IT3040- KỸ THUẬT LẬP TRÌNH
LỚP KSTN K62, NĂM HỌC 2018-2019

Giảng viên: PGS. TS.Huỳnh Quyết Thắng
BM Công nghệ phần mềm
Viện CNTT-TT, ĐHBK HN
https://users.soict.hust.edu.vn/thanghq
Điện thoại: 0913536752

I. Code tuning

1. Hiệu năng của chương trình và Code Tuning
2. Các phương pháp Code Tuning

1.1. Hiệu năng

Sau khi áp dụng các kỹ thuật xây dựng CT PM:

- CT đã có tốc độ đủ nhanh
  - Không nhất thiết phải quan tâm đến viêc tối ưu hóa hiệu năng
  - Chỉ cần giữ cho CT đơn giản và dễ đọc
- Hầu hết các thành phần của 1 CT có tốc độ đủ nhanh
  - Thường chỉ một phần nhỏ làm cho CT chạy chậm
  - Tối ưu hóa riêng phần này nếu cần
- Các bước làm tăng hiệu năng thực hiện CT
  - Tính toán thời gian thực hiện của các phần khác nhau trong CT
  - Xác định các “hot spots” – đoạn mã lệnh đòi hỏi nhiều thời gian thực hiện
  - **Tối ưu hóa phần CT đòi hỏi nhiều thời gian thực hiện**
  - Lặp lại các bước nếu cần

Tối ưu hóa hiệu năng của CT ?

- 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 $O(n^2)$ 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
- 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 CT chạy chậm
  - Đòi hỏi LTV nắm vững kiến thức về phần cứng, trình dịch và quy trình thực hiện CT

→ Code tuning

1.2. Code tuning (tinh chỉnh mã nguồn) là gì ?

- 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 CTC, 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

1.3. Cải thiện hiệu năng thông qua cải thiện 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 CT
- 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 CT chạy chưa đúng
    - Việc sửa lỗi có thể làm giảm hiệu năng CT
    - Việc tinh chỉnh thường làm cho việc kiểm thử và gỡ rối trở nên phức tạp
  - Sau khi đã bàn giao CT
    - Duy trì và cải thiện hiệu năng
    - Theo dõi việc giảm hiệu năng của CT khi đưa vào sử dụng

1.4. Quan hệ giữa hiệu năng và tinh chỉnh mã nguồn

- Việc giảm thiểu số dòng lệnh viết bằng 1 NNLT bậc cao KHÔNG có nghĩa là :
  - Làm tăng tốc độ chạy CT
  - 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

Quan hệ giữa hiệu năng và tinh chỉnh mã nguồn

- 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.

Quan hệ giữa hiệu năng và tinh chỉnh mã nguồn

- 1 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

2. Các 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
- Lưu ý: Càng thay đổi nhiều thì càng không cải thiện được hiệu năng

2.1. 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 )
     ….
```

2.1. Tinh chỉnh các biểu thức logic

- Không kiểm tra khi đã biết kết quả rồi
- Ví dụ: tinh chỉnh như thế nào ???

```
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%

2.1. 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
```

2.1. Tinh chỉnh các biểu thức logic

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%**

```
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
```

2.1. 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%**

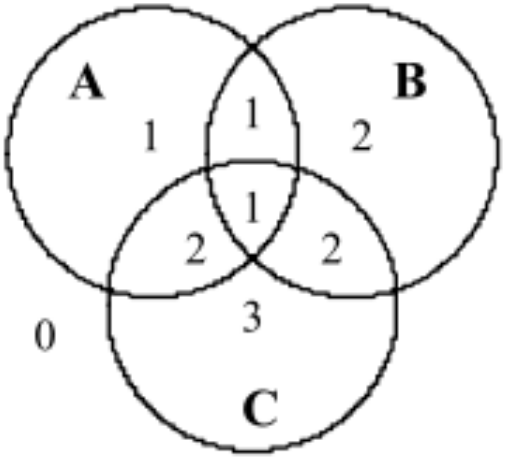
2.1. Tinh chỉnh các biểu thức logic

- 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

2.1. Tinh chỉnh các biểu thức logic

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

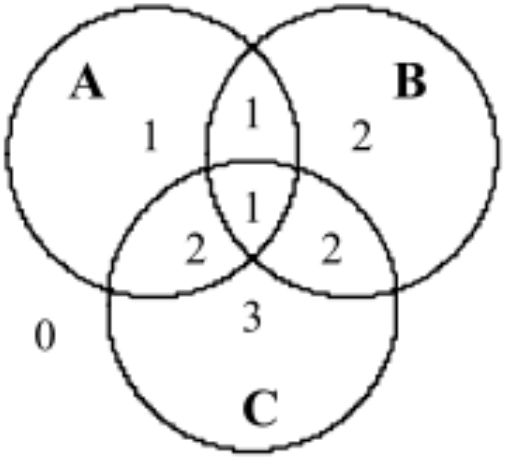


Initial code

