Trịnh Thành Trung (ThS)
trungtt@soict.hust.edu.vn

Bài 9
GỠ LỖI VÀ KIỂM THỬ

Nội dung

1. Gỡ lỗi
2. Kiểm thử

1.
Gỡ lỗi

Debug

Gỡ rối
Debug

- Gỡ rối là gì?
  - *Khi chương trình bị lỗi, gỡ rối là các công việc cần làm để làm cho chương trình dịch thông, chạy thông*
  - *Thật không may, gỡ rối luôn là thao tác phải làm khi lập trình, thao tác này rất tốn kém*
- Cách tốt nhất vẫn là phòng ngừa
  - *Khi bắt đầu gỡ rối chương trình, bạn đã biết là chương trình không chạy.*
  - *Nếu bạn biết lý do tại sao chương trình không chạy, bạn có thể sửa được chương trình cho nó chạy*
  - *Nếu bạn hiểu chương trình của bạn, bạn sẽ có ít sai lầm và dễ dàng sửa chữa sai sót hơn. Bí quyết là viết mã đơn giản, hiệu quả, chú thích hợp lý.*

Gỡ rối
Debug

- Đối với mã nguồn, tiêu chí nào quan trọng hơn: rõ ràng hay chính xác?
  - *Nếu mã nguồn rõ ràng, bạn có thể làm cho chương trình trở nên chính xác.*
  - *Bạn có chắc là làm cho chương trình trở nên chính xác nếu nó không rõ ràng hay không?*
- 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.

Tìm kiếm và gỡ rối

- Khi có lỗi, ta thường đổ cho trình dịch, thư viện hay bất cứ nguyên nhân khách quan nào khác… tuy nhiên, cuối cùng thì lỗi vẫn là lỗi của chương trình, và trách nhiệm gỡ rối thuộc về LTV
- Phải hiểu vấn đề xuất phát từ đâu thì mới giải quyết được:
  - *Lỗi xảy ra ở đâu? Hầu hết các lỗi thường đơn giản và dễ tìm. Hãy khảo sát các đầu mối và cố gắng xác định được đoạn mã nguồn gây lỗi*
  - *Lỗi xảy ra như thế nào? Khi đã có một số thông tin về lỗi và nơi xảy ra lỗi, hãy suy nghĩ xem lỗi xảy ra như thế nào*
  - *Đâu là nguyên nhân gây lỗi? Suy luận ngược trở lại trạng thái của chương trình để xác định nguyên nhân gây ra lỗi*

“Gỡ rối liên quan đến việc suy luận lùi, giống như phá án. Một số vấn đề 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 được vấn đề cần sửa và có thể phát hiện ra một số vấn đề khác

Debugging
Heuristic

Debugging Heuristic	Áp dụng khi nào
(1) Hiểu các thông báo lỗi (error messages)	Build-time (dịch)
(2) Nghĩ trước khi viết lại chương trình	Run-time (chạy)
(3) Tìm kiếm các lỗi (bug) hay xảy ra	
(4) Divide and conquer	
(5) Viết thêm các đoạn mã kiểm tra để chương trình tự kiểm tra nó	
(6) Hiện thị kết quả	
(7) Sử dụng debugger	
(8) Tập trung vào các lệnh mới viết / mới viết lại	

Hiểu
các thông báo lỗi

- Gỡ rối khi dịch (build-time) chương trình dễ hơn gỡ rối khi chạy chương trình nếu LTV hiểu được các thông báo lỗi
- Một số thông báo lỗi đến từ preprocessor

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

Gõ sai tên file cần gọi

Thiếu dấu */

```
$ gcc217 hello.c -o hello
hello.c:1:20: stdioo.h: No such file or directory
hello.c:3:1: unterminated comment
hello.c:2: error: syntax error at end of input
```

Hiểu
các thông báo lỗi

- Một số thông báo lỗi đến từ compiler

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

Sai từ khóa

```
$ 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
```

Hiểu
các thông báo lỗi

- Một số thông báo lỗi đến từ linker

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

Sai tên hàm được gọi

Compiler **warning** (not **error**): prinf() được gọi trước khi khai báo

Linker error: không tìm thấy định nghĩa hàm 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
```

> *Việc thay đổi mã nguồn không hợp lý có thể gây ra nhiều vấn đề hơn là để nguyên không thay đổi gì, do đó phải luôn suy nghĩ trước khi làm*

Suy nghĩ
trước khi viết

- 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)*
  - *Cân nhắc: việc sửa chữa này có ảnh hưởng tới các tình huống khác hay không ?*
- Quan sát lỗi từ góc độ khác
  - *Viết đoạn mã nguồn gây lỗi ra giấy*
    - Đừng chép hết cả đoạn không có nguy cơ gây lỗi, hoặc in toàn bộ code ra giấy in => phá vỡ cây cấu trúc
  - *Vẽ hình minh họa các cấu trúc dữ liệu*
    - Nếu mà giải thuật làm thay đổi CTDL, vẽ lại hình trước khi viết lại giải thuật
  - *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

Suy nghĩ
trước khi viết

- Tạm dừng viết chương trình
  - *Khi gặp vấn đề, khó khăn, chậm tiến độ, lập tức thay đổi công việc => rút ra khỏi luồng quán tính sai lầm …*
  - *Bỏ qua đoạn chương trình có lỗi*
  - *Khi nào cảm thấy sẵn sàng thì chữa*
- Giải thích logic của đoạn mã nguồn:
  - *Cho chính bạn*
    - Tạo điều kiện để suy nghĩ lại
  - *Cho ai khác có thể phản bác*
    - Extrem programming : làm việc theo cặp, pair programming, người này LT, người kia kiểm tra, và ngược lại
  - *Cho cái gì đó không thể phản bác (cây, cốc trà đá, gấu bông…)*
    - Tạo điều kiện củng cố suy luận của mình

Tìm các lỗi tương tự

```
switch (i) {
    case 0:
        …
        /* missing break */
    case 1:
        …
        break;
    …
}
```

