

$$(3LT - 2BT)$$

Efficient Programs

- Trước hết là giải thuật
 - Hãy dùng giải thuật hay nhất có thể
 - Sau đó hãy nghĩ tới việc tăng tính hiệu quả của code
 - Ví dụ : Tính tổng của n số tự nhiên kể từ m

```
void main() {  
    long n,m,i , sum ;  
    cout << ' vào n ' ; cin << n;  
    cout << ' vào m ' ; cin << m;  
    sum =0;  
    for(i = m ; i < m+n; i++)  
        sum += i;  
    cout << ' Tổng = ' <<sum;  
}
```

```
void main()  
{  
    long n,m , sum ;  
    cout << ' vào n ' ; cin << n;  
    cout << ' vào m ' ; cin << m;  
    sum =(m + m+ n-1) * n / 2;  
    cout << ' Tổng = ' <<sum;  
}  
//TD m=3, n=4 => KQ = 18!
```

Dùng chỉ thị chương trình dịch

- Một số Chương trình dịch có vai trò rất lớn trong việc tối ưu chương trình.
 - Chúng phân tích sâu mã nguồn và làm mọi điều “machinely” có thể.
 - Ví dụ GNU g++ compiler trên Linux/Cygwin cho chương trình viết bằng c:
 - g++ -O5 -o myprog myprog.c
- có thể cải thiện hiệu năng từ 10% đến 300%

Nhưng...

- Bạn vẫn có thể thực hiện những cải tiến mà trình dịch không thể.
- Bạn phải loại bỏ tất cả những chỗ bất hợp lý trong code:
 - Làm cho chương trình hiệu quả nhất có thể
- Có thể phải xem lại khi thấy chương trình chạy chậm.

Vậy cần tập trung vào đâu để cải tiến nhanh nhất, tốt nhất ?

Writing Efficient Code

- Xác định nguồn gây kém hiệu quả:
 - Dư thừa tính toán - redundant computation
 - Chủ yếu
 - Trong các procedures
 - Các vòng lặp : Loops

Khởi tạo 1 lần, dùng nhiều lần

- Before

```
float f()
{ double value = sin(0.25);
  //
  .....
}
```

- After

```
double defaultValue = sin(0.25);

float f()
{ double value = defaultValue;
  //
  .....
}
```

Inline functions

- **Nếu 1 hàm trong c++ chỉ gồm những lệnh đơn giản, không có for, while .. thì có thể khai báo inline.**
 - **Inline code sẽ được chèn vào bất cứ chỗ nào hàm được gọi.**
 - **Chương trình sẽ lớn hơn chút ít**
 - **Nhưng nhanh hơn , không dùng stack – 4 bước khi 1 hàm được gọi ...**

Inline functions

```
#include <iostream>
#include <cmath>
using namespace std;
inline double hypotenuse (double a, double b)
{
    return sqrt (a * a + b * b);
}

int main () {
    double k = 6, m = 9;
    // 2 dòng sau thực hiện như nhau:
    cout << hypotenuse (k, m) << endl;
    cout << sqrt (k * k + m * m) << endl;
    return 0;
}
```


Static Variables

- Kiểu dữ liệu Static tham chiếu tới **global hay 'static' variables** , chúng được cấp phát bộ nhớ khi dịch compile-time.

```
int int_array[100];
int main() {
    static float float_array[100];
    double double_array[100];
    char *pchar;
    pchar = (char *)malloc(100);
    /* .... */
    return (0);
}
```

Static Variables

- Các biến khai báo trong CT con được cấp phát bộ nhớ khi CT con được gọi và sẽ được giải phóng khi CT con kết thúc.
- Khi gọi lại CT con, các biến cục bộ lại được cấp phát và khởi tạo lại, ...
- Nếu muốn 1 giá trị vẫn **được lưu lại cho đến khi kết thúc toàn chương trình**, cần khai báo biến cục bộ của CT con đó là **static** và khởi tạo cho nó 1 giá trị.
 - Việc khởi tạo sẽ chỉ thực hiện lần đầu tiên chương trình được gọi và giá trị sau khi biến đổi sẽ được lưu cho các lần gọi sau.
 - Bằng cách này một ct con có thể “nhớ” một vài mẫu tin sau mỗi lần được gọi.
- Dùng biến Static thay vì Global :
 - Cái hay của một biến **static** là nó là của CT con, => tránh được các side effects.

Macros

–

```
#define max(a,b) (a>b?a:b)
```

- Các hàm Inline cũng giống như macros vì cả 2 được khai triển khi dịch (compile time)
 - macros được khai triển bởi preprocessor, còn inline functions được truyền bởi compiler.
- Tuy nhiên có nhiều điểm khác biệt:
 - Inline functions tuân thủ các thủ tục như 1 hàm bình thường.
 - p như các hàm khác, song có thêm từ khóa **inline** khi khai báo hàm.
 - Các biểu thức truyền như là đối số cho inline functions được tính 1 lần. Trong 1 số trường hợp, biểu thức truyền như tham số cho macros có thể được tính lại nhiều hơn 1 lần.
 - Bạn không thể gỡ rối cho macros, nhưng với inline functions thì có thể.

Tính toán trước các giá trị

- Nếu bạn phải tính đi tính lại 1 biểu thức, thì nên tính trước 1 lần và lưu lại giá trị, rồi dùng giá trị ấy sau này

```
int f(int i) {  
    if (i < 10 && i >= 0)  
    {  
        return i * i - i;  
    }  
    return 0;  
}
```

```
static int[] values =  
    {0, 0, 2, 3*3-3, ..., 9*9-  
9};  
int f(int i) {  
    if (i < 10 && i >= 0)  
        return values[i];  
    return 0; }
```

Loại bỏ những biểu thức “thông thường”

- Đừng tính cùng một biểu thức nhiều lần!
- Một số compilers có thể nhận biết và xử lý.

```
for (i = 1; i<=10;i++) x += strlen(str);  
Y = 15 + strlen(str);
```


```
len = strlen(str);  
for (i = 1;i<=10;i++) x += len;  
Y = 15 + len;
```

Sử dụng các biến đổi số học!

- Trình dịch không thể tự động xử lý

```
if (a > sqrt(b))  
    x = a*a + 3*a + 2;
```

p nhân hơn!



```
if (a * a > b)  
    x = (a+1)*(a+2);
```

Dùng “lính canh” -Tránh những kiểm tra không cần thiết

- Trước

```
char s[100], searchValue;  
int pos,tim, size ;  
..... Gán giá trị cho s, searchValue  
...  
size = strlen(s);  
pos = 0;  
while (pos < size) && (s[pos] != searchValue)  
do pos++;  
If (pos >= size) tim =0 else  
tim = 1;
```

Dùng “lính canh”

- Ý tưởng chung
 - Đặt giá trị cần tìm vào cuối xâu: “ *nh canh*”
 - Luôn tìm thấy !
 -

m thấy !

```
size = strlen(s);  
strcat(s, searchValue);  
pos = 0;  
while ( s[pos] != searchValue)  
do pos++;  
If (pos >= size) tim = 0 else  
tim = 1;
```

Có thể làm tương tự với mảng, danh sách ...

