vd( char *s) {
        char m[101];
        strncpy(m,s,100);
        return m;
}
```

  - **Giải phóng bộ nhớ động ?**

```
for (p=listp; p!=NULL; p=p->next) free(p) ;      ???
```

Tóm lại

- Gỡ rối là 1 nghệ thuật mà ta phải luyện tập thường xuyên
- Nhưng đó là nghệ thuật mà ta không mê đc
- Mã nguồn viết tốt có ít lỗi hơn và dễ tìm hơn
- Đầu tiên phải nghĩ đến nguồn gốc sinh ra lỗi
- Hãy suy nghĩ kỹ càng, có hệ thống để định vị khu vực gây lỗi
- Không gì bằng học từ chính lỗi của mình – điều này càng đúng đối với LTV

Thêm – Những lỗi thường gặp với C, C++

1. Array as a parameter handled improperly – Tham số mảng đc xử lý không đúng cách
2. Array index out of bounds – Vượt ra ngoài phạm vi chỉ số mảng
3. Call-by-value used instead of call-by reference for function parameters to be modified – Gọi theo giá trị, thay vì gọi theo tham chiếu cho hàm để sửa
4. Comparison operators misused – Các toán tử so sánh bị dùng sai
5. Compound statement not used - Lệnh phức hợp không đc dùng
6. Dangling else - nhánh else khong hợp lệ
7. Division by zero attempted - Chia cho 0
8. Division using integers so quotient gets truncated – Dùng phép chia số nguyên nên phần thập phân bị cắt
9. Files not closed properly (buffer not flushed) - File không đc đóng phù hợp (buffer không dọn dẹp)
10. Infinite loop - lặp vô hạn
11. Global variables used – dùng biến tổng thể

12. IF-ELSE not used properly – dùng if-else không chuân
13. Left side of assignment not an L-value - phía trái phép gán không phải biến
14. Loop has no body – vòng lặp không có thân
15. Missing "&" or missing "const" with a call-by-reference

function parameter – thiếu dấu & hay từ khóa const với lời gọi tham số hàm theo tham chiếu

16. Missing bracket for body of function or compound statement – Thiếu cặp {} cho thân của hàm hay nhóm lệnh
17. Mission reference to namespace - Thiếu tham chiếu tới tên miền
18. Missing return statement in a value-returning function – Thiếu return
19. Missing semi-colon in simple statement, function prototypes, struct definitions or class definitions – thiếu dấu ; trong lệnh đơn …
20. Mismatched data types in expressions – kiểu dữ liệu không hợp
21. Operator precedence misunderstood - Hiểu sai thứ tự các phép toán

22. Off-by-one error in a loop – Thoát khỏi bởi 1 lỗi trong vòng lặp
23. Overused (overloaded) local variable names - Trùng tên biến cục bộ
24. Pointers not set properly or overwritten in error – Con trỏ không đc xác định đúng hoặc trỏ vào 1 vị trí không có
25. Return with value attempted in void function – trả về 1 giá trị trong 1 hàm void
26. Undeclared variable name – không khai báo biến
27. Un-initialized variables – Không khởi tạo giá trị
28. Unmatched parentheses – thiếu }
29. Un-terminated strings - xâu không kết thúc , thiếu /0
30. Using "=" when "= =" is intended or vice versa
31. Using "&" when "&&" is intended or vice versa
32. "while" used improperly instead of "if“ – while đc dùng thay vì if

Doan store sau tra ve soHD co gia tri bang sohd duoc tao ra gan day nhat +1 (de tao soHD ngam dinh cho hoa don moi).

SoHD nay khong bao gio vuot qua gioi han int, vi moi dot nguoi ta in 1 so gioi han hoa don (khoang 10 000 hoac 20 000 HD), danh so tu 1. Het dot cu, in dot moi, va so lai quay ve 1.Hoa don goc thi chi 1 loai, nhung 1 so duoc in tu may tinh, 1 so viet tay, vi vay duoc luu vao may tinh trong 2 bang (HoaDon va HoaDoanPhu)

Store va chuong trinh tao hoa don chay tot tu nam 2005, gan day, tu nhien xuat hien loi : timeout. Sau gan 1 tuan ra soat, tim hieu, LTV xac dinh loi la tai store nay. Tim hieu nguyen nhan va cach xu ly !