```
int i;
…
scanf("%d", i);
```

```
char c;
…
c = getchar();
```

```
if (i = 5)
    …
```

```
while (c = getchar() != EOF)
    …
```

```
if (5 < i < 10)
    …
```

```
if (i & j)
    …
```

Tips: nếu đặt chế độ cảnh báo (warnings) khi dịch thì hầu hết các lỗi kiểu này sẽ được phát hiện

Tìm các lỗi tương tự

- Khi gặp vấn đề, 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.

Tìm các lỗi tương tự

- Làm cho lỗi xuất hiện lại
  - *Cố gắng làm cho lỗi có thể xuất hiện lại khi cần*
  - *Nếu không được, thì thử tìm nguyên nhân tại sao lại không làm cho lỗi xuất hiện lại*

{C/C++}

1. Array as a parameter handled improperly – Tham số mảng được 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 được 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 được đóng phù hợp (buffer không bị dẹp)
10. Infinite loop - lặp vô hạn
11. Global variables used – dùng biến tổng thể

{C/C++}

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

{C/C++}

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 được 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 "
30. Using "=" when "= =" is intended or vice versa
31. Using "&" when "&&" is intended or vice versa
32. "while" used improperly instead of "if" – while được dùng thay vì if

Chia để trị

- *Thu hẹp phạm vi*
- *Tập trung vào dữ liệu gây lỗi*

Chia để trị

- Khử đầu vào
  - *Thử chương trình với các tham số đầu vào từ đơn giản đến phức tạp, từ nhỏ đến lớn để tìm lỗi*
- Ví dụ: chương trình lỗi với file đầu vào filex
  - *Tạo ra phiên bản copy của filex , tên là filexcopy*
  - *Xoá bớt nửa sau của filexcopy*
  - *Chạy chương trình với tham số đầu vào là filexcopy*
    - Nếu chạy thông => nửa đầu của filex không gây lỗi, loại bỏ nửa này, tìm lỗi trong nửa sau của filex
    - Nếu không chạy => nửa đầu của filex gây lỗi, loại bỏ nửa sau, tìm lỗi trong nửa đầu của filex
  - *Lặp cho đến khi không còn dòng nào trong filex có thể bị loại bỏ*
- Cách khác: bắt đầu với một ít dòng trong filex, thêm dần các dòng vào cho đến khi thấy lỗi

Chia để trị

- Khử mã nguồn
  - *Thử chương trình với các đoạn mã nguồn từ ngắn đến dài để tìm lỗi*
- Ví dụ: đoạn chương trình có sử dụng đến các phần CTC khác không chạy
  - *Viết riêng từng lời gọi hàm trong đoạn chương trình bị lỗi (test client)*
    - hoặc viết thêm phần kiểm tra gọi hàm vào phần CTC được gọi
  - *Chạy thử test client*
  - *Không chạy => lỗi liên quan đến việc gọi/ thực hiện CTC vừa thử*
  - *Chạy thông => lỗi nằm trong phần còn lại, tiếp tục thử gọi các hàm khác*

Viết thêm
mã kiểm tra

- Dùng internal test để khử lỗi trong chương trình và giảm nhẹ công việc tìm kiếm lỗi
  - *Chỉ cần viết thêm một hàm để kiểm tra, gắn vào trước và sau đoạn có nguy cơ, comment lại sau khi đã xử lý xong lỗi*
  - *Các kỹ thuật viết thêm mã tự kiểm tra cho chương trình đã học*
    - Kiểm tra các giá trị không thay đổi
    - Kiểm tra các đặc tính cần bảo lưu
    - Kiểm tra giá trị trả về của hàm
    - Thay đổi mã nguồn tạm thời
    - Kiểm tra mà không làm thay đổi mã nguồn
- Dùng assertion để nhận dạng các lỗi có trong chương trình
  - *Kiểm tra các giá trị không thay đổi*

Viết thêm
mã kiểm tra

```
#ifndef NDEBUG
int isValid(MyType object) {
    …
    Test invariants here.
    Return 1 (TRUE) if object passes
    all tests, and 0 (FALSE) otherwise.
    …
}
#endif

void myFunction(MyType object) {
    assert(isValid(object));
    …
    Manipulate object here.
    …
    assert(isValid(object));
}
```

Can use NDEBUG in your code, just as **`assert`** does

Hiển thị
output

- In giá trị của các biến tại các điểm có khả năng gây lỗi để định vị khu vực gây lỗi, hoặc
- Xác định tiến trình thực hiện : “đến đây 1”
- Poor:

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

stdout is buffered; chương trình 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 output

- 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 chương trình cho người khác..*
- Maybe even better:

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

- Maybe better still:

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

In debugging output ra **stderr**; debugging output có thể tách biệt với đầu ra thông thường bằng cách in ấn của chương trình

Ngoài ra: stderr không dùng buffer

Ghi ra 1 a log file

Sử dụng debugger

- IDE : kết hợp soạn thảo, biên dịch, gỡ rối …
- Các trình gỡ rối với giao diện đồ 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 được kích hoạt trực tiếp khi có lỗi hoặc gắn vào chương trình đang chạy.
- 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

Sử dụng debugger

- Nếu vẫn không phát hiện đc lỗi: dùng các BreakPoint hoạc chạy từng bước – step by step
  - *Chạy từng bước là phương sách cuối cùng*
- 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 dàng tìm lỗi, nếu đưa ra một câu hỏi sai, trình gỡ rối sẽ cho một câu trả lời, nhưng ta có thể không biết là nó đang bị sai

Sử dụng debugger

- Nhiều khi vấn đề tưởng quá đơn giản nhưng lại không phát hiện được, ví dụ các toán tử so sánh trong pascal và VB có độ ưu tiên ngang nhau, nhưng với C ?
- Thứ tự các đối số của lời gọi hàm: ví dụ **`strcpy(s1,s2)`**

```
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 một 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 độngng trong bộ xử lý Pentium

Sử dụng debugger

GDB

- Gỡ rối được các chương trình viết bằng Ada, C, C++, Objective-C, Pascal, v.v., chạy trên cùng máy cài đặt GDB hay trên máy khác
- Hoạt động trên nền UNIX và Microsoft Windows
- Các chức năng hỗ trợ:
  - *Bắt đầu chương trình, xác định những yếu tố làm ảnh hưởng đến hoạt động của chương trình*
  - *Dừng chương trình với điều kiện biết trước*
  - *Khi chương trình bị dừng, kiểm tra những gì đã xảy ra*
  - *Thay đổi các lệnh trong chương trình để LTV có thể thử nghiệm gỡ rối từng lỗi một*

Tập trung vào các thay đổi mới nhất

- Lỗi thường xảy ra ở những đoạn chương trình mới được bổ sung
- Nếu phiên bản cũ OK, phiên bản mới có lỗi => lỗi chắc chắn nằm ở những đoạn chương trình 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)

Tập trung vào các thay đổi mới nhất

- 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 chương trình: 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? Ví dụ*

```
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) ;
```

Tập trung vào các thay đổi mới nhất

- Phải gỡ rối ngay, không nên để sau
  - *Khó: Viết toàn bộ chương trình; kiểm tra toàn bộ chương trình, gỡ rối toàn bộ chương trình*
  - *Dễ hơn: Viết từng đoạn, kiểm tra từng đoạn, gỡ rối từng đoạn; viết từng đoạn, kiểm tra từng đoạn, gỡ rối từng đoạn;*
- Nên giữ lại các phiên bản trước
  - *Khó: Thay đổi mã nguồn, đánh dấu các lỗi; cố gắng nhớ xem đã thay đổi cái gì từ lần làm việc trước*
  - *Dễ hơn: Backup mã nguồn, thay đổi mã nguồn, đánh dấu các lỗi; so sánh phiên bản mới với phiên bản cũ để xác định các điểm thay đổi*

Tập trung vào các thay đổi mới nhất

Giữ lại các thay đổi trước đó

- Cách 1: Sao chép bằng tay vào một thư mục

```
…
$ mkdir myproject
$ cd myproject

   Create project files here.

$ cd ..
$ cp –r myproject myprojectDateTime
$ cd myproject

   Continue creating project files here.
…
```

  - Lặp lại mỗi lần có phiên bản mới
- Cách 2: dùng công cụ quản lý phiên bản

Tóm tắt

- *Gỡ rối là một nghệ thuật mà ta phải luyện tập thường xuyên*
- *Nhưng đó là nghệ thuật mà ta không muốn*
- *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*

2.
Kiểm thử

Testing

Kiểm thử
Testing

- 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

Khái niệm kiểm thử

- 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**

Kiểm thử và gỡ rối

- Testing tìm error; debug định vị và sửa chúng.
- Ta có mô hình "testing/debugging cycle": Ta test, rồi debug, rồi lặp lại.
- Bất kỳ 1 debugging nào nên được tiếp theo là 1 sự áp dụng lại của hàng loạt các test liên quan, đặc biệt là các bài test hồi quy. Điều này giúp tránh nảy sinh các lỗi mới khi debug.
- Test & debug không nên được thực hiện bởi cùng 1 người.