Dịch chuyển những biểu thức bất biến ra khỏi vòng lặp

- Đùng lặp các biểu thức tính toán không cần thiết
- Một số Compilers có thể tự xử lý!

```
for (i =0; i<100;i++)  
    plot(i, i*sin(d));
```

```
M = sin(d);  
for (i =0; i<100;i++)  
    plot(i, i*M);
```

Không dùng các vòng lặp ngắn

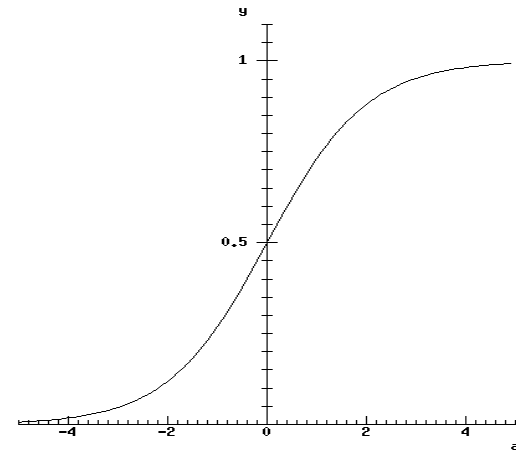
```
for (i =j; i<= j+3;i++)  
    sum += q*i -i*7;
```

```
i = j;  
sum += q*i -i*7;  
i ++;  
sum += q*i -i*7;  
i ++;  
sum += q*i -i*7;
```

Giảm thời gian tính toán

- Trong mô phỏng Neural Network người ta thường dùng hàm có tên **sigmoid**
- Với X dương lớn ta có $\text{sigmoid}(x) \cong 1$
- Với x âm “lớn” $\text{sigmoid}(x) \cong 0$

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-kx}}$$



Tính Sigmoid

```
float sigmoid (float x )  
{  
    return 1.0 / (1.0 + exp(-x))  
};
```

Tính Sigmoid

- Hàm $\exp(-x)$ mất rất nhiều thời gian để tính!
 - Những hàm kiểu này người ta phải dùng khai triển chuỗi
 - Chuỗi Taylor /Maclaurin
 - Tính tổng các số hạng dạng $((-x)^n / n!)$
 - Mỗi số hạng lại dùng các phép toán với số chấm động
- Nói chung các mô phỏng neural network gọi hàm này **trăm triệu lần** trong mỗi lần thực hiện.
- Chính vì vậy , $\text{sigmoid}(x)$ chiếm phần lớn thời gian (khoảng 70-80%)

Tính Sigmoid – Giải pháp

- Thay vì tính hàm mọi lúc, người ta:
 - Tính hàm tại N điểm và xây dựng 1 mảng.
 - Trong mỗi lần gọi sigmoid
 - Tìm giá trị gần nhất của x và kết quả ứng với giá trị ấy
 - Thực hiện nội suy tuyến tính - linear interpolation

x_0 sigmoid(x_0)
 x_1 sigmoid(x_0)
 x_2 sigmoid(x_0)
 x_3 sigmoid(x_0)
 x_4 sigmoid(x_0)
 x_5 sigmoid(x_0)
 x_6 sigmoid(x_0)
.
.
.
 x_{99} sigmoid(x_{99})

Tính Sigmoid (tiếp)

← if (x < x0) return (0.0);

x₀ sigmoid(x₀)

x₁ sigmoid(x₀)

x₂ sigmoid(x₀)

x₃ sigmoid(x₀)

x₄ sigmoid(x₀)

x₅ sigmoid(x₀)

x₆ sigmoid(x₀)

.

.

.

x₉₉ sigmoid(x₉₉) ← if (x > x₉₉) return (1.0);

Tính Sigmoid (tiếp)

- Chọn số các điểm ($N = 1000, 10000, \text{v.v.}$) tùy theo độ chính xác mà bạn muốn
 - Tồn kém thêm không gian bộ nhớ cho mỗi điểm là 2 float hay double tức là 8 – 16 bytes/điểm
- Khởi tạo giá trị cho mảng khi bắt đầu thực hiện.

Tính Sigmoid (tiếp)

- Bạn đã biết X_0
 - Tính Delta = $X_1 - X_0$
 - Tính $X_{\max} = X_0 + N * \text{Delta}$;
- Với X đã cho
 - Tính $i = (X - X_0) / \text{Delta}$;
 - 1 phép trừ số thực và 1 phép chia số thực
 - Tính $\text{sigmoid}(x)$
 - 1 phép nhân float và 1 phép cộng float

Kết quả

- Nếu dùng $\exp(x)$:
 - Mỗi lần gọi mất khoảng **300 nanoseconds** với 1 máy Pentium 4 tốc độ 2 Ghz.
- Dùng tìm kiếm trên mảng và nội suy tuyến tính :
 - Mỗi lần gọi mất khoảng **30 nanoseconds**
- Tốc độ tăng gấp 10 lần
 - **Đổi lại phải tốn kém thêm từ 64K -> 640 K bộ nhớ.**

Lưu ý !

- Với phần lớn các chương trình, việc tăng tốc độ thực hiện là cần thiết
- Tuy nhiên, cố tăng tốc độ cho những đoạn code không sử dụng thường xuyên là vô ích !

Những quy tắc cơ bản Fundamental Rules

- Đơn giản hóa Code – Code Simplification :
 - Hầu hết các chương trình chạy nhanh là đơn giản. Vì vậy, hãy đơn giản hóa chương trình để nó chạy nhanh hơn. Tuy nhiên...
- Đơn giản vấn đề - Problem Simplification:
 - Để tăng hiệu quả của chương trình, hãy đơn giản hóa vấn đề mà nó giải quyết.
- Không ngừng nghi ngờ - Relentless Suspicion:
 - Đặt dấu hỏi về sự cần thiết của mỗi mẫu code và mỗi trường , mỗi thuộc tính trong cấu trúc dữ liệu.
- Liên kết sớm - Early Binding:
 - Hãy thực hiện ngay công việc để tránh thực hiện nhiều lần sau này.

Quy tắc tăng tốc độ

Có thể tăng tốc độ bằng cách sử dụng thêm bộ nhớ (mảng).

- **Dùng thêm các dữ liệu có cấu trúc:**

- Thời gian cho các phép toán thông dụng có thể giảm bằng cách sử dụng thêm các cấu trúc dữ liệu với các dữ liệu bổ sung hoặc bằng cách thay đổi các dữ liệu trong cấu trúc sao cho dễ tiếp cận hơn.

- **Lưu các kết quả được tính trước:**

- Thời gian tính toán lại các hàm có thể giảm bớt bằng cách tính toán hàm chỉ 1 lần và lưu kết quả, những yêu cầu sau này sẽ được xử lý bằng cách tìm kiếm từ mảng hay danh sách kết quả thay vì tính lại hàm.

- ...

Quy tắc tăng tốc độ : cont.

- **Caching:**

- Dữ liệu thường dùng cần phải dễ tiếp cận nhất, luôn hiện hữu.

- **Lazy Evaluation:**

- Không bao giờ tính 1 phần tử cho đến khi cần để tránh những sự tính toán không cần thiết.

