Lập trình phòng thủ
<Defensive Programming>

(3LT-2BT)

Last update 09-2010

DEFENSIVE PROGRAMMING?

- Xuất phát từ khái niệm defensive driving.
- Khi lái xe bạn luôn phải tâm niệm rằng bạn không bao giờ biết chắc được người lái xe khác sẽ làm gì. Bằng cách đó, bạn có thể chắc chắn rằng khi họ làm điều gì nguy hiểm, thì bạn sẽ không bị ảnh hưởng (tai nạn).
- Bạn có trách nhiệm bảo vệ bản thân, ngay cả khi người khác có lỗi
- Trong defensive programming, ý tưởng chính là nếu chương trình (con) được truyền dữ liệu tồi, nó cũng không sao, kể cả khi với CT khác thì sẽ bị fault.
- Một cách tổng quát, lập trình phòng thủ nghĩa là : làm thế nào để tự bảo vệ mình khỏi thế giới

m’

1. Bảo vệ CT khỏi các Invalid Inputs

- Trong thực tiễn :“Garbage in, garbage out.”
- Trong lập trình “*rác rưởi vào – rác rưởi ra*” là điều không chấp nhận
-

o là gì !

- Với 1 CT tốt thì :”rác rưởi vào, không có gì ra”, “rác rưởi vào, có thông báo lỗi” hoặc “không cho phép rác rưởi vào”.
- Theo tiêu chuẩn ngày nay, “garbage in, garbage out” là dấu hiệu của những CT tồi, không an toàn

Ba cách để xử lý rác vào

- ***Kiểm tra giá trị của mọi dữ liệu từ nguồn bên ngoài***
  - Khi nhận dữ liệu từ file, bàn phím, mạng, hoặc từ các nguồn ngoài khác, hãy kiểm tra đê đảm bảo rằng dữ liệu nằm trong giới hạn cho phép.
  - Hãy đảm bảo rằng giá trị số nằm trong dung sai và xâu phải đủ ngắn để xử lý. Nếu một chuỗi được dự định để đại diện cho một phạm vi giới hạn của các giá trị (như một

    i từ chối .
  - ng

    :làm tràn bộ nhớ, injected SQL commands, injected html hay XML code, tràn số …

Ba cách để xử lý rác vào

- ***Check the values of all routine input parameters***
  - Kiểm tra giá trị của tất cả các tham số truyền vào các hàm cũng cần như kiểm tra dữ liệu nhập từ nguồn ngoài khác
- ***Decide how to handle bad inputs***
  - Khi phát hiện 1 tham số hay 1 dữ liệu không hợp lệ, bạn cần làm gì với nó? Tùy thuộc tình huống, bạn có thể chọn 1 trong các phương án được mô tả chi tiết ở các phần sau.

2 Assertions

- 1 macro hay 1 CT con dùng trong quá trình phát triển ứng dụng, cho phép CT tự kiểm tra khi chạy.
- Return true >> OK, false >> có 1 lỗi gì đó trong CT.
- VD : Nếu hệ thống cho rằng file dữ liệu về khách hàng không bao giờ vượt quá 50 000 bản ghi, CT có thể chứa 1 assertion rằng số bản ghi là <= 50 000. Khi mà số

i trong CT.

- Sử dụng assertions để ghi lại những giả thiết được đưa ra trong code và để loại bỏ những điều kiện không mong đợi. Assertions có thể đc dùng để kiểm tra các giả thiết như :
  - Các tham số đầu vào nằm trong phạm vi mong đợi (tương tự với các tham số đầu ra)
  - file hay stream đang được mở (hay đóng) khi 1 CTC bắt đầu thực hiện (hay kết thúc)
  - 1 file hay stream đang ở bản ghi đầu tiên (hay cuối cùng) khi 1 CTC bắt đầu (hay kết thúc) thực hiện
  - 1 file hay stream được mở để đọc, để ghi, hay cả đọc và ghi
  - Giá trị của 1 tham số đầu vào là không thay đổi bởi 1 CTC
  - 1 pointer là non-NULL
  - 1 mảng đc truyền vào CTC có thể chứa ít nhất X phần tử
  - 1 bảng đã đc khởi tạo để chứa các giá trị thực
  - 1 danh sách là rỗng (hay đầy) lkhi 1 CTC bắt đầu (hay kết thúc) thực hiện

- i không cần thấy các thông báo của assertion ;
- c dùng trong quá trình phát triển hay bảo dưỡng ứng dụng.
-

.

Rất nhiều NNLT hỗ trợ assertions : C++, Java và Visual Basic.