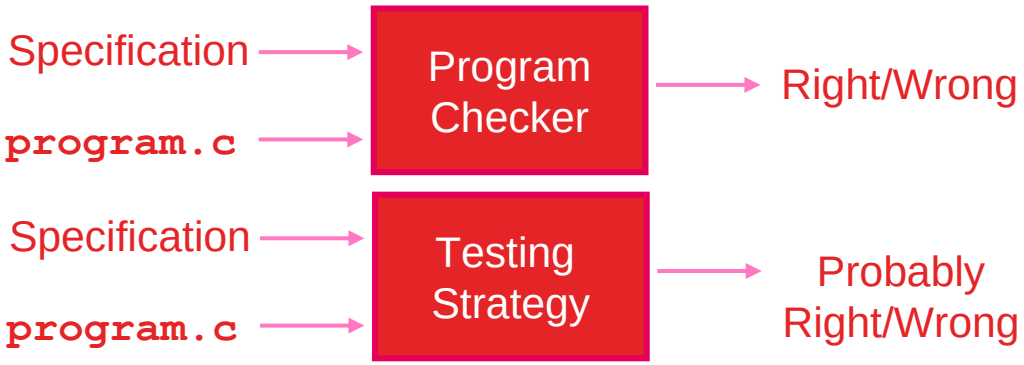
Kiểm thử và kiểm chứng chương trình

Kiểm chứng chương trình

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

Kiểm thử

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



Phân loại kiểm thử

Internal Testing

- Thiết kế dữ liệu để test chương trình

External Testing

- Thiết kế program để chương trình tự test

Các kỹ thuật kiểm thử

KIỂM THỬ NGOÀI

Kiểm thử ngoài
External testing

- External testing: Thiết kế dữ liệu để test chương trình
- Phân loại : External testing
  - ***(1) Kiểm chứng giá trị biên: Boundary testing***
  - ***(2) Kiểm chứng lệnh: Statement testing***
  - ***(3) Kiểm chứng có hệ thống: Path testing***
  - ***(4) Kiểm chứng tải: Stress testing***

Kiểm chứng
giá trị biên

(1) Boundary testing

"Là kỹ thuật kiểm chứng sử dụng các giá trị nhập vào ở trên hoặc dưới một miền giới hạn của 1 đầu vào và với các giá trị đầu vào tạo ra các đầu ra ở biên của 1 đầu ra."

– Glossary of Computerized System and Software Development Terminology

- Hầu hết các lỗi đều xảy ra ở các điều kiện biên - boundary conditions
- Nếu chương trình làm việc tốt ở điều kiện biên, nó có thể sẽ làm việc đúng với các điều kiện khác

Kiểm chứng
giá trị biên

- VD: đọc 1 dòng từ stdin và đưa vào mảng ký tự

```
int i;
char s[MAXLINE];
for (i=0; (s[i]=getchar()) != '\n' && i < MAXLINE-1; i++);
s[i] = '\0';
printf("String: |%s|\n", s);
```

- Xét các điều kiện biên
  - *Dòng rỗng –bắt đầu với '\n'*
    - In ra xâu rỗng (“\0”) => in ra “||” , ok
  - *Nếu gặp EOF trước '\n'*
    - Tiếp tục gọi getchar() và lưu ÿ vào s[i], not ok
  - *Nếu gặp ngay EOF (empty file)*
    - Tiếp tục gọi getchar() và lưu ÿ vào s[i], not ok

Kiểm chứng
giá trị biên

```
int i;
char s[MAXLINE];
for (i=0; (s[i]=getchar()) != '\n' && i < MAXLINE-1; i++);
s[i] = '\0';
printf("String: |%s|\n", s);
```

- Tiếp tục xét các ĐK biên (tt)
  - *Dòng chứa đúng MAXLINE-1 ký tự*
    - In ra đúng, với '\0' tại s[MAXLINE-1]
  - *Dòng chứa đúng MAXLINE ký tự*
    - Ký tự cuối cùng bị ghi đè, và dòng mới không bao giờ được đọc
  - *Dòng dài hơn MAXLINE ký tự*
    - 1 số ký tự, kể cả newline, không được đọc và sót lại trong stdin

Kiểm chứng
giá trị biên

- Viết lại code

```
int i;
char s[MAXLINE];
for (i=0; i< MAXLINE-1; i++)
    if ((s[i] = getchar()) == '\n')
        break;
s[i] = '\0';
```

- Trường hợp gặp EOF

```
for (i=0; i<MAXLINE-1; i++)
    if ((s[i] = getchar()) == '\n' || s[i] == EOF)
        break;
s[i] = '\0';
```

- Các trường hợp khác?
  - *Nearly full*
  - *Exactly full*
  - *Over full*

SAI!!?!

Kiểm chứng giá trị biên

- Viết lại code

```
for (i=0; ; i++) {
   int c = getchar();
   if (c==EOF || c=='\n' || i==MAXLINE-1) {
                s[i] = '\0';
     break;
   }
   else s[i] = c;
}
```

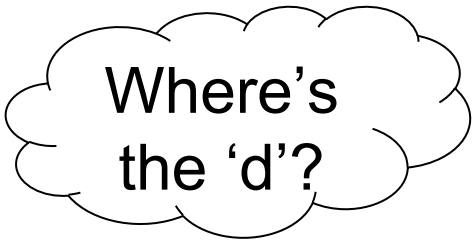
Vấn đề
(với MAXLINE=9)

Input:

```
Four
score and seven
years
```

Output:

```
FourØ
score anØ
 sevenØ
yearsØ
```



Vấn đề không rõ ràng trong đặc tả

- Nếu dòng quá dài, xử lý thế nào?
  - *Giữ MAXLINE ký tự đầu, bỏ qua phần còn lại?*
  - *Giữ MAXLINE ký tự đầu + '\0', bỏ qua phần còn lại?*
  - *Giữ MAXLINE ký tự đầu+'\0', lưu phần còn lại cho lần gọi sau của input function?*
- Có thể phần đặc tả - specification không hề đề cập khi MAXLINE bị quá
  - *Có thể người ta không muốn dòng dài quá giới hạn trong mọi trường hợp*
  - *Đạo đức: kiểm tra đã phát hiện ra một vấn đề thiết kế, thậm chí có thể là một vấn đề đặc điểm kỹ thuật !*
- Quyết định phải làm gì
  - *Cắt những dòng quá dài?*
  - *Lưu phần còn lại để đọc như 1 dòng mới?*

Kiểm tra điều kiện trước và sau

- Xác định những thuộc tính cần đi trước (đk trước) và sau (đk sau) mã nguồn đc thi hành
- Ví dụ: các giá trị đầu vào phải thuộc 1 phạm vi cho trước

```
double avg( double a[], int n) {
   int i; double sum=0.0;
   for ( i = 0; i<n; i++)
          sum+=a[i];
   return sum/n;
}
```

Nếu n=0 ?, nếu n<0 ?

Có thể thay : `return n <=0 ? 0.0: sum/n;`

Tháng 11/1998, chiến hạm Yorktown bị chìm: nhập vào giá trị 0, chương trình không kiểm tra dữ liệu nhập dẫn đến chia cho 0, và lỗi làm tầu rối loạn, hệ thống đẩy ngưng hoạt động, tàu chìm.

Kiểm chứng lệnh

(2) Statement testing

"Testing để thỏa mãn điều kiện rằng mỗi lệnh trong 1 chương trình phải thực hiện ít nhất 1 lần khi testing."