```
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;
}
```

2.1. Tinh chỉnh các biểu thức logic

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



Tuned code

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

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

2.1. Tinh chỉnh các biểu thức logic

- Lazy evaluation: 1 trong các kỹ thuật viết mã chương trình hiệu quả đã học

2.2. 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 ];
    }
}
```

```
if ( sumType == SUMTYPE_NET ) {
    for ( i = 0; i < count; 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%

2.2. 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 ];
    }
}
```

2.2. 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

2.3. 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

2.4. Tinh chỉnh các biểu thức (đã học)

- 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 phép bitwise : <<, >>
- Sử dụng hằng số hợp lý
- Tính trước kết quả
- Sử dụng biến trung gian

2.5. Tinh chỉnh dãy lệnh (đã học)

- Sử dụng các hàm inline

2.6. Viết lại mã nguồn bằng ngôn ngữ 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 % CT 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 = assembler để tăng tốc độ thực hiện

Giúp trình dịch làm tốt công việc của nó

- 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

Kết luận

- 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

II. Xây dựng tài liệu

1. Các tài liệu trong và ngoài (internal và external)
2. Các quy tắc xây dựng tài liệu
3. Các quy định và kỹ thuật quy định Code Convention và Comment

1. Các loại tài liệu chương trình

- Tài liệu trong (Internal documentation)
  - Các chú thích cho mã nguồn (comments)
- Tài liệu ngoài (External documentation)
  - Dành cho các lập trình viên khác khi làm việc với mã nguồn
- Tài liệu dành cho người sử dụng
  - Cẩm nang dành cho những người sử dụng mã nguồn

1.1. Làm thế nào để viết được các chú thích hợp lý (good comment)

- Các chú thích có giúp người đọc hiểu được mã nguồn hay không ?
- Các chú thích có thực sự bổ ích hay không ? Lập trình viên viết chú thích vì chú thích là thực sự cần thiết để hiểu mã nguồn hay viết để cho có ?
- Người đọc có dễ dàng làm việc với mã nguồn hơn khi có chú thích hay không ?
- Nếu không chú thích được thì nên đặt tham chiếu đến một tài liệu cho phép người đọc hiểu vấn đề sâu hơn.

Các cách chú thích cần tránh: chú thích vô nghĩa

- Làm chủ ngôn ngữ
  - Hãy để chương trình tự diễn tả bản thân
  - Rồi…
- Viết chú thích để thêm thông tin

```
i= i+1; /* Add one to i */
```

```
for (i= 0; i < 1000; i++) { /* Tricky bit */
.
. Hundreds of lines of obscure uncommented code here
.
}
int x,y,q3,z4;  /* Define some variables */

int main()
/* Main routine */

while (i < 7) { /*This comment carries on and on */
```

Các chú thích cần tránh:
chú thích chỉ nhằm mục đích phân đoạn mã nguồn

```
while (j < ARRAYLEN) {
    printf ("J is %d\n", j);
    for (i= 0; i < MAXLEN; i++) {
/* These comments only */
        for (k= 0; k < KPOS; k++) {
/* Serve to break up */
            printf ("%d %d\n",i,k);
/* the program */
        }
/* And make the indentation */
    }
/* Very hard for the programmer to see */
    j++;
}
```

Viết bao nhiêu chú thích là đủ ?