Kể cả khi NNLT không hỗ trợ, thì cũng có thể dễ ràng xd

VD:

```
#define ASSERT( condition, message ) {
    if ( !(condition) ) {
        fprintf( stderr, "Assertion %s failed: %s\n",
                 condition,message );
        exit( EXIT_FAILURE );
    }
}
```

Guidelines for Using Assertions

- ***xử lý lỗi với những điều kiện ta chờ đợi sẽ xảy ra; Dùng assertions cho các ĐK không mong đợi (không bao giờ xảy ra)***
  - -
  - o
  - bugs trong CT..
  - a PM.
- ***xử lý vào trong assertions***
  - Điều gì xảy ra khi ta turn off the assertions ?

Guidelines for Using Assertions(tt)

- ***Với các hệ thống lớn, assert, và sau đó xử lý lỗi***
  - Với 1 nguyên nhân gây lỗi xác định, hoặc là dùng assertion hoặc error-handling , nhưng không dùng cả 2. ?
  - Với các HT lớn, nhiều người cùng Pt và kéo dài 5-10 năm, hoặc hơn nữa ?
  - Cả assertions và error handling code có thể đc dùng cho cùng 1 lỗi. Ví dụ trong source code cho Microsoft Word, những điều kiện luôn trả về true thì đc dùng assertion, nhưng đồng thời cũng đc xử lý.
  - Với các hệ thống cực lớn, (VD Word) , assertions là rất có lợi vì nó giúp loại bỏ rất nhiều lỗi trong quá trình PT HT

3. Kỹ thuật xử lý lỗi (Error Handling Techniques)

- Error handling dùng để xử lý các lỗi mà ta chờ đợi sẽ xảy ra
- Tùy theo tình huống cụ thể, ta có thể trả về 1 giá trị trung lập, thay thế đoạn tiếp theo của dữ liệu hợp lệ, trả về cùng giá trị như lần trước, thay

o hay tắt máy.

Chắc chắn thay vì chính xác

- Giả sử một ứng dụng hiển thị thông tin đồ họa trên màn hình. Một vài điểm ảnh ở góc tọa độ dưới bên phải hiển thị màu sai. Ngày tiếp theo, màn hình sẽ làm mới, lỗi không còn.Phương pháp xử lý lỗi tốt nhất là gì?

- Chính xác có nghĩa là không bao giờ gặp lại lỗi
- Chắc chắn có nghĩa là phần mềm luôn chạy thông, kể cả khi có lỗi
- Ưu tiên tính chắc chắn để có tính chính xác. Bất cứ kết quả nào đó bao giờ cũng thường là tốt hơn so với Shutdown. Thỉnh thoảng trong các trình xử lý văn hiển thị một phần của một dòng văn bản ở phía dưới màn hình. Khi đó ta muốn tắt CT ? Chỉ cần nhấn trang lên hoặc xuống trang, màn hình sẽ làm mới, và sẽ hiển thị sẽ trở lại bình thường.

- Đôi khi, để loại bỏ 1 lỗi nhỏ, lại rất tốn kém. Nếu lỗi đó chắc chắn không ảnh hưởng đến mục đích cơ bản của ứng dụng, không làm CT bị die, hoặc làm sai lệch kết quả chính, người ta có thể bỏ qua, mà không cố sửa để có thể gặp phải các nguy cơ khác.
- Hiện tại có 1 dạng phần mềm “ chịu lỗi “ tức là loại phần mềm sống chung với lỗi, để đảm bảo tính liên tục, ổn định …

4. Xử lý ngoại lệ - Exceptions

- Exception : bắt các tình huống bất thường và phục hồi chúng về trạng thái trước đó.
- Dùng Ngoại lệ để thông báo cho các bộ phận khác của chương trình về lỗi không nên bỏ qua
  - Lợi ích của ngoại lệ là khả năng báo hiệu điều kiện lỗi . Phương pháp tiếp cận khác để xử lý các lỗi tạo ra khả năng mà một điều kiện lỗi có thể truyền bá thông qua một cơ sở mã không bị phát hiện. Ngoại lệ có thể loại trừ khả năng đó.
- Chỉ dùng ngoại lệ cho những điều kiện thực sự ngoại lệ
  - Exceptions đc dùng trong những tình huống giống assertions—cho các sự kiện không thường xuyên, nhưng có thể không bao giờ xảy ra
  - Exception có thể bị lạm dụng và phá vỡ các cấu trúc, điều này dễ gây ra lỗi, vì làm sai lệch luồng điều khiển