– Glossary of Computerized System and Software Development Terminology

Kiểm chứng lệnh

- Example pseudocode:

```
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 được 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*

Kiểm chứng lộ trình

(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 chương trình 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 chương trình*

Kiểm chứng lộ trình

- Example pseudocode:

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

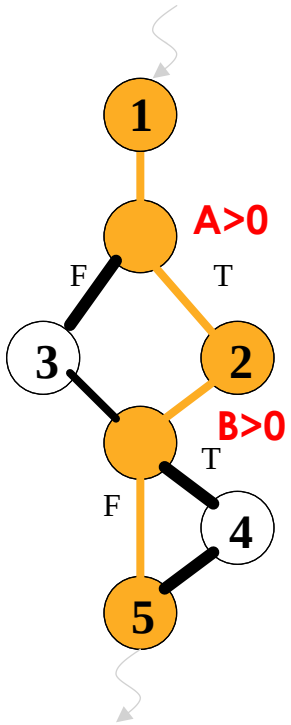
Path testing:

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!!!

Kiểm chứng lộ trình

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



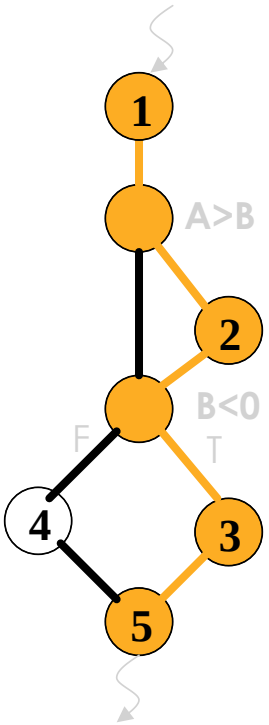
What is the path condition for path **<1,2,5>?**

(A>0) && (B≤0)

Kiểm chứng lộ trình

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

(5)  output(Z)
```



What is the path condition for path **<1,2,3,5>?**

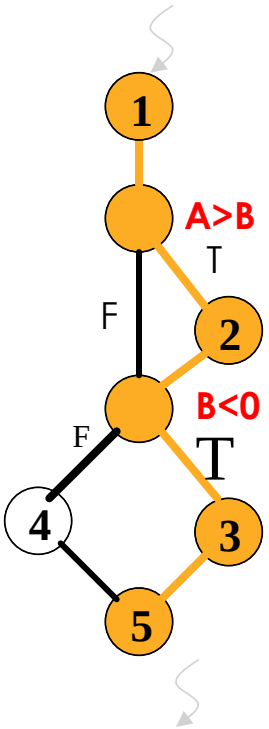
(A>B) && (B<0)

Kiểm chứng lộ trình

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

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

(5)  output(Z)
```



What is the path condition for path <1,2,3,5>?

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

Kiểm chứng tải

(4) Stress testing

"Tiến hành thử nghiệm để đánh giá một hệ thống hay thành phần tại hoặc vượt quá các giới hạn của các yêu cầu cụ thể của nó"

– Glossary of Computerized System and Software Development Terminology

- Phải tạo
  - *Một tập lớn đầu vào*
  - *Các đầu vào ngẫu nhiên (binary vs. ASCII)*
- Nên dùng máy tính để tạo đầu vào

Kiểm chứng tải

- Example program:

Stress testing: Phải cung cấp **random** (binary and ASCII) inputs

```
#include <stdio.h>
int main(void) {
    char c;
    while ((c = getchar()) != EOF)
        putchar(c);
    return 0;
}
```

- Mục tiêu: Copy tất cả các ký tự từ stdin vào stdout; nhưng lưu ý bug!!!
- Làm việc với tập dữ liệu ASCII chuẩn (tự tạo)
- Máy tính tự tạo ngẫu nhiên tập dữ liệu dạng 255 (decimal), hay 11111111 (binary), và EOF để dừng vòng lặp

Kiểm chứng tải

- Example program:

```
#include <stdio.h>
int main(void) {
    short charCount = 0;
    while (getchar() != EOF)
        charCount++;
    printf("%hd\n", charCount);
    return 0;
}
```

Stress testing: Phải cung cấp số lượng input **rất lớn**

- Mục tiêu: Đếm và in số các kỹ tự trong stdin
- Làm việc với tập dữ lieu có kích thước phù hợp
- Sẽ có lỗi với tập dữ liệu do máy tạo chứa hơn 32767 characters

Các kỹ thuật kiểm thử

KIỂM THỬ TRONG

Kiểm thử trong
Internal testing

- Internal testing: Thiết kế chương trình để chương trình tự kiểm thử
- Internal testing techniques

  (1) Kiểm tra bất biến - Testing invariants

  (2) Kiểm tra các thuộc tính lưu trữ -Verifying conservation properties

  (3) Kiểm tra các giá trị trả về - Checking function return values

  (4) Tạm thay đổi code - Changing code temporarily

  (5) Giữ nguyên mã thử nghiệm - Leaving testing code intact

Kiểm tra tính bất biến

(1) Testing invariants

- *Thử nghiệm các đk trước và sau*
- *Vài khía cạnh của cấu trức dữ liệu không đc thay đổi*
- *1 hàm tác động đến cấu trúc dữ liệu phải kiểm tra các bất biến ở đầu và cuối nó*

- *Ví dụ: Hàm "doubly-linked list insertion"*
  - Kiểm tra ở đầu và cuối

Xoay doubly-linked list
Khi node x trỏ ngược lại node y, thì liệu node y có trỏ ngược lại node x?

- *Ví dụ: "binary search tree insertion" function*
  - Kiểm tra ở đầu và cuối

Xoay tree
Các nodes có còn đc sắp xếp không?

Kiểm tra tính bất biến

```
#ifndef NDEBUG
int isValid(MyType object) {
    …
    Test invariants here.
    Return 1 (TRUE) if object passes
    all tests, and 0 (FALSE) otherwise.
    …
}
#endif

void myFunction(MyType object) {
    assert(isValid(object));
    …
    Manipulate object here.
    …
    assert(isValid(object));
}
```

- Có thể dung assert()

Có thể dùng NDEBUG trong code, giống như **assert**

Kiểm tra các thuộc tính lưu trữ

- *Khái quát hóa của testing invariants*
- *1 hàm cần kiểm tra các cấu trúc dữ liệu bị tác động tại các điểm đầu và cuối*

- *VD: hàm* `str_concat()`
  - Tại điểm đầu, tìm độ dài của 2 xâu đã cho; tính tổng
  - Tại điểm cuối, tìm độ dài của xâu kết quả
  - 2 độ dài có bằng nhau không ?
- *VD: Hàm chèn thêm PT vào danh sách -List insertion function*
  - Tại điểm khởi đầu, tính độ dài ds
  - Tại điểm cuối, Tính độ dài mới
  - Độ dài mới = độ dài cũ + 1?

Kiểm tra
giá trị trả về

- Trong Java và C++
  - *Phương thức bị phát hiện có lỗi có thể tung ra một "checked exception"*
  - *Phương thưc triệu gọi phải xử lý ngoại lệ*

- Trong C
  - *Không có cơ chế xử lý exception*
  - *Hàm phát hiện có lỗi chủ yếu thông qua giá trị trả về*
  - *Người LT thường dễ dàng quên kiểm tra GT trả về*
  - *Nói chung là chúng ta nên kiểm tra GT trả về*

Kiểm tra
giá trị trả về

- *VD:* ***`scanf()`*** *trả về số của các giá trị được đọc*

Bad code

```
int i;
scanf("%d", &i);
```

Good code

```
int i;
if (scanf("%d", &i) != 1)
   /* Error */
```

Bad code???

```
int i = 100;
printf("%d", i);
```

Good code, or overkill???