Quy tắc lặp : Loop Rules

Những điểm nóng - Hot spots trong phần lớn các chương trình đến từ các vòng lặp:

- **Đưa Code ra khỏi các vòng lặp:**
 - Thay vì thực hiện việc tính toán trong mỗi lần lặp, tốt nhất thực hiện nó chỉ một lần bên ngoài vòng lặp- nếu được.
- **Kết hợp các vòng lặp – loop fusion:**
 - Nếu 2 vòng lặp gần nhau cùng thao tác trên cùng 1 tập hợp các phần tử thì cần kết hợp chung vào 1 vòng lặp.

Quy tắc lặp : Loop Rules

- **Kết hợp các phép thử - Combining Tests:**
 - Trong vòng lặp càng ít kiểm tra càng tốt và tốt nhất chỉ một phép thử. LTV có thể phải thay đổi điều kiện kết thúc vòng lặp. “Lính canh (sentinel)” là một ví dụ cho quy tắc này.
- **Loại bỏ Loop :**
 - Với những vòng lặp ngắn thì cần loại bỏ vòng lặp, tránh phải thay đổi và kiểm tra điều kiện lặp.

Procedure Rules

- Khai báo những hàm ngắn và đơn giản (thường chỉ 1 dòng) là inline
 - **Tránh phải thực hiện 4 bước khi hàm được gọi**
 - **Tránh dùng bộ nhớ stack**

GOOD PROGRAMMING STYLE

Plauger. Cần lưu ý rằng các quy tắc của “programming style”, giống như quy tắc văn phạm English, đôi khi bị vi phạm, thậm chí bởi những nhà văn hay nhất. Tuy nhiên khi 1 quy tắc bị vi phạm, thì thường được bù lại bằng một cái gì đó, đáng để ta mạo hiểm.

Nói chung sẽ là tốt nếu ta tuân thủ các quy tắc sau đây :

- “Tính nhất quán”. Nếu bạn chấp nhận một cách thức đặt tên hàm hay biến, hằng thì hãy tuân thủ nó trong toàn bộ chương trình.
- Đầu mỗi CT, nên có một đoạn chú thích ...
- Mỗi CT con phải có một nhiệm vụ rõ ràng. Một CT con phải đủ ngắn để người đọc có thể nắm bắt như một đơn vị, 1 chức năng.
- Hãy dùng tối thiểu số các tham số của CT con. **> 6 tham số cho 1 CT con là quá nhiều.**

GOOD PROGRAMMING STYLE

- Có 2 loại Ct con : functions và procedures. Functions chỉ nên tác động tới duy nhất 1 giá trị - giá trị trả về của hàm.
- Không nên thay đổi giá trị của biến chạy trong thân của vòng lặp for, ví dụ không nên làm như sau :

```
for (i=1; i<=10; i++) i++;
```

- có cùng tên. Nếu “i” được dùng làm biến chạy cho vòng lặp trong 1 CT con, thì đừng dùng nó cho việc khác trong các CT con khác.

GOOD PROGRAMMING STYLE

1. *Write clearly / don't be too clever – Viết rõ ràng – đừng quá thông minh (kỳ bí)*
2. *Say what you mean, simply and directly – Trình bày vấn đề 1 cách đơn giản, trực tiếp*
3. *Use library functions whenever feasible. – Sử dụng thư viện mọi khi có thể*
4. *Avoid too many temporary variables – Tránh dùng nhiều biến trung gian*

—
ng cho hiệu quả

GOOD PROGRAMMING STYLE

6. *Let the machine do the dirty work* – Hãy để máy tính làm những việc nặng nhọc của nó. (tính toán ...)
7. *Replace repetitive expressions by calls to common functions.* – Hãy thay những biểu thức lặp đi lặp lại bằng cách gọi các hàm
8. *Parenthesize to avoid ambiguity.* – Dùng () để tránh rắc rối
9. *Choose variable names that won't be confused* – Chọn tên biến sao cho tránh được lẫn lộn
10. *Avoid unnecessary branches.* – Tránh các nhánh không cần thiết
11. *If a logical expression is hard to understand, try transforming it* – Nếu 1 biểu thức logic khó hiểu, cố gắng chuyển đổi cho đơn giản
12. *Choose a data representation that makes the program simple* – Hãy lựa chọn cấu trúc dữ liệu để chương trình thành đơn giản

GOOD PROGRAMMING STYLE

13. *Write first in easy-to-understand pseudo language; then translate into whatever language you have to use. – Trước tiên hãy viết ct bằng giả ngữ dễ hiểu, rồi hãy chuyển sang ngôn ngữ cần thiết.*
14. *Modularize. Use procedures and functions. – Mô đul hóa. Dùng các hàm và thủ tục*
15. *Avoid gotos completely if you can keep the program readable. – Tránh hoàn toàn việc dùng goto*
16. *Don't patch bad code /{ rewrite it. – Không chấp vá mã xấu – Viết lại đoạn code đó*
17. *Write and test a big program in small pieces. – Viết và kiểm tra 1 CT lớn thành từng CT con*

GOOD PROGRAMMING STYLE

18. *Use recursive procedures for recursively-defined data structures. – Hãy dùng các thủ tục đệ qui cho các cấu trúc dữ liệu đệ qui.*
19. *Test input for plausibility and validity. – Kiểm tra đầu vào để đảm bảo tính chính xác và hợp lệ*
20. *Make sure input doesn't violate the limits of the program. – Hãy đảm bảo đầu vào không quá giới hạn cho phép của CT*
21. *Terminate input by end-of-file marker, not by count. – Hãy kết thúc dòng nhập bằng ký hiệu EOF, không dùng phép đếm*
22. *Identify bad input; recover if possible. – Xác định đầu vào xấu, khôi phục nếu có thể*
23. *Make input easy to prepare and output self-explanatory. – Hãy làm cho đầu vào đơn giản, dễ chuẩn bị và đầu ra dễ hiểu*

GOOD PROGRAMMING STYLE

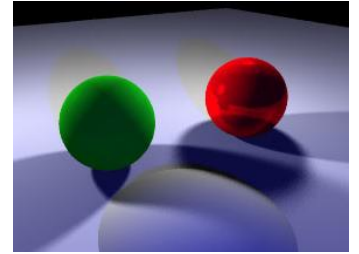
- 24. *Use uniform input formats.* – Hãy dùng các đầu vào theo các định dạng chuẩn.
- 25. *Make sure all variable are initialized before use.* – Hãy đảm bảo các biến được khởi tạo trước khi sử dụng
- 26. *Test programs at their boundary values.* – Hãy kiểm tra CT tại các cận
- 26. *Check some answers by hand.* – Kiểm tra 1 số câu trả lời bằng tay
- 27. *10.0 times 0.1 is hardly ever 1.0.* – 10 nhân 0.1 không chắc đã = 1.0
- 28. *7/8 is zero while 7.0/8.0 is not zero. 7/8 = 0 nhưng 7.0/8.0 \neq 0 ?*
- 29. *Make it right before you make it faster.* – Hãy làm cho CT chạy đúng, trước khi làm nó chạy nhanh

