



# Bài 6

## LẬP TRÌNH PHÒNG THỦ

---

Trịnh Thành Trung  
[trungtt@soict.hust.edu.vn](mailto:trungtt@soict.hust.edu.vn)



*1*

KHÁI NIỆM

---

# Khái niệm

- Lập trình phòng thủ - 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 chương trình 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 lạnh lùng, tàn nhẫn của dữ liệu không hợp lệ, các sự kiện mà có thể "không bao giờ" xảy ra, và các lập trình viên khác 'sai lầm'

# Invalid inputs

- Trong thực tiễn: "Garbage in, garbage out."
- Trong lập trình "rác rưởi vào – rác rưởi ra" là điều không chấp nhận
- Một chương trình tốt không bao giờ sản xuất ra rác rưởi, bất kể đầu vào là gì !
- Với 1 chương trình 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 chương trình tồi, không an toàn

# Ví dụ

```
switch(value1)
{
    case 1:
        value2 = 1;
        break;
    case 2:
        value2 = 4;
        break;
}
return(1 / value2);
```

```
for (int i = 0; i != limit; i++) {...}
for (double i = 0; i != 10.0; i += 1) // safe neu i
    khong bi thay doi trong than vong lap
for (double i = 0; i != 1.0; i += 0.1) // not ok ?
```

# Xử lý rác

1. 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 để đảm bảo rằng dữ liệu nằm trong giới hạn cho phép.
  - Hãy đảm bảo rằng giá trị số nằm trong dung sai và sâu 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 ID giao dịch tài chính hoặc một cái gì đó tương tự), hãy chắc chắn rằng các chuỗi là hợp lệ cho mục đích của nó; nếu không từ chối.
  - Nếu bạn làm việc trên 1 ứng dụng bảo mật, hãy đặc biệt lưu ý đến những dữ liệu có thể tấn công hệ thống: Cố làm tràn bộ nhớ, injected SQL commands, injected html hay XML code, tràn số ...

# Xử lý rác

## 2. 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

## 3. 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

BẢO ĐẢM

---



# Assertions

- 1 macro hay 1 chương trình con dùng trong quá trình phát triển ứng dụng , cho phép chương trình tự kiểm tra khi chạy.
- Return true >> OK, false >> có 1 lỗi gì đó trong chương trình.
- VD: Nếu hệ thống cho rằng file dữ liệu về khách hàng không bao giờ vượt quá 50 000 bản ghi, chương trình có thể chứa 1 assertion rằng số bản ghi là  $\leq 50\,000$ . Khi mà số bản ghi  $\leq 50,000$ , assertion sẽ không có phản ứng gì. Nếu đếm được hơn 50 000 bản ghi, nó sẽ “khẳng định” rằng có lỗi trong chương trình

- End users không cần thấy các thông báo của assertion
- Assertions chủ yếu được dùng trong quá trình phát triển hay bảo dưỡng ứng dụng.
- Dịch thành code khi phát triển, loại bỏ khỏi code trong sản phẩm >>performance

- Rất nhiều NNLT hỗ trợ assertions: C++, Java và Visual Basic.
- Kể cả khi NNLT không hỗ trợ, thì cũng có thể dễ dàng xây dựng

VD:

```
#define ASSERT( condition, message ) {  
    if ( !(condition) ) {  
        cout << "Assertion" << condition  
<<message;  
        exit( EXIT_FAILURE );  
    }  
}
```

# Assert definition

```
#ifndef DEBUG
#define assert(EXP) \
(void)((EXP) || ( assert(#EXP, \
    FILE , \
    LINE ), \
    0))
#else
#define assert(EXP) \
((void) 0)
#endif
void assert(const char *cond, \
    const char *fn, int ln)
{fflush(stdout);
fprintf(stderr, "%s failed at \
    File: %s, Line: %d'", cond, \
    fn, ln);
fflush(stderr);
exit(1);
}
```

```
/* -Version of strlen with \
assert- */
```

```
void strlen(char *inp) { \
    int i=0; \
    assert(inp != (char *) \
    NULL); \
    while (inp[i] != 0) { \
        i++; \
        assert(i <= \
    MAXSTRINGSIZE); \
    } \
    assert(inp[i] == 0); \
    return (i); \
}
```

# Guidelines for Using Assertions

- Sử dụng code xử lý lỗi với những điều kiện ta chờ đợi sẽ xảy ra. Dùng assertions cho các điều kiện không mong đợi (không bao giờ xảy ra)
  - Error-handling : checks for bad input data
  - Assertions : check for bugs in the code.
  - Error handling hướng tới việc xử lý lỗi, còn assertion thì hướng đến việc hiệu chỉnh chương trình , tạo ra new version of software
- Tránh đưa code xử lý vào trong assertions
  - Điều gì xảy ra khi ta turn off the assertions ?