```
ALTER Proc [dbo].[GetSoHDTC_Integer] as
SELECT top 1 (a.SoHDTC+1) as SoHDTC
FROM
(SELECT  CAST(SoHoaDonTC AS int)as SoHDTC,ngay as ngaylap  FROM HoaDon
 union all
 SELECT CAST(SoHoaDonTC AS int)as SoHDTC ,ngaylap FROM HoaDonPhu
) a order by ngaylap desc
```

Phong benh hon chua benh : Cac ky thuat viet code chat luong

- Viet code co chat luong nham tranh gay ra loi
  1. **Think before coding**
  2. **Fix bugs immediately**
  3. **Test individual functional elements**
  4. **Test complete puzzle (Đừng tưởng đã test từng hàm là xong …)**
  5. **Write robust code components (đừng làm việc nửa chừng …)**

6. **Fail as early as possible (Phát hiện và ngăn chặn lỗi ngay từ đầu=> tiết kiệm ..)**
7. **Prefer strong typing over dynamic binding**
8. **Write self-explanatory code**
9. **Avoid sophisticated code**
10. Use good programming style
11. **Avoid magic constants** (đừng dùng các hằng trực tiếp trong code !)
12. **Keep related code close together**
13. **Build a house, not an empire** (K.I.S.S. principle (Keep it simple, stupid) and the YAGNI principle (You aren't gonna need it))

Bài tập Chốt

- Sưu tầm, tìm hiểu các vấn đề lý thú liên quan đến C/C++, Kỹ thuật lập trình…
- Tối thiểu 5 trang DOC,docx
- Thời hạn, trước ngày 18/6
- Gộp toàn bộ cac bài tập đã làm vào 1 file zip, ten file là tensv.zip hay .zar
- Nộp file .doc va file nen vào assigment bai tap tong hop trong muc general !!!