```
int i = 100;
if (printf("%d", i) != 3)
   /* Error */
```

- *VD:* ***`printf()`*** *có thể bị lỗi nếu ghi ra file và đĩa bị đầy; Hàm này trả về số ký tự được ghi (không phải giá trị)*

Tạm thay đổi mã nguồn

- *Tạm thay đổi code để tạo ranh giới nhân tạo hoặc stress tests*
- *VD: chương trình sắp xếp trên mảng*
  - Tạm đặt kích thước mảng nhỏ
  - chương trình có xử lý tràn số hay không ?
- *Viết 1 phiên bản hàm cấp phát bộ nhớ và phát hiện ra lỗi sớm, để kiểm chứng đoạn mã nguồn bị lỗi thiếu bộ nhớ*

```
void *testmalloc( size_t n) {
   static int count =0;
   if (++count  > 10)
          return NULL;
   else
          return malloc(n);
}
```

Giữ nguyên mã kiểm tra

- *Để nguyên trạng các đoạn kiểm tra trên code*
- *Có thể khoanh lại =* ***`#ifndef NDEBUG`*** *…* ***`#endif`***
- *Kiểm tra với tùy chọn –DNDEBUG gcc*
  - Bặt/Tắt assert macro
  - Cũng có thể Bật/tắt debugging code

- Cẩn trọng với conflict:
  - *Mở rộng thử nghiệm nội bộ có thể giảm chi phí bảo trì*
  - *Code rõ ràng có thể giảm chi phí bảo trì*
  - *Nhưng mở rộng thử nghiệm nội bộ có thể làm giảm độ rõ ràng của Code*

Các chiến lược kiểm thử

Các chiến lược kiểm thử

Testing strategies

(1) Kiểm chứng tăng dần - Testing incrementally

(2) So sánh các cài đặt - Comparing implementations

(3) Kiểm chứng tự động - Automation

(4) Bug-driven testing

(5) Tiêm, gài lỗi - Fault injection

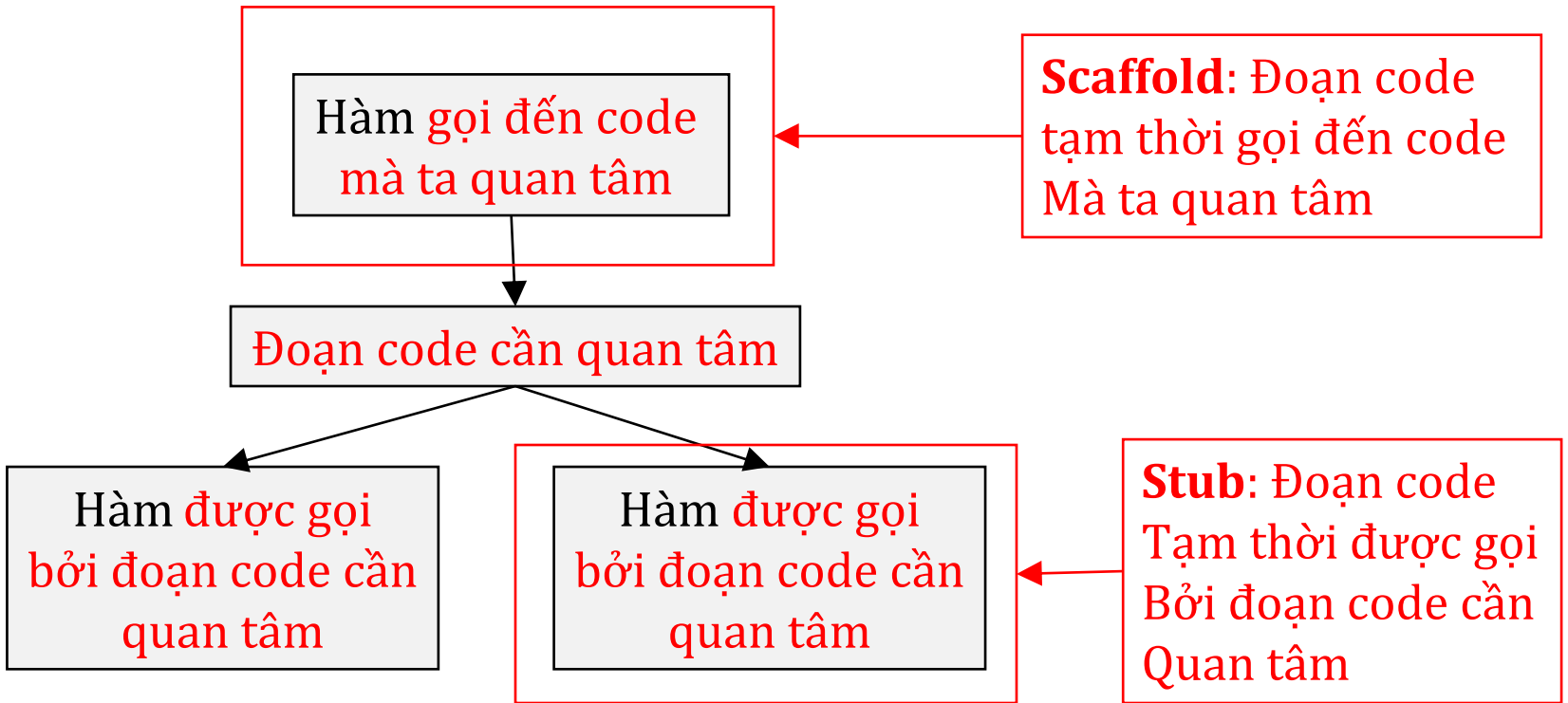
Kiểm chứng
tăng dần

(1) Testing incrementally

- *Test khi viết code*
  - Thêm test 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 hệ 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

Kiểm chứng
tăng dần

- *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 hoạt động như nhau*
- *Ví dụ: So sánh hành vi của các hàm bạn tạo trong str.h với các hàm trong thư viện string.h*
- *Đôi khi 1 kết quả có thể đc tính bằng 2 cách khác nhau, 1 bài toán có thể giải bằng 2 phương pháp, thuật toán khác nhau. Ta có thể xây dựng cả 2 chương trình, nếu chúng có cùng kết quả thì có thể khẳng định cả 2 cùng đúng, còn kết quả khác nhau thì ít nhất 1 trong 2 chương trình 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

Kiểm chứng tự động
Automation

- 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*

Kiểm chứng tự động
Automation

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

Kiểm chứng hướng lỗi
Bug-driven

(4) Kiểm chứng hướng lỗi: Bug-driven testing

- *Tìm thấy 1 bug => Ngay lập tức tạo 1 test để bắt lỗi*
- *Đơn giản hóa việc kiểm chứng lùi*

(5) Fault injection

- *Chủ động (tạm thời) cài các bugs!!!*
- *Rồi quyết định nếu tìm thấy chúng*
- *Kiểm chứng bản thân kiểm chứng!!!*

Các phương pháp kiểm thử

Các 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 đc gọi là structural testing.

Kiểm thử
hộp trắng

- Còn được gọi là clear box testing, glass box testing, transparent box testing, or 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

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 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 không hợp lệ và 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

- 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 đen

Các kỹ thuật chính của kiểm thử hộp đen

- Decision Table testing
- Pairwise testing
- State transition tables
- Tests of Customer Requirement
- Equivalence partitioning
- Boundary value analysis
- Failure Test Cases

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 các kĩ thuật của cả kiểm thử hộp đen và hộp trắng

Ai kiểm thử cái gì?

- Programmers
  - *White-box testing*
  - *Ưu điểm: Người triển khai nắm rõ mọi luồng dữ liệu*
  - *Nhược: Bị ảnh hưởng bởi cách thức code đc thiết kê/viết*
- Quality Assurance (QA) engineers
  - *Black-box testing*
  - *Pro: Không có khái niệm về implementation*
  - *Con: Không muốn test mọi logical paths*
- Customers
  - *Field testing*
  - *Pros: Có các cách sử dụng chương trình bất ngờ;dễ gây lỗi*
  - *Cons: Không đủ trường hợp; khách hàng không thích tham gia vào quá trình test ;*

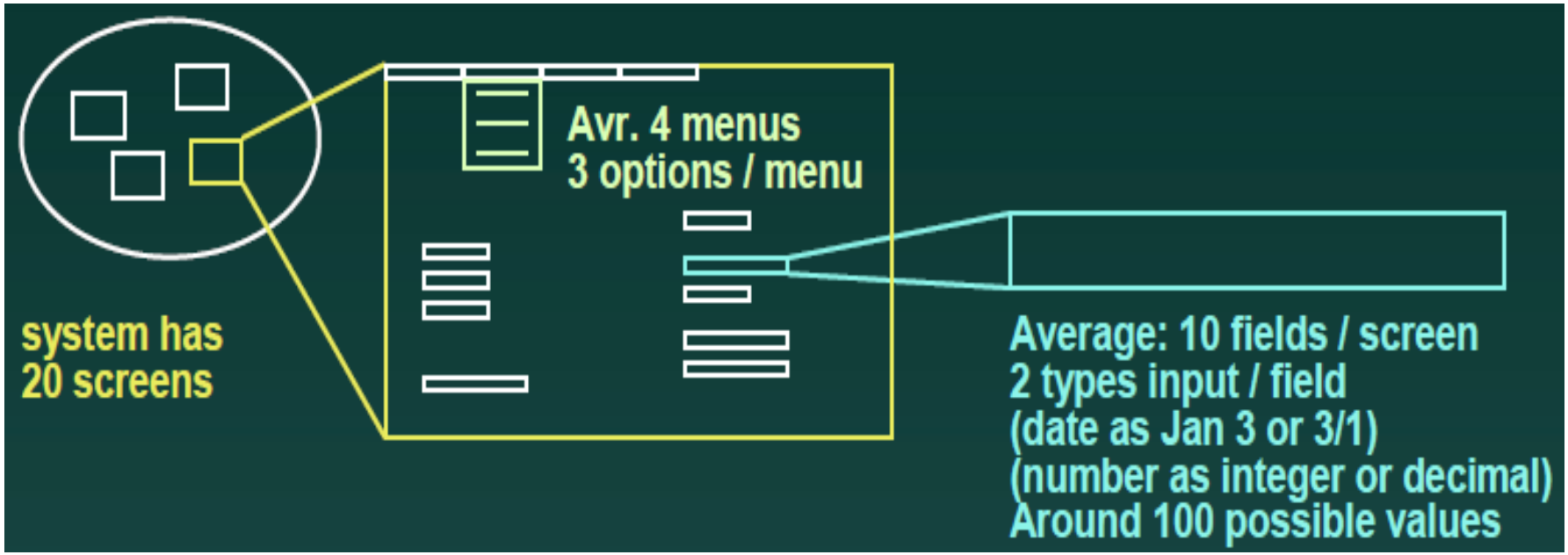
Các phương pháp khác

- Sanity testing
- Smoke testing
- Software testing
- Stress testing
- Test automation