# Guidelines for Using Assertions

- 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 hệ thống lớn, nhiều người cùng phát triển và kéo dài 5-10 năm, hoặc hơn nữa ?
  - Cả assertions và error handling code có thể được dùng cho cùng 1 lỗi. Ví dụ trong source code cho Microsoft Word, những điều kiện luôn trả về true thì được dùng assertion, nhưng đồng thời cũng được 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 phát triển hệ thống

# 3

XỬ LÝ LỖI

---

# Kỹ thuật xử lý lỗi

- 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 thế giá trị hợp lệ gần nhất
  - Ghi vết 1 cảnh báo vào tệp
  - trả về 1 mã lỗi
  - gọi 1 thủ tục hay đối tượng xử lý
  - hiện 1 thông báo 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 chương trình ? Chỉ cần nhấn PgUp hoặc PgDn, màn hình sẽ làm mới => hiển thị 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 chương trình 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.
  - Phần mềm "chịu lỗi"? : phần mềm sống chung với lỗi, để đảm bảo tính liên tục, ổn định ...

# Xử lý ngoại lệ

- 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 được 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
    messagebox.show(ex.message)
End Try
```

# FULL EXAMPLE

```
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, "ThemCTHD",
            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 được truy xuất thì chúng nên được 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 )

# 4

GỖ LỖI

---

# Gỡ rối

- Các chương trình đã viết có thể đã có nhiều lỗi? – tại sao phần mềm lại phức tạp vậy?
- Sự phức tạp của chương trình liên quan đến cách thức tương tác của các thành phần của chương trình 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 nhằm 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, phải loại bỏ bằng gỡ rối - debug!

- Ngay LTV chuyên nghiệp cũng tốn nhiều thời gian cho debug!
- 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 debug
- Debug không đơn giản, tốn thời gian => cần tránh gây ra lỗi. Các cách làm giảm thời gian debug là :
  - Thiết kế tốt
  - Phong cách lập trình tốt
  - Kiểm tra các điều kiện biên
  - Kiểm tra các “khẳng định” – assertion và tính đúng đắn trong mã nguồn
  - 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