2. Kiểm thử

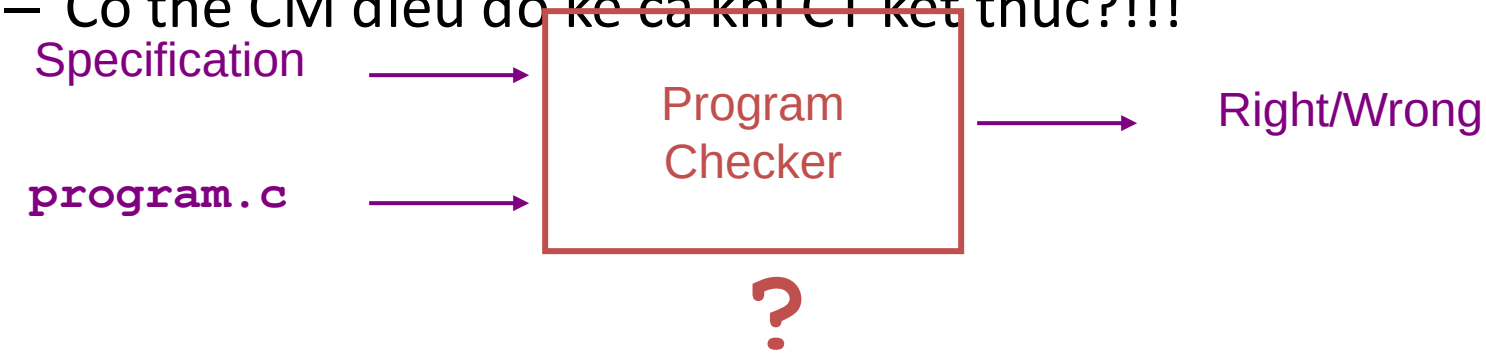
- Khó có thể khẳng định 1 chương trình lớn có làm việc chuẩn hay không
- Khi xây dựng 1 chương trình lớn, 1 lập trình viên chuyên nghiệp sẽ dành thời gian cho việc viết test code không ít hơn thời gian dành cho viết bản thân chương trình
- Lập trình viên chuyên nghiệp là người có khả năng, kiến thức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

Program Verification

- Lý tưởng: Chứng minh được rằng CT của ta là chính xác, đúng đắn
  - Có thể chứng minh các thuộc tính của CT?
  - Có thể CM điều đó kể cả khi CT kết thúc?!!!



Program Testing

- Hiện thực : Thuyết phục bản thân rằng CT có thể làm việc

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

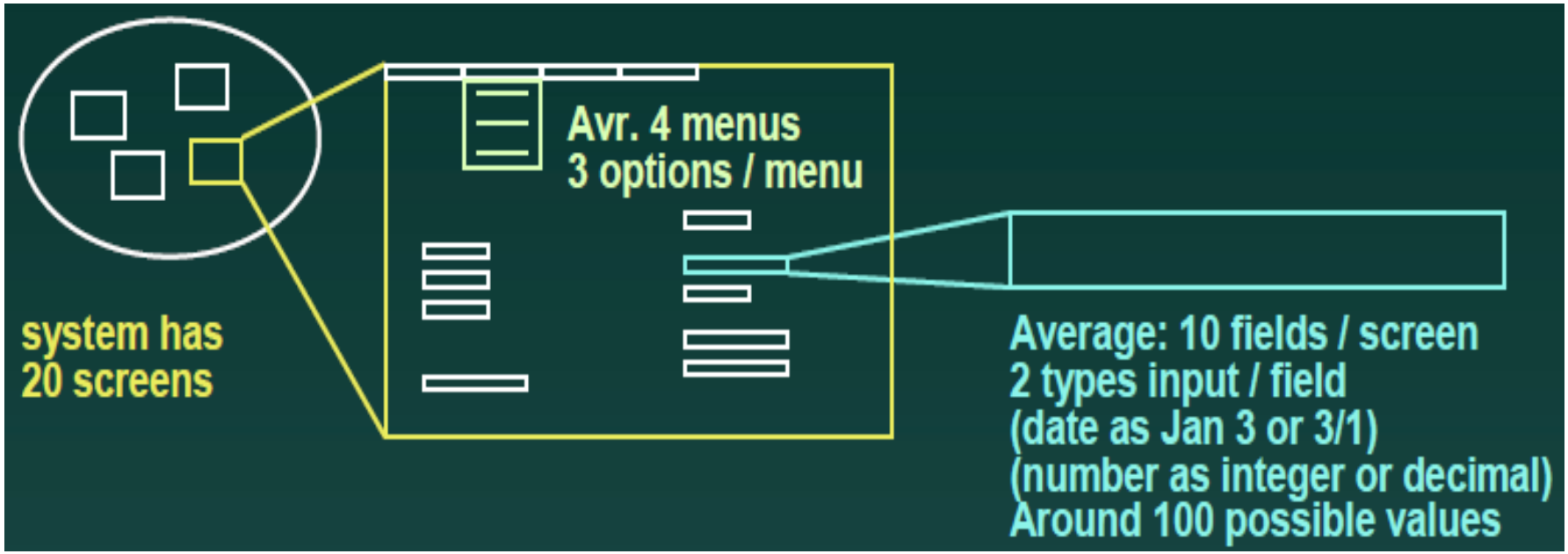
Các mức độ kiểm thử

- 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…)

Nguyên tắc quan trọng nhất

- **Ưu tiên tests**

Để,

Khi bạn kết thúc testing,

Bạn đã thực hiện việc test tốt nhất trong quãng thời gian có thể.

The testing paradox

- **Mục đích của testing: để tìm ra lỗi**
- **Tìm thấy lỗi làm mất (hủy hoại) sự tự tin - confidence**

=> Mục đích của testing: hủy hoại sự tự tin

- **Nhưng mục đích của testing: Xây dựng niềm tin, tự tin**

=> Cách tốt nhất để xây dựng niềm tin là :

Cố gắng hủy hoại nó

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
```