GOOD PROGRAMMING STYLE

- 30. *Make it clear before you make it faster.* – Hãy viết code rõ ràng, trước khi làm nó chạy nhanh
- 31. *Let your compiler do the simple optimizations.* – Hãy để trình dịch thực hiện các việc tối ưu hóa đơn giản
- 32. *Don't strain to re-use code; reorganize instead.* – Đừng cố tái sử dụng mã, thay vì vậy, hãy tổ chức lại mã
- 33. *Make sure special cases are truly special.* – Hãy đảm bảo các trường hợp đặc biệt là thực sự đặc biệt
- 34. *Keep it simple to make it faster.* – Hãy giữ nó đơn giản để làm cho nó nhanh hơn
- 35. *Make sure comments and code agree.* – Chú thích phải rõ ràng, sát code
- 36. *Don't comment bad code | rewrite it.* – Đừng chú thích những đoạn mã xấu, hãy viết lại
- 37. *Use variable names that mean something.* – Hãy dùng các tên biến có nghĩa
- 38. *Format a program to help the reader understand it.* – Hãy định dạng CT để giúp người đọc hiểu đc CT
- 39. *Don't over-comment.* – Đừng chú thích quá nhiều

Program Style

- Who reads your code?
 - The compiler
 - Other programmers



This is a working ray tracer! (courtesy of Paul Heckbert)

Program Style

- Vì sao program style lại quan trọng?
 - Lỗi thường xảy ra do sự nhầm lẫn của LTV
 - Biến này được dùng làm gì?
 - Hàm này được gọi như thế nào?
 - Good code = code dễ đọc
- Làm thế nào để code thành dễ đọc?
 - Cấu trúc chương trình rõ ràng, dễ hiểu, khúc triết
 - Sử dụng thành ngữ phổ biến - idiom
 - Chọn tên phù hợp, gợi nhớ
 - Viết chú thích rõ ràng
 - Sử dụng môdul

Structure: Spacing

- Use readable/consistent spacing
 - VD: Gán mỗi phần tử mảng $a[j] = j$.
 - Bad code

```
for (j=0;j<100;j++) a[j]=j;
```

- Good code

```
for (j=0; j<100; j++)  
    a[j] = j;
```

- Thường có thể dựa vào auto-indenting, tính năng trong trình soạn thảo

Structure: Indentation (cont.)

- Use readable/consistent indentation

–VD:

```
if (month == FEB) {  
    if (year % 4 == 0)  
        if (day > 29)  
            legal = FALSE;  
    else  
        if (day > 28)  
            legal = FALSE;  
}
```

Wrong code
(else của “if day > 29”)

```
if (month == FEB) {  
    if (year % 4 == 0) {  
        if (day > 29)  
            legal = FALSE;  
    }  
    else {  
        if (day > 28)  
            legal = FALSE;  
    }  
}
```

Right code

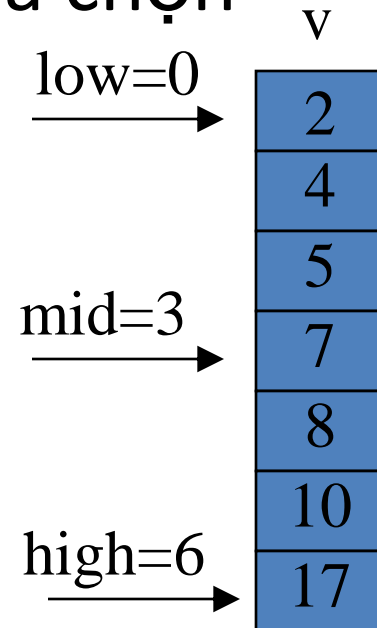
Structure: Indentation (cont.)

- Use “else-if” cho cấu trúc đa lựa chọn

–VD: Bước so sánh trong tìm kiếm nhị phân - binary search.

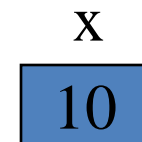
–Bad code

```
if (x < v[mid])  
    high = mid - 1;  
else  
    if (x > v[mid])  
        low = mid + 1;  
    else  
        return mid;
```



–Good code

```
if (x < v[mid])  
    high = mid - 1;  
else if (x > v[mid])  
    low = mid + 1;  
else  
    return mid;
```



Structure: “Paragraphs”

thành các phần

```
...
#include <stdio.h>
#include <stdlib.h>

int main(void)

/* Read a circle's radius from stdin, and compute and write its
   diameter and circumference to stdout.  Return 0 if successful. */

{
    const double PI = 3.14159;
    int radius;
    int diam;
    double circum;

    printf("Enter the circle's radius:\n");
    if (scanf("%d", &radius) != 1)
    {
        fprintf(stderr, "Error: Not a number\n");
        exit(EXIT_FAILURE); /* or:  return EXIT_FAILURE; */
    }
}
```

...

Structure: “Paragraphs”

- Dùng dòng trống để chia code thành các phần chính

```
diam = 2 * radius;  
circum = PI * (double)diam;  
  
printf("A circle with radius %d has diameter %d\n",  
      radius, diam);  
printf("and circumference %f.\n", circum);  
  
return 0;  
}
```


Structure: Expressions

- Dùng các biểu thức dạng nguyên bản
 - VD: Kiểm tra nếu n thỏa mãn $j < n < k$
 - Bad code

```
if (! (n >= k) && ! (n <= j))
```
 - Good code

```
if ((j < n) && (n < k))
```
 - BT điều kiện có thể đọc như cách thức bạn viết thông thường
 - Đừng viết BT điều kiện theo kiểu mà bạn không bao giờ sử dụng

Structure: Expressions (cont.)

- Dùng () để tránh nhầm lẫn
 - VD: Kiểm tra nếu n thỏa mãn $j < n < k$
 - Moderately bad code

```
if (j < n && n < k)
```

- Moderately better code

```
if ((j < n) && (n < k))
```

- Nên nhóm các nhóm một cách rõ ràng
 - Toán tử quan hệ (vd ">") có độ ưu tiên cao hơn các toán tử logic (vd "&&"), **nhưng ai nhớ điều đó ?**

Structure: Expressions (cont.)

- Dùng () để tránh nhầm lẫn (cont.)
 - VD: đọc và in các ký tự cho đến cuối tệp.
 - Wrong code (điều gì xảy ra ???)

```
while (c = getchar() != EOF)
    putchar(c);
```

- Right code

```
while ((c = getchar()) != EOF)
    putchar(c);
```

- Nên nhóm các nhóm một cách rõ ràng :

- *Toán tử Logic ("!=") có độ ưu tiên cao hơn toán tử gán ("=")*

Structure: Expressions (cont.)