- Web Application Security Scanner
- Fuzzing
- Acceptance testing
- Sanwich Testing

Các mức độ kiểm thử

- Unit: testing các mẫu 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: testing 1 tập hợp các units tạo thành 1 thành phần (vd: program, package, task, interacting object classes,…)
- Product: testing các thành phần tạo thành 1 sản phẩm (subsystem, application,…)
- System: testing toàn bộ hệ thống
- Testing thường:
  - *Bắt đầu = functional (black-box) tests,*
  - *Rồi thêm = structural (white-box) tests, và*
  - *Tiến hành từ unit level đến system level với 1 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
- Càng ít thời gian, càng nhiều để thời gian để test luôn có giới hạn
- 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…)*

The Testing Paradox

- Mục đích của testing: để tìm ra lỗi
- Tìm thấy lỗi làm hủy hoại sự tự tin

=> 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

Thanks!

Any questions?

Email me at trungtt@soict.hust.edu.vn

4.
Tinh chỉnh mã nguồn

Code tuning

Tăng hiệu năng chương trình

- Sau khi áp dụng các kỹ thuật xây dựng chương trình phần mềm
- Chương trình đã có tốc độ đủ nhanh
  - *Không nhất thiết phải quan tâm đến viêc tối ưu hóa hiệu năng*
  - *Chỉ cần giữ cho chương trình đơn giản và dễ đọc*
- Hầu hết các thành phần của 1 chương trình có tốc độ đủ nhanh
  - *Thường chỉ một phần nhỏ làm cho chương trình chạy chậm*
  - *Tối ưu hóa riêng phần này nếu cần*

Tăng hiệu năng chương trình

- Các bước làm tăng hiệu năng thực hiện chương trình
  - *Tính toán thời gian thực hiện của các phần khác nhau trong chương trình*
  - *Xác định các "hot spots" – đoạn mã lệnh đòi hỏi nhiều thời gian thực hiện*
  - *Tối ưu hóa phần chương trình đòi hỏi nhiều thời gian thực hiện*
  - *Lặp lại các bước nếu cần*

Tăng hiệu năng chương trình

- Cấu trúc dữ liệu tốt hơn, giải thuật tốt hơn
  - *Cải thiện độ phức tạp tiệm cận (asymptotic complexity)*
    - Tìm cách khống chế tỉ lệ giữa số phép toán cần thực hiện và số lượng các tham số đầu vào
    - Ví dụ: thay giải thuật sắp xếp có độ phức tạp **O(n2)** bằng giải thuật có độ phức tạp **O(n log n)**
  - *Cực kỳ quan trọng khi lượng tham số đầu vào rất lớn*
  - *Đòi hỏi LTV phải nắm vững kiến thức về CTDL và giải thuật*

Tăng hiệu năng chương trình

- Mã nguồn tốt hơn: viết lại các đoạn lệnh sao cho chúng có thể được trình dịch tự động tối ưu hóa và tận dụng tài nguyên phần cứng
  - *Cải thiện các yếu tố không thể thay đổi*
    - Ví dụ: Tăng tốc độ tính toán bên trong các vòng lặp: từ 1000n thao tác tính toán bên trong vòng lặp xuống còn 10n thao tác tính toán
  - *Cực kỳ quan trọng khi 1 phần của chương trình chạy chậm*
  - *Đòi hỏi LTV nắm vững kiến thức về phần cứng, trình dịch và quy trình thực hiện chương trình*

Tinh chỉnh
mã nguồn

- Thay đổi mã nguồn đã chạy thông theo hướng hiệu quả hơn nữa
- Chỉ thay đổi ở phạm vi hẹp, ví dụ như chỉ liên quan đến 1 chương trình con, 1 tiến trình hay 1 đoạn mã nguồn
- Không liên quan đến việc thay đổi thiết kế ở phạm vi rộng, nhưng có thể góp phần cải thiện hiệu năng cho từng phần trong thiết kế tổng quát

Tinh chỉnh mã nguồn

- Có 3 cách tiếp cận để cải thiện hiệu năng thông qua cải thiện mã nguồn
  - *Lập hồ sơ mã nguồn (profiling): chỉ ra những đoạn lệnh tiêu tốn nhiều thời gian thực hiện*
  - *Tinh chỉnh mã nguồn (code tuning): tinh chỉnh các đoạn mã nguồn*
  - *Tinh chỉnh có chọn lựa (options tuning): tinh chỉnh thời gian thực hiện hoặc tài nguyên sử dụng để thực hiện chương trình*

Tinh chỉnh
mã nguồn

- Khi nào cần cải thiện hiệu năng theo các hướng này
- Sau khi đã kiểm tra và gỡ rối chương trình
  - *Không cần tinh chỉnh 1 chương trình chạy chưa đúng*
  - *Việc sửa lỗi có thể làm giảm hiệu năng chương trình*
  - *Việc tinh chỉnh thường làm cho việc kiểm thử và gỡ rối trở nên phức tạp*
- Sau khi đã bàn giao chương trình
  - *Duy trì và cải thiện hiệu năng*
  - *Theo dõi việc giảm hiệu năng của chương trình khi đưa vào sử dụng*

Tinh chỉnh mã nguồn và tăng hiệu năng chương trình

- Việc giảm thiểu số dòng lệnh viết bằng 1 NNLT bậc cao KHÔNG có nghĩa là
  - *Làm tăng tốc độ chạy chương trình*
  - *Làm giảm số lệnh viết bằng ngôn ngữ máy*