- Chú thích là tốt, nhưng điều đó không có nghĩa là dòng lệnh nào cũng cần viết chú thích
- 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, ….
- Nhiều chú thích quá sẽ làm cho mã nguồn trở nên khó đọc
- Ít chú thích quá làm mã nguồn trở nên khó hiểu
- Chỉ viết chú thích nếu trong vòng 1 phút bạn không thể hiểu nổi đoạn lệnh đó làm gì, như thế nào
- Viết chú thích tương ứng với code!!!
  - Và thay đổi khi bản thân code thay đổi. ☺

Comments (cont.)

- Chú thích các đoạn (“paragraphs”) code, đừng chú thích từng 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;
}
```

Những thành phần nào của mã nguồn bắt buộc phải có chú thích

- Tất cả các file (nếu chương trình gồm nhiều file) đều cần chú thích về nội dung của file đó
- Tất cả các hàm: dùng để làm gì, dùng các biến đầu vào nào, trả ra cái gi.
- Biến có tên không rõ ràng
  - `i,j,k` cho vòng lặp, `FILE *fptr` không cần chú thích
  - nhưng `int total;` cần
- Tất cả các struct/typedef (trừ phi nó thực sự quá tầm thường)

File comments

```
/*************************************************************************
class:  GigaTron (GIGATRON.CPP)
author: Dwight K. Coder
date:   July 4, 2014
Routines to control the twenty-first century's code evaluation
tool. The entry point to these routines is the EvaluateCode()
routine at the bottom of this file.
*************************************************************************/
```

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***

Các quy tắc viết chú thích khác

- Chú thích nếu bạn cố tình thực hiện 1 thao tác kỳ cục khiến các LTV khác điên đầu
- Nếu chú thích quá dài, tốt nhất là nên đặt tham chiếu sang đoạn văn bản mô tả chi tiết ở chỗ khác
- Đừng cố gắng định dạng chú thích theo cách có thể gây nhầm lẫn với mã nguồn (ví dụ, đặt gióng hàng riêng cho chú thích)

1.2. Tài liệu ngoài cho các LTV khác

- Giới thiệu với các LTV khác mã nguồn dùng để làm gì
- Nhiều công ty lớn tự đặt chuẩn riêng để viết tài liệu ngoài
- Mục tiêu là cho phép các LTV khác sử dụng và thay đổi mã nguồn mà không cần đọc và hiểu từng dòng lệnh
- Đừng quên viết tài liệu ngoài cho bài tập lớn

Viết tài liệu ngoài: bước 1

- Miêu tả một cách tổng quát cách thức hoạt động của CT
  - CT phải làm gì ?
  - Phải đọc từ nguồn dữ liệu nào, ghi vào đâu?
  - Giả thiết gì với đầu vào ?
  - Dùng giải thuật nào ?

Viết tài liệu ngoài: bước 2

- Miêu tả 1 cách tổng quát quy trình nghiệp vụ của CT (giống như cách miêu tả 1 flowchart)
- Có thể vẽ biểu đồ
- Giải thích các giải thuật phức tạp được ưử dụng trong chương trình, hoặc cho biết có thể tìm được lời giải thích ở đâu

Viết tài liệu ngoài: bước 3

- Nếu CT bao gồm nhiều file, giải thích nội dung từng file
- Giải thích cấu trúc dữ liệu được sử dụng phổ biến trong CT
- Giải thích việc sử dụng các biến toàn cục trong các CTC

Viết tài liệu ngoài: bước 2

- Miêu tả các hàm chính trong CT
  - LTV tự quyết định hàm nào là hàm chính trong CT của mình
  - Xem xét hàm nào là hàm nghiệp vụ thực sự, ko nhất thiết phải là hàm dài nhất hay khó viết nhất
- Miêu tả các tham số đầu vào và giá trị trả về

1.3. Viết tài liệu cho người dùng

- Đây chính là hướng dẫn sử dụng (user manual)
- Là phần không thể thiếu khi viết tài liệu cho 1 dự án phần mềm, nhưng không phải phần quan trọng nhất

1.4. Viết tài liệu kiểm thử

- Tài liệu kiểm thử là 1 trong số các tài liệu quan trong của 1 dự án phần mềm
- Nếu được, bạn nên viết ra 1 số bằng chứng về việc bạn đã kiểm thử chương trình của bạn với nhiều đầu vào khác nhau
- Việc không viết tài liệu kiểm thử có thể gây ra nhiều hậu quả nặng nề