- Đơn giản hóa các biểu thức phức tạp
 - VD: Xác định các ký tự tương ứng với các tháng của năm
 - Bad code

```
if ((c == 'J') || (c == 'F') || (c ==  
'M') || (c == 'A') || (c == 'S') || (c  
== 'O') || (c == 'N') || (c == 'D'))
```

- Good code

```
if ((c == 'J') || (c == 'F') ||  
    (c == 'M') || (c == 'A') ||  
    (c == 'S') || (c == 'O') ||  
    (c == 'N') || (c == 'D'))
```

Nên sắp xếp các cơ cấu so sánh

C Idioms

- Chú ý khi dùng ++, --
 - VD: Set each array element to 1.0.
 - Bad code (or perhaps just “so-so” code)

```
i = 0;  
while (i <= n-1)  
    array[i++] = 1.0;
```

- Good

```
for (i=0; i<n; i++)  
    array[i] = 1.0;
```

Naming

- Dùng tên gợi nhớ, có tính miêu tả cho các biến và hàm
 - VD : hovaten , **CONTROL** , **CAPACITY**
- Dùng tên nhất quán cho các biến cục bộ
 - VD, **i** (not **arrayIndex**) cho biến chạy vòng lặp
- Dùng chữ hoa, chữ thường nhất quán
 - VD., Buffer_Insert (Tên hàm)
CAPACITY (hằng số)
buf (biến cục bộ)
- Dùng phong cách nhất quán khi ghép từ
 - VD., **frontsize**, **FrontSize**, **front_size**
- Dùng động từ cho tên hàm
 - VD., docsolieu () , inkq () , **Check_Octal ()** , ...

Comments

- Làm chủ ngôn ngữ
 - Hãy để chương trình tự diễn tả bản thân
 - Rồi...
- Viết chú thích để thêm thông tin
`i++; /* add one to i */`
- Chú thích các đoạn (“paragraphs”) code, đừng chú thích từng dòng
 - vd., “Sort array in ascending order”
- Chú thích dữ liệu tổng thể
 - Global variables, structure type definitions,
- Viết chú thích tương ứng với code!!!
 - Và thay đổi khi bản thân code changes. 😊

Comments (cont.)

- đoạn (“paragraphs”) không chú

thích từ dòng code

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main(void)
```

```
/* Read a circle's radius from stdin, and compute and write its
   diameter and circumference to stdout.  Return 0 if successful. */
```

```
{
```

```
    const double PI = 3.14159;
```

```
    int radius;
```

```
    int diam;
```

```
    double circum;
```

```
    /* Read the circle's radius. */
```

```
    printf("Enter the circle's radius:\n");
```

```
    if (scanf("%d", &radius) != 1)
```

```
    {
```

```
        fprintf(stderr, "Error: Not a number\n");
```

```
        exit(EXIT_FAILURE); /* or: return EXIT_FAILURE; */
```

```
    }...
```


Comments (cont.)

```
/* Compute the diameter and circumference. */  
diam = 2 * radius;  
circum = PI * (double)diam;  
  
/* Print the results. */  
printf("A circle with radius %d has diameter  
%d\n",  
    radius, diam);  
printf("and circumference %f.\n", circum);  
  
return 0;  
}
```

Function Comments

- Mô tả **những gì cần thiết để** gọi hàm 1 cách chính xác
 - Mô tả **Hàm làm gì**, chứ không phải **nó làm như thế nào**
 - Bản thân Code phải rõ ràng, dễ hiểu để biết cách nó làm việc...
 - Nếu không, hãy viết chú thích bên trong định nghĩa hàm
- Mô tả **đầu vào**: Tham số truyền vào, đọc file gì, biến tổng thể được dùng
- Mô tả **outputs**: giá trị trả về, tham số truyền ra, ghi ra files gì, các biến tổng thể mà nó tác động tới

Function Comments (cont.)

- Bad function comment

```
/* decomment.c */
```

```
int main(void) {
```

```
/* Đọc 1 ký tự. Dựa trên ký tự ấy và trạng  
thái DFA hiện thời, gọi hàm xử lý trạng thái  
tương ứng. Lặp cho đến hết tệp end-of-file.  
*/
```

```
...
```

```
}
```

–Describes **how the function *works***

Function Comments (cont.)

- Good function comment