```
for (i = 1;i<11;i++)  a[i] = i;
```

```
a[ 1 ] = 1 ; a[ 2 ] = 2 ;
a[ 3 ] = 3 : a[ 4 ] = 4 ;
a[ 5 ] = 5 ; a[ 6 ] = 6 ;
a[ 7 ] = 7 ; a[ 8 ] = 8 ;
a[ 9 ] = 9 ; a[ 10 ] = 10 ;
```

Language	*for*-Loop Time	Straight-Code Time	Time Savings	Performance Ratio
Visual Basic	8.47	3.16	63%	2.5:1
Java	12.6	3.23	74%	4:1

Tinh chỉnh mã nguồn và tăng hiệu năng chương trình

- Luôn định lượng được hiệu năng cho các phép toán
- Hiệu năng của các phép toán phụ thuộc vào:
  - *Ngôn ngữ lập trình*
  - *Trình dịch / phiên bản sử dụng*
  - *Thư viện / phiên bản sử dụng*
  - *CPU*
  - *Bộ nhớ máy tính*
- Hiệu năng của việc tinh chỉnh mã nguồn trên các máy khác nhau là khác nhau.

Tinh chỉnh mã nguồn và tăng hiệu năng chương trình

- Một số kỹ thuật viết mã hiệu quả được áp dụng để tinh chỉnh mã nguồn
- Nhưng nhìn chung không nên vừa viết chương trình vừa tinh chỉnh mã nguồn
  - *Không thể xác định được những nút thắt trong chương trình trước khi chạy thử toàn bộ chương trình*
  - *Việc xác định quá sớm các nút thắt trong chương trình sẽ gây ra các nút thắt mới khi chạy thử toàn bộ chương trình*
  - *Nếu vừa viết chương trình vừa tìm cách tối ưu mã nguồn, có thể làm sai lệch mục tiêu của chương trình*

5.
Một số kỹ thuật

Tinh chỉnh mã nguồn

Một số kỹ thuật tinh chỉnh mã nguồn

- *Tinh chỉnh các biểu thức logic*
- *Tinh chỉnh các vòng lặp*
- *Tinh chỉnh việc biến đổi dữ liệu*
- *Tinh chỉnh các biểu thức*
- *Tinh chỉnh dãy lệnh*
- *Viết lại mã nguồn bằng ngôn ngữ Assembler*

Tinh chỉnh các biểu thức logic

- Không kiểm tra khi đã biết kết quả rồi
  - *Initial code*

```
if ( 5 < x ) && ( x < 10 ) ….
```

  - *Tuned code*

```
if ( 5 < x )
     if ( x < 10 )
      ….
```

Tinh chỉnh các biểu thức logic

- Không kiểm tra khi đã biết kết quả rồi

Ví dụ

```
negativeInputFound = False;
for ( i = 0; i < iCount; i++ ) {
    if ( input[ i ] < 0 ) {
        negativeInputFound = True;
    }
}
```

Dùng break

Language	Straight Time	Code-Tuned Time	Time Savings
C++	**4.27**	**3.68**	**14%**
Java	4.85	3.46	29%

Tinh chỉnh các biểu thức logic

- Sắp xếp thứ tự các phép kiểm tra theo tần suất xảy ra kết quả đúng
  - *Initial code*

```
Select inputCharacter
Case "+", "="
    ProcessMathSymbol( inputCharacter )
Case "0" To "9"
    ProcessDigit( inputCharacter )
Case ",", ".", ":", ";", "!", "?"
    ProcessPunctuation( inputCharacter )
Case " "
    ProcessSpace( inputCharacter )
Case "A" To "Z", "a" To "z"
    ProcessAlpha( inputCharacter )
Case Else
    ProcessError( inputCharacter )
End Select
```

Tinh chỉnh các biểu thức logic

- Sắp xếp thứ tự các phép kiểm tra theo tần suất xảy ra kết quả đúng
  - *Tuned code*

```
Select inputCharacter
Case "A" To "Z", "a" To "z"
    ProcessAlpha( inputCharacter )
Case " "
    ProcessSpace( inputCharacter )
Case ",", ".", ":", ";", "!", "?"
    ProcessPunctuation( inputCharacter )
Case "0" To "9"
    ProcessDigit( inputCharacter )
Case "+", "="
    ProcessMathSymbol( inputCharacter )
Case Else
    ProcessError( inputCharacter )
End Select
```

Language	Straight Time	Code-Tuned Time	Time Savings
C#	0.220	0.260	-18%
Java	2.56	2.56	0%
Visual Basic	**0.280**	**0.260**	**7%**

Tinh chỉnh các biểu thức logic

- Sắp xếp thứ tự các phép kiểm tra theo tần suất xảy ra kết quả đúng
  - *Tuned code: chuyển lệnh switch thành các lệnh if - then - else*

Language	Straight Time	Code-Tuned Time	Time Savings
C#	0.630	0.330	48%
Java	0.922	0.460	50%
Visual Basic	**1.36**	**1.00**	**26%**

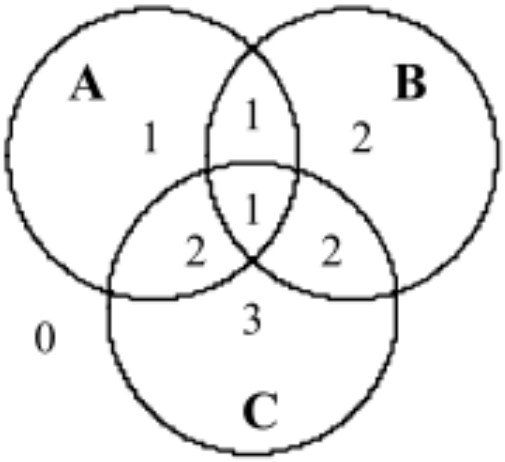
Tinh chỉnh các biểu thức logic

- So sánh hiệu năng của các lệnh có cấu trúc tương đương

Language	*case*	*if-then-else*	Time Savings	Performance Ratio
C#	0.260	0.330	-27%	1:1
Java	2.56	0.460	82%	6:1
Visual Basic	0.260	1.00	258%	1:4

Tinh chỉnh các biểu thức logic

- Thay thế các biểu thức logic phức tạp bằng bảng tìm kiếm kết quả

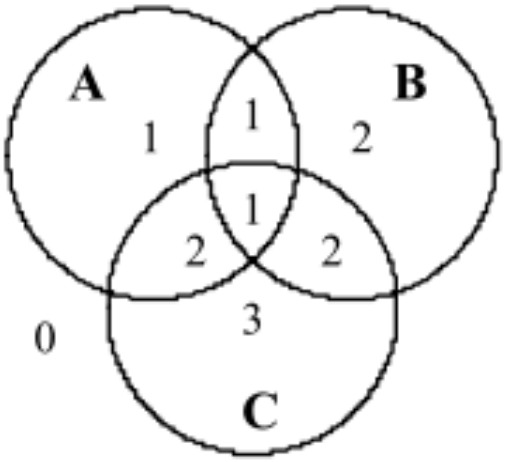


Initial code