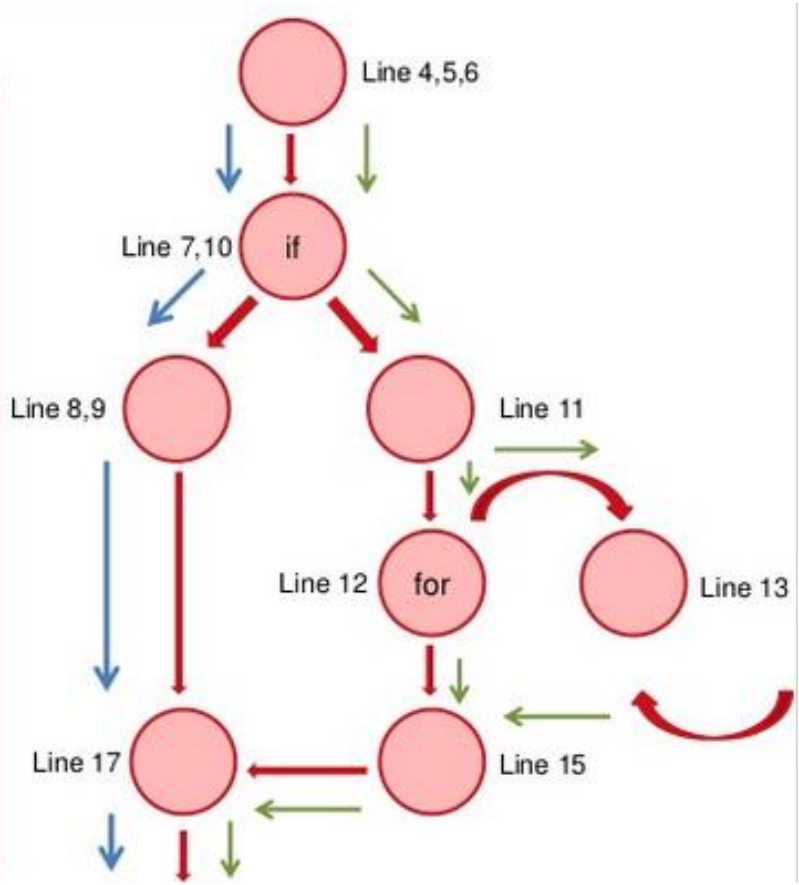
Độ bao phủ Statement Coverage đạt 100%

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%

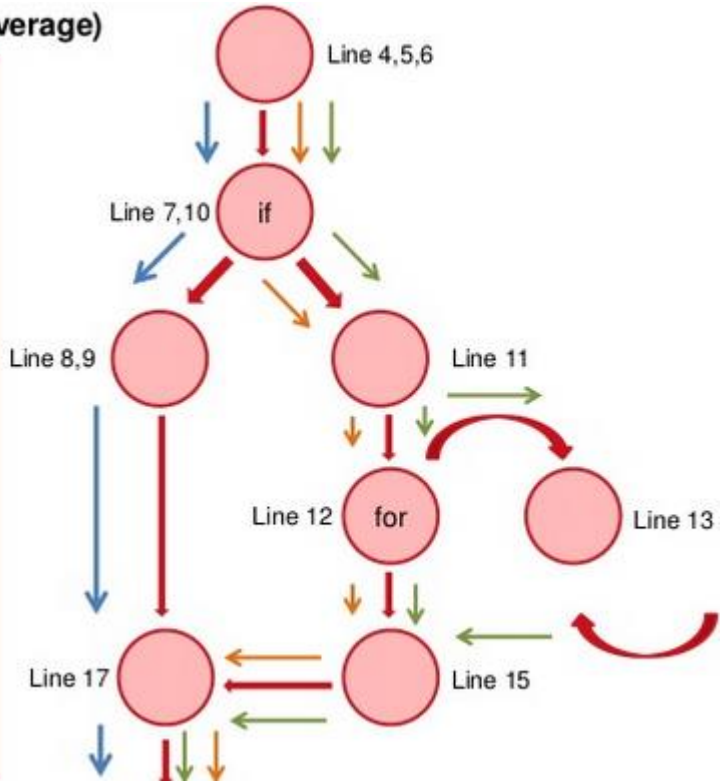
Nhưng độ bao phủ này không đảm bảo rằng hàm F1( ) sẽ được thực thi