```
/* decomment.c */
```

```
int main(void) {
```

```
    /* Đọc 1 CT C qua stdin.
```

```
       Ghi ra stdout với mỗi chú thích thay bằng 1 dấu  
cách.
```

```
       Trả về 0 nếu thành công, EXIT_FAILURE nếu không  
thành công. */
```

```
    ...  
}
```

–Describes **what the function *does***

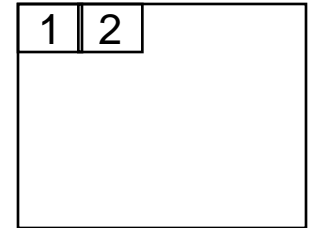
Modularity!!!

- Chương trình lớn viết khó hơn Ct nhỏ
- Trừu tượng hóa là chìa khóa để xử lý sự phức tạp
 - Abstraction cho phép LTV biết code làm gì, mà không cần biết làm như thế nào
- Ví dụ : hàm ở mức trừu tượng
 - Hàm sắp xếp 1 mảng các số nguyên
 - Character I/O functions : **getchar()** and **putchar()**
 - Mathematical functions : **sin(x)** and **sqrt(x)**

Bottom-Up Design is Bad

- Bottom-up design ☹️

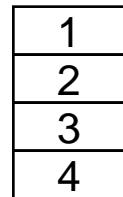
- Thiết kế chi tiết 1 phần
- Thiết kế chi tiết 1 phần khác
- Lặp lại cho đến hết



- Bottom-up design in programming

- Viết phần đầu tiên của CT 1 cách chi tiết cho đến hết
- Viết phần tiếp theo của CT 1 cách chi tiết cho đến hết
- Lặp lại cho đến hết

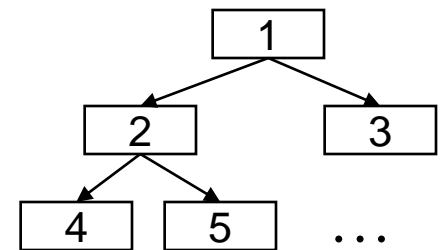
...



...

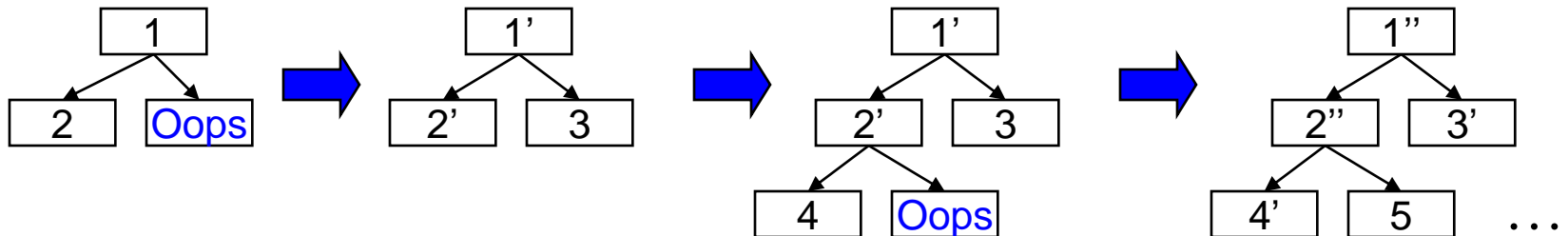
Top-Down Design is Good

- Top-down design 😊
 - Thiết kế toàn bộ sản phẩm một cách sơ bộ, tổng thể
 - Tinh chỉnh cho đến khi hoàn thiện
- Top-down design in **programming**
 - Xây dựng sơ lược hàm main() bằng pseudocode
 - Tinh chỉnh từng lệnh giả ngữ
 - Công việc đơn giản => thay bằng real code
 - Công việc phức tạp => thay bằng lời gọi hàm
 - Lặp lại sâu hơn, cụ thể, chi tiết hơn
 - Kết quả: Sản phẩm được modul hóa 1 cách tự nhiên



Top-Down Design in Reality

- Thiết kế CT Top-down trong thực tiễn :
 - Định nghĩa hàm main() = pseudocode
 - Tinh chỉnh từng lệnh pseudocode
 - Nếu gặp sự cố Oops! Xem lại thiết kế, và...
 - Quay lại để tinh chỉnh pseudocode đã có, và tiếp tục
 - Lặp lại (mostly) ở mức sâu hơn, cụ thể hơn, cho đến khi các hàm đc định nghĩa xong



Ví dụ: Text Formatting

- Mục tiêu :
 - Minh họa good program và programming style
 - Đặc biệt là modul hóa mức hàm và top-down design
 - Minh họa cách đi từ vấn đề đến viết code
 - Ôn lại và mô tả cách xây dựng CT C
- Text formatting
 - Đầu vào: ASCII text, với hàng loạt dấu cách và phân dòng
 - Đầu ra: Cùng nội dung, nhưng căn trái và căn phải
 - Dồn các từ tối đa có thể trên 1 dòng 50 ký tự
 - Thêm các dấu cách cần thiết giữa các từ để căn phải
 - Không cần căn phải dòng cuối cùng
 - Để đơn giản hóa, giả định rằng :
 - 1 từ kết thúc bằng dấu cách space, tab, newline, hoặc end-of-file
 - Không có từ nào quá 20 ký tự

Ví dụ về Input and Output

I
N
P
U
T

Tune every heart and every voice.
Bid every bank withdrawal.
Let's all with our accounts rejoice.
In funding Old Nassau.
In funding Old Nassau we spend more money every year.
Our banks shall give, while we shall live.
We're funding Old Nassau.

O
U
T
P
U
T

Tune every heart and every voice. Bid every bank
withdrawal. Let's all with our accounts rejoice.
In funding Old Nassau. In funding Old Nassau we
spend more money every year. Our banks shall give,
while we shall live. We're funding Old Nassau.

- # Thinking About the Problem
- Khái niệm “từ”
 - Chuỗi các ký tự không có khoảng trắng, tab xuống dòng, hoặc EOF
 - Tất cả các ký tự trong 1 từ phải được in trên cùng 1 dòng
 - Làm sao để đọc và in dc các từ
 - Đọc các ký tự từ stdin cho đến khi gặp space, tab, newline, or EOF
 - In các ký tự ra stdout tiếp theo bởi các dấu space(s) or newline
 - Nếu đầu vào lộn xộn thì thế nào ?
 - Cần loại bỏ các dấu spaces thừa, các dấu tabs, và newlines từ input
 - Làm sao có thể căn phải ?
 - Ta không biết được số dấu spaces cần thiết cho đến khi đọc hết các từ
 - Cần phải lưu lại các từ cho đến khi có thể in được trọn vẹn 1 dòng
 - Nhưng, Bao nhiêu space cần phải thêm vào giữa các từ?
 - Cần ít nhất 1 dấu space giữa các từ riêng biệt trên 1 dòng
 - Có thể thêm 1 vài dấu spaces để phủ kín 1 dòng

Writing the Program

- Key constructs
 - Từ - Word
 - Dòng - Line
- Các bước tiếp theo
 - Viết pseudocode cho hàm main()
 - Tinh chỉnh
- Lưu ý :

Chú thích hàm và một số dòng trống được bỏ qua vì những hạn chế không gian

Trình tự thiết kế là lý tưởng

Trong thực tế, nhiều backtracking sẽ xảy ra

The Top Level

- pseudocode hàm main()...

```
int main(void) {  
    <Xóa dòng>  
    for (;;) {  
        <Đọc 1 từ>  
        if (<Hết từ>) {  
            <In dòng không cần căn phải>  
            return 0;  
        }  
        if (<Từ không vừa dòng hiện tại>) {  
            <In dòng có căn lề phải>  
            <Xóa dòng>  
        }  
        <Thêm từ vào dòng>  
    }  
    return 0;  
}
```

Reading a Word

```
#include <stdio.h>
enum {MAX_WORD_LEN = 20};
int main(void) {
    char word[MAX_WORD_LEN + 1];
    int wordLen;
    < Xóa dòng >
    for (;;) {
        wordLen = ReadWord(word);
        if (<Hết từ>) {
            < In dòng không cần căn phải >
            return 0;
        }
        if (< Từ không vừa dòng hiện tại >) {
            < In dòng có căn lề phải >
            < Xóa dòng >
        }
        < Thêm từ vào dòng >
    }
    return 0;
}
```

```
int ReadWord(char *word) {
    <Bỏ qua whitespace>
    <Luu các ký tự cho đến MAX_WORD_LEN của từ>
    <Trả về độ dài từ>
}
```

- <Đọc 1 từ> nghĩa là gì? Việc này khá phức tạp nên cần tách thành 1 hàm riêng ...

Reading a Word (cont.)

```
int ReadWord(char *word) {  
    int ch, pos = 0;  
  
    /* Bỏ qua whitespace. */  
    ch = getchar();  
    while ((ch == ' ') || (ch == '\n') || (ch == '\t'))  
        ch = getchar();  
  
    /* Lưu các ký tự vào từ cho đến MAX_WORD_LEN. */  
    while ((ch != ' ') && (ch != '\n') && (ch != '\t') && (ch != EOF)) {  
        if (pos < MAX_WORD_LEN) {  
            word[pos] = (char)ch;  
            pos++;  
        }  
        ch = getchar();  
    }  
    word[pos] = '\0';  
  
    /* Trả về độ dài từ. */  
    return pos;  
}
```

Reading a Word (cont.)