- Ví dụ : trong các PM ứng dụng, khi xử lý dữ liệu … (C#)

```
try
    {
        cmd.ExecuteNonQuery();
        ErrorsManager.SetError(ErrorIDs.KhongCoLoi);
    }
catch
    {
        ErrorsManager.SetError(ErrorIDs.SQLThatBai,
                                database.DbName,“ten_strore");
    }
```



```
VB.NET
Try
    Return CBO.FillCollection(CType(SqlHelper.ExecuteReader(ConStr,
    "TimHDon", iSoHoaDon), IDataReader),GetType(ThanhToan.ChiTietHDInfo))
 Catch ex As Exception
    mesagebox.show(ex.message)
End Try
```

```
 Dim tran As SqlTransaction
Try
    conn.Open()
     tran = conn.BeginTransaction()
     SqlHelper.ExecuteNonQuery(tran,  "ThemHDon",_
           HDInfo.SoHoaDonTC, HDInfo.TenKhach, _
           HDInfo.PhuongThucTT)
     iMaHD = GetMaHoaDon_Integer(tran)
     For Each objCT In arrDSCT
           SqlHelper.ExecuteNonQuery(tran, "ThemChiTietHD",
    objCT.ChiTiet, _
           objCT.SoTienVND,  iMaHDP)
    Next
    tran.Commit()
Catch ex As Exception
     tran.Rollback()
End Try
```

- Phục hồi tài nguyên khi xảy ra lỗi ?
  - Thường thì không phục hồi tài nguyên
  - Nhưng sẽ hữu ích khi thực hiện các công việc nhằm đảm bảo cho thông tin ở trạng thái rõ ràng và vô hại nhất có thể
  - Nếu các biến vẫn còn đc truy xuất thì chúng nên đc gán các giá trị hợp lý
  - Trường hợp thực thi việc cập nhật dữ liệu, nhất là trong 1 phiên – transaction – liên quan tới nhiều bảng chính, phụ, thì việc khôi phục khi có ngoại lệ là vô cùng cần thiết. (rollback)

5.Gỡ rối – debbuging

- Các chương trình đã viết có thể 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 !

- 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 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 …

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!!!

- **(preprocessor)**

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

Misspelled #include file

Missing */

```
$ 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
```

Understand Error Messages (tt)

(1) Hiểu đc các thông báo lỗi!!!

- ch (**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!!!

- m (**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

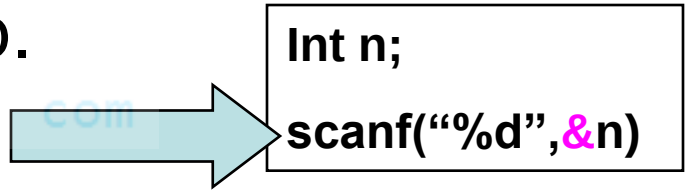
- **Trình gỡ rối :**
  - IDE : kết hợp soạn thảo, biên dịch, gỡ rối …
  - họa cho phép chạy chương trình từng bước qua từng lệnh hoặc từng hàm, dừng ở những dòng lệnh đặc biệt hay khi xuất hiện những đk đặc biệt, bên canh đó có các công cụ cho phép định dạng và hiển thị giá trị các biến, biểu thức
  - 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
  - c
    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 mỗi , phát hiện dễ ràng (Good clues, easy bugs) :**
  - 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
  - o?
  - 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 sung
  - 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*/
                outname = argv[i]; break;
        case 'f' :
               from = atoi(argv[i]); break;
        case 't' :
                to = atoi(argv[i]); break;        }
}
```


p sai vì luôn có –o ở trước tên =>gp: 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

- **t (stack trace)**
  –

t VD sau:

```
int arr[N];
qsort (arr, N, sizeof(arr[0],icmp)
```

t:

0. strcmp(0x1a2,0x1c2)[“strcmp, s”:31
1. scmp (p1=0x10001048,p2=0x1000105c)[badqs.c”:13]
2. qst(0x 10001048, 0x 10001074,0 x 400b20,0x4) [“qsort.c”:147]

….

5. _istart () [“crt…s”:13]

trcmp…

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 …**

i

(No clues, hard bugs)

- **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**

i

(No clues, hard bugs)

- **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 (last resorts-p 128)**
  - **Dùng 1 trình gỡ rối để chạy từng bước**
  - **Nhiều khi vấn đề tưởng quá đơn giản nhưng lại không phát**

    c trưng NN.

    (== và != nhỏ hơn !)
  - **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 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) ;
```

???

i (p 131)

-

Tóm lại

- Gỡ rối là 1 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 – 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 bị 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 “
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