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)

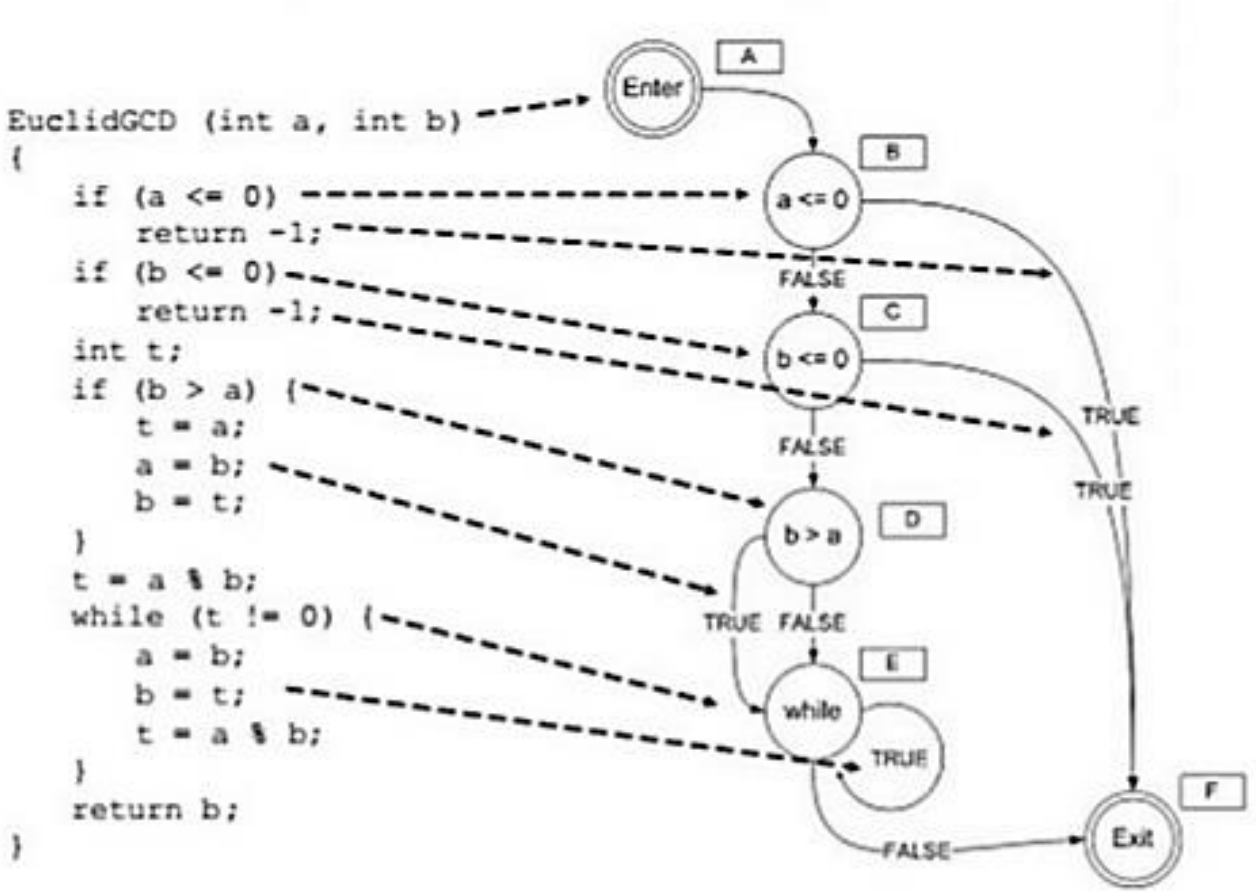
Decision Coverage (branch coverage)

```
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%?



Statement Testing Example

- VD:

```
if (condition1)
   statement1;
else
   statement2;
if (condition2)
   statement3;
else
   statement4;
…
```

Statement testing:

Phải chắc chắn các lệnh "if" và 4 lệnh trong cac nhánh phải đc thực hiện

- Đòi hỏi 2 tập dữ liệu;vd:
  - *condition1* là đúng và *condition2* là đúng
    - Thực hiện *statement1* và *statement3*
  - *condition1* là sai và *condition2* là sai
    - Thực hiện *statement2* và *statement4*

Path Testing

(3) Path testing

- "Kiểm tra để đáp ứng các tiêu chuẩn đảm bảo rằng mỗi đường dẫn logic xuyên suốt chương trình được kiểm tra. Thường thì đường dẫn xuyên suốt chương trình này được nhóm thành một tập hữu hạn các lớp . Một đường dẫn từ mỗi lớp sau đó được kiểm tra. "
  - Glossary of Computerized System and Software Development Terminology

- Khó hơn nhiều so với statement testing
  - Với các CT đơn giản, có thể liệt kê các nhánh đường dẫn xuyên suốt code
  - Ngược lại, bằng các đầu vào ngẫu nhiên tạo các đường dẫn theo ct

Path Testing Example

- VD:

```
if (condition1)
   statement1;
else
   statement2;
…
if (condition2)
   statement3;
else
   statement4;
…
```

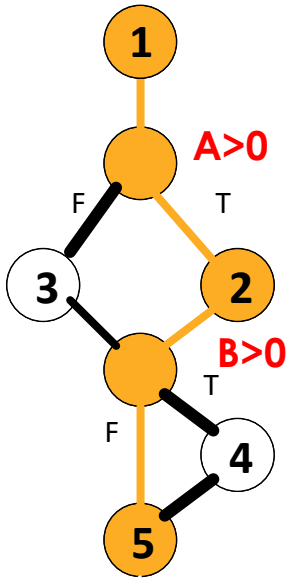
Path testing:

Cần đảm bảo tất cả các đường dẫn được thực hiện

- Đòi hỏi 4 tập dữ liệu:
  - *condition1* là true và *condition2* là true
  - *condition1* là true và *condition2* là false
  - *condition1* là false và *condition2* là true
  - *condition1* là false và *condition2* la false
- Chương trình thực tế => bùng nổ các tổ hợp!!!

Quan sat vd sau...

```
(1)  input(A,B)
     if (A>0)
(2)     Z = A;
     else
(3)     Z = 0;
     if (B>0)
(4)     Z = Z+B;
(5)  output(Z)
```



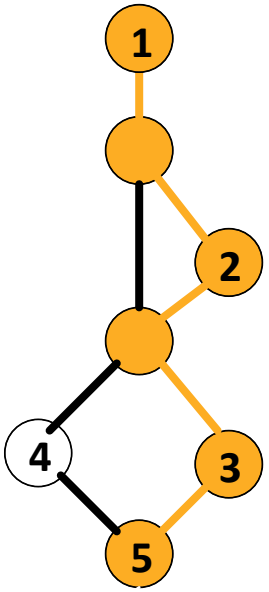
Điều kiện để nhánh **<1,2,5> thực hiện ?**

(A>0) && (B≤0)

VD khác...

```
(1)  input(A,B)
     if (A>B)
(2)   B = B*B;
       if (B<0)
(3)      Z = A;
     else
(4)      Z = B;

(5)  output(Z)
```



Nhánh **<1,2,3,5>?**

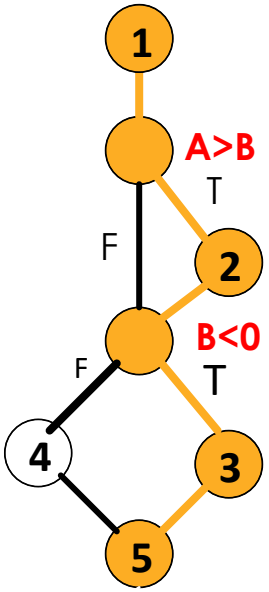
(A>B) && (B<0)

Vd tiếp…

```
(1)  input(A,B)
     if (A>B)
(2)     B = B*B;

     if (B<0)
(3)     Z = A;
     else
(4)     Z = B;