```
int ReadWord(char *word) {
    int ch, pos = 0;

    /* Bỏ qua whitespace. */
    ch = getchar();
    while (IsWhitespace(ch))
        ch = getchar();

    /* Lưu các ký tự vào từ cho đến MAX_WORD_LEN. */
    while (!IsWhitespace(ch) && (ch != EOF)) {
        if (pos < MAX_WORD_LEN) {
            word[pos] = (char)ch;
            pos++;
        }
        ch = getchar();
    }
    word[pos] = '\0';

    /* trả về độ dài từ. */
    return pos;
}
```

- Hmmmm. ReadWord() chứa 1 vài đoạn code lặp lại => tách thành 1 hàm riêng :

IsWhitespace(ch)

```
int IsWhitespace(int ch) {
    return (ch == ' ') || (ch == '\n') || (ch == '\t');
}
```


Cutting a Word

```
#include <stdio.h>
#include <string.h>
enum {MAX_WORD_LEN = 20};
enum {MAX_LINE_LEN = 50};
int main(void) {
    char word[MAX_WORD_LEN + 1];
    int wordLen;
    char line[MAX_LINE_LEN + 1];
    int lineLen = 0;
    <Xóa dòng>
    for (;;) {
        wordLen = ReadWord(word);
        if (<Hết từ>) {
            <In dòng không căn lề>
            return 0;
        }
        if (<Từ không vừa dòng>) {
            < In dòng có căn lề >
            <Xóa dòng>
        }
        AddWord(word, line, &lineLen);
    }
    return 0;
}
```

Quay lại main().

<Thêm từ vào dòng>
có nghĩa là gì ?

- Tạo 1 hàm riêng cho việc đó :

**AddWord(word, line,
&lineLen)**

```
void AddWord(const char *word, char
*line, int *lineLen) {
    <Nếu dòng đã chứa 1 số từ, thêm 1 dấu
trắng>
    strcat(line, word);
    (*lineLen) += strlen(word);
}
```

Saving a Word (cont.)

- AddWord()

```
void AddWord(const char *word, char *line, int *lineLen) {  
  
    /* Nếu dòng đã chứa 1 số từ, thêm 1 dấu trắng. */  
    if (*lineLen > 0) {  
        line[*lineLen] = ' ';  
        line[*lineLen + 1] = '\0';  
        (*lineLen)++;  
    }  
  
    strcat(line, word);  
    (*lineLen) += strlen(word);  
}
```

Printing the Last Line

```
...
int main(void) {
    char word[MAX_WORD_LEN + 1];
    int wordLen;
    char line[MAX_LINE_LEN + 1];
    int lineLen = 0;
    <Xóa dòng>
    for (;;) {
        wordLen = ReadWord(word);

        /* Nếu hết từ, in dòng không căn lề. */
        if ((wordLen == 0) && (lineLen > 0)) {
            puts(line);
            return 0;
        }
        if (<Từ không vừa dòng>) {
            <In dòng có căn lề>
            <Xóa dòng>
        }
        AddWord(word, line, &lineLen);
    }
    return 0;
}
```

- <Hết từ> và <In dòng không căn lề> nghĩa là gì ?
- Tạo các hàm để thực hiện

Deciding When to Print

```
...
int main(void) {
    char word[MAX_WORD_LEN + 1];
    int wordLen;
    char line[MAX_LINE_LEN + 1];
    int lineLen = 0;
    <Xóa dòng>
    for (;;) {
        wordLen = ReadWord(word);
        /* If no more words, print line
           with no justification. */
        if ((wordLen == 0) && (lineLen > 0)) {
            puts(line);
            return 0;
        }
        /* Nếu từ không vừa dòng, thì ... */
        if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {
            <In dòng có căn lề>
            < Xóa dòng >
        }
        AddWord(word, line, &lineLen);
    }
    return 0;
}
```

- <Từ không vừa dòng>
Nghĩa là gì?

Printing with Justification

- Bây giờ , đến trong tâm của CT. <In dòng có căn lề> nghĩa là gì ?
- Rõ ràng hàm này cần biết trong dòng hiện tại có bao nhiêu từ. Vì vậy ta thêm **numWords** vào hàm main ...

```
...
int main(void) {
    ...
    int numWords = 0;
    <Xóa dòng>
    for (;;) {
        ...
        /* Nếu từ không vừa dòng, thì... */
        if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {
            WriteLine(line, lineLen, numWords);
            <Xóa dòng>
        }

        AddWord(word, line, &lineLen);
        numWords++;
    }
    return 0;
}
```

Printing with Justification (cont.)

- Viết pseudocode cho WriteLine()...

```
void WriteLine(const char *line, int lineLen, int numWords) {  
    <Tính số khoảng trống dư thừa cho dòng>  
  
    for (i = 0; i < lineLen; i++) {  
        if (<line[i] is not a space>)  
            <Print the character>  
        else {  
            <Tính số khoảng trống cần bù thêm>  
  
            <In 1 space, cộng thêm các spaces cần bù>  
  
            <Giảm thêm không gian và đếm số từ>  
        }  
    }  
}
```

Printing with Justification (cont.)

```
void WriteLine(const char *line, int lineLen, int numWords)
{
    int extraSpaces, spacesToInsert, i, j;
    /* Tính số khoảng trống dư thừa cho dòng. */
    extraSpaces = MAX_LINE_LEN - lineLen;
    for (i = 0; i < lineLen; i++) {
        if (line[i] != ' ')
            putchar(line[i]);
        else {
            /* Tính số khoảng trống cần thêm. */
            spacesToInsert = extraSpaces / (numWords - 1);

            /* In 1 space, cộng thêm các spaces phụ. */
            for (j = 1; j <= spacesToInsert + 1; j++)
                putchar(' ');

            /* Giảm bớt spaces và đếm từ. */
            extraSpaces -= spacesToInsert;
            numWords--;
        }
    }
    putchar('\n');
}
```

- Hoàn tất hàm WriteLine()...

Số lượng các khoảng trống

VD:
Nếu extraSpaces = 10
và numWords = 5,
thì space bù sẽ là
2, 2, 3, and 3 tương ứng

Clearing the Line

- Cuối cùng. <Xóa dòng> nghĩa là gì ?
- Tuy đơn giản, nhưng ta cũng xd thành 1 hàm

```
...
int main(void) {
    ...
    int numWords = 0;
    ClearLine(line, &lineLen, &numWords);
    for (;;) {
        ...
        /* If word doesn't fit on this line, then... */
        if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {
            WriteLine(line, lineLen, numWords);
            ClearLine(line, &lineLen, &numWords);
        }

        addWord(word, line, &lineLen);
        numWords++;
    }
    return 0;
}
```

```
void ClearLine(char *line, int
*lineLen, int *numWords) {
    line[0] = '\0';
    *lineLen = 0;
    *numWords = 0;
}
```


Modularity: Tóm tắt ví dụ

- **Với người sử dụng CT**

- n xộn
- Output: Cùng nội dung, nhưng trình bày căn lề trái, phải, rõ ràng, sáng sủa

- **Giữa các phần của CT**

- Các hàm xử lý từ : Word-handling functions
- Các hàm xử lý dòng : Line-handling functions
- main() function

- **Lợi ích của modularity**

- Đọc code: dễ dàng, qua các mẫu nhỏ, riêng biệt
- Testing : Test từng hàm riêng biệt
- Tăng tốc độ: Chỉ tập trung vào các ơhaanf chậm
- Mở rộng: Chỉ thay đổi các phần liên quan

The “iustify” Program

```
#include <stdio.h>
#include <string.h>
enum {MAX_WORD_LEN = 20};
enum {MAX_LINE_LEN = 50};
int IsWhitespace(int ch) {
    return (ch == ' ') || (ch == '\n') || (ch == '\t');
}
int ReadWord(char *word) {
    int ch, pos = 0;
    ch = getchar();
    while (IsWhitespace(ch))
        ch = getchar();
    while (!IsWhitespace(ch) && (ch != EOF)) {
        if (pos < MAX_WORD_LEN) {
            word[pos] = (char)ch;
            pos++;
        }
        ch = getchar();
    }
    word[pos] = '\0';
    return pos;
}
```