Debug khi **build-time** dễ hơn lúc **run-time**, khi và chỉ khi ta

## (1) Hiểu các thông báo lỗi

- Một số là từ **preprocessor**

```
#include <stdiio.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: stdiio.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

## (1) Hiểu 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 các thông báo lỗi

- Một số là từ **linker**

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

Misspelled  
function name

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

Linker error: Cannot find  
definition of printf()

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

# Các phương pháp gỡ rối

- **Trình debug :**

- IDE: kết hợp soạn thảo, biên dịch, debug ...
- Các trình debug 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 điều kiện đặc biệt, bên cạnh đó có các công cụ cho phép định dạng và hiển thị giá trị các biến, biểu thức
- Trình debug có thể được kích hoạt trực tiếp khi có lỗi.
- Thường để tìm ra lỗi, ta phải xem xét thứ tự các hàm đã được kích hoạt (theo vết) và hiển thị các giá trị các biến liên quan
- Nếu vẫn không phát hiện được lỗi: dùng các BreakPoint hoặc chạy từng bước – step by step
- Có nhiều công cụ debug mạnh và hiệu quả, tại sao ta vẫn mất nhiều thời gian và trí lực để debug?
- Nhiều khi các công cụ không thể giúp dễ dàng tìm lỗi, nếu đưa ra 1 câu hỏi sai, trình debug 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 dàng :
  - 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 chương trình
  - Rất may là hầu hết các lỗi thường đơn giản và dễ tìm. Hãy khảo sát các đầu mối của việc xuất ra kết quả có lỗi và cố gắng suy ra nguyên nhân gây ra nó
  - Khi có được 1 số thông tin về lỗi và nơi xảy ra lỗi, hãy tạm dừng để nghĩ xem lỗi xảy ra như thế nào.
  - Suy luận ngược trở lại trạng thái của chương trình bị hỏng để xác định nguyên nhân gây ra lỗi
  - Debug liên quan đến việc lập luận lùi, giống như tìm kiếm các bí mật của 1 vụ án. 1 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 1 số vấn đề khác

# Các phương pháp gỡ rối

- 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.



# Các phương pháp gỡ rối

- Kiểm tra sự 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)

# Các phương pháp gỡ rối

- Debug 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 điều kiện để suy nghĩ
  - Thậm chí có thể giải thích cho người không phải LTV
  - Extreme 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 lầm ...

# 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 được, thì thử tìm nguyên nhân tại sao lại không được
- 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 được các vấn đề đã xảy ra, và giải quyết các vấn đề tương tự sau này, cũng như khi chuyển giao chương trình cho người khác..

# Hiển thị kết quả

- In các giá trị tại các điểm nhạy cảm

- 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 kết quả

- Maybe even better:

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

In debugging output ra **stderr**; debugging output có thể tách biệt với các in ấn của chương trình

- 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 debug 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 và 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 chương trình: không phải do thuật toán, mà do thông tin bị thay đổi qua mỗi lần chạy
  - Các biến đã được 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

- 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

# Những lỗi thường gặp với 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 không 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 dọn)
10. Infinite loop - lặp vô hạn
11. Global variables used – dùng biến tổng thể

12. IF-ELSE not used properly – dùng if-else không chuẩn
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. Missing 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 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

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
```

# Các kỹ thuật viết code chất lượng


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

- Đa năng hóa toán tử  $+$ ,  $*$ ,  $+=$ ,  $*=$  của lớp Matrix



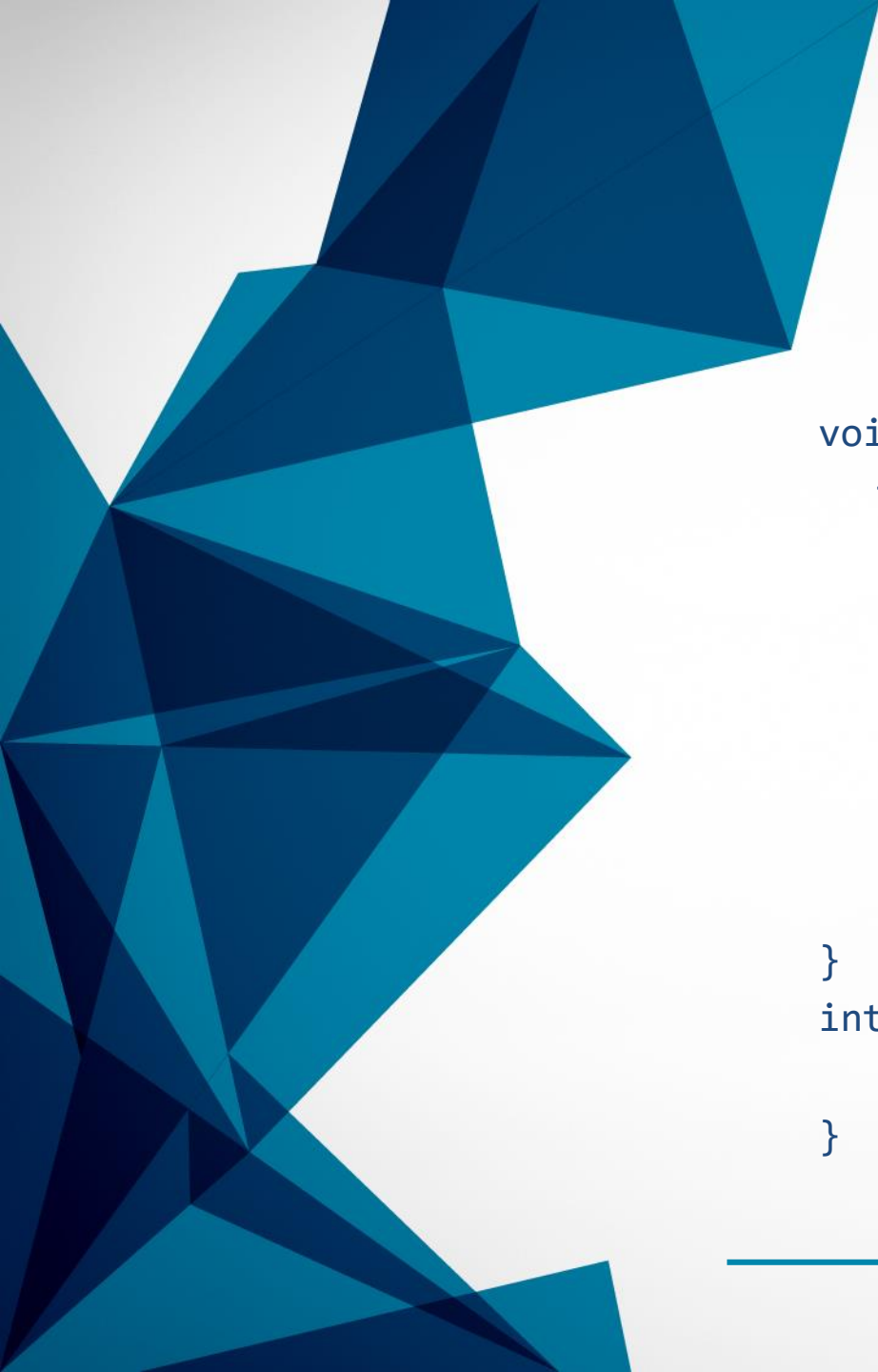
```
#include <iostream.h>
class Matrix {
    private:  int R,C;
              double  **Data;
    public:
    Matrix(int M=2,int N=2, double V=0);
    ~Matrix();
    void Print() const;
    double  & operator () (int M,int N);
    Matrix operator +(Matrix m);
    Matrix operator +=(Matrix m);
    Matrix operator *(Matrix m);
    Matrix operator *=(Matrix m);
    void Nhaps1();
};
```