(5)  output(Z)
```



<1,2,3,5>?

$(A>B) \wedge (B<0)\ (B^2<0)$ = FALSE

Các chiến lược testing

- General testing strategies
  (1) Kiểm chứng tăng dần -Testing incrementally
  (2) So sanh các cài đặt -Comparing implementations
  (3) Kiểm chứng tự động - Automation
  (4) Bug-driven testing
  (5) Tiêm, gài lỗi - Fault injection

Testing Incrementally

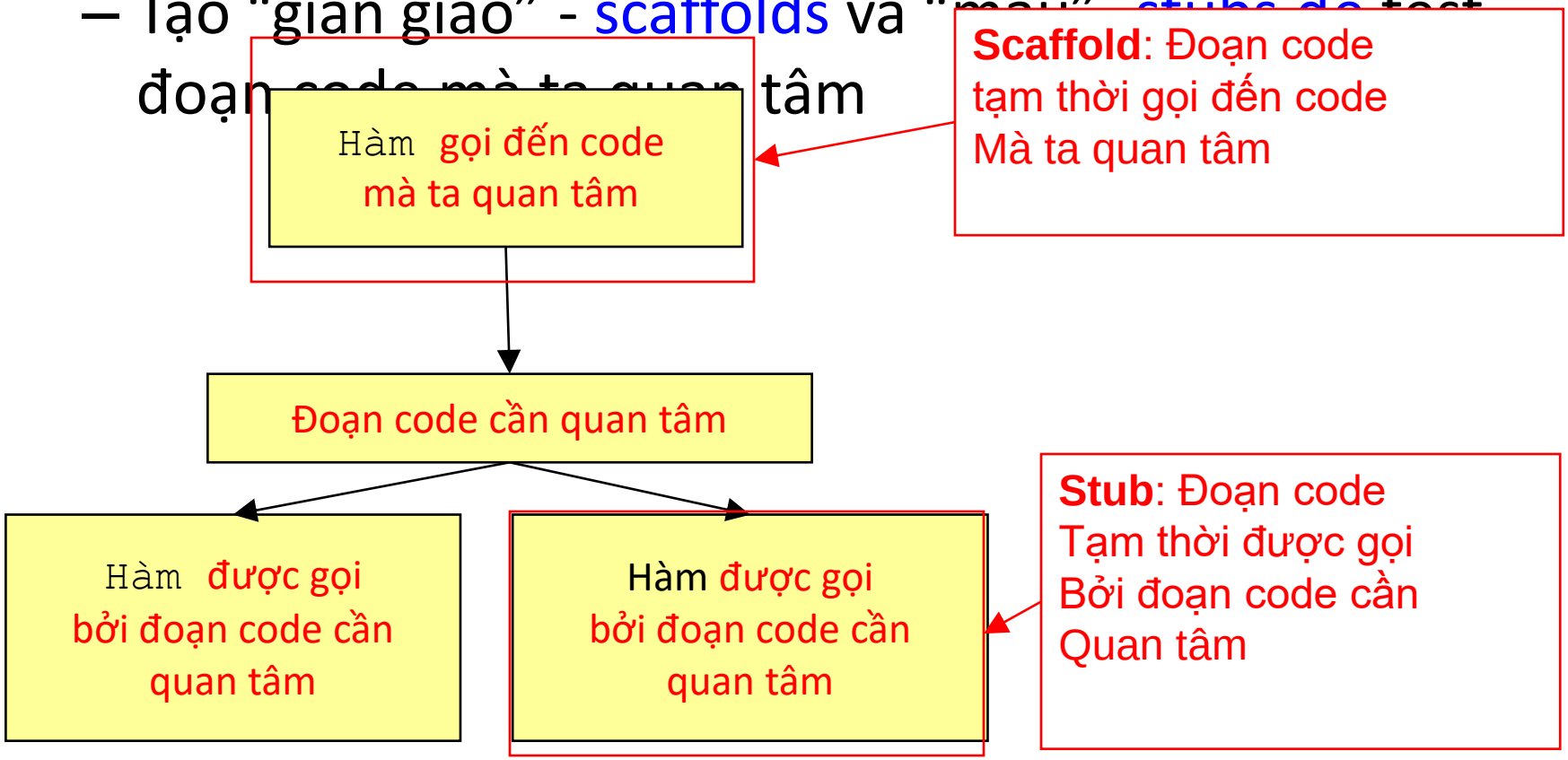
(1) Testing incrementally

- Test khi viết code
  - Thêm tests khi tạo 1 lựa chọn mới - new cases
  - Test phần đơn giản trước phần phức tạp
  - Test units (tức là từng module riêng lẻ) trước khi testing toàn hệ thống
- Thực hiện regression testing – kiểm thử hồi quy
  - Xử lý đc 1 lỗi thường tạo ra những lỗi mới trong 1 he thống lớn, vì vậy …
  - Phải đảm bảo chắc chắn hệ thống không "thoái lui" kiểu như chức năng trước kia đang làm việc giờ bị broken, nên…
  - Test mọi khả năng để so sanh phiên bản mới với phiên bản cũ

Testing Incrementally (cont.)

(1) Testing incrementally (cont.)

– Tạo "giàn giáo" - scaffolds và "mẫu" stubs để test đoạn code mà ta quan tâm



So sánh các cài đặt

(2) Compare implementations

- Hãy chắc chắn rằng các triển khai độc lập hành xử như nhau
- Example: So sánh hành vi của CT mà bạn dịch (TB C++3.0) với GCC
- Example: So sánh hành vi của các hàm bạn tạo trong str.h với các hàm trong thư viện string.h
- Đôi khi 1 kq có thể đc tính = 2 cách khác nhau, 1 bài toán có thể giải = 2 phương pháp, thuật toán # nhau. Ta có thể xd cả 2 CT, nếu chúng có cùng KQ thì có thể khẳng định cả 2 cùng đúng, còn kq khác nhau thì ít nhất 1 trong 2 ct bị sai