```
void ClearLine(char *line, int *lineLen, int
*numWords) {
    line[0] = '\0';
    *lineLen = 0;
    *numWords = 0;
}
void AddWord(const char *word, char *line, int
*lineLen) {
    if (*lineLen > 0) {
        line[*lineLen] = ' ';
        line[*lineLen + 1] = '\0';
        (*lineLen)++;
    }
    strcat(line, word);
    (*lineLen) += strlen(word);
}
```

```
void WriteLine(const char *line, int lineLen, int numWords) {
    int extraSpaces, spacesToInsert, i, j;
    extraSpaces = MAX_LINE_LEN - lineLen;
    for (i = 0; i < lineLen; i++) {
        if (line[i] != ' ')
            putchar(line[i]);
        else {
            spacesToInsert = extraSpaces / (numWords - 1);
            for (j = 1; j <= spacesToInsert + 1; j++)
                putchar(' ');
            extraSpaces -= spacesToInsert;
            numWords--;
        }
    }
    putchar('\n');
}
```

```

int main(void) {
char word[MAX_WORD_LEN + 1];
    int wordLen;
    char line[MAX_LINE_LEN + 1];
    int lineLen = 0;
    int numWords = 0;
    ClearLine(line, &lineLen, &numWords);
    for (;;) {
        wordLen = ReadWord(word);
        if ((wordLen == 0) && (lineLen > 0)) {
            puts(line);
            break;
        }
        if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {
            WriteLine(line, lineLen, numWords);
            ClearLine(line, &lineLen, &numWords);
        }
        AddWord(word, line, &lineLen);
        numWords++;
    }
    return 0;
}

```

Optimizing C and C++ Code

- **Đặt kích thước mảng = bội của 2**
Với mảng, khi tạo chỉ số, trình dịch thực hiện các phép nhân, vì vậy, hãy đặt kích thước mảng bằng bội số của 2 để phép nhân có thể được chuyển thành phép toán dịch chuyển nhanh chóng
- **Đặt các case labels trong phạm vi hẹp**
Nếu số case label trong câu lệnh switch nằm trong phạm vi hẹp, trình dịch sẽ biến đổi thành if – else - if lồng nhau, mà tạo thành 1 bảng các chỉ số, như vậy thao tác sẽ nhanh hơn
- **Đặt các trường hợp thường gặp trong lệnh switch lên đầu**
Khi số các trường hợp tuyển chọn là nhiều và tách biệt, trình dịch sẽ biến lệnh switch thành các nhóm if – else – if lồng nhau. Nếu bố trí các case thường gặp lên trên, việc thực hiện sẽ nhanh hơn
- **Tái tạo các switch lớn thành các switches lồng nhau**
Khi số cases nhiều, hãy chủ động chia chúng thành các switch lồng nhau, nhóm 1 gồm những cases thường gặp, và nhóm 2 gồm những cases ít gặp=> kết quả là các phép thử sẽ ít hơn, tốc độ nhanh hơn

Optimizing C and C++ Code (tt)

- **Minimize local variables**

Các biến cục bộ được cấp phát và khởi tạo khi hàm đc gọi, và giải phóng khi hàm kết thúc, vì vậy mất thời gian

- **Khai báo các biến cục bộ trong phạm vi nhỏ nhất**

- **Hạn chế số tham số của hàm**

- **Với các tham số và giá trị trả về > 4 bytes, hãy dùng tham chiếu**

- **Đừng định nghĩa giá trị trả về, nếu không sử dụng (void)**

- **Lưu ý vị trí của tham chiếu tới code và data**

Các bộ xử lý dữ liệu hoặc mã giữ được tham chiếu trong bộ nhớ cache để tham khảo về sau, nếu được nó từ bộ nhớ cache. Những tài liệu tham khảo bộ nhớ cache được nhanh hơn. Do đó nó nên mã và dữ liệu đang được sử dụng cùng nhau thực sự nên được đặt với nhau. Điều này với object trong C++ là đương nhiên. Với C? (Không dùng biến tổng thể, dùng biến cục bộ ...)

Optimizing C and C++ Code (tt)

- Nên dùng int thay vì char hay short (mất thời gian convert), nếu biết int không âm, hãy dùng unsigned int
- Hãy định nghĩa các hàm khởi tạo đơn giản
- Thay vì gán, hãy khởi tạo giá trị cho biến
- Hãy dùng danh sách khởi tạo trong hàm khởi tạo

```
Employee::Employee(String name, String designation) {      m_name =  
    name;  
    m_designation = designation;  
}
```

```
/* === Optimized Version === */
```

```
Employee::Employee(String name, String designation): m_name(name),  
    m_designation (designation) { }
```

- Đừng định nghĩa các hàm ảo tùy hứng : "just in case" virtual functions
- Các hàm gồm 1 đến 3 dòng lệnh nên định nghĩa inline

Một vài ví dụ tối ưu mã C, C++

```
typedef unsigned int uint;
uint div32u (uint a) {
    return a / 32;
}
int div32s (int a){
    return a / 32; // return a >>5;
}
```

```
switch ( queue ) {  
    case 0 : letter = 'W'; break;  
    case 1 : letter = 'S'; break;  
    case 2 : letter = 'U'; break;  
}
```

Hoặc có thể là :

```
if ( queue == 0 )  
    letter = 'W';  
else if ( queue == 1 )  
    letter = 'S';  
else letter = 'U';
```

```
static char *classes="WSU";  
letter = classes[queue];
```

(x >= min && x < max) có thể chuyển thành
(unsigned)(x - min) < (max - min)

```
int fact1_func (int n) {  
    int i, fact = 1;  
    for (i = 1; i <= n; i++) fact *= i;  
    return (fact);  
}
```

```
int fact2_func(int n) {  
    int i, fact = 1;  
    for (i = n; i != 0; i--) fact *= i;  
    return (fact);  
}
```

Fact2_func nhanh hơn, vì phép thử != đơn giản hơn <=

Tối ưu đoạn code sau :

```
found = FALSE;
for(i=0;i<10000;i++) {
    if( list[i] == -99 ) {
        found = TRUE;
    }
}
if( found ) printf("Yes, there is a -99. !\n");
```

← i while...

Floating_point

So sánh :

$x = x / 3.0;$

Và

$x = x * (1.0/3.0) ;$

?

(biểu thức hằng được thực hiện ngay khi dịch)

Hãy dùng float thay vì double

Tránh dùng sin, exp và log (chậm gấp 10 lần *)

Lưu ý : nếu x là float hay double thì : $3 * (x / 3) \neq x$.

Thậm chí thứ tự tính toán cũng quan trọng: $(a + b) + c \neq a + (b + c)$.

- Tránh dùng ++, -- trong biểu thức lặp , vd
while (n--) {...}
- Dùng $x * 0.5$ thay vì $x / 2.0$.
- $x+x+x$ thay vì $x*3$
- Mảng 1 chiều nhanh hơn mảng nhiều chiều
- Tránh dùng đệ quy