---



```
Matrix::Matrix(int M,int N,int V) { //  
    Hàm khởi tạo  
    int I,J;  
    R=M;  
    C=N;  
    Data = new double *[R];  
    double *Temp=new double[R*C];  
    for(I=0;I<R;++I) {  
        Data[I]=Temp;  
        Temp+=C;  
    }  
    for(I=0;I<R;++I)  
    for(J=0;J<C;++J)  
        Data[I][J]=V;  
}  
Matrix::~~Matrix() { // Hàm hủy  
    delete [] Data[0];  
    delete [] Data; }
```

---



```
void Matrix::Print() const {  
    for(int I=0;I<R;++I) {  
        for(int J=0;J<C;++J) {  
            cout.width(5);  
            // canh lề phải chiều dài 5 ký tự  
            cout<<Data[I][J];  
        }  
        cout<<endl;  
    }  
}  
  
int & Matrix::operator () (int M,int N) {  
    return Data[M][N];  
}
```

---

```

Matrix Matrix::operator +(Matrix m) {
    if(R !=m.R || C !=m.C) {
        cout << “Không thể cộng 2 MT \n”;
        return 1;
    }
    Matrix T(R,C);
    for ( int i=0;i<R;i++)
        for (int j=0;j<C;j++)
            T.Data[i][j] =
                Data[i][j]+m.Data[i][j];
    return T;
}

```

```

Matrix operator +=(Matrix m) {
    ???


```

Tại sao toán tử += phải trả về 1 matrix ?  
 Khi trả về 1 đối tượng có thành viên dữ liệu đã cập nhật xong, điều gì sẽ xảy ra ??? Cần hoàn thiện chương trình thế nào ? ( định nghĩa lại toán tử gan )

```

...
}

```



```
Matrix Matrix::operator *(Matrix m) {
    if(C !=m.R ) {
        cout << “ Không thể nhân 2 MT \n”;
        return 1;
    }
    Matrix T(R,m.C);
    for ( int i=0;i<R;i++)
        for (int j=0;j<m.C;j++)
            for (int k =0;k<C;k++)
                T.Data[i][j] += Data[i][k] *
                m.Data[k][j];
    return T;
}
Matrix Matrix::operator *(Matrix m) {...}
void Matrix::nhaps1( ){...}
.....
```

---