Kiểm chứng tự động - Automation

(3) Automation

- Là quá trình xử lý 1 cách tự động các bước thực hiện test case = 1 công cụ nhằm rut ngắn thời gian kiểm thử.
- Ba quá trình kiểm chúng bao gồm :
  - Thực hiện kiểm chứng nhiều lần
  - Dùng nhiều bộ dữ liệu nhập
  - Nhiều lần so sánh dữ liệu xuất
- vì vậy cần kiểm chứng = chương trình để : tránh mệt mỏi, giảm sự bất cẩn …

- Tạo testing code
  - Viết 1 bộ kiểm chứng để kiểm tra toàn bộ chương trình mỗi khi có sự thay đổi, sau khi biên dịch thành công

- Cần biết cái gì được chờ đợi
  - Tạo ra các đầu ra, sao cho dễ dàng nhận biết là đúng hay sai

- Tự động kiểm chứng có thể cung cấp:
  - **Tốt hơn nhiều** so với kiểm chứng thủ công

- Tự động hóa kiểm chứng lùi
  - Tuần tự kiểm chứng so sánh các phiên bản mới với những phiên bản cũ tương ứng.
  - Mục đích : đảm bảo việc sửa lỗi sẽ không làm ảnh hưởng những phần khác trừ khi chúng ta muốn
  - 1 số hệ thống có công cụ trợ giúp kiểm chứng tự động :
    - Ngôn ngữ scripts : cho phép viết các đoạn script để test tuần tự
    - Unix : có các thao tác trên tệp tin như cmp và diff để so sanh dữ liệu xuất, sort sắp xếp các phần tử, grep để kiểm chứng dữ liệu xuất, wc, sum va freq để tổng kết dữ liệu xuất
  - Khi kiểm chứng lùi, cần đảm bảo phiên bản cũ là đúng, nếu sai thì rất khó xác định và kết quả sẽ không chính xác
  - Cần phải kiểm tra chính việc kiểm chứng lùi 1 cách định kỳ để đảm bảo nó vẫn hợp lệ

- Dùng các công cụ test
  - QuickTest professional
  - TestMaker
  - Rational Robot
  - Jtest
  - Nunit
  - Selenium
  - ….