```
if ( ( a && !c ) || ( a && b && c ) ) {
        category = 1;
}
else if ( ( b && !a ) || ( a && c && !b
) ) {
        category = 2;
}
else if ( c && !a && !b ) {
        category = 3;
}
else {
        category = 0;
}
```

Tinh chỉnh các biểu thức logic

- Thay thế các biểu thức logic phức tạp bằng bảng tìm kiếm kết quả




```
// define categoryTable
static int categoryTable[2][2][2] = {
// !b!c  !bc  b!c  bc
0,   3,   2,   2,   //   !a
1,   2,   1,   1    //    a
};
...
```

Tuned code

```
category = categoryTable[ a ][ b ][ c ];
```

Tinh chỉnh các vòng lặp

- Loại bỏ bớt việc kiểm tra điều kiện bên trong vòng lặp
  - *Initial code*

```
for ( i = 0; i < count; i++ ) {
if ( sumType == SUMTYPE_NET ) {
        netSum = netSum + amount[ i ];
}
else {
        grossSum = grossSum + amount[ i ];
}
}
```

Tinh chỉnh các vòng lặp

- Loại bỏ bớt việc kiểm tra điều kiện bên trong vòng lặp
  - *Tuned code*

```
if ( sumType == SUMTYPE_NET ) {
    for ( i = 0; i < count; i++ ) {
        netSum = netSum + amount[ i ];
    }
}
else {
    for ( i = 0; i < count; i++ ) {
        grossSum = grossSum + amount[ i ];
    }
}
```

Language	Straight Time	Code-Tuned Time	Time Savings
C++	**2.81**	**2.27**	**19%**
Java	3.97	3.12	21%
Visual Basic	2.78	2.77	<1%
Python	8.14	5.87	28%

Tinh chỉnh các vòng lặp

- Nếu các vòng lặp lồng nhau, đặt vòng lặp xử lý nhiều công việc hơn bên trong
  - *Initial code*

```
for ( column = 0; column < 100; column++ )
{
    for ( row = 0; row < 5; row++ ) {
        sum = sum + table[ row ][ column ];
    }
}
```

  - *Tuned code*

```
for (row = 0; row < 5; row++ ) {
    for (column = 0; column < 100; column++) {
        sum = sum + table[ row ][ column ];
    }
}
```

Tinh chỉnh các vòng lặp

- Một số kỹ thuật viết các lệnh lặp hiệu quả đã học
  - *Ghép các vòng lặp với nhau*
  - *Giảm thiểu các phép tính toán bên trong vòng lặp nếu có thể*

```
for (i=0; i<n; i++) {
    balance[i] += purchase->allocator->indiv->borrower;
    amounttopay[i] = balance[i]*(prime+card)*pcentpay;
}
newamt = purchase->allocator->indiv->borrower;
payrate = (prime+card)*pcentpay;
for (i=0; i<n; i++) {
    balance[i] += newamt;
    amounttopay[i] = balance[i]*payrate;
}
```

Tinh chỉnh việc
biến đổi dữ liệu

- Một số kỹ thuật viết mã hiệu quả đã học:
  - *Sử dụng kiểu dữ liệu có kích thước nhỏ nếu có thể*
  - *Sử dụng mảng có số chiều nhỏ nhất có thể*
  - *Đem các phép toán trên mảng ra ngoài vòng lặp nếu có thể*
  - *Sử dụng các chỉ số phụ*
  - *Sử dụng biến trung gian*
  - *Khai báo kích thước mảng = 2^n*

Tinh chỉnh các biểu thức

- Thay thế phép nhân bằng phép cộng
- Thay thế phép lũy thừa bằng phép nhân
- Thay việc tính các hàm lượng giác bằng cách gọi các hàm lượng giác có sẵn
- Sử dụng kiểu dữ liệu có kích thước nhỏ nếu có thể
  - *long int* → *int*
  - *floating-point* → *fixed-point, int*
  - *double-precision* → *single-precision*
- Thay thế phép nhân đôi / chia đôi số nguyên bằng các toán tử bit: <<, >>
- Sử dụng hằng số hợp lý
- Tính trước kết quả
- Sử dụng biến trung gian
- Sử dụng các hàm inline

Viết lại mã nguồn bằng Assembler

- Viết chương trình hoàn chỉnh bằng 1 NNLT bậc cao
- Kiểm tra tính chính xác của toàn bộ chương trình
- Nếu cần cải thiện hiệu năng thì áp dụng kỹ thuật lập hồ sơ mã nguồn để tìm "hot spots" (chỉ khoảng 5 % chương trình thường chiếm 50% thời gian thực hiện, vì vậy ta có thể thường xác định đc 1 mẩu code như là hot spots)
- Viết lại những mẩu nhỏ các lệnh bằng assembler để tăng tốc độ thực hiện

Tận dụng khả năng của trình dịch

- Trình dịch có thể thực hiện 1 số thao tác tôi ưu hóa tự động
  - *Cấp phát thanh ghi*
  - *Lựa chọn lệnh để thực hiện và thứ tự thực hiện lệnh*
  - *Loại bỏ 1 số dòng lệnh kém hiệu quả*
- Nhưng trình dịch không thể tự xác định
  - *Các hiệu ứng phụ (side effect) của hàm hay biểu thức: ngoài việc trả ra kết quả, việc tính toán có làm thay đổi trạng thái hay có tương tác với các hàm/biểu thức khác hay không*
  - *Hiện tượng nhiều con trỏ trỏ đến cùng 1 vùng nhớ (memory aliasing)*
- Tinh chỉnh mã nguồn có thể giúp nâng cao hiệu năng
  - *Chạy thử từng đoạn chương trình để xác định "hot spots"*
  - *Đọc lại phần mã viết bằng assembly do trình dịch sản sinh ra*
  - *Xem lại mã nguồn để giúp trình dịch làm tốt công việc của nó*

Khai thác hiệu quả phần cứng

- Tốc độ của 1 tập lệnh thay đổi khi môi trường thực hiện thay đổi
- Dữ liệu trong thanh ghi và bộ nhớ đệm được truy xuất nhanh hơn dữ liệu trong bộ nhớ chính
  - *Số các thanh ghi và kích thước bộ nhớ đệm của các máy tính khác nhau*
  - *Cần khai thác hiệu quả bộ nhớ theo vị trí không gian và thời gian*
- Tận dụng các khả năng để song song hóa
  - *Pipelining: giải mã 1 lệnh trong khi thực hiện 1 lệnh khác*
    - Áp dụng cho các đoạn mã nguồn cần thực hiện tuần tự
  - *Superscalar: thực hiện nhiều thao tác trong cùng 1 chu kỳ đồng hồ (clock cycle)*
    - Áp dụng cho các lệnh có thể thực hiện độc lập
  - *Speculative execution: thực hiện lệnh trước khi biết có đủ điều kiện để thực hiện nó hay không*

Tổng kết

- Hãy lập trình một cách thông minh, đừng quá cứng nhắc
  - *Không cần tối ưu 1 chương trình đủ nhanh*
  - *Tối ưu hóa chương trình đúng lúc, đúng chỗ*
- Tăng tốc chương trình
  - *Cấu trúc dữ liệu tốt hơn, giải thuật tốt hơn: hành vi tốt hơn*
  - *Các đoạn mã tối ưu: chỉ thay đổi ít*
- Các kỹ thuật tăng tốc chương trình
  - *Tinh chỉnh mã nguồn theo hướng*
    - Giúp đỡ trình dịch
    - Khai thác khả năng phần cứng

Thanks!

Any questions?

Email me at trungtt@soict.hust.edu.vn